UNIVERSITATEA POLITEHNICA TIMIŞOARA
Facultatea de Automatică şi Calculatoare

TEZĂ DE DOCTORAT

Contribuţii la elaborarea unor structuri pentru securizarea informaţiei

Conducător ştiinţific
Prof. Dr. Ing. MIRCEA VLĂDUŢIU

Doctorand
Ing. GERDA ERICA MANG

2000

CUPRINS

CAPITOLUL 4. Configurarea mediului experimental în vederea Implementării hardware a criptochip-ului

CAPITOLUL 5. CRIPTOR – Circuit de criptare cu facilități de autotestare

CAPITOLUL 0

INTRODUCERE

Până la mijlocul deceniului nouă, termenul de fiabilitate a fost considerat ca fiind corespondentul în limba engleză pentru noțiunea de *realibility*. În urma activității prestate, grupul de lucru IFIP 10.4 condus de A. Avizienis şi I.C. Laprie [Aviz-82], [Lapr-85] lansează noțiunea de dependabilitate (*dependability*), stabilind şi dependențele clare relativ la triunghiul cădere-eroare-defect.

Termenul dependabilitate (dependability) acoperă conceptele de funcționare fără defecțiuni (reliability), disponibilate (availability), mentenabilitate (maintenability), securitate (security) şi testabilitate (testability), fiecare dintre acestea oferind indicatori numerici meniți a permite cuantificarea dependabilității unui sistem [Vasi-95]. Ca urmare, putem afirma că dependabilitatea reprezintă calitatea serviciului pe care îl asigură un sistem particular, adică fiabilitatea lui [Vlăd-89]. În figura 0.1 este reprezentată structura arborescentă a vastului domeniu al fiabilității [CăBa-85].

Dezvoltarea foarte puternică a tehnologiei electronice şi informatice şi - ca urmare – scăderea prețurilor sistemelor de calcul au determinat introducerea calculatoarelor în toate sectoarele activității economice şi sociale. Pe lângă aplicațiile industriale, calculatoarele sunt puternic utilizate în aplicații medicale, ştiințifice dar şi bancare, comerciale, militare şi diplomatice. Comunicațiile digitale, rețelele naționale şi internaționale de calculatoare fac posibile astăzi efectuarea de tranzacții economice şi bancare direct prin computerul de la birou sau chiar de acasă. Acest fapt impune desigur realizarea de sisteme de calcul foarte sigure în funcționare, pentru utilizarea lor ca servere de rețea, dar şi conceperea unor metode şi tehnici care să asigure siguranța, integritatea şi confidențialitatea datelor care se transmit între diverşi utilizatori conectați prin rețelele internaționale de calcul. Din aceste cauze, fiabilitatea sistemelor nu mai este un atribut doar al sistemelor militare sau speciale, ea devenind în ultimii ani un atribut important şi esențial al calculatoarelor indiferent de domeniul de aplicabilitate al lor [CăMi-83] [CăBa-89].

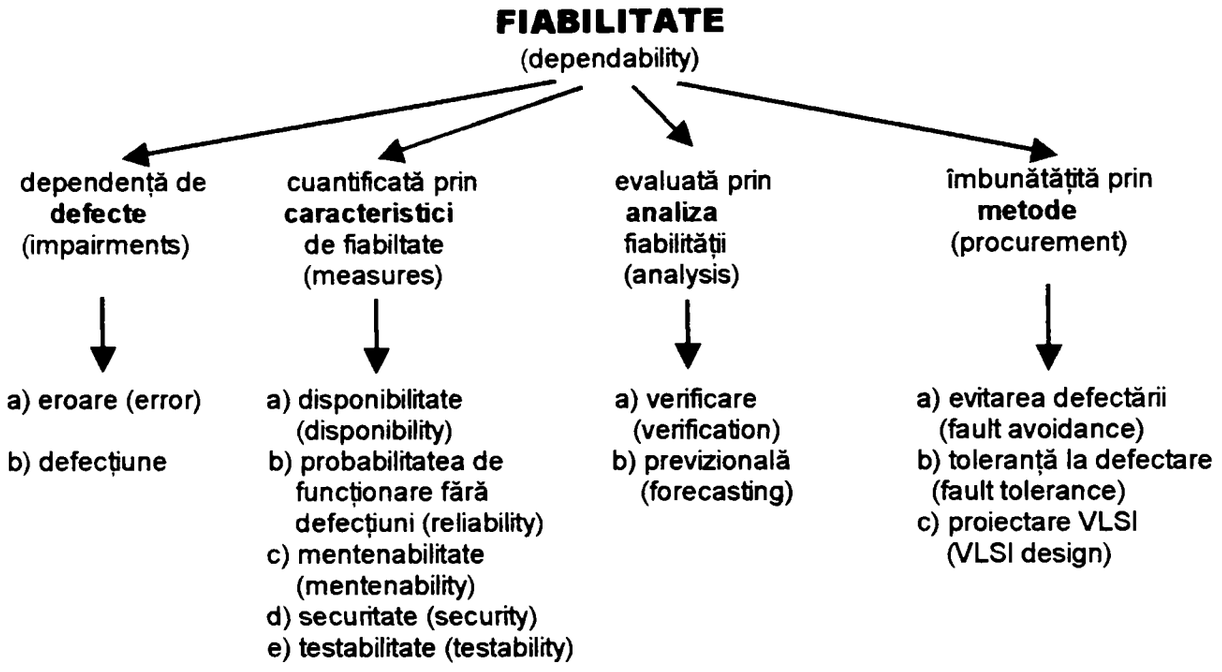


Figura 0.1. Structura arborescentă a fiabilității

În această teză, pornind de la importanța realizării unui sistem de securitate a datelor, mi-am propus să implementez hardware un algoritm de securitate. Am ales pentru aceasta algoritmul RC6, elaborat de firma RSA din SUA, unul dintre liderii mondiali ai domeniului. Desigur că alegerea s-a făcut în urma unei analize foarte puternice a algoritmilor candidați la noul standard de criptare, considerați ca cei mai noi şi mai puternici algoritmi de criptare.

Pentru a elimina erorile datorate defectelor circuitului am abordat problemele proiectării pentru testabilitate a circuitelor. În acest mod, erorile care pot să apară în timpul funcționării circuitului vor opri activitatea de criptare şi în acest mod se elimină riscul de a cripta eronat datele, lucru care poate conduce fie la imposibilitatea decriptării acestora de către persoanele autorizate fie la o uşoară decriptare a lor de către persoane neautorizate.

Întreaga activitate de proiectare şi implementare hardware s-a făcut utilizând tehnici de proiectare VLSI [MaPo-88] [MaMa-95b]. Am ales şi dezvoltat un mediu de simulare şi proiectare bazat pe circuitele FPGA ale firmei XILINX şi programele de proiectare VHDL corespunzătoare.

Ca o primă concluzie, pot spune că în această teză am abordat trei subdomenii importante ale ştiinței fiabilității, aşa cum se prezintă în figura 0.2.

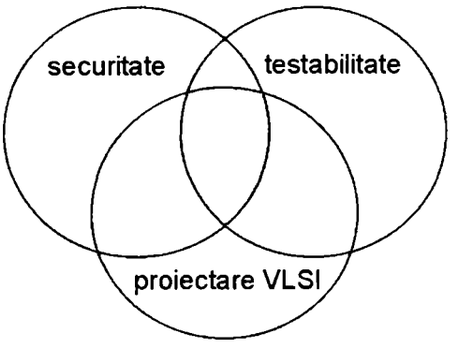


Figura 0.2. Domeniile abordate în teză

0.1. Oportunitatea tezei

Ca în multe alte domenii ştiinţifice, dezvoltarea reţelelor de comunicaţie şi în special a reţelei INTERNET s-a făcut înaintea dezvoltării mecanismelor de securitate arătate mai sus, ceea ce a permis apariţia în timp a unui număr mare de incidente, voluntare sau involuntare. Cu scopul de a puncta cât mai categoric necesitatea şi actualitatea problematicii securităţii datelor, voi prezenta în continuare câteva dintre incidentele petrecute în ultimii ani în diverse reţele de calcul din lume:

- În 1988, o mare agenţie de turism a descoperit un intrus care a pătruns în sistemul de rezervări şi vânzări de bilete şi a tipărit ilegal bilete de avion. Au fost făcute chiar speculaţii privind faptul că atacul terorist din 1988, în care membrii ai familiei regale din Kuwait au fost ţinuţi ostatici, ar fi avut legătură cu scurgerea de informaţii de acest fel.
- Tot în 1988, Cliff Stoll relatează în cartea sa "The Cuckoo's Egg" cum a urmărit timp de un an un "hacker" din Germania, care a încercat să pătrundă în 450 de calculatoare din întreaga lume. Intrusul a reuşit să pătrundă în 30 de sisteme, a avut acces la fişierele militare care conţineau informaţii despre arme nucleare, biologice şi chimice şi aproape a întrerupt un experiment medical.
- În 1989, un copil de 14 ani din Kansas, utilizând un calculator personal Apple a încercat să spargă un sistem de pozitionare a sateliţilor al Air Force. Copilul, care era un utilizator împătimit al calculatorului, a format neautorizat codurile de acces de la distanţă ale calculatoarelor din peste 200 de firme şi de asemenea a răscolit prin fişierele confidenţiale ale acestora. Când a fost prins, el a spus că el a sperat să influenţeze firmele să-l apeleze în probleme de consultanţă pentru securitate.
- În 1990, un student al unei facultăţi de calculatoare din Australia a fost învinuit că a decuplat pentru 24 de ore un sistem de calcul al NASA în Norfolk, Virginia.

- În 1991, înaintea votului final pentru noua constituție columbiană, un tehnician de calculatoare, care făcea actualizarea finală a textului online pe un calculator împrumutat, a făcut o eroare care a determinat ştergerea textului. Nu existau copii ale textului. Membrii comitetului constituțional au căutat printre variantele de proiect mai vechi şi notele referitoare la schimbările finale ale textului constituției.
- Cazurile de viruşi şi pagubele produse de aceştia au crescut în mod dramatic. Astfel, Computer Virus Industry Association a raportat că doar virusul Brain a afectat în 1990 peste 25000 de utilizatori de PC-uri.
- Furturile prin calculator au devenit un mare pericol pentru afaceri. În acest sens, Federal Bureau of Investigation arată că frauda prin calculator este cea mai costisitoare formă de fraudă comercială – cu o estimare a costului de 450.000$ pe fraudă. Acesta reprezintă pe departe cel mai mare risc pentru corporații, comparativ cu focul sau orice alt tip de pericol. Estimările privind suma totală în dolari pentru fraudele în calculatoare ajung la 5 miliarde de dolari pe an.

Exemplele de acest fel pot continua. A existat o publicitate substanțială în ultimii ani referitoare la riscurile şi atacurile asupra calculatoarelor dar, cu toate acestea, majoritatea pătrunderilor în sisteme au rămas neraportate. Unele rapoarte estimează că 90% din fraude şi pătrunderi în sisteme nu au fost raportate în afara organizațiilor în cauză. O serie de corporații şi agenții guvernamentale ezită să comunice astfel de evenimente deoarece riscă să se piardă încrederea clienților, datorită riscului de publicitate adversă. Spre exemplu, în Londra o serie de firme au înțelegeri semnate cu hoții prin calculator, oferindu-le amnestia în schimbul returnării unei părți a banilor furați şi a păstrării tăcerii în legătură cu furtul făcut. În unul dintre cazuri, un asistent programator al unui om de afaceri de la o bancă a direcționat 8 milioane de lire sterline într-un cont personal dintr-o bancă elvețiană. Într-un agreement prin care el era protejat, programatorul a promis să nu divulge pătrunderea în sistem în schimbul a un milion de lire sterline.

Toate acestea au făcut ca recunoaşterea publică a securității calculatoarelor să crească în mod spectaculos. O nouă generație de consultanță în securitate – numită de Bussines Week drept "hacker busters" - şi-a făcut simțită prezența şi astfel securitatea calculatoarelor a devenit o industrie de ordinul a miliarde de dolari. O serie de organizații stau acum pregătite pentru a furniza asistență în cazul în care ar apărea un eveniment asemănător. Astfel "Computer Emergency Response Team" (CERT) de la Software Engineering Institute de la Carnegie Mellon University, fondată de Defense Advanced Research Projects Agency (DARPA) serveşte drept "birou de informații" şi suport pentru orice probleme viitoare de internet. Centrul Carnegie Mellon are peste 100 de experți care pot fi apelați telefonic în toată țara. Departamentul de energie a înființat Computer Incident Advisor Capability (CIAC), orientat propriilor nevoi ale agenției. National Security Agency şi National Institute of Standards and Technology îşi concentrează resursele pentru a crea un centru care să răspundă cererilor guvernului privitor la crizele de securitate ale calculatoarelor. Unii distribuitori au pus bazele unor organizații care sunt pregătite să vină în ajutorul clienților lor care vor depista lipsuri de securitate sau atacuri directe. Un exemplu de

astfel de organizație este Customer Warning System al firmei Sun Microsistem. De asemenea, o serie de instituții academice şi de cercetare atât din Statele Unite cât şi din afară şi-au stabilit propriile organizații pentru cazuri de urgențe.

0.2. Securitatea datelor – domeniu de actualitate al sistemelor informatice

Toate cele expuse mai sus demonstrează că astăzi securitatea datelor este o problemă de o importanță deosebită şi de mare actualitate. Securitatea este un complex de măsuri procedurale, logice sau fizice destinate prevenirii, detectării şi corelării diferitelor tipuri de accidente, de utilizări defectuoase sau de căderi ale sistemelor (ECMA). Conform [RuGa-95] securitatea calculatoarelor înseamnă protecția calculatorului şi a tot ce este asociat cu acesta – clădirea, terminalele şi imprimantele, cablajele, discurile şi benzile. Mai mult, securitatea calculatoarelor presupune protecția informației memorate în sistem. Din acest motiv adeseori prin securitatea calculatoarelor se înțelege securitatea informației.

În concepția populară, securitatea calculatoarelor reprezintă doar secretizare. Secretizarea este un aspect foarte important dar nu este singurul. Pe lângă secretizare (numită uneori confidențialitate) mai trebuiesc luate în considerare şi acuratețea (uneori numită integritate) şi disponibilitatea. În unele aplicații un aspect al securității poate fi mai important decât celălalt [Patr-94]. În funcție de tipul de securitate cerut se va alege tipul de tehnică de securitate şi de produs necesar pentru a se îndeplini cerințele.

Un sistem sigur nu trebuie să permită accesul la informații decât acelora care au acest drept. În sistemele guvernamentale cu securitate ridicată, secretizarea este asigurată prin faptul că utilizatorii au acces doar la informațiile la care aceştia au dreptul în funcție de gradul lor. În afaceri, confidențialitatea asigură protecția informațiilor private. Secretizarea are o importanță primordială în cazul protejării informațiilor de apărare națională sau în cazul informațiilor de afaceri exclusive. În astfel de exemple integritatea şi disponibilitatea sunt de asemenea importante, dar nu sunt critice. Câteva dintre metodele cele mai importante de asigurare a secretizării sau confidențialității sunt: controlul celor care au acces la informații şi controlul a ceea ce poate face fiecare utilizator în sistem [Salo-93]. O metodă importantă pentru asigurarea secretizării este aceea de criptare a informației [Schn-94].

Un sistem de calcul trebuie să mențină integritatea continuă a informației memorate. Acuratețea sau integritatea este asigurată prin faptul că sistemul nu alterează sau corupe informația respectiv nu permite schimbarea neautorizată, malițioasă sau accidentală, a informației. În rețelele de comunicații, o variantă a integrității este cunoscută şi sub denumirea de autenticitate. Prin aceasta se înțelege că atât autorul cât şi receptorul unui mesaj sunt cei doriți [Schn-96]. Acuratețea este cel mai important element de securitate în sistemele financiare. Metodele de criptare asigură autentificarea mesajelor asigurând astfel acuratețea transmisiei.

Disponibilitatea se referă la faptul că sistemul informatic trebuie să fie perfect funcțional în întregul lui. Un sistem de calcul sigur trebuie să păstreze informația disponibilă pentru utilizatorii săi. Disponibilitatea este o necesitate de bază a securității. Imposibilitatea de a utiliza calculatorul nu ne permite nici să afirmăm că sunt îndeplinite deziderateIe de secreitizare şi acurateţe. Chiar şi utilizatorii care nu agrează securitatea sunt totuşi de acord cu faptul că sistemul trebuie să poată funcționa. Mulți nu realizează faptul că menținerea în funcționare a unui sistem reprezintă de fapt un aspect al securității.

Primele măsuri relatate, legate de securitatea calculatoarelor şi securitatea informației se pot asocia anilor 1950, când a fost elaborat primul standard TEMPEST de securitate, când au fost luate primele măsuri de securitate în proiectarea sistemelor de calcul şi a fost înființată prima organizație guvernamentală, COMSEC (Communications Security). Anii '60 pot fi considerați anii de început pentru securitatea calculatoarelor, odată cu inițiativele unor organizații din Statele Unite ale Americii, ca Department of Defence (DoD), National Security Agency, National Bureau of Standards alături de primele avertismente publice referitoare la securitate.

Interesul public în securitatea calculatoarelor a crescut în ultima perioadă a anilor '60. Spring Joint Computer Conference din 1967 este recunoscută ca fiind prima prezentare publică făcută pentru un auditoriu tehnic. Un interes sporit referitor la securitatea calculatoarelor şi a informației se manifestă în cadrul Departamentului Apărării care în 1970 publică un raport cu titlul "Security Controls for Computer Systems", raport considerat ca fiind de referință în istoria securității calculatoarelor. În anii '70 DoD şi alte agenții au sponsorizat o serie de proiecte de cercetare de bază care aveau drept scop identificarea cerințelor de securitate formularea modelelor de politе de asigurare a securității şi definirea liniIor directoare recomandate. Tot în această perioadă au fost publicate o serie de rapoarte care defineau sistemele sigure şi cerințele de securitate.

David Bell şi Leonard LaPadula au elaborat primul model matematic pentru asigurarea securității multi-nivel, model care a stat atât la baza standardelor de securitate a calculatoarelor cât şi la elaborarea modelelor ulterioare de securitate.

La sfârşitul anilor '70 două inițiative guvernamentale au influențat elaborarea standardelor şi a metodelor de securitate. DoD a reunit în cadrul unor seminarii membrii ai Ministerului Apărării şi specialişti din industrie pentru a găsi răspuns la întrebările: Cum poate fi evaluată securitatea unui sistem? Cum pot fi determinați producătorii de calculatoare să producă sisteme sigure? Cea de a doua inițiativă a venit din partea NIST (National Institute of Standards and Technology) care a devenit responsabilă pentru elaborarea de standarde în diverse domenii, incusiv cel al calculatoarelor. În 1981 a fost înființat în cadrul DoD, Computer Security Center (CSC) care era responsabil de securitatea informației. Câțiva ani mai târziu, aceste sarcini au fost preluate de toate agențiile federale, luând naştere National Computer Security Center (NCSC). A fost elaborată şi publicată lucrarea "Department of Defense Trusted Computer System Evalution Criteria" (TCSEC), cunoscută azi sub numele de cartea portocalie, şi care este biblia pentru elaborarea unor sisteme sigure. În această lucrare sunt prezentate criteriile de evaluare utilizate pentru a aprecia nivelul de încredere în care poate fi plasat un anumit sistem de calcul.

Există trei cuvinte cheie care apar atunci când vorbim despre securitatea calculatoarelor: vulnerabilitate, amenințări ale securității şi măsuri de apărare ce trebuiesc folosite. Vulnerabilitatea se referă la un punct în care sistemul este susceptibil de atac. O amenințare este un posibil pericol care poate fi fie o persoană (system cracker) fie un obiect (o piesă defectă) sau un eveniment (incendiu, inundație, cutremur, etc.) care pot influența funcționarea sigură a sistemului. Cu cât sunt observate mai multe vulnerabilități ale sistemului, cu cât se crede că există mai multe amenințări cu atât mai multă atanție se acordă modului de protejare a sistemului şi a informației. Tehnicile de protecție ale sistemului reprezintă măsurile de apărare.

Orice sistem este vulnerabil. Măsurile de securitate şi produsele existente vor reduce probabilitatea ca un atac să reuşească sau vor mări timpul necesar sau resursele necesare ale unui intrus pentru ca acesta să reuşească să pătrundă în sistem. Dar practic nu se poate afirma că un sistem este complet sigur. Atunci când analizăm vulnerabilitatea unui sistem de calcul, trebuie să avem în vedere toate aspectele vulnerabile, şi anume:

- *vulnerabilitatea fizică*, care se referă la posibilitatea intruşilor de a pătrunde în clădirile şi birourile unde se află sistemele de calcul. Există o serie de măsuri ce pot fi luate pentru controlul accesului în interiorul clădirilor sau a anumitor încăperi;
- *vulnerabilitatea naturală*, care se referă la faptul că sistemele de calcul sunt foarte vulnerabile la dezastre naturale sau de mediu cum ar fi: incendii, inundații, cutremure, trăznete şi căderi accidentale ale tensiunii electrice ce pot provoca distrugerea datelor;
- *vulnerabilitatea hardware şi software* se referă la erori şi defecte ce pot apare într-un sistem de calcul, erori care influențează corectitudinea datelor şi deci şi securitatea sistemului;
- *vulnerabilitatea mediilor de memorare*, care se referă la posibilitatea deteriorării discurilor sau a benzilor magnetice şi implicit ale datelor memorate pe acestea;
- *vulnerabilitatea emanațiilor*, care se referă la faptul că toate echipamentele electronice emit radiații electrice şi electromagnetice. Aceste radiații pot fi interceptate de semnalele transmise între sistemele de calcul şi astfel pot perturba transmisiile de date până la alterarea lor;
- *vulnerabilitatea comunicațiilor*, caracterizată prin faptul că un calculator legat la o rețea, direct sau printr-un telefon, este vulnerabil la intruşi care pot altera informațiile din sistem. De asemenea şi liniile de legătură sunt vulnerabile la distrugere;
- *vulnerabilitatea umană*, care se referă la faptul că securitatea unui sistem de calcul depinde în mare măsură de calificarea şi buna credință a persoanei care asigură administrarea sistemului. Vulnerabilitatea poate fi: fizică, naturală, hardware şi software, media, emanații, comunicații sau umană.

Tipurile de pericole (ameninţări) care pot afecta un sistem de calcul pot fi împărţite în trei categorii:

- *naturale*, prin posibilitatea apariţiei unor incendii, inundaţii, căderi ale tensiunii electrice, etc.;
- *neintenţionate*, datorate ignoranţei şi nepriceperii unor utilizatori;
- *intenţionate*, datorate existenţei unor indivizi sau instituţii care au drept scop pătrunderea în sistemele informatice ale altora, în general urmată de distrugerea datelor memorate.

Măsurile de apărare care trebuiesc luate se referă la totalitatea tehnicilor şi a metodelor ce trebuiesc folosite pentru protecţia calculatoarelor şi a informaţiilor. Ele vor avea în vedere toate punctele vulnerabile şi toate ameninţările posibile la un sistem informatic dat.

Dintre toate aceste tehnici şi metode de protecţie a datelor, secretizarea (confidenţialitatea) datelor constituie un domeniu foarte larg şi actual al securităţii datelor. Secretizarea datelor se asigură de obicei prin criptarea acestora [Stin-96]. Există un număr foarte mare de algoritmi de criptare, unii mai simpli iar alţii foarte complicaţi şi greu de descifrat. În ultimii doi ani, se încearcă standardizarea unor algoritmi noi, cu caracteristici foarte puternice şi aplicabili în toate domeniile prelucrării datelor (indiferent de tipul de procesor utilizat şi de lungimea textului secretizat). Unul dintre cei mai bine cotaţi dintre aceşti algoritmi candidaţi la standardizare este algoritmul RC6, algoritm prezentat pe larg în această teză.

0.3. Caracteristici generale ale proiectării pentru testabilitate

Testării circuitelor digitale îi revine o mare parte din efortul de proiectare, realizare şi utilizare. Metodele de testare şi eficienţa lor au evoluat o dată cu trecerea de la componentele discrete la circuitele integrate pe scară largă. Cum numărul circuitelor care pot fi integrate pe o singură bucată de siliciu este foarte mare, s-a considerat că o parte a acestor circuite ar trebui dedicate testării funcţionării corecte a întregului. Astfel s-a născut conceptul de autotestare sau testare integrată.

Când electronica digitală se baza doar pe componente discrete (tuburi, tranzistori, rezistente, condensatoare şi diode), testarea se realiza în trei etape. În prima etapă fiecare componentă era testată conform specificaţiilor tehnice, în a cea de a doua etapă componentele erau asamblate în elemente digitale: porţi, bistabile şi în final erau testate din punct de vedere funcţional. În cea de a treia etapă se realizează testarea funcţională a aplicaţieie la nivel de sistem, unde toate influenţele determinate de întârzieri, încărcare, temperatură şi zgomot influenţează funcţionarea. Deoarece calculatoarele şi sistemele digitale deveneau din ce în ce mai complexe, această ultimă etapă devenea aproape imposibil de realizat. Este imposibil să se determine un test potrivit şi de asemenea să se localizeze şi să se elimine componenta defectă prin simpla încercare de a opera cu sistemul. R. D. Eldred

(1959) a arătat că este mai uşor să se verifice hardware-ul unui sistem decât funcționarea sa. Acesta a fost începutul testării structurale, aşa cum o cunoaştem noi azi. Lucrarea acestuia a pus bazele unei noi discipline care se ocupă de testare şi de generarea testelor pentru rețele digitale.

Prima aplicație a acestor noi tehnici de generare a testelor a fost realizată pentru plăcile cu circuite imprimate. Acestea permiteau producătorului să elimine aproape toate defectele legate de asamblare din subansamblele existente înainte de instalarea în sistem. Reducând astfel numărul de defecte, încercarea de a demonstra funcționalitatea sistemului este mai uşoară, dar totuşi este o sarcină greu de rezolvat.

O dată cu apariția circuitelor integrate, problemele legate de testare nu au fost cu mult diferite față de cele ale plăcilor cu circuite imprimate cu excepția unei subtile diferențe. Dacă obiectivul testării plăcilor cu circuite era de a găsi componente defecte şi de a repara placa pentru a întruni specificațiile tehnologice, obiectivul testării circutelor integrate este de a determina dacă circuitul este funcțional. În cazul circuitelor integrate, repararea nu este posibilă, astfel dacă apare un defect la testare, acesta determină decizia de a elimina respectiva componentă. Pe măsură ce numărul de circuite dintr-un chip cu circuite integrate a crescut la peste 100 de bucăți (anii 1970), problemele de testare au început să se schimbe. Suprafața mică a circuitelor integrate permite doar un număr mic de conectări să fie testate pe lângă terminalele circuitului. Problema acelei perioade a fost deci să se testeze sute de circuite prin intermediul a zeci de terminale. Pe măsură ce tehnologia a evoluat, la începutul anilor 1980, rata de 1000 de circuite pentru fiecare pin a devenit caracteristica problemelor de testare.

La un nivel de zeci de mii de circuite într-un chip de siliciu au apărut multe aplicații pentru circuitele integrate la costuri scăzute. Pe măsură ce acestea şi-au găsit locul în viața de zi cu zi a fiecăruia, nevoia de a fi testate corespunzător a devenit stringentă. Inconvenienta sau pericolul evident care poate rezulta din cauza unui circuit integrat defect în cazul rulării unei aplicații nu poate fi ignorat de către producători. Această presiune pentru calitate s-a transformat într-o cerere pentru tehnici de testare mai bune care să poată fi aplicate fără creşterea apreciabilă a costurilor circuitelor utilizate în aceste aplicații ieftine.

Tehnologia de testare a evoluat pe trei direcții distincte dar totuşi dependente: testarea hardware-lui, testarea software-lui şi teoria testării. Astfel, pentru a se progresa pe aceste direcții a fost nevoie să se includă componente pentru autotestare.

Testarea hardware-lui, sau altfel spus testarea sistemelor, a evoluat de la o simplă colecție de alimentări şi echipamente de măsură interconectate prin magistrale şi relee la sisteme electronice mari care realizau conversii digital-analogic sau analog-digital la nivelul fiecărui pin. Existau de asemenea buffer-e de memorie la nivelul fiecărui pin astfel încât să se poată aplica modele de test cu o viteză apropiată de cea normală a sistemelor testate. Unele sisteme dispun de generatoare de modele de test algoritmice construite în hardware, iar altele care sunt proiectate pentru a testa componentele de memorie au facilități de analiză a rezultatelor în timp real. Deşi astfel de sisteme de testare sunt performante ele totuşi sunt foarte scumpe. Aceste sisteme de testare sunt la rândul lor reallizate din aceleaşi circuite integrate care sunt

proiectate pentru a fi testate, cu toate că pe parcursul proiectării, realizării, achiziționării şi instalării de către utilizator, vârful tehnologiei a evoluat, apărând astfel necesitatea unui sistem de testare nou, actualul nemaifiind corespunzător cerințelor. Totuşi, având în vedere prețul inițial mare al acestor echipamente (peste un milion de dolari), majoritatea firmelor trebuie să păstreze aceste echipamente în uz pentru mai mulți ani pentru a-şi acoperi cheltuielile [Tsui-87]. Aceasta este una din problemele care determină dezvoltarea circuitelor digitale autotestabile. Dacă o parte din teste sau toate sunt realizate chiar de către circuit, dependența de echipamentele de testare se reduce în mare măsură.

Testarea software-lui cuprinde doua grupe de aplicații: pe de o parte aplicații pentru software-ul care rulează sistemul testat şi cealaltă grupă se referă la porțiunile automate din procesul de proiectare. Complexitatea cerințelor a cescut foarte mult şi deci generarea testelor a devenit un proces limitat.

Generarea automată a testelor pentru rețelele combinaționale, simularea defectelor şi verificarea regulilor de proiectare fac parte din programul care realizează testarea. Aceste aplicații au fost dezvoltate la un înalt nivel de complexitate, dar pe măsură ce circuitele testate devin din ce în ce mai mari, complexitatea de calcul a unor sarcini, cum ar fi generarea testului, încep să limiteze progresul.

Teoria testării s-a dezvoltat până la un punct în care nu pot fi spuse multe despre testabilitatea unui circuit. În timp ce teoria poate fi îmbunătățită pentru a descrie unele metode ad-hoc care sunt apoi etichetate cu "design for testability", dovada efectivă a acestor tehnici care arată că teoria este într-o formă destul de bună.

În primele etape de producție, testarea se realizează pentru a asigura o anumită structură sau o anumită funcție a componentei, subansamblului sau sistemului. Atunci când sistemul se află în modul de lucru normal spunem că el se află în mod "on-line". Adeseori este necesară testarea sistemului când funcționează în condiții "on-line". Această testare se realizează pentru a verifica modul cum se realizează o anumită sarcină mai deosebită, sau pentru a măsura toleranța la defecte a sistemului. Dacă sistemul trebuie oprit şi/sau dedicat doar procedurii de test la un moment dat, astfel încât funcționarea normală a sistemului să fie complet oprită, procedura de testare se spune că este "off-line". Cu alte cuvinte sistemul trebuie scos "off-line" (din funcționare) pentru a fi testat.

Evident că operațiile de testare din primele etape de producție sunt de tip "off-line". De fapt, în domeniul testării digitale de până acum o atenție deosebită a fost acordată testării "off-line".

Ținând cont de creşterea densității circuitelor integrate, rezultând într-o creştere a timpului necesar testării, metodele convenționale de testare prin aplicarea de stimuli externi pentru testarea structurală sau funcțională devin din ce în ce mai puțin satisfăcătoare. O alternativă a testelor aplicate extern este testarea built-in. Aceasta poate deveni parte a procesului de proiectare. Întrucât structurile VLSI fac acum parte din aplicațile comerciale şi având în vedere densitatea mare de integrare, cerințele de calitate au determinat ca proiectarea pentru testabilitate să devină o parte importantă a procesului de proiectare.

Diversele tehnici de proiectare pentru testabilitate (Design for Testability – DFT) sunt utilizate în special când este cerută o testare externă. BIST (Built-in Self-test) este o tehnică de proiectare în care părți ale circuitului sunt folosite pentru autotestarea acestuia. BIST este o combinație a conceptelor de built-in test (BIT) şi self-test şi a devenit un sinonim al acestor termeni. Adausul unor circuite şi a unui număr mic de pini de test pot facilita testarea circuitului integrat de către utilizator chiar în timpul funcționării acestuia. Cu ajutorul tehnicilor BIST, prezentate de altfel şi în capitolul 1 al tezei, testarea circuitelor FPGA va fi mai uşoară, mai rapidă şi mai simplă.

0.4. Structura tezei de doctorat

Având în vedere faptul că tema abordată de mine în această lucrare se situează la intersecția a trei discipline de mare actualitate am structurat teza pe şapte capitole importante. Am abordat în aceste capitole problemele specifice testării, securității informației şi proiectării logice care au stat la baza realizării primei implementări hardware a algoritmului de criptare RC6. Atrasă fiind de domeniul criptografiei am studiat pe parcursul anilor mai mulți algoritmi de criptare şi diversele implementări ale acestora, implementări care au fost realizate pe plan mondial. Pe de altă parte, în urma unor contacte avute prin intermediul programelor Tempus şi Socrates la Universitatea Tehnică Națională din Atena – Grecia şi Universitatea Catolică din Leuven – Belgia, am avut posibilitatea de a studia noile circuite FPGA precum şi noile metode de proiectare logică, bazate pe programe VHDL şi Verilog, oferite de firma Xilinx. Chiar mai mult, am intrat în contact cu acestă firmă şi prin intermediul organizației Europractice, al cărei reprezentant sunt pentru Universitatea din Oradea, am obținut o importantă donație pentru laboratorul de Analiza şi Sinteza Circuitelor Logice. Aceasta constă din programul de proiectare Foundation Express versiunea 2.1 şi suportul fizic necesar pentru implementarea proiectelor realizate.

Capitolul 1 al prezentei teze reprezintă o analiză a metodelor de creştere a testabilității şi abordează probleme ale redundanței hardware, ale redundanței informaționale, diverse tehnici de compresie de la cele mai simple şi până la cele complexe pentru ca în final să ajung la prezentarea unor arhitecturi BIST implementabile în criptochip-uri.

În capitolul 2 am făcut o analiză a celor 15 algoritmi candidați pentru noul standard de criptare care va fi adoptat în anul 2002. Această analiză reprezintă rezultatul unui studiu profund pe care l-am făcut pe parcursul ultimilor ani şi în urma căruia am ales algoritmul care se preta cel mai bine implementării hardware, acest algoritm fiind RC6. În continuare tot în acest capitol am prezentat pe larg acest algoritm, performanțele acestuia şi problemele legate de securitatea lui.

Cele două capitole descrise mai sus pregătesc trecerea la capitolul 3 al tezei, intitulat „Facilități de autotestare ale cifrului RC6 în vederea implementării hardware". În prima parte a acestui capitol am făcut o analiză privind oportunitatea aplicării tehnicilor BIST cifrului RC6, am elaborat diagrama fluxului de date al cifrului şi

diagrama fluxului de date a arhitecturii. În continuare am făcut un studiu privind propagarea modelelor aleatoare pentru cifrul RC6 şi facilităţile de testare concurentă ale acestuia. Se poate spune că în acest capitol au fost prezentate caracteristicile cele mai importante ale cifrului RC6 în scopul implementării acestuia într-un circuit cu facilităţi de autotestare.

Capitolul 4 este o descriere amănunţită cu ajutorul a numeroase scheme şi diagrame a mediului experimental care a servit la implementarea algoritmului RC6. Sunt descrise atât circuitele hard utilizate pentru implementare, şi aici trebuie să menţionez că în această teză sunt descrise pentru prima dată la noi în ţară circuitele din familia Virtex, cele mai noi circuite realizate de firma Xess, cât şi programul Foundation Express al firmei Xilinx.

În capitolul 5 am prezentat modul de implementare a algoritmului ales. Implementarea a fost realizată în două versiuni, una utilizând limbajul de nivel înalt VHDL şi cea de a doua utilizând limbajul Verilog. Toate rezultatele obţinute precum şi verificarea corectitudinii funcţionării circuitului care implementează algoritmul RC6 sunt prezentate în cele 12 anexe ale tezei.

Ultimul capitol, cel de concluzii, sintetizează contribuţiile originale prezentate în teză, contribuţii care sunt prezentate sub forma unui paragraf distinct, la sfârşitul fiecărui capitol.

Menţionez că pe parcursul lucrării, în cazurile în care am considerat că terminologia în limba română nu este încă consacrată, am folosit şi unele terminologii preluate din limba engleză, mai ales că unele dintre acestea fac parte deja din vocabularul de specialitate al limbii române.

În final, dar nu în ultimul rând, aduc profunde mulţumiri conducătorului ştiinţific Prof. Univ. Dr. Ing. Mircea Vlăduţiu pentru îndrumarea permanentă, competentă şi de un înalt nivel ştiinţific pe care am primit-o pe parcursul anilor, atât în perioada de pregătire ca doctorand cât şi în elaborarea acestei teze. Mulţumesc de asemenea domnului profesor şi pentru cunoştinţele şi experienţa ştiinţifică şi didactică pe care mi le-a împărtăşit pe parcursul tuturor anilor, încă din perioada de studenţie.

De asemenea, doresc să aduc mulţumiri tuturor profesorilor şi conducătorilor instituţiilor unde mi-am desfăşurat activitatea şi care au contribuit la formarea mea ca inginer şi cadru didactic.

Doresc să-i menţionez aici şi să aduc mulţumiri profesorului dr. ing. Bart Prenel şi profesorului dr. ing. Jean Peperstraete de la Universitatea Catolică din Leuven precum şi profesorului Kiamal Pekmestzi de la Universitatea Tehnică din Atena, care m-au ajutat în primii paşi pe care i-am făcut în proiectarea logică cu circuitele FPGA şi în demersurile pentru obţinerea acestor circuite.

CAPITOLUL 1

ANALIZA METODELOR DE CREŞTERE A TESTABILITĂŢII ÎN VEDEREA SELECTĂRII LOR PENTRU IMPLEMENTAREA ÎN CRIPTOCHIP-URI

1.1. Redundanţă hardware

Având în vedere dimensiunile mici şi costul redus al componentelor cea mai comună şi practică formă de obţinere a redundanţei este reproducerea de hardware. Există trei forme de bază de redundanţă [John-84]: pasivă, activă şi hibridă. Tehnicile pasive utilizează conceptul de mascare a defectului în scopul ascunderii defectului şi a prevenirii transformării defectului în eroare. Metodele pasive sunt concepute în aşa manieră încât să se evite intervenţia unui operator sau modificarea sistemului pentru atingerea toleranţei la defecţiuni. Aceste metode se bazează pe mecanismele de votare, marea majoritate fiind construite pe baza conceptului de vot majoritar.

Redundanţa activă se bazează pe detecţia existenţei defectului şi apoi înlăturarea părţii defecte din sistem. Acest lucru înseamnă că sistemul este reconfigurat pentru a se obţine toleranţa la defecţiuni. Redundanţa hardware activă face apel la tehnicile de detecţie, de localizare şi corectare a defectului. Există mai multe metode de redundanţă activă dintre care amintesc:

- Duplicarea cu comparaţie. La baza acestui concept stă dublarea pur şi simplu a hard-ului, execuţia în paralel a calculelor şi compararea apoi a rezultatelor obţinute. Prin această metodă se detectează doar existenţa defectului, nu se realizează toleranţa la acel defect.
- Standby sparing. În acest caz un modul este operaţional iar unul sau mai multe module servesc ca elemente de rezervă. Dacă prin diverse metode este detectată o eroare sau un modul defect, acesta este înlocuit de modulul rezervă. Metoda se pretează foarte bine în cazul sistemelor care au la bază module identice
- Pair-and-a-Spare. Această metodă este o combinaţie a celor două prezentate anterior. Pe parcursul întregului proces lucrează în paralel două

module a căror rezultate sunt comparate, asigurându-se în acest fel detecţia erorii. Semnalul de eroare furnizat de către comparator este utilizat pentru iniţierea procesului de reconfigurare şi înlocuirea elementelor defecte.

- Watchdog Timers – o metodă extrem de utilă pentru detectarea defectelor. La baza conceptului stă ideea că lipsa unei acţiuni reprezintă un defect. Watchdog Timer este un timer care trebuie resetat pe bază repetitivă. Se presupune că un sistem capabil să execute repetitiv o funcţie, ca de exemplu cea de setare a timer-ului, este fără defecţiuni. Imposibilitatea execuţiei funcţiei de reset va determina resetarea întregului sistem sau oprirea acestuia pentru prevenirea căderii sistemului.

Tehnicile hibride combină avantajele celor două metode: mascarea defectelor este utilizată în sistemele hibride pentru evitarea rezultatelor eronate, iar detecţia şi localizarea defectelor pentru mărirea toleranţei la defecţiuni prin înlăturarea şi înlocuirea componentelor defecte. Elementele suplimentare din sistem sunt utilizate doar în momentul în care alte elemente trebuiesc înlocuite. Metodele hibride sunt cele mai utilizate în aplicaţiile de calcul critic unde se reclamă mascarea defectului pentru evitarea erorilor momentane şi unde trebuie atins un grad înalt de fiabilitate [MaMa-95c]. Redundanţa hardware hibridă este o formă de redundanţă cu cost ridicat de implementare.

1.2. Redundanţă informaţională

Redundanţa informaţională înseamnă adaosul de informaţie redundantă la dată în scopul detecţiei de defecţiuni, mascării defecţiunii sau obţinerii toleranţei la defecţiuni. Codurile detectoare de erori şi codurile corectoare de erori, formate prin adaosul de informaţie cuvintelor de dată sau prin maparea acestora în reprezentări care conţin informaţie redundantă, reprezintă exemple tipice de redundanţă informaţională.

O caracteristică fundamentală a codurilor detectoare, respectiv corectoare de erori este aşa numita distanţă Hamming care reprezintă numărul de poziţii de biţi prin care diferă două cuvinte binare. Minimul distanţei Hamming dintre oricare două cuvinte valide de cod se numeşte distanţa codului. Dacă un cod are, de exemplu, distanţa doi atunci orice eroare singulară de bit introdusă într-un cuvânt al codului va determina invaliditatea acelui cuvânt deoarece toate cuvinte de cod valide diferă prin cel puţin două poziţii de bit. Deci, un cuvânt de cod corect se poate distinge faţă de cuvântul de cod eronat. În general un cod poate corecta până la c biţi eronaţi şi poate detecta erori de până la d biţi adiţionali dacă şi numai dacă:

$$2c + d + 1 \leq H_d$$

unde H_d este distanţa Hamming a codului.

Un alt concept fundamental în clasificarea codurilor este caracterul de separabilitate. Un cod se spune că este separabil dacă datei originale i se adaugă o informație suplimentară în scopul obținerii cuvântului de cod. În acest caz procesul de decodificare constă în simpla înlăturare a informației nedorite, păstrând doar data originală. Codurile neseparabile nu au această proprietate de separabilitate, motiv pentru care şi procedurile de decodificare sunt mult mai complicate.

În următoarele paragrafe ale acestui capitol voi analiza câteva dintre codurile binecunoscute în literatură şi voi prezenta imediat şi modalități de implementare ale acestora în scopul detecției de erori şi a realizării de sumatoare şi multiplicatoare self-checking.

1.2.1. Detecția erorilor şi corecția erorii singulare la adunarea binară şi la operațiile logice prin coduri bazate pe paritate

Codurile de paritate sunt cele mai simple coduri. Codurile de paritate cu bit singular necesită adaosul unui bit astfel încât noul cuvânt de cod să aibă fie un număr par fie un număr impar de 1-uri. Codurile de paritate cu bit singular au distanța Hamming egală cu 2, permițând detecția erorilor bit singular.

În general codurile bazate pe paritate nu acoperă controlul operațiilor logice, cu excepția celor de SAU EXCLUSIV şi SAU-NU EXCLUSIV. În [RaFu-89] este prezentată o tehnică care se bazează pe ideea că rezultatul unei operații arbitrare Φ poate fi transformat linear într-unul al unei operații de SAU EXCLUSIV. Aceasta va permite predicția de paritate pentru operația Φ.

Sunt considerate două cuvinte de date, astfel:

$$A = (a_{k-1}, a_{k-2}, \ldots, a_i, \ldots, a_1, a_0) \text{ şi } B = (b_{k-1}, b_{k-2}, \ldots, b_i, \ldots, b_1, b_0).$$

Biții de paritate corespunzători sunt generați prin relațiile:

$$p_y = \sum_{i=0}^{k-1}{}^{\oplus} y_i$$

unde $\sum^{\oplus}$ semnifică suma modulo 2. După cum rezultă din figura 1.1 modulul pentru unitatea aritmetică şi logică (ALU) prezintă două intrări $[A, p_A]$ şi $[B, p_B]$ şi o singură ieşire $[Y, p_y]$, definită prin vectorul $Y = (y_{k-1}, y_{k-2}, \ldots, y_i, \ldots, y_1, y_0)$ şi bitul de paritate.

Pentru operația SAU EXCLUSIV, controlul de paritate respectiv generarea sindromului S, are loc astfel:

$$S - \left(\sum_{i=0} y_i \right) \oplus p_A \oplus p_B, \text{ cu } y_i - a_i \oplus b_i \text{ pentru } 1 - 0,1,\ldots,k-1$$

şi cu concluzia că funcţionarea este eronată când $S = 1$, respectiv că e normală, fără defecte când $S = 0$. Funcţia de transformare $f_\Phi(y_i)$, exprimă transformarea a celui de al i-lea rezultat al operaţiei ALU în al i-lea rezultat al operaţiei SAU EXCLUSIV (respectiv $y_i = a_i \oplus b_i$, după cum rezultă din figura 1.2).

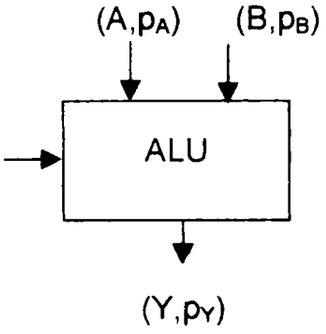


Figura 1.1. Modul ALU

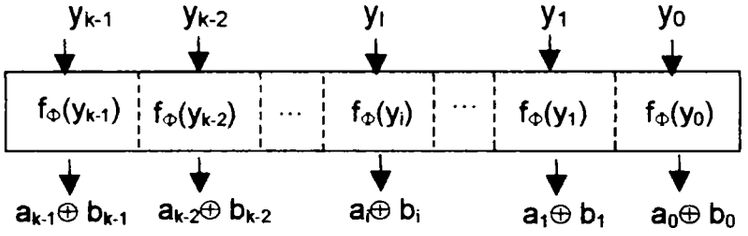


Figura 1.2. Funcţia de transformare $f_\Phi(y_i)$

Ieşirea funcţiei $f_\Phi(y_i)$, este dată de: $f_\Phi(y_i) = a_i \oplus b_i$ pentru $i = 0,1,...,k-1$. Deoarece funcţia $f_\Phi(y_i)$ satisface relaţiile menţionate în teorema ce urmează atunci o eroare la y_i se propagă la ieşirea $f_\Phi(y_i)$ şi deci paritatea pentru ieşirea ALU poate fi anticipată.

Definiţia 1.1. Diferenţa booleană a unei funcţii $F(x_0,x_1,....,x_i,....,x_{n-1})$ în raport cu variabila de intrare x_i este dată de relaţia:

$$\frac{dF}{dx_i} = F(x_0, x_1, \ldots, x_i, \ldots, x_{n-1}) \oplus F(x_0, x_1, \ldots, \overline{x_i}, \ldots, x_{n-1}) \qquad (1.1)$$

Teorema 1.1: Dacă raportat la funcţia de transformare $f_\Phi(y_i)$ este satisfăcută condiţia:

$$\frac{d(f_\Phi(y_i))}{dy_i} = 1, \text{ pentru } i = 0,1,\ldots,k-1 \qquad (1.2)$$

$$f_{\Phi}(y_i) = a_i \oplus b_i \tag{1.3}$$

atunci paritatea p_y pentru ieşirea unei ALU poate fi anticipată după cum urmează:

a) pentru $r_i = y_i \oplus a_i \oplus b_i$, avem $p_y = p_A \oplus p_B\left(\sum_{i=0}^{k-1}{}^{\oplus} r_i\right)$ (1.4)

b) pentru $r_i = y_i \oplus a_i \oplus b_i$, avem

$$p'y = \begin{cases} p_A \oplus p_B \oplus \left(\sum_{i=0}^{k-1}{}^{\oplus} r_i\right), \text{ cand k este par} \\ \overline{p_A \oplus p_B} \oplus \left(\sum_{i=0}^{k-1}{}^{\oplus} r_i\right), \text{ cand k este impar} \end{cases} \tag{1.5}$$

Demonstrație: Următoarea relație satisface ecuația (1.2).

$$f_{\Phi}(y_i) = y_i \oplus r_i \tag{1.6}$$

r_j poate fi întotdeauna obținut prin:

$$r_j = y_i \oplus r_i \oplus y_l \tag{1.7}$$

Pe de altă parte, efectuând adunarea modulo 2 în raport cu *i* în ambii membri ai ecuațiilor (1.3) şi (1.7) obținem:

$$\sum_{i=0}^{k-1}{}^{\oplus} f_{\Phi}(y_i) = \left(\sum_{i=0}^{k-1}{}^{\oplus} a_i \oplus b_i\right) = p_{Ai} \oplus p_B$$

$$\sum_{i=0}^{k-1}{}^{\oplus} f_{\Phi}(y_i) = \left(\sum_{i=0}^{k-1}{}^{\oplus} y_i \oplus r_i\right) = \left(\sum_{i=0}^{k-1}{}^{\oplus} y_i\right) \oplus \left(\sum_{i=0}^{k-1}{}^{\oplus} r_i\right)$$

Din aceste ecuații, luând în considerare proprietățile intrinseci ale operatorului EX - OR, se obține pentru paritatea anticipată p_y:

$$p_y^{'} = \sum_{i=0}^{k-1}{}^{\oplus} y_i = p_{Ai} \oplus p_B\left(\sum_{i=0}^{k-1}{}^{\oplus} r_i\right), \tag{1.8}$$

care satisface ecuația (1.4)

Pe de altă parte:

$$f_{\Phi}(y_i) = \overline{y_i \oplus r_i}$$

Procedând în manieră similară, pentru $r_i = y_i \oplus \overline{a_i \oplus b_i}$ rezultă paritatea anticipată p'_y dată de ecuația (1.5).

Din definiția ecuației booleene, ecuația (1.2) arată în mod clar că o eroare singulară *y* este propagată la ieşirea funcției $f_\Phi(y_i)$. Ecuațiile (1.2) şi (1.3) sunt importante nu numai pentru faptul că permit evaluarea biților de paritate anticipată, ci şi pentru că stabilesc condiția de propagare erorii din y_i la ieşirea funcției $f_\Phi(y_i)$. Admițând că R este dat de vectorul R = $(r_{k-1}, r_{k-2}, \ldots, r_i, \ldots, r_1, r_0)$, în tabelul din figura 1.8 sunt date funcțiile r_i pentru operațiile aritmetice şi logice fundamentale. În acest tabel s-a notat prin *c*-1 bitul de transport care intră în poziția *i*.

Teorema 1.1 permite efectuarea controlului bazat pe paritate pentru o operație ALU oarecare prin utilizarea ecuației (1.4) de evaluarea a parității anticipate p_Y după cum se vede în Tabelul 1.1.

Tabelul 1.1.

Operația Φ	y_i	r_i	
		*=EX-OR	*=EX-NOR
AND	$a_i \wedge b_i$	$a_i \vee b_i$	$\bar{a}_i \wedge \bar{b}_i$
OR	$a_i \vee b_i$	$a_i \wedge b_i$	$\bar{a}_i \vee \bar{b}_i$
EX-OR	$a_i \oplus b_i$	0	1
EX-NOR	$\overline{a_i \oplus b_i}$	1	0
ADD	$a_i + b_i$	c_{i-1}	$\overline{c_{i-1}}$

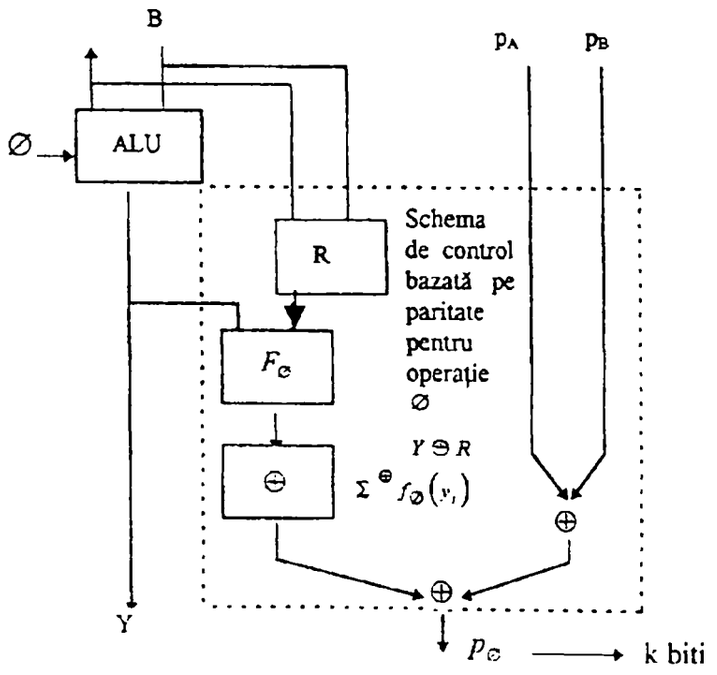


Figura 1.3. Verificarea parității pentru o operație arbitrară Φ

Teorema 1.2. Pentru intrările A şi B, fiecare de câte *k* biți, controlul bazat pe paritate, respectiv sindromul pentru o operație Φ a ALU devine:

$$S_\Phi = p_v \oplus p_y' = \left(\sum_{i=0}^{k-1}{}^{\oplus} y_i\right) \oplus \left(\left(\sum_{i=0}^{k-1}{}^{\oplus} r_i\right) \oplus p_{Ai} \oplus p_B\right) = \left(\sum_{i=0}^{k-1}{}^{\oplus} f_\Phi(y_i)\right) \oplus p_{Ai} \oplus p_B$$

unde $f_\Phi(y_i) = y_i \oplus r_i = a_i \oplus b_i$ iar $S_\Phi = 1$ pentru eroare detectată, respectiv $S_\Phi = 0$ pentru funcționare normală, fără eroare.

Figura 1.3 prezintă schematic controlul bazat pe paritate pentru o operație Φ a ALU.

Teorema 1.3. Schema de control bazată pe paritate din Figura 1.4 detectează toate erorile singulare din intrările A şi B, dacă nu există defecte nici în ALU, nici în schema de control.

Demonstrație: Ecuația (1.8) poate fi supusă următoarei transformări evidente.

$$S_\Phi = \left(\sum_{i=0}^{k-1}{}^{\oplus} f_\Phi(y_i)\right) \oplus p_{Ai} \oplus p_B = \left(\sum_{i=0}^{k-1}{}^{\oplus} a_i \oplus b_i\right) \oplus p_{Ai} \oplus p_B \qquad (1.9)$$

Prin urmare:

$$\frac{dS_\Phi}{da_i} = 1 \text{ si } \frac{dS_\Phi}{db_i} = 1. \qquad (1.10)$$

Aceasta arată că o eroare singulară pe intrări poate fi întotdeauna detectată. În cele ce urmează să considerăm, bazat pe rezultate anterioare, corecția erorilor care afectează operații ALU. Controlul bazat pe paritate în discuție poate fi aplicat pentru a controla grupuri de biți definite de rândurile matricii de control H. Pentru un cod (*n,k*) există *n* - *k* = *r* grupuri de control, fiecare dintre acestea constituind seturi de biți incluşi într-un rând al următoarei matrici H.

$$H = [H_e, H_r]_{rxn}, \qquad (1.11)$$

unde H_e reprezintă matricea de codificare (*r x k*) şi I_r reprezintă matricea identitate (*r x v*). Două cuvinte de cod intrate constituie vectorii de forma (A, C_A) şi (B, C_B) în care

$$C_A = (C_{A,0}, C_{A,1}, \ldots, C_{A,\,r-1}) \text{ şi } C_B = (C_{B,0}, C_{B,1}, \ldots, C_{B,r-1})$$

sunt grupuri de biți de control.

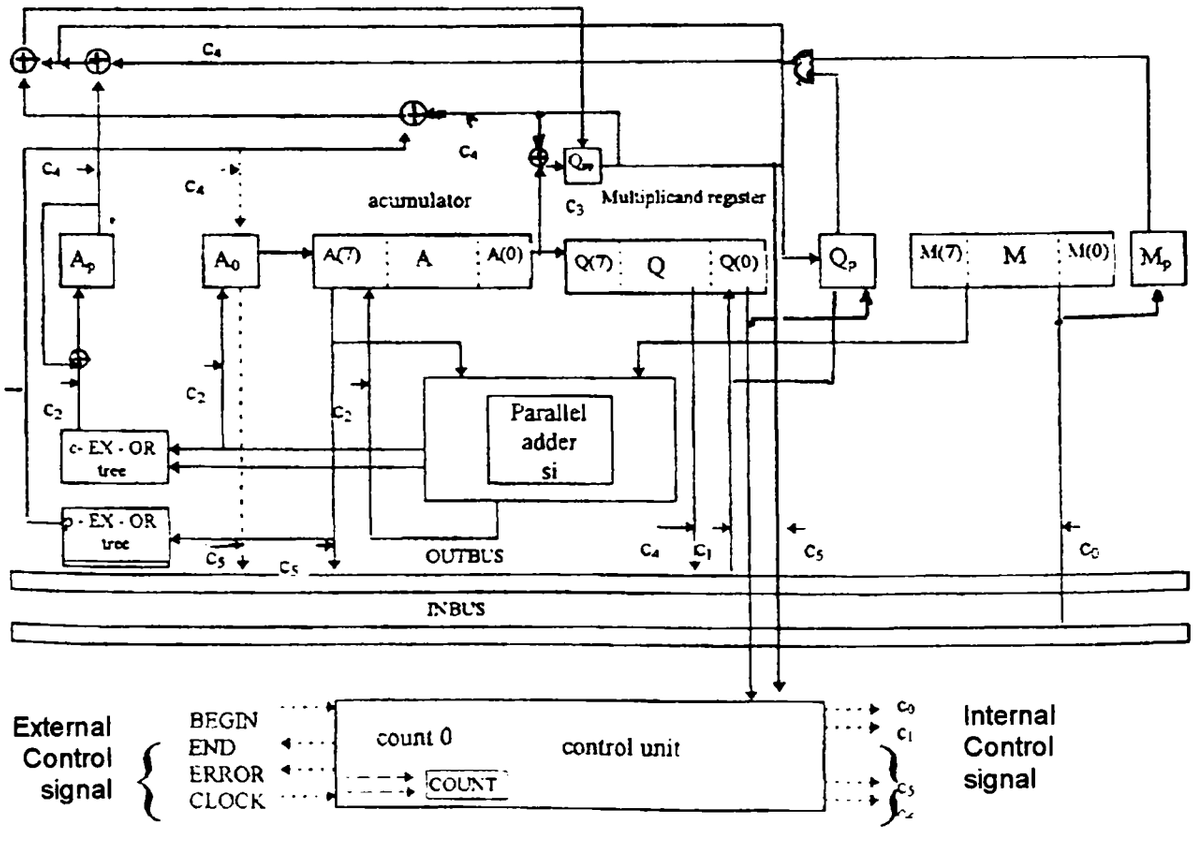


Figura 1.4. Controlul bazat pe paritate pentru o operație Φ a ALU.

De asemenea, definim pe C astfel: $C = C_A \oplus C_B = (c_0, c_1, \ldots, c_{r-1})$ unde

$$c_i = c_{A,i} \oplus c_{B,i}$$

Ieşirea ALU se prezintă în forma (Y, C_y) unde $C_y = (c_{y,0}, c_{y,1}, \ldots, c_{y,r-1})$ reprezintă grupul de biți de control pentru ieşirea Y.

O procedură de corectare a erorii constă din trei paşi mai importanți: (1) generarea sindromului, (2) determinarea poziției erorii (decodificarea sindromului) şi (3) inversarea bitului eronat.

(1) Generarea sindromului pentru ieşirea Y implică soluționarea următoarelor probleme.

a) Generarea biților de control: $C_y = Y.H_e^T$

b) Anticiparea biților de control: $C_p = (c_{p,o}, c_{p,1}, \ldots, c_{p,r-1})$ iar $C_p = R.H_e^T \oplus C$

c) Generarea sindromului:

(2) Determinarea pozției eronate se efectuează din sindrom, astfel:

W (S) = 0 - funcționare corectă, fără eroare; W (S) = 1 - eroare în partea biților de control C_p şi W (S) $\geq$ 2 - eroare în ieşirea Y, unde W(S) semnifică ponderea sindromului. Aceasta este determinată cu precizie din matricea H, în special pentru cazul corecției de eroare a ieşirii Y.

(3) Indicatorul de eroare E = (E_y, E_C) specifică modelul de biți corespunzător erorii care se impune corectată, unde E_y specifică eroarea ieşirii Y şi E_C specifică eroarea în partea biților de control [Mang-97]. Ieşirea corectată (Y, C_y) se obține prin intermediul menționatului indicator de eroare astfel:

$$Y = Y \oplus E_y \qquad \text{şi} \qquad C = C_p \oplus E_C.$$

1.2.2. Coduri AN

Codurile aritmetice au la baza codificării fie operația de înmulțire, fie cea de împărțire [RaFu-89], [Park-90], [Brya-91]. Mecanismele de detecție şi corecție sunt dependente de tipul erorilor potențiale - care pot fi singulare, multiple, de byte singular sau unidirecționale -, fiind implicate coduri specifice diferind în implementare ca redundanță solicitată [BrSt-88], [NaAb-90], [WoGo-94].

Cele mai simple coduri aritmetice sunt codurile AN. Un cod AN este format prin înmulțirea fiecărui cuvânt de dată N cu o constantă A. Aceste coduri sunt invariante la adunări şi scăderi dar nu şi la înmulțiri şi împărțiri. Fie N_1 şi N_2 doi operanzi codificați, cuvintele de cod obținute fiind AN_1 şi AN_2, care sunt supuşi spre exemplu operației de adunare. Rezultatul este $A(N_1+N_2)$, iar corectitudinea lui se verifică prin împărțirea la modulul A utilizat la codificare. Dacă această operație suplimentară se soldează cu rest 0, atunci este validată operația de adunare verificată, iar obținerea unui rest nenul este interpretat ca detecție de eroare, astfel încât punerea în evidență a erorii singulare necesită ca A să fie diferit de o putere a lui 2. Faptul că verificarea se bazează pe un cod AN care este nesistematic, precum şi redundanța care o solicită, cu precădere, la operații mai complexe decât adunarea determină răspândirea practică mai strânsă a acestui tip de cod aritmetic - precum şi a formei sale evoluate în cod (AN+3) care este înzestrat cu proprietatea de autocomplementaritate [Prad-86], [RaFu-89] – în raport cu clasa aşa numitelor coduri cu rest (residue codes) la care codificarea operanzilor se bazează pe operația de împărțire. Într-o descriere sintetică, fiind dat un număr întreg şi pozitiv N reprezentat în formă binară, prin împărțirea acestuia la un alt număr întreg pozitiv A, numit modul, se obține astfel restul R = N mod A, care concatenat la N în forma (NR) formează cuvântul codificat. Notând cu N_1 şi N_2 doi operanzi întregi şi pozitivi cu resturile asociate R_1 şi R_2 obținute prin împărțirea la acelaşi modul A, verificarea operațiilor aritmetice se bazează pe următoarele proprietăți ale codurilor cu resturi [Beck-88], [RaFu-89], [Kant-93], [StFK-93], [Hilj-94]:

$$(N_1 \pm N_2) \bmod A = (R_1 \pm R_2) \bmod A \quad (1.12)$$

$$(N_1 . N_2) \bmod A = (R_1 . R_2) \bmod A \quad (1.13)$$

În mod schematic, în figura 1.5 se prezintă blocurile care intervin la execuția operațiilor care intervin în (1.12) şi (1.13) sugerând asupra investiției în circuitele care solicită detecția erorilor bazată pe coduri cu resturi.

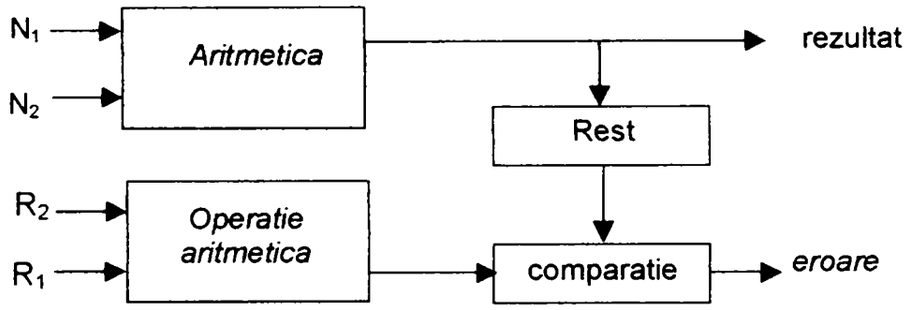


Figura 1.5. Schema bloc pentru verificarea operațiilor aritmetice

Astfel, se observă că pe de o parte sunt operate cele două numere N_1 şi N_2, fiind calculat restul rezultatului, şi pe de altă parte resturile R_1 şi R_2, calculate în prealabil sunt supuse aceleiaşi operații. Coincidența la nivel de bit a celor două resturi indică execuția corectă a operației, iar necoincidența chiar şi a unui singur bit conduce la semnalarea unei erori. Din păcate, o proprietate asemănătoare cu (1.12) şi (1.13) nu este valabilă şi pentru operația de împărțire pentru care în baza identității specifice acestei operații se poate însă scrie:

$$N_2 = QN_1 + S \quad (1.14)$$

în care Q şi S reprezintă câtul, respectiv restul obținut la împărțirea lui N_2 şi N_1.

Calculând pe lângă resturile R_1 şi R_2 corespunzătoare celor doi operanzi N_1 şi N_2 şi resturile R_Q şi R_S corespunzătoare rezultatelor Q şi S, toate utilizând acelaşi modul împărțitor A, detecția erorilor la operația de împărțire se poate baza pe următoarea relație, obținută prin adaptarea identității anterioare (1.14):

$$(R_2 - R_S) \bmod A = (RQ_1 . R_2) \bmod A \quad (1.15)$$

Verificarea corectitudinii execuției operației implică calcule consistente, consumatoare de timp, care trebuie să succeadă execuția operației verificate. În plus, controlul prin coduri cu resturi implică dificultăți de implementare din punct de vedere al rezolvării penalități suplimentare de performanță şi/sau cost [RaFu-89], [PhVa-94].

De exemplu, la adunarea numerelor binare întregi N_1 şi N_2 poate rezulta o sumă care, având în vedere că registrele prezintă un număr determinat de ranguri permițând reprezentarea numărului cel mai mare, să zicem M, poate să depăşească capacitatea registrului fiind obținută în locul sumei (N_1-N_2) dorite, valoarea (N_1+N_2-M). Calculând restul modulo A pentru aceasta din urmă, el nu este în mod necesar egal cu (R_1+R_2) modA, aşa cum o cere relația (1.12) ceea ce necesită restricționarea valorii modulului A pentru ca totuşi detecția erorilor să poată fi totuşi efectuată. Într-adevăr, dacă M este un multiplu de A, atunci controlul poate fi extins şi asupra situației de depăşire a capacității registrelor, fiind bazat pe următoarea relație:

$$(N_1+N_2-M) \bmod A = (R_1+R_2) \bmod A \qquad (1.16)$$

Un alt aspect deficitar al utilizării codurilor cu resturi apare atunci când, din rațiuni de economie se apelează, în detrimentul performanței, la o structură verticală de implementare a verificării sugerate prin figura 1.5, bazată pe operare bytesliced şi când o anumită poziție binară afectată de către defect provoacă, în mod repetat, eronarea respectivului bit din mai mulți bytes [Mang-97].

De altfel, ca o caracteristică generală a implementării reale a codurilor cu resturi este dependența decisivă a complexității părții de aritmetică de alegerea modulului A, de care depinde de asemenea capacitatea de detecție. Calculul restului modulo A poate fi simplificat în mare măsură dacă - admițând un număr întreg x de forma $x = (x_{n-1},x_{n-2},....,x_i,....,x_0)$ unde x_i constituie un byte de b biți pentru i = 0,1,..., n -1 - valoarea modulului A se alege egală cu 2^b - 1, unde b este un număr întreg mai mare decât 1. Se obține astfel un aşa numit low-cost-code care necesită în scopul codificării numerelor un sumator binar de b biți cu transport end around.

1.2.3. Coduri sumă de control

Checksum codes (coduri sumă de control) pot fi folosite pentru detectarea erorilor singulare de byte. Un cod sumă de control este un set de vectori alcătuiți din $(n+1)$ ai simbolurilor din setul Z_q. Fiecare vector are o componentă numită simbol de control, care egalează suma modulo q a altor componente ale vectorului, denumite simboluri de control. Din această definiție este evident că minimul distanței Hamming a unui cod sumă de control este 2.

Definiția 1.2. Un cod sumă de control este un set

$$\left\{(x_c, x_{n-1},...x_0) \mid (x_i, x_c \in Z_q) \text{si} \left(x_c = \sum_{0 \le i \le n-1} x_i \bmod q \right) \right\}$$

În particular, prezintă interes codurile pentru care $q = 2^b$ unde b reprezintă un număr întreg mai mare decât 1. Dacă $b = 1$, se obține cunoscutul cod de paritate.

În aceste coduri, fiecare a simbol aparținând lui Z^b_2 poate fi codificat ca un cuvânt de b biți. Prin urmare, fiecare cuvânt de cod are n x b biți de informație şi b biți de control, conform cu reprezentarea din Figura 1.6.

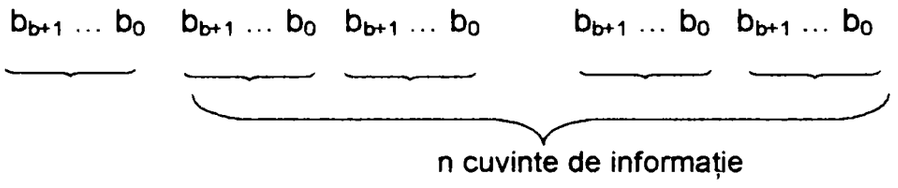


Figura 1.6. Structura cuvântului de cod.

Fie adunarea modulo 2^b a vectorilor cu un număr oarecare de simboluri şi admitem $(n + 1)$ ca aparținând la Z_2^b.

Să admitem că doi vectori reprezentați în codul sumă de control au forma:

$$A = (a_C, a_{n-1}, a_{n-2}, \ldots, a_i, \ldots, a_0)$$

$$B = (b_C, b_{n-1}, b_{n-2}, \ldots, b_i, \ldots, b_0)$$

în care a_C şi b_C sunt simboluri de control, iar $(a_{n-1}, a_{n-2}, \ldots, a_i, \ldots, a_0)$ şi $(b_{n-1}, b_{n-2}, \ldots, b_i, \ldots, b_0)$ reprezintă părțile de informație utilă, constituită din vectorii:

$$A_d = (a_{n-1}, a_{n-2}, \ldots, a_i, \ldots, a_0)$$

$$B_d = (b_{n-1}, b_{n-2}, \ldots, b_i, \ldots, b_0)$$

Părților de informație utilă A_d şi B_d le corespund numerele întregi (A_d) şi (B_d) date de relațiile:

$$[A_d] = \sum_{0 \le i \le n-1} a_i \cdot 2^{bi} \quad \text{si} \quad [B_d] = \sum_{0 \le i \le n-1} b_i \cdot 2^{bi} \tag{1.17}$$

Adunarea ordinară a părților de informație utilă (A_d şi B_d) corespunzătoare cuvintelor de cod este definită astfel:

$$S_d = A_d + B_d + C, \text{ unde}$$

$$C = (c_{n-2}, c_{n-3}, \ldots, c_1, c_0, c_{-1})$$

cu

$$C_i = \begin{cases} 0 \text{ daca } a_i + b_i + c_i < 2^b \\ 1 \text{ daca } a_i + b_i + c_i \geq 2^b \end{cases} \quad \text{pentru}$$
$$0 \leq i \leq n-2 \text{ si } c_{-1} = c_{in}$$

Vectorului sumă S_d îi corespunde numărul întreg S_d dat de relația

$$[S_d] = ([A_d] + [B_d]) \bmod M$$

unde $M = 2^{bn}$ şi $c_{in} = 0$ pentru adunarea în complement de 2, respectiv $M = 2^{bn} - 1$ şi $c_{in} = c_{n-1}$ pentru adunarea în complement de 1.

Pe de o parte, simbolul de control al sumei celor două părţi de informaţie utilă este obţinut prin însumarea simbolurilor sumă obţinute, adică

$$S_c = \left(\sum_{0 \leq i \leq n-1} s_i \right) \bmod 2^b$$

S_C poate fi anticipat prin utilizarea simbolurilor de control corespunzătoare vectorilor intrare, şi C_C constituind simbolul de control corespunzător transportului, astfel:

$$S_c = (a_c + b_c + c_c) \bmod 2^b, \text{ unde}$$
$$C_c = \left(\sum_{0 \leq i \leq n-1} c_{i-1} \right) \bmod 2^b, \text{ iar } c_{-1} = c_{in}$$

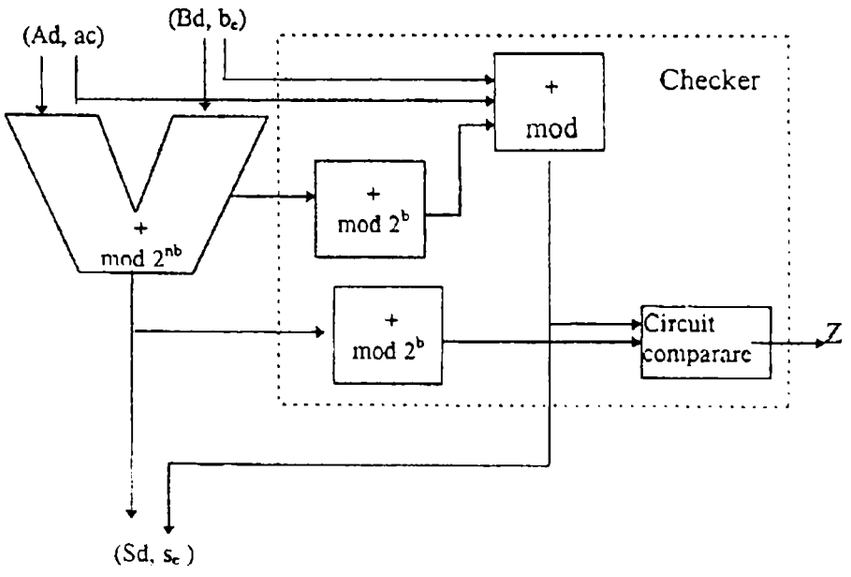


Figura 1.7. Checker predictiv

Ca şi la însumarea cu control de paritate, simbolurile de control, calculat şi anticipat sunt comparate iar rezultatul este interpretat în caz de inegalitate ca eroare (figura 1.7). Această schemă de control aparţine clasei de predicţie şi suferă de aceeaşi problemă relevată la însumarea cu control de paritate care constă în faptul că defectele ce afectează partea de schemă referitoare la generarea transportului nu pot fi detectate pentru că ele produc erori care se compensează.

Altfel, dacă S_C şi C_C sunt eronate simultan, cu toate că suma nu este corectă, nu apare indicaţia de eroare. Această problemă poate fi rezolvată prin duplicarea logicii de generare a transportului, strategie care poate fi aplicată eficient la însumarea bitesliced.

1.2.4. Coduri combinate utilizate pentru detecţia erorilor în operaţia de adunare binară

Codurile bazate pe paritate s-au dovedit foarte eficiente şi favorabile prin prisma costului pentru memorii şi operaţii de transfer de date. Codurile cu resturi, pe de altă parte, apar foarte atractive pentru controlul operaţiilor aritmetice, cum ar fi ADD, MULTIPLY şi altele.

A fost propusă combinarea codurilor de paritate cu cele reziduale în aşa numite coduri combinate care să permită eficienţa corecţiei de erori pentru toate operaţiile unei unităţi aritmetice şi logice.

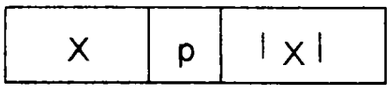


Figura 1.8. Structura unui cuvânt de cod combinat

Un cod combinat este un cod format din blocuri de lungime *n*, în care informaţia este urmată de un grup de biţi de paritate şi un rest de control, după cum se prezintă în figura 1.8.

Pentru informaţia X cuvântul de cod corespunzător este $[X,P,|X|_m]$ unde P este un grup de biţi de paritate calculaţi pentru bytes lui X, iar *xxxx* exprimă restul lui X modulo *m* ($=2^b-1$). Codul şi mecanismul său de corecţie va fi descris prin intermediul unui exemplu. Fie informaţia X reprezentată de cel mult 7 bytes (B_i, $i = 0, ...,6$) fiecare format din 3 biţi astfel:

$$X = [\underbrace{x_{20}\,x_{19}\,x_{18}}_{B_6}\ \underbrace{x_{17}\,x_{16}\,x_{15}}_{B_5}\ \underbrace{x_{14}\,x_{13}\,x_{12}}_{B_4}\ \underbrace{x_{11}\ x_{10}\,x_9}_{B_3}\ \underbrace{x_8\ x_7\,x_6}_{B_2}\ \underbrace{x_5\ x_4\ x_3}_{B_1}\ \underbrace{x_2\ x_1\,x_0}_{B_0}]$$

a) Porțiunea P a biților de paritate este alcătuită din 3 biți (P_2, P_1, P_0) prin alegerea de aşa manieră a biților pentru generarea parităților încât bytes să corespundă ecuațiilor de control dintr-o matrice de control asociată unui cod Hamming corector al erorii singulare.

$$H = \begin{pmatrix} 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{pmatrix} \Rightarrow \begin{aligned} p_0 &= c_0 \oplus c_2 \oplus c_4 \oplus c_6 \\ p_1 &= c_1 \oplus c_2 \oplus c_5 \oplus c_6 \\ p_2 &= c_3 \oplus c_4 \oplus c_5 \oplus c_6 \end{aligned}$$

$c_0\ c_1\ c_2\ c_3\ c_4\ c_5\ c_6$

Vom avea deci pentru cei 3 biți de paritate ai codului:

$$P_0 = B_0 \oplus B_2 \oplus B_4 \oplus B_6 = (x_0 \oplus x_1 \oplus x_2) \oplus (x_6 \oplus x_7 \oplus x_8) \oplus (x_{12} \oplus x_{13} \oplus x_{14}) \oplus (x_{18} \oplus x_{19} \oplus x_{20})$$

$$P_1 = B_1 \oplus B_2 \oplus B_5 \oplus B_6 = (x_3 \oplus x_4 \oplus x_5) \oplus (x_6 \oplus x_7 \oplus x_8) \oplus (x_{15} \oplus x_{16} \oplus x_{17}) \oplus (x_{18} \oplus x_{19} \oplus x_{20})$$

$$P_2 = B_3 \oplus B_4 \oplus B_5 \oplus B_6 = (x_9 \oplus x_{10} \oplus x_{11}) \oplus (x_{12} \oplus x_{13} \oplus x_{14}) \oplus (x_{15} \oplus x_{16} \oplus x_{17}) \oplus (x_{18} \oplus x_{19} \oplus x_{20})$$

b) Restul de control $|X|_7$ este obținut prin adunarea tuturor bytes de informație B_0, B_1,..., B_6 utilizând sumatoare modulo 7.

Astfel având X dat de următorul vector binar:

$$X = 10110111001101101011 1$$

obținem:

$$P_0 = 1,\ P_1 = 0 \text{ şi } P_2 = 0.$$

Pe de altă parte $|X_7| = (5+5+6+3+3+2+7)=3$, deci cuvântul codificat va fi:

$$[X,P,|X|_7] = \underbrace{1\,0\,1\,1\,0\,1\,1\,1\,0\,0\,1\,1\,0\,1\,1\,0\,1\,1\,1}_{X}\ \underbrace{0\,0\,1}_{P}\ \underbrace{0\,1\,1}_{\lfloor x \rfloor_7}$$

În continuare să analizăm capacitățile de detecție şi corecție ale codului combinat astfel generat. Admitem că în cuvântul eronat X bitul 46 care are valoarea corectă 0 devine $x_{16} = 1$ (tipul de eroare 0xxxx1, respectiv eroare de 0). Atunci prin evaluarea ecuațiilor de control a parității referitoare la cuvântul eronat X se obține sindromul de participare Sp = 110 indicând o eroare în byte-ul B_5 a lui X. Într-adevăr la exemplul considerat avem $P_0 = 1$, $P_1 = 1$ şi $P_2 = 1$, deci

$$S_p = (P_2 P_1 P_0) \oplus (P_2^* P_1^* P_0^*) = (001) \oplus (111) = 110$$

Pentru a obține poziția bitului eronat, se calculează sindromul rezidual:

$$S_r = |X-X| = ||X|_7 - |X|_7|_7.$$

Într-adevăr prin eroarea 0→1 a poziției binare x_{16} rezultă o mărime a erorii de 2^{16} care modulo 7 dă $/2^{16}/_7 = 2$ indicând asupra celui de al doilea bit din cadrul byte-ului eronat. Pentru exemplul considerat, avem

$$|X'|_7 = (5+6+3+3+2+7) \quad S_r = ||X|_7 - |X|_7|_7 = |5-3|_7 = 2$$

Pe de altă parte, dacă eroarea este de tipul 1→0 (eroare de 1) în cadrul aceluiaşi byte detectat ca eronat prin aceleaşi ecuații de control a parității, dar vizând poziția binară $X_7 = 1$, deci $X_{17} = 0$ atunci se obține mărimea erorii -2^{17}, care modulo 7 dă valoarea 3 indicând că eroarea a afectat al 3-lea bit din cadrul byte-ului eronat. Revenind la exemplul numeric considerat pentru cazul $X_{17} = 1$, obținem mai întâi:

$$||X'|_7| = |6-3|_7 = 3+6+3+3+2+7 \bmod 7 = 6$$

şi apoi

$$S_r = ||X''|_7 - |X|_7|_7 = |6-3|_7 = 3$$

Se impune remarcat că există un sindrom de paritate unic pentru fiecare dintre cei 7 bytes şi un sindrom rezidual unic pentru fiecare bit şi tip de eroare din cadrul byte-ului eronat. De asemenea, trebuie subliniat că atunci când eroarea afectează partea de informație utilă, atât biții de control de paritate cât şi cei de control rezidual detectează eroarea şi împreună permit corecția acesteia. Pe de altă parte, dacă eroarea vizează biții de paritate P sau cei de control rezidual, doar unul dintre sindromuri este diferit de 0. În consecință este valabilă următoarea strategie de localizare a erorii, respectiv de corelație a acesteia:

$S_p = 0$, $S_r = 0 \Rightarrow$ nu există eroare;

$S_p = 0$, $S_r \neq 0 \Rightarrow$ eroare la $|X|_7$

$S_p \neq 0$, $S_r = 0 \Rightarrow$ eroare la P;

$S_p \neq 0$, $S_r \neq 0 \Rightarrow$ eroare la X.

Dacă $S_p = i \neq 0$ şi $S_r = j \neq 0$, atunci este eronat B_{i-1} (i = 1,2,....,7) şi dacă:

j = 1 ⇒ eroare la primul bit, de tipul 0 → 1

j = 6 ⇒ eroare la primul bit, de tipul 1 → 0

j = 2 ⇒ eroare la al doilea bit, de tipul 0 → 1

j = 5 ⇒ eroare la al doilea bit de tipul 1 → 0

j = 4 ⇒ eroare la al treilea bit, de tipul 0 → 1

j = 3 ⇒ eroare la al treilea bit, de tipul 1 → 0

Prin urmare, orice eroare de bit singular în X poate fi detectată şi corectată.

Din punct de vedere al implementării, în [RaFu-89] [Katt-96] codul combinat prezentat este supus unei modificări pentru a-l face mai atractiv. Aceasta constă în substituirea byte-ului de paritate P prin biţi de paritate, câte unul pentru fiecare din cei *k* bytes de informaţie corespunzători lui X. Este de remarcat că această modificare nu alterează proprietăţile de detecţie/corecţie ale codului. Totuşi prin această modificare creşte redundanţa codului (*k* biţi de paritate în locul celor $\log_2 k$) dar implementarea este mai directă. Codul combinat modificat este ilustrat în figura 1.9. În [Mang-99] este analizată implementarea controlului cu acest cod combinat modificat.

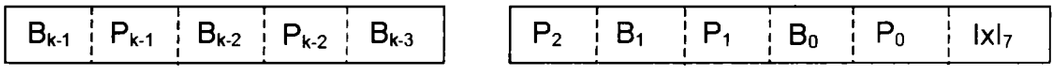


Figura 1.9. Codul combinat modificat.

1.3. Tehnici de compresie

Metodele convenţionale de testare presupun o comparare bit-cu-bit a valorilor de ieşire cu valori calculate şi salvate anterior. Aceste metode au dezavantajul că necesită o cantitate mare de memorie pentru păstrarea valorilor corecte ale ieşirilor asociate cu toţi vectorii de test. În paragrafele ce urmează vom considera o metodă alternativă mai simplă şi care necesită mai puţină memorie. Informaţiile salvate sunt comprimate şi se numesc semnături. Acest concept este ilustrat în figura 1.10, unde se poate observa că defectul este detectat dacă semnătura S(R') obţinută din circuitul aflat sub test (CUT – circuit under test) diferă de semnătura precalculată $S(R_0)$, corespunzătoare unui circuit fără defecţiuni.

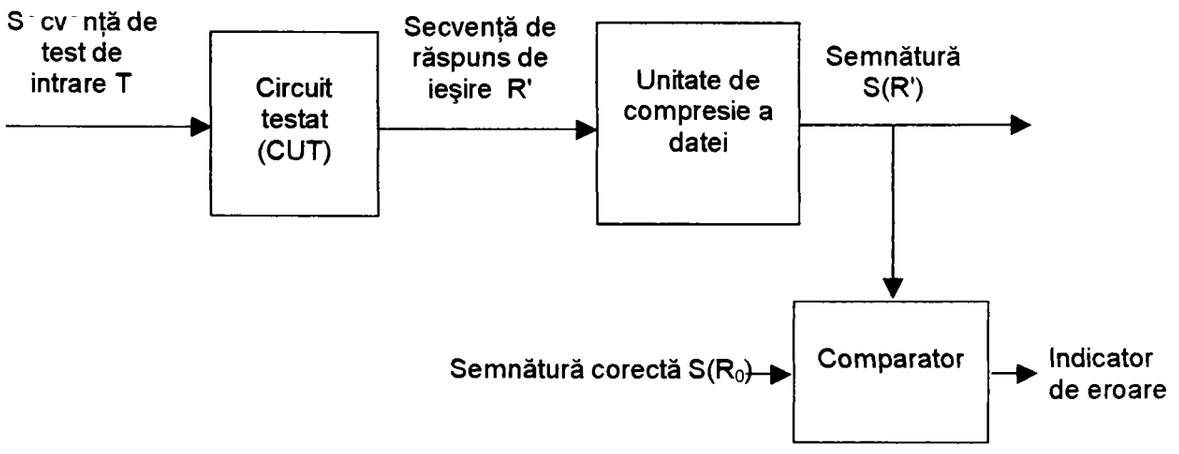


Figura 1.10. Testare pe baza compresiei răspunsului de test

Dacă circuitul de comprimare este simplu, este posibilă includerea acestuia în circuitul care se testează, realizându-se în acest fel unul din aspectele BIST (built-in self-test) importante, şi anume detectarea erorilor de răspuns.

În continuare vor fi trecute în revistă cinci tehnici de comprimare: numărarea de unu-uri, numărarea tranzițiilor, verificarea parității, verificarea de sindrom şi analiza de semnătură – o metodă mult utilizată.

1.3.1. Aspecte generale ale tehnicilor de comprimare

Una dintre cerințele tehnicilor de comprimare este ca acestea să fie uşor de implementat printr-un circuit simplu, care să poată fi inclus în circuitul aflat sub test ca parte a logicii sale BIST.

De asemenea, este important ca procedurile de compresie să nu introducă întârzieri ale semnalelor care pot influența comportarea normală sau timpii de execuție ai testului pentru CUT. Pe de altă parte, lungimea semnăturii de test trebuie să fie un factor logaritmic al lungimii datei de răspuns de la ieşire pentru ca volumul de memorie să poată fi mult micşorat. O altă cerință este aceea ca pentru orice defect şi răspuns asociat al ieşirii care conține una sau mai multe erori, semnăturile să fie diferite de cele corecte; în acest caz se poate spune că metoda de comprimare nu pierde informații.

Practic, nu sunt cunoscute proceduri de comprimare care să satisfacă toate aceste cerințe. Cea mai greu de respectat este aceea ca semnătura corectă să difere de cea care conține defectul, deoarece un defect poate determina erori "offsetting" care să nu fie detectate. Aceste situații sunt numite mascări de eroare iar răspunsul eronat este numit "alias" al răspunsului de ieşire corect.

Există trei posibilități de măsurare a caracteristicilor de mascare asociate cu o tehnică de comprimare. Prima este simularea circuitului şi a tehnicii de comprimare şi determinarea defectelor care sunt detectate. Această metodă necesită simularea defectului şi este costisitoare, în special dacă secvențele de test sunt lungi. A doua metodă clasifică secvențele de răspuns de ieşire ale circuitelor defecte în categorii, ca de exemplu erori "single-bit" sau "burst". Acestea sunt apoi analizate pentru a se determina gradul de mascare asociat diverselor proceduri. A treia metodă constă în calcularea fracțiunii tuturor secvențelor posibile de răspuns eronate care pot produce mascarea. Pentru a avea rezultate reale trebuie cunoscută distribuția secvențelor eronate. Acest lucru este deseori imposibil de determinat. În mod obişnuit toate secvențele posibile de ieşire se presupune că sunt la fel de probabile.

Metodele de compresie BIST se pot împărți în trei categorii:

1. - care nu necesită seturi speciale de test sau circuite speciale,
2. - care necesită seturi de test speciale,
3. - care necesită seturi de test speciale şi circuite speciale.

Pentru obținerea de semnături identice pentru copii multiple ale unui circuit fără defecte, toate circuitele secvențiale trebuiesc aduse la o stare inițială fixă înainte de aplicarea secvenței de test.

O altă problemă a tehnicilor de comprimare a răspunsului este cea a calculării semnăturii corecte. O posibilitate de obținere a acesteia este de a identifica o parte bună, de aplicare a testului actual acelei părți şi a avea hard-ul de comprimare pentru a genera semnătura. O altă metodă constă în simularea CUT şi a procedurii de compresie utilizând modele de test, dar această metodă devine costisitoare dacă vectorii de test sunt mari. O a treia posibilitate constă în implementarea mai multor copii ale CUT şi deducerea semnăturii corecte prin găsirea unui subset de circuite care produc aceaşi semnătură şi se presupune că nu au defecte.

1.3.2. Compresia prin numărarea de unu-uri

Fie un circuit C cu o singură ieşire şi fie $R = r_1, r_2, \ldots, r_m$ răspunsul la ieşirea lui C. Semnătura IC(R) este numărul de 1-uri din R. De exemplu

$$IC(R) = \sum_i r_i$$

unde $0 \le IC(R) \le m$.

Compresorul poate fi un simplu numărător, iar gradul de comprimare este:

$$\lceil \log_2(m+1) \rceil.$$

Fie un circuit testat cu m vectori de intrare aleatori şi fie $IC(R_0) = r$, $0 \le r \le m$. Numărul de secvențe de m biți cu r 1-uri este $\binom{m}{r}$. Deci, $\binom{m}{r}$-1 astfel de secvențe sunt alias-uri. Raportul dintre secvențele de mascare şi totalitatea secvențelor eronate posibile este:

$$P_{IC}(M \mid m,r) = \frac{\binom{m}{r} - 1}{2^m - 1}$$

Dacă toate $2^m - 1$ secvențe de eroare au aceeaşi probabilitate, ceea ce nu este o presupunere reală, atunci $P_{IC}(M|m,r)$ poate fi considerată probabilitatea de mascare.

Testarea prin numărarea de 1-uri are următoarele proprietăți:

- probabilitatea de mascare este mică atunci când semnătura are valori apropriate de extremele domeniului,
- dacă $IC(R_0) = 0$ sau m atunci nu apare mascarea,

- un defect care produce un număr impar de erori este întotdeauna detectat; dacă numărul de erori este par, defectul poate fi detectat,
- inversarea ieşirilor nu va schimba proprietăţile statistice ale acestei tehnici,
- dacă circuitul aflat sub test este combinaţional, atunci vectorii de test pot fi permutaţi fără a fi afectată acoperirea defectului.

Circuitele cu ieşiri multiple se pot testa folosind numărătoare separate pe fiecare ieşire, iar pentru economie de suprafaţă se poate utiliza o conversie paralel-la-serial a ieşirilor împreună cu un singur numărător.

În [AbBF-90] s-a demonstrat că pentru circuitele combinaţionale probabilitatea de mascare, în cazul comprimării prin numărare de 1-uri, descreşte asimptotic la valoarea $(\pi m)^{-1/2}$.

1.3.3. Compresia prin numărarea tranziţiilor

În testarea prin numărarea tranziţiilor (TC – Transition Counting), semnătura reprezintă numărul de tranziţii de la 0 la 1 şi de la 1 la 0 din şirul datei de ieşire. Astfel numărul de tranziţii asociat cu o secvenţă $R = r_1, r_2, \ldots, r_m$ este

$$TC(R) = \sum_{i=1}^{m-1} (r_i \oplus r_{i+1})$$

unde Σ reprezintă o adunare aritmetică şi $\oplus$ este suma modulo 2. Întrucât $0 \leq TC(R) \leq (m - 1)$, circuitul de compresie a răspunsului este un detector de tranziţii şi un numărător cu $\lceil \log_2 m \rceil$ stări.

Fie T o secvenţă de test, de lungimea m, a unui circuti şi fie R_0 răspunsul fără defecţiuni, unde $TC(R_0) = r$. Fie R' o secvenţă binară arbitrară de lungimea m. R' are $(m - 1)$ vecinătăţi între biţi unde poate apărea o tranziţie. Există $\binom{m-1}{r}$ posibilităţi de asigurare a r tranziţii celor $m - 1$ vecinătăţi astfel ca R' să aibă un număr de tranziţii egal cu r. Întrucât secvenţa $\overline{R'}$ obţinută prin complementarea fiecărui bit a lui R' are acelaşi număr de tranziţii ca şi R', există $2\binom{m-1}{r}$ secvenţe posibile care au un număr de r tranziţii, din care doar una este răspunsul unui circuit fără defecţiuni.

Astfel există $2\binom{m-1}{r} - 1$ secvenţe de eroare posibile care determină un alias. Dacă toate secvenţele defecte au aceeaşi probabilitate de apariţie ca răspuns a unui circuit defect, atunci probabilitatea de mascare este dată de relaţia:

$$P_{TC}(M \mid m,r) = \frac{2\binom{m-1}{r}-1}{2^m-1}$$

Această funcție are proprietăți similare cu cea obținută în cazul numărării de 1-uri.

Spre deosebire de numărarea de 1-uri, numărarea tranzițiilor depinde de ordinea biților din vectorul răspuns. De asemenea, trebuie menționat faptul că numărarea tranzițiilor nu garantează detectarea tuturor erorilor de tip bit-singular. În [AbBF–90] s-a demonstrat că într-o secvență arbitrară de *m* biți, probabilitatea de mascare a erorii bit-singular este de (*m* - 2)/2*m*. De asemenea se demonstrează că probabilitatea de compresie prin numărarea tranzițiilor pentru un circuit combinațional este de $(\pi m)^{-1/2}$.

1.3.4. Compresia prin verificarea parității

Compresia prin verificare parității se poate realiza folosind un circuit ca cel din figura 1.11.

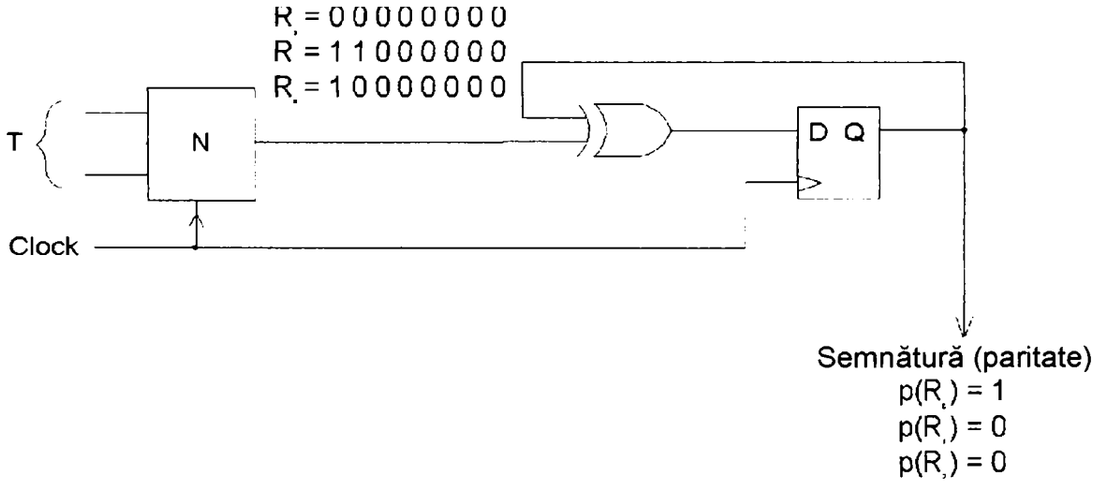


Figura 1.11. Compresia prin verificarea parității

Dacă starea inițială a bistabilului D este 0, semnătura S este paritatea răspunsului circuitului, mai exact, este 0 dacă paritatea este pară şi este 1 dacă paritatea este impară. Această schemă detectează toate erorile de tip bit-singular şi toate erorile biți-multipli care constau dintr-un număr impar de biți de eroare în secvența de răspuns. Defectele care crează un număr par de erori nu sunt detectate.

Presupunând că toate şirurile defecte de biţi au aceeaşi probabilitate de apariţie, probabilitatea de mascare se apropie de 1/2 cu cât *m* creşte.

Această metodă poate fi extinsă circuitelor cu ieşiri multiple în mai multe moduri. O posibilitate ar fi cea de înlocuire a porţii SAU EXCLUSIV cu o poartă cu intrări multiple sau o reţea de SAU EXCLUSIV şi toate ieşirile circuitului testat să comande această reţea. Dar în acest fel o eroare internă a circuitului afectează mai multe linii de ieşire. Dacă o astfel de eroare se propagă simultan la un număr par de linii de ieşire, atunci ea nu va avea efect asupra semnăturii. O a doua metodă presupune utilizarea unor compresoare de verificare a paritătii separat pentru fiecare ieşire, dar în acest fel este nevoie de hard suplimentar.

1.3.5. Testarea de sindrom

Testarea de sindrom constă într-o testare exhaustivă, de exemplu aplicarea tuturor celor 2^n vectori de test unui circuit combinaţional cu *n* intrări.

Fie un circuit cu o singură ieşire care implementează o funcţie *f*. Sindromul S (sau semnătura) reprezintă numărul normalizat de 1-uri din şirul de biţi rezultat: de exemplu, $S = k/2^n$, unde *k* este numărul de mintermeni ai funcţiei *f*. Astfel testarea de sindrom este un caz special de numărare de 1-uri. Mai exact, $0 \leq S \leq 1$. Sindromul unei porţi Si cu 3 intrări este 1/8 şi a unei porţi SAU cu trei intrări este 7/8. Sindromul lui *f* este o proprietate funcţională a lui *f*, indiferent de implementarea ei.

Testarea de sindrom este importantă datorită conceptului de testabilitate de sindrom; de exemplu, orice funcţie *f* poate fi realizată astfel ca toate defectele de "ţinere-pe" să fie detectabile prin sindrom [AbBF-90].

1.3.6. Analiza de semnătură

Analiza de semnătură este o tehnică de compresie bazată pe conceptul de "cyclic redundancy checking" (CRC) şi realizată hard prin utilizarea LFSR-urilor (Linear Feedback Shift Register) [Pfit-96].

1.3.6.1. LFSR

LFSR-urile precum şi variante ale acestora sunt folosite foarte mult în proiectarea DFT (Design for Testability) şi BIST (Built-in self-test).

În figura 1.12 sunt prezentate trei LFSR-uri. Acestea sunt circuite autonome (nu au decât intrări de tact). Fiecare celulă este formată din bistabile D sincrone.

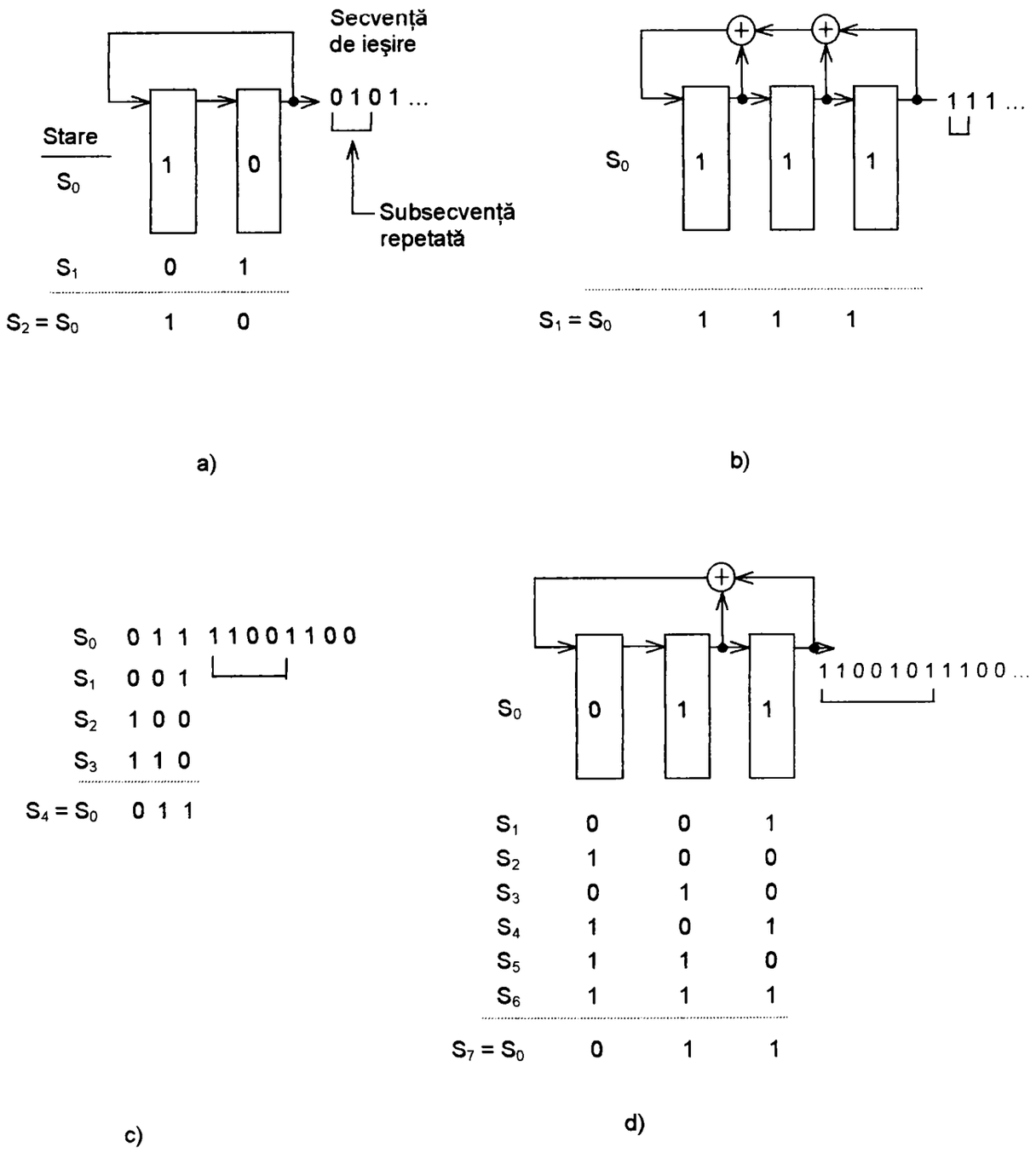


Figura 1.12. Feedback shift registers

Se ştie că aceste circuite sunt ciclice în sensul că la impulsuri repetate de tact ele trec printr-o secvență de stări fixe. În figura 1.12.a, LFSR-ul trece doar prin două stări. Dacă starea inițială ar fi 00 sau 11, starea nu s-ar schimba niciodată. Un registru de deplasare de *n* biți va trece prin cel mult *n* stări. Secvența de ieşire

generată de un astfel de circuit este de asemenea ciclică. Circuitul din figura 1.12.b pornind din starea inițială 111 (sau 000) produce o secvență ciclică de stări de lungime 1. În figura 1.12.c este prezentată secvența generată pentru circuitul din figura 1.12.b dacă starea inițială este 011. În figura 1.12.d este ilustrat cazul în care secvența stărilor generată de LFSR este de lungime 2^3-1. Pentru circuitele prezentate, starea "*all-0*" determină o secvență de lungime 1, şi anume însăşi starea "*all-0*". Circuitul din figura 1.12.d este un registru de deplasare de lungime maximă deoarece generează o secvență ciclică de stări de lungime 2^n-1, atâta timp cât starea inițială nu este "*all-0s*".

Un circuit liniar este o rețea logică formată din următoarele componente de bază: circuite de întârziere sau bistabile D, sumatoare modulo 2 şi multiplicatoare scalare modulo 2. În analiza unor astfel de circuite, toate operațiile sunt efectuate modulo 2. Astfel: $x + x = -x - x = x - x = 0$.

În figurile 1.13 şi 1.5 sunt prezentate două LFSR-uri, numite autonome. Aici C_i este o constantă binară şi $C_i = 1$ implică existența unei conexiuni iar $C_i = 0$ înseamnă lipsa conexiunii.

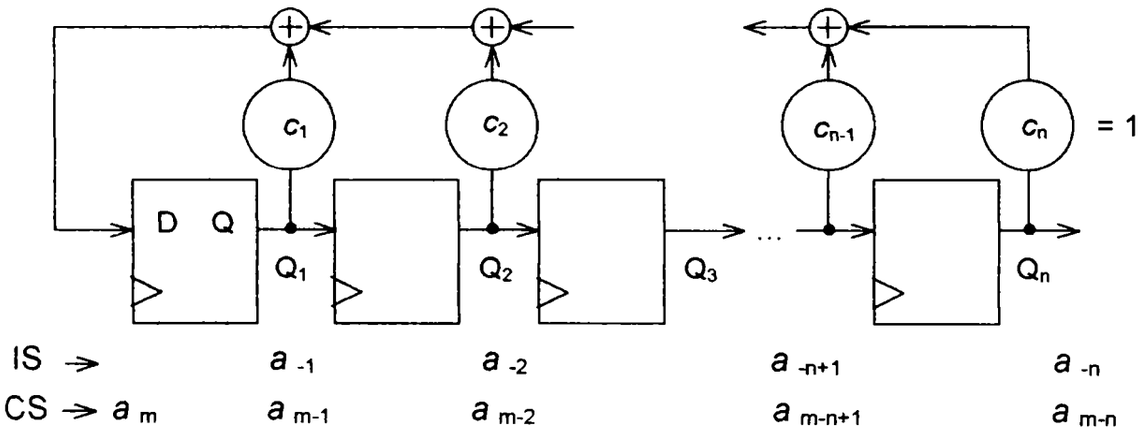


Figura 1.13. LFSR de tipul 1

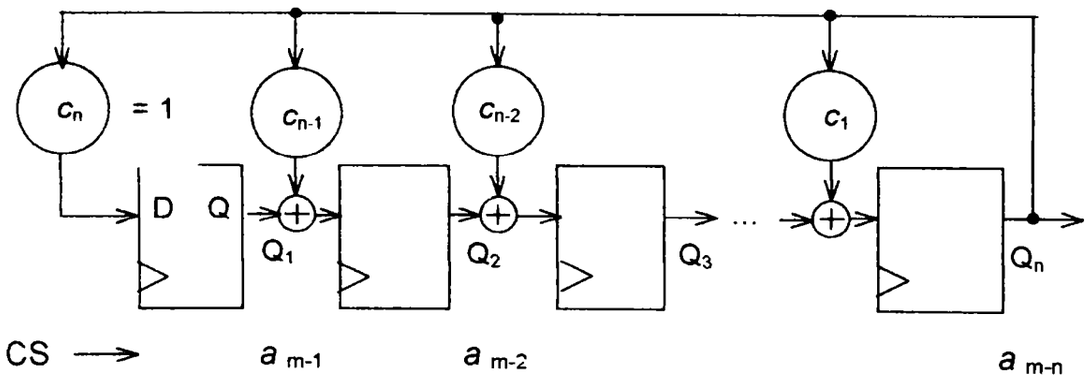


Figura 1.14. LFSR de tipul 2

Dacă $\{a_m\} = a_0, a_1, a_2, \ldots$ reprezintă secvența de ieşire generată de un LFSR, unde $a_i = 0$ sau 1, atunci această secvență poate fi exprimată astfel:

$$G(x) = \sum_{m=0}^{\infty} a_m \cdot x^m$$

unde G(x) este funcția de generare: $G(x) = a_0 + a_1 x + a_2 x^2 + \ldots + a_m x^m + \ldots$

Pentru structura LFSR de tip 1 se observă că dacă starea curentă (CS) a lui Q este a_{m-1}, pentru $i = 1,2,\ldots,n$ atunci:

$$a_m = \sum_{i=1}^{n} c_i \cdot a_{m-i}$$

În acest mod, operarea circuitului este definită cu ajutorul unei relații de recurență. Fie $a_{-1}, a_{-2}, \ldots, a_{-n+1}, a_{-n}$ starea inițială (IS) a LFSR-ului [RaTy-96]. Operarea circuitului începe cu n perioade de tact înainte de a se genera ieşirea a_0. Întrucât

$$G(x) = \sum_{m=0}^{\infty} a_m \cdot x^m$$

înlocuind a_m se obține:

$$\begin{aligned} G(x) &= \sum_{m=0}^{\infty} \sum_{i=1}^{n} c_i \cdot a_{m-i} \cdot x^m = \sum_{i=1}^{n} c_i \cdot x^i \cdot \sum_{m=0}^{\infty} a_{m-i} \cdot x^{m-i} \\ &= \sum_{i=1}^{n} c_i \cdot x^i \left[a_{-i} x^{-i} + \ldots + a_{-1} x^{-1} + \sum_{m=0}^{\infty} a_m x^m \right] \\ &= \sum_{i=1}^{n} c_i \cdot x^i \left[a_{-i} x^{-i} + \ldots + a_{-1} x^{-1} + G(x) \right] \end{aligned}$$

Deci,

$$G(x) = \sum_{i=1}^{n} c_i \cdot x^i \cdot G(x) + \sum_{i=1}^{n} c_i \cdot x^i \left(a_{-i} x^{-i} + \ldots + a_{-1} x^{-1} \right)$$

sau

$$G(x) = \frac{\sum_{i=1}^{n} c_i \cdot x^i \left(a_{-i} x^{-i} + \ldots + a_{-1} x^{-1} \right)}{1 + \sum_{i=1}^{n} c_i \cdot x^i} \qquad (1.18)$$

G(x) este o funcție de starea inițială $a_{-1},a_{-2},...,a_{-n}$ a LFSR-ului şi coeficienții de feedback $c_1,c_2,..,c_n$. Denominatorul din relația (1.18) dat de

$$P(x) = 1 + c_1x + c_2x^2 + ... + c_nx^n$$

se numeşte polinom caracteristic al secvenței $\{a_m\}$ şi al LFSR-ului. Pentru un LFSR cu n niveluri, $c_n = 1$. Este important de observat că P(x) este o funcție doar de coeficienții de feedback. Dacă $a_{-1} = a_{-2} = ... = a_{1-n} = 0$ şi $a_{-n} = 1$ atunci relația (1.18) devine:

$$G(x) = 1/P(x)$$

Polinomul caracteristic împreună cu starea inițială dau caracterul ciclic al LFSR-ului şi deci secvența de ieşire.

Pentru $a_{-1} = a_{-2} = ... = a_{1-n} = 0$ şi $a_{-n} = 1$

$$G(x) = \frac{1}{P(x)} = \sum_{m=0}^{\infty} a_m \cdot x^m \qquad (1.19)$$

şi întrucât secvența $\{a_m\}$ este ciclică, cu perioada p, relația (1.19) poate fi scrisă

$$\begin{aligned} 1/P(x) &= \left(a_0 + a_1x + ... + a_{p-1}x^{p-1}\right) \\ &\quad + x^p\left(a_0 + a_1x + ... + a_{p-1}x^{p-1}\right) \\ &\quad + x^{2p}\left(a_0 + a_1x + ... + a_{p-1}x^{p-1}\right) + ... \\ &= \left(a_0 + a_1x + ... + a_{p-1}x^{p-1}\right)\left(1 + x^p + x^{2p} + ...\right) \\ &= \frac{a_0 + a_1x + ... + a_{p-1}x^{p-1}}{1 - x^p} \end{aligned}$$

astfel:

Astfel se observă că P(x) se divide cu $1 - x^p$.

În [AbBF–90] se arată de asemenea că

$$P^*(x) = c_n + c_{n-1}x + c_{n-2}x^2 + ... + c_1x^{n-1} + x^n$$

unde P*(x) este polinomul reciproc al lui P(x) şi $P^*(x) = x^n \cdot P(1/x)$.

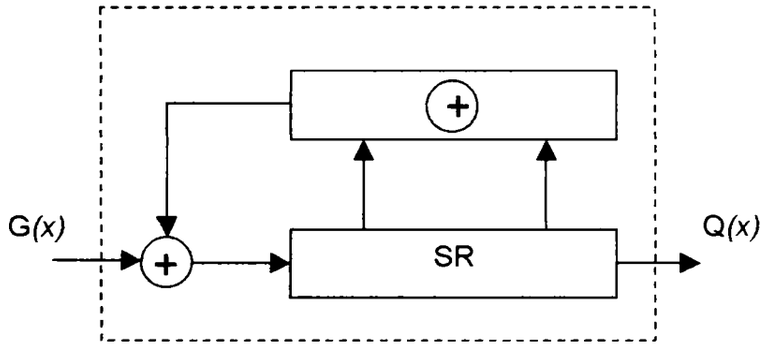
Fiecărui LFSR i se pot asocia două polinoame caracteristice. În figurile 1.12c şi 1.12d, dacă $P(x) = 1 + x + x^2 + x^3$ atunci $P^*(x) = P(x)$ şi dacă $P(x) = 1 + x^2 + x^3$ atunci $P^*(x) = 1 + x + x^3$. Pentru structura din circuitul 1.13 dacă $Q_i = x^i$ atunci P(x) se poate citi direct din figură. Dacă x^i este asociat cu Q_{n-i} şi se notează cu Q_0 intrarea la primul bistabil, atunci P*(x) se poate citi direct din figură.

Dându-se polinomul caracteristic, de exemplu Q(*x*) se pot obține două LFSR-uri diferite, funcție de notație folosită.

În mod asemănător în [AbBF–90] se arată că şi structura LFSR de tip 2 poate fi asociată cu două polinoame caracteristice, reciproce unul față de celălalt.

1.3.6.2. LFSR utilizat ca analizator de semnătură

Analiza de semnătură este o tehnică bazată pe conceptul de verificare a redundanței ciclice (cyclic redundancy checking – CRC). Cea mai simplă schemă a unui generator de semnătură este un LFSR cu o singură intrare. Semnătura este conținutul acestui registru după ce ultimul bit de intrare a fost examinat (figura 1.15).



Starea initiala: I(*x*) = 0
Starea finala: R(*x*)

$$\frac{G(x)}{P(x)} = Q(x) + \frac{R(x)}{P(x)}$$

sau $G(x) = Q(x)\,P(x) + R(x)$

Figura 1.15. LFSR de tipul 1 utilizat ca analizor de semnatură

Analiza de semnătură este una dintre cele mai cunoscute metode de testare prin compresia datelor deoarece are cel mai mic grad de mascare [Spec-91].

Utilizarea LFSR-urilor în analiza de semnătură se bazează pe conceptul de împărțire a polinoamelor, unde restul rămas în registru după terminarea procesului de test corespunde semnăturii finale. Pentru LFSR-ul din figura 1.15 secvența de intrare $\{a_m\}$ poate fi reprezentată prin polinomul G(*x*) şi secvența de ieşire prin Q(*x*). Notăm cu R(*x*) starea finală a LFSR-ului a cărui stare inițială este "*all 0s*". Se poate arăta că:

$$\frac{G(x)}{P(x)} = Q(x) + \frac{R(x)}{P(x)}$$

unde $P(x)$ este polinomul caracteristic al LFSR-ului. Deci la ieşirea LFSR-ului se obţine rezultatul $Q(x)$ şi restul $R(x)$ al împărţirii dintre şirul de la intrare şi polinomul caracteristic.

Fie $E(x)$ un polinom de eroare; de exemplu fiecare coeficient diferit de 0 reprezintă o eroare care apare pe poziţia de bit corespunzătoare.

Exemplu 1.1. Fie R_0 = 10111 răspunsul corect şi R' = 11101 răspunsul eronat. Atunci diferenţa, sau polinomul de eroare este 01010. Deci $G_0(x) = x^4 + x^2 + x + 1$, $G'(x) = x^4 + x^3 + x^2 + 1$ şi $E(x) = x^3 + x$. Mai exact, $G'(x) = G(x) + E(x)$ (modulo 2). Deoarece $G(x) = Q(x) \cdot P(x) + R(x)$, o secvenţă de răspuns nedectabilă este cea care satisface ecuaţia $G'(x) = G(x) + E(x) = Q'(x) \cdot P(x) + R(x)$; de exemplu $G'(x)$ şi $G(x)$ produc acelaşi rest.

Pe baza observaţiei de mai sus se poate enunţa următoarea teoremă:

Teorema 1.4. Fie $R(x)$ semnătura generată pentru o intrare $G(x)$ utilizând polinomul caracteristic $P(x)$ ca divizor într-un LFSR. Pentru un polinom de eroare $E(x)$, $G(x)$ şi $G'(x) = G(x) + E(x)$, au aceeaşi semnătură $R(x)$ dacă şi numai dacă $E(x)$ este un multiplu a lui $P(x)$.

Pornind de la teorema de mai sus, în [AbBF–90] se demonstrează că probabilitatea ca un generator de semnătură de n-biţi să nu detecteze o eroare este:

$$P(M) = \frac{2^{m-n} - 1}{2^m - 1}$$

care, pentru $m >> n$, este de 2^{-n}.

Teorema 1.5. Un analizator de semnătură LFSR bazat pe orice polinom cu doi sau mai mulţi coeficienţi nenuli detectează toate erorile bit-singular.

Demonstraţie: Fie $P(x)$ cu doi sau mai mulţi coeficienţi nenuli. Atunci toţi multipli nenuli ai lui $P(x)$ trebuie să aibă cel puţin doi coeficienţi nenuli. Deci, o eroare care are doar un coeficient nenul nu poate fi multiplu al lui $P(x)$ şi trebuie să fie detectabilă.

De exemplu, $P(x) = x+1$ are doi coeficienţi nenuli şi astfel detectează toate erorile singulare.

Definiţia 1.3. O eroare "burst" (k, k) este o eroare n care toţi biţii eronaţi se află pe k poziţii consecutive şi cel mult k biţi sunt eronaţi.

Teorema 1.6. Dacă $P(x)$ este de gradul n şi coeficientul lui x^0 este 1, atunci toate erorile burst (k, k) sunt detectate atâta timp cât $n \geq k$.

Rezultatele experimentale demostrează că utilizarea polinoamelor primitive determină reducerea mascării. În general, analiza de semnătură are rezultate excelente, rezultate care depind de P(x) şi se îmbunătăţesc pe măsură ce n creşte.

1.3.6.3. Multiple – Input Signature Register (MISR)

Analiza de semnătură se poate extinde şi la testarea circuitelor cu ieşiri multiple. Este de la sine înţeles că nu se poate ataşa fiecărei ieşiri câte un analizator de semnătură cu o singură intrare datorită creşterii mari a hard-ului. Cea mai utilizată tehnică este cea a utilizării MISR aşa cum se vede în figura 1.16 [HLMC–83].

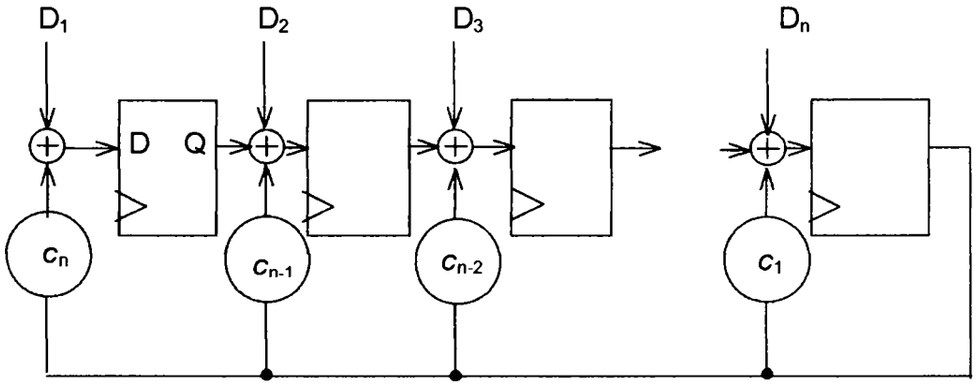


Figura 1.16. MISR

Un MISR cu n niveluri are o probabilitate de mascare de aproximativ 2^{-n} pentru modelele de eroare egale şi şiruri lungi de date. Acest rezultat este independent de P(x). Un polinom de eroare E(x) este mascat dacă el este un multiplu al lui P(x). Pentru reducerea posibilităţilor de mascare a erorilor se utilizează o structură LFSR cu un feedback complex. De exemplu, analizorul de semnătură Hewlett–Packard 5004A are polinomul caracteristic de forma $P(x) = x^{16} + x^9 + x^7 + x^4 + 1$. Deseori, P($x$) se alege astfel ca să fie un polinom primitiv.

Dacă polinomul caracteristic este produsul dintre polinomul generatorului de paritate $g(x) = x + 1$ şi un polinom primitiv de gradul $(n - 1)$, atunci un MISR cu n niveluri va avea proprietatea că paritatea tuturor biţilor din şirul de intrare este aceeaşi cu a semnăturii finale. Deci, fenomenul de mascare nu va apărea în cazul unui număr impar de erori.

O posibilitate de reducere a mascării constă în mărirea lungimii LFSR-ului. De asemenea, un test poate fi repetat utilizând polinoame de feedback diferite.

Dacă există o eroare într-un LFSR, aceasta poate fi anulată de alte erori apărute la intrări – producând astfel mascarea. Prin inspectarea conţinutului analizatorului de semnătură de mai multe ori, şansa ca un circuit defect să nu fie

detectat scade [Zimm-92]. Această tehnică este echivalentă cu examinarea periodică a ieşirii analizatorului de semnătură.

În procesul de proiectare este deseori indicat să se modifice structura registrului astfel ca el să poată opera şi ca un analizator de semnătură.

Tehnicile de compresie prezentate sunt mult utilizate deoarece ele sunt uşor de implementat [Lo-96], pot fi folosite în testare şi autotestare şi pot furniza o acoperire mare a defectelor.

Testarea prin numărarea tranzițiilor oferă o bună acoperire a defectelor cu ajutorul unor secvențe de test determinate, scurte. Această metodă se poate utiliza la monitorizarea liniilor asincrone dacă circuitul de comandă a liniei este fără curse şi fără hazard. Rezultatele depind de ordinea modelelor de test.

Numărarea de 1-uri oferă o bună acoperire a defectelor, cu toate că sunt necesare secvențe mai lungi de test. În cazul circuitelor combinaționale rezultatele sunt independente de ordinea modelelor de test.

Funcțiile booleene de testare pot fi implementate printr-un circuit care este testabil prin sindrom. Întrucât această metodă de testare necesită un set de test exhaustiv, ea nu este aplicată în cazul ciruitelor cu mai multe intrări.

În fine, analiza de semnătură este mult utilizată deoarece oferă o foarte bună acoperire a defectelor şi erorilor. Spre deosebire de alte tehnici, există mai multe posibilități de mărire a acoperirii fără schimbarea testului, ca de exemplu, prin schimbarea polinomului caracteristic sau mărirea lungimii registrului.

1.4. Built-in Self-test (BIST)

În paragrafele anterioare au fost prezentate diverse tehnici de proiectare pentru testabilitate (Design for Testability - DFT). Aceste metode sunt utilizate în special când este cerută o testare externă. BIST (Built-in Self-test) este o tehnică de proiectare în care părți ale circuitului sunt folosite pentru autotestarea acestuia. BIST este o combinație a conceptelor de built-in test (BIT) şi self-test şi a devenit un sinonim al acestor termeni.

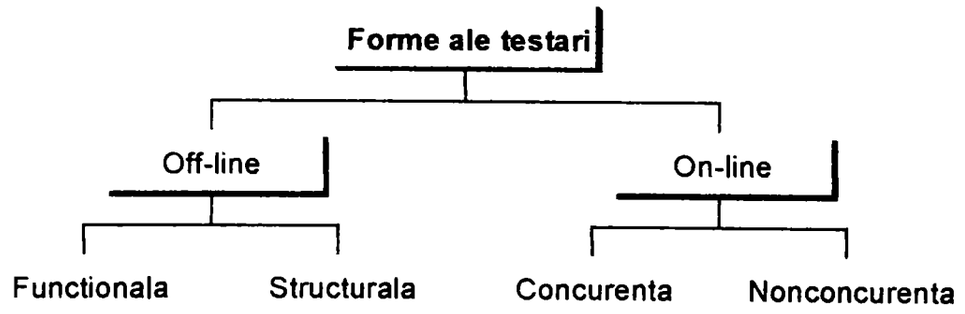


Figura 1.17. Forme ale testării

Tehnicile BIST se pot clasifica în două categorii şi anume: on-line BIST care include tehnicile concurente şi nonconcurente şi off-line BIST care include metodele funcționale, respectiv structurale (figura 1.17).

În on-line BIST, testarea se face în condiții de funcționare normală a circuitului. On-line BIST concurent este o formă de testare care apare simultan cu modul de operare funcțională normală şi foloseşte tehnici de codificare sau duplicare şi comparare. În tehnicile on-line BIST nonconcurente, testarea se face atunci când sistemul se află într-o stare de aşteptare (timp mort). Acesta este momentul când se execută rutinele soft de diagnoză sau rutinele cablate de diagnoză [McCl-85]. Procesul de testare poate fi întrerupt în orice moment.

Off-line BIST este o testare a sistemului care are loc atunci când sistemul nu se află în starea normală de funcționare. Acest mod de test este util pentru testarea sistemelor, a plăcilor şi chip-urilor. De cele mai multe ori testarea este făcută prin utilizarea de generatoare de modele de test on-chip sau on-board (TPG-uri) şi analizoare ale răspunsului de la ieşire sau rutine de microdiagnoză. Testarea off-line nu detectează erorile în timp real, de exemplu atunci când ele apar, aşa cum se întâmplă în tehnicile BIST on-line concurente.

În testarea BIST off-line funcțională se execută un test bazat pe descrierea funcțională a circuitului testat. De obicei un astfel de test este implementat ca soft de diagnoză sau cablat. În testarea BIST off-line structurală se execută un test bazat pe structura defectelor structurale. Sunt generate teste iar răspunsurile sunt comprimate cu ajutorul unor LFSR-uri.

Unele părți ale circuitului trebuie să fie operaționale pentru a executa autotestul. Această parte a circuitului este numită "hardcore" şi în configurația minimă include alimentarea, masa şi un circuit de distribuție a tactului. Hardcore este greu de testat în mod explicit. Dacă el este defect, autotestul nu se poate executa. De obicei această parte de circuit este testată cu echipamente externe de test sau se proiectează astfel încât să fie autotestabile, utilizând diverse forme de redundanță, ca de exemplu duplicarea sau self-checking checker-e.

1.4.1. Testarea exhaustivă

Testarea exhaustivă este acea metodă de testare a unui circuit combinațional cu n intrări în care se aplică toate cele 2^n intrări. În acest caz pentru generarea modelelor de test se foloseşte un numărător binar. Dacă se foloseşte un LFSR autonom cu lungime maximă, schema lui poate fi modificată pentru a include şi starea "*all 0*". Un astfel de circuit se numeşte LFSR complet şi el este descris în [McBo–81] şi [WaMc–86].

Testarea exhaustivă garantează detectarea tuturor defectelor detectabile care nu determină o comportare secvențială. Funcție de frecvența tactului, această metodă nu este aplicabilă dacă n este mare. Conceptul de testare exhaustivă nu este general aplicabil circuitelor secvențiale.

1.4.2. Testarea pseudoaleatoare

În testarea pseudoaleatoare, circuitului îi sunt aplicate modele de test care au multe caracteristici ale modelelor aleatoare, dar ele sunt generate deterministic şi deci sunt repetabile. Acest tip de testare este aplicabil atât în cazul circuitelor combinaţionale cât şi în cazul celor secvenţiale. Modelele pseudoaleatoare pot fi generate cu înlocuire sau fără înlocuire. Generarea cu înlocuire înseamnă că fiecare model se poate genera de mai multe ori, iar cea fără înlocuire presupune că fiecare model este unic şi acesta poate fi generat de un LFSR autonom. Acoperirea defectelor poate fi determinată prin simularea defectelor. Lungimea testului se alege astfel ca să se obţină un nivel acceptabil de acoperire.

În testarea pseudoaleatoare generarea modelelor de test se face cu ajutorul unui generator de test adaptiv sau cu Weighted Test Generator.

Weighted Test Generator (WTG) produce modele de test în care distribuţia de 0-uri şi 1-uri nu este neapărat uniformă. Un astfel de generator poate fi construit cu ajutorul unui LFSR autonom şi un circuit combinaţional. Atunci când se face testarea unui circuit folosind un weighted test generator este necesară o procedură de preprocesare pentru a se determina unul sau mai multe seturi de ponderi. Pentru modele pseudoaleatoare cu distribuţii diferite, unele părţi ale circuitului s-ar putea să fie testate mai efectiv faţă de altele. O dată stabilite ponderile, circuitul poate fi proiectat pentru a genera modelele pseudoaleatoare cu distribuţia dorită.

Generarea de teste adaptive utilizează un generator de modele de test ponderate. În această tehnică rezultatele simulării defectelor sunt utilizate pentru modificarea ponderilor, rezultând de aici una sau mai multe distribuţii pentru modelele de test [ChGu-96]. O dată determinate aceste distribuţii, se poate proiecta TPG-ul corespunzător. Testele generate de astfel de echipamente sunt eficiente din punct de vedere al lungimii testelor, dar hard-ul necesar poate fi complex.

1.4.3. Testarea pseudoexhaustivă

Testarea pseudoexhaustivă are avantajele testării exhaustive, dar necesită un număr mai mic de modele de test. Ea se bazează pe diferite forme ale segmentării circuitului şi încearcă să testeze fiecare segment în mod exhaustiv. Un segment este un subcircuit al circuitului C. Segmentele nu trebuie să fie disjuncte. Segmentarea poate fi logică sau fizică.

O metodă de segmentare logică este aşa numita segmentare con în care un circuit cu m ieşiri se segmentează logic în m conuri, fiecare con conţinând întreaga logică asociată unei ieşiri. Fiecare con este testat exhaustiv şi toate conurile sunt testate concurent. Această formă de testare a fost sugerată de McCluskey şi se numeşte testare prin verificare. O altă metodă de segmentare logică folosită, bazată pe partiţionarea circuitului, este cea numită segmentarea prin sensibilizarea căilor.

Pentru circuitele mari, tehnicile de testare pseudoexhaustivă necesită adeseori un număr mare de seturi de test. În aceste cazuri se foloseşte segmentarea fizică. Circuitul se împarte în subcircuite utilizând tehnici de segmentare hardware.

Când se face o testare pseudoexhaustivă a unui circuit cu n intrări, este posibilă o reconfigurare a liniilor de intrare astfel ca testele necesare să fie generate pe m linii, unde $m < n$. Aceste m semnale se numesc semnale de test.

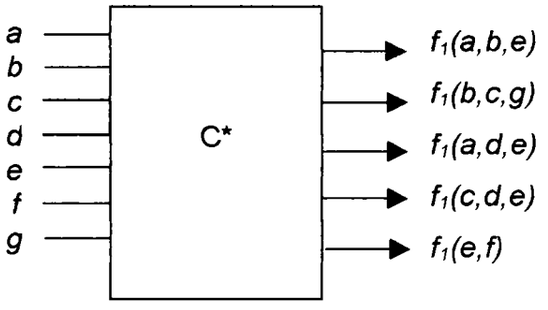


*Figura 1.18. Circuitul C**

Fie circuitul din figura 1.18. Determinarea unui set minim de semnale de test se face urmărind următoarea procedură:

Pasul 1. Se partiționează circuitul în circuite disjuncte.
C* are doar o partiție.

Pasul 2. Pentru fiecare subcircuit disjunct se parcurg următorii paşi.
a. se generează matricea de dependență
b. se partiționează matricea în grupe de intrări astfel ca două sau mai multe intrări dintr-un grup să nu afecteze aceaşi ieşire
c. se condensează fiecare grup pentru a forma o intrare echivalentă, numită semnal de test de intrare.

Pentru un circuit cu n intrări şi m ieşiri, matricea de dependență $D = [d_{i,j}]$ are m rânduri şi n coloane, unde $d_{i,j} = 1$ dacă ieşirea i depinde de intrarea j; în caz contrar $d_{i,j} = 0$. Pentru circuitul C*, D este:

$$D = \begin{array}{c} \begin{matrix} a & b & c & d & e & f & g \end{matrix} \\ \begin{bmatrix} 1 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 \end{bmatrix} \end{array} \begin{matrix} f_1 \\ f_2 \\ f_3 \\ f_4 \\ f_5 \end{matrix}$$

Rearanjând şi grupând intrările se obține matricea D_g.

$$D_g = \begin{array}{c} \begin{array}{cc|cc|c|cc} \multicolumn{2}{c}{\text{I}} & \multicolumn{2}{c}{\text{II}} & \text{III} & \multicolumn{2}{c}{\text{IV}} \\ a & c & b & d & e & f & g \end{array} \\ \left[\begin{array}{cc|cc|c|cc} 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 \end{array}\right] \end{array} \begin{array}{c} \\ f_1 \\ f_2 \\ f_3 \\ f_4 \\ f_5 \end{array}$$

Fiecare grup trebuie să conțină mai puțin de două 1-uri în fiecare rând şi numărul de grupuri trebuie să fie cât mai minim. Acest lucru asigură ca nici o ieşire să nu fie comandată de mai mult de o intrare din fiecare grup.

Matricea condensată D_c se obține făcând operația SAU între coloanele din grup pentru a forma o singură coloană. Astfel D_c corespunzătoare circuitului C* este:

$$D_c = \begin{bmatrix} 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \end{bmatrix}$$

Pasul 3. Se notează cu *p* numărul de partiții din D_c, care se numeşte lățimea lui D_c şi reprezintă numărul maxim de semnale de intrare necesare testării unui subcircuit disjunct, şi cu *w* numărul maxim de 1-uri dintr-un rând, unde *w* se numeşte ponderea lui D_c şi reprezintă numărul maxim de semnale de care depinde ieşirea.
Pentru D_c avem $p = 4$ şi $w = 3$.

Pasul 4. Se construiesc modelele de test pentru circuit pentru următoarele trei cazuri:

Cazul I: $p = w$. Testul constă din toate cele 2^p modele de test de *p* biți. Testul poate fi uşor generat de un numărător sau un LFSR complet. Mai exact, acesta un set de test pseudoexhaustiv universal minim.

Cazul II: $p = w + 1$. Acesta este cazul unde $n = k + 1$. Setul de test constă din toate modelele posibile de *p* biți cu paritate fie pară fie impară. Există $2^{p-1} = 2^w$ astfel de modele. Pentru paritate impară, testele pentru C* sunt:

$$\left.\begin{matrix}0&0&0&1\\0&0&1&0\\0&1&0&0\\1&0&0&0\end{matrix}\right\} \text{paritate} = 1$$

$$\left.\begin{matrix}0&1&1&1\\1&0&1&1\\1&1&0&1\\1&1&1&0\end{matrix}\right\} \text{paritate} = 3$$

Cazul III: $p > w + 1$. În acest caz testul constă din din două sau mai multe subseturi de modele, fiecare conținând toate toate modelele posibile de p biți cu o anumită pondere a constantei.

Pentru generarea modelelor de test necesare unei testări pseudoexhaustive se pot folosi următoarele generatoare de modele de test:

- *syndrome-driver counter* (SDC), care poate fi sau un numărător binar sau un LFSR care conține doar *p* celule de memorie,
- *LFSR/SR combinat*. Costurile necesare în această metodă sunt mai mici decât în cazul unui constant-weight counter dar circuitul care rezultă va genera un număr mai mare de vectori de test. Numărul de modele de test generate este minim când *w* este mult mai mic decât *n*. În general LFSR-urile sunt proiectate astfel încât să aibă un mod de operare de deplasare [LeSh-96]. Acesta este utilizat pentru încărcarea valorilor inițiale în registru. Dacă registrul este utilizat şi ca analizor de semnătură, modul de deplasare este utilizat şi pentru scanarea semnăturii finale.
- *LFSR/porți SAU-EXCLUSIV combinate*. Proiectarea acestor rețele se bazează pe utilizarea sumelor liniare sau a codurilor liniare. Aceste scheme necesită cel mult două aproximări inițiale şi numărul modelelor de test necesare pentru asigurarea unei testări exhaustive este apropiat de cel necesar în schemele LFSR/SR.
- *LFSR-uri condensate.* Această schemă propusă de Wang şi McCluskey utilizează cel mult două aproximări inițiale, este simplă şi produce un set de test eficient dacă $w \geq n/2$. Dacă $w < n/2$ această tehnică utilizează mai multe teste decât metoda LFSR/SR. LFSR-urile condensate se bazează pe conceptul codurilor liniare.
- *LFSR-uri ciclice.* Pentru $w < n/2$ LFSR-urile ciclice determină teste eficiente şi necesită puțin hard suplimentar. LFSR-urile ciclice se bazează pe codurile ciclice, care reprezintă o subclasă a codurilor liniare.

1.5. Arhitecturi BIST off-line generice

În acest paragraf vor fi prezentate câteva structuri BIST off-line generale care pot fi aplicate chip-urilor şi plăcilor formate din blocuri de logică combinaţională interconectate prin celule de memorie.

Arhitecturile BIST off-line la nivel de chip sau placă pot fi clasificate după următoarele criterii: circuite BIST centralizate sau distribuite şi elemente BIST incluse sau separate.

Arhitecturile BIST conţin următoarele elemente cheie: generatoare de modele de test, analizoare ale răspunsului de la ieşire, circuitul testat, un sistem de distribuţie (DIST) pentru transmiterea datelor de la TPG-uri (test pattern generator) la circuitul testat şi apoi de la circuitul testat la analizorul răspunsului de ieşire, şi în fine un controller BIST pentru controlul circuitului BIST şi a celui testat în timpul autotestului. De multe ori părţi sau tot controller-ul sunt off-chip. Sistemul de distribuţie (DIST) constă în primul rând din interconexiuni directe (legături), buss-uri, multiplexoare şi căi de scanare.

O arhitectură BIST centralizată este de forma celei din figura 1.19. Circuitul testat (CUT – circuit under test), generatorul de modele de test (TPG) şi analizorul răspunsului de ieşire au circuite comune. Acest lucru determină scăderea numărului de componente dar măreşte timpul de test.

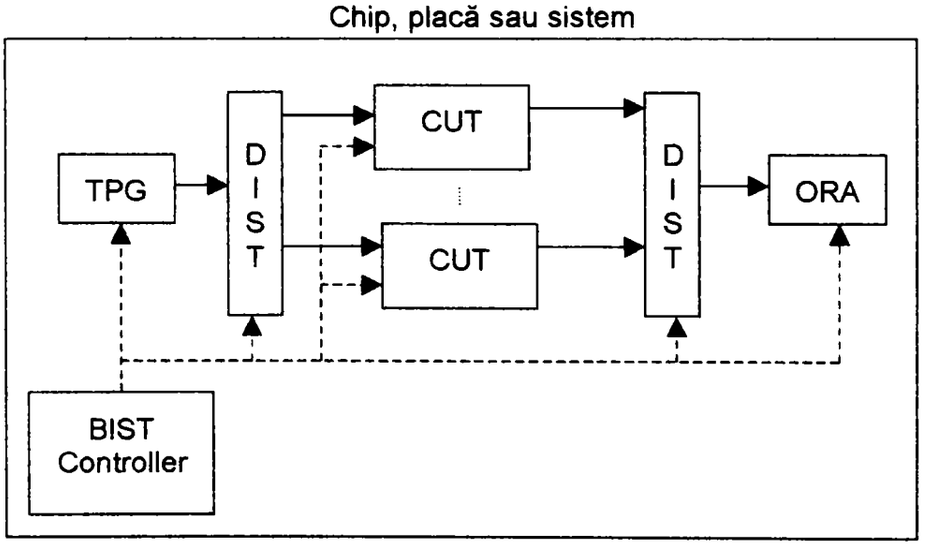


Figura 1.19. Forma generică a arhitecturii BIST centralizată şi separată

În timpul testării, controller-ul BIST poate îndeplini una sau mai multe dintre următoarele funcţii: parcurge câte un pas din secvenţa de test a circuitelor aflate sub test; inhibă tactele sistemului şi controlează tactul de test, comunică cu celelalte controller-e de test utilizând buss-urile de test sau controlează autotestul.

În figura 1.20 este prezentată o arhitectură BIST distribuită. Aici fiecare CUT are propriul circuit TPG şi ORA (Output Respons Analyzer). O astfel de arhitectură necesită un număr mai mare de componente, dar timpul de test scade şi diagnoza este mai bună. În figură nu este menţionat circuitul de control BIST. Schemele din figurile 1.19 şi 1.20 sunt exemple de arhitecturi BIST separate în care circuitele TPG şi ORA sunt externe circuitului testat şi nu constituie parte a circuitului funcţional.

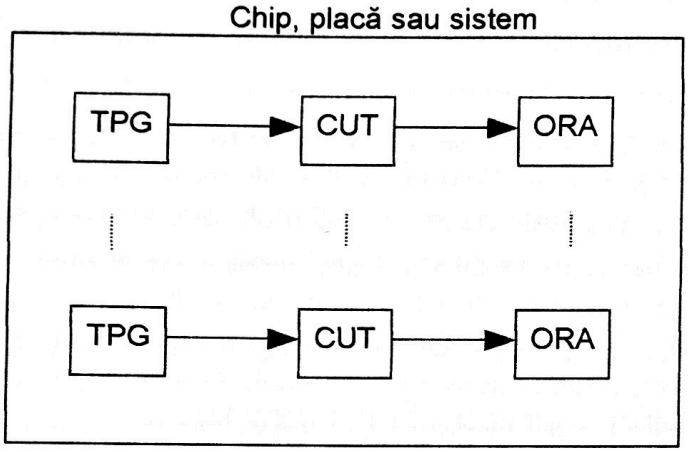


Figura 1.20. Forma generică a arhitecturii BIST distribuite şi separate

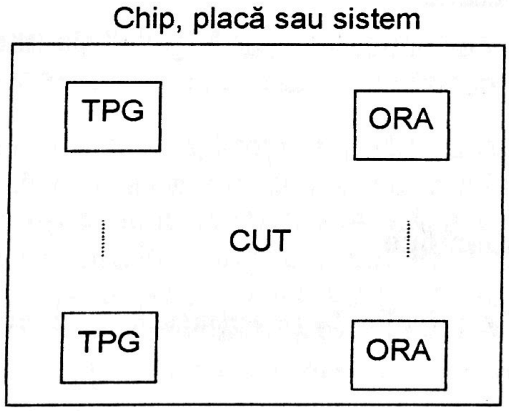


Figura 1.21. Forma generică a arhitecturii BIST distribuite şi embedded

În figura 1.21 este prezentată forma generală a unei arhitecturi BIST distribuite şi incluse. Aici elementele TPG şi ORA sunt configurate din elemente funcţionale ale circuitului testat, de exemplu registru. În acest caz schema de control este complexă, dar cu mai puţin hard decât în arhitecturile distribuite şi separate.

Alegerea arhitecturii BIST se face în funcție de următorii factori:

1. gradul de paralelism: arhitectura BIST distribuită oferă un grad mai mare al paralelismului testului deoarece la un moment dat sunt testate mai multe CUT.
2. acoperirea defectelor: arhitectura distribuită determină o mai mare acoperire a defectelor deoarece circuitele TPG şi ORA pot fi adaptate fiecărui CUT. De exemplu, o tehnică BIST pentru un bloc de logică combinațională nu este adecvată şi unui RAM.
3. nivelul de împachetare: la nivel înalt se recomandă BIST centralizat.
4. timpul de test: arhitecturile BIST distribuite reduc timpul de testare.
5. restricțiile fizice: mărime, greutate, putere, costurile de ventilație (răcire) şi alți factori care influențează proiectarea. Deseori arhitecturile BIST incluse şi separate necesită mai mult hard şi scad performanțele.
6. complexitatea unităților înlocuite: dacă o placă este o unitate ce poate fi înlocuită şi ea trebuie să fie autotestabilă, atunci ea trebuie să conțină circuite TPG şi ORA. Dacă un sistem este unitatea înlocuită la cel mai mic nivel, atunci placa lui nu trebuie să aibă circuite TPG şi ORA şi se poate utiliza o arhitectură BIST centralizată.
7. factory and field test-and-repair strategy: tipul ATE şi gradul de folosire a lui pentru testarea şi diagnoza defectelor influențează şi este influențat de BIST. De exemplu, datorită utilizării mărite a echipamentelor montate pe suprafață, testarea "in-circuit" devine mai dificilă de utilizat.
8. degradarea performanțelor: adăugarea de hard BIST pe căile critice de timp a unui circuit necesită o reducere a frecvenței de tact a sistemului.

1.6. Arhitecturi BIST specifice

1.6.1. Arhitectură BIST centralizată şi separată la nivel de placă

În figura 1.22 este prezantată o arhitectură BIST centralizată şi separată la nivel de placă (CSNP), propusă de Benowitz, cu următoarele caracteristici: arhitectură BIST centralizată şi separată, fără boundary scan şi circuit testat de tip combinațional sau secvențial.

Pe parcursul modului de test off-line, intrările sunt comandate de un generator de modele pseudoaleatoare (PRPG), iar ieşirile sunt monitorizate folosind un analizator de semnătură cu o singură intrare. Pentru a se reduce costurile hard, testul se repetă de *m* ori, o dată pentru fiecare ieşire.

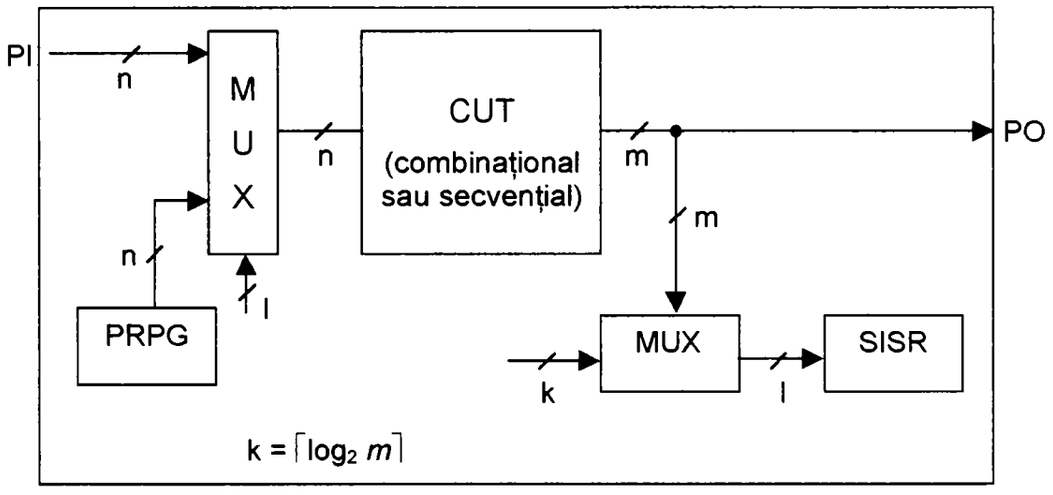


Figura 1.22. Arhitectură BIST centralizată şi separată (CSBL)

Această metodă este adecvată în circuitele cu feedback limitat. Controller-ele cu stări finite generale sunt mai greu de testat. Microprocesoarele reprezintă un caz de testare foarte complex. Pentru determinarea numărului de vectori de test necesari pentru atingerea unui nivel adecvat de acoperire al defectelor se necesită o simulare extensivă a defectelor.

1.6.2. Autotest şi evaluare built-in (BEST)

O astfel de arhitectură este o aplicație a CSNP şi este dată în figura 1.23. În general logica ce se testează este un circuit secvențial. Intrările la CUT sunt comandate de PRPG, iar ieşirile sunt comprimate utilizând un MISR. Există ambele versiuni ale acestei arhitecturi, cea inclusă şi cea separată. De exemplu, dacă intrările primare ale CUT merg direct la registre şi ieşirile primare sunt comandate de registre atunci se utilizează o arhitectură inclusă sau incorporată. În caz contrar este necesară adăugarea de PRPG şi MISR şi va rezulta o arhitectură separată.

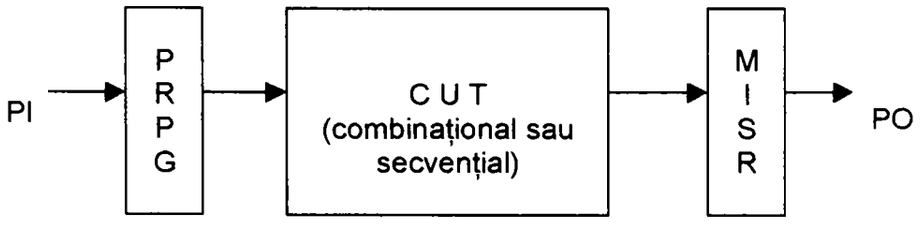


Figura 1.23. Arhitectură BIST cu evaluare built-in şi autotest (BEST)

Hard-ul creşte puţin în cazul arhitecturii BEST, dar este necesară o simulare extensivă a defectelor pentru a se obţine un echilibru între acoperirea defectelor şi lungimea testului.

Arhitecturile BIST prezentate adeseori oferă o mică acoperire a defectelor deoarece ele se bazează pe utilizarea modelelor pseudoaleatoare în testarea circuitelor secvenţiale. Pentru evitarea acestei probleme se poate utiliza o cale internă de scan în CUT, astfel ca testarea să poată fi redusă la problema testării logicii combinaţionale.

1.6.3. Random–Test Socket (RTS)

Random–Test Socket este o arhitectură BIST adevărată deoarece circuitul de test este extern celui care se testează. În figura 1.24 este prezentată o arhitectură RTS.

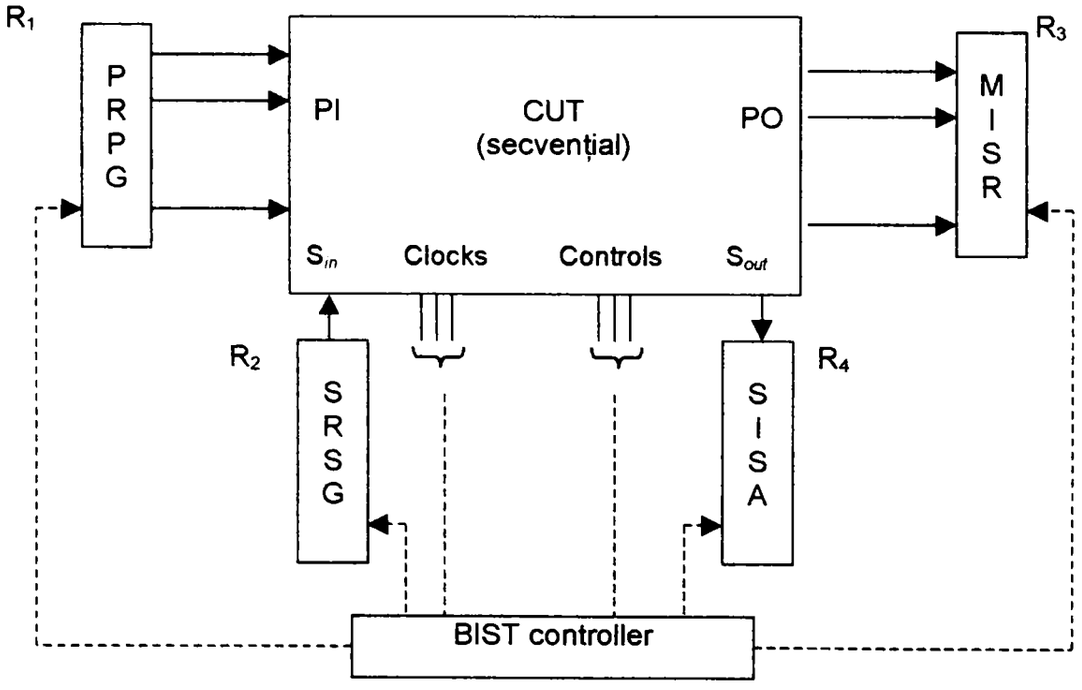


Figura 1.24. Random-test socket (RTS)

Testarea CUT se face astfel:

1. se iniţializează LFSR-ul
2. se încarcă (local) modelul testului pseudoaleator în calea de scan prin R_2
3. se generează un nou vector de test pseudoaleator utilizând R_1. Acest lucru durează un ciclu de tact. Vectorul este aplicat intrărilor primare ale CUT.

4. se culege răspunsul la ieşirile primare ale CUT prin aplicarea unui impuls de tact lui R_3
5. se execută o operație de încărcare paralelă a celulelor de memorie a sistemului pentru a se obține răspunsul la modelul de test aleator
6. se scanează data de pe calea internă de scan a CUT şi se comprimă această dată în R_4

Paşii 2-6 se repetă până la obținerea unei acoperiri adecvate a defectului sau până ce este atins timpul maxim admis de test. Circuitul se consideră a fi fără defecte dacă valorile finale din R_3 şi R_4 sunt corecte.

Metoda RTS are dezavantajul pe de o parte că testarea este lentă şi apoi că testarea cu modele de test pseudoaleatoare necesită mai multe modele de test decât în cazul testării cu modele de test deterministe.

1.6.4. Autotestarea utilizând MISR şi SRSG paralel (STUMPS)

Un SRSG este echivalent cu un PRPG. O arhitectură STUMPS este prezentată în figura 1.25 şi are următoarele atribute: arhitectură BIST centralizată şi

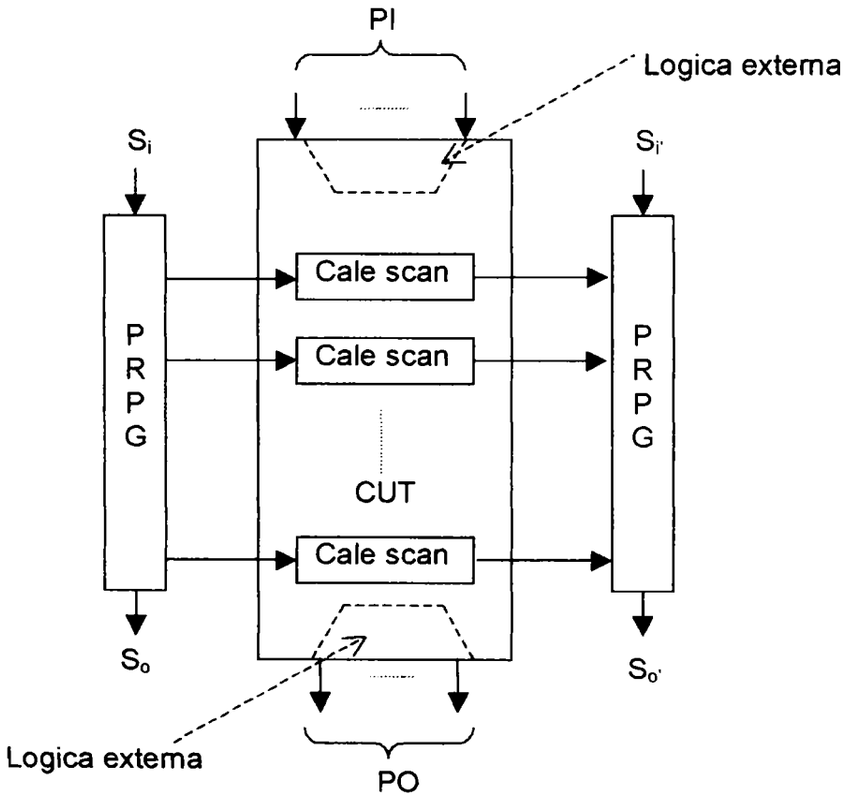


Figura 1.25. Autotest utilizând MISR şi SRSG paralel (STUMPS)

separată, căi multiple de scan şi fără boundary-scan. Căile de scanare sunt comandate în paralel de un PRPG şi de asemenea şi semnătura este generată în paralel din fiecare cale de scan, cu ajutorul unui MISR. La nivel de placă, fiecare cale de scan corespunde cu o cale de scan din câte un chip.

Utilizarea căilor multiple de scan determină o reducere semnificativă a timpului de test. Având în vedere că lungimile căilor de scan pot fi diferite, PRPG va rula timp de *k* cicluri de tact pentru a încărca aceste căi, unde *k* este cea mai mare lungime a căii de scan.

1.6.5. Arhitectură BIST concurentă (CBIST)

Saluja în [SaSK-88] a extins conceptul de BIST off-line pentru a include BIST on-line. O versiune de arhitectură CBIST este dată în figura 1.26.

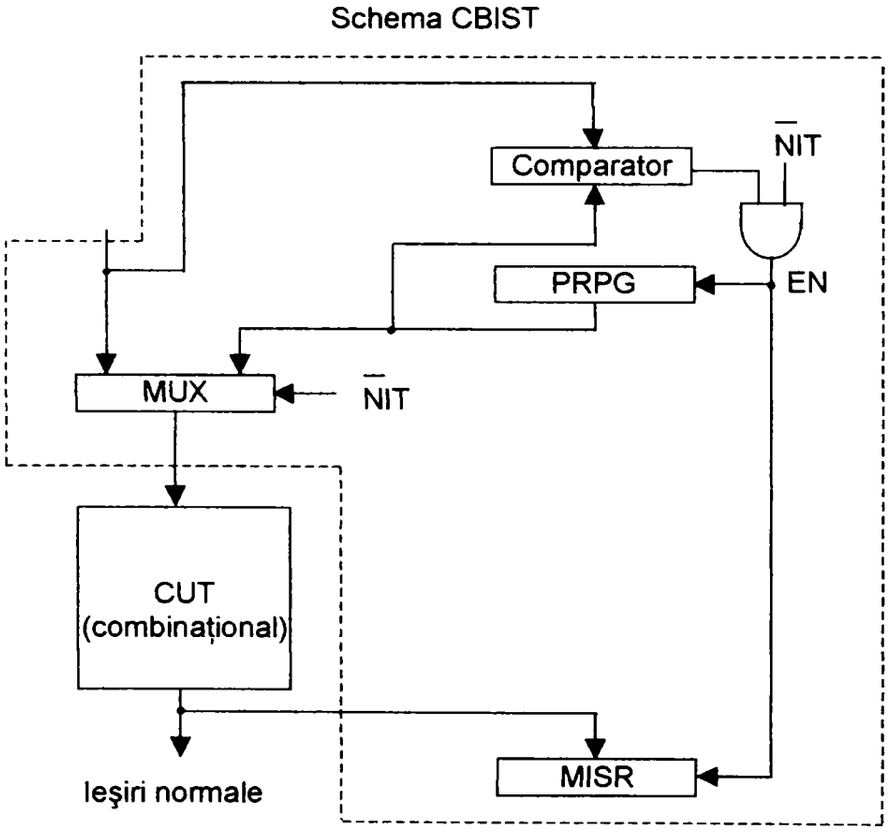


Figura 1.26. BIST concurent pentru logică combinațională

Circuitul care se testează trebuie să fie un circuit combinațional. Hardul BIST este separat de cel normal. Tehnica poate fi aplicată în circuitele secvențiale dacă se

face o partiționare în blocuri de logică combinațională care pot fi testate ca entități separate. Nu este necesară utilizarea de căi de scan interne sau periferice. Pentru simplificarea hard-ului se recomandă arhitectura BIST centralizată. În timpul testării off-line, PRPG-ul comandă CUT iar răspunsul acestuia este comprimat în MISR. Pentru testarea on-line, PRPG şi MISR sunt mai întâi inițializate şi menținute în starea inițială până ce sunt validate de semnalul EN. Circuitului testat i se aplică intrările normale. Când acestea corespund cu starea PRPG-ului, se generează un semnal de validare care permite trecerea PRPG-ului în starea următoare, în timp ce MISR eşantionează ieşirea şi trece de asemenea în starea sa următoare. Acest proces se repetă ori de câte ori apare o potrivire între datele de intrare normală şi starea curentă a PRPG-ului. Când PRPG-ul atinge o anumită stare finală prespecificată se verifică semnătura din MISR.

1.6.6. Arhitectură BIST centralizată şi inclusă cu Boundary Scan (CEBS)

În figura 1.27 este prezentată o arhitectură CEBS. Primii *r* biți ai registrului boundary scan de intrare, notat RPG, funcționează ca un PRPG şi SRSG, iar ultimii *s* biți ai registrului boundary scan de ieşire acționează atât ca un MISR cât şi ca un SISR. Testarea se face după cum urmează: se setează un semnal al modului test şi apoi RPG va încărca calea de scan cu date de test pseudoaleator. Registrele căilor de scan sunt încărcate în paralel cu datele sistemului, cu excepția registrului de semnătură (SR), care operează în mod MISR. Calea de scan este din nou încărcată cu date pseudoaleatoare în timp ce registrul de semnătură operează în mod SISR, comprimând datele aflate în calea de scan.

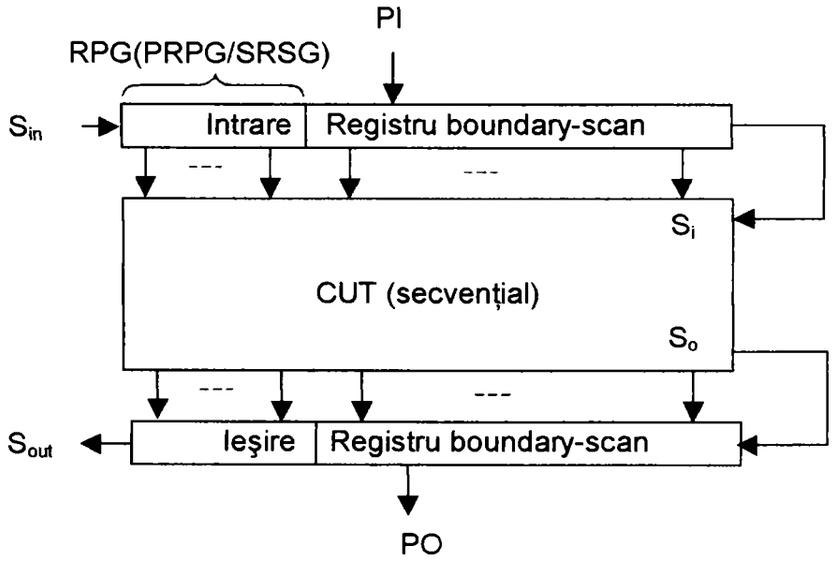


Fig. 1.27. Arhitectură centralizată şi inclusă, cu boundary-scan (CEBS)

1.6.7. Random Test Data (RTD)

O problemă majoră a majorității tehnicilor prezentate anterior este aceea că aplicarea unui singur model de test circuitului testat necesită ca întreaga cale de scan să fie încărcată cu noua dată. Această problemă poate fi eliminată prin utilizarea arhitecturii BIST RTD, prezentată în figura 1.28, şi care are următoarele proprietăți: arhitectură BIST distribuită şi inclusă precum şi boundary scan.

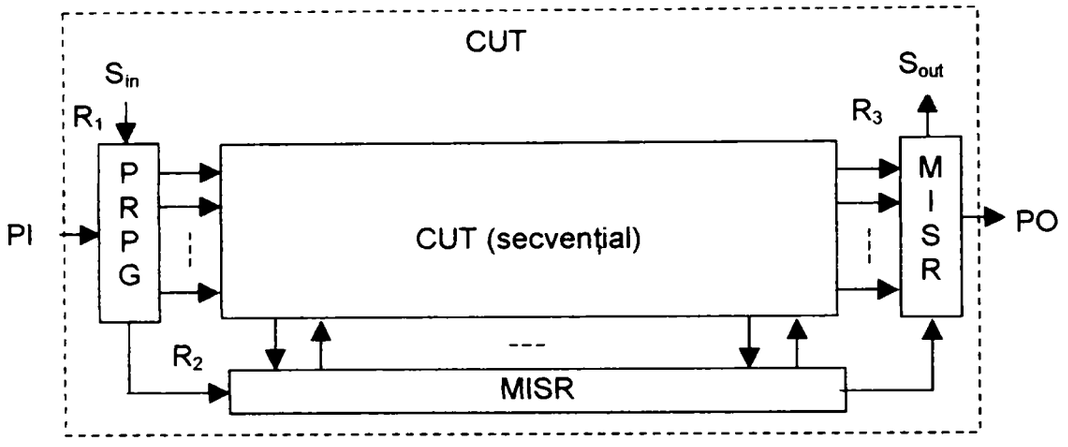


Figura 1.28. Arhitectură BIST RTD

Într-o astfel de arhitectură analizoarele de semnătură sunt utilizate atât pentru compresie cât şi pentru generarea modelelor de test.

Registrele R_1, R_2 şi R_3 sunt setate mai întâi în modul scan, astfel că ele pot fi încărcate cu modele de aproximare inițiale. R_1 nu poate fi încărcat cu modele "*all 0s*". Registrele sunt apoi aduse în modul de test şi menținute astfel atâta timp cât circuitul este testat. Pentru fiecare ciclu de tact, R_1 şi R_2 generează un nou model de test iar R_2 şi R_3 operează ca un MISR.

Despre testele generate de R_2 se pot spune puține lucruri întrucât modelele sunt starea MISR-ului, mai exact modelele generate sunt funcții ale logicii C care se testează. O problemă a acestei tehnici constă în faptul că unele modele binare se pot repeta iar altele nu vor fi niciodată generate. Pentru determinarea numărului de cicluri de tact necesare operării în modul test, pentru a se atinge un anumit nivel de acoperire a defectelor este necesară o simulare a defectelor.

Majoritatea metodelor BIST prezentate utilizează diverse forme de LFSR-uri pentru generarea modelelor de test şi a semnăturii. În tehnica RTD se presupune că data normală din circuit, dacă este combinată cu unele forme ale mecanismului de captare a erorii, este suficientă pentru testarea unui circuit. Acest concept stă la baza următoarelor trei arhitecturi BIST.

1.6.8. Autotestul simultan (SST)

Autotestul simultan (Simultaneous Self-Test sau SST) este o metodă BIST cu următoarele atribute: arhitectură BIST distribuită şi incorporată, arhitectură scan a circuitului testat, fără boundary scan şi nu utilizează LFSR-uri.

În această metodă fiecare celulă de memorie a circuitului testat este modificată pentru a fi o celulă de memorie autotestabilă. O astfel de celulă de memorie are trei moduri de operare, şi anume modul normal, modul scan şi modul de test. În figura 1.29 este prezentată o celulă de memorie SRL. În modul normal de operare valoarea semnalului de date D a sistemului este încărcat în bistabil de către CK1. Pentru modul scan, T este setat la 0 şi se utilizează tactul CK2. În timpul autotestului, T este setat la 1 şi se utilizează CK2. Noua stare a bistabilului este $D \oplus S_i$. În figura 1.29 este prezentată o parte de circuit cu autotest simultan.

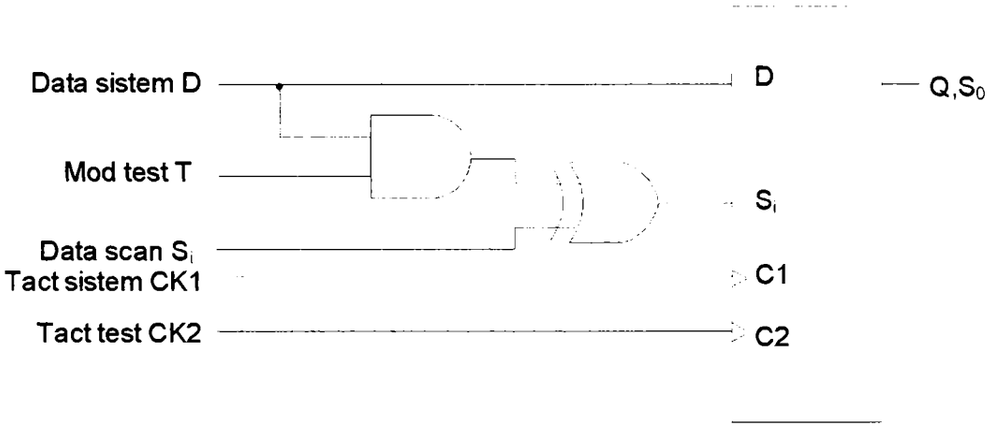


Figura 1.29. Celulă de memorie autotestabilă

Pentru testarea unei scheme SST se încarcă mai întâi calea de scan cu data inițială, apoi circuitul este adus în modul test şi sincronizat cu un număr suficient de perioade (N) pentru a fi testat în mod adecvat. După cele N cicluri de tact, circuitul este pus înapoi în modul scan iar conținutul căii de scan este trimis la ieşire. Dacă în timpul procesului de test apare o eroare la ieşire sau dacă conținutul final al căii de scan este incorectă, este detectată o eroare. În timpul procesului de test calea scan de autotest colectează simultan rezultatele testului de la C şi furnizează valorile de test în C. Astfel procesul de testare este rapid întrucât data de test nu este nevoie să fie deplasată în calea de test.

Există totuşi o serie de probleme legate de această metodă. Una dintre acestea este legată de testarea logicii externe iar cealaltă problemă este cea a calității procesului de testare. Dacă se generează o eroare la ieşirea lui C (figura 1.30) atunci ea va apărea în calea de test, de exemplu în celula Q_j. Pe de altă parte,

în timpul următorului ciclu de tact această eroare se poate propaga prin C şi poate produce alte erori în calea de test, Această eroare inițială este posibil să se propage la următoarea celulă Q_{j+1} din calea de scan. Dacă intrarea D_{j+1} la această celulă este de asemenea în eroare în timpul acestui ciclu de tact, atunci cele două erori se pot anula şi apare mascarea.

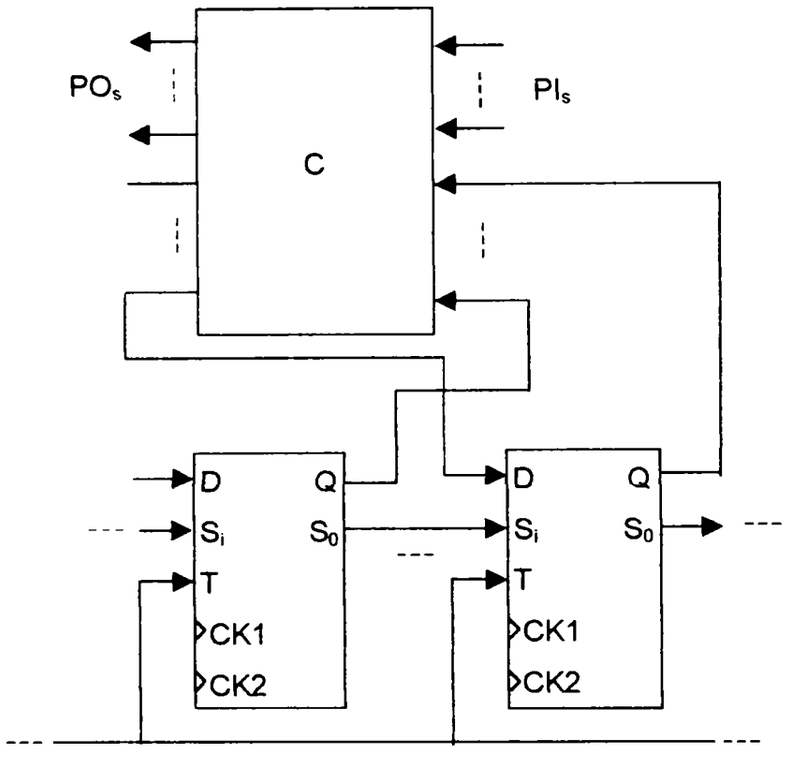


Figura 1.30. Arhitectură BIST cu celule de memorie autotestabile

1.6.9. Sistem de testare prin analiză ciclică (CATS)

În figura 1.31 este prezentată o arhitectură BIST numită cyclic analysis testing systems (CATS). Aspectele legate de testare ale acestei arhitecturi se bazează pe faptul că circuitele secvențiale, prin natura lor, sunt generatoare de secvențe binare neliniare.

În modul test, ieşirea lui S comandă intrările. Dacă există mai multe ieşiri decât intrări, atunci ieşirile suplimentare se pot combina utilizând părți SAU-EXCLUSIV. Dacă numărul de ieşiri este mai mic decât cel al intrărilor, atunci unele ieşiri vor comanda mai multe intrări. Analiza acestui tip de arhitectură BIST este complexă. Diferitele permutări ale interconexiunilor dintre ieşiri şi intrări determină diverse eficiențe ale testelor.

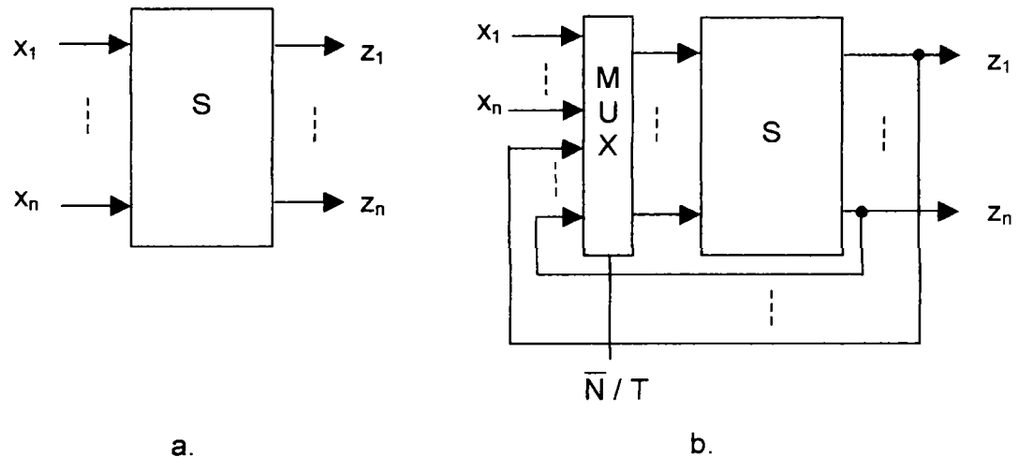


Figura 1.31. Arhitectură BIST CATS

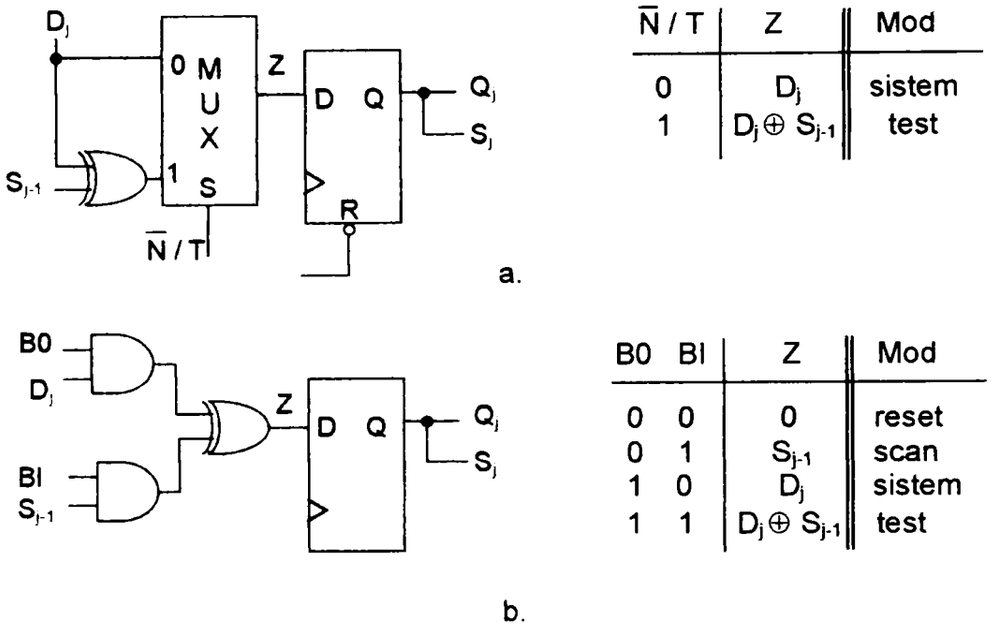
Pentru testarea CUT, ieşirile sunt conectate la intrări şi circuitului i se aplică un număr predeterminat de cicluri de tact. Pentru circuite mari aceasta poate însemna zeci de mii de paşi. Acest număr este funcție acoperirea dorită a defectelor şi trebuie determinat prin simularea defectelor. Acest tip de arhitectură necesită puțin hard suplimentar şi eficiența lui depinde de circuit.

1.6.10. Cale de autotest circulară (CSTP)

Arhitectura BIST de tip "circular self-test path" (CSTP) este similară cu schema SST. Există doar trei deosebiri: are o schemă de autotestare a celulei ca cea din figura 1.32a, este o arhitectură care are la bază registre – de exemplu celulele autotestabile sunt grupate în registre – şi necesită un autotest parțial – de exemplu nu toate registrele trebuie să fie formate din celule autotestabile. Această arhitectură necesită îndeplinirea a două condiții: 1) toate intrările şi ieşirile trebuiesc asociate celulelor boundary scan şi 2) toate celulele de memorie trebuie să fie inițializate la o stare cunoscută înainte de testare.

Fie structura din figura 1.33. Registrele R_1, R_2, R_3, R_7 şi R_8 sunt parte a căii circulare de autotest. Dacă această cale are *m* celule, atunci ea corespunde unui MISR cu polinomul caracteristic $(1 + x^m)$. Registrele R_4, R_5 şi R_6 nu trebuie să facă parte din calea de autotest şi nici nu necesită linii de set sau reset deoarece ele pot fi inițializate pe baza stării restului circuitului. Adică, odată ce R_1, R_2 şi R_3 sunt inițializate, dacă R_4, R_5 şi R_6 au două tacte de sistem, atunci şi ele vor fi aduse la o stare cunoscută.

Nu se poate afirma acelaşi lucru despre R_3 datorită buclei de feedback formată de calea R_3-C_3-R_6-C_6-R_3. În orice caz dacă R_3 are o linie de reset, atunci acesta nu trebuie să facă parte din calea de autotest. Dacă numărul de celule din calea de autotest creşte atunci atât hard-ul se măreşte cât şi acoperirea defectelor pentru o anumită lungime a testului.



N̄ / T	Z	Mod
0	D_j	sistem
1	$D_j \oplus S_{j-1}$	test

B0	BI	Z	Mod
0	0	0	reset
0	1	S_{j-1}	scan
1	0	D_j	sistem
1	1	$D_j \oplus S_{j-1}$	test

Figura 1.32. Scheme de celule de memorie utilizate pe căile autotestabile din arhitecturile BIST

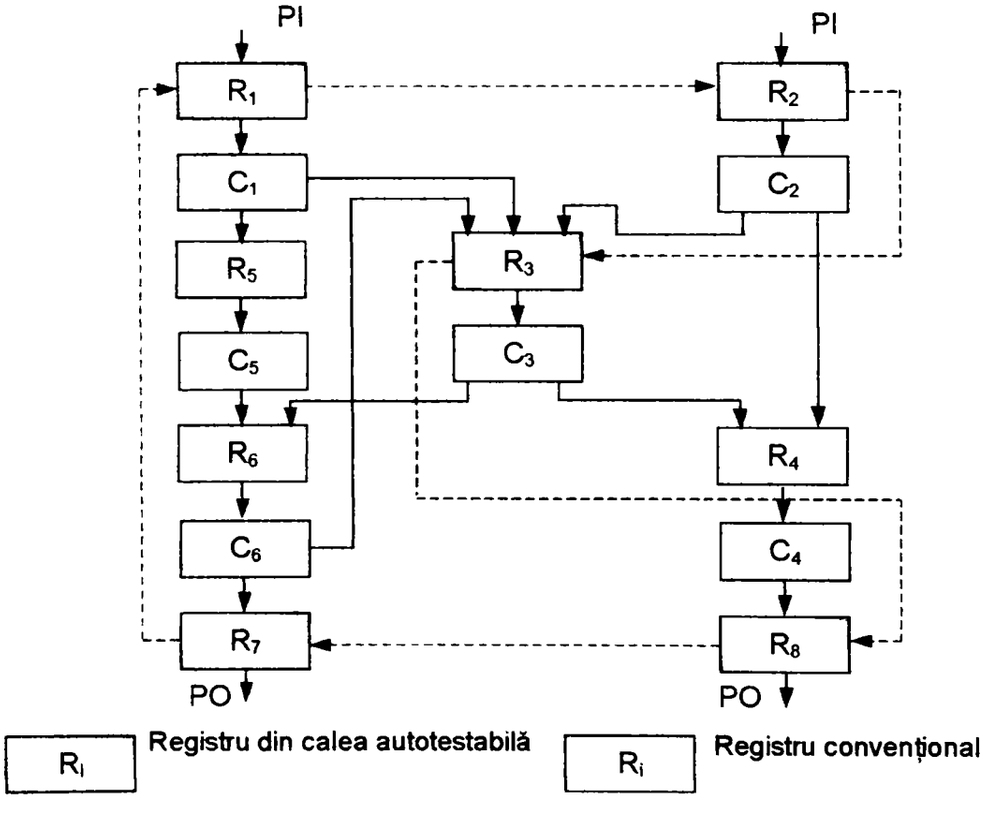


Figura 1.33. Schemă care utilizează o arhitectură CSTP

Procesul de test se desfăşoară în trei etape:

1. Inițializare: toate registrele sunt plasate într-o stare cunoscută
2. Testarea circuitului: circuitul funcționează în modul test; registrele care nu fac parte din calea de test operează în modul lor normal
3. Evaluarea răspunsului: în etapa a doua, calea de autotest operează atât ca generator de modele aleatoare cât şi ca un compresor al răspunsului. În etapa a treia circuitul funcționează din nou în modul test. Acum secvențele de ieşire din una sau mai multe celule de autotest se compară cu valorile corecte precalculate. Comparația se poate face fie on-chip fie off-chip.

O metodă BIST asemănătoare a fost prezentată şi de Stroud, unde analizatorul de semnătură este incorporat în schema BIST pentru a simplifica identificarea operării defectuase a circuitului şi se utilizează o celulă autotestabilă ca cea din figura 1.32b.

1.6.11. Built–in Logic Bloc Observation (BILBO)

Una dintre problemele majore ale schemelor prezentate anterior este aceea că ele analizează versiuni nepartiționate ale circuitelor testate; de exemplu toate intrările primare sunt grupate într-un set, toate ieşirile primare sunt într-un al doilea set şi toate celulele de memorie formează un al treilea set. Aceste seturi sunt asociate cu PRPG-uri şi MISR-uri. Întrucât numărul de celule din aceste registre este de obicei mare, nu sunt recomandate tehnicile de testare exhaustive sau pseudoexhaustive. De exemplu, un chip poate avea peste 100 de intrări şi câteva sute de celule de memorie. Pentru evitarea acestor probleme celulele de memorie sunt grupate în registre. În general aceste grupe corespund registrelor funcționale din diverse scheme, de exemplu, numărătoare de program, registre de instrucțiuni şi acumulatoare – ca în schema unui microprocesor. Unele arhitecturi BIST preiau facilități ale registrelor din multe scheme pentru a obține o metodologie de test mai bună.

În registrele de tip BILBO ieşirea negată $\overline{Q}$ a celulei de memorie este conectată printr-o poartă SAU – NU şi o poartă SAU – EXCLUSIV la intrarea de date a următoarei celule. Un registru BILBO operează în unul din cele patru moduri specificate de intrările B_1 şi B_2. Pentru $B_1 = B_2 = 1$, registrul BILBO operează în modul de încărcare paralel (figura 1.34b). Dacă $B_1 = B_2 = 0$, el operează ca un registru de deplasare cu intrarea de scan S_i (figura 1.34c). Data este complementată când intră în registrul scan. Dacă $B_1 = 0$ şi $B_2 = 1$ toate celulele de memorie sunt resetate. Pentru $B_1 = 1$ şi $B_2 = 0$, registrul BILBO este configurat ca un LFSR (figura 1.34d), sau mai exact ca un MISR. Dacă Z_i-urile sunt ieşirile unui CUT, atunci registrul comprimă răspunsul pentru a forma o semnătură. Dacă intrările $Z_1, Z_2, \ldots, Z_n$ sunt ținute la valoarea 0, şi valoarea inițială a registrului nu este "*all 0s*", atunci LFSR-ul operează ca un generator de modele pseudoaleatoare.

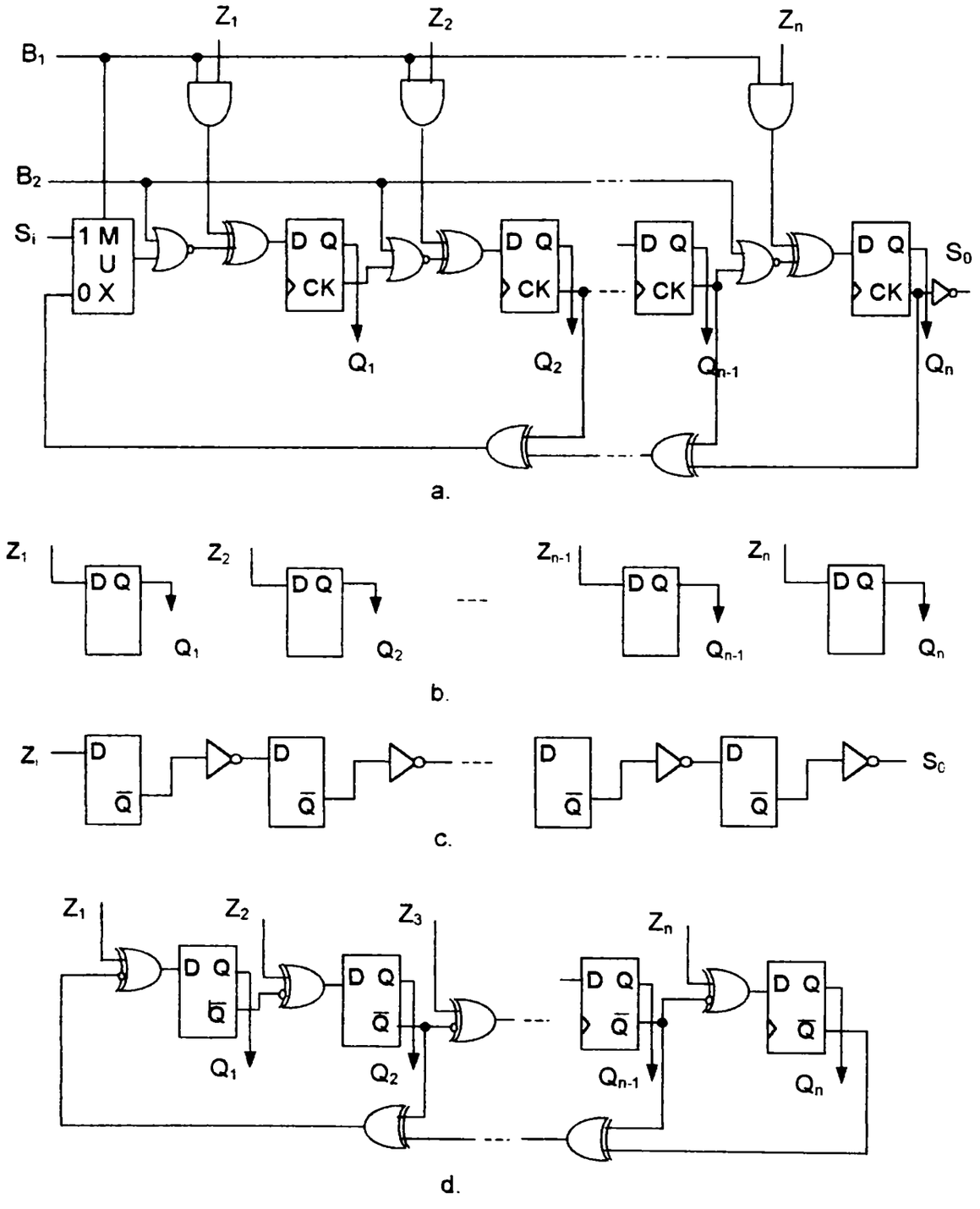


Figura 1.34. a-registru BILBO pe n biți, b-mod normal (B_1=B_2=1), c-mod registru de deplasare (B_1=B_2=0), d-mod LFSR (test) (B_1=1 B_2=0)

În arhitectura BIST BILBO, se partiționează circuitul într-un set de registre şi blocuri de logică combinațională, unde registrele normale sunt înlocuite cu registre

BILBO. În plus, intrările unui bloc de logică combinațională sunt comandate de un registru BILBO, R_i, iar ieşirile de un alt registru BILBO, R_j.

Fie circuitul prezentat în figura 1.35a, unde registrele sunt de tip BILBO. Pentru testarea lui C_1, R_1 şi R_2 sunt inițializate şi apoi R_1 este adus în mod PRPG şi R_2 în mod MISR. Presupunem că intrările lui R_1 sunt menținute la valoarea 0. Circuitul va funcționa în acest mod timp de N cicluri de tact. Dacă numărul intrărilor lui C_1 nu este prea mare, C_1 poate fi testat exhaustiv, exceptând modelul "*all 0*". La finele acestui proces de test, numit sesiune de test, conținutul lui R_2 poate fi transmis la ieşire şi semătura poate fi verificată. Similar C_2 poate fi testat prin configurarea lui R_1 într-un MISR şi R_2 în PRPG. Astfel circuitul va fi testat în două sesiuni de test.

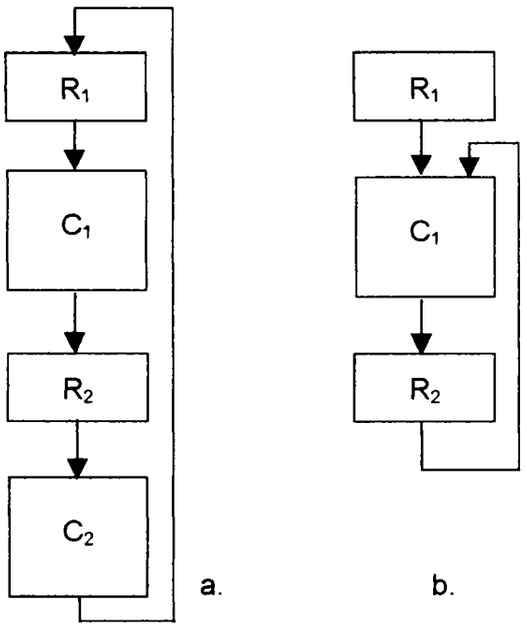
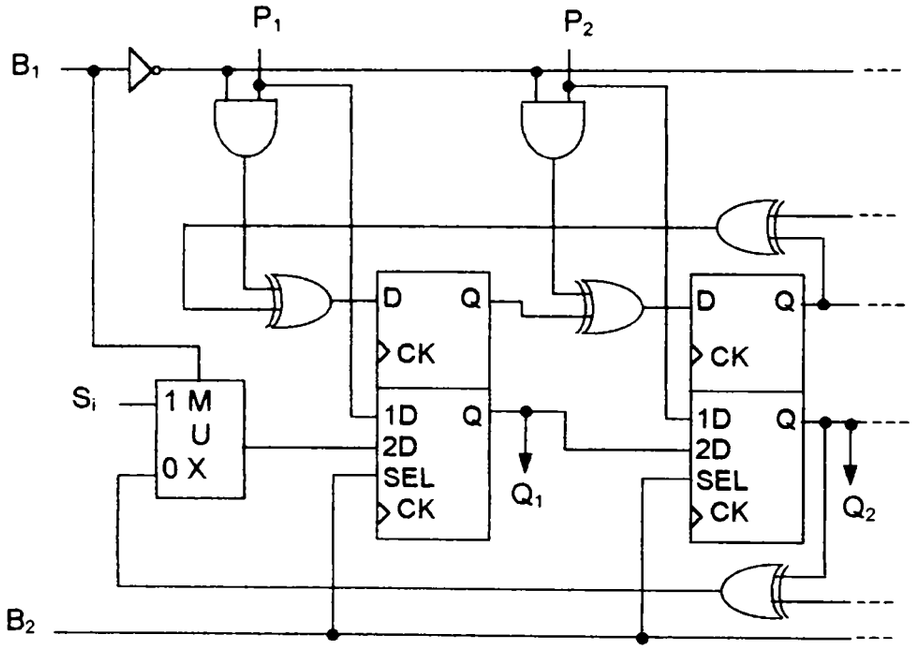


Figura 1.35. Schemă BIST cu registre BILBO

În figura 1.35b este prezentată o altă configurație de circuit, care are o semibuclă în jurul registrului BILBO R_2. Această schemă nu este o arhitectură BILBO normală. Pentru testarea lui C_1, R_1 trebuie să funcționeze în mod PRPG. De asemenea R_2 trebuie să fie atât în mod MISR cât şi PRPG. Acest lucru nu este posibil pentru schema prezentată în figura 1.34a. Ceea ce se poate face este aducerea lui R_2 în modul MISR şi astfel ieşirile lui vor fi vectori aleatori care pot fi folosiți în testarea lui C_1. Erorile din MISR vor produce modele de test eronate care vor fi aplicate lui C_1, care va produce la rândul său mai multe erori în R_2. Din păcate pot exista situații în care defectul nu va fi niciodată detectat, ca de exemplu cea în care data de la intrarea lui C_1 nu va propaga niciodată efectul defectului spre ieşirea lui C_1.

Această situație poate fi corectată prin utilizarea unui registru de tip "concurent built-in logic-bloc observation" (CBILBO). Acest registru este prezentat în figura 1.36.

El operează simultan ca un MISR şi un PRPG. Bistabilele D din partea superioară şi logica aferentă formează un MISR. Bistabilele dual-port din rândul de jos şi logica corespunzătoare formează un registru care poate opera fie în modul de încărcare paralelă normală, fie în mod scan, fie ca un PRPG. Atunci când registrul BILBO este în modul PRPG, intrările lui trebuie să fie ţinute la o valoare constantă. Acest lucru se poate realiza în mai multe moduri. Atunci când metodologia de test BILBO se aplică unui sistem modular orientat pe magistrală, în care modulele funcţionale ca ALU, RAM şi ROM sunt conectate printr-un registru la o magistrală (figura 1.37), la dezactivarea tuturor driver-elor de magistrală şi utilizând circuite pull-up sau pull-down intrările registrului pot fi ţinute într-o stare constantă.



B_1	B_2	mod de operare
-	0	normal
1	1	scan
0	1	PRPG/MISR

Figura 1.36. Registru BILBO concurent

Unele arhitecturi au structuri pipeline, ca cea din figura 1.38. Pentru dezactivarea intrărilor la un registru BILBO, atunci când acesta operează în modul PRPG, se poate utiliza o schemă BILBO modificată care are trei semnale de control (B_1, B_2, B_3), aşa cum se vede în figura 1.39. Există opt stări de control posibile dintre care una poate fi utilizată pentru specificarea modului MISR şi alta pentru modul PRPG.

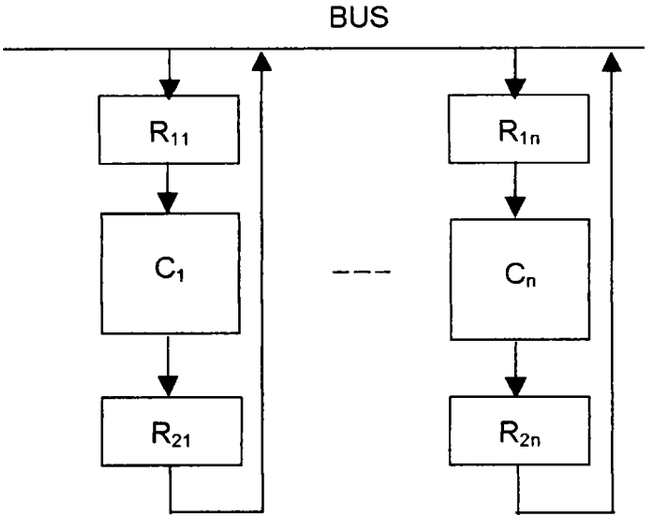


Figura 1.37. Arhitectură BILBO orientată pe magistrală

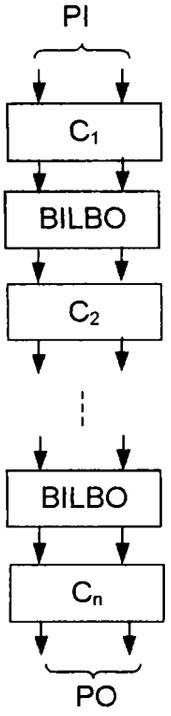
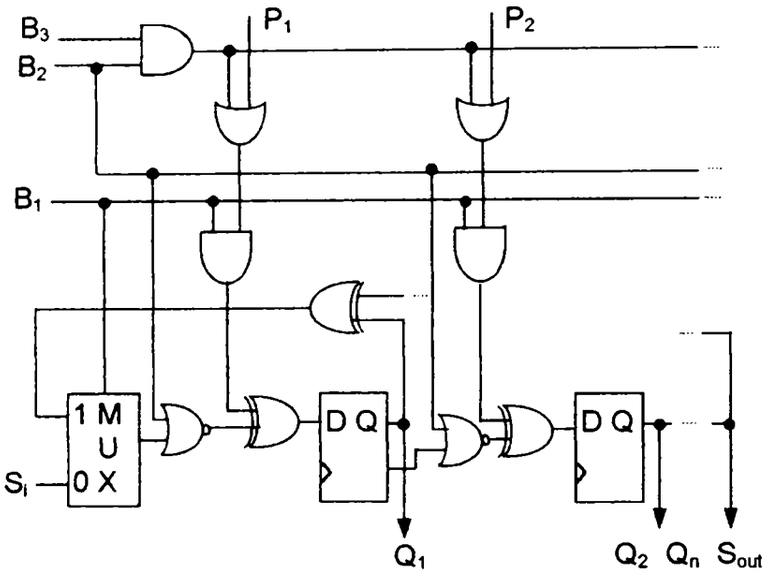


Figura 1.38. Arhitectură BILBO pipeline

Unul dintre aspectele care diferențiază arhitectura BILBO de cea BIST discutată anterior este partiționarea celulelor de memorie pentru a forma registre şi partiționarea logicii combinaționale în blocuri de logică. O dată realizată această partiționare, multe dintre tehnicile de testare pseudoexhaustivă prezentate anterior pot fi aplicate. Registrele BILBO pot fi înlocuite de alte tipuri de registre.



B_1	B_1	B_1	mod de operare
1	1	0	normal
0	1	0	reset
1	0	0	analiză de semnătură (MISR)
1	0	1	generare de modele (PRPG)
0	0	0	scan

Figura 1.39. Registru BILBO cu intrare de control

În acest capitol a fost prezentată evoluția arhitecturilor BIST. Inițial aplicarea unor tehnici ca CSBL şi BEST a permis testarea circuitelor secvențiale folosind modele de test pseudoaleatoare şi comprimarea răspunsurilor utilizând analizoare de semnătură. Pentru un acces mai bun la circuitele interne se pot utiliza căi de scan astfel că nucleul testat este combinațional. Următorul pas a fost incorporarea PRPG şi MISR în CUT şi aplicarea boundary scan pentru testarea logicii izolate. Pentru a reduce timpul de test, s-au elaborat diverse metode, ca de exemplu SST şi CSTP, în care căile de scan nu trebuiesc încărcate serial înainte de aplicarea fiecărui model de test. Aici s-au utilizat scheme "pseudo-scan-path" mai complexe. În final a fost prezentată o arhitectură BIST bazată pe registre (BILBO).

Una dintre problemele asociate diverselor scheme BIST, ca de exemplu RTD, SST şi CSTP este că în timpul autotestului, intrările D la celulele de memorie nu sunt testate. Din acest motiv este nevoie ca în timpul operării normale să fie executate câteva teste suplimentare. Acest lucru se poate face prin încărcarea căii de scan cu un model de test, aplicarea unui tact normal şi apoi verificarea conținutului căii de scan. În mod normal este suficientă testarea logicii care nu este testată în procesul de autotest.

CAPITOLUL 2

DESCRIEREA CIFRULUI BLOC RC6

2.1. Caracteristici ale algoritmilor candidați AES

În anii 1997-1998 au fost prezentați Institutului Național de Standarde şi Tehnologii (National Institute for Standard and Technology – NIST) 15 algoritmi de criptare, printre care şi algoritmul RC6, în scopul adoptării unuia dintre aceştia ca nou standard avansat de criptare (Advanced Encryption Standard - AES). Acest prim paragraf al capitolului 2, analizează pe scurt candidații AES din diferite puncte de vedere – securitate, viteză, performanțe pe diferite platforme, posibilități de implementare, etc.

Cele 15 cifruri bloc înaintate către NIST sunt prezentate în tabelul 2.1, iar structura generală a acestora este descrisă sumar în tabelul 2.2 [NIST-97].

Tabelul 2.1. Cei 15 candidați AES

Numele cifrului	Elaborat de	Țara
CAST-256	Entrust	Canada
Crypton	Future Systems	Coreea
Deal	Outerbridge	Canada
DFC	ENS-CNRS	Franța
E2	NTT	Japonia
Frog	Tec Apro	Costa Rica
HPC	Schroeppel	USA
LOKI97	Brown, Pieprzyk, Seberry	Australia
Magenta	Deutsche Telekom	Germania
MARS	IBM	USA
RC6	RSA	USA
Rijndael	Daemen, Rijmen	Belgia
Safer+	Cylink	USA
Serpent	Anderson, Biham, Knudsen	UK, Israel, Norvegia
Twofish	Counterpane	USA

Cifrurile diferă prin:

- Ideile de proiectare de nivel înalt – bazate pe cifruri existente, cifruri complet noi, rețele Feinstel/SP, proiectarea funcțiilor unei runde, număr de runde
- Instrucțiunile utilizate – sau-exclusiv, cutii S, adunări-scăderi, rotații-deplasări, multiplicări modulo 2^{32}, 2^{64} sau $2^{64}+13$, rotații variabile
- Tehnici – bitslicing, decorelare, runde noncriptografice, utilizarea altor cifruri
- Obiectivul optimizării – Pentium, MMX, Pro, Pentium II, procesoare pe 64 biți, procesoare pe 16 biți, procesoare pe 8 biți, smartcard-uri

Tabelul 2.2. Structura generală

Cifru	Tip	Runde	Utilizează
CAST-256	Ext. Feinstel	48	
Crypton	Square	12	
Deal	Feinstel	6, 8, 8	DES
DFC	Feinstel	8	Module de decorelare, mult.
E2	Feinstel	12	
Frog	Special	8	BombPermu
HPC	Omni	8	Hasty Pudding
LOKI97	Feinstel	16	
Magenta	Feinstel	6, 6, 8	
MARS	Ext. Feinstel	32	rot. var., mult., runde noncript.
RC6	Feinstel	20	rot. var., mult.
Rijndael	Square	10, 12, 14	
Safer+	SP network	8, 12, 16	PHT
Serpent	SP network	32	Bitslice
Twofish	Feinstel	16	

rot. var.= rotatii variabile mult.=multiplicare.

Scopul principal în proiectarea unui algoritm de criptare trebuie să fie securitatea. Totuşi, în lumea reală un rol important îl au şi performanțele şi costul de implementare al algoritmului [MaMa-95a] [MaMa-96] [Leib-99]. Presupunând că cei 15 candidați AES prezintă siguranță, analiza pe care am făcut-o se focalizează mai mult asupra performanțelor lor în implementări pe mai multe platforme.

Viteza majorității candidaților AES este independentă de lungimea cheii. Aceasta înseamnă că timpul necesar setării cheii şi criptării unui bloc de text este acelaşi indiferent dacă cheia are 128, 192, sau 256 de biți. Totuşi o parte dintre ei nu au această proprietate. Din acest punct de vedere, cei 15 algoritmi vor forma trei grupe; nouă algoritmi - CAST-256, Crypton, DFC, E2, Frog, HCP, MARS, RC6 şi Serpent – sunt independenți de lungimea cheii, doi algoritmi - Loki97 şi Twofish – criptează şi decriptează independent de lungimea cheii dar necesită timpi diferiți pentru setarea cheilor de mărimi diferite, iar patru algoritmi - DEAL, Magenta, Rijndael şi SAFER+ - criptează şi decriptează la viteze diferite, funcție de lungimea cheii [Mokr-99], [Mass-94].

Unul dintre criterile impuse de către NIST candidaților la noul standard de criptare este eficiența pe 32 de biți. De fapt trebuiesc luate în considerare două tipuri de eficiențe. Primul este cel evidențiat de NIST, şi anume pe CPU-uri Pentium high-end – care vor domina piața calculatoarelor în următorii ani. Al doilea tip se referă la CPU-uri low-end pe 32 de biți – cum sunt 80386 şi variante ale lui 68000, precum şi diferite chip-uri RISC simple -, care în scurt timp vor înlocui CPU-urile pe 8 biți de pe smartcard-urile high-end.

Tabelul 2.3. Performanțele candidaților AES cu chei de 128 biți pe CPU-uri din clasa Pentium

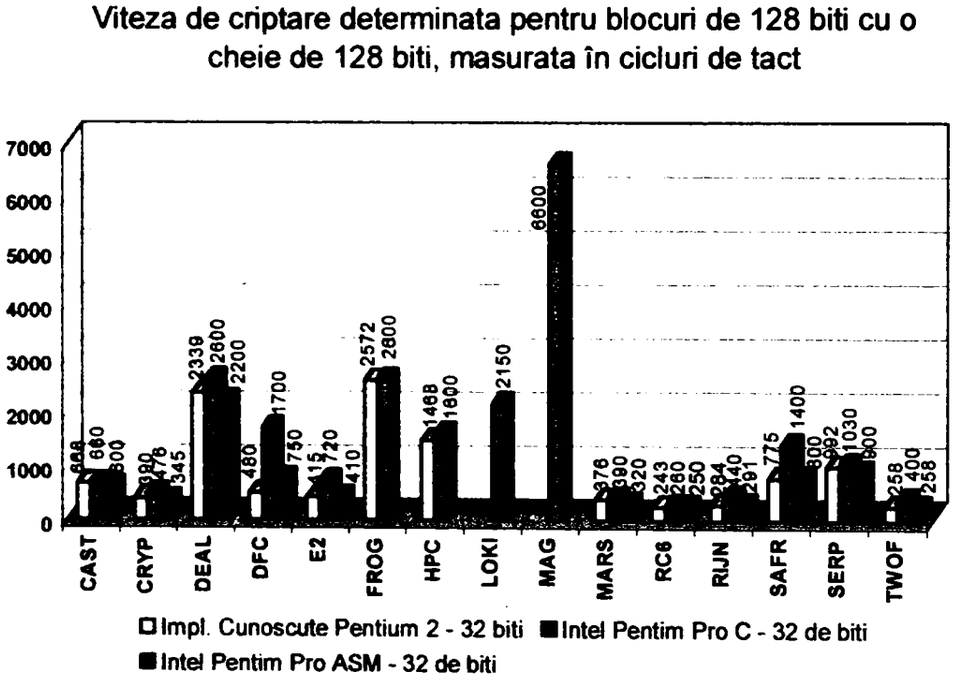
Nume algorithm	Setare cheie Pentium Pro C (clocks)	Criptare Pentium Pro C (clocks)	Criptare Pentium Pro ASM (clocks)	Criptare Pentium ASM (clocks)
CAST-256	4300	660	600	600
Crypton	955	476	345	390
Deal	4000	2600	2200	2200
DFC	7200	1700	750	?
E2	2100	720	410	410
Frog	1386000	2600	?	?
HPC	120000	1600	?	?
LOKI97	7500	2150	?	?
Magenta	**50**	6600	?	?
MARS	4400	390	320	550
RC6	1700	**260**	**250**	700
Rijndael	850	440	291	320
Safer+	4000	1400	800	1100
Serpent	2500	1030	900	1100
Twofish	8600	400	258	**290**

În tabelul 2.3 sunt date performanțele candidaților AES atât pe platforme Pentium cât şi pe Pentium Pro. Aici trebuie menționat faptul că cifrurile Crypton şi Rijndael au viteze foarte diferite de setare ale cheii, pentru criptare, respectiv decriptare. Setarea cheii de cripatere pentru Rijndael necesită 300 de tacturi în timp ce pentru decriptare sunt necesare 1400 de tacturi. În cazul lui Crypton pentru criptare şi decriptare sunt necesare 540, respectiv 1370 de tacturi. Menționez că în tabel, în coloana corespunzătoare setării cheii, am înscris media valorilor din cazul criptării respectiv a decriptării.

Aşa cum reiese din tabelul 2.3, candidații AES au performanțe foarte diferite pe CPU-urile de 32 de biți, de la 250 de cicluri de tact – în cazul cifrului RC6 – până la 6600 în cazul lui Magenta. Pentru criptările masive, cei mai rapizi algoritmi sunt: Twofish, Rijndael, Crypton, şi E2, urmați de MARS şi RC6 [CoYi-98] [CRRY-99].

În literatura de specialitate se găsesc foarte multe date referitoare la viteza candidaților implementați pe diverse platforme. Majoritatea lor sunt legate de platforme de 32 de biți, ceea ce nu reprezintă neapărat o referință pentru majoritatea aplicațiilor AES care vor fi implementate în următorii 20 de ani. Se cunosc de asemenea date referitoare la implementările pe platforme de 64 de biți şi mai puține

pentru implementările pe 8 biți. În graficul 2.1 am sintetizat o parte a datelor cunoscute pentru platformele de 32 de biți. Pe baza datelor cunoscute se pot trage următoarele concluzii: Rijndael şi Twofish au viteze uniforme, şi performanțe foarte bune pe toate platformele, MARS şi RC-6 au cele mai bune performanțe pe platformele de 32 de biți şi mai puțin bune pe restul, DFC şi HPC sunt orientate pe arhitecturi de 64 de biți şi nu se comportă bine dacă mărimea cuvântului este mai mică, CAST-256, CRYPTON E2 şi Serpent se comportă în mod asemănător pe oricare dintre platforme, dar au viteze mai mici decât Rijndael şi Twofish şi în fine DEAL, FROG, LOKI97, MAGENTA şi SAFER+ sunt lenți pe toate platformele. NIST a considerat însă că viteza nu este factorul primar de decizie pentru promovarea unui candidat şi drept urmare Serpent a fost selecționat pentru runda 2, fiind un exemplu în acest sens.



Graficul 2.1. Viteza de criptare, determinată în cicluri de tact, pentru blocuri de 128 de biți utilizînd o cheie de 128 de biți.

Pentru criptările blocurilor de text-clar de dimensiuni mici, este indicat să se urmărească vitezele atât pentru criptare cât şi pentru decriptare. În tabelul 2.4 sunt comparate vitezele pentru chei, respectiv criptare, a şase dintre candidații AES, funcție de dimensiunea textului şi tipul platformei de test [CaDN-99]. Pentru dimensiuni mici ale blocurilor de texte cel mai rapid este algoritmul Rijndael, urmat îndeaproape de Crypton. Pe de altă parte, se remarcă o creştere semnificativă a vitezei lui RC6 şi MARS pe Pentium Pro/II față de Pentium [SKWW-99a].

Întrucât smart card-urile se utilizează acum în multe aplicații, unul dintre parametrii care poate avantaja candidații AES este implementabilitatea pe aceste

smartcard-uri. În acest caz trebuiesc luate în considerare două probleme. În primul rând, cantitatea de RAM disponibilă pe cele mai multe smartcard-uri este limitată. În al doilea rând, unele arhitecturi au la bază circuite Motorola 6085 (foarte populare pe smartcard-uri) care au rotații de un singur byte cu o poziție. Acest lucru determină ca rotațiile cuvintelor de 32 de biți, mai ales rotațile dependente de date (cazul lui MARS şi RC6) să fie foarte complexe, lente sau ambele [CoYi-98] [AnBK-98b] [DaRi-98b].

Tabelul 2.4. Cicluri de tact, pe byte, pe cheie şi criptarea textelor de diverse mărimi pe un Pentium, respectiv Pentium Pro/II.

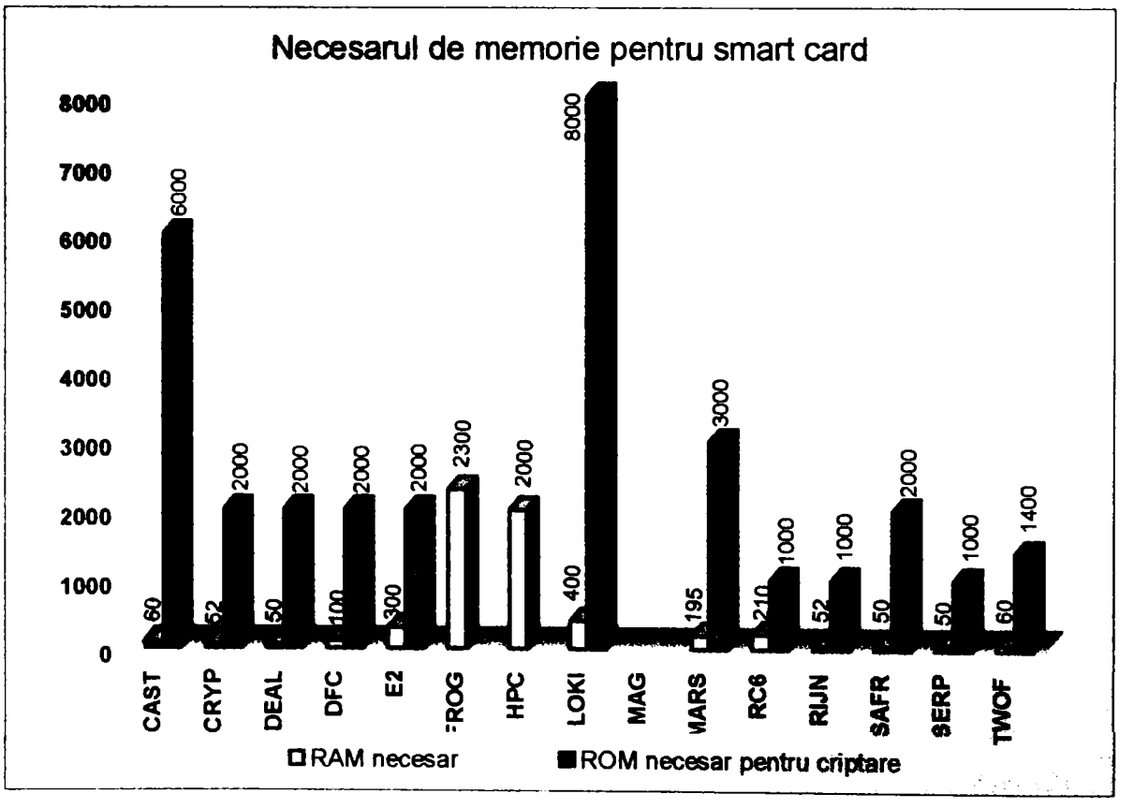
Mărimea textului (bytes)	Crypton		E2		MARS		RC6		Rijndael		Serpent		Twofish	
	Pent.	Pro/II	Pent.	Pro/II	Pent.	Pro/II	Pent.	Pro/II	Pent.	Pro/II	Pent.	Pro/II	Pent.	Pro/II
16	73	70	100	100	260	246	146	118	59	**53**	205	193	175	132
32	49	46	63	63	147	133	95	67	39	**36**	137	125	119	93
64	37	34	44	44	91	76	69	41	30	**27**	103	90	91	73
128	30	28	35	35	63	48	57	28	25	**23**	86	73	70	64
256	27	25	30	30	48	34	50	22	22	**20**	77	65	48	48
512	26	23	38	28	41	27	47	**19**	21	**19**	73	61	38	33
2^{10}	25	22	27	27	38	24	45	**17**	21	19	71	58	31	25
2^{11}	25	22	26	26	36	22	45	**16**	20	18	70	57	25	20
2^{12}	25	22	26	26	35	21	44	**16**	20	18	69	57	22	18
2^{13}	24	22	26	26	35	20	44	**16**	20	18	69	57	21	17
2^{14}	24	22	26	26	35	20	44	**16**	20	18	69	56	20	17
2^{15}+	24	22	26	26	34	20	44	**16**	20	18	69	56	19	16

Tabelul 2.5. Necesarul de RAM pe smartcard-uri pentru candidații AES

Algoritm	ROM (byti)	RAM (byti)
CAST-256	6000	60
Crypton	2000	52
Deal	2000	50
DFC	2000	200
E2	2000	300
Frog		2300+
HPC		2000
LOKI97	8000	400
Magenta		?
MARS	3000	195
RC6	1000	210
Rijndael	1000	52
Safer+	2000	50
Serpent	1000	50
Twofish	1400	60

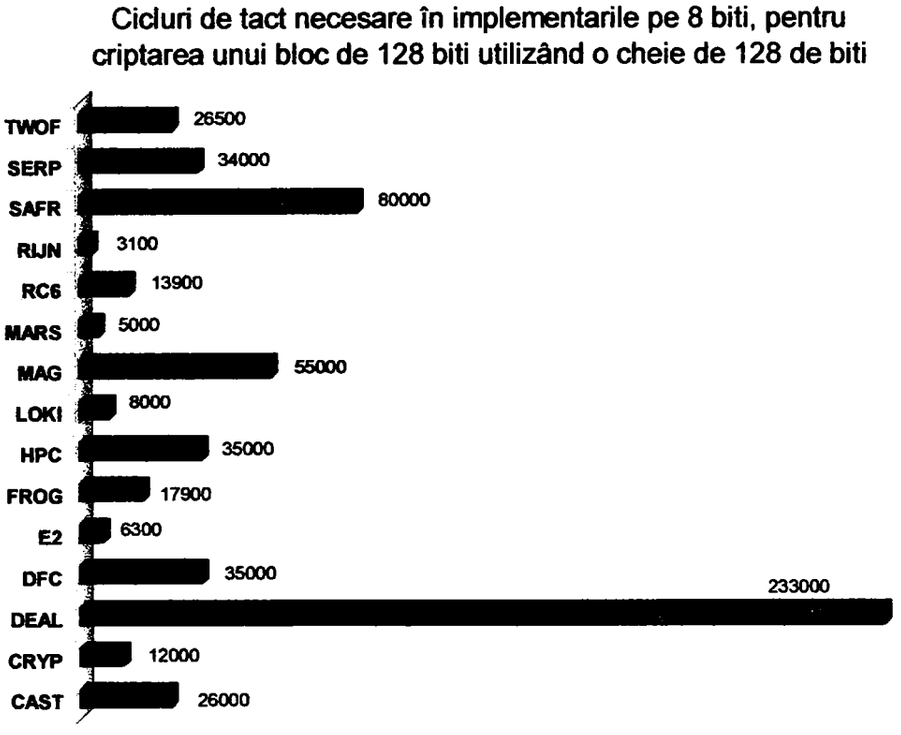
Este foarte important să se compare modul de implementare al fiecărui algoritm pe un smart card CPU, poate chiar mai important decât a se urmări cât de

rapid rulează. Un indicator esențial îl reprezintă cantitatea de RAM necesară. Subcheile, necesită între 150 şi 256 byți de RAM suplimentar, ceea ce înseamnă mai mult decât este de obicei disponibil pe majoritatea CPU-urilor folosite. În tabelul 2.5 sunt date aceste valori pentru fiecare cifru în parte. Aceste date sunt valabile dacă se presupune că în RAM este memorată cheia şi că această cheie trebuie să mai fie disponibilă şi după criptarea unui bloc singular de text. După cum se observă, există patru categorii: algoritmi care se pot implementa pe orice smartcard (necesită mai puțin de 128 byți de RAM), algoritmi care se pot implementa doar pe smartcard-uri cu performanțe mai ridicate (necesită între 128 şi 256 byți de RAM), algoritmi care se pot implementa doar pe high-end smartcard-uri (necesită peste 256 byți de RAM) şi algoritmi care nu se pot implementa pe smartcard-uri (Frog) [Biha-99]. Totuşi, evoluția tehnologică a smartcard-urilor ne face să credem că în viitorul apropiat vor fi disponibile microprocesoare moderne (68000 sau chiar ARM). De asemenea, necesarul de ROM trebuie să se păstreze în limitele rezonabilului, ceea ce înseamnă 2000 de byți [NBDDFR-99]. Datele care au stat la baza elaborării graficului 2.2 sunt furnizate de [SKWWHF-99].



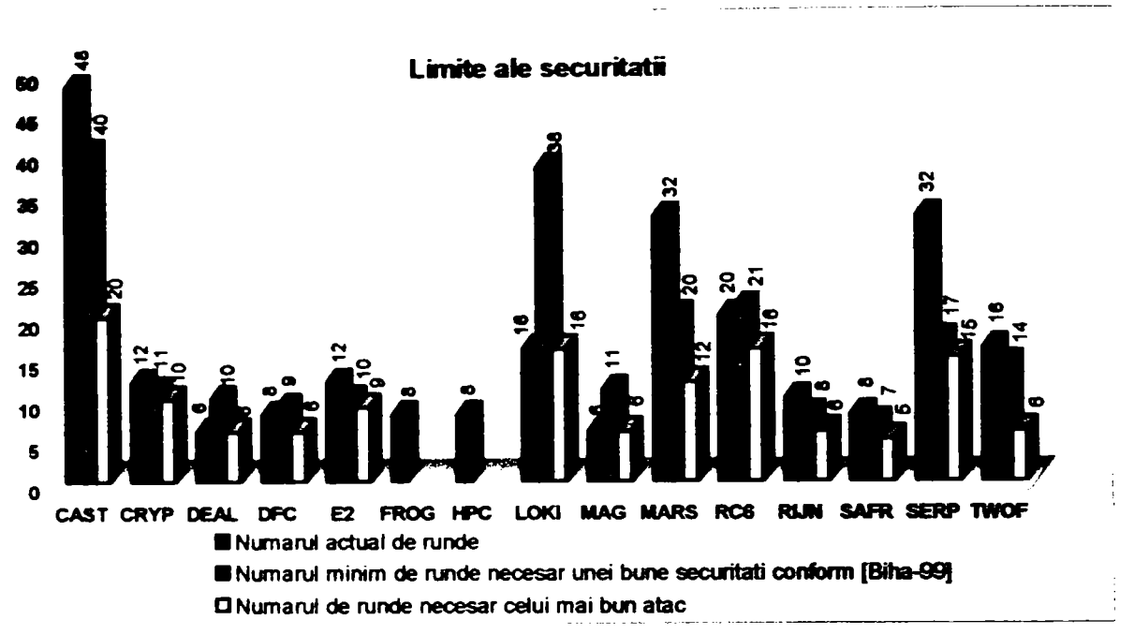
Graficul 2.2. Necesarul de memorie RAM şi ROM pentru implementarea algoritmilor pe smart card-uri.

În graficul 2.3 sunt ilustrate, pe baza datelor furnizate de către [AFSPL-99], vitezele de criptare ale celor 15 candidați pentru blocuri de 128 de biți utilizând chei cu o lungime de 128 de biți, pe platforme de 8 biți.



Graficul 2.3. Viteza de criptare pentru blocuri de 128 de biți, utizând chei de 128 de biți, pe platforme de 8 biți.

Multe etape ale unui algoritm constituie subiectul unui posibil atac. Rezistența la atac a unei anumite faze este greu să fie măsurată cantitativ. Pe de altă parte, unele dintre atacuri (de exemplu criptanaliza diferențială şi liniară) sunt direct orientate pe runde. În cazul unor astfel de atacuri se poate obține rezistență doar dacă se măreşte numărul de runde. Acest lucru ne sugerează o posibilă măsură a securității candidaților, şi anume numărul actual de runde versus numărul minim de runde necesar asigurării securității. Din păcate cea de a doua este greu de măsurat în sens absolut. În [Biha-99] este prezentată o schemă pentru măsurarea numărului minim de runde necesar pentru asigurarea securității față de atacurile cunoscute, schemă care a constituit însă subiectul unor controverse. O schemă alternativă de măsurare se referă exclusiv la numărul cel mai mare de runde existent în atacurile curente. Nici una dintre aceste scheme nu este corectă sau incorectă în sens absolut, deci aceste măsuri trebuiesc considerate ca estimări. Acceptarea validității uneia dintre aceste scheme ne conduce la noțiunea corespondentă de "margine a securității". De exemplu dacă numărul minim de runde pentru asigurarea securității este 10 şi numărul de runde specificat este 12, atunci putem spune că marginea securității este măsurată ca 20%. În graficul 2.4 sunt reprezentate aceste margini pentru cei 15 candidați AES, utilizând cele două scheme menționate.



Graficul 2.4. Limite ale securității pentru cei 15 candidați AES.

În continuare sunt trecute în revistă aspecte ale implementărilor celor cei 15 candidați AES [Glad-99] [Lipm-99].

CAST-256 este un algoritm convențional care este uşor de implementat şi are costuri reduse de implementare [Adam-98].

CRYPTON este un algoritm nou care permite utilizarea aceloraşi rutine atât la criptare cât şi la decriptare [Lim-98]. Este un algoritm destul de complicat, motiv pentru care ia destul de mult timp la implementare. În literatură sunt specificate modalități de optimizare ale algoritmului, care pot fi uşor de aplicat. Listele de chei sunt mult mai rapide pentru criptare decât pentru decriptare.

DEAL este uşor de implementat deoarece codurile sursă DES sunt disponibile. Există puține posibilități de optimizare iar eficiența algoritmului este determinată de cea a lui DES, de care depinde [Knud-98].

DFC este un algoritm care necesită un volum mare de muncă pentru implementarea pe 32 de biți deoarece toată aritmetica se bazează pe structurile pe 64 de biți. Având în vedere acest lucru, poate ar fi mai corect să se analizeze performanțele algoritmului folosind procesoare şi compilatoare corespunzătoare.

E2 este complicat, motiv pentru care este foarte costisitor de implementat şi de optimizat [NTT-98].

FROG este un algoritm uşor de implementat întrucât pseudo codul lui este furnizat de părțile lui componente. Însă lista cheilor este lentă şi eficiența ei nu poate

fi mărită [GeLC-98]. Din acest motiv FROG nu este un adevărat candidat pentru noul standard de criptare.

Algoritmul HPC complet implică cinci sub-cifruri, motiv pentru care implementarea lui este costisitoare. Totodată, datorită faptului că lista cheilor este costisitoare în comparație cu rutinele de criptare şi decriptare nici acest cifru nu constituie un candidat AES serios.

Loki97 este complicat şi utilizează indici care se calculează prin mascarea unor părți de cuvânt de lungime de 11 sau 13 biți, ceea ce presupune un consum de timp destul de mare [BrPi-98].

MAGENTA este relativ uşor de implementat. Există de asemenea posibilități de optimizare care ar putea să transforme MAGENTA într-un candidat AES serios [JaHu-98].

MARS este un cifru uşor implementabil datorită utilizării extensive a pseudo codului care îl descrie [BCDG-98]. De asemenea el poate fi uşor optimizat.

RC6 este pe departe cifrul cel mai uşor implementabil dintre toți candidații AES. Implementarea lui se poate face repede iar simplitatea lui nu lasă loc greşelilor de implementare [RRSY-98]. Performanțele lui sunt remarcabile pe Pentium II, fiind cel mai rapid dintre candidați pe această platformă. De asemenea, RC6 se optimizează uşor în C, unde performanțele sale se încadrează în procentul de 10% din performanțele care se pot atinge în limbaj de asamblare.

Rijndael este o variantă a pătratului cu o structură ordonată care permite o optimizare foarte bună pe procesoarele pe 32 de biți. Deoarece utilizează doar instrucțiuni comune şi eficiente, performanțele lui Rijndael sunt foarte bune pe multe procesoare. Lista cheilor este asimetrică şi este mult mai rapidă pentru criptare decât pentru decriptare [DaRi-98a]. Fiind uşor de implementat şi având performanțe bune pentru un număr mic, mediu şi mare de blocuri criptate, Rijndael este un candidat important.

SAFER+ este un algoritm orientat pe byte dar care nu profită de operațiile pe 32 de biți disponibile în procesoarele Pentium II. Ca urmare, performanțele lui nu sunt foarte bune pe astfel de procesoare [CMKK-98].

Serpent este un algoritm care foloseşte metoda bitslice în implementare. Performanțele lui sunt relativ scăzute în comparație cu ai celorlalți candidați deoarece necesită un număr foarte mare de runde [AnBK-98a]. Versiunea bitsclice a algoritmului depinde de găsirea unor funcții booleene pentru reprezentarea cutiile S ce pot fi calculate într-un număr minim de cicluri procesor. Astfel de optimizări au fost interprinse ca parte a procesului de implementare.

Twofish este un algoritm complex care combină diferite tehnici. Pentru o implementare cu performanțe optime costurile sunt destul de ridicate. Avantajul pe care îl prezintă acest algoritm este acela că el poate fi implementat în multe moduri, ceea ce permite o optimizare a lui pentru numeroase aplicații [SKWW-99b].

2.2. Descrierea cifrului bloc RC6

Cifrul RC6 este un cifru bloc nou, propus de către laboratoarele RSA din Statele Unite ale Americii, laboratoare conduse de catre Ronald L. Rivest – creator al algoritmului RSA, cifrurilor RC2, RC4, RC5. Cifrul RC6 reprezintă o îmbunătățire a cifrului RC5 [Rive-96], care a fost elaborat în anul 1995 şi a reprezentat prima propunere a colectivului laboratorului RSA pentru a fi cifrul care să înlocuiască algoritmul DES – actualul standard de criptare.

Proiectarea cifrului RC6 a pornit de la cifrul RC5 existent [Rive-96], cifru care a fost îmbunătățit în scopul respectării cerințelor AES, a creşterii securității şi îmbunătățirii performanțelor.

Cifrul RC5 este un cifru deosebit de simplu. Diversele studii făcute din 1995 [BiKu-98], [KaYi-95a], [KnMe-96], dată la care a fost elaborat acest cifru, au analizat modul în care structura şi operațiile lui contribuie la securitatea acestuia. Aceste studii au propus atacuri teoretice interesante asupra cifrului, atacuri bazate pe faptul că "suma rotațiilor" din RC5 nu depinde de toți biții din registru. RC6 a fost proiectat astfel încât să zădărnicească astfel de atacuri şi chiar orice fel de atac, îndeplinind astfel cerințele AES.

Una dintre cerințele AES este aceea ca mărimea blocurilor de intrare/ieşire să fie de 128 de biți. Cifrul RC5 este un cifru bloc foarte rapid. Extinderea lui la blocuri de 128 de biți ar presupune în mod normal utilizarea unor registre de 64 de biți, dar arhitectura şi limbajele specificate de AES nu suportă operații pe 64 de biți într-o manieră eficientă şi curată. Din acest motiv cifrul a fost modificat pentru a se putea utiliza registre de 32 de biți. Acest lucru are două avantaje. În primul rând, faptul că se execută două rotații în fiecare rundă față de una, care se executa în jumatatea de rundă din cifrul RC5, şi apoi faptul că sunt utilizați mai mulți biți de date pentru a determina suma rotațiilor din fiecare rundă.

Unul dintre atuurile cifrului RC5 este acela de a exploata operațiile (de exemplu, rotațiile) care sunt implementate eficient în procesoarele moderne [Rive-96]. Această caracteristică este păstrată şi de RC6, care utilizează multiplicări de întregi pe 32 de biți, operație implementată eficient pe aproape toate procesoarele. Multiplicarea întregilor este o primitivă de difuzie deosebit de eficientă. Ea este utilizată în RC6 pentru calculul sumelor rotațiilor, astfel că acestea sunt dependente de toți biții unui alt registru şi nu doar de biții mai puțin semnificativi (ca şi în cazul lui RC5). Ca urmare RC6 are o difuzie mai rapidă decât RC5, şi în plus permite ca execuția lui RC6 să se facă cu mai puține runde dar cu securitate sporită.

La fel ca şi RC5 şi RC6 [RRSY-98] reprezintă o întreagă familie parametrizată de algoritmi de criptare. Specificarea corectă a lui RC6 este de forma RC6-*w/r/b*, unde *w* reprezintă lungimea cuvântului exprimată în biți, *r* este un număr nenegativ care specifică rundele, iar *b* este lungimea cheii dată în biți. Intrucât specificațiile AES precizează că *w* trebuie să fie 32 şi *r* este 20, în continuare vom folosi doar notația RC6.

Pentru toate variantele, RC6-*w/r/b* operează cu blocuri de patru cuvinte de *w*-biți utilizând următoarele şase operații de bază:

a+b adunare întreagă modulo 2^w
a-b scădere întreagă modulo 2^w
a⊕b sau-exclusiv la nivel de bit a cuvinte de *w*-biți
a×b multilpicare întreagă modulo 2^w
a‹‹‹b rotație a cuvântului de *w*-bit la stânga cu cantitatea dată de cei mai puțin lg*w* biți ai lui *b*
a›››b rotație a cuvântului de *w*-bit la dreapta cu cantitatea dată de cei mai puțin lg*w* biți ai lui *b*

unde lg*w* reprezintă logaritmul în baza 2 din *w*:

Termenul de rundă folosit în descriere este oarecum similar cu cel cunoscut din descrierea cifrului DES: jumătate dintre date sunt actualizate de cealaltă jumătate iar apoi cele două sunt interschimbate.

Key schedule for RC6-*w/r/b*

Input: User-supplied *b* byte key preloaded into the *c*-word array L[0,...,*c* -1]
Number *r* of rounds

Output: *w*-bit round keys S[0,...2*r* +3]

Procedure: $S[0]=P_w$

for *i*=1 **to** 2*r* +3 **do**
$S[i]= S[i -1]+Q_w$

A=B=*i*=*j*=0

v=3×max{*c*,2*r* +4}
for *s*=1 **to** *v* **do**
{
A=S[*i*]=(S[*i*]+A+B)‹‹‹3
B=L[*j*]=(L[*j*]+A+B)‹‹‹ (A+B)
i=(*i* +1)mod(2*r* +4)
j=(*j* +1)mod*c*
}

Figura 2.1. Programarea cheii.

Listarea cheilor în algoritmul RC6 (Figura 2.1) diferă față de cel din RC5 doar prin faptul că din cheia furnizată de utilizator se derivă mai multe cuvinte care vor fi utilizate în timpul procesului de criptare sau decriptare. Utilizatorul furnizează o cheie de *b* biți, unde $0 \leq b \leq 255$.

Algoritmul de expansiune a cheii are ca scop obținerea unui tabel al cheilor expandate, notat cu S, din cheia utilizator K. S este un şir care conține $t = 2(r + 1)$

cuvinte binare aleatoare determinate de K. Algoritmul utilizează două "constante magice" şi constă din trei părţi simple.

Cele două constante magice, P_w şi Q_w, au lungimea egală cu cea a unui cuvânt şi se definesc pentru un *w* arbitrar în felul următor:

$$P_w = Odd((e - 2)\, 2^w)$$

$$Q_w = Odd((\Phi - 1)\, 2^w)$$

unde:

e = 2,718281828459.... (baza logaritmului natural)

Φ = 1.618033988749... (raportul de aur)

şi unde Odd(*x*) este întregul impar cel mai apropiat de x (rotunjit în sus dacă *x* este un întreg par, ceea ce nu se va întâmpla aici) [Rive-96].

Primul pas al algoritmului de expansiune al cheii constă în copierea cheii secrete K[0,...,*b* - 1] într-un şir L[0,...,*c* - 1] de $c = \lceil b / u \rceil$ cuvinte, unde $u = w / 8$ şi reprezintă numărul de byţi/cuvânt. Această operaţie se face într-o manieră naturală, utilizând *u* byţi consecutivi ai cheii K pentru a completa fiecare cuvânt succesiv în L, de la byte-ul cel mai puţin semnificativ spre byte-ul cel mai semnificativ. În poziţiile necompletate ale byte-ului sunt înscrise zerouri. Dacă $b = c = 0$ atunci vom reseta *c* la valoarea 1 şi L[0] la zero.

Al doilea pas al algoritmului de expansiune al cheii constă în iniţializarea şirului S la un "pattern" de bit pseudo-aleator fixat în mod particular (independent de cheie), utilizând o progresie aritmetică modulo 2^w determinată de constantele magice P_w şi Q_w. Întrucât Q_w este impar, progresia aritmetică are perioada 2^w.

```
S[0] = Pw;
for i = 1 to t -1 do
        S[i] = S[i-1] + Qw;
```

Al treilea pas al algoritmului este cel de mixare în cheia utilizator secretă, a şirurilor S şi L, pe parcursul a trei etape. Mai exact, având în vedere dimensiunile diferite ale lui S şi L, şirul cu lungime mai mare va fi procesat de trei ori iar celălat şir poate fi prelucrat (handled) de mai multe ori.

```
i = j = 0;
A = B = 0;
do 3 * max(t,c) times;
        A = S[i] = (S[i] + A + B <<< 3;
        B = L[j] = (L[j] + A + B <<< (A + B);
        i = (i + 1) mod(t);
        j = (j + 1) mod(c);
```

Observație. Se remarcă faptul că funcția de expansiune a cheii face parte într-o oarecare măsură din categoria funcțiilor greu inversabile deoarece nu este foarte simplu de dedus K din S.

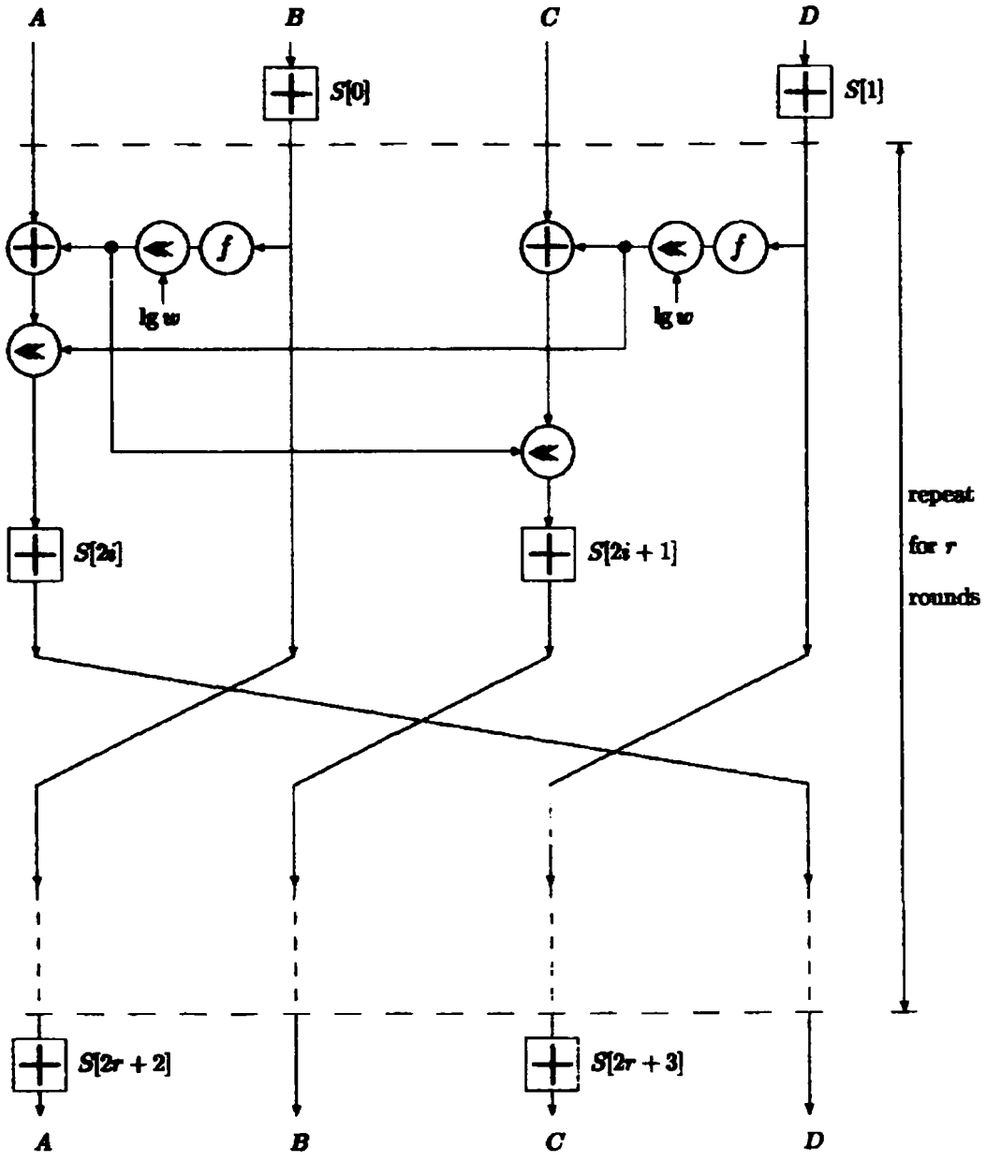


Figura 2.2. Criptarea cu RC6-w/r/b.

RC6 utilizează patru registre de *w*-biți, notate cu A,B,C,D, care conțin textul clar inițial precum şi textul cifrat de ieşire, obținut în urma criptării. Primul byte al textului clar sau al celui cifrat este plasat pe byte-ul cel mai puțin semnificativ a lui A; ultimul byte al textului clar sau al celui cifrat este plasat în byte-ul cel mai semnificativ a lui D. Echivalența (A,B,C,D) = (B,C,D,A) reprezintă asignarea paralelă a valorilor din dreapta regiştrilor din stânga. Vectorii de test pentru criptarea cu RC6 sunt dați în anexa 1. În figurile 2.2, 2.3 şi 2.4 sunt detaliate procesele de criptare şi decriptare cu cifrul bloc RC6.

Encryption with RC6-*w/r/b*

Input: Plaintext stored in four *w*-bit input registers A,B,C,D
Number r of rounds
w-bit round keys S[0,...,2*r* +3]

Output: Ciphertext stored in A,B,C,D

Procedure:
```
B=B+S[0]
D=D+S[1]
        for i=1 to r do
          {
              t=(B×(2B+1)) ‹‹‹ lgw
              u=(D×(2D+1)) ‹‹‹ lgw
              A=((A⊕t) ‹‹‹ u)+S[2i]
              C=((C⊕u) ‹‹‹ t)+S[2i +1]
              (A,B,C,D)=(B,C,D,A)
          }
A=A+S[2r +2]
C=C+S[2r +3]
```

Figura 2.3. Criptarea cu RC6

Decryption with RC6-*w/r/b*

Input: Ciphertext stored in four *w*-bit input registers A,B,C,D
Number *r* of rounds
w-bit round keys S[0,...,2*r* +3]

Output: Plaintext stored in A,B,C,D

Procedure:
```
C=C-S[2r +3]
A=A-S[2r +2]
for i=r to 1 do
  {
      (A,B,C,D)=(B,C,D,A)
u=(D×(2D+1)) ‹‹‹ lgw
t=(B×(2B+1)) ‹‹‹ lgw
      C=((C-S[2i +1]) ››› t) ⊕ u
A=((A-S[2i]) ››› u) ⊕ u

  }
D=D-S[1]
      B=B-S[0]
```

Figura 2.4. Decriptarea cu RC6.

2.3. Performanțele cifrului RC6

În aceast paragraf sunt prezentate rezultatele unor măsurători ai timpilor de criptare şi decriptare ai lui RC6 şi timpul necesar pentru setarea cheii. Rezultatele specificate pentru implementarea în ANSI C a cifrului au fost obținute folsind un compilator Borland C^{++} Development Suite 5.0, aşa cum se precizează în cerințele AES. Măsurătorile au fost făcute pe un Pentium II la 266 Mhz, cu 32MB RAM folosind Windows 95. Pentru îmbunătățirea rezultatelor au fost dezactivate întreruperile mascabile în timpul execuției testelor. Fiecare set de teste a fost făcut de zece şi apoi s-a făcut media rezultatelor, medie care apare de fapt în tabele. Rezultatele din tabelul 2.6 corespunzătoare implementării în limbaj de asamblare au fost obținute pe acelaşi calculator şi în condiții identice. Rezultatele implementării lui RC6 au fost obținute pe un Pentium Pro la 180 MHz, cu 64 MB RAM şi rulând sub Windows NT 4.0. Această implementare a fost compilată cu compilatorul JavaSoft's JDK 1.1.6 iar performanța byte-ului de cod obținut a fost măsurată atât cu interpretorul JavaSoft's JDK 1.1.6 cât şi cu Symantec Corporation's Java! JustInTime Compiler versiunea 210.054 pentru JDK 1.1.2. Şi în acest caz măsurătorile au fost făcute de 10 ori iar rezultatul reprezintă media acestora.

Rezultatele din tabelul 2.6 nu includ setarea cheii şi sunt independente de lungimea cheii furnizate de utilizator [RBSY-98]. Timpii în ANSI C au fost obținuți prin criparea şi decriptarea unei singure date de 3000 de blocuri în modul ECB, cei în Java au fost obținuți prin criptarea şi decriptarea unor date de 100000 de blocuri în modul ECB, iar timpii în limbaj de asamblare au fost obținuți prin criptarea şi decriptarea iterativă a unui singur bloc de 3000 de ori.

Este posibil să existe şi implementări mai rapide.

Tabelul 2.6. Viteza de criptare şi decriptare a lui RC6 în ANSI C, Java şi asamblare

	Schema	Cicluri/bloc	Blocuri/sec la 200 MHz	MB/sec la 200 MHz
ANSI C	RC6 criptare (ori ce mărime a cheii)	616	325000	5,19
ANSI C	RC6 decriptare (ori ce mărime a cheii)	566	353000	5,65
Java (JDK)	RC6 criptare (ori ce mărime a cheii)	16200	12300	0,197
Java (JDK)	RC6 decriptare (ori ce mărime a cheii)	16500	12100	0,194
Java (JIT)	RC6 criptare (ori ce mărime a cheii)	1010	197000	3,15
Java (JIT)	RC6 decriptare (ori ce mărime a cheii)	955	209000	3,35
asamblare	RC6 criptare (ori ce mărime a cheii)	254	787000	12,6
asamblare	RC6 decriptare (ori ce mărime a cheii)	254	788000	12,6

Timpul necesar pentru setare cheii cu RC6-32/20/*b* când se folosesc chei cu lungimea *b*=128, *b*=192 şi *b*=256 de biți este dat în tabelul 2.7. În cazul implementării în ANSI C timpii au fost obținuți prin calculul a 3000 de programări ale cheii iar pentru Java a 10000 de programări.

Tabelul 2.7. Timpii de setare ai cheii pentru RC6 în ANSI C şi Java

	Schema	Cicluri	µsec la 200 MHz	Setări cheie/sec la 200 MHz
ANSI C	RC6-32/20/16	4710	23,5	42500
Java (JDK)	RC6-32/20/16	107000	537	1860
Java (JIT)	RC6-32/20/16	14300	71,4	14000
ANSI C	RC6-32/20/24	4710	23,6	42400
Java (JDK)	RC6-32/20/24	108000	542	1840
Java (JIT)	RC6-32/20/24	14300	71,5	14000
ANSI C	RC6-32/20/32	4720	23,6	42400
Java (JDK)	RC6-32/20/32	110000	548	1820
Java (JIT)	RC6-32/20/32	15000	75,1	13300

În continuare sunt prezentate performanţele cifrului bloc RC6 pe procesoare de 8 biţi şi în particular pe familia de microcontroller-e Intel MCS 51 [Intel-94]. Estimările făcute se pot considera valide şi pentru familia de procesoare Philips'80C51 [PhSe-96] deoarece seturile de instrucţiuni şi timpii celor două familii sunt similare.

Criptare/decriptare. Vom considera mai întâi funcţia unei runde a lui RC6 (secţiunea 2.2). Aceasta constă din şase adunări, două operaţii sau-exclusiv, două ridicări la putere, două rotaţii la stânga cu cinci poziţii şi două rotaţii la stânga cu un număr variabil r de poziţii. Expresia $B \times (2B + 1) = 2B^2 + B$ se va calcula ca o ridicare la putere şi o adunare.

Operaţiile de bază pot fi implementate pe un procesor pe 8 biţi în felul următor:

1. O adunare pe 32 de biţi se poate calcula utilizând patru adunări pe 8 biţi cu transport (ADDC).
2. Sau-exclusiv pe 32 de biţi se obţine cu ajutorul a patru operaţii sau-exclusiv pe 8 biţi (XRL).
3. Ridicarea la puterea 2 se poate calcula folosind şase multiplicări de 8 biţi cu 8 biţi (MUL) şi 11 adunări cu transport (ADDC). De menţionat că sunt suficiente şase multiplicări deoarece avem nevoie doar de cei mai puţin semnificativi 32 de biţi din cei 64 ai produsului care se obţine.
4. Rotaţia la stânga cu cinci poziţii a unui cuvânt de 32 de biţi se poate calcula prin rotaţia cuvântului la dreapta cu o poziţie de trei ori şi permutarea apoi a patru byţi. Rotaţia la dreapta cu un bit a cuvântului se poate obţine prin rotaţii la dreapta cu patru byţi cu rotaţie (RRC).
5. Rotaţia la stânga cu r biţi a unui cuvânt de 32 de biţi se poate calcula prin rotirea cuvântului la stânga sau la dreapta cu un bit de r' ori ($r' \leq 4$, cu media doi) şi permutarea lui apoi cu patru byţi.Cei mai puţin semnificativi cinci biţi a lui r sunt utilizaţi pentru a determina r' şi permutarea care poate fi controlată prin utilizarea salturilor (JB).

6. Majoritatea instrucțiunilor necesită un ciclu, cu excepția multiplicării care necesită patru cicluri şi salturile care necesită două cicluri.

Pe baza celor enumerate se poate estima numărul total de cicluri necesare pentru o rundă a lui RC6, aşa cum se vede şi în tabelul 2.8.

Ținând cont de instrucțiunile de adresare, de depăşirile suplimentare se poate estima că pentru criptarea unui bloc de date cu cifrul RC6 sunt necesare aproximativ (174 × 20) × 4 = 13920 de cicluri. Având în vedere că pentru procesoarele Intel MCS 51 fiecare ciclu necesită o microsecundă se poate estima că viteza de criptare a lui RC6 este de aproximativ (1000000/13920)×128 = 9,2 Kbiti/sec. Acest rezultat este confirmat şi de implementarea făcută pe procesorul Intel 8051 unde numărul de cicluri necesare criptării unui bloc de date este de 13535 de cicluri.

Tabelul 2.8. Calculul numărului de cicluri al unei runde a cifrului RC6 în implementările pe procesoare pe 8 biți

Operație	Instrucțiune	Cicluri/operație	Total cicluri
Adunare	4 ADDC	4	4 × 6 = 24
Sau-exclusiv	4 ×RL	4	4 × 2 = 8
Ridicare la pătrat	6 MUL, 11 ADDC	35	35 ×2 = 70
Rotație la stânga cu 5	12 RRC	12	12 × 2 = 24
Rotație la stânga cu r (media lui r)	8RRC sau RLC, 8 JB	24	24 × 2 = 48
Total			**174**

În privința operației de setare a cheii se pot face de asemenea o serie de observații. Bucla dominantă din procesul de setare a cheii este ultima buclă **for** (figura 2.1). Pentru b = 16, 24, 32 şi r = 20, numărul de iterații din buclă este dat de relația $v = 3\times\max\{20\times2+4, b/4\} = 132$, care este independent de b. Deci toate estimările vor fi adevărate pentru toate lungimile de chei care corespund cerințelor AES. Fiecare iterație din buclă utilizează patru adunări de 32 de biți, o rotație la stânga cu trei şi o rotație variabilă la stânga cu r poziții. În plus mai există câteva operații care vor fi incluse ca depăşiri. Urmând o analiză asemănătoare pentru criptare se obține numărul total de cicluri pentru fiecare iterație (ignorând instrucțiunile de adresare) este 52. Făcând din nou o estimare în ce priveşte depăşirile suplimentare, se obține: (52×132)×4 = 27456 cicluri pentru setarea unei chei de 128, 192 sau 256 de biți, adică 27 milisecunde pe un procesor Intel MCS 51.

2.4. Estimări hardware ale lui RC6

Pentru majoritatea aplicațiilor implementarea soft este cea mai potrivită întrucât la ora actuală operațiile elementare (adunarea, scăderea, multiplicarea sau-exclusiv şi rotațiile) sunt uşor realizabile în procesoarele moderne. Totuşi în anumite

situații este posibil să apară necesitatea unei implementări dedicate a lui RC6. Acest lucru se poate întâmpla dacă se cere să se atingă viteza maximă realizabilă sau să se integreze alte funcții pe lângă algoritmul RC6.

Având în vedere operațiile elementare menționate, folosite de RC6, se poate profita de experiența existentă în domeniul proiectării modulelor de circuit care implementează aceste primitive. De exemplu RC6 poate fi implementat pe de o parte utilizând tehnologia standard de tip matrici de porți sau s-ar putea beneficia de efortul investit deja în proiectarea unui circuit eficient de multiplicare. O implementare de tip matrici de porți ar putea avea performanțe mai slabe decât implementarea bazată pe procesor. Dar acesta nu este o situație de excepție şi circuitele care încorporează cele mai bune circuite multiplicatoare disponibile pot fi uşor proiectate ca submodule.

Pentru o implementare dedicată sau semidedicată cei mai importanți parametri sunt suprafața de siliciu, viteza şi consumul redus de putere a unei multiplicări de 32×32 întregi. Rezultatele investigațiilor în acest sens au pus în evidență următoarele date:

- o suprafață de 120×100 microni în suprafața cu un proces CMOS standard de 0,25 microni
- aproximativ 3 ns pentru fiecare multiplicare
- un consum de putere de cinci miliwați

Se poate estima că o rotația unei variabile de 32 de biți ("barrel shifter") va ocupa jumătate din suprafața unui multiplicator şi o nanosecundă pentru execuție. De asemenea se poate estima că un sumator complet va necesita un sfert din suprafața multiplicatorului şi o nanosecundă pentru execuție. De asemenea se mai poate observa că funcția $f(x) = x\ (2x + 1) \pmod{2^w}$ se poate calcula utilizând doar un multiplicator care returnează ultimii 32 de biți din cei 64 ai produsului. Un astfel de multiplicator parțial ocupă aproximativ 60% din suprafața unui multiplicator complet şi necesită 3 nanosecunde pentru execuție. Pentru o cale critică a lui RC6 putem estima datele din tabelul 2.9.

Tabelul 2.9. Timpul şi suprafața necesară implementării lui RC6

Operație	Timp (ns)	Suprafața (mm^2)
32×32 multiplicator partial	3	0,007
Sau-exclusiv 32 biți	0	0,000
"barrel shifter" 32 biți	1	0,006
Adunare cu propagarea transportului 32 biți	1	0,003
Total	**5**	**0,016**

Pentru o implementare eficientă sunt necesare două astfel de circuite într-un chip. Ca urmare suprafața totală de siliciu va fi de aproximativ 0,032 mm^2 pentru partea corespunzătoare lui RC6. La aceasta se mai adaugă aproximativ 0,018 mm^2 pentru control I/O, etc. Astfel putem considera că suprafața calculată este de 0,05

mm^2. Presupunând că puterea consumată este proporțională cu suprafața putem afirma că ea va fi de aproximativ 21 mW.

Dacă numărul de runde pe bloc este de 20 timpul total necesar pentru criptare este de 5×20=100 ns pentru fiecare bloc, ceea ce înseamnă că rata estimată va fi de aproximativ 1,3 Gbiți/s. Ne aşteptăm ca timpul necesar decriptării să fie similar cu cel de criptare şi ca acestea să fie independente de lungimea cheii furnizate de către utilizator.

Aşa cum se observă şi din descrierea algoritmului, RC6 este deosebit de compact. Se estimează că o implementare în limbaj de asamblare rapid pe un microprocesor Intel Pentium Pro se poate scrie uşor folosind mult sub 256 byți de cod pentru setarea cheii, criptarea blocului sau decriptarea acestuia.

Spre deosebire de alți algoritmi, RC6 nu utilizează tabele de căutare pe parcursul criptării. Aceasta înseamnă că codul RC6 şi data pot fi uşor încărcate în memoria cache on-chip de astăzi, făcând astfel economie de spațiu. Criptarea şi decriptarea utilizează o listă de chei de 176 byți şi un minim de memorie suplimentară. Pentru calcularea listei de chei de 176 byți, procesul de setare al cheii necesită puțin mai mult decât un şir auxiliar de aproximativ aceaşi mărime cu cheia furnizată de utilizator. În plus, deoarece lista cheilor este de doar 176 byți este posibilă precalcularea şi memorarea listelor de chei pentru sute de chei. Apoi alegerea uneia dintre ele înseamnă doar comutarea pointer-ului la lista cheii relevante.

2.5. Securitate şi simplitate

Cele mai importante obiective urmărite pe parcursul proiectării cifrului RC6 au fost securitatea, simplitatea şi performanțele bune.

Cifrul RC5 s-a remarcat prin simplitate şi ca urmare a fost un obiect de studiu atractiv pentru mulți cercetători. Fiind accesibil atât pentru o analiză primară cât şi una sofisticată, mulți cercetatori au încercat să-l studieze din punct de vedere al securității pe care o oferă [KaYi-95b]. Cifrul bloc RC6 a fost elaborat pe baza experienței acumulate în proiectarea lui RC5, a simplității şi securității acestuia. Punctele comune precum şi îmbunătățirile aduse noului candidat AES pot fi puse în evidență dacă se urmăresc următorii paşi din proiectarea acestuia:

1. Se începe cu bucla de bază a semirundei lui RC5:

```
for i=1 to r do
    {
      A=((A⊕B)<<<B)+S[i]
      (A,B)=(B,A)
    }
```

2. Execută două copii a lui RC5 în paralel: una în regitrele A,B şi cealaltă în registrele C,D.

```
for i=1 to r do
    {
      A=((A⊕B)<<<B)+S[2i]
      C=((C⊕D)<<<D)+S[2i+1]
      (A,B)=(B,A)
      (C,D)=(D,C)
    }
```

3. La nivelul de interschimbare, în loc de a schimba pe A cu B şi pe C cu D, se permută registrele (A,B,C,D)=(B,C,D,A), astfel încât calculul AB este mixat cu calculul CD. La acest nivel bucla internă arată astfel:

```
for i=1 to r do
    {
      A=((A⊕B)<<<B)+S[2i]
      C=((C⊕D)<<<D)+S[2i +1]
      (A,B,C,D)=(B,C,D,A)
      }
```

4. Se mixează mai mult calculul lui AB cu cel a lui CD prin inversare în cele două formule de calcul, în locul în care apar rotaţiile.

```
for i=1 to r do
    {
      A=((A⊕B)<<<D)+S[2i]
      C=((C⊕D)<<<B)+S[2i +1]
      (A,B,C,D)=(B,C,D,A)
      }
```

5. În loc de a utiliza B şi D într-o manieră simplă ca mai sus, se utilizează o versiune transformată a acestor registre, pentru anumite transformări potrivite. Pentru o bună securitate, rotaţiile dependente de date care derivă din acestă transformare trebuie să depindă de toţi biţii cuvântului de intrare iar transformarea să asigure o bună mixare în interiorul cuvântului. Transformarea aleasă pentru cifrul bloc RC6 este funcţia $f(x)=x(2x+1)(\text{mod } 2^w)$ urmată de o rotaţie la stânga cu 5 poziţii. Această funcţie cu primitive simple şi uşor de implementat pe majoritatea procesoarelor modeme oferă

securitate cifrului RC6. Se observă că *f(x)* este modulo 2^w unu-la-unu şi biţii cei mai semnificativi ai lui *f(x),* care determină numărul de rotaţii folosit, depind de toţi biţii lui x. Se obţine:

for *i* = 1 **to** *r* **do**
{
t = (B×(2B + 1)) ⋘ 5
u = (D×(2D + 1)) ⋘ 5
A = ((A ⊕ *t*) ⋘ *u*) + S[2*i*]
C = ((C ⊕ *u*) ⋘ *t*) + S[2*i* + *1*]
(A,B,C,D) = (B,C,D,A)
}

6. La începutul şi sfârşitul celor r runde se adaugă paşi de "pre-whitening" respectiv "post-whitening". Fără aceşti paşi, textul clar dezvăluie parte din început primei runde de criptare iar textul cifrat dezvăluie parte din începutul ultimei runde de criptare. Paşii de pre şi post "whitening" au rolul de a masca (ascunde) acestea şi a ne lăsa cu RC6:

B = B + S[0]
D = D + S[1]
for *i* = 1 **to** *r* **do**
{
t = (B×(2B + 1)) ⋘ 5
u = (D×(2D + 1)) ⋘ 5
A = ((A ⊕ *t*) ⋘ *u*) + S[2*i*]
C = ((C ⊕ *u*) ⋘ *t*) + S[2*i* + *1*]
(A,B,C,D) = (B,C,D,A)
}
A = A + S[2*r* + 2]
C = C + S[2*r* + 3]

Cu toate că evoluţia de la RC5 la RC6 ar putea părea simplă, ea de fapt a complicat proiectarea şi analiza mulţimii alternativelor. RC6 este un cifru care îmbină efectiv cele trei deziderate: securitate, simplitate şi performanţe.

2.6. Atacuri criptanalitice

Majoritatea rezultatelor de criptanaliză ale lui RC5 [BiKu-98], [KaYi-95a], [KnMe-96], [Selc-98] depind de ceea ce se poate spune că este o avalanşă lentă de schimbări între runde. Adunarea întregilor furnizează un număr rezonabil de schimbări datorate efectului transportului, dar cele mai dramatice schimbări au loc când sunt apelate două numere diferite de rotaţii în puncte similare ale criptării a două texte clare.

Schimbările făcute cifrului RC5 pentru obţinerea lui RC6 au fost menţionate mai sus. Cele mai importante două schimbări făcute sunt introducerea funcţiei pătratice B×(2B+1) (respectiv, D×(2D+1)) şi a rotaţiei fixe cu cinci biţi. Funcţia pătratică are scopul de a determina o rată mai mare a difuziei, mărind astfel şansele ca simple diferenţiale să distrugă numărul de rotaţii. Valorile pătratice transformate ale lui B şi D sunt utilizate în locul lui B şi D pentru a modifica registrele A şi C, mărind neliniaritatea schemei. Rotaţia fixă cu cinci biţi are rolul simplu dar important de a complica criptanaliza liniară şi diferenţială.

Cea mai indicată metodă de atac a cifrului RC6, pe care o au la dispoziţie criptanaliştii, este căutarea exhaustivă a cheii de criptare de b byţi (sau a şirului S[0,...,43] dacă cheia furnizată de utilizator este prea lungă). Aceasta presupune un effort egal cu $\min\{2^{8b},2^{1048}\}$ operaţii.

Metodele mai avansate de atac cum sunt criptanaliza diferenţială [BiSh-93] şi liniară [Mats-94] sunt posibile doar în versiunile cu un număr mic de runde ale cifrului, nu şi în cazul cifrului RC6 cu 20 de runde. Dificultatea majoră constă în găsirea caracteristicilor iterative sau aproximărilor liniare bune care ar putea sta la baza unor atacuri.

Pentru a-şi atinge scopul, aceste atacuri ncesită un număr mare de date, iar obţinerea a 2^a blocuri de perechi de tip text clar - text cifrat, cunoscute sau alese, este cu totul altceva faţă de încercarea de reface o cheie din cele 2^a posibilităţi. Este important de observat că cu un cifru care rulează cu o rată de un terabit/secundă (adică o criptare a datei la o rată de 10^{12}biţi/s), timpul necesar pentru 50 de calculatoare care lucrează în paralel la criptarea a 2^{64} blocuri de date este mai mare de un an; pentru a cripta 2^{80} blocuri de date este nevoie de mai mult de 98.000 de ani; pentru a cripta 2^{128} blocuri de date sunt necesari 10^{19} ani. Pentru ca un atac să aibă succes, în termeni practici, sunt suficiente 2^{64} blocuri de date. RC6 oferă în schimb un nivel mult mai ridicat de securitate. Dacă numărul blocurilor de date necesare asigurării securităţii se va dovedi că poate fi mai mic decât 2^{128}, atunci numărul de runde ale cifrului RC6 poate fi scăzut de la 20 (număr sugerat) iar acest lucru ar avea efectul creşterii performanţelor.

În tabelul 2.10 sunt prezentate datele necesare unui atac liniar sau diferenţial de bază împotriva cifrului RC6. O analiză sumară a descrierii procesului de decriptare indică faptul că proprietăţile şi structura acestuia sunt asemănătoare cu cele ale criptării. Atacurile şi consideraţiile referitoare la securitatea criptării par să fie aceleaşi şi pentru decriptare. În continuare vor fi trecute în revistă o serie de atacuri şi va fi prezentată o analiză a variantelor simplificate ale lui RC6.

Tabelul 2.10. Prezentarea celor mai bune atacuri criptanalitice diferențiale şi liniare

Atacuri	Număr de runde				
	8	12	16	20	24
Criptanaliza diferențială	2^{56}	2^{117}	2^{190}	2^{238}	2^{299}
Criptanaliză liniară	2^{47}	2^{83}	2^{119}	2^{155}	2^{191}

În tabelul 2.10 sunt evidențiate în chenar situațiile în care datele necesare unui atac reuşit depăşesc 2^{128}, numărul total al textelor clare posibile. Se poate spune că RC6 cu 20 de runde este sigur la atacurile criptanalitice liniare şi diferențiale.

2.6.1. Criptanaliza diferențială

Criptanaliza diferențială este un atac cu text clar ales asupra cifrurilor bloc iterative. Prin alegerea perechilor de texte clare care se criptează, diferențele dintre intrări în runda finală a cifrului pot fi prevăzute cu o anumită probabilitate. Definirea termenului de diferență este dependentă de cifrul bloc aflat sub atac. Scopul este ca diferențele prezise dintre perechile care intră în runda finală să poată fi folsite alături de diferențele din perechea de texte cifrate pentru a deduce informații referitoare la subcheile utilizate în runda finală ale cifrului [BiSh-93]. Probabilitatea ca o pereche de texte clare alese să furnizeze diferența dorită la sfârşitul rundei (r-1) va descreşte pe măsură ce r creşte.

Astfel de atacuri pe versiuni reduse ale cifrurilor iterative pot fi efective, în timp ce versiunea completă a aceluiaşi cifru, cu un număr suficient de runde, să ofere siguranță. Alegerea optimă a diferențelor în perechile de text clar se determină prin investigarea caracteristicilor; acestea vor specifica diferențele prevăzute pentru fiecare rundă a cifrului.

În studiile teoretice [LaMM-94] referitoare la aplicabilitatea atacurilor diferențiale diferitelor tipuri de cifruri iterative s-a introdus noțiunea de "diferențiale", care reprezintă o versiune mai largă a caracteristicii; sunt specificate doar diferențele de intrare iar cele de ieşire nu sunt luate în considerare.

Există o serie de variante ale atacului diferențial de bază. În acest paragraf sunt relatate pe scurt cocluziile analizelor referitoare la eficiența lor precum şi influența acestora asupra cifrului RC6. Knudsen a introdus noțiunea de diferențială trunchiată (numită formal diferențială parțială) [Knud-94]. În acest caz, criptanalistul prezice comportarea unei părți a diferenței, nu a întregii diferențe de 128 de biți. Diferențiala trunchiată dă o flexibilitate mai mare în atacul unui cifru, deşi uneori aplicația lui pare limitată. Există cifruri care sunt vulnerabile la analiza cu diferențiale trunchiate, dar rareori acest atac, sau variante ale lui, au succes mai mare decât metodele mai serioase. Ca o analogie la diferențierea în calcul, Lai a introdus atacurile diferențiale de nivel înalt [Lai-92]. Pe de altă parte Knudsen a demonstrat că

pot fi construite cifruri care să fie vulnerabile la atacurile diferențiale de nivel înalt, dar rezistente la atacurile diferențiale convenționale [Knud-95]. De asemenea, s-a demonstrat că atacurile interpolate [JaKn-97] pot fi folosite împotriva unora dintre cifrurile existente. În ori ce caz, aceste atacuri sunt limitate atunci când sunt aplicate cifrurilor mai sofisticate cu un număr rezonabil de runde. Acesta este şi cazul cifrului RC6. În fine, în domeniul criptanalizei diferențiale trebuiesc luate în considerare şi aşa numitele cvasi-diferențiale [KnBe-96], care ar putea pune în evidență unele incompatibilități cu care proiectantul a încercat să împiedice criptanaliza. Cele mai evidente două noțiuni de diferență ale lui RC6 sunt sau-exclusiv şi scaderea întregilor.

Studiile laboratoarelor RSA [RBSY-98] au concluzionat că cifrul bloc RC6 oferă o rezistență bună la criptanaliza diferențială şi că cele 20 de runde propuse sunt suficiente pentru respectarea cerințelor AES. De asemenea s-a demonstrat că funcția pătratică combinată cu rotația fixă împiedică construirea unei diferențiale efective.

Pentru o justificare a celor afirmate mai sus, în continuare am enunțat şi demonstrat un număr de 9 leme care stau la baza diverselor tipuri de atacuri criptanalitice diferențiale.

Funcția pătratică

Introducerea funcției pătratice reprezintă cea mai importantă îmbunătățire adusă în procesul de trecere de la cifrul RC5 la cifrul RC6. Scopul principal al securității este de a face ca numărul de rotații dependent de date (număr care derivă din ieşirea funcției pătratice) să fie dependent de toți biții cuvântului de intrare. Acest lucru ar elimina atacurile diferențiale existente asupra cifrului RC5.

Pentru a analiza funcția pătratică sub aspectul diferențial trebuieşte aleasă o măsură pentru diferență. Întrucât toate operațiile implicate sunt multiplicări şi adunări întregi, pare evidentă alegerea scăderii întregilor ca măsură pentru diferență.

Prin următoarele patru leme demonstrate se vor arăta proprietățile de bază ale funcției pătratice. Vom arăta că funcția $f(x) = x\,(2\,x + 1) \bmod 2^{32}$ este o permutație prin demonstrarea unor rezultate mai generale.

Lema 1. Fie $f(x) = x\,(ax + b) \bmod 2^{32}$, unde a este par şi b este impar. Atunci $f(x)$ este o mapare unu-la-unu din $\{0,1\}^w$ la $\{0,1\}^w$.

Demonstrație. Prin contradicție. Să presupunem că pentru $x_1 \neq x_2$, $f(x_1) = f(x_2)$. Apoi $a\,x_1^2 + b\,x_1 = a\,x_2^2 + b\,x_2 \bmod 2^w$. Combinând termenii obținem:

$$(x_1 - x_2)\cdot(a\,x_1 + a\,x_2 + b)\cdot \bmod 2^w = 0$$

Deoarece $(a\,x_1 + a\,x_2 + b)$ este impar $(x_1 - x_2)$ trebuie să fie un multiplu a lui 2^w. Aceasta este o contradicție.

Vom considera acum câteva proprietăți diferențiale de bază ale funcției pătratice. Pentru două intrări x_1 şi x_2, fie $y_1 = f(x_1)$ şi $y_2 = f(x_2)$ şi definim

$$\delta_x = x_2 - x_1$$

$$\delta_y = y_2 - y_1$$

Lema 2. Pentru intrările x_1 şi x_2, fie $y_1 = f(x_1)$ şi $y_2 = f(x_2)$. Definim $\delta_x = x_2 - x_1$ şi $\delta_y = y_2 - y_1$. *Atunci*

$$\delta_y = (4x_1\delta_x + \delta_x + 2\delta_x^2) \bmod 2^{32}$$

Demonstrație. Evaluarea simplă a lui $f(x_1)$ şi $f(x_2)$.

Lema 3. Dându-se notația din Lema 2, $\delta_y = \delta_x$ dacă şi numai dacă $4x_1\delta_x + 2\delta_x^2 = 0 \bmod 2^{32}$.

Demonstrația rezultă din Lema 2.

Lema 4. Dându-se o intrare x_1 aleasă aleator din $\{0,1\}^{32}$, fie $g_{i,j}$ probabilitatea ca bitul i a lui x_1 să întoarcă bitul j a lui $y_1 = f(x_1)$. Atunci,

$$g_{i,j} = \begin{cases} o & \text{pentru } j < i \\ 1 & \text{pentru } j = i \\ 1 & \text{pentru } j = 1 \text{ si } i = 0 \end{cases}$$

$g_{i,j} \in [1/4, 3/4]$ pentru $j > i \geq 1$ sau $j \geq 2$ şi $i = 0$.

Pentru ultimul caz, $g_{i,j}$ este apropiat de 3/4 dacă $j = 2i + 2$, şi pentru majoritatea celorlalte perechi i,j, $g_{i,j}$ este apropiat de 1/2.

Demonstrație. Fie $x_2 = x_1 \oplus 2^i$ şi fie $y_2 = f(x_2)$. Fără a pierde din generalitate, presupunem că $x_1[i] = 0$, de exemplu, al i-lea bit a lui x_1 este zero. Din lema 2 rezultă:

$$\delta_y = y_2 - y_1 = 2^{i+2} \cdot x_1 + 2^i + 2^{2i+1} \bmod 2^{32} \quad (1)$$

$$\approx 2^{i+2} \cdot x_1 + 2^{2i+1} \bmod 2^{32} \quad (2)$$

$$\approx 2^{i+2} \cdot x_1 \bmod 2^{32} \quad (3)$$

Să considerăm fiecare caz al lemei, în parte.

Cazul 1. $j < i$.

Din ecuația (1) se observă că bitul j a lui δ_y este zero. Astfel bitul i a lui x_1 nu va întoarce bitul j a lui $f(x_1)$.

Cazul 2. $j = i$.

Tot din ecuația (1) se observă că bitul i a lui δ_y este unu. Deci bitul i a lui x_1 întoarce bitul i a lui $f(x_1)$.

Cazul 3. $j = 1$ şi $i = 0$.

Din ecuația (1), $\delta_y = 4 \cdot x_1 + 1 + 2$. De aici: $x_1[0] = 0$, $y_1[0] = 0$. Din acest motiv, când se calculează $y_2 = y_1 + \delta_y$, nu apare transport de la bitul 0 la bitul 1. Deci bitul 1 a lui δ_y este 1, bitul 0 a lui x_1 va întoarce întotdeauna bitul 1 a lui $f(x_1)$.

Cazul 4. $j > i \geq 1$ sau $j \geq 2$ şi $i = 0$.

Să considerăm mai întâi cazul în care $j = i + 1$. Din ecuaţia (1) se observă că bitul $i + 1$ a lui δ_y este întotdeauna zero şi bitul i a lui δ_y este întotdeauna 1. Când se calculează $y_2 = y_1 + \delta_y$, apare transport spre bitul $i+1$ cu probabilitatea de aproximativ 1/2, şi deci $g_{i,i+1} \approx 1/2$.

Considerăm cazul în care $j = 2i + 2$. Deoarece $x_1[i] = 0$, bitul $2i + 2$ a lui $2^{2i+2}\,x$ este zero. Analizăm efectul transportului când se calculează $y_2 = y_1 + \delta_y$ folosind aproximaţia 2. Dacă $x_1[i-1] = 1$, atunci există întotdeauna transport spre bitul $2i + 2$. Dacă $x_1[i-1] = 0$, atunci există transport spre bitul $2i + 2$ cu probabilitatea de 1/2. În general, există transport cu probabilitatea de 3/4 şi astfel $g_{i,j} \approx 3/4$.

Dacă $j \geq i + 2$ şi $j \neq 2i + 2$, folosind o aproximare de 3, obţinem că bitul j a lui δ_y este aleator şi x_1 este aleator. Astfel bitul i a lui x_1 întoarce bitul j a lui $f(x_1)$ cu o probabilitate de 1/2. Valorile inexacte se datorează prezenţei efectului transportului.

Rezultatele experimentale au demonstrat că $g_{i,j}$ variază între 1/4 şi 3/4, şi majoritatea probabilităţilor sunt apropiate de 1/2 dacă $i \geq 16$.

Lema 4 demonstrează că şi o schimbare a unui singur bit poate schimba cel puţin un bit dintre cei mai semnificativi biţi care vor fi utilizaţi ca mărime a rotaţiei. Acest lucru va ajuta la obţinerea unei avalanşe rapide a schimbării.

Utilizarea scăderii întregi ca măsură a diferenţei

Vom studia caracteristicile funcţiei pătratice de forma $\delta_y = \delta_x$. Aceste caracteristici vor fi numite caracteristici statice ale funcţiei pătratice.

Lema 5. Dacă δ_x este impar atunci caracteristica $\delta_y = \delta_x$ are probabilitatea zero.

Demonstraţie. Deoarece δ_x este impar, $2\cdot\delta_x = 2 \bmod 4$. Deci $4\cdot x_1\cdot\delta_x + 2\cdot\delta_x^2$ nu este niciodată egal cu zero modulo 2^{32}.

Lema 6. Dacă $\delta_x = v\cdot 2^i$ pentru anumiţi v întregi impari şi $1 \leq i \leq 30$, atunci caracteristica $\delta_y = \delta_x$ are probabilitatea 2^{i-30}. Dacă $\delta_x = 2^{31}$, atunci caracteristica $\delta_y = \delta_x$ are probabilitatea 1.

Demonstraţie. Din Lema 3, $\delta_y = \delta_x$ dacă şi numai dacă $4\cdot x_1\cdot v\cdot 2^i + 2(v\cdot 2^i)^2 = 0 \bmod 2^{32}$. Pentru $1 \leq i \leq 15$ aceasta este echivalentă cu $x_1\cdot v\cdot 2^{i-1} = 0 \bmod 2^{30-i}$ şi pentru $16 \leq i \leq 30$ este echivalent cu $x_1 = 0 \bmod 2^{30-i}$. Se observă că în fiecare caz cei mai semnificativi $(i+2)$ biţi ai lui x_1 pot avea orice valoare. Singura restricţie impusă este cea asupra ultimilor $(30-i)$ biţi ai lui x_1. Din acest motiv, calculând media tuturor valorilor posibile ale lui x_1, probabilitatea ca $\delta_y = \delta_x$ este de:

$$\frac{2^{i+2}}{2^{32}} = 2^{i-30}$$

Aceste ultime două leme enunţate acoperă toate caracteristicile statice posibile şi arată că probabilitatea unei astfel de caracteristici depinde de diferenţa de intrare,

δ_x. Este important de observat că pentru aceste caracteristici, probabilitatea menținerii caracteristicii scade rapid când *i* se apropie de valorile care semnifică pozițiile biților mai puțin semnificativi. Această proprietate a funcției pătratice este importantă în reducerea atacurilor criptanalitice diferențiale.

Utilizarea lui sau-exclusiv ca măsură a diferenței

Definim

$$\delta_x^{\oplus} = x_2 \oplus x_1$$

$$\delta_y^{\oplus} = y_2 \oplus y_1$$

$$p_i = \begin{cases} 1 & \text{pentru } i = 31 \\ 2 & \text{pentru } 15 \le i \le 20 \\ 0 & \text{pentru } 0 \le i \le 14 \end{cases} \quad (4)$$

Lema 7. Fie p_i probabilitatea caracteristicii $\delta_y^{\oplus} = \delta_x^{\oplus} = 2^i$. Atunci

Şi $$p_i \approx 2^{i-31} \text{ pentru } 21 \le i \le 30. \quad (5)$$

Demonstrație. Trebuiesc considerate două situații când avem $\delta_x^{\oplus} = 2^i$. Fie:

(a) $x_2 = x_1 + 2^i$ şi $x_1[i] = 0$ şi

(b) $x_1 = x_2 + 2^i$ şi $x_2[i] = 0$

Deoarece x_1 este uniform distribuit, fiecare caz poate apărea cu probabilitatea 1/2. Considerăm doar cazul (a) şi vom analiza diferite situații.

Cazul 1. $16 \le i \le 30$.

Pe baza lemei 2, avem:

$$y_2 - y_1 = 2^{i+2} \cdot x_1 + 2^i + 2^{2i+1} \bmod 2^{32}$$
$$= 2^{i+2} \cdot x_1 + 2^i \bmod 2^{32} \quad \text{(deoarece } i \ge 16\text{)}.$$

Se observă că $\delta_y^{\oplus} = 2^i$ dacă şi numai dacă sunt valabile următoarele relații:

A $2^{i+2} \cdot x_1 = 0 \bmod 2^{32}$

B $y_1[i] = 0$.

Urmărind argumentele din Lema 6, se obține

$$\text{prob (A)} = 2^{i-30}.$$

Din acest motiv probabilitatea caracteristicii $\delta_y^{\oplus} = \delta_x^{\oplus} = 2^i$ este egală cu

$$p_i = \text{prob (A)} \times \text{prob (B| A)} = 2^{i-30} \times \text{prob (B| A)}$$

Fie q_i = prob (B| A). Vom arăta că $q_i = 1$ dacă $16 \le i \le 20$, şi că $q_i \approx 1/2$ dacă $21 \le i \le 30$. Acest lucru va fi suficient pentru a demonstra acest caz.

Dacă $16 \le i \le 20$, atunci A implică ca cei mai puțin semnificativi (30 – i) biți ai lui x_1 să fie 0. Deci cei mai puțin semnificativi $2(30 - i)+1 = 61 - 2i$ biți ai lui $2x_1^2$ sunt zero. Deoarece $y_1 = 2x_1^2 + x_1$ acest lucru se va întâmpla cu probabilitatea 1/2, iar bitul i face parte din acest domeniu dacă $21 \le i \le 30$. Deoarece $y_1 = 2 \cdot x_1^2 + x_1$ acest lucru se va întâmpla cu probabilitatea 1/2 , $y_1[i] = x_1[i] = 0$.

Această aproximare se datorează faptului că fiecare bit a lui x^2 nu este uniform distribuit chiar dacă x este uniform distribuit. Valorile lui p_i pentru $21 \le i \le 30$ sunt date de următorul tabel:

Tabelul 2.11. Valorile parametrului p_i

i	2^{31-i}	$1/p_i$
21	1024	819,200000
22	512	455,111111
23	256	248,242424
24	128	126,025089
25	64	63,750731
26	32	31,938838
27	16	15,992672
28	8	7,998251
29	4	3,999823
30	2	1,999954

Cazul 2. $i = 15$.

Acest caz urmează în esență aceaşi argumentare ca şi în cazul 1.

Având $\delta_x = 2^{15}$, din Lema 2 se obține:

$$y_2 - y_1 = 2^{17} \cdot x_1 + 2^{15} + 2^{31} \bmod 2^{32}$$

Se observă că $\delta_y^{\oplus} = 2^{15}$ dacă şi numai dacă sunt adevărate relațiile:

A $\quad 2^{17} \cdot x_1 - 2^{31} = 0 \bmod 2^{32}$

B $\quad y_1[15] = 0.$

Urmărind argumentele din Lema 6, se obține:

$$\text{prob (A)} = 2^{-15}$$

Din acest motiv probabilitatea caracteristicii $\delta_y^{\oplus} = \delta_x^{\oplus} = 2^{15}$ este egală cu

$$p_{15} = \text{prob (A)} \times \text{prob (B| A)} = 2^{-15} \times \text{prob (B| A)}$$

Observăm că relația A implică ca cei mai puțin semnificativi 14 biți ai lui x_1 să fie 0 şi ca bitul 14 să fie 1. Deci, cei mai puțin semnificativi (2 × 14) + 1 = 29 de biți ai lui $2 \cdot x_1^2$ sunt 0. Deoarece $y_1 = 2 \cdot x_1^2 + x_1$, avem:

$$y_1[15] = x_1[15] = 0.$$

Din acest motiv: $p_{15} = 2^{-15} = 2^{i-30}$.

Cazul 3. $1 \leq i \leq 14$.

În acest caz:

$$y_2 - y_1 = 2^{i+2} \cdot x_1 + 2^{2i+1} + 2^i \bmod 2^{32}$$

Acum, $y_2 - y_1$ poate fi 1 dacă şi numai dacă $x_1 = -2^{i-1} \bmod 2^{30-i}$.

Această condiție ne arată că biții de la $(i - 1)$ până la $(29 - i)$ ai lui x_1 trebuie să fie 1. Dar aceasta contrazice presupunerea inițială ca $x_1[i] = 0$.

Cazul 4. $i = 0$ sau $i = 31$.

Pentru $i = 0$ rezultatul este dat de Lema 5. Pentru $i = 31$ rezultatul este simplu deoarece scăderea întreagă şi sau-exclusiv sunt aceleaşi când diferența este bitul cel mai semnificativ.

Rezultatele Lemei 7 sunt surprinzătoare. În paragraful următor se va argumenta de ce se preferă utilizarea scăderii întregi ca măsură a diferenței în locul lui sau-exclusiv, când se analizează cifrul RC6.

Pe baza Lemei 6 şi a Lemei 7 se pot compara probabilitățile caracteristicilor statice de tip bit-singular ale funcției pătratice, folosind cele două măsuri. Rezultatele sunt date în tabelul 2.12.

Din acest tabel reiese că probabilitățile caracteristicilor sunt uşor mai mari când se foloseşte scăderea întreagă decât atunci când se foloseşte sau-exclusiv. Dar mai important este faptul că atunci când i are valori mici ($i \leq 14$), se poate utiliza doar scăderea întragă deoarece probabilitățile pentru caracteristicile relevante sunt 0 dacă se foloseşte sau-exclusiv.

Înainte de a concluziona că scăderea întreagă este cea care trebuie folosită în analiza lui RC6, mai trebuie ținut cont de un factor, şi anume dacă măsura aleasă a diferenței este potrivită şi în alte tipuri de operații prezente în RC6.

Tabelul 2.12. Probabilități ale caracteristicilor statice bit-singular ale funcției pătratice.

Poziția i a bitului	Probabilitatea $\delta_y = \delta_x = 2^i$	Probabilitatea $\delta_y^{\oplus} = \delta_x^{\oplus} = 2^i$
31	1	1
$21 \leq i \leq 30$	2^{i-30}	$\approx 2^{i-31}$
$15 \leq i \leq 20$	2^{i-30}	$\approx 2^{i-30}$
$1 \leq i \leq 14$	2^{i-30}	0
0	0	0

În particular, voi analiza modul de propagare prin sau-exclusiv a diferenței bit-singular. Funcția pătratică $f(x)$ are o comportare diferențială complicată şi este mai indicat să se studieze funcția $w = z \oplus f(x)$ ca o componentă singulară.

Pentru două seturi de intrări x_1, z_1 şi x_2, z_2 definim:

$$w_1 = z_1 \oplus f(x_1) \quad \text{şi } w_2 = z_2 \oplus f(x_2)$$

Setăm $\delta_x = x_2 - x_1$, $\delta_z = z_2 - z_1$, şi $\delta_w = w_2 - w_1$.

Lema 8. Fie p_i probabilitatea caracteristicii $(\delta_x, \delta_z) = (2^i, 2^i)$ şi $\delta_w = 0$. Atunci,

$$p_i = q_i = \begin{cases} 2^{i-3} & \text{pentru } 15 \le i \le 31 \\ x \in \left[2^{i-35}, 2^{i-35}\right] & \text{pentru } 0 \le i \le 14 \end{cases}$$

Această lemă a fost demonstrată experimental iar rezultatele au demonstrat că p_i este exact 2^{i-31} pentru $15 \le i \le 31$. Rezultatele experimentale pentru valori mai mici ale lui i sunt date în tabelul 2.13:

Tabelul 2.13. Rezultatele experimentale ale lemei 8 pentru valori mici ale lui i

i	2^{31-i}	$1/p_i$	i	2^{31-i}	$1/p_i$
0	2^{31}	$2^{30.10}$	1	2^{30}	$2^{29.15}$
2	2^{29}	$2^{28.57}$	3	2^{28}	$2^{28.17}$
4	2^{27}	$2^{27.71}$	5	2^{26}	$2^{27.21}$
6	2^{25}	$2^{26.89}$	7	2^{24}	$2^{26.42}$
8	2^{23}	$2^{25.92}$	9	2^{22}	$2^{25.44}$
10	2^{21}	$2^{24.98}$	11	2^{20}	$2^{23.09}$
12	2^{19}	$2^{19.58}$	13	2^{18}	$2^{18.10}$
14	2^{17}	$2^{17.01}$			

Se remarcă faptul că probabilitatea p_i descreşte monoton pe măsură ce i descreşte. Rata de descreştere nu este constantă şi există o scădere accentuată între probabilitățile p_{12} şi p_{11}.

Ceea ce se poate conclude din Lema 8 este faptul că scăderea întreagă este un concept mai bun al diferenței față de SAU-EXCLUSIV în ceea ce priveşte analiza lui RC6.

Caracteristica statică bit-singular studiată în paragrafele anterioare are forma generală $\delta_y = \delta_x = 2^i$. În cele ce urmează voi considera caracteristicile în care diferențele de doi biți ale intrării vor indica o diferență bit-singular la ieşire. Funcția pătratică este folosită pentru a simplifica diferențele între anumite situații.

Notez cu $e_{i,j}$ expresia $2^i + 2^j$ pentru orice $i \neq j$.

Lema 9. Fie $m_{i,j}$ probabilitatea caracteristicii $\delta_x \rightarrow \delta_y$ unde $\delta_x = e_{i,j}$ şi $\delta_y = 2^i$ pentru orice i ≠ j. Atunci:

Demonstrație. Din Lema 2 se obține:

$$m_{i,j} = \begin{cases} 2^{i-30} & pentru \;\; 29 \geq i \geq 1 \;\; si \;\; j \geq i+2 \\ 0 & pentru \;\; i = 0 \;\; si \;\; j \geq i+2 \\ 0 & pentru \;\; 29 \geq i \geq 1 \;\; si \;\; j = i+1 \\ 2^{i-30} & pentru \;\; i = 0 \;\; si \;\; j = i+1 \\ 0 & pentru \;\; 31 \geq i \geq 1 \;\; si \;\; j < i \end{cases}$$

$$\delta_y = x_1 \cdot (2^{i+2} + 2^{j+2}) + (2^i + 2^j) + 2 \cdot (2^i + 2^j)^2 \bmod 2^{32} \qquad (6)$$

Vom analiza cele trei cazuri, funcție de valoarea lui j.

Cazul 1. Presupunem $j \geq i + 2$.

Din ecuația (6), rezultă că $\delta_y = 2^i$ dacă şi numai dacă:

$$2^{i+2} \cdot x_1 \cdot (1 + 2^{j-i}) + [2^j + 2 \cdot (2^i + 2^j)^2] = 0 \bmod 2^{32}$$

Fie $a = 1 + 2^{j-i}$ şi $b = 2^j + 2 \cdot (2^i + 2^j)^2$. Deoarece a este impar, $\delta_y = 2^i$ dacă şi numai dacă:

$$2^{i+2} \cdot x_1 + a^{-1} \cdot b = 0 \bmod 2^{32} \qquad (7)$$

Dacă $i \geq 1$, atunci $b \geq 2^{i+2}$. Astfel $\delta_y = 2^i$ dacă şi numai dacă

$$x_1 = a^{-1} \cdot b \; 0 \bmod 2^{30-i}$$

Cei mai semnificativi $(i + 2)$ biți ai lui x_1 sunt ficşi. Facând madia tuturor x_1 posibili, rezultă că probabilitatea $m_{i,j} = 2^{i-30}$. Dacă $i = 0$, atunci $b = 2 \bmod 4$, deci ecuația (7) nu este niciodată adevărată şi $m_{i,j} = 0$ în acest caz.

Cazul 2. Presupunem că $j = i + 1$.

Pornind de la ecuația (6), pentru c întreg rezultă:

$$\begin{aligned} \delta_y &= x_1 \cdot (2^{i+2} + 2^{i+3}) + 2^{i+1} + 2 \cdot (2^i + 2^{i+1})^2 \bmod 2^{32} \\ &= 2^i + 2^{i+1} + 2^{2i+1} + 2^{i+2} \cdot c \end{aligned}$$

Dacă $i \geq 1$, atunci bitul $(i + 1)$ a lui δ_y este întotdeauna 1, şi deci δ_y nu este niciodată egal cu 2^i. În acest caz, $m_{i,j} = 0$. Dacă $i = 0$, atunci termenii 2^{i+1} şi 2^{2i+1} se anulează reciproc. Un calcul simplu arată că $\delta_y = 1 + 4 \cdot (3x + 5) \bmod 2^{32}$. Similar cazului 1, se obține că $m_{i,j} = 2^{-30} = 2^{i-30}$.

Cazul 3. Presupunem $31 \geq i \geq 1$ şi $j < i$.

În acest caz, este uşor de remarcat pe baza ecuaţiei (6) că bitul j a lui δ_y este întotdeauna 1. Deci probabilitatea este: $m_{i,j} = 0$.

Din cele prezentate mai sus se poate conchide că utilizarea doar a funcţiei pătratice nu este suficientă pentru a împiedica criptanaliza diferenţială [Saar-98]. În [CRRY-98] s-a demonstrat că o combinaţie a funcţiei pătratice cu o rotaţie fixă cu cinci poziţii asigură o zădărnicire a atacului sau a găsirii unor caracteristici bune care să înlesnească atacul lui RC6. Concluziile care se pot trage din analiza făcută [CRRY-98] sunt acelea că RC6 este foarte rezistent la atacurile diferenţiale şi liniare, care sunt până în momentul de faţă cele mai efective atacuri pentru cifrurile bloc.

2.6.2. Criptanaliza liniară

Criptanaliza liniară reprezintă un atac asupra cifrurilor iterative şi are o mare asemănare cu atacul criptanalitic diferenţial a lui Biham şi Shamir. Acest fapt îi determină pe mulţi cercetători să facă o analiză simultană a rezistenţei unui cifru la criptanaliza diferenţială şi liniară. Scopul atacului criptanalitic liniar este de a găsi o expresie liniară între câţiva biţi ai textului clar, câţiva biţi de ieşire ai rundei r şi câţiva biţi ai cheii. Această expresie liniară va fi validă în r runde.

Pot fi utilizate tehnici suplimentare care vor uşura atacul criptanalitic liniar. Analiza altor expresii poate da informaţii asupra altor biţi ai cheii iar numărătoarele pot fi folosite pentru căutări asupra unui subset de biţi ai cheii; aceasta poate permite criptanalistului să prevadă acţiunea unor runde şi să utilizeze o aproximaţie liniară mai scurtă [Mats-94].

Şi alte tehnici s-au dovedit a fi utile în anumite situaţii. Utilizarea aproximaţiilor liniare multiple [KaRo-94], [KaRo-95] [GoPe-96] pot intensifica uneori un atac criptanalitic liniar de bază. Ideea este de a reutiliza data deja posedată într-o altă manieră. Aceasta se face prin utilizarea unei alte aproximări liniare care dă posibilitatea extragerii mai multor informaţii privitoare la cheie prin utilizarea împreună a diverselor aproximări liniare. Cea mai importantă proprietate a acestei tehnici este influenţa diferitelor aproximări folosite. Utilitatea aproximărilor multiple poate fi uneori limitată dacă influenţele aproximărilor adiţionale sunt mult mai mici decât ale aproximării dominante [Goli-96]. Utilizare aproximărilor multiple presupune însă un volum mare de muncă.

În cadrul atacurilor criptanalitice liniare care pot fi aplicate lui RC6, se conturează două tipuri de aproximări. Primul, numit şi tipul I, utilizează un tip particular de aproximare liniară a rotaţiei dependente de dată. Aceste atacuri conduc la cele mai eficiente atacuri criptanalitice liniare asupra tuturor variantelor simplificate ale lui RC6 şi chiar asupra lui RC6. Tipul II, implică un alt stil de aproximare a rotaţiilor dependente de dată, care conduc la aproximări ale rotaţilor fixe şi ale funcţiei pătratice. Analiza prezentată în [RBSY-98] conchide că aproximările de tip II oferă atacuri mai puţin eficiente decât cele de tip I. Există însă posibilitatea combinării celor două tipuri de atacuri, obţinându-se astfel atacuri mai eficiente.

Tehnica avansată de utilizare a aproximărilor neliniare [KnRo-96] este puțin probabil să fie utilă în atacurile criptanalitice liniare considerate [RBSY-98]. Această tehnică oferă în mod obişnuit doar mici îmbunătățiri ale atacurilor şi pot fi privite ca cizelări ale eficienței unor atacuri existente.

2.6.3. Criptanaliza diferențial-liniară

Criptanaliza diferențial-liniară a fost prezentată de Langford şi Hellman la Crypto'94 [LaHe-94]. Este un atac foarte elegant care utilizează o diferențială pentru a prevedea diferența dintre două părți de text criptat. Cunoscând această diferență este posibilă utilizarea unei aproximări liniare începând din acest punct al criptării. Uneori, cu o aproximare liniară bună, poate fi acoperită o mai mare parte a cifrului decât în cazul în care s-ar folosi doar o diferențială bună. Scopul acestei combinări este de a se obține un atac asupra mai multor runde cu un preț mai mic. Acest lucru presupune existența unei difernțiale bune pentru început şi existența unei aproximări liniare bune pentru o etapă următoare a atacului. În [RBSY-98] s-a demonstrat că aceste două cerințe sunt foarte puțin probabil să poată fi îndeplinite, mai ales dacă scopul este construirea unui atac diferențial-liniar care să amenințe cifrul RC6 complet.

2.6.4. Securitatea listelor de chei

O componentă importantă a cifrurilor bloc iterative o reprezintă lista cheilor. În majoritatea cifrurilor se transformă o cheie master cu o lungime specificată pentru a se obține subchei ale rundelor. Această transformare poartă denumirea de listarea cheilor. Cele mai puternice metode de analiză ale cifrurilor bloc iterative sunt atacurile care au drept scop găsirea acestor subchei. Aceste metode includ criptanaliza liniară şi diferențială.

Listele cheilor lui RC6 sunt identice cu cele ale lui RC5 şi ca urmare considerentele de securitate referitoare la cifrul RC5 vor fi valabile şi pentru RC6.

De la publicarea lui RC5 practic nu au existat exemple de "chei slabe", deci practic se poate afirma că nici RC6 nu are chei slabe. Termenul de "chei slabe" este mult utilizat dar nu este întotdeauna aplicat în acelaşi mod. Cel mai cunoscut exemplu de "chei slabe" este cel al cifrului DES, unde a fost exploatată proprietatea structurală a cifrului.

O altă categorie de atacuri care au atras atenția sunt atacurile cheilor asociate [KeSW- 96]. Premiza esențială a atacului este acea că cheile asociate sunt folosite în timpul criptării şi prin observarea relațiilor dintre textele clare şi cele criptate, este posibilă deducerea unor informații referitoare la cele două chei. Din nou se poate argumenta cu faptul că de la publicarea lui RC5 nu au fost amintite nici un fel de

atacuri de acest fel, deci putem spune că nu există atacuri de acest fel nici asupra lui RC6. În plus se poate argumenta şi cu faptul că lista cheilor este destul de complicată şi are un design care se poate spune că este oarecum incompatibil cu structura procesului de criptare. Este greu de imaginat o cale în care schimbările cheii pot fi translatate în schimbări cunoscute şi utile ale subcheilor utilizate pe parcursul criptării.

În [CaDN-99], [DaRi-99] este analizat cifrul RC6 din punct de vedere al securităţii listei cheilor. Aşa cum am mai spus, această listă este identică cu cea a lui RC5 exceptând faptul că numărul total al subcheilor necesare este $2r + 4$ pentru RC6, faţă de $2r + 2$ pentru RC5, unde r este numărul de runde ale cifrului. Pentru RC6 r este 20, deci numărul necesar de subchei va fi 44. Subcheile se generează după cum urmează.

Dintr-o cheie master de 16, 24 sau 32 de byţi se construieşte un şir L de 4, 6 sau 8 cuvinte de 32 de biţi. Se iniţializează cu constante cunoscute un al doilea şir S care are 44 de cuvinte de 32 de biţi şi după iniţializarea variabilelor i,j,A,B se execută de 132 (3×44) de ori următoarea buclă:

For s=1 to 132

$S[i] = (S[i] + A + B) \lll 3$

$A = S[i]$

$L[j] = (L[j] + A + B) \lll (A + B)$

$B = L[j]$

$i \equiv (i + 1) \pmod{44}$

$j \equiv (j + 1) \pmod{c}$

Next s

unde c = 4, 6, sau 8 dacă cheia master are 128, 192 respectiv 256 de biţi.

Să presupunem că se cunoaşte cheia rundei $S^3[i] = (S^2[i] + A + B) \lll 3$. Din aceste date este imposibilă determinarea vreuneia dintre mărimile care apar în calculul lui $S^3[i]$. Chiar dacă aceste mărimi ar fi cunoscute, nu este posibilă determinarea lui $S^3[i + 1]$ sau $S^3[i - 1]$ deoarece mărimile din aceste expresii sunt legate relaţional de cele din $S^3[i]$ într-un mod greu inversabil. Deci cunoaşterea subcheii unei runde nu indică vreun bit al vreunei subchei a unei alte runde sau acheii master. Mai mult, fiecare intrare a şirului S este reactualizată de trei ori în bucla de mai sus. La sfârşitul primei reactualizări, să o notăm cu S^1, corespunzătoare lui s=44, $S^1[0]$ până la $S^1[7]$ inclusiv sunt singurele elemente ale şirului S^1 care nu sunt dependente de toate elementele şirului L, adică toate elementele cheii master. La sfârşitul celui de al doilea pas, toate elementele lui S^2 sunt dependente de toate elementele cheii master. De aici rezultă că cifrul RC6 oferă o bună securitate din punct de vedere al listei cheilor.

2.7. Concluzii

În acest capitol am trecut în revistă, cei 15 candidați pentru noul standard avansat de criptare (AES) şi apoi am prezentat rezultatele analizei lor atât din punct de vedere al performanțelor obținute pe diverse platforme şi cu diverse mărimi ale cheii cât şi din punct de vedere al uşurinței de implementare. În continuare, am descris detaliat cifrul bloc RC6. Pe lângă descrierea cifrului, am analizat acest candidat din punct de vedere al securității oferite.

Rezumând, în acestă parte lucrarea cuprinde următoarele contribuții:

1. O analiză comparativă a celor 15 candidați AES şi a performanțelor acestora pe diferite platforme;
2. Evidențierea cu ajutorul tabelelor şi a graficelor a principalilor parametrii care trebuiesc îndepliniți de candidați, conform cerințelor impuse de către NIST, în scopul evidențierii celor mai eficienți algoritmi, algoritmi care de altfel au şi fost promovați în runda a doua de evaluare;
3. Prezentarea clară şi detailiată a cifrului bloc RC6 şi a performanțelor acestuia pe diverse platforme;
4. Analiza originală a celor mai importante tipuri de atacuri criptanalitice asupra cifrului bloc RC6;
5. Enunțarea şi demonstrarea unui număr de nouă leme care au stat la baza analizei atacurilor diferențiale;
6. Evidențierea clară, prin criptanaliza diferențială, liniară şi a securității listelor de chei, a avantajelor cifrului RC6 față de ceilalți candidați AES.

Acel candidat care va fi ales ca nou standard de criptare va trebui să se comporte bine pe diverse platforme – procesoare high- şi low-end, pe smart card-uri şi în hardware. Din acest motiv ar fi greşit să se elimine unul sau altul dintre candidați ținându-se cont doar de performanțele lui. Oricum factorul decisiv trebuie să fie securitatea oferită. Alte aspecte importante de care trebuie ținut cont sunt costul implementării şi viteza.

În urma analizei făcute se poate spune că cifrul bloc RC6 este unul dintre cei mai puternici candidați la noul standard de criptare, el remarcându-se prin simplitate, flexibilitate, uşurința implementării, rapiditate. Toate acestea m-au determinat să aleg acest cifru bloc pentru implementare. Ca o confirmare a analizei corecte făcute, la a doua rundă de evaluare a candidaților AES care a avut loc în luna august 1999, cifrul bloc RC6 se află printre cele şase cifruri rămase în competiția de alegere a viitorului standard de criptare care va fi anunțat în vara anului 2001.

CAPITOLUL 3

FACILITĂȚI DE AUTOTESTARE ALE CIFRULUI RC6 ÎN VEDEREA IMPLEMENTĂRII HARDWARE

3.1. Analiza de oportunitate pentru aplicarea tehnicilor BIST

3.1.1. Proprietățile necesare pentru stimulii pseudoaleatori de testare

Ultimele descoperiri în materie de criptologie şi posibilitatea de integrare a unor funcțiuni complexe într-o sigură capsulă au dus la o utilizare intensivă a echipamentelor criptografice hardware. Cifrurile moderne necesită atât timp îndelungat cât şi echipament extraordinar pentru a putea fi descifrate; practic, se poate spune că sunt indescifrabile. Ca o consecință, potențialii spărgători îşi concentrează atacurile asupra slăbiciunilor echipamentului care realizează cifrarea şi descifrarea.

Posibilitatea intruziunii fizice în sisteme nu poate fi niciodată prevenită în totalitate. Oricum, miniaturizarea VLSI este una dintre măsurile de protecție corespunzătoare a informațiilor secrete.

Circuitele de mare complexitate, cum e cazul celor din generația VLSI necesită o proiectare specifică în vederea testabilității (design for testability, DfT). Marea varietate a posibilelor defecțiuni interne contrastează cu numărul mic de conexiuni exterioare care pot fi folosite pentru testare. Tehnicile convenționale DfT se bazează pe generarea exterioară a stimulilor şi analiza răspunsului, deci depind de observabilitatea şi controlabilitatea nodurilor interne. Această necesitate de accesibilitate este în conflict cu cerința criptografică de a ascunde informațiile secrete din interiorul chip-ului.

O soluție la această contradicție între securitate şi testabilitate este cea a alegerii auto-testului intern (built-in-self-test: BIST). BIST necesită integrarea în capsulă a generatorului de stimuli şi a părții de evaluare a răspunsului. Datele generale despre procesul BIST precum şi rezultatele autotestului sunt singurele informații ce pot fi obținute de la dispozitiv.

Pentru majoritatea aplicațiilor, metoda BIST este mult prea scumpă din punctul de vedere al proiectantului şi al suprafeței de siliciu necesare. Oricum, exploatarea unor proprietăți specifice ale cifrului permit integrarea unor scheme de auto-test de înaltă calitate cu un consum minim de suprafață [RoBo-91].

Posibilitatea practică de a utiliza asemenea proprietăți ale cifrului în scopuri de autotestare este demonstrată de implementarea VLSI a cifrului IDEA. Circuitul VINCI, care implementează hardware cifrul IDEA, este prevăzut cu două moduri de autotestare: un mod "off-line", incluzând un test complet şi un mod "on-line" care controlează concurent procesul de cifrare şi semnalează imediat orice defecțiune. Folosind această schemă compusă, VINCI se aliniază cerințelor actuale ale standardelor de securitate.

Evaluarea răspunsurilor la autotest se realizează frecvent prin comprimarea unui număr mare de rezultate într-o singură semnătură. Ca urmare, astfel de dispozitive nu oferă posibilitatea de localizare a defectului deoarece o semnătură greşită spune puțin despre momentul apariției şi despre numărul de erori. În schimb, în cazul circuitului VINCI, capacitatea sporită de localizare a defectelor se realizează prin urmărirea tuturor rezultatelor intermediare ale autotestului şi semnalarea oricărei malfuncționări.

Aşa cum am arătat în capitolul 1, ştiința care se ocupă cu conversia (cifrarea) datelor (text clar) în formă codificată (text cifrat) prin combinarea lor cu o informație suplimentară (cheie), cunoscută numai persoanelor autorizate, este criptografia. Cifrarea datelor se foloseşte pentru a asigura secretul mesajului transmis. Din numărul mare de algoritmi de cifrare standardizați, cei mai populari sunt cei cu cifru bloc deoarece se pot folosi la transferul rapid de date. Aceşti algoritmi se caracterizează prin faptul că sunt algoritmi cunoscuți, dar cheia folosită e secretă. Siguranța criptografică a algoritmilor moderni se bazează pe "imposibilitatea de spargere" practică: dacă nu se cunoaşte cheia, datele nu se pot descifra într-un interval de timp acceptabil, necesitând un efort de calcul mult prea mare. Aceasta se numeşte "imposibilitatea practică de spargere" [Beck-97] [Mass-88] şi e una din ipotezele de bază în această lucrare.

În cazul în care costul de spargere a unui cod nu este acceptabil, următorul pas pe care îl poate încerca o persoană neautorizată ar fi pătrunderea în echipamentul fizic pentru a obține informații confidențiale cum ar fi texte clare sau chei. Manipulări severe ale sistemelor criptografice, culminând cu înlocuirea completă a unor subansamble trebuiesc prevenite sau cel puțin detectate [Kaes-90]. Se poate afirma că orice sistem poate fi spart dacă cineva este dispus să plătească prețul corespunzător. Deci, proiectarea unui sistem "sigur" este un compromis între valoarea informațiilor şi costurile aferente măsurilor de securitate.

Operarea unui sistem criptografic presupune existența unei surse de mesaje. Scopul unui asemenea sistem este criptarea şi decriptarea datelor cu anumite chei. Deci este necesară existența unei surse pentru datele de intrare respectiv preluarea datelor de ieşire. Controlul fluxului de date între sistem şi lumea înconjurătoare necesită interfețe şi moduri de operare bine stabilite.

Complexitatea crescândă a circuitelor integrate cu sute de mii de tranzistori şi un număr mic de intrări respectiv ieşiri necesită măsuri speciale pentru validarea

funcționării corecte. Dezvoltarea metodelor care îmbunătățesc observabilitatea exterioară şi controlabilitatea pentru a avea un acces mai bun la structurile interne e o consecință logică a acestui fapt. Tratarea simplistă a testabilității expune interiorul circuitului, existând astfel posibilitatea de a expune şi informații confidențiale.

Proiectarea sistemului trebuie să fie făcută de aşa manieră încât căile de acces să nu poată fi utilizate de un potențial spărgător pentru obținerea de informații. Trebuie acordată o atenție deosebită împiedicării accesului nedorit pe căi concepute pentru alt scop. Liniile de alimentare, de date şi comenzi sunt esențiale în funcționarea unui sistem criptografic. Căile de acces necesare testării depind de metodologia aleasă. Un număr de standarde de securitate [Fede-82],[ANSI-85] se ocupă de siguranța mentenanței sistemelor cu date secrete. Metodele de test folosite în implementarea unui astfel de sistem trebuie să corespundă acestor standarde.

Cerințele față de sistemele sigure au fost sintetizate în numeroase standarde pentru echipament criptografic. Voi aborda aceste standarde şi voi defini implementarea VLSI sigură ca un concept de bază al lucrării.

Pentru a defini ceea ce se înțelege prin securitatea hardware a unui sistem criptografic voi considera exemplul cuiva care neputând să descifreze un cifru modern folosit la transmisia de date pe un canal, încearcă să pătrundă în echipamentul de cifrare/descifrare pentru a afla: informații parțiale despre textul clar, informații parțiale despre cheia de cifrare sau informații parțiale despre textul cifrat intermediar în cazul în care cifrarea se realizează în mai mulți paşi.

Pentru a împiedica orice încercare de acest fel ca urmare a unui defect, erori de proiectare sau în mod accidental, s-au propus numeroase standarde privind instalarea, inițializarea şi funcționarea echipamentelor criptografice.

În anul 1977 prin standardul FIPS 46 (Federal Information Processinng Standard) se prevedea un algoritm care a fost folosit pe scară largă şi cunoscut sub acronimul DES (Data Encryption Standard). În cele ce urmează voi înțelege prin DES şi algoritmul de criptare, Data Encription Algorithm (DEA). Trei ani mai târziu apare FIPS 140 care specifică cerințele de securitate ale echipamentului de telecomunicație ce foloseşte DES. Bazându-se pe acestea, Asociația Bancherilor Americani dezvoltă un standard specific domeniului lor de activitate [ANSI-85].

DES este un cifru bloc cu cheie secretă, având mărimea blocului de date de şaizeci şi patru de biți şi mărimea cheii tot de şaizeci şi patru de biți, dintre care doar cincizeci şi şase relevanți, restul fiind biți de paritate. DES prezintă mai multe moduri de operare, specificate de standardul FIPS 74. Modul Electronic Code Book (ECB) cifrează un bloc din mesaj folosind cheia specificată şi rezultatul se obține imediat. În modurile de lucru Cipher Block Chaining (CBC) respectiv Cipher FeedBack (CFB) criptarea blocului curent este influențată de blocurile precedente. În aceste moduri de operare DES se poate folosi la cifrarea unui flux de date. Pentru a cifra un număr de blocuri de date e nevoie de un vector de inițializare, cunoscut doar emițătorului şi receptorului [BiSh-90].

Standardele menționate mai sus formulează o seamă de cerințe de bază referitoare la echipamentul criptografic, cerințe care au fost menționate în capitolul 1.

În această teză voi accepta cerințele menționate ca fiind fundamentale pentru orice sistem criptografic.

Cerințele formulate de standarde se pot împărți pe categorii referitoare la inițializarea sistemului, validare, funcționare şi la metode de tratare a erorilor. Fiecare standard amintit se referă şi la partea electronică a echipamentului criptografic. Această teză tratează tema din punctul de vedere al circuitelor din generația VLSI, interpretând normele aminte în sensul unei implementări VLSI sigure.

Standardul FIPS 140 tratează obiectivele de securitate de bază: evitarea transmisiei accidentale de text clar, evitarea folosirii respectiv modificării neautorizate a echipamentului criptografic, prevenirea dezvăluirii sau modificării neautorizate a cheii şi prevenirea transmisiei de date în cazul unei defectări. Pentru aceasta se impune ca informațiile privind cheia în clar să nu poată fi obținute de la sistemul criptografic [ANSI-85] şi această informație să fie adusă la zero dacă se detectează o încercare de acces la componentele interne.

La punerea sub tesiune este necesară ştergerea tuturor elementelor de memorare şi încărcarea cheii inițiale şi a vectorilor de inițializare. Cheile încărcate în circuite integrate trebuiesc memorate în locații cu posibilitate de ştergere [Fede-82]. Vectorii de inițializare trebuiesc memorați astfel încât să nu se piardă în cazul căderii tensiunii de alimentare. Aceştia nu trebuie fixați prin conexiuni fizice pentru a se putea alege oricând un alt vector.

În timpul inițializării şi a încărcării fiecărei chei noi trebuie efectuat un test pentru a se putea verifica funcționarea corectă [ANSI-85] şi [Fede-82]. În [Fede-82] se recomandă utilizarea unui test cu un cuvânt de verificare care va cifra un bloc de date cunoscut. Rezultatul de referință va fi comparat cu rezultatul criptărilor ulterioare ale aceluiaşi bloc. Testul complet al generării cheii şi al echipamentului de cifrare este necesar pentru a limita riscul de a transmite date eronate la schimbarea cheii sau a vectorului de inițializare.

Echipamentul criptografic trebuie să implementeze cel puțin un mod de test al circuitelor de criptare şi al tuturor circuitelor de alarmă. Ieşirea cifrată se va dezactiva în timpul testelor [Fede-82].

Se recomandă un test continuu, adică în timpul funcționării normale a circuitului. În [ANSI-85] se arată că: "echipamentul criptografic trebuie proiectat astfel încât să se minimizeze erorile, să se detecteze defecțiunile şi să se evite posibilitatea de compromitere a cheii". Detecția unei defecțiuni sau a compromiterii cheii trebuie să fie imediată şi detectabilă prin procedurile de validare. Validarea sistemului şi funcționarea normală se recomandă să fie operații concurente sau să fie făcute în intervale de timp care se suprapun.

După detectarea unei erori, operația de cifrare sau întreaga funcționare a sistemului trebuie oprită şi semnalată. În cazul detecției (susceptării) situației de compromitere a cheii, trebuie distruse toate cheile [ANSI-85]. Echipamentul criptografic defect sau compromis trebuie ținut în stare de alarmă până când personalul de deservire va elimina cauza erorii (nu trebuie să se prevadă revenire automată) [Fede-82].

Un echipament criptografic sigur prezintă următoarele proprietăți: niciodată nu permite accesul la textul clar, parțial criptat sau la chei clare, orice malfuncționare este detectată şi semnalată imediat, orice încercare de acces neautorizat este detectată imediat, cheile şi datele confidențiale sunt şterse şi funcționarea sistemului se opreşte [Baue-97].

În literatura de specialitate s-au propus mai multe proceduri pentru evaluarea testabilității unui circuit. Toate aceste metode, spre exemplu cele sintetizate de Fujiwara [Fuji-85], se bazează pe măsurarea gradului de controlabilitate şi de observabilitate. Controlabilitatea se defineşte ca o măsură a uşurinței cu care interiorul unui circuit poate fi controlat de intrările sale primare, iar observabilitatea se defineşte ca o măsură a uşurinței cu care interiorul unui circuit se poate observa prin ieşirile sale primare. Proiectarea în vederea testabilității se poate considera în mod legitim ca un efort de a îmbunătăți atât controlabilitatea cât şi observabilitatea unui sistem, altfel spus accesibilitatea sa.

Procesul de testare constă în stimularea intrărilor circuitului şi compararea rezultatelor obținute cu o referință corectă. Orice tehnică de proiectare în vederea testabilității (DfT) include o metodă de generare a stimulilor, un mecanism de propagare a acestora şi o strategie de evaluare a răspunsului circuitului [VlGr-81]. Metodele DfT diferă prin modul în care se generează stimulii şi prin modul în care răspunsurile sunt propagate spre ieşiri şi evaluate. Tehnicile mai vechi necesitau o generare exterioară a stimulilor şi a analizei răspunsului, pe când tehnicile mai noi cer ca generarea de stimuli şi/sau analiza răspunsurilor să fie integrate în circuit.

După transformarea unui proiect oarecare într-unul uşor testabil, testul este realizat prin aplicarea numărului specificat de stimuli. Dacă toate răspunsurile sunt egale cu rezultatele corecte, funcționarea circuitului este validată. Gradul de încredere depinde de felul în care stimulii acoperă plaja de defecte care pot să apară.

Testabilitatea slăbeşte securitatea unui sistem datorită măririi accesibilității. În acest capitol voi prezenta şi câteva implementări de circuite pentru a clarifica pierderile de securitate datorate testabilității. Conceptele de securitate şi testabilitate sunt contradictorii în implementarea echipamentului criptografic. Reconcilierea ambelor cerințe pentru implementarea VLSI este obiectivul principal al acestei lucrări.

Acest capitol tratează câteva proprietăți ale cifrurilor bloc care pot fi exploatate pentru o metodă BIST eficientă. În prima parte am descris formal fluxul de date folosind diagrame de flux de date, iar apoi am prezentat informația extrasă din două cifruri existente, DES şi RC6. Ca o consecință se va propune o strategie BIST eficientă pentru implementările RC6. Ultima parte se ocupă de proprietățile cifrurilor care pot fi folosite pentru realizarea autotestului concurent.

După cum s-a prezentat în capitolele anterioare, autotestul intern Built-In-Self-Test (BIST), a fost acceptat ca o metodă care satisface cerințele de securitate ale echipamentelor de cifrare [Goor-91]. Un exemplu bine-cunoscut de autotest pseudoaleator este BILBO (Built In Logic Block Observation), care necesită ca fiecare registru de intrare şi ieşire a unui anumit nivel combinațional să poată fi folosit ca şi generator de tipare şi ca analizor de semnătură [Vlăd-86]. Metoda de test BILBO pentru fiecare bloc combinațional este:

1. generarea tiparelor în regiştrii de intrare (mod generator);
2. propagarea tiparelor în circuitul combinaţional şi compresia răspunsurilor de test prin însumarea cu registrul de ieşire (mod analizor);
3. compararea cu o semnătură validă.

Când registrul de intrare este alcătuit dintr-un registru de deplasare LFSR (Linear Feedback Shift Register), metoda BILBO de generare a tiparelor este considerată a fi o metodă pseudoaleatoare. Regiştrii BILBO ocupă o suprafaţă de două ori mai mare decât regiştrii normali şi comanda lor este complexă. După cum se indică în [Verb-91] şi [HRSW-91], folosirea proprietăţilor codurilor promite reducerea costului pentru autotestare fără a pierde avantajele aferente, prin reducerea numărului de regiştri multifuncţionali în vederea reducerii ariei de siliciu iar prin simplificarea complexităţii părţii de comandă se va obţine o reducere a timpului de autotest precum şi a timpului de proiectare.

Prin folosirea proprietăţilor codurilor, metoda de test prezentată mai sus se reduce la:

1. generarea tiparelor (în interiorul circuitului);
2. propagarea tiparelor şi comprimarea răspunsurilor prin adunarea cu un registru LFSR cu mai multe intrări;
3. compararea stării finale a registrului LFSR cu o valoare de referinţă.

Această metodă necesită un singur generator de stimuli în paralel cu intrările circuitului, un singur analizor pe ieşirile sale şi un planificator primitiv pentru operaţiile de test care va comuta intrările între conexiunile externe ale circuitului şi generatorul de stimuli. Această metodă de test este fiabilă dacă: datele de intrare pentru fiecare nivel combinaţional sunt suficient de (pseudo) aleatoare şi ieşirea fiecărui nivel combinaţional este propagată spre registrul LFSR cu mai multe intrări [VICr-89].

Rezultă că fiecare nivel combinaţional trebuie să menţină calitatea (pseudo) aleatoare a ieşirilor sale pe care le va propaga nivelului următor. De asemenea, prin ciclarea de câteva ori printr-un număr de nivele combinaţionale trebuie să se realizeze ca măcar un nivel să producă date suficient de (pseudo) aleatoare. În general, pentru funcţii combinaţionale arbitrare este greu a se determina gradul de "menţinere al caracterului aleatoriu". În [FuRi-88] s-au dezvoltat mai multe criterii pentru evaluarea caracterului (pseudo) aleator care pot fi aplicate unei secvenţe de tipare. Aceste criterii sunt foarte stricte, dar sunt satisfăcute spre exemplu de aşa numitele secvenţe de registre LFSR de lungime maximă [MaMa-97]. Se face observaţia că în principiu caracterul (pseudo) aleator se poate pune în evidenţă doar pentru o secvenţă de intrare dată.

3.1.2. Proprietăţi de propagare a modelelor aleatoare

Cifrurile binecunoscute de genul DES [Fede-77], FEAL [MoYa-91], IDEA [LaMa-90], [LaMa-91] sau RC6 îşi sporesc siguranţa prin iterarea repetată a unei funcţii criptografice nesigure. Implementările hardware ale cifrurilor bloc nu conţin în

general partea necesară obținerii simultane a tuturor iterațiilor. Considerente referitoare la aria de siliciu ocupată limitează numărul de iterații la una sau două. Datele se cifrează prin reutilizarea circuitului pentru obținerea numărului dorit de iterații. Primul pas în implementarea VLSI este maparea fluxului de date pe o anumită arhitectură. Pentru a se obține aceleaşi funcțiuni cu un număr redus de blocuri este necesar ca orice cale din fluxul de cifrare să poată fi parcursă şi în fluxul de date al circuitului.

Pentru testul pseudoaleator al tuturor nivelelor nu este de ajuns ca fiecare nivel să păstreze caracterul aleator al intrărilor sale ci se impune ca să îl accentueze prin propagare [CuBK-90] [BoCK-90]. Acest lucru este important deoarece aplicarea la intrarea unui circuit a aceluiaşi stimul pe o perioadă de timp nu contribuie la detectarea mai bună defectelor.

Pentru a demonstra că fiecare nivel primeşte la intrare date (pseudo) aleatoare, proprietățile şi ordinea operațiilor implicate sunt relevante. Un singur nivel care diminuează caracterul aleator va împiedica testarea corectă cel puțin a nivelului imediat următor. Ceea ce contează este ordinea şi proprietățile unei anumite operații şi nu numărul de efectuări a acesteia. Această observație poate duce la realizarea de arhitecturi care prezintă aceeaşi secvență de operații dar diferă ca şi arie de siliciu utilizată. Pentru a afla dacă o anumită arhitectură poate fi folosită pentru a realiza secvența de operații necesare cifrării, se procedează în felul următor: în prima fază se construieşte graful care ia în considerare structura specifică a fluxului de date, numit diagrama fluxului de date al cifrului. Al doilea pas este analiza acestei diagrame pentru a se determina compatibilitatea ei cu o arhitectură propusă. Modul de analiză va fi prezentat în cele ce urmează.

Voi arăta că diagrama fluxului cifrului se poate folosi şi pentru studiul păstrării şi generării caracterului aleator în timpul operațiilor circuitului.

3.1.3. Descrieri formale

Pentru început voi prezenta câteva definiții de bază din teoria grafurilor.

***Definiția 3.1*:** Un graf orientat G este format din perechea (V, E) astfel încât:

1. V = $\{v_r, \ldots, v_n\}$ este o mulțime finită, nevidă, ale cărei elemente se numesc noduri;
2. E este o mulține nevidă de perechi de noduri, ale cărei elemente se numesc arce. Arcul (a, b) este orientat de la a la b.

Exemplul 3.1: Figura 3.1. ilustrează graful orientat $(\{v_1, v_2, v_3, v_4\}, \{(v_1, v_2), (v_2, v_4), (v_3, v_2), (v_4, v_3)\}$.

***Definiția 3.2*:** Fie (V, E) un graf orientat.

1. Pentru un arc (v_i, v_j) din E, v_i se numeşte predecesorul lui v_j iar v_j se numeşte succesorul lui v_i;

2. Pentru $v \in V$, mulțimea $\bullet v := \{v_i \mid (v_i, v) \in E\}$ se numeşte mulțimea predecesorilor lui v;
3. Pentru $v \in V$, mulțimea $v\bullet := \{v_i \mid (v, v_i) \in E\}$ se numeşte mulțimea succesorilor lui v;
4. Nodul v se numeşte izolat dacă $\bullet v = v\bullet = \{\}$, adică nici un arc nu porneşte şi nu se termină în v.

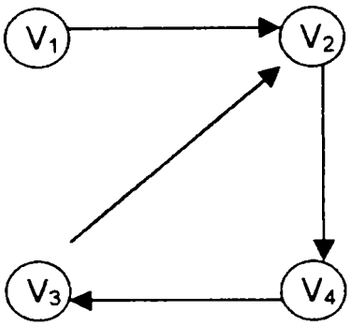


Figura 3.1. Un graf simplu direcțional

Definiția 3.3: Tuplul de noduri $(v_1, \ldots, v_m)$ se numeşte drum de la v_1 la v_m în graful orientat (V, E) dacă mulțimea $\{(v_1, v_2), \ldots, (v_{m-1}, v_m)\}$ este o submulțime a lui E.

Acum se poate defini un graf potrivit pentru a descrie diagrama fluxului de date al unui cifru. Se va face diferența între mulțimea nodurilor de intrare, a nodurilor interne şi a celor de ieşire.

Definiția 3.4: Diagrama fluxului de date al cifrului (CDGF) este un graf orientat (V, E), fără noduri izolate, care conține cel puțin un nod fără predecesori, cel puțin un nod fără succesori şi cel puțin un nod care are atât predecesori cât şi succesori.

În acest graf nodurile fără predecesori se numesc noduri de intrare, cele fără succesori noduri de ieşire iar restul nodurilor se numesc noduri interne.

Exemplul 3.2: Graful din figura 3.2.a. este o diagramă de flux de date al unui cifru cu trei noduri de intrare şi unul de ieşire. Graful din figura 3.2.b. nu reprezintă o diagramă de flux de date deoarece nu are noduri de ieşire.

In continuare voi prezenta modul de construire al diagramei fluxului de date al unui cifru. Intrările pentru text clar şi pentru cheie reprezintă intrările grafului. Textul cifrat reprezintă nodurile de ieşire [Mang-98]. Fiecare operație se reprezintă printr-un nod intern. Voi face observația că fiecare dintre nodurile interne are cel puțin un predecesor şi cel puțin un succesor, dar predecesorii şi succesorii nu trebuie să fie noduri interne. Arcul (v,w) dintre două noduri v şi w arată că rezultatul operației simbolizate prin v este intrarea operației w [MaMa-93].

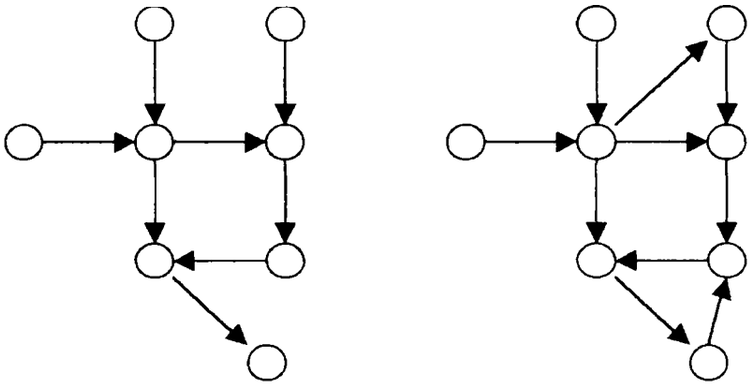


Figura 3.2. (a) un graf CDFG şi (b) un graf ce nu împlineşte proprietăţile CDFG

Exemplul 3.3: Pentru cifrul IDEA, care este o variantă uşor modificată a cifrului PES, diagrama de flux de date prezentată în figura 3.3. Mulţimea nodurilor de intrare în diagrama fluxului cifrului IDEA este $\{Z_i^{(r)}, Z_j^{(9)}, X_1, X_2, X_3, X_4\}$, unde i ia valori în mulţimea {1,...,4}, j aparţine mulţimii {1, ..., 9} iar mulţimea nodurilor de ieşire este $\{Y_1, Y_2, Y_3, Y_4\}$. Proprietăţile cele mai importante ale cifrului sunt următoarele: toate operaţiile sunt operaţii de grup şi nici o operaţie nu e succedată de o operaţie de acelaşi tip. IDEA este un cifru bloc care operează pe blocuri de text de 64 biţi. Cheia este de 128 biţi. Algoritmul se bazează pe principiul de a "amesteca operaţii ale unor grupuri algebrice diferite" [Mang-99]. Sunt amestecate 3 grupuri algebrice şi toate sunt uşor de implementat atât hardware cât şi software: XOR, adunarea modulo 2^{16} şi înmulţirea modulo $2^{16}+1$ (această operaţie poate fi privită ca şi cutia S a lui IDEA). Toate aceste operaţii (şi acestea sunt singurele operaţii în algoritm - nu există permutări la nivel de bit) operează în subblocuri de 16 biţi. Acest algoritm este eficient chiar pe procesoarele de 16 biţi [BCKF-91]. Blocul de date pe 64 biţi este divizat în subblocuri de 16 biţi $X_1,..., X_4$. Acestea devin intrarea pentru primul pas al algoritmului. Există în total 8 paşi. În fiecare pas cele patru subblocuri sunt supuse operaţiei XOR, adunate şi multiplicate unul cu altul sau cu şase subblocuri pe 16 biţi ale cheii. Între paşi, al 2-lea şi al 3-lea subloc sunt schimbate între ele. În final cele patru subblocuri sunt combinate cu patru subchei, într-o transformare de ieşire. În fiecare pas, secvenţa evenimentelor este următoarea:

1. Multiplică X_1 şi prima subcheie
2. Adună X_2 şi a doua subcheie
3. Adună X_3 şi a 3-a subcheie
4. Multiplică X_4 şi a patra subcheie
5. XOR rezultatele evenimentelor 1 şi 3
6. XOR rezultatele evenimentelor 2 şi 4
7. Multiplică rezultatele de la 5 cu a cincea subcheie
8. Adună rezultatele 6 cu 7
9. Multiplică rezultatele de la 8 cu a şasea subcheie

10. Adună rezultatele de la 7 şi 9
11. XOR rezultatele evenimentelor 1 şi 9
12. XOR rezultatele evenimentelor 3 şi 9
13. XOR rezultatele evenimentelor 2 şi 10
14. XOR rezultatele evenimentelor 4 şi 10.

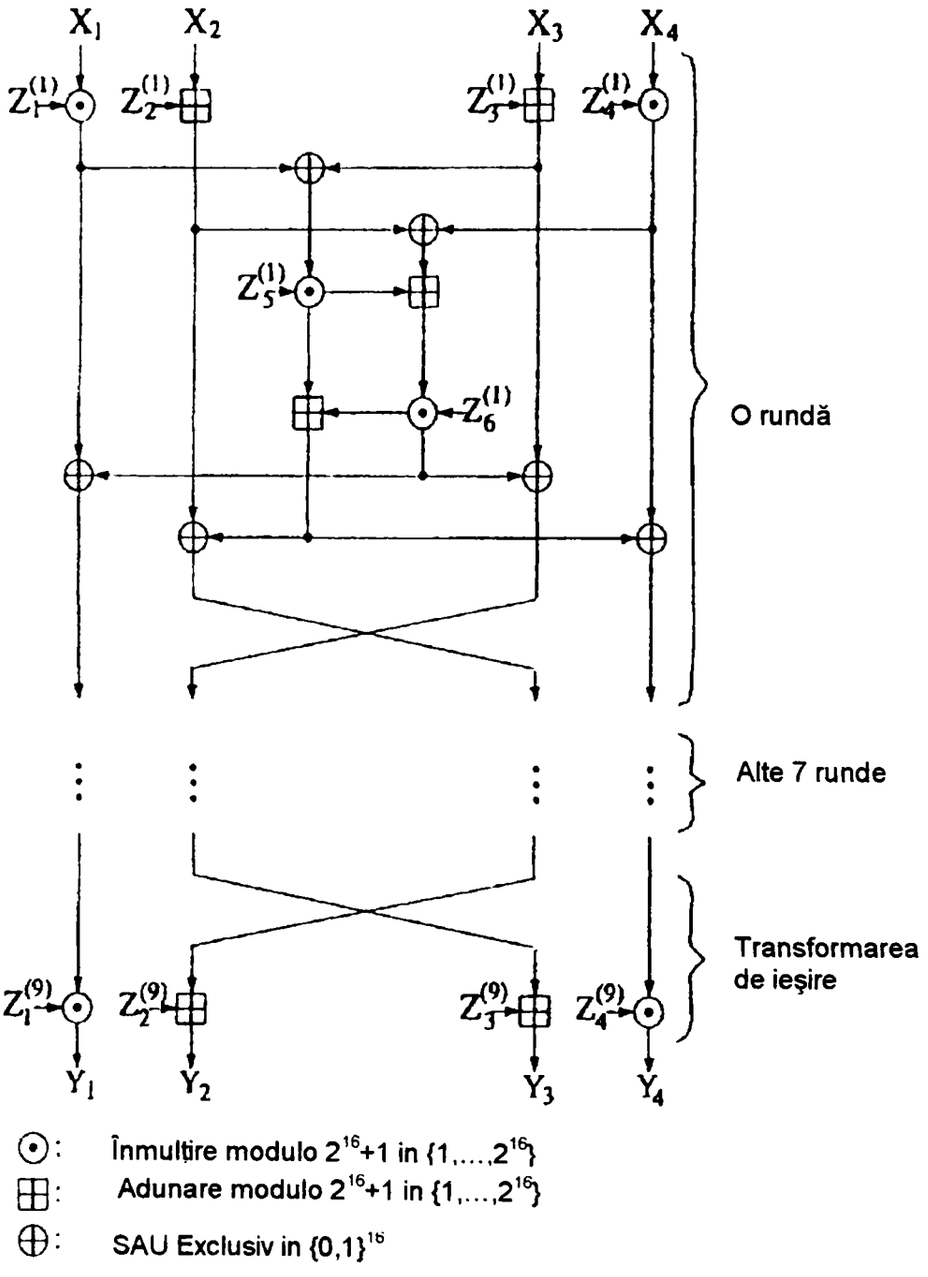


Figura 3.3. Data flow graph pentru cifrul IDEA

Rezultatul unui pas sunt cele patru subblocuri, rezultate a evenimentelor 11, 12, 13 şi 14. Se înlocuiesc între ele cele două blocuri interioare (cu excepţia ultimului pas) şi aceasta este intrarea în pasul următor. După cel de-al 8-lea pas se face transformarea de ieşire:

1. Multiplică X_1 şi prima subcheie
2. Adună X_2 şi a doua subcheie
3. Adună X_3 şi a treia subcheie
4. Multiplică X_4 şi a patra subcheie

În final cele patru subblocuri sunt reunite pentru a produce textul cifrat.

În principiu, o implementare hardware a unui cifru se poate realiza prin implementarea blocurilor corespunzătoare fiecărei operaţii şi integrarea lor într-o capsulă făcând legăturile corespunzătoare. Evident, fiecare tip de operaţie are un număr constant de intrări. Constrângeri privind aria de siliciu utilizată împiedică o corespondenţă directă a operaţiilor cu blocurile fizice. Sarcina proiectantului este de a găsi o soluţie potrivită pentru transformarea datelor de intrare folosind mai puţine componente.

Putem obţine o posibilă arhitectură prin modificarea uşoară a diagramei fluxului de date [Mang-98]. Această diagramă specială, numită în continuare diagrama fluxului de date a arhitecturii, se poate implementa fizic în acelaşi mod ca şi diagrama completă a fluxului de date a cifrului. Arhitectura mai restrânsă poate să realizeze transformarea datelor conform diagramei cifrului dacă:

1. cele n tipuri de noduri interne sunt aceleaşi în diagramele arhitecturii şi ale cifrului;
2. fiecare drum de la o intrare la o ieşire în diagrama cifrului este un drum şi în diagrama arhitecturii (posibil conţinând un ciclu), drum care are aceeaşi intrare, ieşire şi parcurge noduri intermediare de acelaşi tip.

Să considerăm o arhitectură mai restrânsă decât diagrama originală a fluxului cifrului. Reutilizarea aceloraşi blocuri fizice pentru calculul mai multor iteraţii necesită drumuri de reacţie suplimentare de la ieşirile unor blocuri spre intrările altor blocuri.

***Definiţia 3.5*:** O diagramă a fluxului de date a arhitecturii (ADGF) este o diagramă a fluxului de date de cifrare având două mulţimi disjuncte de noduri, S şi W astfel încât:

1. fiecare nod *s* din S are cel puţin un predecesor de tip intrare şi/sau operaţie şi un singur succesor de tip operaţie;
2. fiecare nod *w* din W are predecesori noduri de selecţie şi succesori noduri de selecţie şi de ieşire; toate nodurile corespunzătoare operaţiilor de acelaşi tip W_i au acelaşi număr de noduri de selecţie a predecesorilor.

A se remarca faptul că nici un nod de selecţie nu este urmat de un alt nod de selecţie şi nici un nod operaţie nu este urmat de o altă operaţie. Diagrama fluxului cifrului se poate transforma într-o diagramă a fluxului arhitecturii prin introducerea

unui nod de selecție pe toate arcele (*v, w*) unde *v* este nod de intrare sau intern iar *w* este un nod intern.

Exemplul 3.4. În figura 3.4 se prezintă diagrama fluxului arhitecturii ce implementează o iterație a cifrului IDEA. Nodurile de selecție sunt reprezentate ca puncte negre. Nodurile de selecție care au un singur predecesor vor fi omise în implementarea fizică, fiind realizate print-o simplă legătură [BCZF-93].

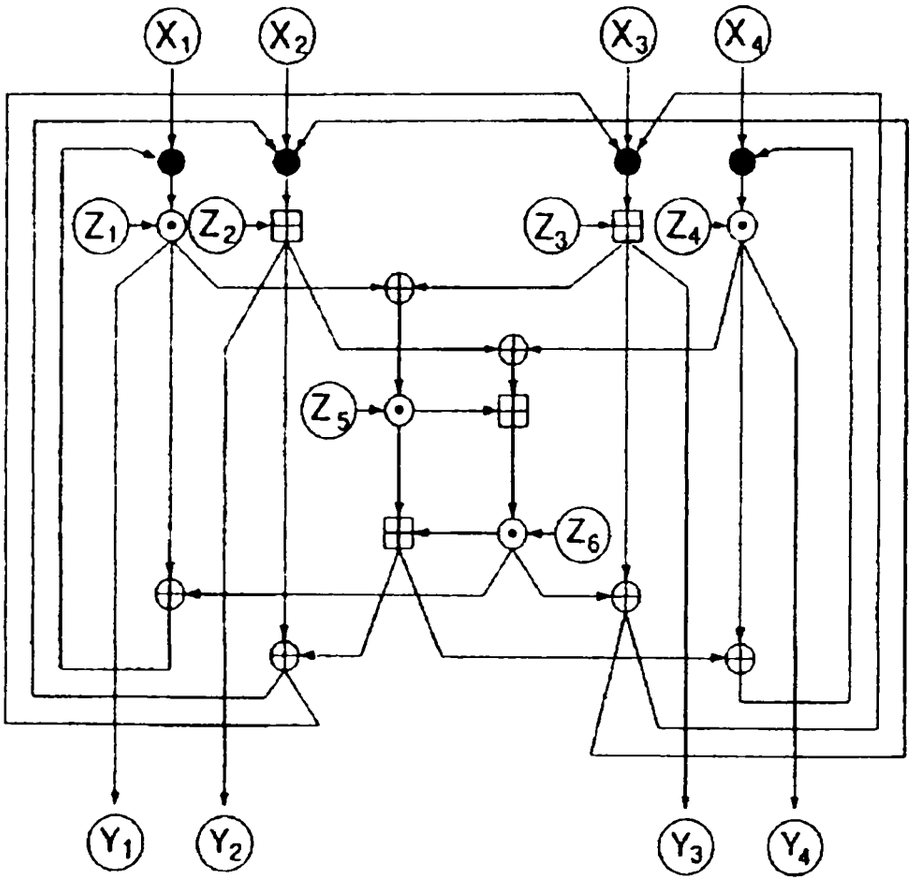


Figura 3.4. Graful de tip ADFG al unei arhitecturi hardware pentru implementarea cifrului IDEA

În general, multe dintre nodurile interne ale unei diagrame de cifru pot să fie de acelaşi tip. Dacă există *n* tipuri de operații, prin W_i se va nota mulțimea de noduri interne de tip *i*. O arhitectură va fi capabilă să realizeze cifrarea conform algoritmului dacă fiecare secvență de operații din diagrama cifrului este o secvență de tipuri de operații în diagrama fluxului arhitecturii. Ca o consecință, se va lua în considerare secvența de tipuri de operații din diagrama fluxului cifrului.

Suprafața de siliciu necesară pentru implementarea hardware a multiplicării reziduale pentru cifrul IDEA exclude alternativa unei mapări directe a fluxului de date, adică implementarea fizică a celor opt iterații. Graful din figura 3.4 se aseamănă cu diagrama fluxului arhitecturii dar implementează doar o iterație, folosind o structură mai simplă. Lista predecesorilor este identică cu cea a fluxului IDEA. Reutilizarea aceloraşi blocuri pentru calculul tuturor iterațiilor este posibil prin intermediul arcurilor de reacție (v, w), cu v din mulțimea W_{XOR}, w aparținând mulțimii W_{Mul} şi (s, t) cu s din mulțimea W_{XOR} şi t din mulțimea W_{Add}.

Din această secvență se obține lista predecesorilor tipurilor de operații. Pentru fiecare nod al grafului se va adăuga o poziție în listă; noua poziție conține tipul operației şi tipul predecesorilor săi. Pozițiile identice din listă se omit [Mang-99].

Exemplul 3.5. În figura 3.5 se prezintă lista predecesorilor nodurilor diagramei cifrului IDEA. Prima coloană indică nodul iar a doua indică predecesorii săi (în ordine de la stânga la dreapta).

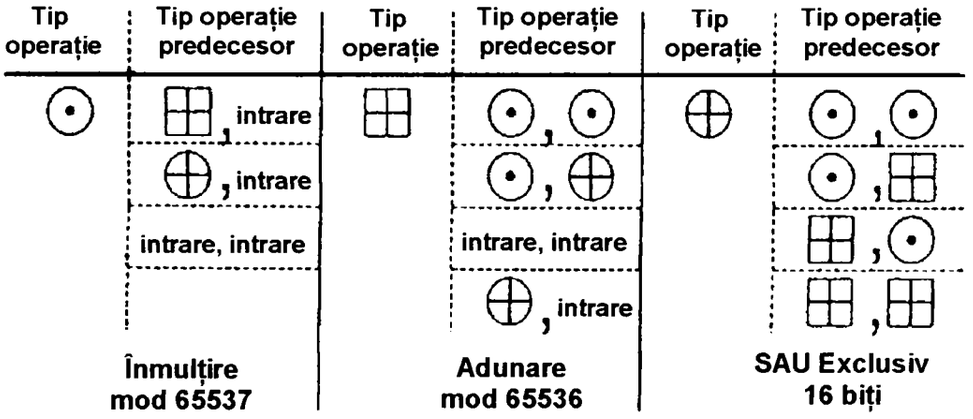


Figura 3.5. Lista predecesorilor pentru cifrul IDEA

Extragerea listei predecesorilor din diagrama arhitecturii se poate realiza conform următorului algoritm:

***Algoritmul 3.1*:** Pentru construirea listei predecesorilor se parcurg următorii paşi:

1. Se porneşte de la nodurile de ieşire ale diagramei arhitecturii. Se alege dintre ele unul, care încă nu a fost vizitat şi se selectează unul dintre nodurile sale predecesoare care reprezintă operații.
2. Se selectează predecesorul nevizitat cel mai din stânga al fiecărui nod de selecție care precede nodul curent. Se inserează în listă o poziție corespunzătoare tipului operației dacă încă nu există operații de acest tip. Se repetă acest pas până când toți predecesorii nodurilor de selecție predecesoare au fost parcurşi de la stânga la

dreapta. Pentru numere diferite de predecesori ale nodurilor de selecție se alege tipul de operație cel mai din dreapta al nodurilor de selecție cu cei mai puțini predecesori. Se marchează nodul curent ca vizitat. Nodul curent devine predecesorul cel mai din stânga al nodului de selecție predecesor care nu este nod de intrare şi încă nu a fost vizitat. Se continua cu pasul (2). Dacă toți predecesorii tuturor nodurilor de selecție predecesoare au fost vizitați deja, ne întoarcem la succesorul nodului curent şi se continuă cu pasul (2).

3. La întoarcerea la nodul inițial, se continuă cu pasul (1) până la vizitarea tuturor nodurilor.

3.2. CDGF-ul şi ADGF-ul cifrului RC6

Pentru studiul cifrului RC6 se consideră diagrama fluxului de date prezentată în figura 3.8. Mulțimea nodurilor de intrare în diagrama fluxului cifrului RC6 este {A,B,C,D,S[0],...,S[43]}, iar mulțimea nodurilor de ieşire este {A,B,C,D}. Proprietățile cele mai importante ale cifrului sunt următoarele: toate operațiile sunt operații de grup şi nici o operație nu este succedată de o operație de acelaşi tip. Aceasta se reflectă în lista predecesorilor pentru cifrul RC6 (figura 3.9) prin faptul că nici o operație nu apare ca propria sa predecesoare.

Suprafața de siliciu necesară pentru implementarea hardware a multiplicării reziduale pentru cifrul RC6 exclude alternativa unei mapări directe a fluxului de date, adică implementarea fizică a celor douăzeci de iterații. Graful din figura 3.10 se aseamănă cu diagrama fluxului arhitecturii dar implementează doar o iterație, folosind o structură mai simplă. Lista predecesorilor este identică cu cea a fluxului RC6. Reutilizarea aceloraşi blocuri pentru calculul tuturor iterațiilor este posibilă prin intermediul arcurilor de reacție (v, w), cu v din mulțimea W_{Add} şi w aparținând mulțimii W_{XOR}.

În cele ce urmează am analizat modul de organizare eficientă a autotestului pentru un circuit de cifrare RC6. Considerând că întrega diagramă a fluxului de date a cifrului RC6 este implementată fizic, o aplicație dată la intrările circuitului şi care pare suficient de (pseudo)aleatoare poate fi folosită pentru testarea blocurilor interne direct conectate. Dacă ieşirile acestor blocuri păstrează caracterul aleator, următoarele blocuri vor fi de asemenea testabile. Acest lucru se va aplica succesiv pentru toate nivelurile arhitecturii. Aplicarea unui număr suficient de mare de stimuli la intrare va asigura o bună testare a tuturor subblocurilor (în cazul în care acestea sunt testabile în mod pseudoaleator).

În mod curent doar o parte din diagrama fluxului este implementată fizic, spre exemplu doar o iterație. Reutilizarea aceluiaşi bloc fizic în mai multe iterații necesită modificarea stimulilor aplicați la intrarea unui anumit bloc în fiecare iterație pentru a se obține o acoperire satisfăcătoare a defectelor posibile. Aceasta înseamnă producerea de tipare aleatoare la fiecare iterație. Considerând lista predecesorilor unei posibile diagrame a arhitecturii RC6, am tratat proprietatea de producere a

tiparelor aleatoare de către secvența de operații RC6. Din considerente teoretice am asumat faptul că intrările RC6 produc chei cu adevărat aleatoare şi nu subchei cum se prezintă în [LaMa-90].

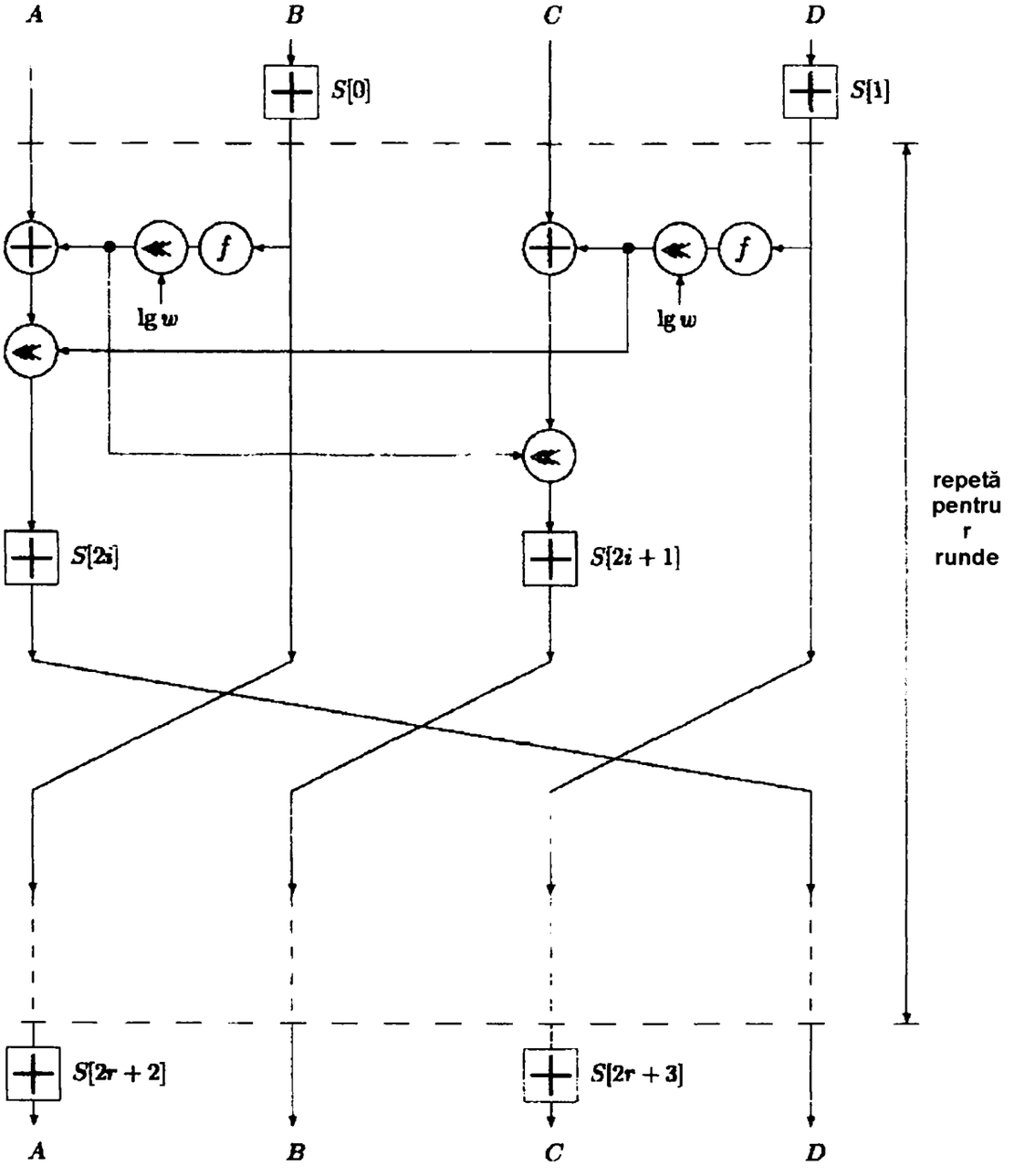


Figura 3.8. Data flow graph pentru cifrul RC6

Tipul operației	Tipul operației predecesorilor
⊕	intrare, lgw
⊞	lgw
⊞	<<<, intrare
	intrare, intrare
<<<	⊕, lgw
f	⊞
lgw	f

Figura 3.9. Lista predecesorilor pentru cifrul RC6

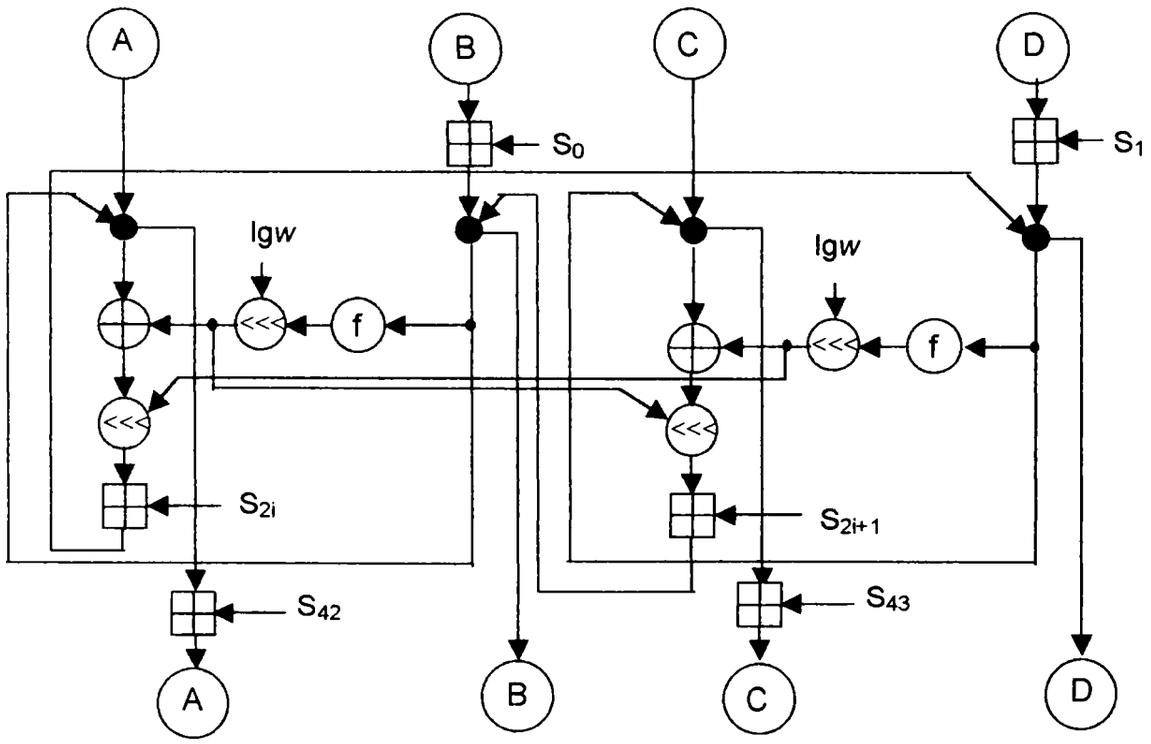


Figura 3.10. ADFG-ul pentru cifrul RC6

3.3. Propagarea de modele aleatoare pentru cifrul RC6

Un cifru se defineşte ca fiind perfect secret dacă textul cifrat este independent de mesajul clar. Voi demonstra secretul perfect al cifrului Vernam cu următoarea teoremă:

Teorema 3.1: Fie X, Y şi Z trei variabilele aleatoare din grupul finit a cărui operaţie se notează cu "*" şi Y = X*Z. Dacă X şi Z sunt independente şi Z are aceeaşi probabilitate de a fi oricare element al grupului, atunci X şi Y sunt independente.

Demonstraţie: Z are o distribuţie uniformă, deci probabilitatea sa este:

$$P_Z(z) = \frac{1}{\text{numarul de elemente ale grupului}}$$

unde z este un element al grupului. Deoarece operaţia "*" este operaţie de grup, atunci egalitatea Y = X*Z presupune ca Z = X'*Z şi reciproc, unde X' este inversul lui Z. Notăm cu $P_{Y|X}(y|x)$ probabilitatea textului cifrat *y* condiţionată de probabilitatea mesajului clar *x*. Următoarele ecuaţii sunt adevărate:

$$P_{Y|X}(y \mid x) \tag{3.1}$$

$$= P_{X'*Y|X}(x'*y \mid x) \tag{3.2}$$

$$= P_{Z|X}(x'*y \mid x) \tag{3.3}$$

$$= P_Z(x'*y) \tag{3.4}$$

$$= \frac{1}{\text{numarul de elemente ale grupului}} \tag{3.5}$$

Egalităţile de mai sus se obţin pe baza următorului raţionament: a doua ecuaţie (3.2) se obţine prin proprietatea de asociativitate la stânga a lui Z' faţă de Y. Se observă că nu se presupune faptul că grupul este abelian. Înlocuind X' * Y cu Z, se obţine cea de-a treia ecuaţie (3.3). Folosind independenţa statistică a lui X şi Z se obţine ecuaţia (3.4), iar distribuţia uniformă a lui Z duce la rezultatul final. Astfel am demonstrat că probabilitatea textului cifrat condiţionată de probabilitatea mesajului clar este uniform distribuită în grup şi deci Y este independent faţă de X [Mass-93], [Konh-81].

Teorema 3.1 poate fi generalizată pentru câteva operaţii de grup astfel:

Corolarul 3.1: Să presupunem că * şi • sunt operaţii de grup ale aceleiaşi mulţimi finite, că A, B şi C sunt independente şi că variabila aleatoare A poate să fie oricare element al grupului cu aceeaşi probabilitate. Atunci în relaţia E = (A*B) • D, variabilele B şi E respectiv D şi E sunt independente.

Demonstrație: Deoarece A poate fii orice element al grupului, cu aceeaşi probabilitate, şi este independentă de B, atunci (A*B) poate de asemenea să fie orice element al grupului cu aceeaşi probabilitate. Deoarece variabilele A, B şi D sunt independente, (A*B) şi D sunt independente. Aplicând teorema 3.1 rezultă că şi D şi E = (A*B) • D sunt independente.

Cifrul RC6 se obține be baza operațiilor de grup. Să presupunem că intrările diagramei cifrului RC6 A, B, C, D, S[0], S[1], ..., S[2r+3] sunt variabile aleatoare şi independente două câte două. Se pot afirma următoarele despre operația de grup a cifrului:

***Teorema 3.2*:** Dacă X, Y sunt două variabile aleatoare şi independente două câte două şi Z=X+Y este ieşirea unui sumator din algoritmul RC6, atunci intrările şi ieşirile fiecărui sumator sunt independente.

Demonstrație: Având în vedere că toate operațiile sunt operații de grup şi nici o operație nu este succedată de o operație de acelaşi tip, ceea ce se reflectă în lista predecesorilor pentru cifrul RC6 (figura 3.9), întocmită de mine şi prezentată în paragraful 3.2, şi pe baza diagramei fluxului de date se pot pune în evidență două cazuri, după cum urmează:

a). Dacă operanzii sunt noduri de intrare, independența este o consecință a teoremei 3.1, demonstrată mai sus, şi a calității aleatoare a intrărilor. Deoarece operația "+" este operație de grup, atunci egalitatea Z = X+Y presupune ca Y = X'+Y şi reciproc, unde X' este inversul lui Y. Probabilitatea textului cifrat *z* condiționată de probabilitatea mesajului clar *x* este

$$P_{z|x}(z \mid x) = P_{y|x}(x'+z \mid x) = P_y(x'+z) = \frac{1}{nr.de\,elemente\,ale\,grupului}$$

Deci, am demonstrat că probabilitatea textului cifrat condiționată de probabilitatea mesajului clar este uniform distribuită în grup şi deci ieşirea Z este independentă față de intrarea X, respectiv intrarea Y.

b). în cazul în care intrările sunt: una este rezultatul unei rotații şi cealaltă este nod de intrare, aplicând raționamentul de mai sus se demonstrează din nou că ieşirea Z este independentă față de X şi Y.

***Teorema 3.3*:** Dacă X este o variabilă aleatoare şi independentă şi Z=X(2X+1) este ieşirea unui circuit care implementează funcția pătratică din algoritmul RC6, atunci intrările şi ieşirile acestui circuit sunt independente.

Demonstrație: Având în vedere că toate operațiile din cifrul RC6 sunt operații de grup şi nici o operație nu este succedată de o operație de acelaşi tip, ceea ce se reflectă în lista predecesorilor pentru cifrul RC6 (figura 3.9), prezentată în paragraful 3.2, şi urmărind diagrama fluxului de date a cifrului (figura 3.8) se poate observa că predecesor al nodului funcției *f* este doar sumatorul, ale cărui intrări şi ieşiri sunt independente, aşa cum s-a demonstrat în teorema 3.2. Pe de altă parte probabilitatea textului cifrat *z* condiționată de probabilitatea mesajului clar *x* este:

$$P_{z|x}(z \mid x) = \frac{1}{nr.de\ elemente\ ale\ grupului}$$

Folosind independența statistică a lui X şi Z şi ținând cont de distribuția uniformă a lui Z se poate spune că probabilitatea textului cifrat condiționată de probabilitatea mesajului clar este uniform distribuită în grup şi deci ieşirea Z este independentă față de intrarea X.

***Teorema 3.4*:** Dacă X, Y sunt două variabile aleatoare şi independente două câte două şi Z=X⊕Y este ieşirea unui sumator modulo 2 din algoritmul RC6, atunci intrările şi ieşirile fiecărui circuit XOR sunt independente.

Demonstrație: Urmărind diagrama fluxului de date a cifrului RC6 (figura 3.8) şi lista predecesorilor pentru cifrul RC6 (figura 3.9) se observă că predecesorii operației XOR sunt fie intrări considerate independente, fie sumatori, fie registre de rotație, pentru care proprietatea a fost demonstrată.

Dacă Z = X ⊕ Y = X Y'+X' Y, având în vedere că operația XOR este operație de grup şi deoarece X poate fi orice element al grupului cu aceaşi probabilitate şi X este independent față de Y', atunci XY' poate de asemenea să fie orice element al grupului cu aceaşi probabilitate. Deoarece şi X, Y' şi X' sunt independente, atunci şi XY', respectiv X'Y sunt independente. Aplicând teorema 3.1, demonstrată mai sus, rezultă că şi X'Y şi Z sunt independente, deci din nou putem afirma că ieşirea Z este independentă față de intrările X şi Y.

***Teorema 3.5*:** Dacă X este o variabilă aleatoare şi independentă şi Z este rezultatul unei rotații fixe la stânga cu (lg*w*) poziții sau a unei rotații variabile, atunci intrările şi ieşirile registrului de rotații sunt independente.

Demonstrație: Conform cu diagrama fluxului de date a cifrului RC6 (figura 3.8) şi cu lista predecesorilor pentru cifrul RC6 (figura 3.9) se observă că predecesorii rotațiilor din RC6, fie că sunt fixe (lg*w*), fie că sunt variabile, pot fi fie un sumator şi o rotație fie o funcție pătratică. Aşa cum am demonstrat în teoremele 3.2, 3.3 şi 3.4, acestea au intrările şi ieşirile independente. Conform teoremei 3.1, respectiv a corolarului 3.1, a putem afirma şi de această dată că probabilitatea textului cifrat condiționată de probabilitatea mesajului clar este uniform distribuită în grup şi deci ieşirea Z este independentă față de intrarea X.

Calitatea aleatoare a cifrului RC6 a fost demonstrată pe baza listei predecesorilor. Deci, toate diagramele fluxurilor arhitecturilor cu aceeaşi listă de predecesori ca şi cea a cifrului RC6 au aceeaşi proprietate de propagare a tiparelor aleatoare.

Ca o consecință, se poate afirma că toate arhitecturile având o diagramă a arhitecturii a cărei listă de predecesori este identică cu lista predecesorilor diagramei fluxului cifrului RC6 produc date aleatoare şi independente pe toate arcele. Dacă se face o diferențiere a intrărilor în intrări pentru mesajul clar şi pentru chei, doar cheile trebuiesc alese în mod aleator, iar mesajul este suficient să fie independent doar de chei.

3.4. Capacităţi de testare concurentă ale cifrului RC6

Standardele de securitate recomandă ca echipamentul criptografic să fie testabil în mod concurent [BePi-82] [Fede-77]. Uzual capacitatea de autotest este realizată prin introducerea de redundanţă. Acest paragraf tratează unele proprietăţi ale codurilor care permit o metodă eficientă de test concurent.

Pentru ca autotestele concurente să fie de încredere, trebuie asigurat un anume grad de fiabilitate iniţială [Vlăd-89]. Pentru ca o componentă să poată fi verificată prin redundanţă hardware, egalitatea rezultatelor blocurilor identice nu este suficientă, ci se cere corectitudinea acestor rezultate. Cazul aceloraşi defecte de fabricaţie în ambele blocuri este exclus. În acest scop, corectitudinea cel puţin a unuia dintre blocuri trebuie validată printr-un autotest separat (off-line). Deci, o strategie de testare sigură trebuie să implementeze atât o testare concurentă cât şi o testare off-line [ITSE-93].

3.4.1. Redundanţă

Există mai multe posibilităţi de redundanţă în vederea obţinerii proprietăţilor de autotestare. Redundanţa hardware înseamnă în mod normal duplicarea componentelor. Se prevăd două blocuri identice cu aceleaşi intrări. Dacă rezultatele sunt identice, se poate considera corectitudinea ambelor blocuri; rezultate diferite indicând defectarea cel puţin a unuia dintre blocuri. În acest fel se poate obţine doar detectarea defectelor. Redundanţa triplă modulară permite corectarea erorilor prin utilizarea a trei blocuri. În cazul în care unul dintre rezultate este diferit iar celelalte două sunt identice se poate detecta blocul defect. Una dintre caracteristicile redundanţei hardware este identificarea corectă a erorii.

Redundanţa codului înseamnă folosirea codurilor detectoare de erori. La biţii de informaţie se adaugă câţiva biţi de control. Orice operaţie efectuată se va reflecta în biţii de control. Dacă rezultatul este o combinaţie validă atunci acesta se consideră a fi corect. Utilizarea unor coduri corectoare de erori permit corectarea unor defecte. Pentru un număr mare de biţi de verificare raportat la numărul de biţi de informaţie, probabilitatea aliasing descreşte spre zero.

3.4.2. Proprietăţi specifice cifrului RC6

3.4.2.1. Cifruri de involuţie

Una dintre proprietăţile cele mai importante urmărite în proiectarea unui algoritm de criptare este posibilitatea de implementare atât hard cât şi soft a acestuia. Un alt aspect la fel de important este similaritatea criptării şi decriptării, care permite folosirea aceluiaşi dispozitiv pentru ambele operaţii. Aşa cum am arătat în capitolul 2, această proprietate este îndeplinită şi de către cifrul RC6. Ideea de bază este utilizarea cifrurilor de involuţie [XLai-92].

Definiția 3.6. O funcție $In(\bullet,\bullet)$, $In:\{0, 1\}^m \times\{0, 1\}^k \rightarrow \{0, 1\}^m$ se numeşte cifru de involuție dacă pentru orice $z\in\{0, 1\}^k$, $In(In(x, z), z) = x$ pentru orice $x\in\{0, 1\}^m$.

Deci, un bloc de date de intrare arbitrar, notat cu x, aplicat unui cifru de involuție In, folosind cheia z, are ca rezultat un bloc de ieşire y de aceeaşi lungime. Dacă y se aplică funcției In pentru o cheie identică se obține: $In(x,z) = y$ şi $In(y,z) = x$. Figura 3.11 prezintă structura care transformă o parte a cifrului RC6 într-un cifru de involuție. Funcția *f(..)* se poate alege arbitrar şi nu influențează deloc proprietățile cifrului.

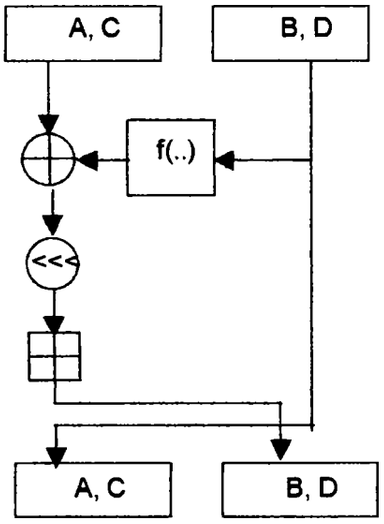


Figura 3.11. Cifrul de involutie RC6

Structura cifrurilor de involuție poate fi folosită pentru autotestul concurent în anumite situații. Se observă faptul că proprietatea de involuție creează invarianți: se obțin valori identice indiferent de datele şi cheile aplicate.

Să presupunem că nu una ci două iterații au fost implementate într-un circuit. Deci există două funcții de involuție identice. Oricând una dintre ele este nefolosită de procesul de transformare a datelor, i se poate aplica aceeaşi cheie şi ieşirea celeilalte structuri de involuție. Rezultatul trebuie să fie identic cu intrarea primei structuri de involuție.

Să presupunem că există o singură structură de involuție. Oricând este inactivă, unul dintre rezultate se poate aplica la intrare. Rezultatul trebuie să fie egal cu intrarea originală. Însă, acest test nu poate verifica defectele statice ale funcției *f(..)* în blocul XOR. Se reaminteşte faptul că *f(..)* se poate alege în mod arbitrar. Totuşi această verificare se poate folosi şi pentru testul structurii XOR. Dacă *f(..)* este implementată ca având mai multe niveluri, cele intermediare sunt testate.

3.4.2.2. Testarea concurentă a controller-ului

Autotestul concurent al controller-ului este întotdeauna posibil prin duplicare hard. Găsirea unei soluții mai simple este dificilă din cauza complexității inerente, spre exemplu adâncimea secvențială a automatelor cu stări finite. Analiza semnăturii tuturor ieşirilor se poate realiza doar în cazul în care secvența de ieşire este identică pentru toate execuțiile de test. Deci este fezabilă doar pentru o secvență determinată de intrări şi stări. După terminarea secvenței, semnătura rezultată se poate evalua concurent. Analiza semnăturii se poate realiza mai uşor cu un registru de deplasare cu mai multe intrări, obținut dintr-un LFSR (Linear Feedback Shift Register).

Dacă numărul semnalelor de ieşire depăşeşte o anumită limită, mărimea registrului de deplasare creşte nepermis de mult. Un analizor de semnătură poate să devină astfel mai mare decât însuşi controller-ul şi se poate realiza mai uşor prin duplicarea acestuia. Se poate recunoaşte o similaritate între controller şi analizorul de semnătură: ambele structuri sunt automate secvențiale cu o funcție de control complexă. Oricum, ieşirea controller-ului nu este readusă la intrare şi deci nu contribuie la semnătură. Ca urmare, primul pas spre unificarea funcției de controller cu cea de analizor de semnătură este eliminarea logicii de la ieşire. Aceasta se poate realiza simplu prin:

1. transformarea automatului original Mealy într-un automat echivalent Moore;
2. fiecare semnal de ieşire să provină direct dintr-un bistabil de stare.

În general pasul (2) sporeşte considerabil numărul bistabilelor de stare. Automatul finit rezultat este un automat Medwedjew, reprezentat în figura 3.12 [Yarb-97]. Pentru ieşirile identice în stări diferite ale unui automat Mealy sau Moore în cazul automatului Medwedjew se adaugă bistabile suplimentare pentru diferențierea stărilor. Ele nu contribuie la configurația ieşirilor dar sunt necesare pentru determinarea stării următoare.

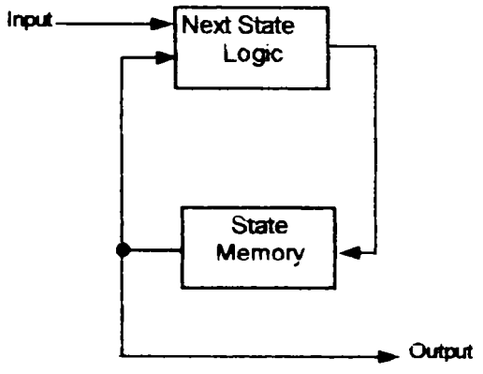


Figura 3.12. Automatul Medwedjew

Cu aceste modificări, automatul finit se comportă ca un registru LFSR cu o funcție de reacție mai complexă, obținută din logica de determinare a stării următoare a controller-ului. Funcția logică este în general neliniară. Un automat Medwedjew este deci un registru de deplarare cu reacție neliniară (NFSR – Nonlinear Feedback Shift Register).

Controlul secvenței este identic pentru toate iterațiile. Se poate proiecta un controller tipic care să genereze secvența de control iterativ pentru un ciclu. Pentru test concurent însă ciclurile nu sunt avantajoase. Un controller care ciclează necesită un numărător adițional pentru iterații. Totuşi distincte de controller, bistabilele numărătorului sunt de fapt bistabile de stare adiționale. Un controller Medwedjew care ciclează se comportă într-un mod nedorit: fiecare stare a ciclului este atinsă de un număr egal cu numărul de iterații. Acest lucru este nepotrivit pentru analiza semnăturii, care urmăreşte să atingă secvențe maxime ale stărilor distincte a regiştrilor.

Secvențializarea ciclurilor are ca urmare o configurație unică a bistabilelor pentru fiecare stare [John-89]. Validarea concurentă a controllerului se poate face astfel:

1. Se inițiază un test concurent;
2. Se execută secvența de verificare;
3. Se compară starea controllerului după această secvență cu semnătura cunoscută. În cazul egalității se validează funcționarea corectă (cu excepția identificării greşite).

Pentru un registru LFSR de lungime maximă, probabilitatea de identificare greşită este aproximativ $1/2^n$, unde n este numărul bistabilelor controllerului [BCFK-93]. Această afirmație nu poate fi generalizată pentru un registru de deplasare cu reacție neliniară. Pentru această aplicație şi cu un controller Medwedjew fără cicluri având un număr mare de bistabile, se consideră probabilitatea de identificare greşită ca fiind neglijabilă. Fezabilitatea acestei abordări pentru o implementare practică se poate demonstra spre exemplu prin simularea defectelor.

3.5. Concluzii

Pe baza analizelor unor algoritmi de cifrare, cum ar fi DES sau IDEA, în perspectiva realizării unei implementări eficiente a noului cifru RC6, propus ca viitor standard de criptare, în acest capitol am realizat o analiză de oportunitate pentru aplicarea tehnicilor BIST. În acest sens, teza cuprinde următoarele contribuții:

1) O sistematizare a proprietăților necesare stimulilor pseudoaleatori de testare şi pentru propagarea modelelor aleatoare în sistemele de cifrare;
2) Am enunțat cinci definiții urmate de numeroase exemple şi am prezentat un algoritm, toate fiind necesare pentru definirea unui graf potrivit pentru a

descrie diagrama fluxului de date al unui cifru, respectiv pentru construirea listei predecesorilor;

3) Am realizat o analiză originală al modului de organizare eficientă a autotestului unui circuit de cifrare RC6;

4) Am enunţat un număr de cinci teoreme originale şi un corolar, necesare pentru întocmirea listei predecesorilor şi demonstrarea caracterului aleator al cifrului RC6;

5) Am întocmit graful fluxului de date al cifrului RC6 şi diagrama fluxului de date a arhitecturii cifrului şi am arătat că dacă se face o diferenţiere a intrărilor în intrări pentru mesajul clar şi pentru chei, doar cheile trebuiesc alese în mod aleator, iar mesajul este suficient să fie independent doar de chei;

6) Am determinat structura care transformă partea corespunzătoare a cifrului RC6 într-un cifru de involuţie în scopul folosirii acesteia pentru autotestul concurent.

În concluzie se poate spune că în acest capitol am prezentat caracteristicile cele mai importante ale cifrului RC6 în scopul implementării acestuia într-un circuit de criptare hardware cu facilităţi de autotestare, aşa cum se va vedea în capitolul 5.

CAPITOLUL 4

CONFIGURAREA MEDIULUI EXPERIMENTAL ÎN VEDEREA IMPLEMENTĂRII HARDWARE A CRIPTOCHIP-ULUI

4.1. Logică programabilă

Circuitele integrate pe scară mică (SSI) şi medie (MSI) sunt adesea denumite circuite integrate cu logică aleatoare. Denumirea specificată separă aceste circuite - din a căror categorie fac parte porţile, bistabilele, numărătoarele, registrele - de memorii şi de circuitele cu logică programabilă. Circuitele integrate pe scară largă (LSI) includ memorii, microprocesoare şi circuite logice programabile sau PLD-uri (Programmable Logic Device). PLD-urile, indiferent de tipul lor au o caracteristică comună: toate pot fi configurate de către utilizator, astfel încât ele să execute o anumită funcţie [Mang-94].

La baza structurii unui PLD stă o matrice de celule identice din punct de vedere funcţional. În mod obişnuit, matricea de celule este formată din porţi ŞI, SAU şi adesea ea conţine şi bistabile. Unele circuite pot implementa doar funcţii combinaţionale, pe când altele pot implementa atât funcţii combinaţionale cât şi funcţii secvenţiale.

PLD este un nume generic dat unui grup de circuite integrate care pot fi configurate în scopul implementării unei funcţii logice. Circuitele interne ale PLD-urilor sunt structurate în timpul procesului de fabricaţie şi apoi acestea pot fi configurate de către utilizator, pentru a îndeplini anumite funcţii logice. ROM-urile, PROM-urile şi EPROM-urile sunt circuite programabile, dar acestea nu fac parte din categoria PLD-urilor.

Primele circuite logice programabile care au apărut au fost PLA-urile (Programmable Logic Array). PLA-urile conţin o matrice ŞI care realizează termenii produs ai unei funcţii şi o matrice SAU care realizează suma acestora. Atât termenii produs cât şi termenii sumă au conexiuni programabile. PLA-rile sunt circuite logice combinaţionale. Majoritatea PLA-urilor sunt programate de către utilizator cu ajutorul unui program care îi permite proiectantului să introducă funcţia booleană pe baza căreia apoi sunt realizate interconexiunile [Yarb-97] [MaMa-94] [Mang-94a]. În figura 4.1 este ilustrat un exemplu simplu de matricii de porţi ŞI şi SAU şi interconexiunile fuzibile utilizate pentru realizarea funcţiilor de ieşire.

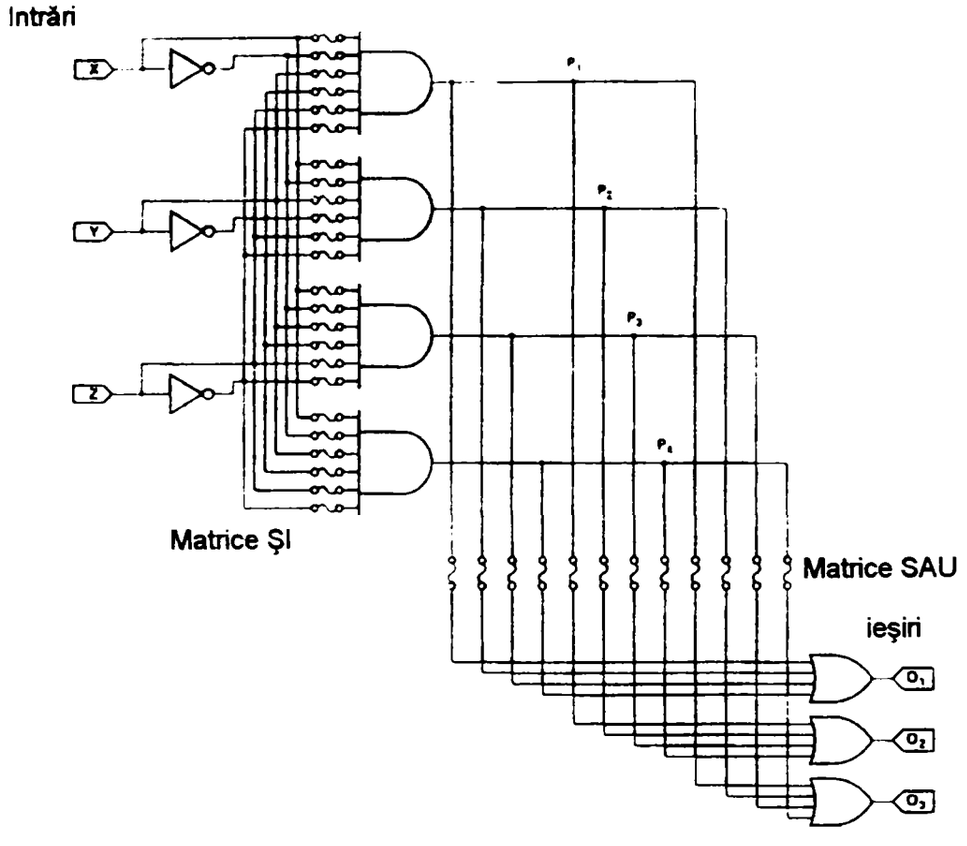


Figura 4.1. Interconexiunile porţilor ŞI şi SAU dintr-un PLA

PLA-urile au fost introduse de firma Philips in anii '70. Dezavantajele acestora în ce priveşte costul de fabricaţie şi performanţele scăzute de viteză au determinat înlocuirea lor cu circuitele PAL, circuite elaborate de firma Monolithic Memories.

PAL (Programmable Aray Logic) sunt circuite similare cu PLA-urile, cu excepţia faptului că matricea SAU este fixă şi nu programabilă. Acest lucru reduce flexibilitatea dar simplifică programarea. În figura 4.2 se pot observa diferenţele dintre un PLA şi un PAL. Matricile ŞI sunt identice în cele două circuite. X-urile marchează punctele de interconexiune. În ce priveşte matricea SAU, PLA are câte un X la fiecare intersecţie dintre termenii produs şi intrările porţii SAU. Fiecare astfel de intersecţie este programabilă în timp ce în circuitul PAL matricea SAU este fixă. Pentru implementarea circuitelor secvenţiale PAL-urile au prevăzute bistabile legate la ieşirile porţilor SAU. Introducerea circuitelor PAL au influenţat mult proiectarea circuitelor digitale şi ele au stat la baza unor arhitecturi noi, sofisticate. Circuitele PAL se produc în diferite variante ca mărimi şi număr de intrări şi ieşiri.

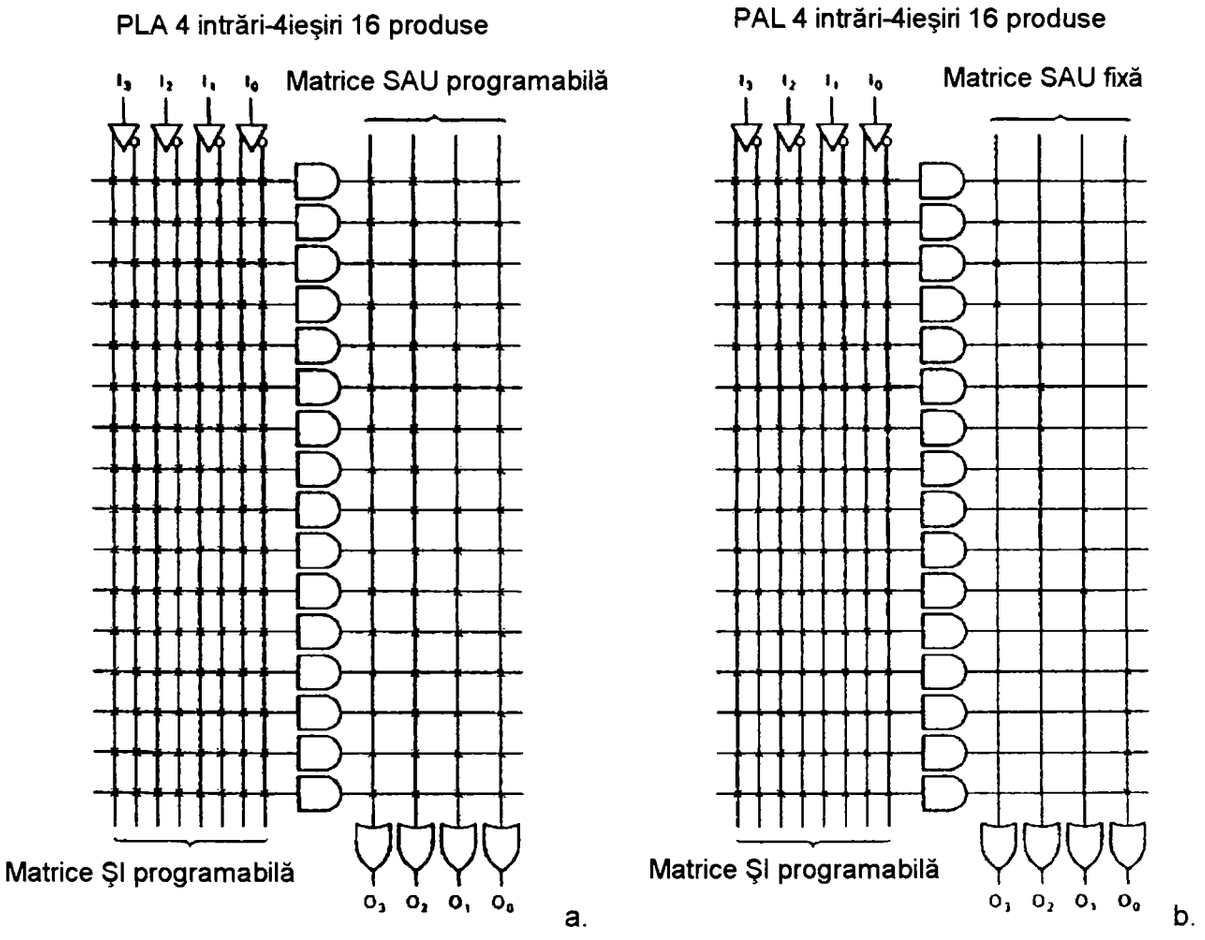


Figura 4.2. Comparație între un PLA (a) şi un PAL (b)

Un alt circuit din categoria circuitelor logice programabile este circuitul GAL (Generic Array Logic), introdus de firma Lattice Semiconductor Company. GAL conține o matrice ŞI programabilă, o matrice SAU fixă şi un nivel de ieşire numit, de către firma Lattice, macrocelulă logică de ieşire (output logic macro cell – OLMC). Această macrocelulă conține multiplexoare, bistabile şi buffer-e tristate de ieşire.

În figura 4.3 este prezentat un circuit GAL cu 8 ieşiri şi care poate avea până la 20 de intrări, iar în figura 4.4 este dată structura unei macrocelule de ieşire. Un OLMC conține trei multiplxoare de intrare şi un multiplexor de ieşire, un bistabil tristate şi o matrice SAU fixă. Intrările de la matricea ŞI programabilă sunt aduse la matricea SAU fixă care furnizează o intrare la o poartă SAU-EXCLUSIV. Una dintre intrările la matricea fixă SAU este ieşirea multiplexorului PT. AC_0 şi AC_1 sunt intrări de control ale multiplexoarelor PT şi TS. Ieşirea porții SAU-EXCUSIV este intrare a bistabilului D a celulei. Nivelul de excitație poate fi activ high sau low, funcție de semnalul XOR(n). Ieşirea bistabilului D şi a porții SAU-EXCUSIV constituie intrări ale multiplexorului de

ieşire OMUX. Acest lucru înseamnă că celula poate fi configurată fie ca o funcție combinațională fie ca o funcție registru, conform valorilor lui AC_0, respectiv $AC_1(n)$. Feedback-ul la matricea ŞI este furnizat de ieşirea Q', ieşirea tristate a OMUX sau de la o celulă adiacentă, sub controlul semnalelor AC_0 şi $AC_1(n)$. Ieşirea tristate poate fi controlată de OE sau o ieşire de termen produs direcționată prin TSMUX la tristate enable [Mano-84].

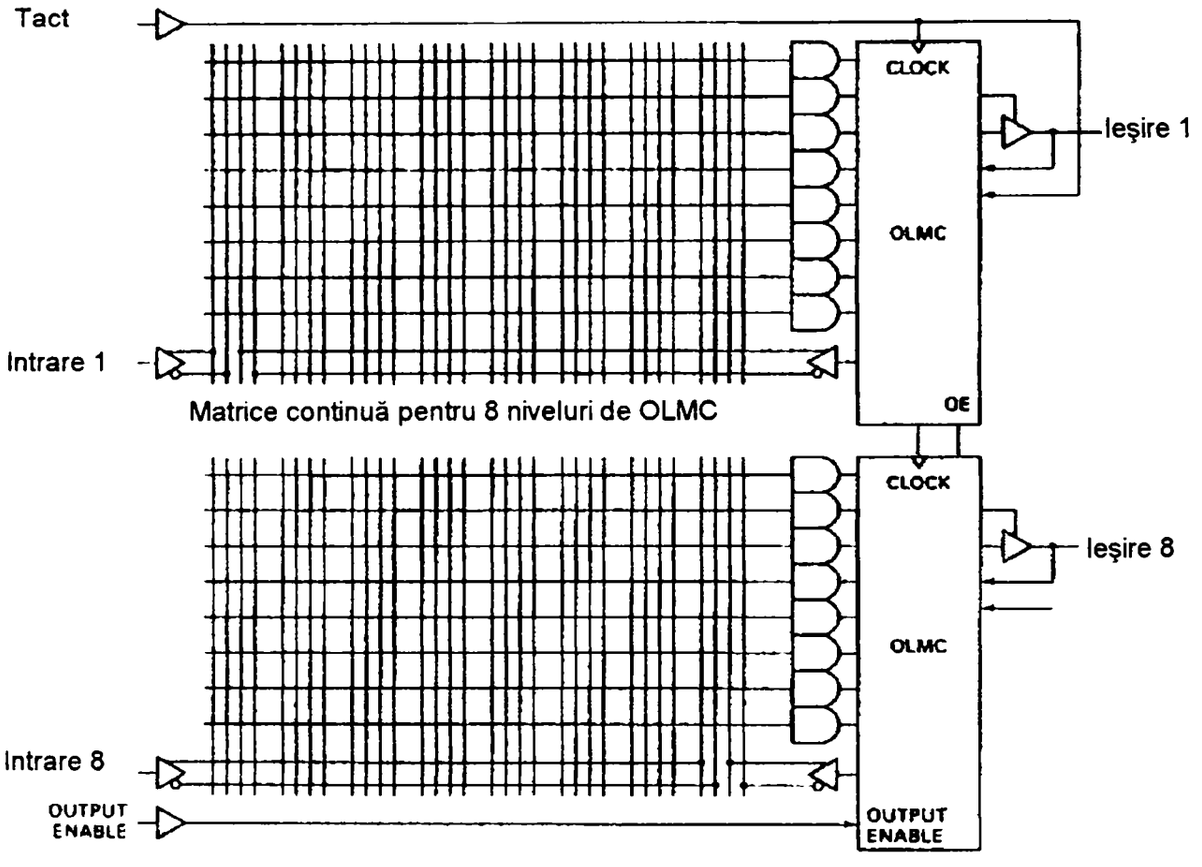


Figura 4.3. Schema logică a unui circuit GAL

Circuitele GAL fac parte din categoria circuitelor logice programabile reinscriptibile care pot fi reprogramate pe placă.

Primele circuite logice programabile (EPLD – Erasable Programmable Logic Devices) au fost proiectate de firma Altera în 1984. Aceste circuite combină facilitățile PAL-urilor şi ale EPROM-urilor într-un chip logic reprogramabil, care poate fi şters la raze ultraviolete. EPLD-urile Altera conțin o matrice programabilă ŞI, o matrice SAU fixă şi o celula I/O programabilă de către utilizator, celulă care conține bistabile, multiplexoare şi buffer-e tristate [Yarb-97]. Celulele de EPROM se găsesc în matricea ŞI şi partea de control a celulei I/O.

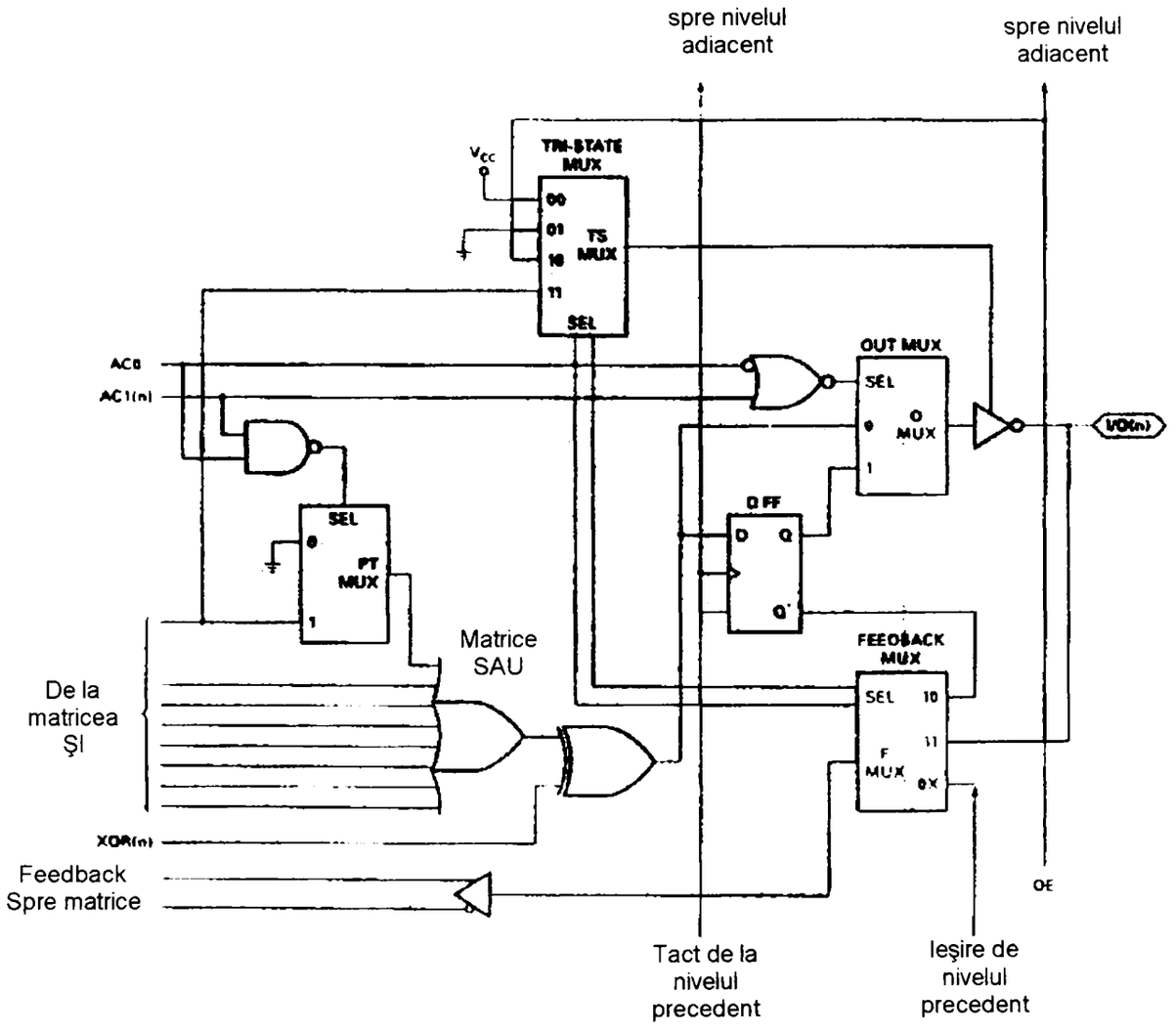


Figura 4.4. Macrocelula logică de ieşire a unui circuit GAL.

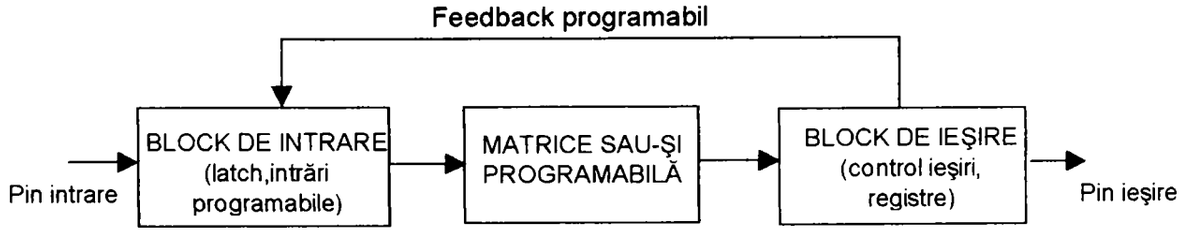


Figura 4.5. Structura generală a unui EPLD.

Tabelul 4.1. Caracteristici ale circuitelor PLD şi EPLD

Caracteristici	PLD-uri fuzibile	PLD-uri reprogramabile
Consum de putere	Ridicat (bipolar)	Scăzut (CMOS)
Porgramare	O dată	Ştergere cu UV, reprogramare
Densitate	Scăzută	Ridicată
Testabilitate	Dificilă	Facilă
Viteză	Ridicată	Scăzută
Securitate	Nu, schema poate fi reconstruită	Da, biţii de protecţie sunt implantaţi în silicon

Primul circuit EPLD al firmei Altera conţine echivalentul a 600 de porţi, are 16 macrocelule care conţin 20 de intrări, 16 ieşiri şi până la 16 bistabile. Bistabilele pot fi configurate ca fiind de tip D, T, R-S sau J-K. Structura generală a unui EPLD este dată în figura 4.5, iar în tabelul 4.1 sunt ilustrate avantajele şi dezavantajele acestuia comparativ cu PLD-urile.

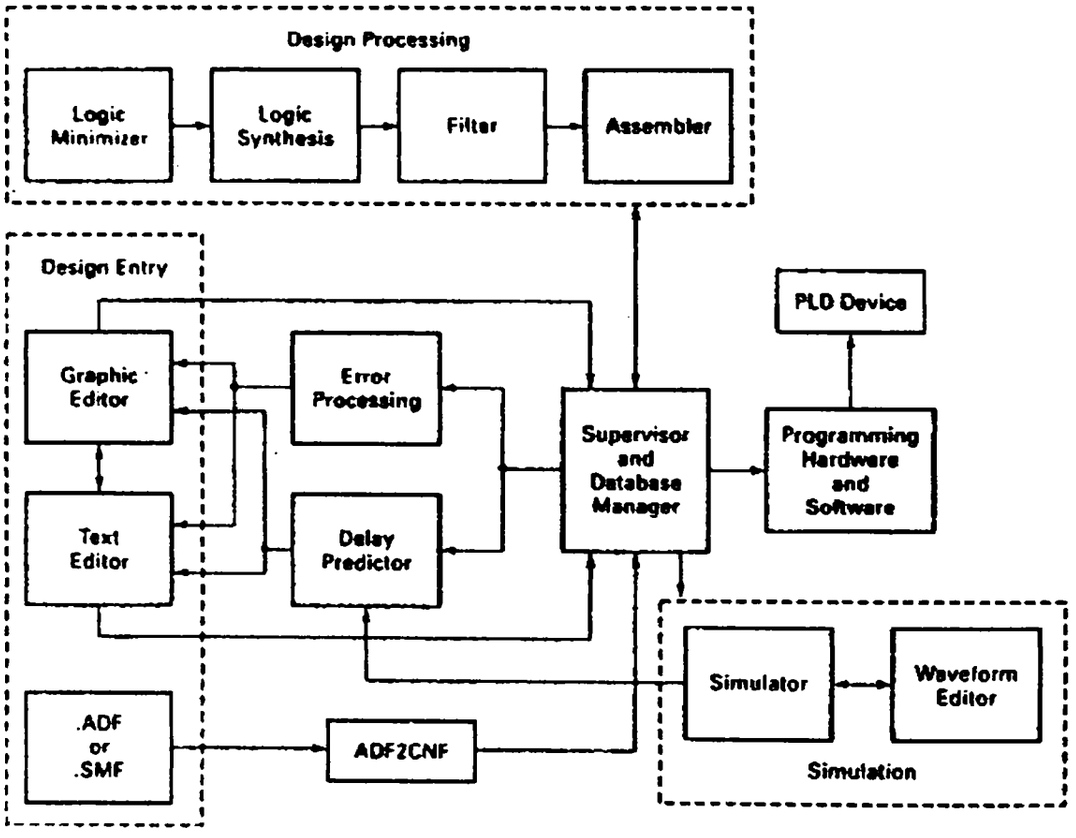


Figura 4.6. Sistemul de proiectare MAX+Plus

Fiecare firmă producătoare de EPLD-uri a elaborat şi câte un sistem soft propriu de elaborare a proiectelor cu aceste circuite. Schema bloc a sistemului soft MAX+Plus al firmei Altera este prezentată în figura 4.6 [Alte-95].

Proiectarea logică începe cu apelarea modulului de introducere a schemei. Introducerea se poate face fie pe bază grafică fie text. Editorul grafic permite introducerea schemei folosind simboluri grafice grupate într-o librărie de circuite SSI şi MSI. Editorul text permite utilizarea unui limbaj de descriere hardware AHDL (Altera hardware description language) [Alte-95]. Pentru implementarea schemei într-un EPLD compilatorul MAX+Plus face conversia fişierului care conţine informaţiile schemei introduse într-un fişier netlist. Orice eroare sesizată este semnalată şi utilizatorul este trimis înapoi la modulul de introducere a schemei. Apoi schema este optimizată şi este creat un fişier netlist de simulare. Acest fişier nou creat permite simularea schemei logice înainte ca aceasta să fie implementată într-un EPLD. Softul de simulare este prevăzut cu facilităţi de precizare ai timpilor de întârziere şi posibilitatea de a culege probe pentru a urmări comportarea circuitului. Se pot de asemenea vizualiza diagrame de timp.

Circuitele fabricate ulterior de firma Altera, şi anume cele din seriile MAX oferă un nivel mai ridicat de integrare şi performanţe marite. De exemplu circuitul EP1800 conţine 48 de macrocelule şi până la 64 de intrări şi 48 de ieşiri. Cu ajutorul circuitelor EPLD pot fi integrate scheme logice combinaţionale, maşini de stare, numărătoare şi registre.

4.2. Arii logice reconfigurabile (Field Programmable Gate Arrays – FPGA)

Ariile logice reconfigurabile reprezintă o altă categorie a circuitelor logice programabile. Cuvântul "field" din denumirea acestora precizează facilitatea de programare a matricii de porţi, pentru o anumită funcţie, de către utilizator şi nu de către producătorul circuitului. Cuvântul "array" sugerează faptul că este vorba de mai multe cloane şi linii de porţi care pot fi configurate de către utilizator. FPGA-urile conţin un număr mai mare de porţi echivalente decât EPLD-urile. De exemplu, circuitul XC3090 conţine 9000 de porţi echivalente. O poartă echivalentă este un ŞI cu două intrări. FPGA-urile diferă de matricile standard de porţi prin faptul că ele pot fi programate de către utilizator, în timp ce matricile standard sunt configurate de producător, necesitând astfel un timp mai îndelungat de producţie. În figura 4.7 este prezentată arhitectura tipică a unui FPGA.

Celula de bază a matricilor standard este mai simplă decât a FPGA-ului iar interconexiunea celulei se realizează printr-un proces de mascare [HaDu-98]. Matricile standard de porţi sunt eficient utilizate atunci când numărul de produse este foarte ridicat, rezultând astfel un cost scăzut.

Caracteristica de programare de către utilizator este asigurată de conexiunile programabile. Prima conexiune programabilă realizată a fost siguranţa fuzibilă utilizată în PLA-uri. Acestea sunt utilizate şi la ora actuală în unele circuite de

dimensiuni mai mici. Pentru circuitele integrate de dimensiuni mai mari se utilizează tehnologia CMOS iar pentru realizarea conexiunilor se folosesc diverse metode. De exemplu, pentru FPGA-uri conexiunile sunt realizate cu SRAM-uri (Static RAM) şi antifuzibile. În figura 4.8 sunt ilustrate două aplicaţii ale conexiunilor comandate de SRAM, una pentru controlul nodurilor conexiunilor cu tranzistori de trecere şi cealaltă pentru liniile de selecţie ale multiplexoarelor care comandă intrările blocurilor logice.

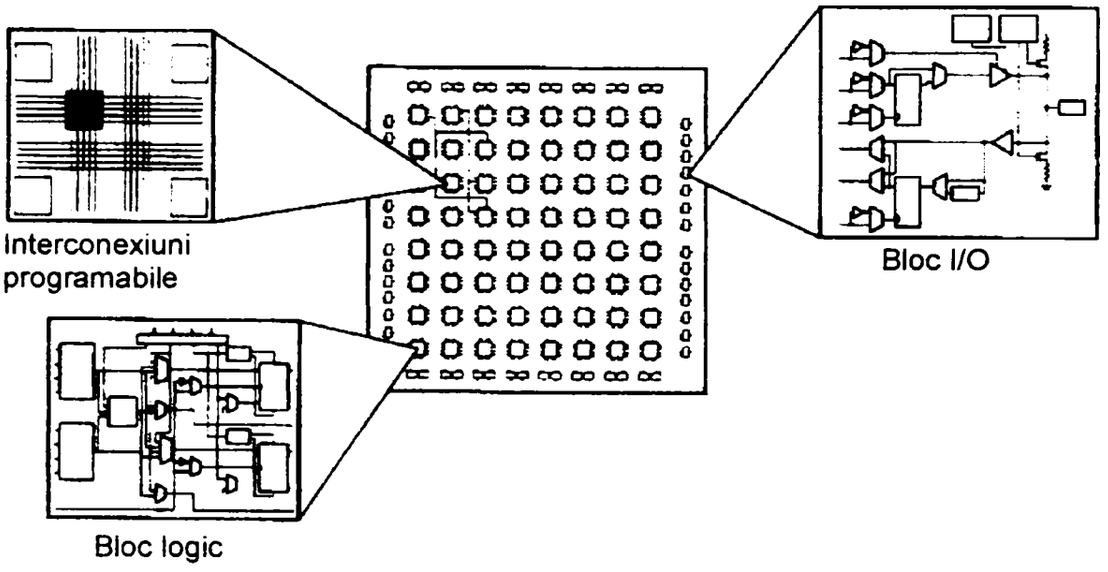


Figura 4.7. Structura FPGA.

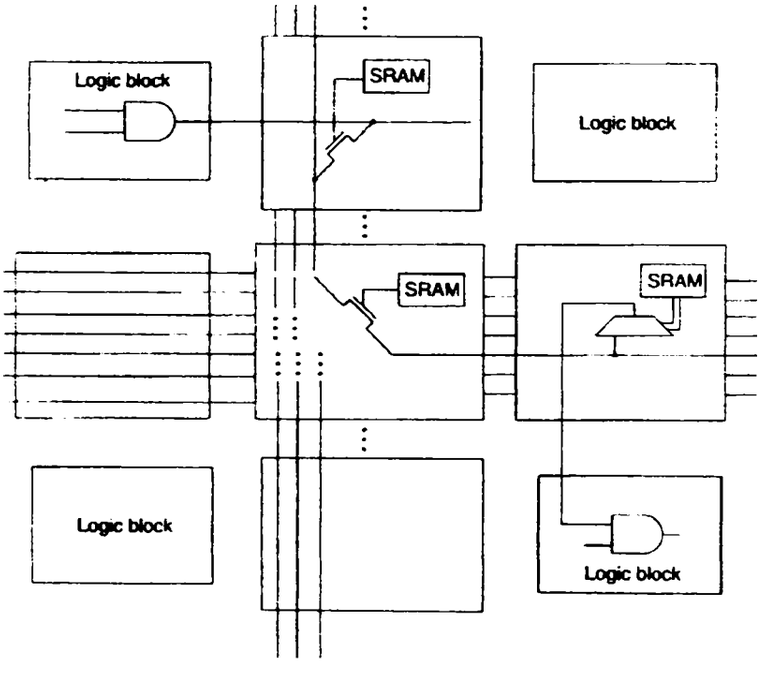


Figura 4.8. Conexiuni programabile comandate de SRAM.

În figura 4.8 este prezentată conexiunea unui bloc logic (reprezentat de poarta ŞI din colţul stânga sus) cu un alt bloc logic, realizată prin două conexiuni cu tranzistori de trecere şi apoi un multiplexor, toate controlate de celule SRAM.

Proiectarea cu circuite PLA şi PAL necesită un soft de translatare (pentru conversia intrărilor în fuzibile arse) şi hard de programare (pentru arderea fuzibilelor). Circuitele EPLD necesită instrumente soft mai dezvoltate, decât PLA-urile şi PAL-urile, pentru implementare, iar trecerea la FPGA-uri presupune instrumente de proiectarea chiar mai complexe decât cele necesare în proiectarea cu EPLD-uri.

Datorită complexităţii ridicate a circuitelor, proiectarea cu circuite FPGA necesită instrumente suplimentare de proiectare. Astfel ele utilizează un translator de tehnologie pentru a transforma porţile logice de bază în blocuri logice ale FPGA-ului, un instrument de plasare pentru a alege blocurile logice specifice şi un ruter pentru alocarea segmentelor de legătură care realizează interconectarea blocurilor logice.

Există două arhitecturi de bază ale FPGA-urilor, arhitecturi oferite de cele două firme producătoare: Xilinx şi Actel [Acte-92] [Xili-92]. Ambele au la bază celule sau module care pot fi programate de către utilizator şi ambele necesită matrici de comutare pentru interconexiunea celulelor şi modulelor. Acestea sunt însă singurele asemănări.

Arhitectura Xilinx are la bază blocuri logice configurabile, blocuri I/O şi un chip extern de memorie pentru realizarea funcţiei logice [Xili-98a]. Blocurile logice configurabile (CLB – Configurable Logic Block) pot fi folosite după dorinţă şi interconectate cu alte CLB-uri pentru a construi întregul circuit. În chip-ul extern de memorie este memorată configuraţia matricii de comutare. Plasarea informaţilor de interconectare într-o memorie (EPROM sau RAM) permite efectuarea de modificări, ceea ce înseamnă că circuitul poate fi reprogramat prin simpla modificare a configuraţiei datelor din memorie. Blocurile logice sunt interconectate prin canale de rutare orizontale şi verticale (vezi figura 4.7).

Structura oferită de Actel are o celulă de bază mai simplă decât CLB-ul Xilinx, numită modul logic (LM), dar matricea de comutare este mai complexă [Acte-92]. Dezavantajul acestei arhitecturi constă însă în faptul că nu permite reprogramarea; circuitul odată configurat, nu mai poate fi schimbat.

FPGA-urile sunt momentan cele mai răspândite circuite logice programabile. Ele au fost introduse în 1985 de firma Xilinx, firmă care şi la ora actuală oferă cea mai completă soluţie de proiectare. Primele circuite realizate de firma Xilinx au fost cele din seria XC2000; XC 2064 cu 1200 de porţi şi XC2018 cu 1800 de porţi. A doua generaţie de circuite o reprezintă cele din seria 3000, cu o densitate de până la 9000 de porţi şi realizate în tehnologie CMOS. La ora actuală, nivelul ridicat de integrare al acestor dispozitive - până la 500.000 porţi - permite implementarea unor structuri complexe [Xili-98c]. Un avantaj important al utilizării FPGA-urilor este timpul scurt necesar pentru proiectare precum şi facilităţile de reprogramare ale acestor dispozitive.

Xilinx produce dispozitive cu un număr diferit de porţi şi cu diverse structuri arhitecturale [Xili-98c]. Familiile de circuite produse şi câteva din caracteristicile lor sunt enumerate în tabelul 4.2.

Tabelul 4.2. Familii de circuite Xilinx şi caracteristicile acestora

Arhitectura	Nr. de porţi logice	Nr. maxim de CLB-uri	Nr. maxim de IOB-uri
XC 3000A	1.000-7.500	484	176
XC3100A	1.000-7.500	484	176
XC4000E	2.000-45.000	784	256
XC4000EX	18.000-130.000	2.300	384
XC4000XL	18.000-130.000	2.300	384
XC5200	2.000-15.000	484	244

Dispozitivele utilizate pentru integrarea structurilor prezentate în lucrare sunt XCV1000 si conţin 1.200.000 porţi echivalente.

4.2.1. Descrierea seriei XC 4000

Structura FPGA-urilor are la bază trei module: blocuri logice configurabile (CLB), blocuri de intrare/ieşire (IOB) şi matrici de comutare pentru interconexiuni. În figura 4.9 este prezentată shema bloc a structurii matricii de porţi programabile.

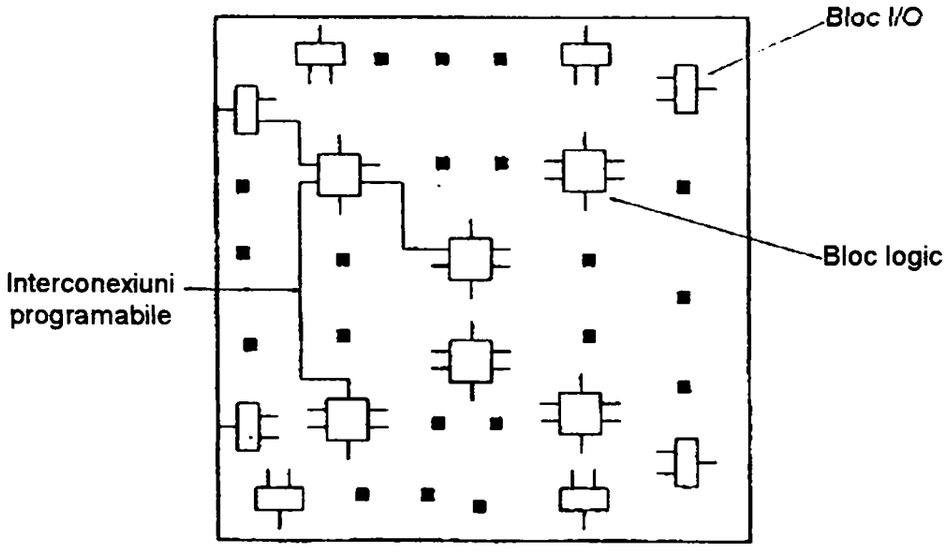


Figura 4.9. Structura FPGA-ului Xilinx

Structura unui CLB are la bază un tabel de căutare ("lookup table"). Un tabel de căutare este o matrice de memorie cu lățimea de un bit; liniile de adresă ale memoriei sunt intrări ale blocului logic iar ieşirea memoriei de 1 bit este ieşirea tabelului de căutare. Un tabel de căutare cu k intrări corespunde unei memorii de 2^kx1 bit iar utilizatorul poate realiza orice funcție logică cu k intrări prin programarea tabelului de adevăr al funcției logice direct în memorie. În configurația prezentată în figura 4.10, un CLB din circuitul XC4000 conține două tabele de căutare cu patru intrări alimentate de intrările CLB-ului şi un al treilea tabel de căutare care are ca intrări ieşirile celorlalte două. Această structură permite CLB-ului să implementeze funcții logice cu până la 9 variabile, două funcții separate de câte patru variabile, sau alte variante. Fiecare CLB conține de asemenea două bistabile.

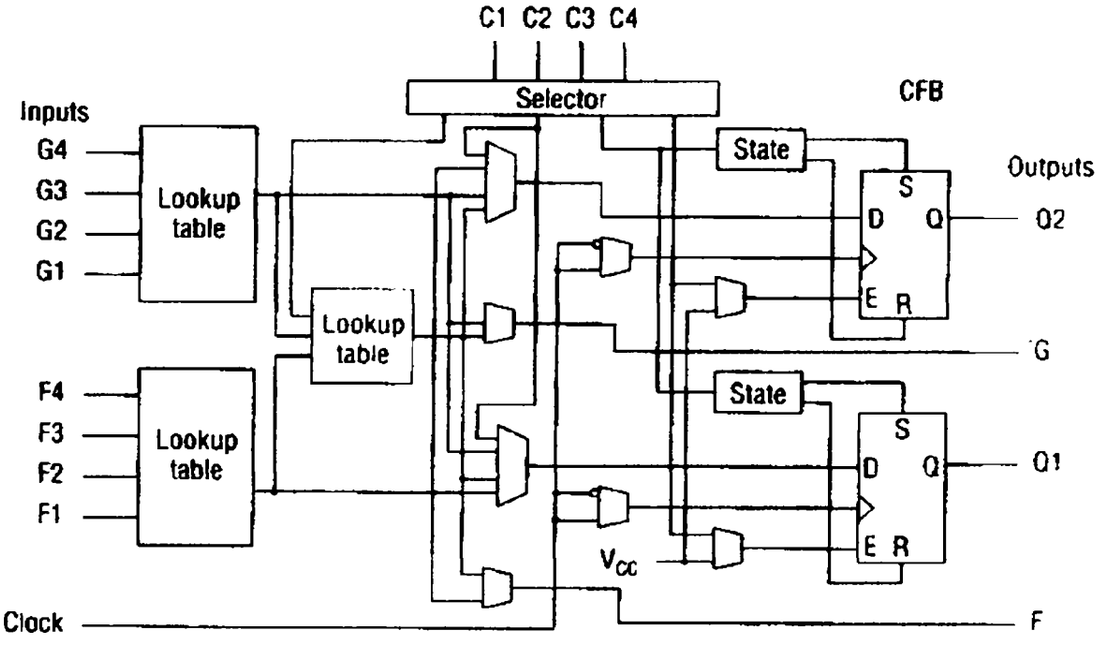


Figura 4.10. Shema simplificată a unui CLB

Circuitele XC4000 sunt proiectate astfel încât să permită integrarea unor sisteme. De exemplu, fiecare CLB conține circuite care permit să se efectueze eficient operații aritmetice (adică, un circuit care implementează operații "fast carry" pentru circuite sumatoare). De asemenea, utilizatorul poate configura tabelele de căutare ca celule RAM read/write. Fiecare chip XC4000 include suprafețe ŞI întinse, distribuite în jurul matricii de blocuri logice, pentru a facilita implementarea de blocuri de circuite, ca de exemplu decodificatoare mari [Xili-98b].

În afara părții de logică, un FPGA se remarcă prin structura de interconectare. O caracteristică a interconexiunilor chip-ului XC4000 o reprezintă canalele orizontale

şi verticale. Fiecare canal conţine segmente de legătură scurte, care traversează un singur CLB (numărul de segmente din fiecare canal diferă la fiecare circuit din familia XC4000), segmente mai lungi care traversează două CLB-uri şi segmente foarte lungi care traversează întreaga lungime sau lăţime a chip-ului. Pentru conectarea intrărilor şi ieşirilor CLB-urilor la segmentele de legătură sau pentru conectarea a două segmente de legătură se folosesc conexiunile programabile. În figura 4.11 este prezentată o mică secţiune a unui canal de rutare dintr-un XC4000 [Vasa-98]. Figura prezintă doar segmentele de legătură dintr-un canal orizontal - fără canalele verticale de rutare -, intrările şi ieşirile CLB-ului şi conexiunile de rutare. Un punct important al interconectării Xilinx este acela că semnalele trebuie să treacă prin conexiuni pentru a ajunge de la un CLB la altul, iar numărul total de conexiuni traversate depinde de setul particular de segmente de legătură folosite. Astfel, performanţele de viteză ale unui circuit implementat depinde în parte de modul în care instrumentele CAD alocă segmentele de legătură semnalelor individuale [Xili-98b].

Multiplexoarele controlate prin program sunt utilizate pentru dirijarea datei în interiorul CLB-ului.

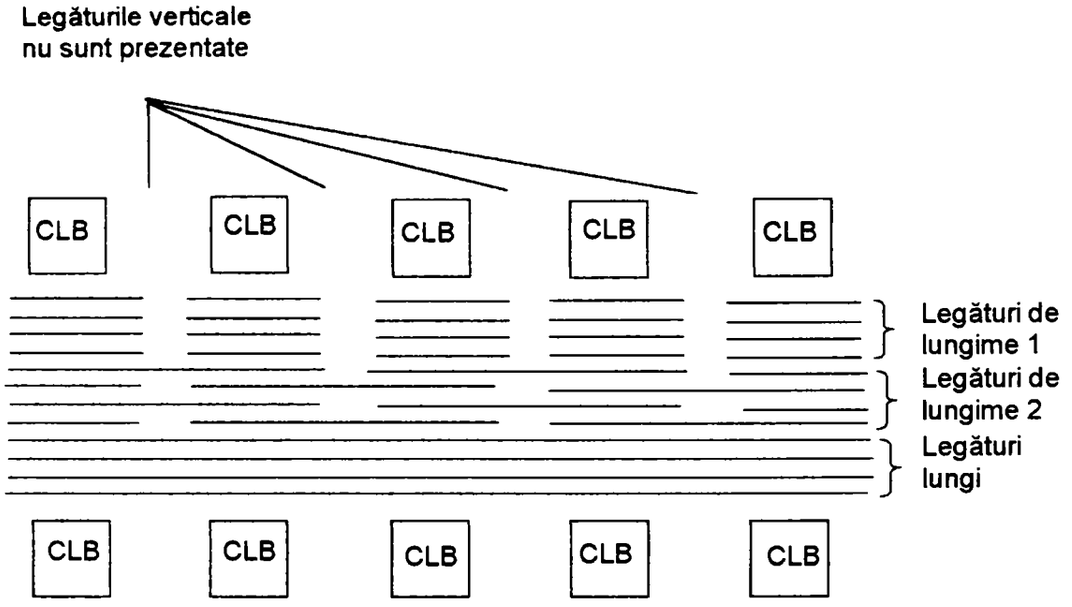


Figura 4.11. Segmentele de legătură din circuitul XC4000.

IOB constituie o interfaţă programabilă între matricea internă de CLB-uri şi pinii externi ai echipamentului. Funcţiile logice sunt determinate de un program de configurare memorat în celulele de memorie statică interne (inaccesibile utilizatorului). Configuraţia este realizată fie în mod activ de către FPGA prin citirea unei memorii EPROM (serie/paralel), sau configuraţia este înscrisă de un microprocesor, microcontroler sau un alt FPGA.

4.2.2. Structura blocurilor logice configurabile

Blocurile logice configurabile (CLB) furnizează elementele funcționale şi realizează structura logică proiectată. În figura 4.12 este prezentată structura unui CLB dintr-un circuit XC 4000.

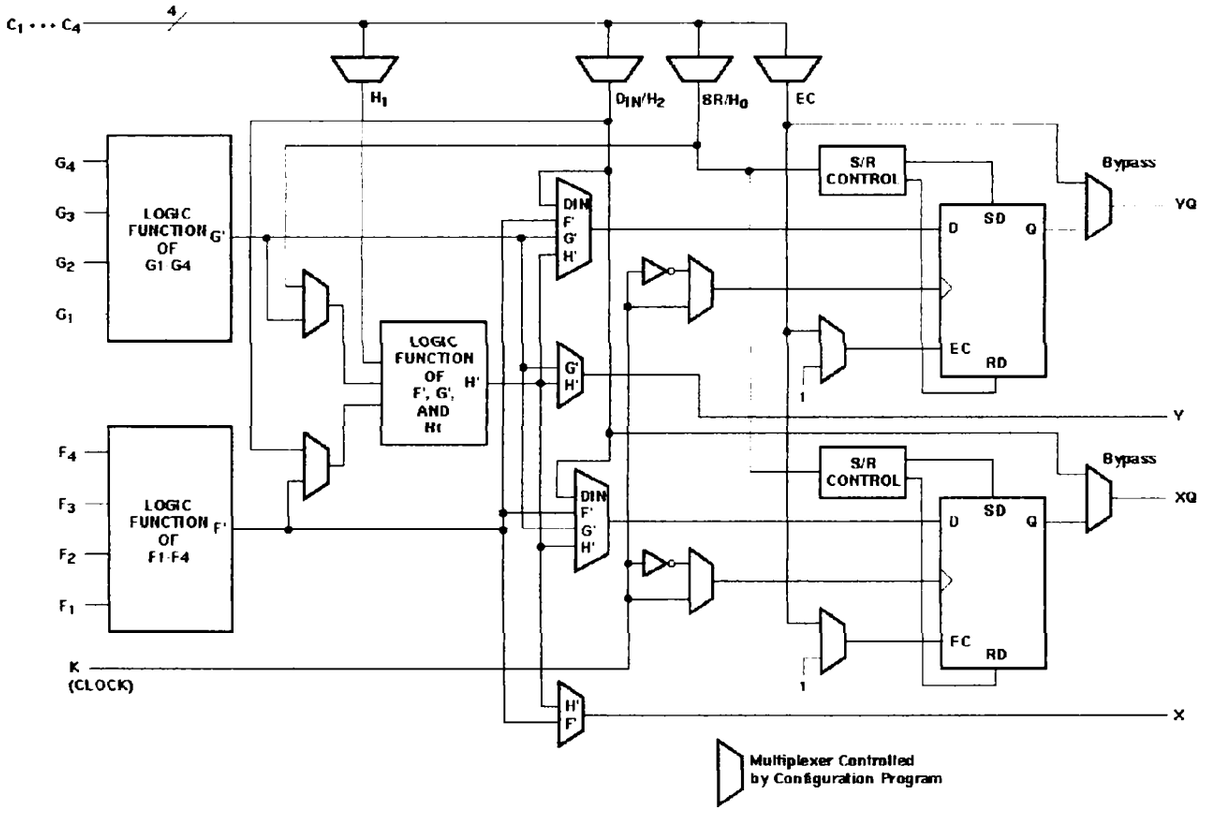


Figura 4.12. Structura unui CLB din dispozitivul XC4000

Majoritatea funcțiilor logice combinaționale au patru sau mai puține variabile Acesta este şi motivul pentru care fiecare CLB a fost prevăzut cu două generatoare de funcții (F, G) cu câte patru intrări. Este prevăzut şi un al treilea generator de funcții (H). Acesta are trei intrări care pot fi sau nu ieşiri ale lui F sau G, sau pot fi externe CLB-ului [Xili-98b]. Astfel, CLB-ul poate implementa funcții cu până la nouă variabile, cum ar fi generatoarele de paritate sau comparatoarele de identitate.

Fiecare CLB conține două elemente de memorare care pot fi utilizate pentru memorarea ieşirilor generatorului de funcții sau pot fi utilizate independent. Aceste elemente de memorare pot fi configurate ca bistabile sau, în circuitele din familia XC4000X, ele pot fi configurate şi ca latch-uri. DIN poate fi utilizat ca intrare directă la oricare dintre cele două elemente de memorare. H1 îl poate comanda pe celălalt prin generatorul de funcții H. Ieşirile generatorului de funcții poate de asemenea comanda

două ieşiri independent de ieşirile elementului de memorie. Această versatilitate măreşte capacitatea logică şi simplifică rutarea.

Fiecare generator de funcţii are patru intrări independente, notate F1–F4 respectiv G1–G4, şi ieşirile F' respectiv G'. Generatoarele de funcţii sunt implementate ca tabele de memorii (memory look-up table). Astfel, timpul de propagare este independent de funcţia implementată. Al treilea generator de funcţii, notat cu H', poate implementa orice funcţie booleană de trei variabile. Două dintre aceste intrări pot fi ieşirile F' şi G' ale generatoarele de funcţii. Alternativ, una sau ambele intrări pot fi semnale externe blocului (H2, H0). A treia intrare, H1, trebuie să provină din exteriorul blocului. Semnalele de ieşire ale generatoarelor de funcţii pot fi şi ieşirile CLB-ului. Ieşirile CLB-ului, F' sau H', sunt conectate la ieşirea X, iar G' sau H' sunt conectate la ieşirea Y.

Versatilitatea generatoarelor de funcţii din blocul CLB măreşte în mod semnificativ viteza de lucru. În mod adiţional programul de implementare-translatare poate trata independent fiecare generator de funcţii. Această flexibilitate măreşte utilizarea celulelor.

Implementarea funcţiilor cu multe variabile într-un singur CLB reduce atât numărul de blocuri necesare cât şi întârzierile semnalelor.

Intrarea asincronă a elementului de memorie (SR) poate fi configurată ca set sau reset. Această opţiune de configurare determină starea în care devine operaţional fiecare bistabil după configurare. De asemenea el determină efectul unui impuls global de Set/Reset în timpul operării normale şi efectul unui puls la pinul SR a CLB-ului. Starea de set/reset poate fi specificată independent pentru fiecare bistabil.

Multiplexoarele interne ale CLB facilitează implementarea funcţiilor alese. Multiplexoarele sunt controlate de semnalele (C1-C4), sau de semnalele interne rezultante (H1, DIN/H2, SR/H0 şi EC). Oricare dintre aceste semnale (C1-C4) sunt sursa semnalelor de control intern. Când funcţia logică este neinhibată atunci aceste semnale sunt:

- EC – Clock Enable
- SR/H0 – Set/Reset asincron sau intrarea H0 a generatorului de funcţie H;
- DIN/H2 – intrarea directă H2 a generatorului de funcţii H
- H1 – intrare directă H1 a generatorului de funcţie H.

Dacă blocul este configurat ca şi memorie RAM/ROM cele patru intrări au următoarea semnificaţie:

- EC – Clock Enable
- WE – Write Enable
- D0 – intrare dată pentru generatorul de funcţii F şi/sau G
- D1 – intrare dată pentru generatorul de funcţii G (mod RAM/ROM 16x1 şi 16x2) sau bitul de adresă 2^4 (mod 32x1)

Abundenţa bistabililor din seria XC4000 permite realizarea proiectelor cu pipe-line. Această posibilitate creşte performanţa circuitelor prin implementarea proiectului

în subfuncții, execuția lor în paralel şi transmiterea rezultatelor prin pipe-line la subfuncția următoare. Bistabilii se pot utiliza ca registre, bistabili de uz general, registre de shift, fără blocarea generatoarelor de funcții, care realizează în paralel funcții logice diferite, eventual independente de funcțiile realizate de bistabili. Această posibilitate creşte cu mult capacitatea circuitelor.

Opțional generatoarele de funcții F' şi G' ale blocurilor configurabile CLB se pot utiliza ca celule de memorie RAM/ROM. Memoriile implementate în CLB sunt extrem de rapide [Xili-92], [Xili-98h]. Timpul de citire este apropiat de timpul de întârziere al celulei CLB, iar timpul de înscriere este puțin mai mic decât în cazul implementării în circuite reale de memorie.

Fiecare generator de funcții din CLB (Fşi G) conține circuite aritmetice dedicate generării rapide a bitului de transport. Acest semnal de ieşire este conectat la blocul configurabil adiacent. Realizarea legăturii bitului de transport este independentă de resursele de conectare. Circuitul logic dedicat realizării bitului de transport măreşte performanțele circuitelor de adunare/scădere şi de numărare [Xili-98d].

Cele două generatoare de funcții pot fi configurate ca sumatoare pe doi biți cu biți de transport interni, sumatoare care pot fi expandate la orice mărime. Generarea rapidă a biților de transport în comparație cu metodele clasice este atât de rapidă încât la sumatoarele de 16/32 biți metodele clasice devin inutilizabile. Această logică de propagare este cel mai însemnat avantaj oferit de aceste circuite; ea permite realizarea unor circuite numărătoare cu o frecvență de lucru de până la 70 MHz. Logica de transport poate fi în ambele sensuri în interiorul circuitului. Propagarea bitului de transport se realizează între celule în seria XC4000 X conform figurii 4.13.

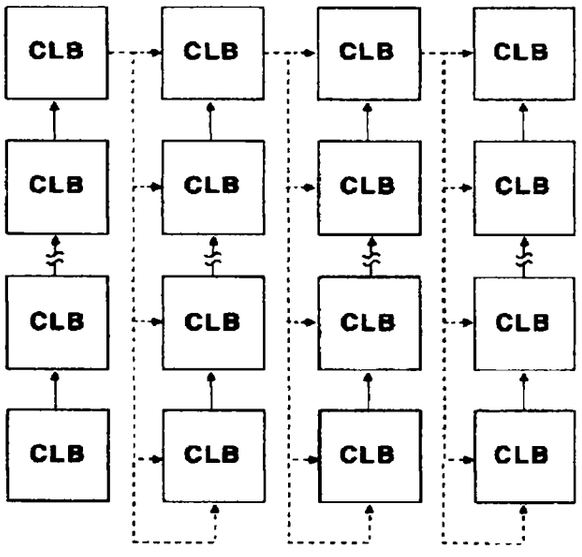


Figura 4.13. Propagarea bitului de transport

Blocurile configurabile de intrare/ieşire (IOB) realizează interfaţa între mediul exterior şi structura internă a circuitului FPGA. Fiecare IOB controlează un pin al circuitului integrat. Blocurile de intrare/ieşire se pot configura ca şi port de intrare, port de ieşire sau port bidirecţional. În figura 4.14 este dată structura simplificată a blocului IOB a circuitelor din seria XC4000X.

Semnalele I1 şi I2 furnizează semnalele de intrare spre interiorul circuitului FPGA. Selectarea bistabilului dorit se face la introducerea schemei electrice, prin selectarea simbolului dorit [Xili-98b]. Semnalul de inhibare ale bistabilelor din IOB poate fi configurat astfel încât să fie comun sau separat pentru cele două registre. Semnalele de ieşire pot fi conectate direct la ieşirea pinului sau la bistabilul de ieşire din IOB.

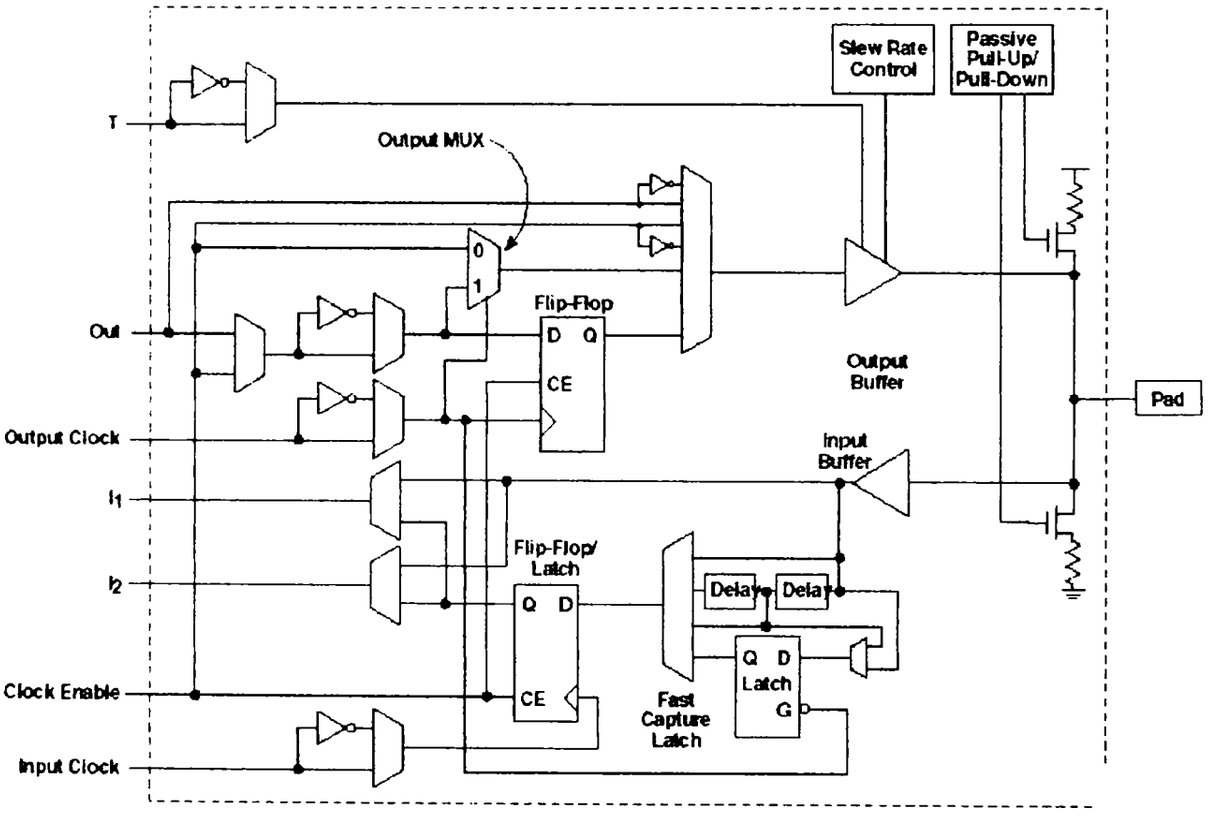


Figura 4.14. Structura IOB

Fiecărui CLB îi este asociat o pereche de buffere de impedanţă ridicată (figura 4.15). Aceste buffere se pot utiliza pentru a conecta blocul CLB la linia orizontală lungă, pentru a realiza funcţii cablate "or/and". La fel se pot implementa reţele multiplexate sau bus bidirecţional salvând astfel resursele logice ale CLB. Un alt buffer cu acces similar este aşezat în dreapta fiecărui bloc de ieşire/intrare. Buffer-ele

three state se pot configura în următoarele trei moduri: buffer three state standard, funcție cablată ŞI cu intrare pe pinul I sau funcție cablată or-and [Xili-98e].

Toate conexiunile interne sunt compuse din segmente de metal cu puncte de cuplare programabile şi matrice de cuplare pentru realizarea legăturilor interne. Resursele de conectare sunt structurate în matrici ierarhice pentru a realiza implementarea legăturilor într-un mod eficient.

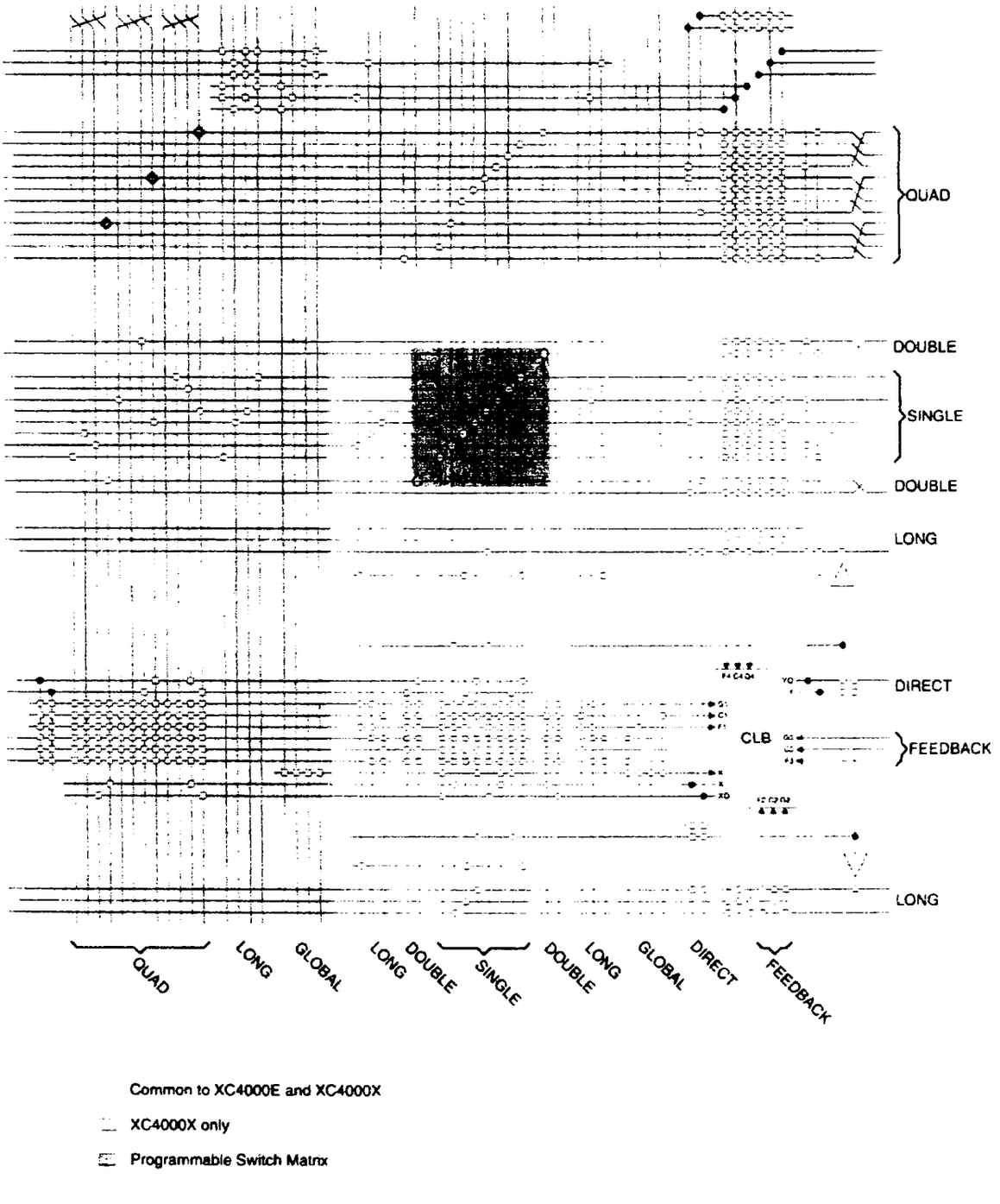


Figura 4.15. Interconexiunile programabile la seria XC4000.

Există următoarele tipuri de interconexiuni:

- Realizarea legăturilor între blocurile CLB este asociată rândurilor şi coloanelor matricii CLB (figura 4.16). Intrările şi ieşirile în blocurile configurabile sunt distribuite în toate cele patru direcţii, pentru ca realizarea conexiunilor să fie cât mai flexibilă. Poziţia intrărilor şi ieşirilor CLB este interschimbabilă pentru evitarea congestiunilor în timpul plasării şi realizării conexiunilor.

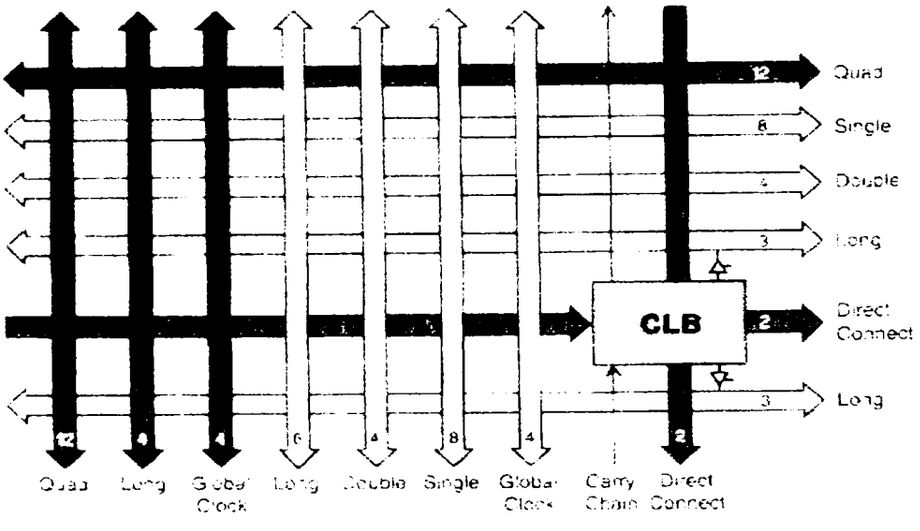


Figura 4.16. Realizarea legăturilor între blocurile CLB.

Liniile de simplă şi dublă lungime verticale şi orizontale se intersectează în aşa numita matrice de cuplare programabilă (Progammable Swiching matrix - PSM). Fiecare matrice este alcătuită din tranzistori de trecere utilizaţi pentru realizarea conexiunilor între linii (figura 4.17). De exemplu dacă o linie desimplă lungime intră în matricea de cuplare în colţul din stânga sus, atunci aceasta poate fi cuplată în orice combinaţie tot cu o linie de lungime simplă. În mod similar, o linie de dublă lungime poate fi conectată tot la o linie de dublă lungime. Liniile de simplă lungime permit interconectarea rapidă a blocurilor adiacente. Fiecărui bloc îi sunt asociate opt linii verticale şi opt linii orizontale de simplă lungime. Conectarea liniilor de simplă lungime este arătată în figura 4.18 iar realizarea conexiunilor este arătată în figura 4.15 [Xili-98c]. Liniile de simplă lungime introduc întârzieri în propagarea semnalelor ori de câte ori intră într-o matrice programabilă. Din această cauză ele nu se utilizează la realizarea legăturilor la distanţă. Liniile de lungime dublă realizează o grilă din segmente de metal. Fiecare linie are lungimea dublă în comparaţie cu liniile simple. Liniile duble sunt grupate în perechi şi intră în fiecare a doua matrice (figura 4.18). Fiecărui CLB i se asociază patru linii verticale şi patru orizontale, pentru realizarea conexiunilor (figura 4.15). Liniile lungi,

asemănător liniilor de dublă lungime, formează o grilă de metal care acoperă toată aria circuitului.

- Realizarea legăturilor între blocurile IOB formează un cordon în jurul matricii CLB (VersaRing), care conectează pinii I/O cu blocurile logice interne
- Conexiunile globale sunt compuse din rețele dedicate, proiectate pentru distribuirea rapidă a semnalelor de comandă şi control utilizate în proiect. Aceste semnale au un timp de propagare minim.

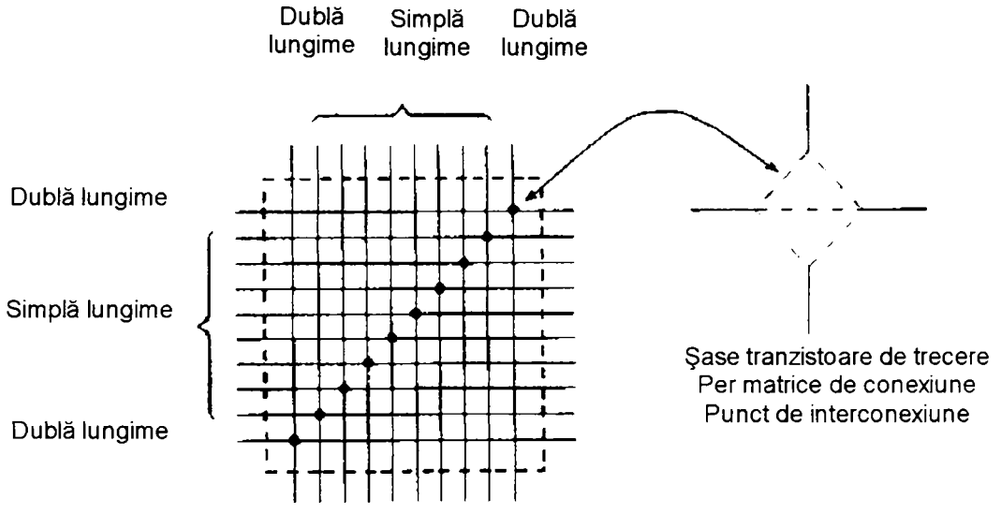


Figura 4.17. Matricea programabilă de cuplare

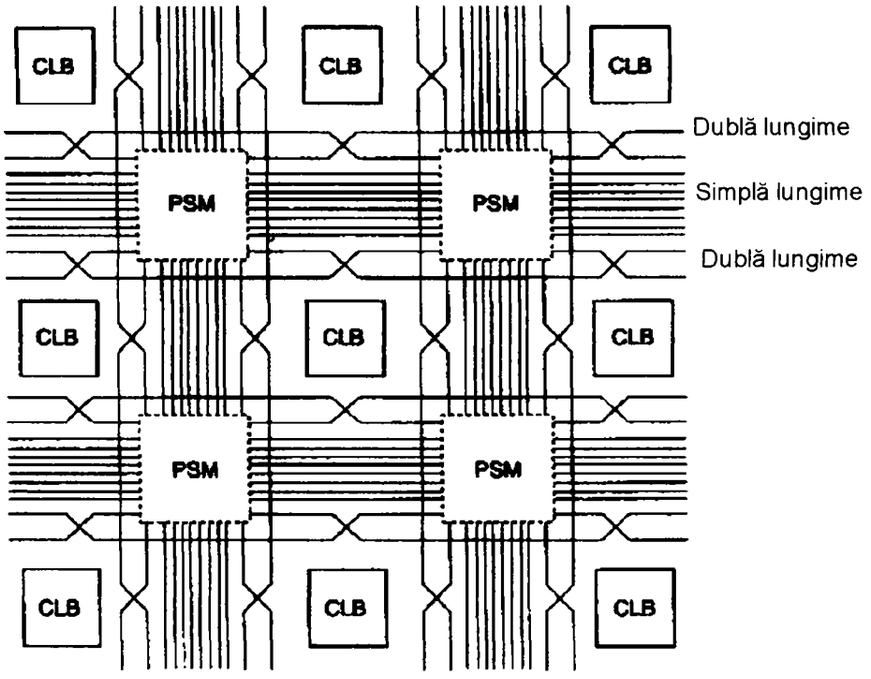


Figura 4.18. Linii de simplă şi dublă lungime cu matricile programabile.

4.3. Circuitele FPGA din familia Virtex 2.5V

Odată cu introducerea familiei de circuite Virtex, în a doua jumătate a anului 1989. FPGA-urile depăşesc bariera de 1.000.000 de porți sistem pe circuit. Pe lângă densitatea mare de integrare, circuitele din familia Virtex 2.5 V prezintă următoarele caracteristici: frecvența sistemului până la 200 MHz, interfețe multistandard, circuite built-in de management ale tactului, sistem ierarhic de memorie, arhitectură flexibilă, sistem de dezvoltare compatibil cu cele oferite de Foundation şi Aliance, reprogramare nelimitată şi patru modalități de programare.

Arhitectura circuitelor Virtex are un caracter regulat şi cuprinde o matrice de blocuri logice reconfigurabile (CLB) înconjurată de blocurile de intrare/ieşire programabile (IOB) care sunt interconectate prin resurse de rutare rapide şi permit implementarea unor scheme complexe.

În figura 4.19 este prezentată schema bloc a circuitului Virtex şi pune în evidență cele două componente majore: CLB care furnizează elementele funcționale necesare construcției logicii şi IOB care furnizează interfața dintre pinii circuitului şi CLB-uri.

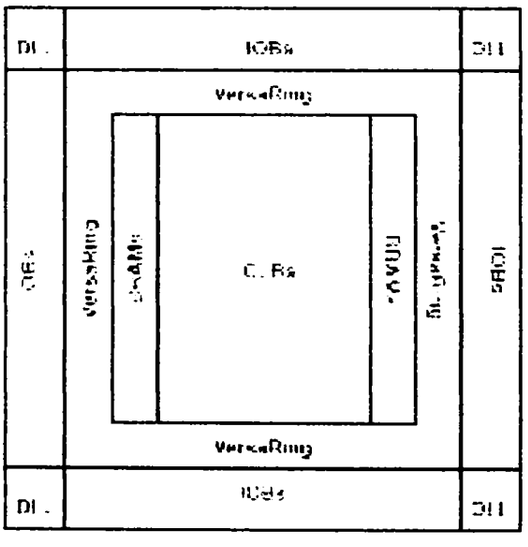


Figura 4.19. Arhitectura Virtex

CLB-urile sunt interconectate prin intermediul unei matrici generale de interconectare (GRM) care constă dintr-o matrice de conexiuni de rutare situate la intersecția canalelor de rutare verticale şi orizontale. Interfața I/O Versaring oferă resurse suplimentare de rutare la marginea circuitului mărind astfel rutabilitatea I/O. Circuitele Virtex conțin de asemenea următoarele circuite conectate la GRM: blocuri de memorie dedicate de 4096 biți fiecare, clock DLL pentru compensarea întârzierilor de distribuire a tactului şi buffer-e tristate (BUFT) asociate fiecărui CLB care comandă segmentele orizontale de rutare dedicate.

Valorile memorate în celule de memorie statică controlează elementele logice configurabile şi resursele de interconectare. Aceste valori se încarcă în celulele de memorie la punerea sub tensiune şi pot fi reîncărcate dacă este nevoie să se schimbe funcţia circuitului.

Structura IOB este prezentată în figura 4.20. Cele trei elemente de memorie funcţionează fie ca bistabile de tip D fie ca latch. Fiecare IOB are un semnal de tact (CLK) comun celor trei bistabile şi câte un semnal de clock enable pentru fiecare bistabil. Semnalul SR (Set/Reset) poate fi configurat ca Set sincron, Reset Sincron, Preset asincron sau Clear asincron. Buffer-ul de ieşire şi toate semnalele de control au control de polaritate independent. Pad-urile sunt protejate la depăşirile de tensiune. Toate IOB-urile permit o testare boundary scan compatibilă IEEE 1149.1.

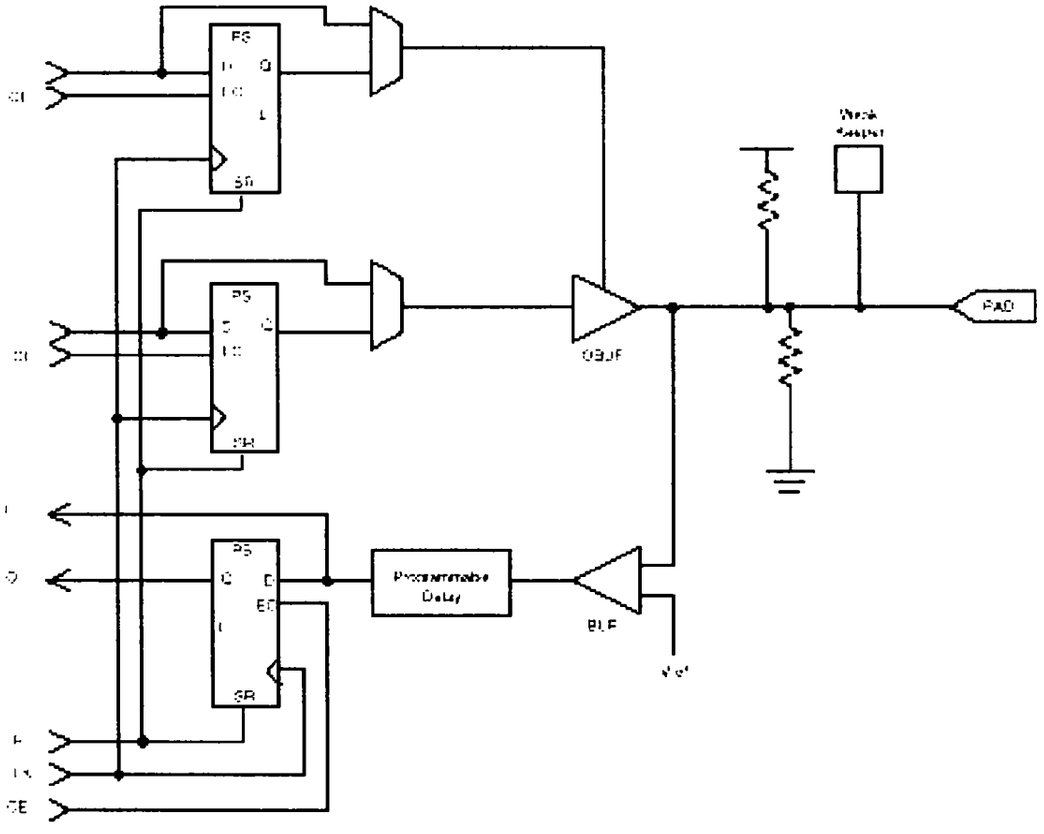


Figura 4.20. IOB Virtex

Blocul de bază al CLB-ului Virtex este celula logică (LC). Aceasta conţine un generator de funcţii cu 4 intrări, logica de transport şi un element de memorare. Ieşirea generatorului de funcţii comandă atât ieşirea CLB-ului cât şi intrarea D a bistabilului. Fiecare CLB Virtex conţine patru LC-uri distribuite în două blocuri identice. Structura unui astfel de bloc este prezentată în figura 4.21.

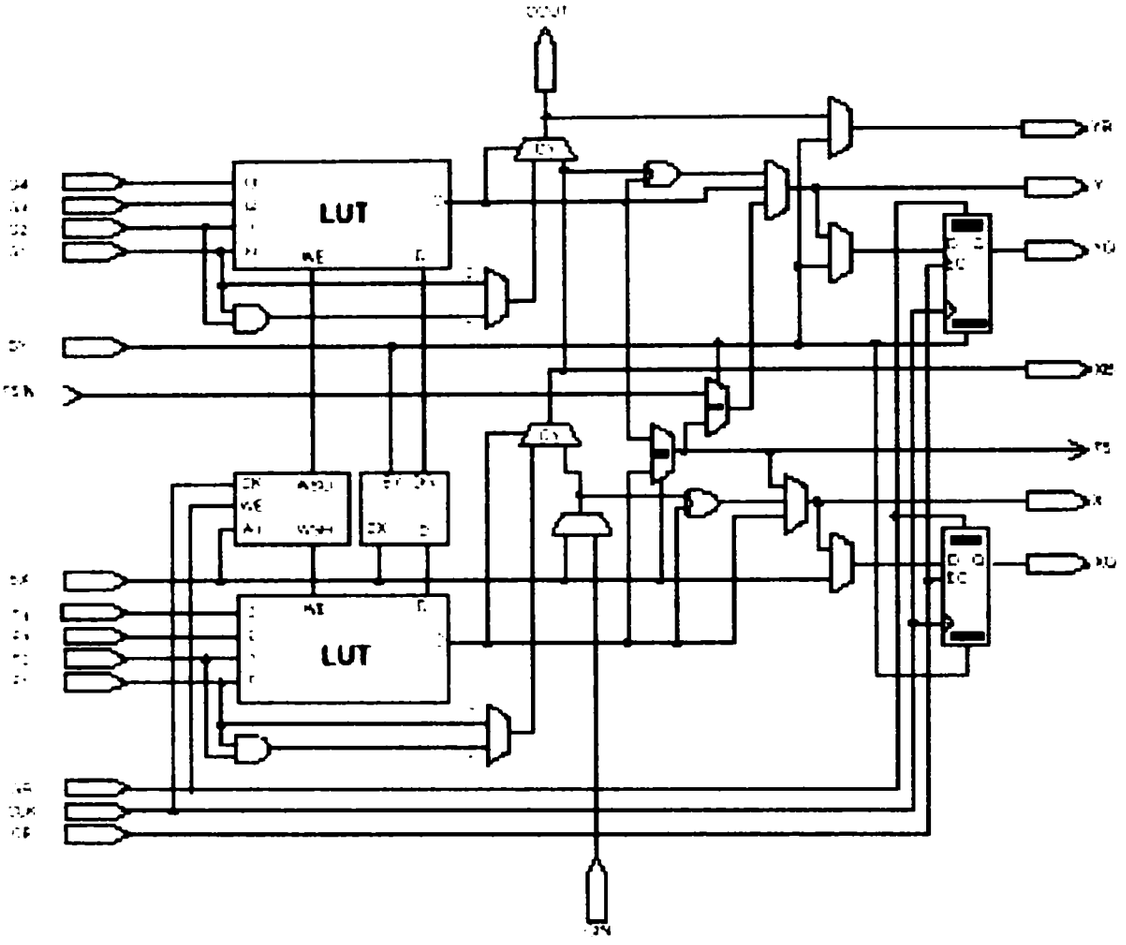


Figura 4.21. Structura parțială a unui CLB Virtex.

Generatoarele de funcții sunt implementate sub forma unor tabele look-up (LUT) cu patru intrări. Pe lângă funcția de generator de funcții fiecare LUT poate furniza un RAM sincron de 16x1 biți. Mai mult, cele două LUT pot fi combinate pentru a forma un RAM sincron de 16x2 sau 32x1 biți. LUT Virtex poate fi configurat şi ca un registru de deplasare de 16 biți. Intrările D ale elementelor de memorie pot fi comandate fie de generatoarele de funcții fie direct de intrări, scurtcircuitând generatoarele de funcții. Pe lângă semnalele de clock şi clock enable, fiecare bloc al CLB-ului are semnale sincrone de set şi reset (SR şi BY). SR aduc elementele de memorie în starea inițială specificată în configurație iar BY aduc aceste elemente în starea opusă. Multiplexorul F5 combină ieşirile generatorului de funcții dintr-un bloc al CLB-ului permitând implementarea oricărei funcții de 5 variabile sau a anumite funcții de până la 9 variabile, iar multiplexorul F6 combină ieşirile tuturor celor patru generatoare de funcții ale CLB-ului permitând implementarea oricărei funcții de 6 variabile sau a anumite funcții de până la 19 intrări. Logica aritmetică include o poartă XOR care permite implementarea unui sumator complet de 1 bit în fiecare LC iar poarta ŞI suplimentară măreşte eficiența implementării unui multiplicator. Logica de transport dedicată furnizează facilități suplimentare funcțiilor aritmetice high-speed.

4.4. Configurarea circuitelor FPGA

Configurarea reprezintă metoda prin care un proiect digital (tradus în fişier hartă de biţi) este implementat şi transmis circuitului FPGA. Metoda este asemănătoare cu programarea unui microprocesor. Pentru configurarea unui CLB şi a conexiunilor asociate se utilizează câteva sute de biţi. Fiecare bit de configurare defineşte starea unei celule de memorie statice care controlează un bit din tabela de stări, intrare multiplexor sau punct de tranziţie programabil [Yarb-97], [Bout-94]. Sistemul de dezvoltare traduce proiectul într-o listă de conexiuni de format Xilinx sau EDIF, partiţionează şi plasează proiectul logic şi genegrează datele de configuraţie în format PROM, sau hartă de biţi.

4.5. Descrierea procesului de proiectare

Procesul de proiectare cuprinde trei părţi: introducerea proiectului, implementarea proiectului şi verificarea acestuia (figura 4.22).

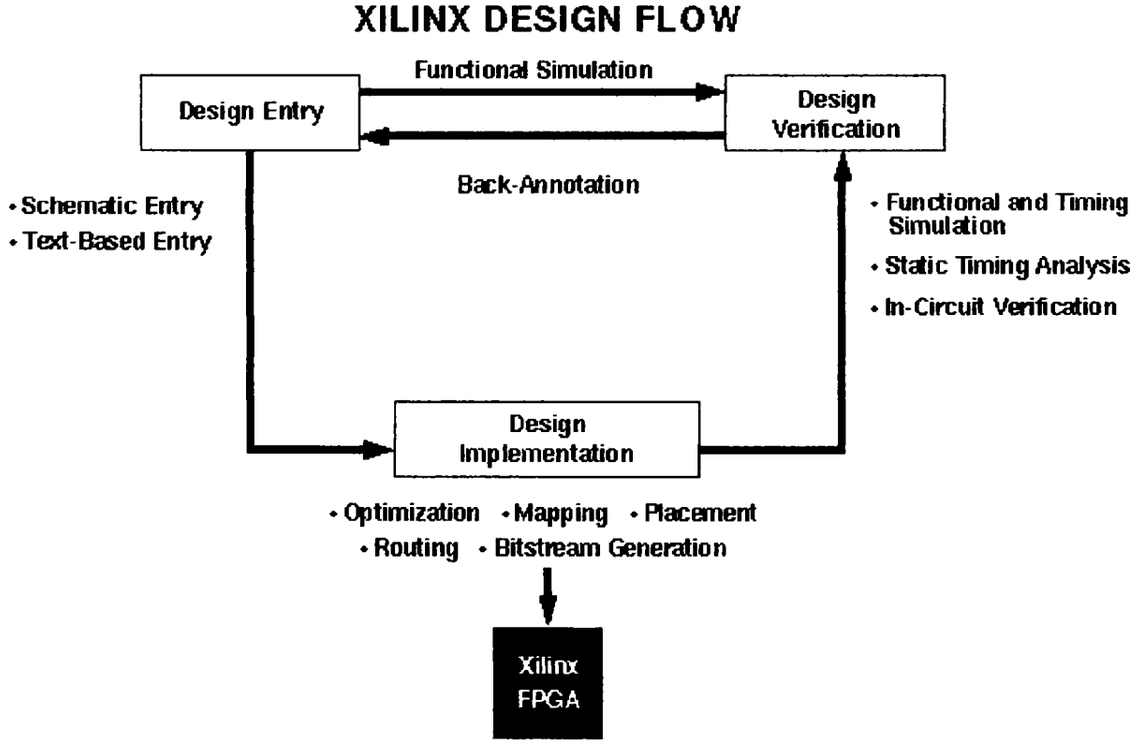


Figura 4.22. Procesul de proiectare Xilinx.

Termenii folosiţi în figura 4.22 au următoarea semnificaţie:

- Schematic entry – introducerea schemei utilizând simboluri grafice

- Text based entry – introducerea schemei pe baza unui limbaj de descriere hardware (HDL)
- Optimization – Conversia dispozitivului sau a descrierii logice într-o formă care poate fi implementată eficient într-un FPGA Xilinx.
- Mapping – reprezentarea logicii unei scheme ca resurse ale FPGA-ului Xilinx
- Placement – Asignarea blocurilor din schemă create în timpul mapării unor locații specifice din FPGA.
- Routing – Asignarea căilor de interconectare
- Bitstream generation – Conversia unei scheme într-un şir de biți care poate fi încărcat într-un FPGA Xilinx
- Back-annotation – Asocierea informațiilor de întârziere ale legăturilor implementate cu legăturile inițiale din schema introdusă
- Simulation – emularea software a logicii şi temporizării folosind stimuli de intrare.

Procesul de introducere a schemei presupune parcurgerea mai multor etape, de la conceperea schemei până la creerea unor fişiere de tip netlist interpretabile de programul utilizat. Există diverse modalități de introducere a circuitului care se proiectează: schemă, expresii booleene sau maşini de stare, limbaje de descriere hard (hardware description languages – HDL) ca de exemplu Verilog şi VHDL, fişiere netlist de tip EDIF sau XNF (Xilinx Netlist Format) ale unor circuite create anterior.

Implementarea circuitului începe cu conversia fişierului netlist în format NGD (Native Generic Database) şi se încheie cu creerea unui şir de biți ai configurației corespunzător dispozitivului FPGA dorit . Această etapă presupune parcurgerea unor etape de optimizare şi mapare, plasare şi rutare şi creerea şirului de biți [Xili-98i].

Etapa de verificare include simularea, analiza statică de timp şi verificarea "in-circuit" a schemei. Pentru simulare este necesară transformarea fişierului NGD (Native Generic Database) într-un fişier netlist de simulare. Un fişier NGD poate fi simulat în orice moment al procesului de proiectare [Xili-98g], [Xili-98f].

În figura 4.23 este prezentat pe larg întregul proces de proiectare a unui FPGA folosind soft-ul Xilinx M1, iar în continuarea acestui capitol este discutată fiecare etapă în parte.

4.5.1. Introducerea proiectului

Introducerea proiectului începe cu concepera circuitului exprimată fie sub forma unei scheme desenate fie sub forma unei descrieri funcționale. Pe baza circuitului este creat apoi un netlist care este sintetizat şi translatat într-un fişier NGO (Generic Object file). Acest fişier este apoi încărcat într-un program denumit NGDBuild care produce o bază de date generică logică (NGD). Procesul de introducere a schemei este ilustrat în schema bloc din figura 4.24.

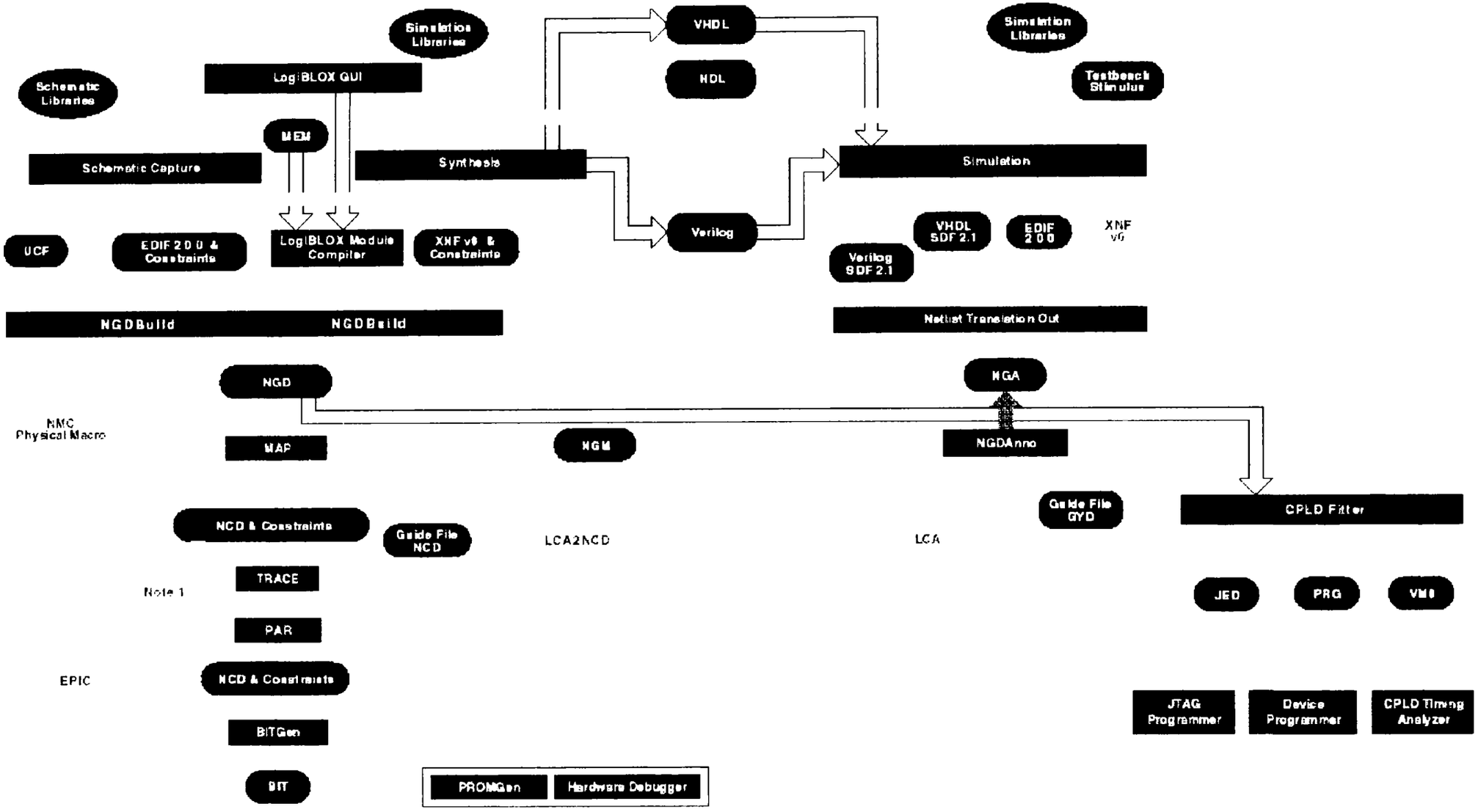


Figura 4.23. soft-ul de proiectare Xilinx M1

Atunci când introducerea circuitului proiectat se face pe baza unei scheme este apelată interfața grafică a programului. Aceasta permite conectarea simbolurilor care reprezintă componentele logice. Schema poate fi construită folosind porți individuale sau acestea pot fi combinate în vederea creerii unor blocuri funcționale. Librăria de componente conține atât primitive, adică porți ŞI şi SAU, cât şi macro-uri care reprezintă funcții de nivel înalt. Macro-urile conțin mai multe elemente din librărie, care pot include primitive sau alte macro-uri.

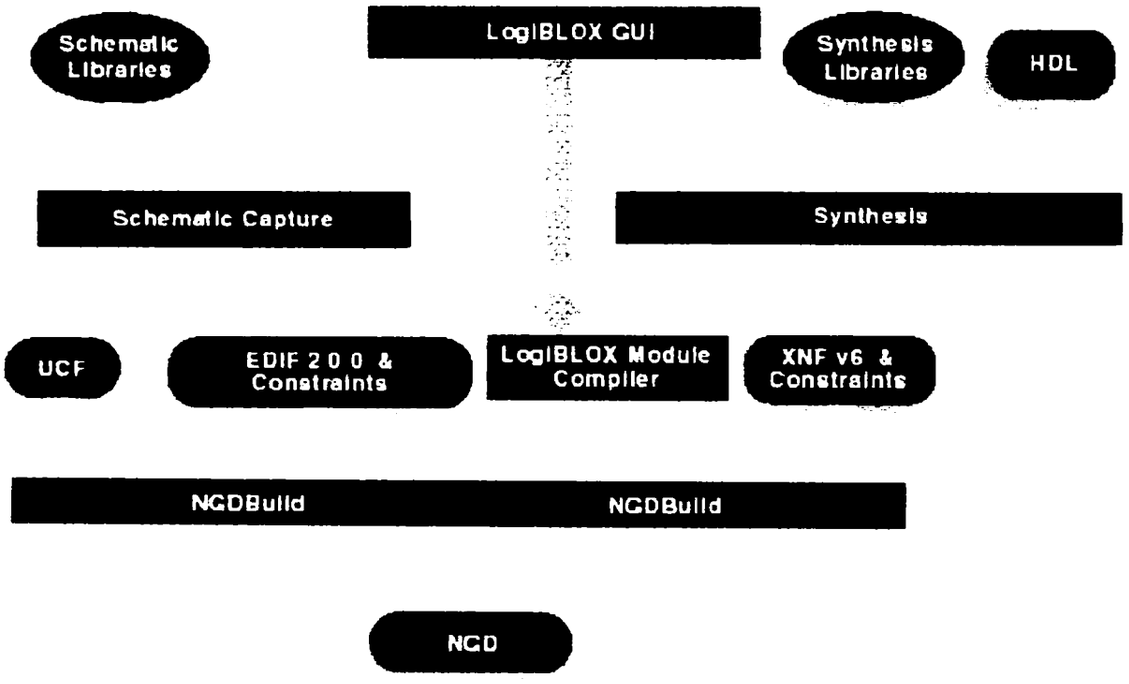


Figura 4.24. Procesul de introducere a proiectului

Există două tipuri de macro-uri ce pot fi folosite cu FPGA-urile Xilinx. Macro-urile soft, disponibile tuturor FPGA-urilor, au funcționalități predefinite, dar au mapări, plasări şi rutări flexibile. Macro-urile plasate relațional (Relationaly Placed Macros RPM-urile) au mapări fixe şi plasări relative. Macrourile nu pot fi folosite pentru sinteză, deoarece instrumentele de sinteză au propriile lor generatoare de module şi nu necesită RPM-uri.

LogiBLOX este un instrument care poate genera o serie de proiecte de nivel MSI şi LSI, de diverse mărimi, construind blocuri ca sumatoare, numărătoare, şi registre de deplasare [Xili-98i]. Aceste module completează librăriile de macro-uri,

care conțin funcții logice simple, de mărime fixă. LogiBLOX integrează de asemenea aceste module în proiecte.

O altă modalitate de introducere a proiectului este cea bazată pe text şi care se pretează la proiectele foarte mari [Xili-98g]. Chitul de proiectare XSI (Xilinx Synopsys Interface) permite implementarea proiectelor FPGA folosind FPGA Compiler, FPGA Express şi Design Compiler Synthesis software. Synopsys Synthesis Software crează şi optimizează schemele introduse în limbajele VHDL şi Verilog HDL.

Înainte de a implementa o schemă pot fi introduse restricții de timp sau ale parametrilor de plasare. La introducerea proiectului se pot face specificații privitoare la mapare, amplasarea blocurilor sau timing. Modul de mapare a unui anumit bloc în CLB, plasarea lui într-o anumită locație sau restricțiile de timp pe o anumită cale se pot specifica în schemă sau într-un fişier al restricțiilor (User Constaint File – UCF).

După introducerea proiectului, acesta poate fi fie simulat fie implementat. Simularea funcțională testează logica proiectului pentru a determina dacă acesta funcționează corect. Se poate economisi mult timp pe parcursul proiectării dacă sunt efectuate simulări funcționale în timpul procesului de proiectare.

Programele EDIF2NGD şi XNF2NGD permit citirea unor fişiere netlist de tip EDIF200, respectiv XNF. În figura Procesul de introducere a proiectului aceste programe sunt conținute în blocul "Netlist translatation". Programul NGDBuild apelează automat aceste programe, ele fiind necesare pentru conversia fişierelor EDIF sau XNF (Xilinx Netlist Format).

EDIF2NGD face conversia unui netlist EDIF într-un fişier NGO. Fişierul de ieşire NGO este o bază de date binară care descrie proiectul în termeni ai componentelor logice şi ierarhie specificate în fişierul de intrare. Dacă fişierul de intrare este de tip EDIF atunci Netlist Launcher apelează automat EDIF2NGD iar programul va face conversia fişierului NGO într-un fişier NGD care poate fi mapat. Fişierul NCD (Native Circuit Description) rezultat poate fi apoi plasat şi rutat. EDIF2NGD are ca intrări un fişier EDIF şi un fişier NCF (Netlist Constraints File). Programul citeşte restricțiile din acest fişier şi le atşează fişierului de ieşire NGO.

XNF2NGD permite conversia unui fişier XNF (Xilinx Netlist Format) într-un fişier NGO. Acesta este un fişier binar care conține o descriere logică a proiectului în termeni ai componentelor originale şi ierarhie. După ce are loc conversia, este rulat programul NGDBuild pentru a creea un fişier NGD care va include o descriere redusă la primitive Xilinx. XNF2NGD utilizează ca fişiere de intrare un fişier XNF – fişier text – şi un fişier NCF (Netlist Constraints File).

NGDBuild este un program care efectuează paşii necesari pentru citirea unui fişier netlist în format XNF sau EDIF şi creează un fişier NGD care descrie schema logică. Fişierul NGD rezultat după rularea programului NGDBuild conține atât descrierea logică a schemei, redusă la primitive Xilinx NGD cât şi o descriere în termeni ai ierahiei originale din fişierul netlist de intrare. Fişierul NGD de ieşire poate fi mapat pentru familia dorită de FPGA.

Pentru conversia unui fişier netlist în fişier NGD, programul NGDBuild parcurge următoarele etape:

- Citeşte fişierul netlist sursă apelând Netlist Launcher, care determină tipul fişierului netlist de intrare şi începe rularea programului de citire a acestuia. Netlist Launcher este parte a programului NGDBuild şi face orice translatare necesară execuţiei comenzii NGDBuild.
- Reduce toate componentele din schemă la primitive NGD
- Verifică schema prin rularea programului "Logical DRC" (Design Rule Check) care analizează schema convertită. Logical DRC constă dintr-o serie de teste efectuate asupra schemei logice
- Scrie la ieşire fişierul NGD

Fişierele de intrare a NGDBuild sunt:

- Fişiere design – care pot fi netlist XNF sau EDIF200. Dacă fişierul netlist de intrare este de un alt format decât cel recunoscut de NetlistLauncher, atunci acesta va apela programul necesar pentru conversia acestuia – EDIF2NGD sau XNF2NGD.
- Fişierul UCF (User Constraints File). Acesta este un fişier ASCII creat de utilizator şi care conţine restricţiile referitoare la timp şi aranjare, restricţii care afectează modul de implementare în echipamentul dorit. Restricţiile din fişier sunt adăugate informaţiei din fişierul de ieşire NGD.
  Implicit NGDBuild citeşte restricţiile din fişierul UCF dacă acesta are acelaşi nume de bază ca şi fişierul design de intrare şi extensia .ucf.
- Fişierul NGC – este un fişier binar care conţine implementarea unui modul din schemă. Dacă fişierul NGC există pentru un modul, NGDBuild citeşte acest fişier direct fără a căuta o sursă netlist de tip EDIF sau XNF. În proiectarea HDL, LogiBLOX crează un fişier NGC pentru a defini fiecare modul
- Fişiere NMC – Physical Macros. Aceste fişiere binare conţin implementarea unui macro fizic din schemă. NGDBuild citeşte fişierul NMC pentru a crea un model de simulare al comportamentului pentru macro.
- Fişiere NEM – LogiBLOX Memory Definition Files. Aceste fişiere text definesc conţinutul modulelor de memorie LogiBLOX. NGDBuild citeşte fişierele NEM din procesul de proiectare acolo unde LogiBLOX nu crează direct fişiere NGC.

Fişierele de ieşire a programului NGDBuild sunt:

- Fisierul NGD – fişier binar care conţine o descriere logică a proiectului în termeni ai comportamentului original şi ai ierarhiei precum şi primitivele NGD la care se reduce schema.
- Fişier BLD – Build Report – conţine informaţii despre rularea NGDBuild. Fişierul are acelaşi nume ca şi fişierul de ieşire NGD, dar are extensia .bld. Fişierul este scris în acelaşi director ca şi fişierul NGD.

După ce este creat fişierul netlist, elementele proiectului sunt adunate într-o singură bază de date pentru ca operaţiile MAP şi PAR să poată fi aplicate deodată întregului proiect.

Funcția finală a programului constă în memorarea proiectului într-un format NGD (Native Generic Database) care conține primitivele folosite pentru a modela toate CPLD-urile şi FPGA-urile. Aceste primitive sunt similare cu primitivele din simularea temporală, numite SIMPRIMS. Fişierul NGD de ieşire conține de asemenea o descriere logică exprimată în termeni ai netlist original. Fişierul NGD poate fi mapat în familia dorită de echipamente [Xili-98e]. În figura 4.25 sunt date etapele care sunt parcurse în programul NGDBuild.

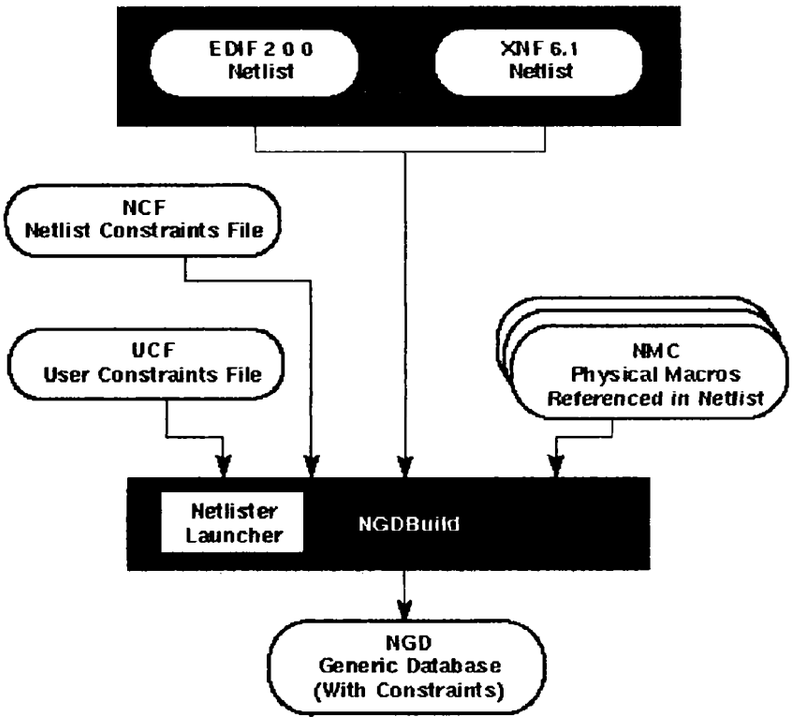


Figura 4.25. Etapele programului NGDBuild.

4.5.2. Restricțiile de timp

Soft-ul Xilinx permite specificarea unor restricții precise de timp pentru circuitele FPGA. Se pot specifica restricții pentru orice semnal sau cale din schemă. O modalitate de specificare este de a identifica mai întâi un set de căi prin identificarea unui grup de puncte de început şi de sfârşit. Aceste puncte de început şi de sfârşit pot fi bistabile, pini I/O, latch-uri sau RAM-uri. Apoi se pot controla timpii din situațiile cele mai defavorabile din setarea căilor prin specificarea unei singure restricții de întârziere pentru toate căile din setare [Xili-98h].

Prima metodă de specificare a restricțiilor de timp este de introducere a acestora în schemă. În orice caz se pot specifica restricțiile de timp în fişierele de restricții UCF şi PCF.

După definirea specificațiilor de timp şi maparea schemei, PAR va plasa şi ruta schema pe baza acestor cerințe.

Pentru a analiza rezultatele specificațiilor de timp se utilizează TRACE (Timing Report and Circuit Evaluator).

Definirea atributelor TS (timing specifications) se face în "TIMESPEC primitive", care serveşte ca loc de păstrare a specificaților de timp. Fiecare atribut TS începe cu literele TS şi se termină cu un identificator unic care poate fi format din litere, cifre şi caracterul _. Lungimea atributului este nelimitată, dar sunt afişate doar 30 de caractere. Fiecare primitivă TIMESPEC poate conține până la 8 atribute TS (figura 4.26). Dacă sunt necesare mai multe atribute, atunci se pot folosi primitive TIMESPEC multiple.

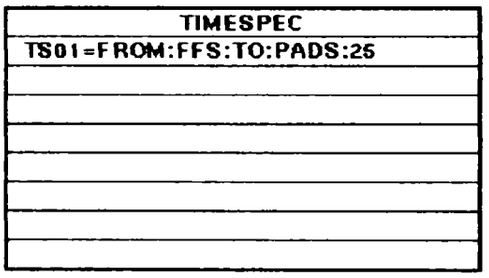


Figura 4.26. Primitive TIMESPEC.

Un atribut TS are următoarea sintaxă:

TS identifier = FROM source_group TO dest_group delay

unde source_group şi dest_group sunt punctele de început şi de sfârşit iar delay defineşte întârzierea maximă pentru calea dintre punctul de început şi de sfârşit. Unitatea de timp implicită este ns. Se pot folosi şi alte unități de timp pentru specificarea întârzierii, de exemplu ps sau MHz.

Specificațile de timp se pot introduce ca restricții într-un fişier UCF. La rularea programului NGDBuild, aceste specificații sunt adăugate bazei de date a schemei ca parte a fişierului NGD.

Sintaxa de bază pentru specificațile de timp introduse în fişierul de restricții este sintaxa atributului TS [Xili-98h]. Într-un atribut TS pot fi specificate seturi de căi care vor fi analizate prin gruparea punctelor de început şi de sfârşit într-unul din următoarele moduri: referire la un grup predefinit prin specificarea unuia din cuvintele cheie – FFS, PADS, LATCHES sau RAMS (Figura 4.27) -, creerea unui grup propriu în cadrul unui grup predefinit prin marcarea simbolurilor cu atributul TNM (Figura 4.28), creerea grupurilor care sunt combinații ale grupurilor existente utilizând simbolul TIMEGRP sau creerea grupurilor prin suprapunere cu numele semnalelor.

```
TS01=FROM FFS TO FFS 30
```

```
TS02=FROM LATCHES TO LATCHES 25
TS03=FROM PADS TO RAMS 70
TS04=FROM FFS TO PADS 55
```

Figura 4.27. Utilizarea grupurilor predefinite

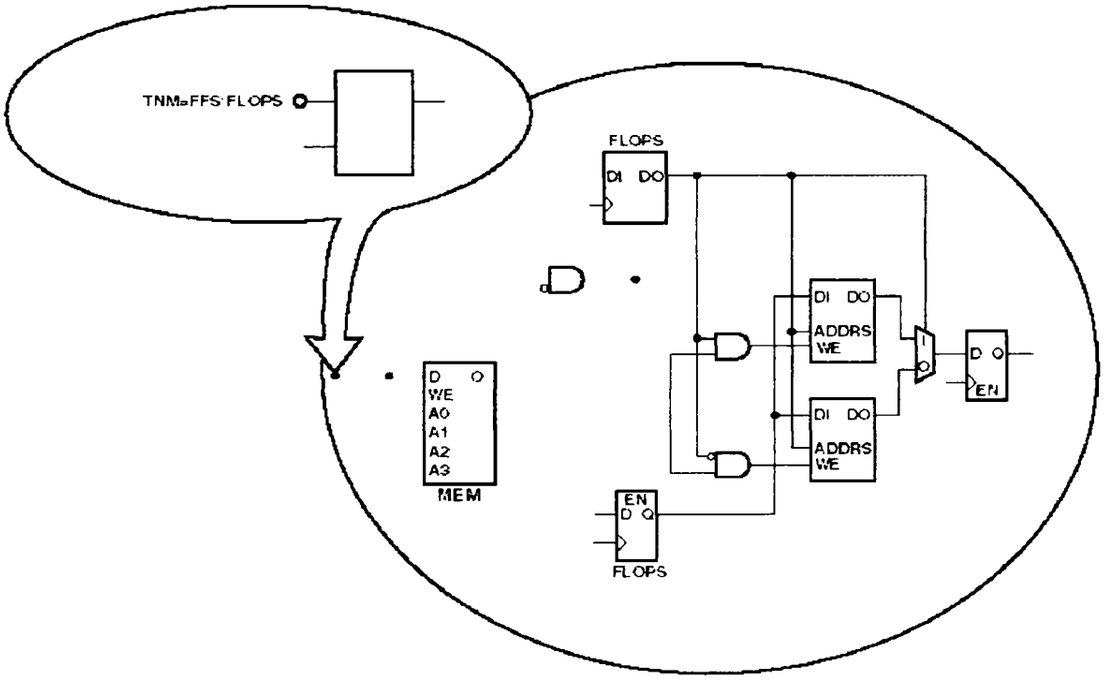


Figura 4.28. Atributul TNM plasat pe pinul unui macro

Specificarea perioadei de tact este ataşată căii de tact. Definirea unei perioade de tact este diferită de tipul FROM-TO deoarece instrumentele de analiză de timp iau în considerare orice inversare a semnalului de tact la pinii de tact ai registrului.

O metodă simplă de definire a unei perioade de tact este de a ataşa următorul atribut direct unui semnal din calea care conduce la pinii de tact ai registrului.

Sintaxa folosită pentru specificarea perioadei de tact într-o schemă este:

PERIOD = period{HIGH | LOW} [high_or_low_time]

sau în cazul fişierelor UCF:

[period_item] PERIOD = period{HIGH | LOW} [high_or_low_time]

unde period_item poate fi:

- NET “net_name”
- TIMEGRP “group_name”
- ALLCLCKNETS (PCF only)

Period - este perioada necesară a tactului. Unitatea implicită este ns dar poate fi şi ps, μs mau ms. Cuvântul cheie HIGH | LOW indică dacă primul puls din perioadă este high sau low. High_or_low_time este opțional. Dacă este specificat un timp actual, acesta trebuie să fie mai mic decât perioada, iar în caz contrar el reprezintă 50%. Unitatea implicită este ns.

Specificațile temporale OFFSET sunt folosite pentru a defini relațiile de timp dintre un tact extern şi pinul asociat, data-in sau data-out. Utilizând această opțiune se pot face următoarele operații:

- să se calculeze dacă timpul de setup a fost respectat la un bistabil a cărui intrări de date derivă din semnale externe
- să se specifice întârzierea unui semnal de ieşire extern derivat din ieşirea Q a unui bistabil intern sincronizat de la un pin exterior al echipamentului.

Utilizarea restricțiilor OFFSET oferă următoarele avantaje:

- scade întârzierea de pe calea de tact din întârzierea căii de date pentru intrări şi adună întârzierea căii de tact la întârzierea căii de date pentru ieşiri.
- Include căile pentru toate tipurile de elemente sincrone (FFS, RAM şi LATCH)
- Utilizează o sintaxă globală care permite ca toate intrările şi ieşirile să fie restricționate printr-un tact
- Permite specificarea restricților IO fie direct ca setup şi clock-to-out necesare unui echipament, fie indirect ca timp folosit de calea externă echipamentului.

În cadrul primitivei TIMESPEC se poate utiliza următoarea sintaxă pentru a specifica cerințele de timp dintre punctele de sfârşit specifice:

```
TS identifier = FROM source_group TO dest_group delay
TS identifier = FROM source_group delay
TS identifier = TO dest_group delay
```

Declarațile de tip FROM-TO sunt atribuite TS care rezidă în primitive TIMESPEC. Parametri source_group şi dest_group trebuie să fie unul din:

- Grupuri predefinite
- Identificatori TNM creați anterior
- Grupuri definite în simboluri TIMEGRP
- Grupuri TPSYNC

Parametrul delay defineşte întârzierea maximă a atributului. Unitatea de timp implicită pentru specificarea întârzierilor în atributele TS este ns. Mai jos sunt date câteva exemple de utilizare ale declarației FROM-TO [Xili-98h].

Sintaxa folosită pentru primitiva TIMESPEC într-o schemă este:

```
TS01=FROM FFS TO FFS 30
TS_OTHER=FROM PADS TO FFS 25
```

```
TS_THIS=FROM FFS TO RAMS 35
TS_THAT=FROMPADS TO LATCHES 35
```

Sintaxa UCF este:

```
TIMESPEC TS01=FROM FFS TO FFS 30
TIMESPEC TS_OTHER=FROM PADS TO FFS 25
TIMESPEC TS_THIS=FROM FFS TO RAMS 35
TIMESPEC TS_THAT=FROMPADS TO LATCHES 35
```

Un exemplu de utilizare a atributului TNM ataşat unui simbol individual este dat in figura 4.29. In această schemă bistabilul D_FF are ataşat atributul TNM= D_FF.

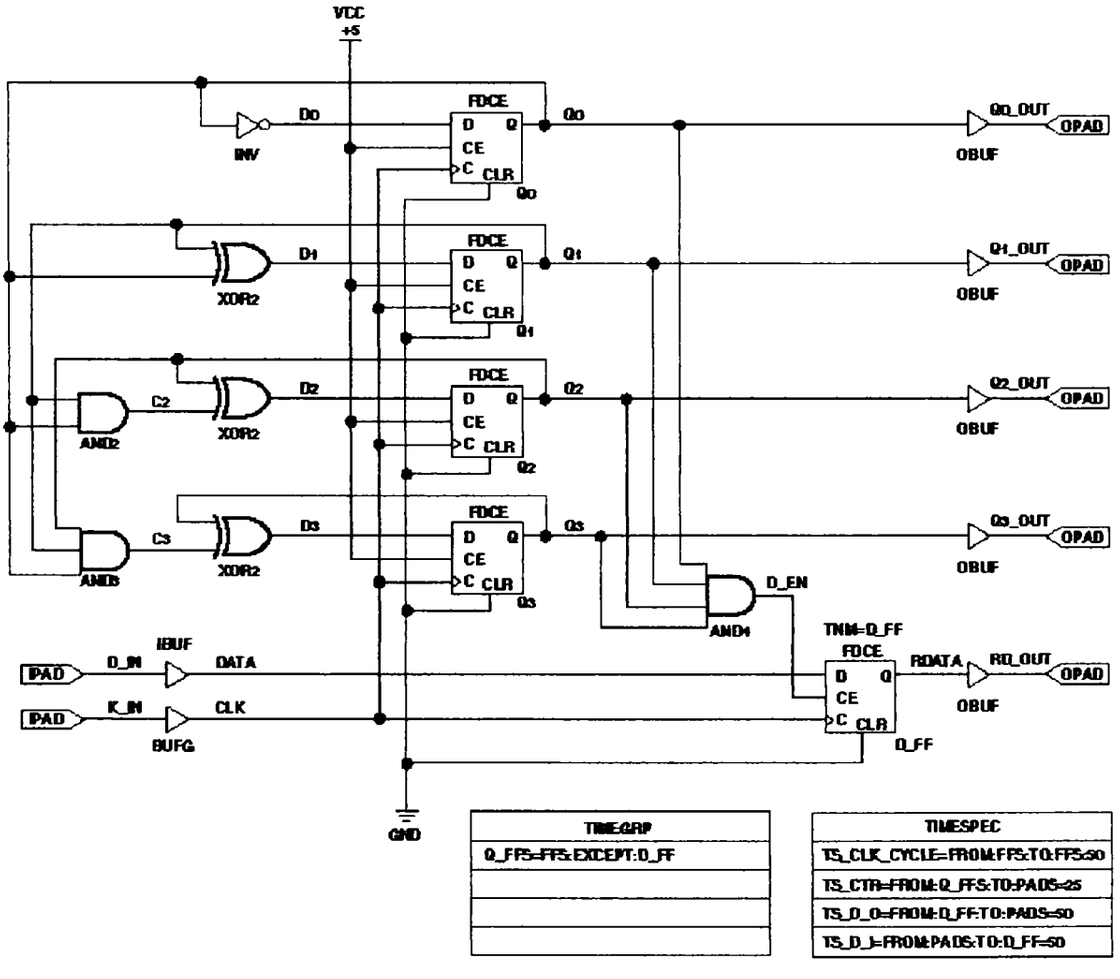


Figura 4.29. Exemplu de utilizare a TNM-urilor şi TIMEGRP-urilor în schemă

Simbolul TIMEGRP conține un atribut care defineşte un grup de bistabile numit Q_FFS, care include toate bistabilele din schemă cu excepția celui numit D_FF. Grupul Q_FFS poate apoi utilizat pentru pentru a creea specificații temporale în

primitiva TIMESPEC. Bistabilul D_FF are clock enable fixat la ½ din frcvența de tact, motiv pentru care specificațiile temporale de la bistabil la pad şi de la pad la bistabil sunt mai lungi decât specificațile de la bistabil la pad din grupul Q_FFS.

4.5.3. Implementarea proiectului

Implementarea proiectului începe cu procesul de mapare şi este completă când proiectul fizic a fost rutat şi s-a generat un "bitstream" (figura 4.30).

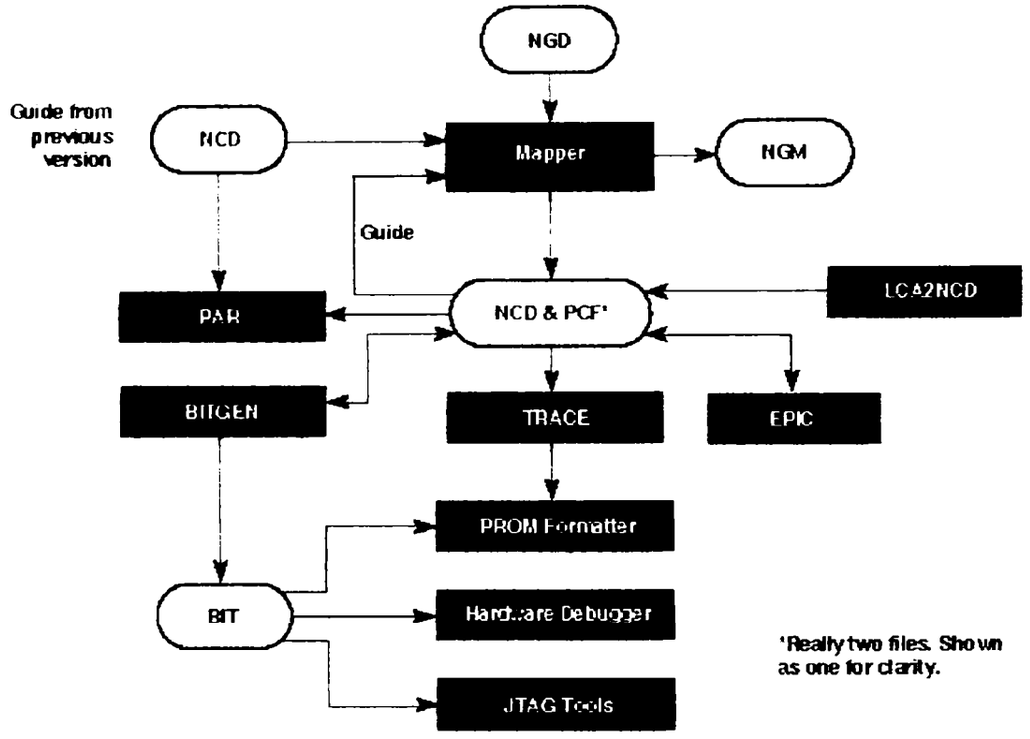


Figura 4.30. Implementarea proiectului

Maparea reprezintă procesul de alocare a CLB-urilor, IOB-urilor sau a altor resurse Xilinx elementelor logice dintr-un proiect. Acest proces este reprezentat în figura 4.31.

Există următoarele tipuri de fişiere de intrare în programul MAP:

- Fişier NGD – Native Generic Database file. Acest fişier conține o descriere logică a proiectului atât în termeni ai componentelor ierarhice utilizate pentru realizarea schemei cât şi a primitivelor Xilinx din nivelurile inferioare. Fişierul conține deasemenea şi informații referitoare la restricțile aplicate proiectului în timpul procesului de introducere sau introduse în UCF (User Constraints File).

- Fişiere NMC – Macro Phisycal Libraries files - fiecare din ele conținând definirea unui macro fizic. Macrourile fizice preplasate sunt integrate in proiect folosind o reprezentare .nmc a macroului. Fişierul librărie macro, .nmc, este creat folosind editorul EPIC (Editor for Programmable Integrated Circuits) de proiect fizic.
- MFP – Map Floorplanner File - generat de Floorplanner este utilizat ca un fişier ghid pentru mapare. La rularea programului MAP pentru a genera un fişier NCD, se poate deschide fişierul mapat NCD din Floorplanner, se poate modifica amplasamentul componentelor şi apoi se generează un fişier MPF. Acesta poate fi folosit ca o intrare.
- Fişierul ghid NCD – este un fişier de intrare opțional generat la o rulare anterioară a programului de mapare. Acest fişier conține o descriere a schemei în termeni ai componentelor din dispozitivul FPGA utilizat. Fişierul ghid NCD este practic un fişier de ieşire al unei mapări anterioare şi care acum este intrare pentru o nouă mapare.

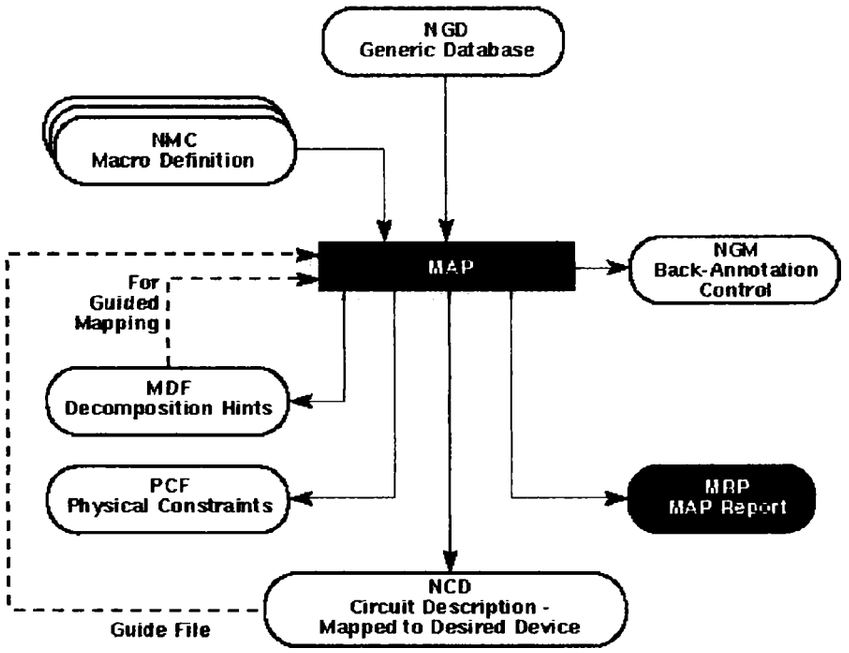


Figura 4.31. Programul MAP

Ca fişiere de ieşire MAP generează:

- Fişier NCD – Native circuit Description - o descriere fizică a schemei în termeni ai componentelor din echipamentul FPGA.
- Fişier PCF (Phisical Constraints) – un fişier text ASCII care conține restricțiile specificate pe parcursul introducerii schemei şi exprimate în

termeni ai elementelor fizice. Fişierul este creat de MAP în cazul în care acesta nu există sau este rescris un fişier deja existent.

- Fişier NGM – un fişier binar al proiectului care conţine toate datele de intrare din fişierul NGD precum şi informaţiile referitoare la schema fizică construită la mapare. Acest fişier este utilizat pentru pentru a corela schema "back-annotated" cu structura şi denumirea schemei sursă.
- MRP (MAP report) – un fişier care conţine informaţii referitoare la rularea MAP. Sunt listate toate eroriile şi atenţionările din schemă, atributele specificate, detaliile referitoare la cum s-a făcut maparea. Fişierul va da de asemenea informaţii statistice referitoare la uzura componentelor din schema mapată.
- MDF (MAP Directive File) – un fişier care descrie modul cum a fost descompusă logica la maparea schemei.

Fişierele MRP, MDF şi NMG au toate acelaşi nume ca şi fişierul de ieşire, dar cu extensile menţionate. Dacă aceste fişiere există deja, atunci ele vor fi suprascrise de noile fişiere.

MAP realizează mai întâi o verificare DRC (Design Rule Check) a proiectului din fişierul NGD. Apoi se realizează o mapare a logicii în componentele echipamentului FPGA dorit. Ieşirea este un fişier NCD (Native Circuit Description) – o reprezentare fizică a proiectului mapat in componentele din FPGA. Fişierul NCD poate fi apoi plasat şi rutat [Xili-98i].

Programul MAP se poate apela din Design Manager/Flow Engine.

Atunci când se face maparea unui proiect care conţine registre, softul MAP permite optimizarea modului de grupare al registrelor în CLB-uri. Această mapare optimizată se numeşte "register ordering" [Vasa-98]. Un CLB conţine două bistabile. MAP realizează împerecherea biţilor registrelor dacă recunoaşte că o serie de bistabile formează un registru. Pentru aceasta la creerea schemei biţii trebuiesc denumiţi.

LCA2NCD. Fişierele LCA sunt fişiere ASCII. Instrumentele de plasare şi rutare ale sistemului de dezvoltare Xilinx M1 operează cu fişiere de format NCD (Native Circuit Description), care sunt fişiere binare. După conversie fişierul NCD poate fi editat in EPIC (Editor for Programmable Integrated Circuits), procesat pentru simulare, rerutat de PAR sau utilizat ca şi fişier ghid pentru MAP, PAR sau ambele. LCA2NCD face conversia unui fişier LCA într-un fişier NCD (figura 4.32).

Pe lângă fişierul NCD la ieşire LCA2NCD crează un fişier MDF (MAP Directive File) care descrie modul în care a fost descompusă logica pentru ca apoi aceste informaţii să fie folosite pentru o mapare ghidată, şi un fişier L2N care este un raport ce conţine informaţii despre rularea LCA2NCD.

PCF (Physical Constraints File). Fişierul NGD creat la citirea unui netlist poate conţine anumite restricţii logice care sunt apoi citite de MAP [Xili-98i]. MAP utilizează unele restricţii pentru maparea schemei şi converteşte alte restricţii logice în restricţii fizice. Aceste restricţii fizice sunt apoi scrise de către MAP într-un fişier PCF.

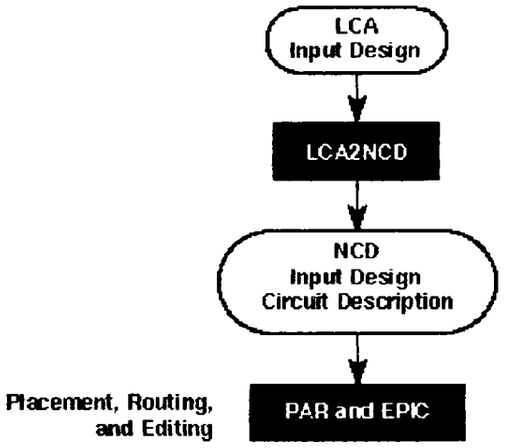


Figura 4.32. Conversia cu LCA2NCD

Fişierul PCF este un fişier ASCII care conține două părți diferite: una pentru restricțiile fizice create de "mapper" şi una pentru restricțiile fizice introduse de utilizator. Prima parte este scrisă de fiecare dată când se rulează programul de mapare. Ea este urmată apoi de restricțiile fizice ale utilizatorului. În cazul apariției unor conflicte, restricțiile create de utilizator sunt ultimele citite şi suprascrise. Fişierul poate conține oricâte restricții sau comentarii [Xili-98c].

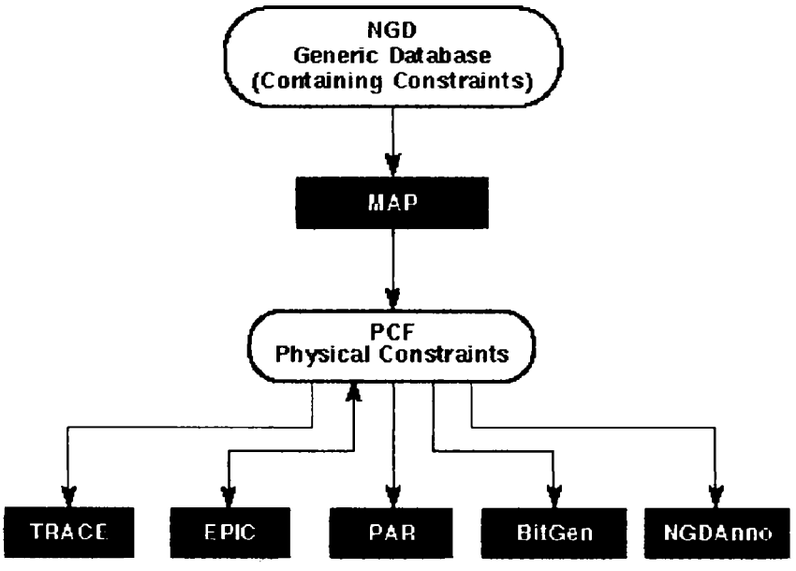


Figura 4.33. Creerea fişierului PCF.

Fişierul PCF este intrare opţională în PAR, EPIC, TRACE, NGDAnno şi BitGen (Figura 4.33).

DRC (Physical Design Rule Checker) constă dintr-o serie de teste care au ca scop depistarea eroriilor fizice din schemă. Trei module ale sistemului de dezvoltare Xilinx solicită DRC fizic. Acesta este utilizat în următoarele moduri.

- MAP rulează în mod automat DRC fizic după ce schema este mapată.
- PAR (Place and Route) rulează automat DRC fizic asupra semnalelor atunci când schema este rutată.
- DRC fizic poate fi rulat din EPIC (design editor). DRC se execută automat şi după anumite operaţii.
- DRC este rulat automat şi de către BitGen, care crează un fişier BIT necesar programării echipamentului.
- DRC poate fi rulat din linia de comandă UNIX sau DOS.

Intrare în DRC este fişierul NCD (Native Circuit Description), iar ieşirea este un fişier de tip TRD. Fişierul TRD este un fişier ASCII şi reprezintă un raport al cărui conţinut este determinat de opţiunile selectate în comanda DRC [Xili-98i].

DRC poate executa patru tipuri de verificări:

- Verificarea semnalelor – examinează una sau mai multe semnale rutate sau nerutate şi raportează orice problemă a numerotării pinilor, a inconsistenţei bufferelor tristate, a semnalelor cu poziţie mobilă.
- Verificarea blocurilor – examinează unul sau mai multe componente plasate sau neplasate şi raportează orice problemă de logică, conexiune fizică a pinilor sau de programare.
- Verificarea chipului – examinează o clasă specială de verificări pentru semnale, componente sau ambele la nivel de chip, ca de exemplu reguli de amplasare referitoare la o faţă a echipamantului.
- Toate verificările – realizează verificări de semnale, blocuri şi chip.

PAR (Place and Route Program). După maparea proiectului pentru o anumită arhitectură, acesta poate fi plasat şi rutat cu ajutorul programului PAR. PAR citeşte un fişier NCD (Native Circuit Description) şi automat efectuează o plasare optimă şi o rutare pe CLB-urile şi IOB-urile mapate. Maparea şi rutarea se poate face ţinând cont de restricţiile temporale specificate în procesul de proiectare. După terminarea programului se pot verifica caracteristicile de timp prin rularea utilitarului TRACE (Timing Reporter and Circuit Evaluator) [Xili-98b] care oferă rapoarte detaliate şi indică atenţionările şi erorile apărute. Intrările în PAR sunt următoarele fişiere:

- Fişier NCD – un proiect mapat
- Fişier PCF (Physical Constraints File) – Un fişier ASCII care conţine restricţiile bazate pe atributele din schemă sau plasate în fişierul UCF.
- Fişier Guide NCD – un fişier opţional dintr-un MAP sau PAR rulat anterior care poate fi folosit drept ghid pentru plasare şi rutare în PAR curent.

Ieşirile din PAR sunt următoarele fişiere:

- Fişier NCD – un fişier binar al proiectului mapat şi rutat care poate fi citit de instrumentele de implementare Xilinx. Fişierul NCD conţine informaţii cu

privire la plasare şi rutare în diferite faze. Acest fişier este folosit apoi de programul BitGen. El poate fi şi un fişier ghid când se reface plasarea şi rutarea unei scheme care a suferit mici modoficări față de ultima iterație.

- Fişier PAR – un raport PAR care include informații sumare ale tuturor iteraţiilor de plasare şi rutare.
- Fişier DLY – conține informații de întârziere pentru fiecare cale din proiect.
- Fişier UCF – User Constraints file, care poate fi modificat cu restricții "pad location" (LOC)
- Fişier PAD – un fişier care conține asignări ale pinilor I/O

Diagrama care reprezintă procesul PAR este dată în figura 4.34.

PAR din Xilinx Development System [Xili-98b] face plasarea şi rutarea unei scheme folosind o combinație a următoarelor două metode:

- Bazată pe cost – acest lucru înseamnă că plasarea şi rutarea schemei sunt făcute pe baza unor tabele care atribuie valori ponderate factorilor relevanți, ca restricții, lungimi de conexiuni şi resurse de rutare disponibile.
- Timing-driven – softul de analiză de timp validează PAR pentru plasarea şi rutarea unei scheme pe baza restricțiilor de timp.

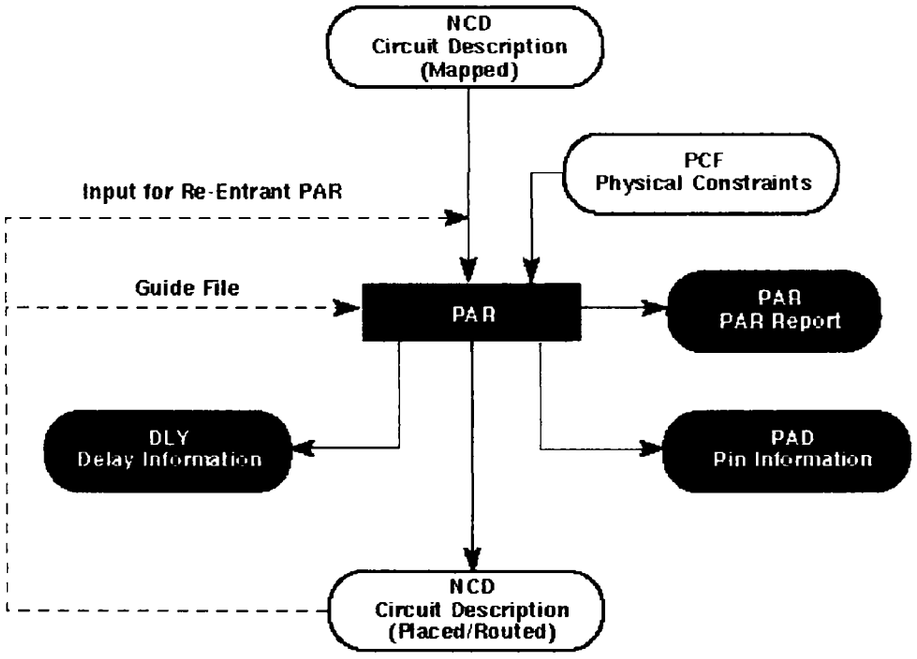


Figura 4.34. Procesul PAR

Pentru a utiliza timing-driven PAR, restricțiile de timp pot fi specificate astfel:

- Ca proprietăți în schemă sau programul HDL.

- Pot fi scrise într-un fişier UCF care este procesat de NGDBuild la generarea bazei de date a proiectului logic.
- Pot fi scrise într-un fişier PCF generat de MAP. Acesta conţine orice restricţie de timp specificată la utilizarea celor două metode prezentate şi orice restricţie de timp introdusă direct în fişier.

Fişierele de intrare sunt:

- Fişierul NCD – o schemă mapată
- Fişier PCF – un fişier ASCII care conţine restricţii bazate pe atribute din schemă sau pe atribute plasate într-un fişier UCF.
- Fişier ghid NCD – un fişier template opţional pentru plasare şi rutare

Fişierele de ieşire sunt următoarele:

- Fişier NCD – un fişier care conţine informaţii de plasare şi rutare de diverse complexităţi
- Fişier PAR – un raport PAR care include informaţii sumare ale tuturor iteraţilor de plasare şi rutare
- Fişier DLY – un fişier care conţine informaţii pentru fiecare semnal din schemă
- Fişier PAD – un fişier care conţine asignarea pinilor I/O

Plasarea se face în două etape: o amplasare constructivă şi una de optimizare. Fişierul NCD este scris după amplasarea constructivă, urmând ca să fie modificat după amplasarea optimizată. La amplasarea constructivă PAR plasează componentele în cuvinte pe baza unor factori ca restricţii specificate în fişierele de intrare (de exemplu anumite componente trebuiesc să fie în anumite locaţii), lungimea conexiunilor şi resursele disponibile de rutare. Această amplasare va ţine cont şi de tabelele de cost care asignează valori factorilor relevanţi. Există 100 de tabele de cost posibile [Bout-94]. Amplasarea optimizată reprezintă o îmbunătăţire a rezultatelor din amplasarea costructivă. Optimizarea are loc doar la anumite niveluri. Par rescrie fişierul NCD după amplasarea optimizată.

Rutarea se face în două etape: rutarea constructivă şi finală. PAR scrie fişierul NCD după fiecare 60 de minute de rutare şi va srie un nou fişier NCD doar dacă se îmbunătăţeşte calitatea schemei.

În PAR poate fi încărcat şi un fişier opţional de ghidare. Acesta este un fişier NCD utilizat ca un template pentru plasarea şi rutarea schemei de intrare. El este util dacă s-au facut schimbări incrementale minore cu scopul creerii unui nou proiect. Pentru mărirea productivităţii se poate utiliza ultima iteraţie ca un ghid pentru următoarea iteraţie (figura 4.35).

Procesul de proiectare ghidată se va desfăşura în felul următor:

- Dacă o componentă din noul proiect are acelaşi nume ca şi componenta din ghid, ea va fi plasată în locul în care se găsea in proiectul ghid.
- Dacă o componentă nenumită din noul proiect este de acelaşi tip ca şi o componentă nenumită din ghid, şi cele două componente au ataşate semnale identice, atunci componenta este plasată acolo unde ea a fost plasată şi în proiectul ghid.

- Dacă semnalele ataşate unei componente în noul proiect corespund cu semnalele ataşate componentei în proiectul ghid, pinii sunt interschimbați pentru a corespunde proiectului ghid, dacă este posibil.
- Dacă numele semnalelor din proiectul de intrare sunt ca cele din ghid şi au aceaşi sursă, informațiile de rutare din proiectul ghid sunt copiate în noul proiect.

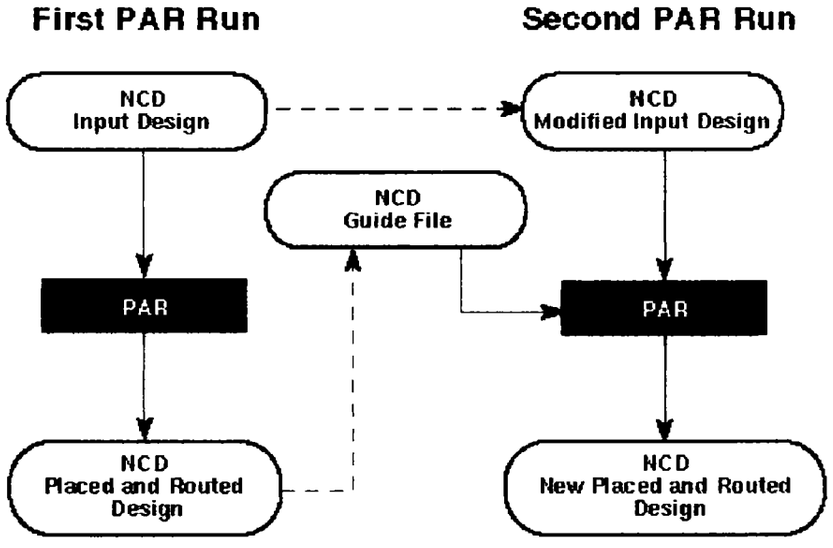


Figura 4.35. PAR ghidat

La rularea PAR folosind proiectul ghid ca intrare se vor plasa şi ruta mai întâi componentele şi semnalele care îndeplinesc criteriile de suprapunere menționate şi apoi plasează şi rutează logica rămasă [Xili-98b].

PIN2UCF este un program care generează restricțiile de blocare a pinilor într-un fişier UCF prin citirea unui fişier NCD (Native Circuit Description) de plasare în cazul FPGA-urilor, sau un fişier GYD în cazul CPLD-uri. Ieşirile sunt scrise într-un fişier UCF existent dacă există sau crează unul nou în caz contrar (figura 4.36).

PIN2UCF realizează următoarele operații:

- Programul extrage locațiile pinilor şi numele pad-urilor locale dintr-un fişier NCD sau GYD şi îşi scrie informațiile într-un fişier UCF.
- Restricțiile de blocare a pinilor sunt scrise într-o secțiune PINLOCK în fişierul UCF.
- Implicit PIN2UCF nu scrie restricții în conflict într-un fişier UCF. Înaintea creerii secțiunii PINLOCK, dacă PIN2UCF găseşte conflicte ale restricțiilor, va raporta într-un fişier numit pinlock.rpt.

- Fişierul raport are două secţiuni: informaţii ale conflictelor restricţiilor şi o listă de erori şi atenţionări.
- Restricţiile de blocare a pinilor nu sunt niciodată rescrise într-un fişier UCF.
- PIN2UCF nu verifică dacă restricţiile existente în UCF sunt restricţii valide de blocare a pinilor.

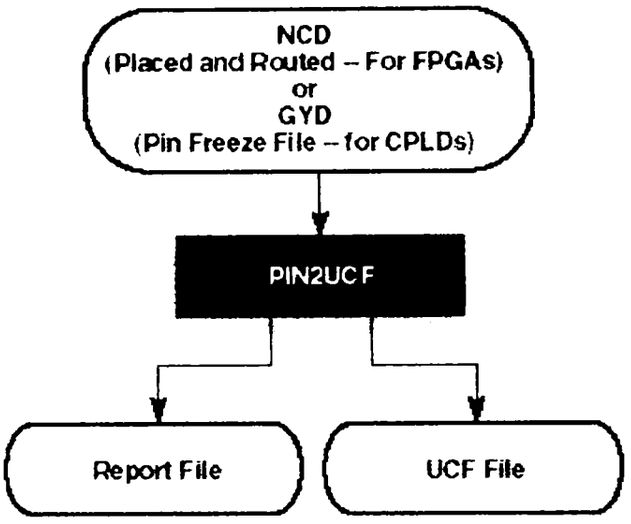


Figura 4.36. PIN2UCF

EPIC (Editor for Programmable Integrated Circuits) este un editor de nivel scăzut care este utilizat pentru afişarea şi configurarea proiectelor şi a FPGA-urilor Xilinx. Principalele funcţii ale EPIC sunt:

- Plasarea şi rutare principalelor componente înainte de rularea automată a instrumentelor de plasare şi rutare asupra întregului proiect.
- Modificarea manuală a plasărilor şi rutărilor.
- Interacţiunea cu fişierul de restricţii fizice pentru a creea şi modifica restricţiile.
- Verificarea temporală vis a vis cu restricţiile.
- Creerea macro-urilor fizice (fişiere .nmc)

Operaţiile de editare efectuate în EPIC modifică configuraţia proiectului şi de asemenea schimbă baza de date a proiectului. Editarea funcţiilor include selectarea, adăugarea şi ştergerea obiectelor, vizualizarea şi schimbarea atributelor obiectului, copierea de componente, interschimbarea componentelor şi a pinilor, plasarea componentelor şi rutarea. În figura 4.37 este indicată desfăşurarea procesului EPIC.

TRACE este un utilitar de analiză statică care poate fi utilizat pentru a vedea dacă restricţiile impuse de-a lungul procesului de proiectare au fost respectate. După aplicarea acestui utilitar rapoartele date de TRACE vor indica toate eroriile apărute.

Informațiile din "timing report" pot fi utilizate pentru evaluarea temporală a unui proiect, pentru a face eventuale ajustări. TRACE poate fi folosit pentru analize de timp chiar dacă nu sunt impuse restricții de către utilizator [Xili-98i].

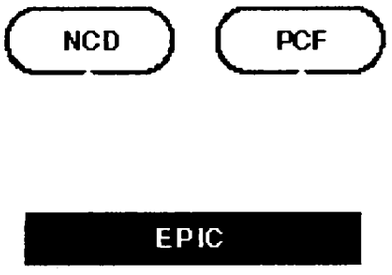


Figura 4.37. EPIC

Fişierele de intrare în TRACE (figura 4.38) sunt:

- Fişier NCD – schemă mapată. Tipul informațiilor depinde de faptul dacă schema este neplasată, a fost plasată sau a fost plasată şi rutată.
- PCF - un fişier ASCII care conține restricțiile fizice. Fişierul este produs de către programul MAP şi poate fi modificat de către utilizator. Restricțiile de timp pot fi scrise în acest fişier.

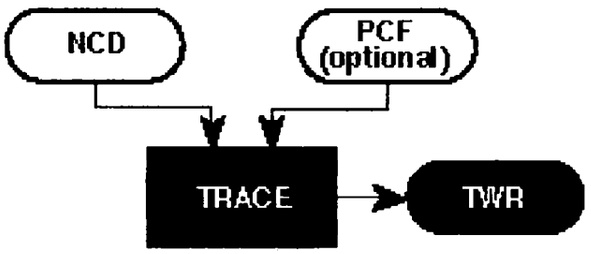


Figura 4.38. TRACE

Restricțiile pot indica: frecvența tactului pentru semnalele de intrare, relații de timp între două sau mai multe semnale, întârzierea maximă absolută pe o cale sau restricții generale de timp pentru o clasă de pini.

Fişierul de ieşire este un raport scris într-un fişier cu extensia TWR şi care va fi scris în directorul curent de lucru. Raportul conține statistici referitoare la proiect, un

sumar al eroriilor şi atenţionărilor şi, opţional, rapoarte detaliate referitoare la întârzierrile pe semnale sau căi. Există trei tipuri de rapoarte: un raport sumar care conţine informaţii sumare şi statistici, un raport al eroriilor care listează aceste erori şi un raport în cuvinte care conţine informaţii referitoare la toate semnalele şi căile. Tipul raportului produs este determinat de opţiunile din linia de comandă TRACE.

Eroriile de timp indică nerespectarea restricţiilor de timp relative sau absolute. Acestea includ eroriile care se referă la depăşirea timpului maxim de întârziere pe o cale a semnalului, depăşirea timpului maxim de întârziere a unui semnal sau deformarea semnalului.

Atenţionările subliniază probleme ca bucle de circuit sau restricţii care nu definesc nici o cale.

BitGen (Bitstream Generator Program). După rutarea schemei trebuiesc generate datele binare care pot fi folosite pentru configurarea echipamentului fizic. Acest lucru se poate realiza cu programul BitGen. Având ca intrare un fişier NCD (Native Circuit Description) programul va genera un fişier binar cu extensia .bit, care conţine toate informaţiile configuraţiei din fişierul NCD şi care definesc logica internă şi interconexiunile FPGA-ului plus informaţiile specifice echipamentului din alte fişiere asociate echipamentului [Xili-98i]. Data binară din fişierul BIT poate fi apoi transferată în celulele de memorie ale FPGA sau poate fi utilizată pentru a crea un fişier PROM. Acest proces este descris în diagrama din figura 4.39.

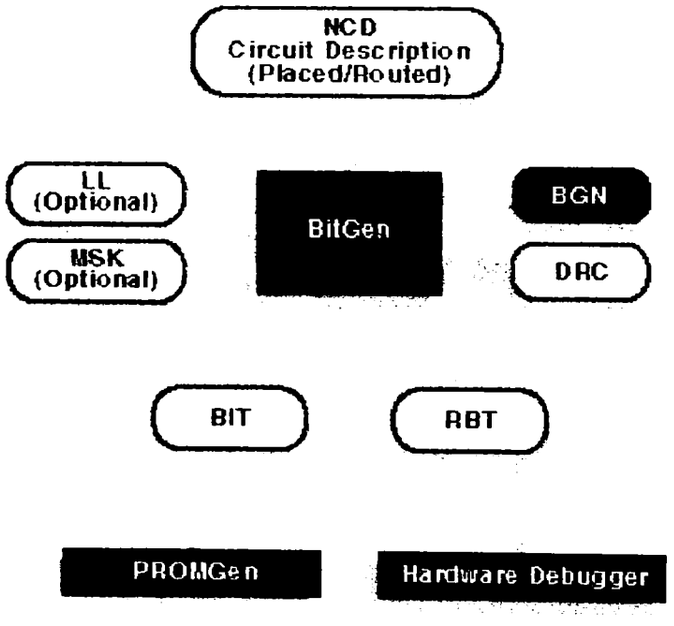


Figura 4.39. BitGen

Fişierele de intrare sunt:

- Fişierul NCD – o descriere fizică a schemei mapate, plasate şi rutate în echipamentul dorit
- Fişierul PCF – un fişier ASCII care poate fi modificat de către utilizator. Dacă în linia de comandă BitGen este specificat un fişier PCF, acest fişier va fi utilizat pentru a determina semnalele critice.

Fişierele de ieşire din BitGen sunt următoarele:

- Fişier BIT – un fişier binar care conţine toate informaţiile referitoare la configuraţie din fişierul NCD în care sunt definite logica internă şi interconexiunile echipamentului FPGA. Data binară din fişierul BIT poate fi încărcată în celulele de memorie ale FPGA sau poate fi folosită pentru a crea un fişier PROM.
- Fişier RBT – este un fişier opţional care conţine biţii iniţiali, neprelucraţi. Fişierul cu biţii iniţiali este un fişier ASCII de zerouri şi unuri care reprezintă datele din fişierul şir de biţi.
- Fişierul LL – un fişier ASCII opţional de alocare care indică poziţia latch-urilor, bistabilelor şi IOB.
- Fişierul MSK – un fişier mască, opţional, folosit pentru compararea locatilor relevante de biţi pentru execuţia unei citiri înapoi a configuraţia datelor conţinute într-un FPGA.
- Fişierul BGN – un fişier raport care conţine informaţii despre execuţia programului BitGen
- Fişier DRC – un fişier de verificare a proiectului.

Înainte de a genera şirul de biţi final trebuie rulat programul cu opţiunea –d pentru a creea un raport DRC (Design Rule Checker). Apoi rularea cu opţiunea –t (Tie operation) va defini niveluri logice pentru orice interconexiune nefolosită din FPGA. În cadrul programului Design Manager, BitGen rulează pe parcursul nivelului de configurare.

PROMGen (PROM File Generator) face conversia unui fişier de tip BIT într-un fişier de format PROM care conţine datele configuraţiei pentru echipamentul FPGA, care poate fi stocată într-un PROM.

Există două versiuni ale programului PROMGen care sunt echivalente din punct de vedere funcţional: o versiune tip text, independentă, care poate fi apelată de la promptul unui sistem de operare şi o versiune tip grafic numită PROM File Formater care poate fi accesată din Design Manager, dar care poate funcţiona şi independent [Xili-98b].

Intrările în PROMGen constau din unul sau mai multe fişiere tip .BIT care conţin datele configuraţiei pentru un proiect FPGA.

Ieşirile din PROMGen (figura 4.40) sunt următoarele fişiere:

- Fişier sau fişiere PROM care conţin informaţii ale configuraţiei PROM. Funcţie de formatul fişierului PROM utilizat de programatorul PROM fişierele de ieşire sunt de tip TEK, MCS sau EXO, iar dacă se foloseşte un

microprocesor pentru configurarea echipamentelor fişierul de ieşire este de tip HEX şi conţine o reprezentare hexazecimală a şirului de biţi.

- Fişiere PRM care conţin o hartă a memoriei a fişierului PROM de ieşire asociat.

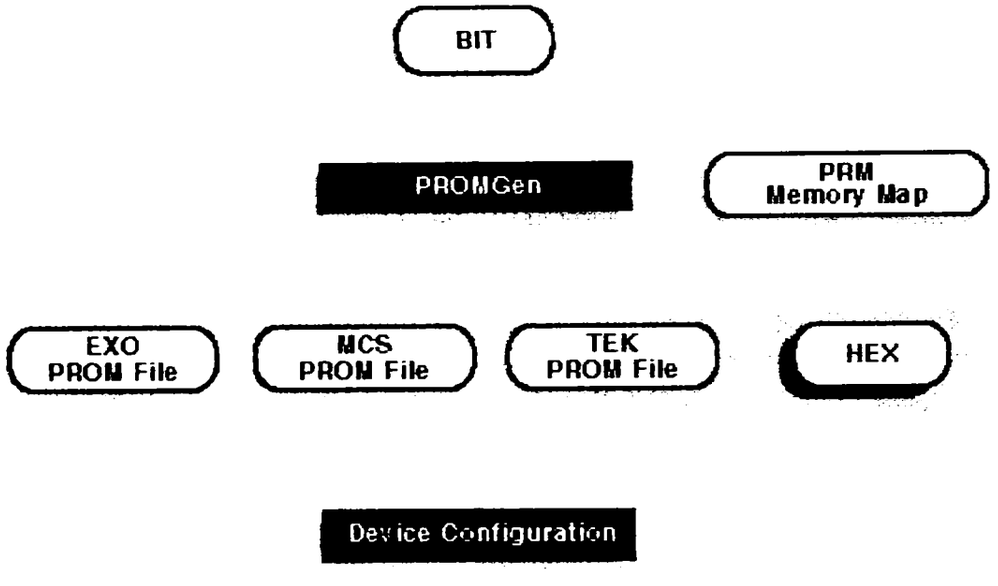


Figura 4.40. Programul PROMGen

4.5.4. Verificarea proiectului

Este procesul de testare a funcţionalităţii şi performanţelor proiectului. Xilinx Development System pune la dispoziţie trei metode complementare de verificare: simulare, analiza statică de timp şi verificarea "in-circuit".

Metodele de verificare sunt ilustrate în figura 4.41.

Fiecare tip de verificare utilizează diferite instrumente de proiectare, care sunt ilustrate în figura 4.42.

Înainte de simulare, informaţiile referitoare la proiectul fizic trebuiesc translatate şi redistribuite proiectului logic. Acest proces numit "back-annotation" este realizat cu programul numit NGDAnno. În procesul "back-annotation" informaţiile proiectului fizic, inclusiv valorile de timp sunt redistribuite proiectului logic pentru simularea "back-end" În Xilinx Development System acest proces se desfăşoară astfel:

NGDAnno distribuie întârzierile, timpii de setup şi de hold precum şi durata pulsului din fişierul NCD al proiectului fizic în proiectul logic reprezentat în fişierul NGD. De asemenea locaţiile componentelor logice pentru PAD-uri sunt combinate cu informaţiile din fişierul NGD. Procesul "back-annotation" este prezentat în figura 4.43.

Fişierul NGA adnotat este intrare pentru unul din inscriptoarele netlist (NGD2EDIF, NGD2VER sau NGD2VHDL), care transformă informaţia "back-annotated" în format netlist pentru simulare [Xili-98c].

Xilinx Development System permite execuţia "back-annotation" unui proiect nerutat sau creerea unui netlist de ieşire pentru a se putea face simularea la diferite niveluri ale proiectării.

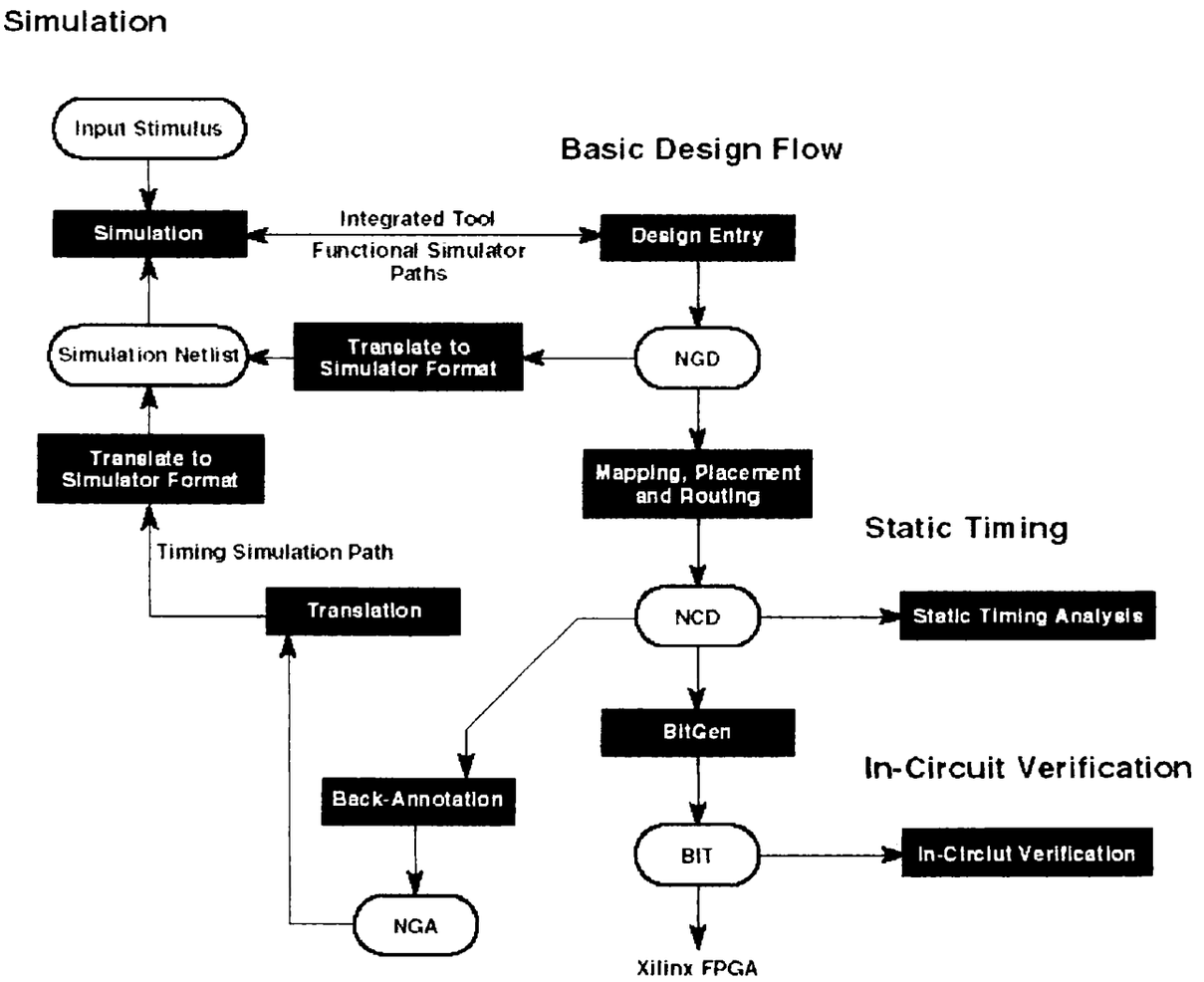


Figura 4.41. Trei metode de verificare în procesul de proiectare

Verification Tools

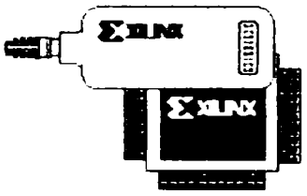
Type of Verifcation	Tools
Simulation 	• Third Party Simulators (Integrated and Non-Integrated)
Static Timing **MHz**	• Trace (Command Line) • Timing Analyzer (Design Manager)
In-Circuit Verification 	• Design Rule Checker • Download or XChecker Cable

Figura 4.42. Instrumente de verificare

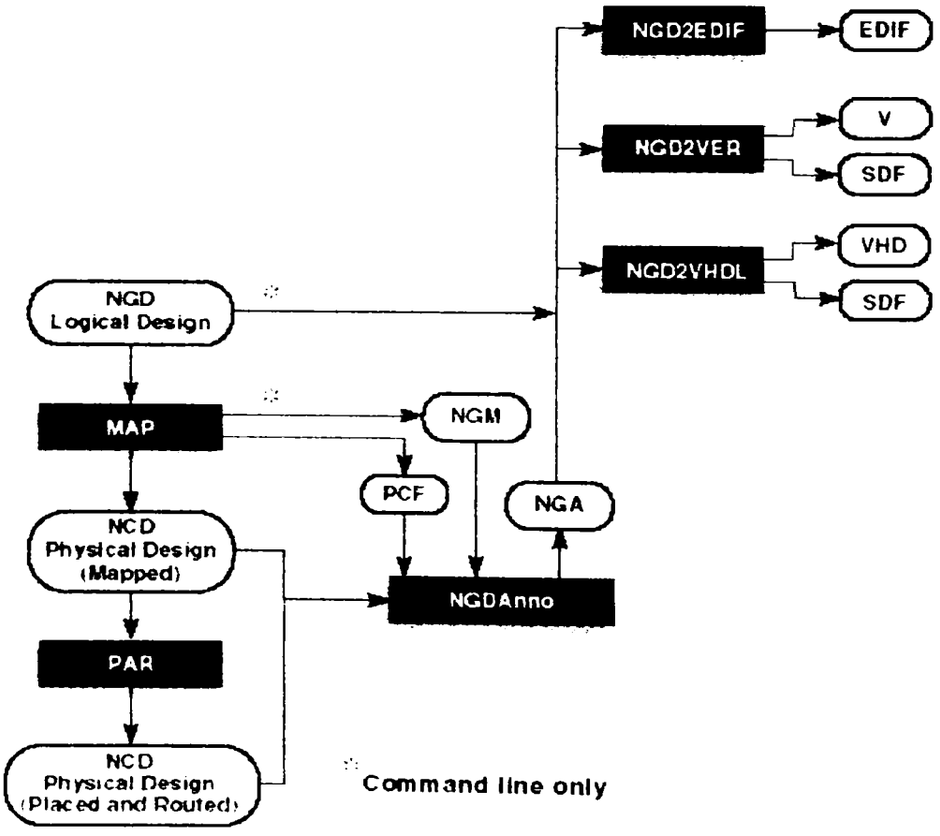


Figura 4.43. "Back-Annotation"

NGDAnno combină informațiile din fişierul NMG cu informațiile de amplasare, rutare şi timing din fişierul NCD (Native Circuit Description). Aceste date sunt combinate într-un fişier NGA. Acest fişier este intrare pentru programele care vor converti baza de date binară Xilinx într-un fişier netlist ASCII [Xili-98b].

Fişierele de intrare în programul NGDAnno sunt:

- Fişier NCD – fişierul proiectului fizic. Proiectul poate fi doar mapat, parțial sau complet plasat şi parțial sau complet rutat.
- Fişier NMG – fişier NGD mapat creat de programul MAP.
- Fişier PCF – fişierul restricților fizice.

Fişierele de ieşire sunt:

- Fişierul NGA
- Fişierul ALF - un fişier raport care conține informații referitoare la execuția programului NGDAnno.

Inscriptoare de netlist preiau ieşirea din NGDAnno şi crează un fişier netlist de simulare în formatul specificat. Intrările sunt fişiere de tip NGD – un fişier al proiectului logic care conține componentele elementare - sau de tip NGA – un fişier un fişier al proiectului logic "back-annotated". Programele netlist recunoscute de programul de implementare M1 sunt: NGD2EDIF, NGD2VER şi NGD2VHDL.

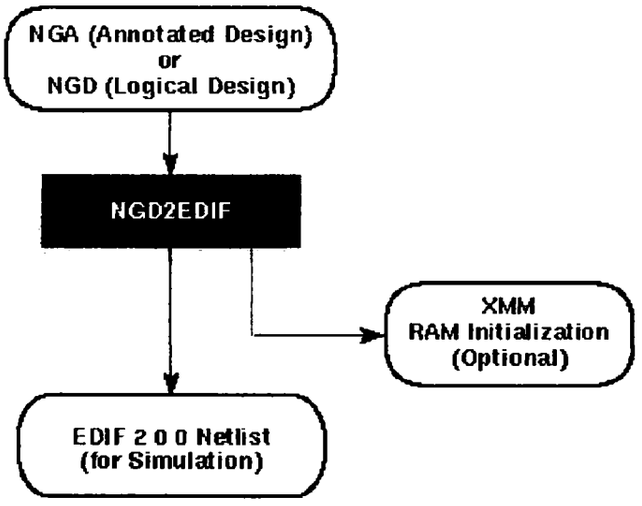


Figura 4.44. Procesul NGD2EDIF

NGD2EDIF produce un netlist EDIF200 în termeni ai setului de primitive Xilinx care permite simularea unui proiect pre şi post rutat. Fişierul EDIF poate reprezenta un proiect în unul din următoarele niveluri:

- Un proiect nemapat – pentru translatarea unui proiect nemapat, intrarea este un fişier NGD. Ieşirea este un fişier EDIF care conţine o descriere funcţională a proiectului fără nici o informaţie de timp.
- Un proiect mapat şi nerutat – pentru translatarea unui proiect mapat care nu a fost plasat şi rutat intrarea este un fişier NGA. Fişierul EDIF de ieşire conţine o descriere funcţională a proiectului şi informaţii de timp referitoare la întârzierile componentelor dar fără întârzireri de rutare
- Un proiect rutat - pentru translatarea unui proiect care a fost plasat şi rutat intrarea este un fişier NGA generat dintr-un proiect fizic rutat. Fişierul EDIF de ieşire conţine o descriere funcţională a proiectului şi informaţii de timp referitoare la întârzierile componentelor şi întârzierile de rutare.

Procesul NGD2EDIF este prezentat în figura 4.44.

NGD2VER transformă schema într-un fişier Verilog HDL care conţine o descriere netlist a proiectului în termeni ai primitivelor de simulare Xilinx [Xili-98g]. Fişierul Verilog poate fi utilizat pentru a executa o simulare "back-end" cu simulatorul Verilog. Acest fişier poate fi produs de NGD2VER în oricare din următoarele niveluri:

- Un proiect nemapat – pentru translatarea unui proiect nemapat intrarea trebuie să fie un fişier NGD – o descriere logică a proiectului. Fişierul Verilog de ieşire conţine o descriere funcţională a proiectului, fără nici o specificaţie de timp.
- Un proiect mapat dar nerutat - pentru translatarea unui proiect mapat care nu a fost plasat şi rutat intrarea trebuie să fie un fişier NGA – o descriere logică adnotată a proiectului – generată dintr-un proiect fizic mapat. Fişierul Verilog de ieşire conţine o descriere funcţională a proiectului şi un fişier SDF (Standard Delay Format) adiţional, care conţine întârzierile fără nici o specificaţie de timp.

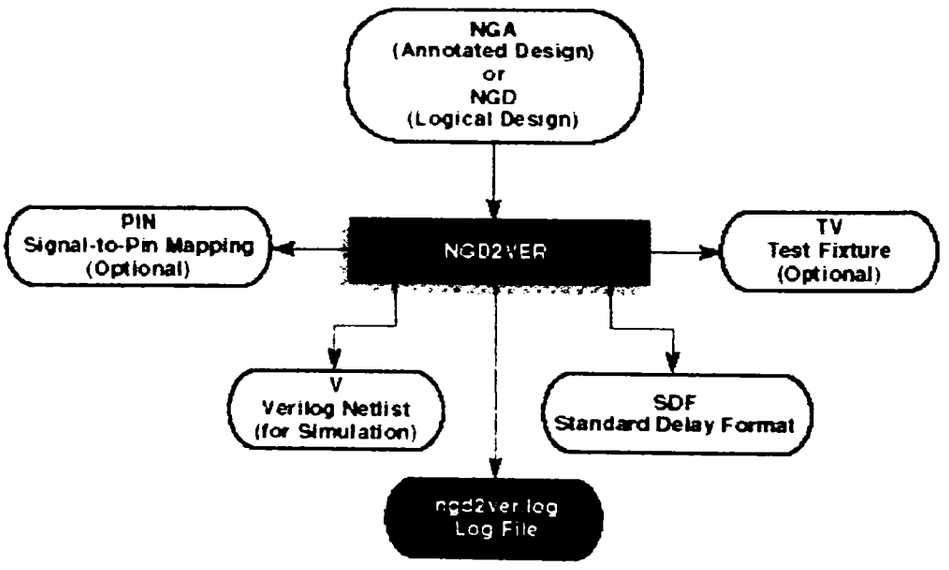


Figura 4.45. Procesul NGD2VER

- Un proiect rutat – pentru translatarea unui proiect care a fost plasat şi rutat intrarea este un fişier NGA generat dintr-un proiect fizic rutat. Fişierul Verilog de ieşire conţine o descriere funcţională a proiectului şi un fişier SDF care conţine întârzierile componentelor şi cele de rutare.

Procesul NGD2VER este descris în figura 4.45.

Ieşirile lui NGD2VER sunt următoarele fişiere:

- Fişier V – un fişier Verilog HDL [Xili-98g] care conţine informaţii netlist obţinute din fişierul de intrare NGD sau NGA. Acest fişier este un model de simulare şi nu poate fi folosit sau sintetizat în nici o altă manieră decât simularea. Acest netlist utilizează primitive de simulare care care nu reprezintă adevărata implementare a echipamentului.
- Fişier SDF – conţine întârzierile obţinute din fişierul de intrare. Acest fişier este generat doar dacă intrarea este un fişier NGA care conţine informaţii de timp.
- Fişier LOG - un fişier opţional care conţine toate mesajele generate în timpul execuţiei programului NGD2VER.
- Fişier TV – un fişier test ataşat.
- Fişier PIN – fişier generat dacă fişierul de intrare conţine pini externi rutaţi.

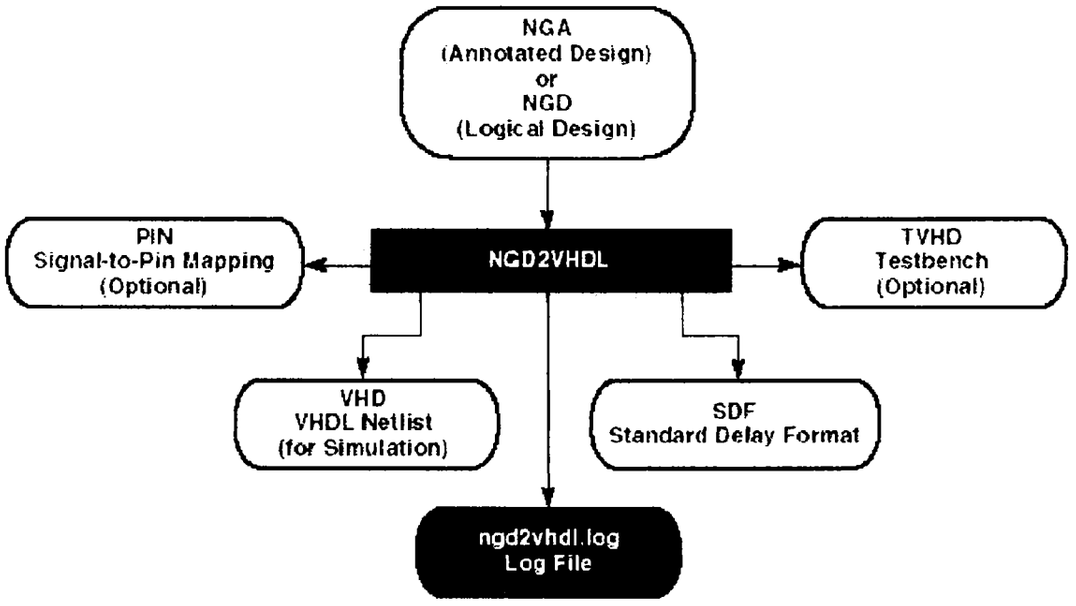


Figura 4.46. Procesul NGD2VHDL.

NGD2VHDL transformă proiectul într-un fişier VHDL care conţine o descriere netlist a proiectului în termeni ai primitivelor de simulare Xilinx. Fişierul VHDL poate fi utilizat pentru a executa o simulare "back-end" cu simulatorul VHDL [Xili-98f]. Acest fişier poate fi produs de NGD2VHDL în oricare din următoarele niveluri:

- Un proiect nemapat – pentru translatarea unui proiect nemapat intrarea trebuie să fie un fişier NGD – o descriere logică a proiectului. Fişierul VHDL de ieşire conţine o descriere funcţională a proiectului, fără nici o specificaţie de timp.
- Un proiect mapat dar nerutat - pentru translatarea unui proiect mapat care nu a fost plasat şi rutat intrarea trebuie să fie un fişier NGA – o descriere logică adnotată a proiectului – generată dintr-un proiect fizic mapat. Fişierul VHDL de ieşire conţine o descriere funcţională a proiectului şi un fişier SDF (Standard Delay Format) adiţional, care conţine informaţii de timp.
- Un proiect rutat – pentru translatarea unui proiect care a fost plasat şi rutat intrarea este un fişier NGA generat dintr-un proiect fizic rutat. Fişierul VHDL de ieşire conţine o descriere funcţională a proiectului şi un fişier SDF care conţine întârzierile componentelor şi cele de rutare.

Procesul NGD2VHDL este descris în figura 4.46.

Ieşirile lui NGD2VHDL sunt următoarele fişiere:

- Fişier VHD – un fişier VHDL care conţine informaţii netlist obţinute din fişierul de intrare NGD sau NGA. Acest fişier este un model de simulare şi nu poate fi folosit sau sintetizat în nici o altă manieră decât simularea. Acest netlist utilizează primitive de simulare care care nu reprezintă adevărata implementare a echipamentului.
- Fişier SDF – conţine întârzierile obţinute din fişierul de intrare. Acest fişier este generat doar dacă intrarea este un fişier NGA care conţine informaţii de timp.
- Fişier LOG - un fişier opţional care conţine toate mesajele generate în timpul execuţiei NGD2VHDL.
- Fişier "Testbench" – un fişier test opţional
- Fişier PIN – fişier generat dacă fişierul de intrare conţine pini externi rutaţi.

Simularea pe baza schemei

Simularea proiectului presupune testarea proiectului folosind modele software. Aceste modele sunt concepute astfel încât să efectueze o caracterizare detaliată a proiectului [Bout-94].

"Functional simulation" determină dacă logica schemei este corectă. Acest lucru se face înainte de implementarea schemei. Simularea funcţională se poate efectua chiar de la nivelurile primare din procesul de proiectare, putând astfel corecta mai rapid şi mai uşor eroriile de proiectare. Deoarece informaţile de timp nu sunt disponibile la acest nivel simulatorul foloseşte unităţi de timp când face testarea logică [Xili-98i].

În figura 4.41 sunt indicate procesele de proiectare pentru instrumentele de simulare integrate şi neintegrate. Metodele integrate, ca Mentor sau Viewlogic conţin interfeţe "built-in" care leagă editorul de scheme cu simulatorul, permiţând folosirea aceluiaşi netlist.

"Timing simulation" verifică dacă schema proiectată funcţionează la viteza dorită pentru echipamentul ales chiar şi în cele mai proaste condiţii. Acest proces are

loc după ce proiectul este mapat, plasat şi rutat. În acest moment sunt deja cunoscuți toți timpii de întârziere. În figura 4.47 este desris modul în care are loc timing simulation. Acum pot fi verificate relațiile de timp şi se pot determina căile critice din schemă pentru condițiile cele mai dezavantajoase. Pentru introducerea informațiilor de timp în proiect trebuieşte convertit fişierul NCD într-un fişier NGA, operație care este făcută de NGD2EDIF, NGD2VER sau XNF2NGD.

Simularea pe baze HDL

Soluțiile de simulare VHDL şi Verilog permit administrarea modificărilor schemei mai rapid şi mai eficient atunci când se folosesc combinații de echipamente FPGA şi CPLD [Xili-98b]. Solutiile de simulare VHDL acceptă standardul VITAL care permite simularea cu Mentor QuickHDL, Synopsys VSS şi Cadence Leapfrog.

Analiza statică de timp folosind aplicația TRACE

Static timing Analysis este cea mai indicată pentru verificări rapide ale schemei după ce s-au efectuat operațiile de plasare şi rutare.

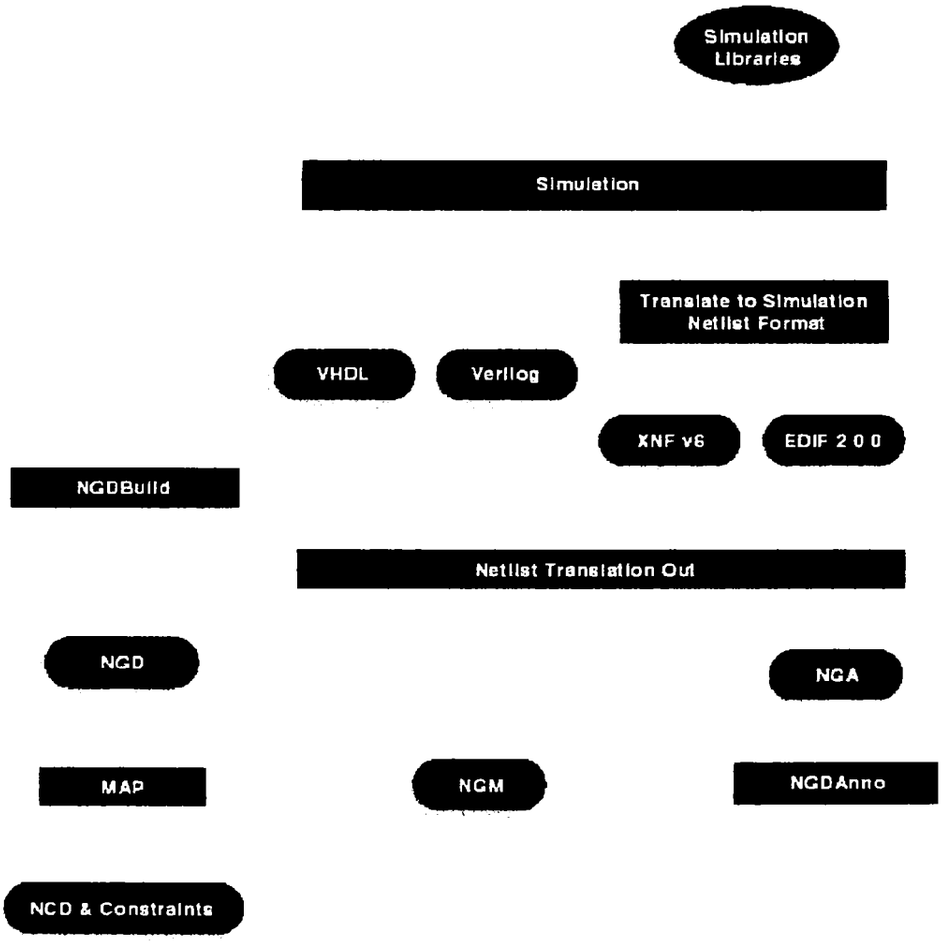


Figura 4.47. Procesul de verificare a proiectului – Simularea la nivel de poartă

TRACE (timing Reporter and Circuit Evaluator) este o aplicație Xilinx proiectată pentru a oferi o analiză statică de timp și poate fi utilizată pentru a evalua modul cum au fost respectate restricțiile de timp de la intrare precum și modul de amplasare și rutare (vezi figura 4.33). Utilizând TRACE se pot rapid verifica problemele de timp și întârzierile pe ramuri. TRACE efectuează două funcții majore: verificări de timp și rapoarte.

Verificarea "in-circuit"

Ca test final se poate verifica modul de funcționare a schemei în echipamentul propriu-zis. Verificarea "in-circuit" testează circuitul în condiții concrete de operare.

Înainte de generarea șirului final de biți este importantă folosirea opțiunii DRC (Design Rule Checker) in BitGen pentru evalua fișierul NCD (Native Circuit Description). Eventualele probleme care apar și pot fi înlăturate vor preveni o funcționare incorectă.

Pentru o verificare "in-circuit" încărcarea șirului de biți în echipament se va face folosind un cablu special, Xchecker.

4.6. Concluzii

Având ca obiectiv implementarea algoritmului de criptare RC6, în acest capitol am descris mediul experimental folosit. În urma analizei atente făcute asupra posibilelor sisteme de proiectare existente, am ales Xilinx Foundation Series versiunea 1.5, care mi-a fost pusă cu generozitate la dispoziție de firma Xilinx și plăcile fabricate de firma Xess cu circuite din familia Virtex 2.5V, cele mai noi produse ale firmei Xilinx. Acest capitol analizează în prima parte a sa circuitele FPGA ale firmei Xilinx, iar în cea de a doua parte sistemul de dezvoltare Foundation.

Principalele contribuții aduse prin acest capitol se referă la:

1. O analiză a circuitelor logice programabile cu accente pe cele mai noi tipuri de astfel de circuite în scopul alegerii circuitelor folosite pentru implementarea circuitului de criptare/decriptare autotestabil bazat pe cifrul RC6;
2. Analiza comparativă a sistemelor de proiectare oferite de firmele Altera, Actel, Intel și Xilinx;
3. O sinteză a sistemului de proiectare Foundation cu explicarea detaliată prin cele 26 de scheme logice a tuturor etapelor parcurse precum și a facilităților oferite de acesta.

CAPITOLUL 5

CRIPTOR – CIRCUIT DE CRIPTARE CU FACILITĂȚI DE ATOTESTARE

5.1. Limbaje de descriere hardware utilizate în implementarea algoritmului RC6

În acest capitol prezint două dintre implementările algoritmului RC6, pe care le-am realizat utilizând limbaje de descriere hardware (HDL- Hardware Description Language), şi anume o implementare cu limbajul Verilog şi o a doua utilizând limbajul VHDL. Am ales acest mod de proiectare întrucât HDL permite o descriere la nivel mixat, unde construcţiile structurale sau netlist-urile se pot combina cu descrierile comportamentale sau algoritmice. În acest mod am putut descrie structura sistemului la un nivel înalt de abstracţie. În plus metodologia de proiectare care utilizează HDL prezintă câteva avantaje fundamentale faţă de metodologia tradiţională de proiectare la nivel de poartă. Astfel, funcţionarea schemei poate fi verificată rapid în procesul de proiectare şi se poate face apoi imediat simularea schemei descrise ceea ce permite testarea soluţiilor arhitecturale şi de proiectare. Un alt avantaj oferit de VHDL şi Verilog îl reprezintă metodele puternice de verificare.

VHDL a fost eleborat în 1982 de către Departamentul Apărării din Statelor Unite în cadrul programului VHSIC (Very-High-Speed Integrated Circuit). Din punct de vedere al stilului şi sintaxei VHDL este asemănător cu limbajele de programare moderne dar include numeroase construcţii specifice hardware. VHDL este recunoscut ca şi un standard HDL de către IEEE (IEEE Standard 1076) din anul 1987 şi de către Departamentul Apărării Statelor Unite (MIL-STD-454L).

Istoria limbajului Verilog este asociată cu începutul anilor 80, când compania Gateway Design Automation a elaborat un simulator logic, numit Verilog XL, şi odată cu acesta a elaborat şi un limbaj de descriere hardware. În 1989, Cadence Design Systems a achiziţionat firma Gateway şi odată cu aceasta şi drepturile asupra limbajului şi simulatorului. Pentru ca limbajul să devină un standard, în 1990 Cadence a introdus limbajul în domeniul public. În 1995 Verilog HDL a devenit standard IEEE numărul 1364-1995, iar la ora actuală se lucrează la definirea extensiilor analogice ale limbajului Verilog.

Pentru implementarea hardware a algoritmului de criptare RC6 am folosit versiunile VHDL şi Verilog furnizate în cadrul pachetului Foundation Series v.1.5. al firmei Xilinx Inc.

Instrumentele oferite de Foundation Series permit sinteza şi optimizarea logicii, astfel că o descriere VHDL sau Verilog se poate converti într-o implementare la nivel de poartă pentru tehnologia dorită [MaŢi-00a]. Optimizarea oferită, permite transformarea automată a schemei sintetizate într-un circuit mai mic şi mai rapid.

5.2. CRIPTOR – circuit de criptare/decriptare care implementează cifrul bloc RC6

În acest paragraf prezint modul de implementare al algoritmului RC6 utilizând limbajul de descriere hardware VHDL. Având în vedere complexitatea schemei şi analizând caracteristicile plăcilor produse, am ales pentru implementarea fizică plăcile din familia VIRTEX, XCV1000, descrise în capitolul 4. Această placă are o capacitate de 1.124.000 de porţi logice echivalente şi un număr de 512 buffer-e de intrare/ieşire.

Algoritmul criptografic RC6 poate fi descris în funcţie de mărimea blocurilor de date cu care se lucrează, de lungimea cheii utilizator şi de numărul de runde. În această lucrare, utilizând limbajul de descriere hardware VHDL, am implementat cifrul bloc RC6 care lucrează cu blocuri de 128 de biţi, cu o lungime a cheii utilizator de 256 de biţi şi are un număr de 20 de runde, asigurând astfel cerinţele de securitate impuse de NIST şi totodată reuşind să mă încadrez în resursele disponibile ale plăcii. Pentru a obţine o schemă cât mai economică, cele 20 de runde sunt executate de un singur modul care este apelat de 20 de ori. Am denumit circuitul CRIPTOR.

VHDL împarte entităţile (circuit sau sistem) în două părţi: una externă sau parte vizibilă (nume entitate şi conexiuni) şi o parte internă sau ascunsă (algoritmi şi implementare). În VHDL o entitate este definită relativ faţă de alte entităţi prin conexiunile ei şi comportare. Se pot studia sau realiza diferite implementări (arhitecturi) ale unei entităţi fără a schimba schema bloc.

Deoarece capacitatea plăcilor nu a permis implementarea pe o singură placă a întregului sistem care realizează criptarea şi decriptarea, am implementat fizic separat modulul de criptare respectiv cel de decriptare. În figura 5.1 este prezentată schema bloc pentru structurile de criptare/decriptare.

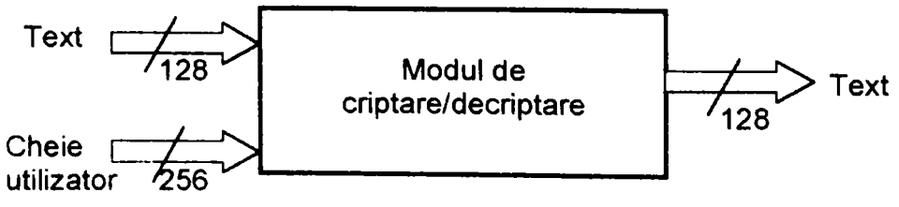


Figura 5.1. Schema bloc a circuitului de criptare/decriptare

Atât circuitul de criptare cât şi cel de decriptare sunt alcătuite din trei componente: modulul de generarea al cheilor, modulul de criptare (de decriptare) şi structura de comandă a criptării (decriptării). Întrucât modulul de generare al cheilor din algoritmul RC6 este identic atât în cazul criptării cât şi al decriptării acesta a fost reprezentat ca bloc distinct în figurile următoare. Modulele de generare ale cheilor fac parte din entitățile corespunzătoare structurilor de comandă pentru criptare, respectiv decriptare. Acest lucru l-am pus în evidență în figura 5.2 prin linia punctată. Aşa cum se vede şi în această figură structura circuitului de criptare/decriptare conține două entități, cea de-a doua fiind cea corespunzătoare modulului de criptare/decriptare pentru o rundă [MaȚi-00b].

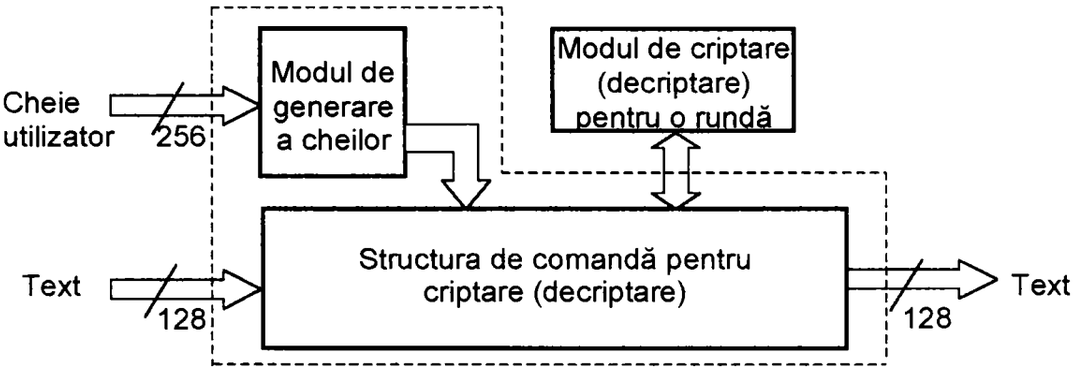


Figura 5.2. Structura circuitului de criptare/decriptare

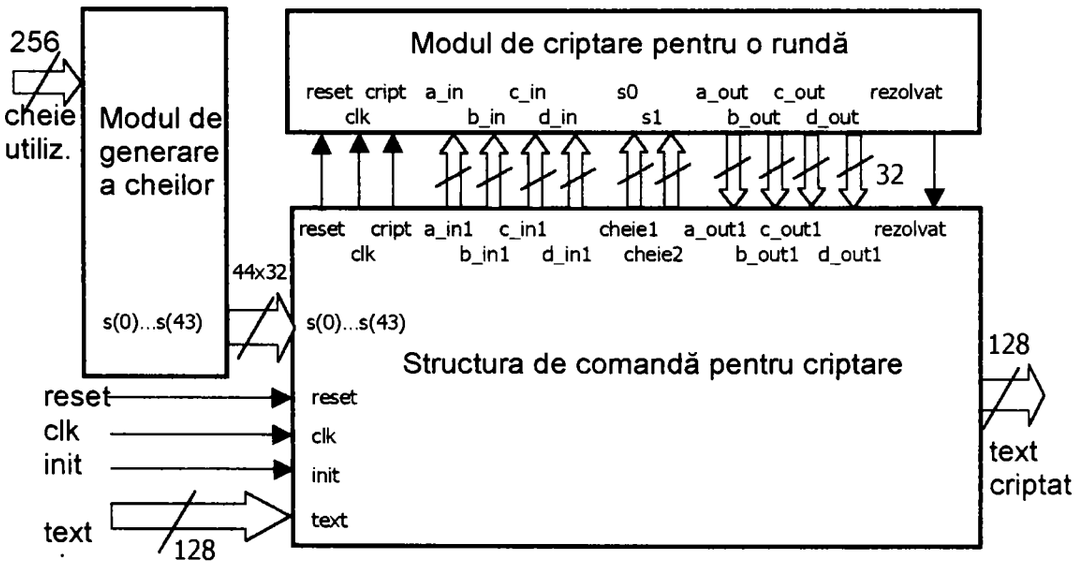


Figura 5.3. Structura detaliată a modulului de criptare

Subcheile sunt furnizate modulului principal care coordonează întregul proces de criptare. Modulul reprezentând structura de comandă pentru criptare apelează modulul de criptare pentru o rundă, furnizând acestuia blocurile de date şi cele două subchei corespunzătoare rundei respective concomitent cu activarea semnalului *cript*. După ce modulul de criptare pentru o rundă a realizat calculele necesare, semnalul *rezolvat* devine "1", iar rezultatele sunt transmise modulului principal. La finalul celor 20 de runde la ieşirea circuitului vom avea textul criptat.

Am implementat algoritmul pe baza maşinilor cu stări finite (FSM - Finite State Machine). Un FSM este de fapt un circuit secvenţial sincron în care starea curentă a maşinii este păstrată în memorie, în bistabile sincronizate cu semnalul de tact. Pentru o anumită secvenţă de intrări, FSM produce o secvenţă de ieşiri dependentă de starea curentă, funcţia tranziţiilor şi funcţia de ieşire.

Pentru implementarea FSM, în acest paragraf am utilizat limbajul de descriere hardware VHDL. În proiectarea HDL trebuiesc respectate anumite reguli, ca de exemplu: se foloseşte un singur FSM în cadrul unui proces, iar în acest proces nu se mai include logică suplimentară faţă de cea care descrie FSM-ul.

Descrierea fiecărui modul din figura 5.3, am făcut-o pe baza câte unui FSM, separat, aşa cum se va vedea în următoarele diagrame de stări [MaŢi-00b].

Diagrama stărilor pentru modulul principal de criptare este prezentată în figura 5.4. FSM-ul corespunzător criptătării conţine cinci stări: *asteapta, gen_subchei, apel_criptare, rundă_nouă şi final*.

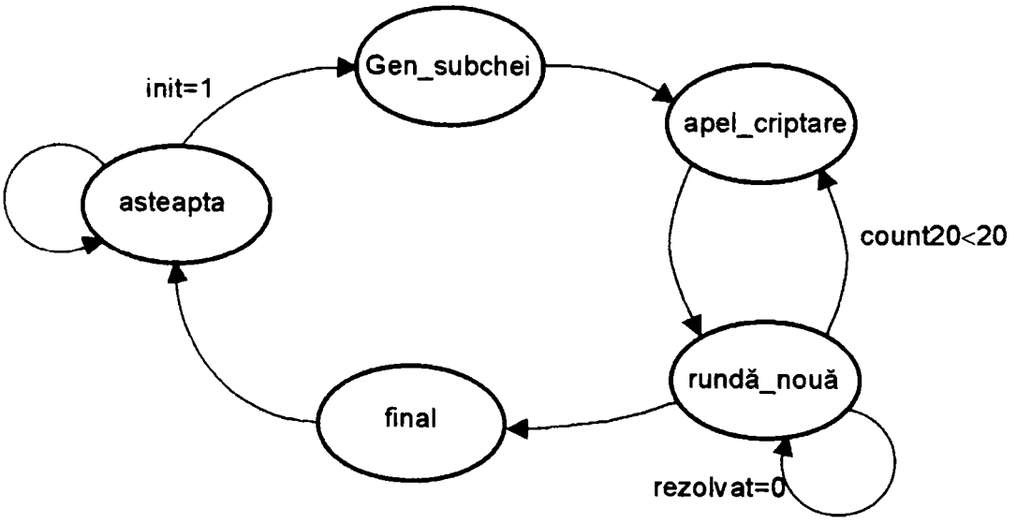


Figura 5.4. Diagrama stărilor pentru modulul principal de criptare

Trecerea de la o stare la alta se face în funcţie de valoarea semnalelor de condiţie (*init, ccunt20, rezolvat*) şi se realizează sincronizat cu semnalul de tact (*clk*).

Asteapta este o stare de aşteptare. În momentul în care se activează semnalul *init*, la primul front crescător al tactului se trece în starea *gen_subchei*. În această stare se realizează iniţializarea variabilelor şi a tabelului cu subchei. Pe următorul front crescător al tactului se realizează trecerea în starea *apel_criptare*, se activează semnalul *cript* şi sunt trimise datele către registrii A, B, C, D şi cele două subchei corespunzătoare primei runde de criptare. Activarea semnalului *cript* determină ieşirea modulului de criptare pentru o rundă din starea de aşteptare. La următorul tact se dezactivează semnalul *cript* şi se trece în starea *rundă_nouă* în care se rămâne atâta timp cât semnalul rezolvat are valoarea 0. Semnalul *rezolvat*, furnizat de modulul de criptare, indică structurii de comandă pentru criptare terminarea procesului şi activarea semnalelor *a_out1*, *b_out1*, *c_out1* şi *d_out1* (textul criptat după acea rundă). În momentul în care *rezolvat* devine 1 se verifică valoarea contorului *count20* care indică numărul de runde parcurse. Dacă contorul are o valoare mai mică decât 20 se revine în starea *apel_criptare* şi se continuă procesul de criptare cu runda următoare. Dacă contorul a atins valoarea 20 se trece în starea *final*. În starea *final* este transmis pe ieşire textul criptat.

Semnalele *reset* şi *init* sunt semnale asincrone, furnizate din exterior şi au rolul de resetare a modulului, respectiv de iniţializare a procesului de criptare.

Implementarea în VHDL a modulului de criptare este prezentată în anexa 2. Pentru aceasta am folosit modelarea structurală. Fiecare entitate, care reprezintă elementul de bază al unei scheme în VHDL, a fost modelată ca şi un set de componente interconectate prin intermediul semnalelor. Modul de funcţionare al entităţii este descris în cadrul arhitecturii. În cadrul entităţii sunt enumerate semnalele de intrare şi de ieşire ale modulului de criptare (*reset, clk, init* – semnalul de iniţializare, *s* - tabelul cu subchei şi *text* – magistrala bidirecţională pentru textul clar respectiv cel criptat).

Arhitectura modulului de criptare conţine procesele care definesc funcţionarea acestuia: *syncronous_p* – unde se urmăreşte starea semnalelor *reset* şi *clk* în scopul sincronizării funcţionării FSM, *async_p* – descrie funcţionarea şi ordinea de parcurgere a stărilor în FSM conform diagramei din figura 5.4, *final_p* – furnizează la ieşire textul criptat, *generarea_cheilor_p* - se iniţializează cele 44 de subchei cu valorile furnizate de modulul de generare al cheilor, *crip_signal_p* - activează semnalul *crip* în cazul unei noi runde de criptare, *criptare_p* – realizează controlul schimbului de date cu modulul de criptare pentru o rundă şi procesul *counter20_p* - incrementează contorul *count20* care indică numărul de runde parcurse. În cadrul arhitecturii, procesele sunt executate concurent iar instrucţiunile din cadrul acestora se execută secvenţial.

În figura 5.5 sunt prezentate procesele care coordonează funcţionarea modulului de criptare, procese ce au fost descrise în aliniatul precedent.

După cum se observă, în cadrul arhitecturii imediat după declaraţiile locale, este declarată componenta de criptare pentru o rundă, *rc6_cript*, iar după instrucţiunea *begin* a arhitecturii se realizează legarea semnalelor în ordinea enumerării lor în lista *port map*.

```
u1: rc6_cript
port map (reset, clk, crip, a_in1, b_in1, c_in1, d_in1, cheie1,
          cheie2, a_out1,  b_out1,  c_out1,  d_out1, rezolvat);

syncronous_p: process(reset, clk, next_s)
      begin
         if reset='1' then
            current_s<=idle_s;
          elsif (clk'event and clk='1') then
              current_s<= next_s;
         end if;
      end process;

async_p: process (current_s, init, count20, rezolvat, clk)
      begin
          case current_s is
              when idle_s =>
                  if init='1' then
                        next_s <=gen_subchei;
                  else
                        next_s <=idle_s;
                  end if;

              when gen_subchei =>
                  next_s <=apel_criptare;

              when apel_criptare =>
                  next_s <=runda_noua;

              when runda_noua =>
                  if rezolvat='1' then
                        if count20 = 20  then
                              next_s <=final;
                        else
                              next_s <=apel_criptare;
                        end if;
                  else
                        next_s<=runda_noua;
                  end if;

              when final =>
                  next_s <=idle_s;

          end case;
       end process;
```

Figura 5.5. Descrierea proceselor syncronous_p şi async_p

Controlul schimbului de date cu modulul de criptare pentru o rundă este descris în procesul *criptare_p* din figura 5.6.

```
criptare_p: process (clk, current_s, next_s, s, text, a_out1,
                     b_out1, c_out1, d_out1, count20)
  variable a_in :  std_logic_vector(31 downto 0);
  variable b_in :  std_logic_vector(31 downto 0);
  variable c_in :  std_logic_vector(31 downto 0);
  variable d_in :  std_logic_vector(31 downto 0);
  variable c0, c1:integer range 4 to 43;
  begin
     if clk'event and clk='1' then
       if current_s=gen_subchei and next_s=apel_criptare
         then
          a_in:=text(103 downto 96) & text(111 downto 104)
               & text(119 downto 112) & text(127 downto 120);
          b_in:=text(71 downto 64) & text(79 downto 72)
               & text (87 downto 80) & text(95 downto 88);
          c_in:=text(39 downto 32) & text(47 downto 40)
               & text (55 downto 48) & text(63 downto 56);
          d_in:=text(7 downto 0) & text(15 downto 8)
               & text (23 downto 16) & text(31 downto 24);

          a_in1<=a_in;
          b_in1<=b_in+s(0);
          c_in1<=c_in;
          d_in1<=d_in+s(1);
          cheie1<=s(2);
          cheie2<=s(3);
       elsif next_s=apel_criptare then
          c0:=2*count20+2;
          c1:=c0+1;
          a_in1<=a_out1;
          b_in1<=b_out1;
          c_in1<=c_out1;
          d_in1<=d_out1;
          cheie1<=s(c0);
          cheie2<=s(c1);
     end if;
   end if;
end process;
```

Figura 5.6. Descrierea proceselui criptare_p

Conform algoritmului RC6, descris în capitolul 2, textul clar este împărțit în patru blocuri de câte 32 de biți fiecare (A, B, C, D). Primul byte al textului clar este plasat pe poziția byte-ul cel mai puțin semnificativ al registrului A. Ultimul byte al textului clar este plasat pe poziția celui mai semnificativ byte al registrului D. Acest lucru l-am realizat în VHDL prin instrucțiunile de atribuire de tipul:

```
a_in:=text(103 downto 96) & text(111 downto 104)
     & text(119 downto 112) & text(127 downto 120);
```

unde & semnifică operatorul de concatenare.

Structura de comandă pentru criptare furnizează modulului de criptare pentru o rundă tot câte două chei (cheile s(2*i) şi s(2*i+1)). Întrucât contorul *count20* este incrementat doar după ce starea curentă devine *apel_criptare*, în procesul *criptare_p* la testarea condiţiilor *next_s = apel_criptare* contorul are încă valoarea precedentă. După îndeplinirea condiţiilor iniţiale, pentru începerea criptării se parcurg secvenţele care urmează după cuvântul cheie *elsif*.

O operaţie similară cu cea de inversare a octeţilor de la intrare are loc şi la sfârşitul procesului de criptare, *final_p*, descris în figura 5.7.

```
final_p: process(clk, next_s, a_out1, b_out1, c_out1, d_out1,s)
  variable a_out: std_logic_vector(31 downto 0);
  variable b_out: std_logic_vector(31 downto 0);
  variable c_out: std_logic_vector(31 downto 0);
  variable d_out: std_logic_vector(31 downto 0);
  begin
   if (clk'event and clk='1') then
   if next_s=final then
     a_out:=a_out1+s(42);
     b_out:=b_out1;
     c_out:=c_out1+s(43);
     d_out:=d_out1;
     text(127 downto 96)<=a_out(7 downto 0)& a_out(15 downto 8)
                & a_out(23 downto 16) & a_out(31 downto 24);
     text(95 downto 64)<=b_out(7 downto 0) & b_out(15 downto 8)
                & b_out(23 downto 16) & b_out(31 downto 24);
     text(63 downto 32)<=c_out(7 downto 0) & c_out(15 downto 8)
                & c_out(23 downto 16) & c_out(31 downto 24);
     text(31 downto 0)<=d_out(7 downto 0) & d_out(15 downto 8)
                & d_out(23 downto 16) & d_out(31 downto 24);
   end if;
   end if;
  end process;
```

Figura 5.7. Descrierea proceselui final_p

În continuare am prezentat modulul de criptare pentru o rundă, modul apelat de către structura de comandă pentru criptare. În figura 5.8 este prezentat FSM-ul corespunzător, care conţine trei stări: *asteapta*, *criptare_interior* şi *criptare_final*. Din starea *asteapta*, în momentul în care se activează semnalul *cript*, la primul front crescător al tactului se trece în starea *criptare_interior*. Semnalul *cript* este activat de către structura de comandă pentru criptare, moment în care sunt furnizate blocurile

de text clar sau parțial criptate. Are loc o citire a acestor date şi a subcheilor necesare criptării din runda respectivă. Totodată sunt calculate noile valori ale regiştrilor A, B, C, D. La următorul tact se trece în starea *criptare_final*. În această stare se activează semnalul *rezolvat* şi se transmit la ieşire noile valori. Pe următorul front cescător semnalul *rezolvat* se dezactivează şi se revine în starea *asteapta*.

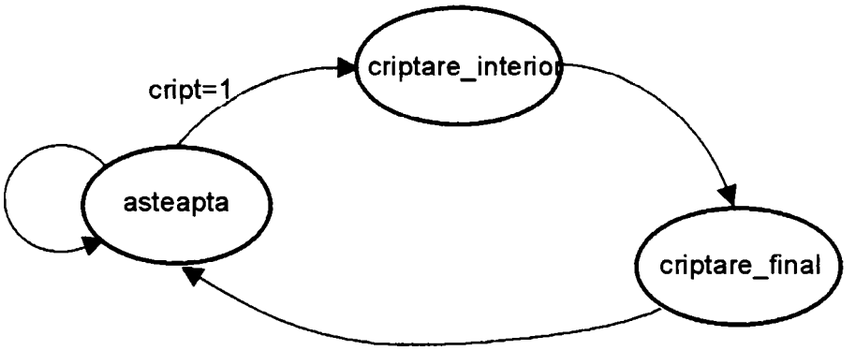


Figura 5.8. Diagrama stărilor pentru modulul de criptare pentru o rundă

În anexa 3 este prezentat codul VHDL corespunzător entității de criptare pentru o rundă. Arhitectura conține următoarele procese: *async_p, syncronous_p* – similare celor prezentate în figura 5.5, *criptare_p* – procesul care realizează calculele corespunzătoare unei runde de criptare şi procesul *rezolvat_p* care activează semnalul *rezolvat* indicând structurii de comandă pentru criptare terminarea calculelor.

În anexa 4 sunt prezentate diagramele de timp corespunzătoare criptării. Pentru a verifica faptul că implementarea realizată de mine funcționează corect am folosit pentru simulare testele din anexa algoritmului RC6 (anexa 1).

Sunt prezentate de asemenea în anexa 5 reprezentările grafice ale interiorului FPGA-ului care conține circuitul de criptare realizat, cu componentele respectiv interconexiunile dintre acestea. Reprezentările sunt date la diferite scări, ajungându-se până la nivel de poartă.

Modulul de generare al cheilor l-am proiectat conform FSM-ului corespunzător, prezentat în figura 5.9. Acesta conține patru stări: *asteapta*, *inițializare*, *gen_subchei* şi *final*. Din starea *asteapta*, la activarea semnalului *init* se trece în starea *initializare*. Are loc o inițializare a variabilelor şi a contorului *count132* şi se descompune cheia în blocuri de 32 de biți conform algoritmului de generare al cheilor din cifrul RC6. Pe următorul front crescător al tactului se realizează trecerea în starea *gen_subchei*. Se calculează subcheile şi se rămâne în această stare până când contorul ajunge la valoarea 132, conform buclei *for* din algoritm. Valoarea 132 rezultă din formula:

$$v = 3 \times max\{c, 2r+3\}$$

unde *c* reprezintă numărul de blocuri de 32 de biți ai cheii iar *r* este numărul de runde. Pentru varianta implementată de mine, *c* = 8 şi *r* = 20. Calculele fiind simple, le-am efectuat manual pentru a evita încărcarea logicii circuitului de criptare. În momentul în care *count132* ia valoarea 132 se trece în starea *final*. În această stare sunt deja calculate toate subcheile care vor fi utilizate în procesul de criptare. La următorul front ascendent se revine în starea *asteapta*.

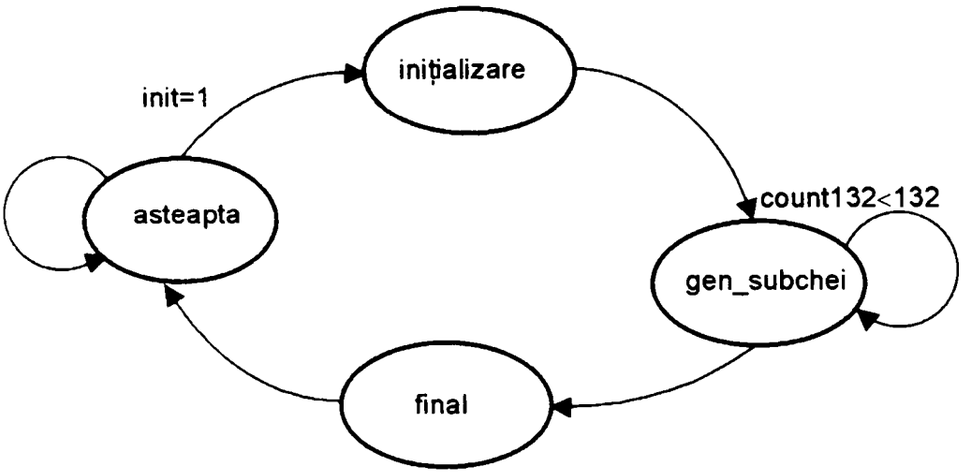


Figura 5.9. Diagrama stărilor pentru modulul de generare a cheilor

Arhitectura modulului de generare al cheilor conține următoareale procese: *syncronous_p*, *async_p* asemănătoare ca structură cu cele prezentate la modulul de criptare dar adaptate la funcționarea acestui modul, *counter_p* – gestionează contorul count132, iar *generarea_cheilor_p* – realizează toate calculele necesare generării cheilor. Pentru calculul subcheilor se pleacă de la două constante notate *p* şi *q* a căror valoare este precizată în documentația algoritmului. Calculele specificate în algoritm:

```
for s=1 to v do
      {
      A = S[i] = (S[i] + A + B) <<< 3
      B = L[j] = (L[j] + A + B) <<< (A+B)
      i = (i + 1) mod (2r + 4)
      j = (j + 1) mod c
      }
```

le-am implementat în procesul *generarea_cheilor_p*. Bucla *for* s-a implementat cu ajutorul FSM-ului şi a contorului *count132*. Astfel, până când *count132* nu ia valoarea 132 se rămâne în starea *gen_subchei*. În procesul *async_p* este realizat acest lucru prin condiționarea tranziției la starea următoare de atingerea valorii 132 de către *count132*, după cum se poate observa în continuare.

```
when gen_subchei =>
     if count132 = 132  then
          next_s <=final;
     else
          next_s <=gen_subchei;
     end if;
```

Operațiile de deplasare s-au realizat cu ajutorul unei funcții case. În funcție de valoarea cu care trebuie să se realizeze deplasarea, se concatenează cele două secvențe de biți sub forma:

```
when 7  =>
    temp:=interm2(24 downto 0) & interm2(31 downto 25);
```

unde temp şi interm2 sunt variabile interne ale procesului generarea_cheilor_p.

Codul VHDL corespunzător modulului de generare al cheilor este prezentat în anexa 6. Modulul de generare al cheilor l-am implementat separat datorită limitărilor impuse de suportul fizic, cu toate că modelul de placă utilizat este cel mai nou apărut şi cu capacitatea cea mai mare până la ora respectivă.

Modulul de generare al cheilor este comun proceselor de criptare şi decriptare, el fiind apelat doar la schimbarea cheii utilizator. În anexa 7 am prezentat diagramele de timp obținute în urma simulării procesului de generare a celor 44 de subchei, care sunt identice cu cele obținute în varianta de implementare Verilog.

Modulul de decriptare este prezentat în figura 5.10. Structura sa este asemănătoare cu cea a modulului de criptare iar modulul de generare al subcheilor este indentic cu cel din structura de criptare.

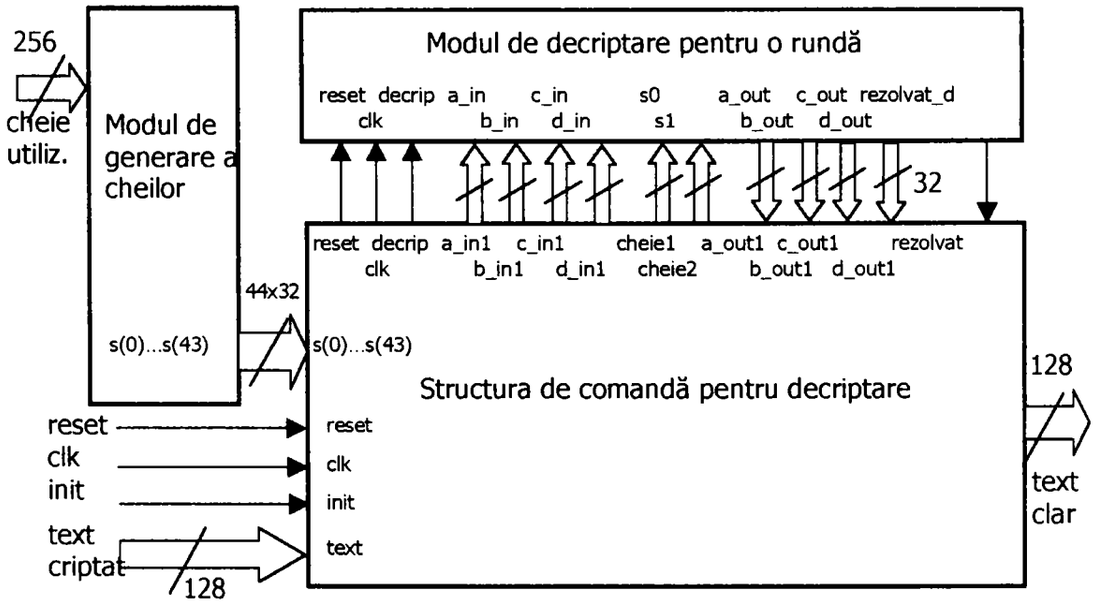


Figura 5.10. Structura detaliată a modulului de decriptare

Modulul principal pentru decriptare apelează modulul de decriptare pentru o rundă, furnizând acestuia blocurile de date şi cele două subchei corespunzătoare rundei respective concomitent cu activarea semnalului *decrip*. Subcheile sunt citite din tabel în ordine inversă faţă de cazul criptării. După ce modulul de decriptare pentru o rundă a realizat calculele necesare, semnalul *rezolvat_d* devine 1 şi rezultatele sunt transmise modulului principal. La finalul celor 20 de runde la ieşirea circuitului vom avea textul decriptat.

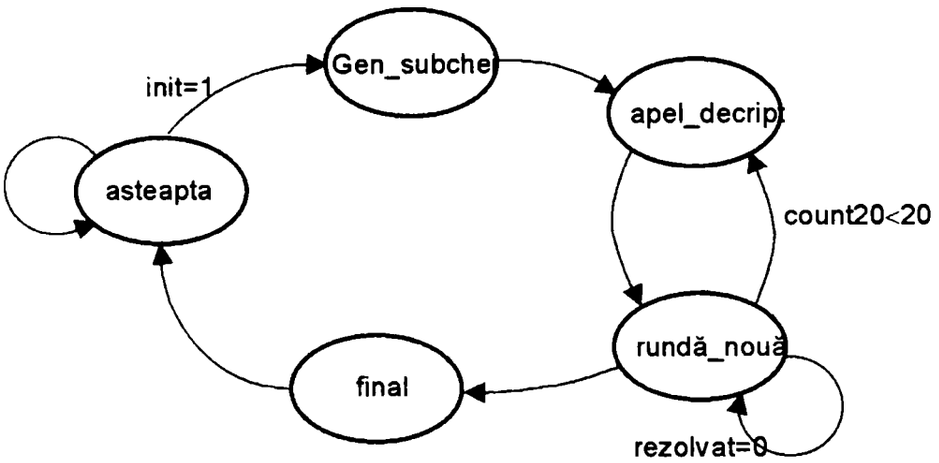


Figura 5.11. Diagrama stărilor pentru modulul principal de decriptare

În figura 5.11 este prezentată diagrama stărilor corespunzătoare structurii de comandă pentru decriptare. Se poate observa că aceasta este asemănătoare cu cea a modulului principal de criptare. Cheile sunt apelate în ordine inversă faţă de procesul de criptare. În figura 5.12 tranziţia de la starea *asteapta* la *decriptare_interior* se realizează pe frontul crescător al tactului când semnalul *decript* este activ. În această stare se parcurg operaţiile descrise în algoritmul de decriptare prezentat în capitolul 2.

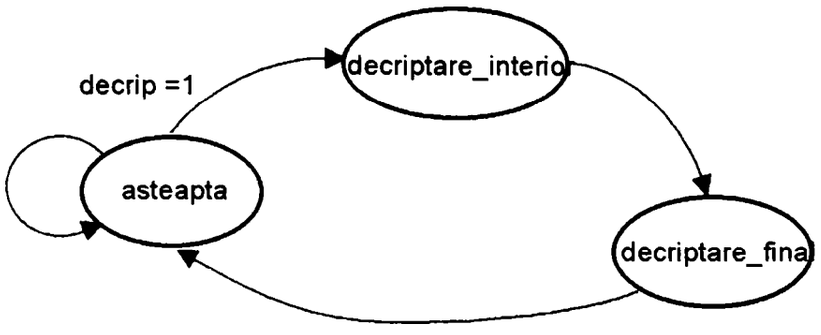


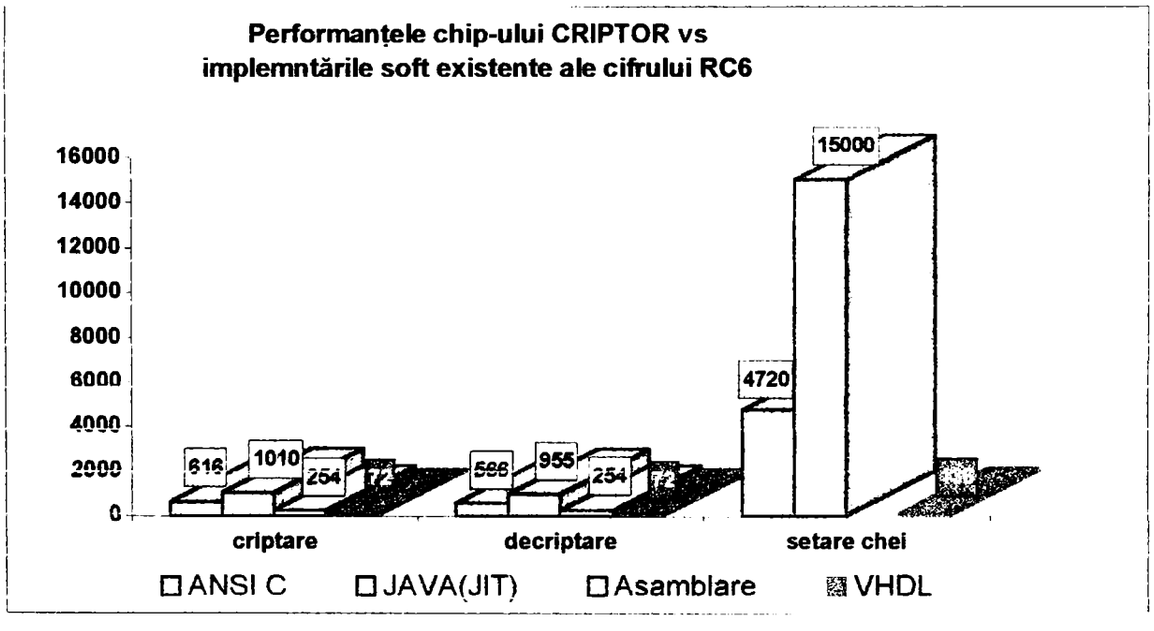
Figura 5.12. Diagrama stărilor pentru modulul de decriptare pentru o rundă

Structura celor două module prezentate mai sus este similară cu cea a modulelor din cazul criptării. Operațiile sunt complementare celor de la criptare iar subcheile sunt citite şi utilizate în ordine inversă comparativ cu situația criptării. Din acest motiv nu voi insista asupra descrierii structurii acestor module.

În anexa 9 este prezentat codul VHDL şi în anexa 10 diagramele de timp corespunzătoare. Se poate observa că în urma decriptării se obține textul clar inițial, ceea ce confirmă corectitudinea implementării pe care am realizat-o.

În anexa 12 am prezentat câteva dintre rapoartele fumizate de programul de proiectare din care rezultă numărul de porți logice echivalente necesare, timpii maximi de întârziere introduşi şi suprafața ocupată din placă exprimată în procente. Programul de proiectare solicită configurații complexe, motiv pentru care pentru realizarea circuitului de criptare/decriptare s-au folosit calculatoare cu procesor Pentium II/350MHz cu 128MB RAM.

In graficul 5.1 am evidențiat performanțele circuitului CRIPTOR prin compararea lor cu datele oferite de Laboratoarele RSA referitoare la implementările soft cunoscute. In grafic am specificat numărul de cicluri de tact necesare proceselor de criptare, decriptare, respectiv setare a cheilor. Aşa cum se poate observa din acest grafic, performanțele circuitului hardware de criptare, denumit CRIPTOR, sunt mult superioare celorlate implementări software cunoscute şi prezentate până în momentul de față în literatura de specialitate.



Graficul 5.1. Diferite implementări ale cifrului RC6 exprimate în cicluri de tact

5.3. Testarea circuitului CRIPTOR

Odată ce schema îndeplineşte cerințele, ultimul pas constă în programarea circuitului dorit. Acest lucru se face cu ajutorul utilitarelor „Device Programming", şi

anume „JTAG Programmer", „Hardware Debuger" sau „PROM File Formater". „JTAG Programmer" încarcă, reciteşte şi verifică configuraţia datelor, iar apoi poate efectua teste funcţionale şi poate verica stările logice interne ale schemei. „PROM File Formater" furnizează o interfaţă utilizator grafică care indică utilizatorului să formateze fişierele BIT într-un fişier PROM compatibil Xilinx, să concateneze toate şirurile de biţi într-un singur fişier PROM şi să memoreze diverse aplicaţii în acelaşi fişier PROM. „Hardware Debuger" este tot o interfaţă grafică care permite încărcarea schemei proiectate într-un circuit FPGA, verificarea configuraţiei obţinute şi afişarea stărilor interne ale circuitului programat.

Procesul de verificare a proiectului constă dintr-o testare funcţională şi o verificare a performanţelor schemei. Verificarea trebuie făcută încă din faza de proiectare. Foundation permite trei metode de verificare ale proiectului: simularea, analiza statică de timp şi verificarea "in-circuit". Prin simulare se poate determina dacă au fost respectate cerinţele de timp şi cele funcţionale. Simularea funcţională se poate face în timpul procesului de proiectare, imediat după ce a fost facută sinteza schemei descrisă în VHDL, iar "Timing simulation" se face în timpul procesului de implementare. Cu ajutorul utilitarului "Express Time Tracker" se pot obţine analizele post-sinteză şi pre-implementare ale proiectului HDL. Verificarea "in-circuit" reprezintă testul final în care se poate verifica modul de lucru al schemei implementate în condiţii de operare tipice.

Testarea circuitelor proiectate şi implementate pe plăciile furnizate de către firma Xess se poate face utilizând facilităţile oferite de JTAG Programmer care permite verificarea datelor de configurare, efectuarea de teste funcţionale şi verificarea stărilor logice interne. Operaţiile care pot fi efectuate de către utilizator sunt:

- Program - încarcă conţinutul fişierelor JDEC sau BIT în registrele de programare a echipamentului (device)
- Verify - reciteşte conţinutul registrelor de programare a plăcii şi le compară cu fişierele JDEC sau BIT
- Erase - Şterge informaţiile de configurare a placii
- Functional test - Aplică vectrii funcţionali specificaţi de utilizatordin fişierul JDEC în placă utilizând instrucţiunea JTAG INTEST şi compară rezultatele obţinute cu cele prestabilite. Orice diferenţă este raportată utilizatorului
- Blank Chek - Verifică dacă placa a mai fost programată sau ştearsă
- Readback Jedec - Citeşte conţinutul registrelor de programare a plăcii şi crează un nou fişier JEDEC care conţine rezultatele
- Get device ID - Citeşte conţinutul registrului JTAG IDCODE şi afişează conţinutul.
- Get device Checksum – citeşte conţinutul registrelor de programare a plăcilor şi calculează o sumă de control pentru a o compara cu valorile precalculate
- Get device Signature/Usercode – această valoare este selectată de utilizator în timpul introducerii şi este apoi transformată în valori binare în fişierul JEDEC. În timpul programării plăcii aceste valoari sunt încărcate în registrul JTAG USERCODE. Această funcţie citeşte conţinutul registrului USERCODE şi afişează rezultatul.

- Bypass – ignoră blocul atunci când se face adresarea blocurilor din canalul boundary scan JTAG.

Pentru a specifica proprietățile boundary scan ale plăcii se utilizează fişiere BSDL (Boundary-Scan Description Language) care utilizează instrucțiuni VHDL.

Având la dispoziție instrumentele oferite de Foundation Express, am efectuat testările funcționale ale circuitului de criptare ale cărei rezultate le-am precizat în anexele care conțin diagramele de timp ce confirmă buna funcționare a circuitului. Analiza statică de timp este raportată în anexa 12.

5.3.1. Facilități de testare off-line ale circuitului CRIPTOR

Pentru testarea off-line a circuitului proiectat am folosit următoarea metodologie BIST:

1. Aleg un set de subchei aleatoare.
2. Aplic la intrările circuitului un set arbitrar al unui model de intrare.
3. Memorez datele criptate de la ieşirile circuitului.
4. Compar rezultatele criptate cu modelele precalculate.
5. Aleg acelaşi set de subchei aleatoare şi aplic la intrările blocului de decriptare datele de la ieşirea blocului de criptare memorate (punctul 3).
6. Compar ieşirile blocului de decriptare cu setul arbitrar dat la intrările blocului de criptare (punctul 2).

Generarea de adevărate modele aleatoare este însă foarte dificilă. Pe de altă parte, caracterul aleator nu este tocmai de dorit deoarece răspunsul testelor ar putea fi reprodus într-o ordine care ar căpăta semnături identice. Din aceste cauze, rezultate bune ale testului pot fi obținute aplicând modelul generatorului pseudo-aleator.

După simularea extensivă a funcționării circuitului am descoperit că este suficient să aplic o cheie principală pseudo-aleatoare care este apoi post-procesată la fel ca o cheie principală externă, pentru producerea setului de subchei. Aceste subchei sunt potrivite pentru a se obține o acoperire mulțumitoare a defectelor tuturor unităților magistralei. Evaluarea informației la ieşire este făcută prin analiza de semnături.

Pentru implementarea schemei auto-testabile, am folosit o arhitectură BIST centralizată şi separată, de tipul celei descrise în capitolul 1.6.1. Am aplicat această tehnică pentru blocul de generare a cheilor şi pentru blocul de criptare. Acelaşi lucru se face apoi şi pentru blocul de decriptare.

Noul concept dezvoltat al auto-testului este tolerarea pentru acoperirea defectelor de magistrală. Acesta a fost extins la un circuit cu testare completă. Aşa cum se observă în figura 5.13, o cheie principală pseudo-aleatoare şi un set fixat de intrări pseudo-aleatoare sunt aplicate "în spatele intrării", în locul intrării externe pe

parcursul auto-testului. Analiza semnăturii este aplicată informației doar "înainte de a părăsi chip-ul". Controller-ul de autotest simulează operația normală pentru toate celelalte subblocuri hard şi alege unica unitate "conştientă" a operației de auto-test. Avantajele acestei scheme sunt multiple. Primul avantaj constă în faptul că, deşi intrarea magistralei este de 256 biți pentru cheie şi 128 de biți pentru text, intrarea portului este o magistrală doar de 32 biți.

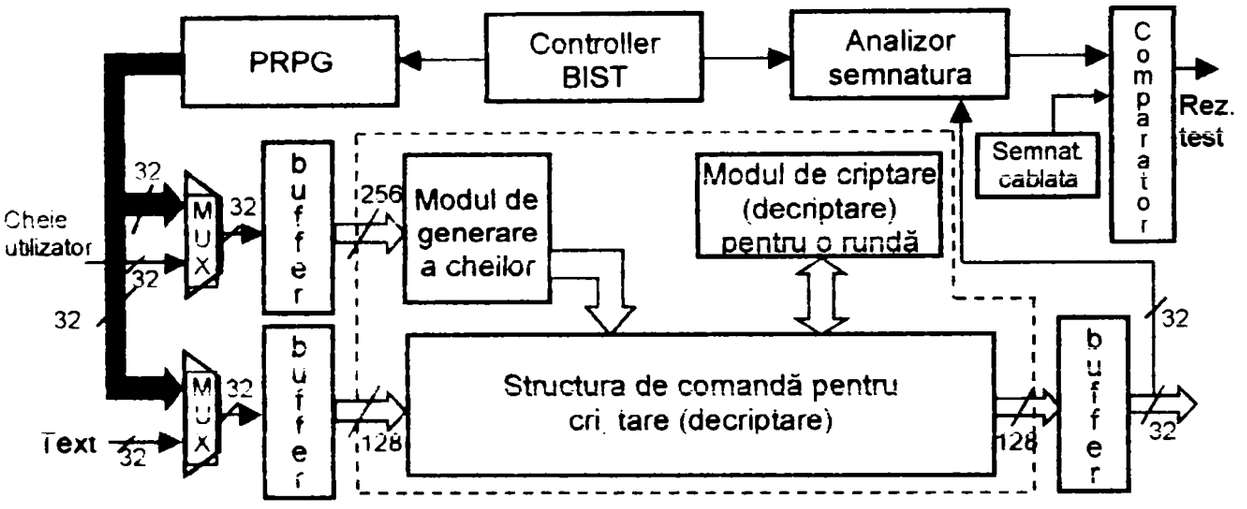


Figura 5.13. Schema BIST off-line a circuitului de criptare CRIPTOR

Al doilea avantaj al acestei soluții este că un model pseudo-aleator de lungime de 32 biți poate fi generat rapid. timpul total rezultat fiind cu puțin peste cel al generării unuia 256 biți, dar cu o mare economie de hardware. Un alt avantaj este că toate intrările, subcheile generate şi programate şi circuitele de ieşire sunt incluse în procesul de autotestare.

Schema nou obținută, în urma implementării ei în VHDL, conține un hardware suplimentar. dar aşa cum rezultă din Tabelul 5.1, creşterea nu este foarte mare.

Tabelul 5.1. Necesar hardware cu şi fără BIST

Caracteristica	Fără BIST	Cu BIST
CLB-uri	5.998	7.673
Generatoare de funcții	11.295	13.763
Bistabile	3.780	4.524
Total porți echivalente	111.663	137.249

Din tabel se observă că necesarul suplimentar este de aproximativ 20% din numărul de porți necesare pentru circuitul fără BIST.

Secvența autotestului cuprinde 4.280 cicluri ale testului generării subcheilor şi aproximativ 2 x 2.880 cicluri pentru operațiile de criptare şi decriptare, traversând astfel toate circuitele din chip. Operația de test off-line BIST durează deci aproximativ 10.040 de cicluri de tact, adică aproximativ 0,1 ms la o frecvență de tact de 100 MHz.

5.4. Variantă de implementare a algoritmului RC6 folosind Verilog

În acest paragraf prezint o variantă de implementare a algoritmului RC6 utilizând limbajul de proiectare hardware de nivel înalt, Verilog. Codul Verilog din anexa 13 a fost compilat pe un sistem PC cu un compilator Verilog sub DOS. Implementarea hardware pentru această variantă nu a mai fost realizată în cadrul acestei teze, motiv pentru care toate procesele sunt conținute într-un singur FSM.

Schema bloc a circuitului de criptare/decriptare este prezentată în figura 5.14. Semnale de intrare (INPUT) ale schemei sunt următoarele:

- *clock* – folosit pentru sincronizarea operațiilor;
- */reset* – folosit pentru a porni chip-up; de fiecare dată când este setat pe 0 aduce chip-ul în starea inițială (START);
- *enable* – dacă este "1" chip-ului i se permite să lucreze;
- *encrypt* – dacă este setat pe "1" chip-ul va realiza o criptare, altfel, o decriptare;
- *new_key* – dacă este "1" avem o nouă cheie utilizator şi trebuie să generăm o nouă tabelă pentru cheile rundei;
- *new_text* – la sfârşitul criptării sau decriptării, dacă acest semnal este setat pe "1" înseamnă că dorim să începem un nou proces de calcul;
- *text_valid* – este setat pe "1" când textul de intrare (clar sau cifrat) este disponibil şi chip-ul va citi acest text;
- *key_valid* – dacă este setat pe "1" cheia furnizată de către utilizator este disponibilă şi chip-ul va citi această cheie;
- *key_type* – intrare pe 2 biți care specifică lungimea cheii utilizator;
- *key* – intrare pe 256 de biți pentru încărcarea cheii furnizate de utilizator;
- *text_in* – intrare pe 128 de biți pentru încărcarea textului, care poate fi clar sau cifrat;

Semnale de ieşire (OUTPUT) sunt:

- *text_out* – ieşire pe 128 de biți pentru citirea rezultatului (text clar sau cifrat);
- *ready* – specifică momentul în care textul de la ieşire poate fi citit.

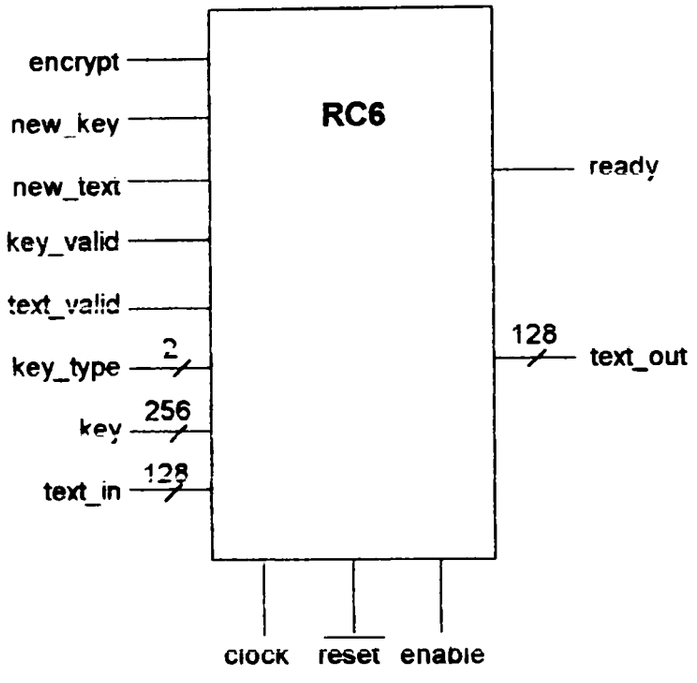


Figura 5.14. Schema bloc

Algoritmul a fost implementat şi în acest caz ca şi un automat cu stări finite cu următoarele caracteristici: K stări, S = {s_1, s_2, ..., s_K}, unde s_1 este starea inițială; N intrări, I = {i_1, i_2, ... , i_N}; M ieşiri, O = {o_1, o_2, ... , o_M}; funcția de tranziție, notată T(S,I) şi care mapează fiecare stare curentă şi intrare câte unei stări următoare; funcția de ieşire O(S) care mapează fiecare stare curentă câte unei ieşiri.

Starea curentă a maşinii este păstrată în memoria stării, un set de bistabile comandate de un singur semnal de tact (de aici "automat sincron"). Vectorul de stare (starea curentă sau doar starea) este valoarea curentă păstrată în memoria stării. Starea următoare a automatului este funcție de vectorul de stare şi de intrări.

Dându-se o secvență de intrări, automatul produce o secvență de ieşiri care depind de starea curentă, funcția de tranziție şi funcția de ieşire.

Există câteva reguli generale care trebuiesc respectate la implementarea unui automat cu stări finite atunci când se foloseşte un limbaj de descriere hardware: se utilizează numai un automat per modul; logica externă este păstrată la minimum; inițierea bistabilelor de stare se face separat de logică.

Automatul cu stări finite este prezentat în figura 5.15. Se pot observa cele trei părți principale ale algoritmului de criptare: una pentru generarea cheilor (stările SK0, SK1, SK2, SK3, SK4), una pentru criptare (stările SE0, SE1, SE2, SE3, SE4) şi una pentru decriptare (stările SD0, SD1, SD2, SD3, SD4). Am mai introdus două stări, denumite START şi FINISH.

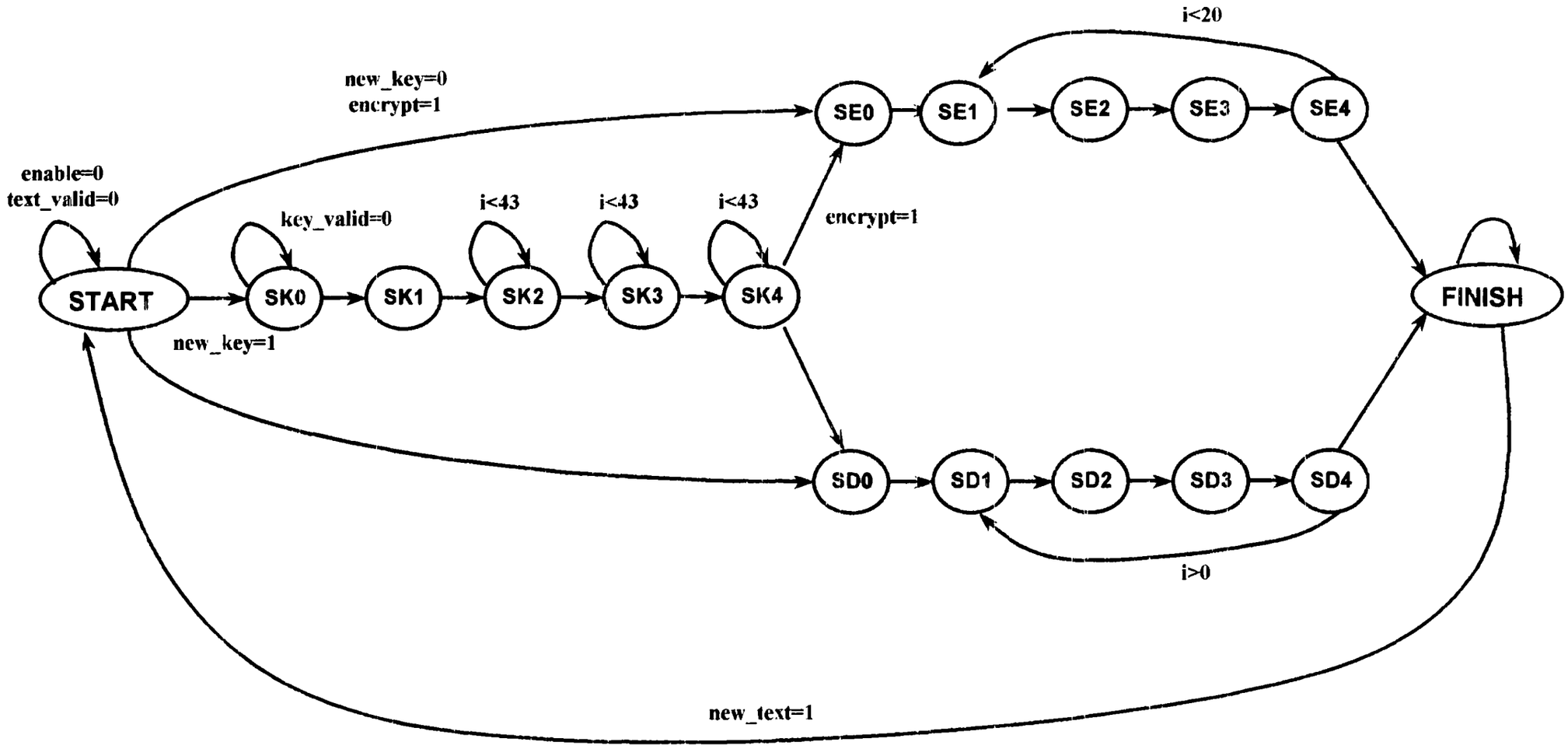


Figura 5.15. Automatul cu stări finite

Pentru codificarea stărilor s-a folosit tehnica *one-hot encoding*, unde numai un singur bit al vectorului de stare este atribuit unei anumite stări, toți ceilalți biți fiind zero. Deci, pentru *n* stări vor fi necesare *n* bistabile de stare.

Structura verifică fiecare bit de stare, pe rând, până când găseşte unul care este setat. Apoi, un bit al stării următoare este setat corespunzător tranziției de stare potrivite. Restul biților stării următoare sunt toți setați pe zero. Decodificarea stării este simplificată deoarece biții de stare înşişi pot fi folosiți direct pentru a indica dacă automatul este într-o stare particulară. Nu este necesară nici o logică adițională.

Utilizarea tehnicii *one-hot* oferă numeroase avantaje. Automatele *one-hot* sunt de obicei mai rapide, viteza fiind independentă de numărul de stări şi depinzând doar de numărul de tranziții într-o stare anume. Un alt avantaj îl reprezintă faptul că nu trebuiesc aplicate regulile de determinare a codificării optime. Acest lucru este benefic atunci când structura maşinii de stare se modifică, atunci când se adaugă noi stări sau altele se modifică. Tehnica *one-hot* este la fel de "optimă" pentru toate automatele. Automatele *one-hot* sunt uşor de proiectat. Codul HDL (Hardware Description Language) poate fi scris direct din diagrama de stare fără a codifica un tabel al stărilor. Modificările sunt uşor de făcut. Adăugarea sau ştergerea de stări, schimbarea ecuațiilor de excitație pot fi uşor realizate fără a afecta restul automatului. Un alt avantaj important este sinteza uşoară realizabilă în Verilog sau VHDL. Pe de altă parte căile critice sunt uşor de găsit folosind analiza statică a timpului şi automatele sunt uşor de depanat, problemele din stările de tranziție fiind evidente.

Automatul cu stări finite lucrează în felul următor: când chip-ul este inițializat (*reset* = 0), vectorul de stare (*state*) de 17 biți este setat pe 0 şi automatul este forțat să treacă în starea START. În caz contrar în vectorul de stare este încărcat vectorul stării următoare (*next_state*). Bistabilul *flag* este folosit pentru a indica dacă maşina trebuie să rămână în aceeaşi stare.

Stările FSM-ului care descrie algoritmul de criptare precum şi succesiunea acestora sunt enumerate mai jos:

1) START

În această stare sunt citite datele de intrare. Algoritmul RC6 lucrează cu patru regiştri a câte 32 de biți: A, B, C şi D, care conțin atât textul clar de intrare cât şi textul cifrat la sfârşitul unei criptări. Primul octet al textului clar sau al textului cifrat de la intrare este plasat în octetul cel mai puțin semnificativ a lui A. Ultimul octet al textului clar sau cifrat este plasat în octetul cel mai semnificativ al lui D.

Dacă ambele semnale *enable* şi *text_valid* sunt "1", regiştrii A, B, C, D sunt încărcați cu date de pe intrarea de 128 de biți, *text_in*. Datorită modului *little-endian*, acest lucru se face în modul următor: în A[31:24] încărcăm text_in[24:31], în A[23:16] text_in[16:23], în A[15:8] text_in[8:15], în A[7:0] text_in[0:7], în B[31:24] text_in[56:63], ..., şi în D[7:0] încărcăm text_in[96:103].

Dacă *new_key* este "1" atunci începe generarea cheilor, următoarea stare fiind SK0. Altfel, dacă *encrypt* este "1" următoarea stare este SE0, marcând începutul criptării, altfel următoarea stare este SD0 şi determină începutul procesului de decriptare.

Dacă fie *enable* fie *text_valid* sunt "0", chip-ul va rămâne în această stare.

2) SK0

SK0 este prima stare a procesului de generarea al cheilor.

Dacă semnalul de intrare *key_valid* este "1" cheia utilizator este disponibilă la intrare şi ea este păstrată în registrul L care are o lungime de 256 de biți. Acest lucru este realizat în acelaşi mod ca şi la încărcarea textului pentru criptare sau decriptare, prezentat anterior.

Pe baza valorii semnalului *key_type* chip-ul ştie dacă lungimea cheii utilizator este de 128, 192 sau 256 de biți (*key_type* este 00, 01 şi respectiv 10 sau 11). Această lungime, în octeți, este păstrată în registrul *c*.

Dacă *key_valid* este "0", automatul rămâne în această stare până când cheia furnizată va fi disponibilă.

3) SK1, SK2, SK3, SK4

Aceste patru stări implementează procesul de generare al cheilor. Pornind de la cheia utilizator sunt calculate cele 44 de subchei, fiecare având o lungime de 32 de biți. Aceste subchei sunt păstrate într-o tabelă de 44x32 biți, numită S.

Odată ce subcheile sunt calculate şi păstrate în tabelele S, acestea pot fi folosite şi pentru următoarele blocuri de date de la intrare. Criptarea sau decriptarea primului bloc de 128 de biți va dura mai mult datorită procesului de generare al subcheilor. Pentru celelalte blocuri timpul necesar va fi mai scurt deoarece subcheile sunt deja calculate.

SK4 este ultima stare a procesului de generare al subcheilor, stare în care se obțin valorile finale ale subcheilor. După cum am văzut, chip-ul poate realiza atât criptarea cât şi decriptarea unui text. Acest lucru este determinat de valoarea semnalului *encrypt*. Dacă *encrypt* este setat pe "1" atunci va avea loc o criptare şi următoarea stare va fi SE0, altfel va avea loc o decriptare şi următoarea stare va fi SD0.

4) SE0, SE1, SE2, SE3, SE4

Aceste stări sunt folosite pentru criptare. În prima stare, valorilor din regiştri B şi D le sunt adunate S[0] şi respectiv S[1] - primele două subchei -, iar contorul i este setat pe 1.

SE1, SE2, SE3 şi SE4 implementează partea principală a criptării, care este bucla *for* din procedura de criptare. Am decis de a diviza această buclă în patru stări cu scopul de a implementa asignarea paralelă (A, B, C, D) = (B, C, D, A). Un motiv pentru a face acest lucru este economia de timp şi de spațiu – un registru de 32 de biți care ar fi fost necesar pentru implementarea acestei asignări.

SE1, SE2, SE3 şi SE4 implementează patru paşi ai buclei. Pentru a atinge cele 20 de runde indicate de către proiectanți, cele patru stări sunt repetate de cinci

ori. Acest lucru este controlat de către contorul *i*. Când cele 20 de runde sunt complete şi criptarea este terminată, următoarea stare este FINISH.

5) SD0, SD1, SD2, SD3, SD4

Aceste stări implementează decriptarea. Prima stare este similară cu SE0, dar în loc de B şi D se lucrează cu C şi A.

La fel ca şi la criptare, stările SD1, SD2, SD3, SD4 implementează partea principală a decriptării. Motivul pentru care am divizat bucla *for* în patru este acelaşi şi din nou aceste stări vor fi repetate de cinci ori pentru a obţine cele 20 de runde ale decriptării.

Diferenţa dintre procesele de criptare şi de decriptare constă în faptul că la decriptare ordinea operaţiilor este inversată şi calculele vor începe cu subcheile cu indicii cei mai mari.

După terminarea celor 20 de runde, se mai fac două operaţii asupra regiştrilor D şi B şi decriptarea este gata. Următoarea stare este FINISH.

6) FINISH

Aceasta este ultima stare a automatului finit. Ieşirea *text_out* este asignată cu valorile concatenate ale regiştrilor A, B, C, D. Acest lucru este făcut în acelaşi mod cu încărcarea blocului de intrare - modul little-endian.

Semnalul de ieşire *ready* este setat pe 1 pentru a indica faptul că decriptarea sau criptarea este terminată şi utilizatorul poate să citească textul de ieşire.

Dacă semnalul *new_text* este setat pe "1" înseamnă că chip-ul trebuie să realizeze o criptare sau o decriptare a unui alt bloc de 128 de biţi, astfel că următoarea stare va fi START.

Programul Verilog complet de implementare hardware a cifrului RC6 este dat în anexa 11, iar în anexa 12 este prezentată varianta software de implementarea a algoritmului în limbajul C, care confirmă de asemenea faptul că operaţiile de calcul al cheilor, criptarea şi decriptarea sunt corecte.

Implementarea algoritmului RC6 în versiunea prezentată în acest paragraf este rezultatul colaborării mele cu colectivul Universităţii Catolice din Leuven, Belgia, colectiv condus de prof. dr. ing. Bart Prenel, în cadrul unui program Tempus. Tot în cadrul Universităţii Catolice din Leuven funcţionează şi colectivul condus de prof. dr. ing. Vincent Rijmen, unul dintre autorii cifrului Rijndael, candidat AES.

Această implementare a fost prezentată şi laboratoarelor RSA care au confirmat faptul că până în vara anului 1999 nu era cunoscută o altă implementare hardware a acestui cifru.

5.5. Concluzii

În capitolul 5 prezint rezultatele obținute la implementarea algoritmului de criptare RC6. Am realizat implementarea algoritmului în trei variante, una software în limbajul C şi două versiuni de implementarea hardware realizate cu ajutorul limbajelor de descriere hardware de nivel înalt – VHDL şi Verilog. Ultimele două implementări le-am prezentat în detaliu în paragrafele 5.2 respectiv 5.3. Implementarea versiunii VHDL am realizat-o fizic pe plăcile XCV1000 din familia VIRTEX, furnizate de firma Xilinx Inc. Rezultatele experimentale pe care le-am obținut le-am prezentat sub forma unor diagrame de timp şi al unor rapoarte oferite de programul Foundation Expres, versiunea 1.5.

Contribuțiile pe care le-am adus prin realizarea implementărilor acestui algoritm de criptare se pot rezuma în următoarele puncte:

1. Am realizat prima implementare hardware a algoritmului RC6. Aceasta am efectuat-o pe noile plăci din familia Virtex, apărute în toamna anului 1999, utilizând programul de proiectare Foundation Expres 1.5 al firmei Xilinx Inc.
2. Am implementat circuitul de criptare utilizând limbaje de descriere hardware de nivel înalt, limbaje care oferă o serie de avantaje, precum: funcționarea schemei poate fi verificată rapid în procesul de proiectare şi se poate face apoi imediat simularea schemei descrise ceea ce permite testarea soluțiilor arhitecturale şi de proiectare.
3. Implementarea algoritmului am realizat-o pe baza unor FSM-uri care descriu succesiunea stărilor din cifrul bloc RC6.
4. Pentru scurtarea timpului necesar criptării respectiv decriptării precum şi pentru reducerea circuitului parcurgerea celor 20 de runde necesare obținerii unei criptări sigure am realizat-o în cadrul aceluiaşi modul.
5. Utilizarea tehnicii *one-hot* de codificare care oferă numeroase avantaje privind viteza de codificare care este independentă de numărul de stări şi depinde doar de numărul de tranziții într-o stare anume.
6. Partea principală a criptării, reprezentată de bucla *for* din procedura de criptare am divizat-o în patru stări realizând în acest mod economie de timp şi de spațiu. Pentru a atinge cele 20 de runde indicate de către proiectanți, cele patru stări sunt repetate de cinci ori.
7. Prezentarea realizată prin intermediul FSM-urilor, în paralel cu modul de implementare al acestora în VHDL şi Verilog reprezintă o abordare originală şi detaliată a proceselor care au loc în timpul criptării, nerealizată până în momentul de față.
8. Prezentarea unei scheme originale de implementarea hardware cu facilități BIST a algoritmului RC6, schemă care prezintă o serie de avantaje:
    a. deşi intrarea magistralei este de 256 biți pentru cheie şi 128 de biți pentru text, intrarea portului este o magistrală doar de 32 biți,

b. un model pseudo-aleator de lungime de 32 biți poate fi generat rapid, timpul total rezultat fiind cu puțin peste cel al generării unuia 256 biți, dar cu o mare economie de hardware,

c. toate intrările, subcheile generate şi programate şi circuitele de ieşire sunt incluse în procesul de autotestare,

d. necesarul suplimentar de hardware este de aproximativ 20% din numărul de porți necesare pentru circuitul fără BIST,

e. operația de test off-line BIST durează aproximativ 10.040 de cicluri de tact, adică aproximativ 0,1 ms la o frecvență de tact de 100 MHz.

9. Am evidențiat în graficul 5.1 performanțele superioare ale criptochip-ului realizat de mine, şi denumit CRIPTOR, prin compararea acestora cu performanțele implementărilor software existente.

CAPITOLUL 6

CONCLUZII FINALE

Prin tema aleasă, "Contribuții la elaborarea unor structuri pentru securizarea informației", prezenta teză de dizertație aparține domeniului fiabilității şi testării echipamentelor de calcul.

Scopul urmărit, declarat în capitolul 0 al tezei, a fost realizarea unui circuit de criptare prin care să se implementeze un algoritm de criptare eficient şi care să fie dotat cu facilități de testare. Din această cauză, în prezenta teză am făcut inițial o analiză a metodelor de creştere a testabilității circuitelor hardware, analiză necesară pentru folosirea unor astfel de metode la proiectarea pentru testabilitate a unui circuit de criptare.

Pentru alegerea algoritmului de criptare, am făcut în capitolul 2, paragraful 2.1, o analiză a celor mai noi algoritmi de criptare, algoritmi candidați la noul standard de criptare. În urma acestei analize am ales pentru implementare algoritmul RC6, alegere care s-a dovedit foarte inspirată deoarece acest algoritm a rămas în continuare în cursa pentru standardizare, fiind cotat şi cu cele mai mari şanse. Tot în capitolul 2 am prezentat algoritmul RC6 şi am făcut o analiză minuțioasă privind atacurile asupra acestui algoritm. În acest scop am abordat probeme de criptanaliză diferențială, criptanaliză liniară şi de securitate a listelor de chei.

Următorul capitol, şi anume capitolul 3, a fost dedicat analizei facilităților de autotestare ale algoritmului RC6. În acest scop am făcut o analiză de oportunitate pentru aplicarea tehnicilor BIST şi am prezentat caracteristicile cele mai importante ale cifrului RC6 în scopul implementării acestuia într-un circuit de criptare hardware cu facilități de autotestare.

Cu ajutorul informațiilor analizate şi prezentate în prima parte a tezei, am trecut apoi la proiectarea unui circuit care implementează algoritmul RC6, circuit pe care l-am denumit CRIPTOR. Acest criptochip reprezintă prima implementare hardware a algoritmului RC6 şi este un circuit prevăzut cu facilități de autotestare. În activitatea

de proiectare şi testare am folosit un mediu experimental configurat de mine, mediu care utilizează circuite FPGA şi tehnici de proiectare VLSI. Toate programele de proiectare şi datele rezultate din testarea circuitului, implementat în mai multe variante, sunt prezentate în cele 14 anexe ale lucrării.

Ca urmare consider că obiectivele propuse au fost îndeplinite. Contribuţiile personale din teza de dizertaţie au fost expuse în extenso prin câte un paragraf distinct la sfârşitul majorităţii capitolelor. În consecinţă, în acest capitol, ele nu vor mai fi înşiruite preferându-se doar o sinteză a lor. Astfel, teza cuprinde:

a. **Contribuţii la clarificarea metodelor de creştere a testabilităţii şi de tolerare a defectelor la circuitele dedicate criptării datelor.**

b. **Contribuţii la analiza comparativă a celor 15 algoritmi candidaţi la standardizare.**

   În acest scop am evidenţiat cu ajutorul tabelelor şi a graficelor întocmite de mine principalii parametrii care trebuiesc îndepliniţi de cei 15 candidaţi, conform cerinţelor impuse de către NIST, şi am selectat cei mai eficienţi algoritmi, algoritmi care de altfel au şi fost promovaţi în runda a doua de evaluare.

c. **O analiză originală a celor mai importante tipuri de atacuri criptanalitice asupra cifrului bloc RC6.**

   Pentru aceasta am enunţat şi am demonstrat în paragraful 2.6.1 un număr de nouă leme care au stat la baza analizei atacurilor diferenţiale.

d. **Contribuţii la evidenţierea clară a avantajelor algoritmului RC6 faţă de ceilalţi candidaţi AES.**

   Astfel, în paragrafele 2.6.2 - 2.6.4 am evidenţiat printr-o criptanaliză diferenţială amănunţită, prin criptanaliză liniară şi prin studiul securităţii listelor de chei avantajele cifrului RC6 faţă de ceilalţi candidaţi AES. În urma studiului făcut pot afirma că cifrul bloc RC6 este unul dintre cei mai puternici candidaţi la noul standard de criptare, el remarcându-se prin simplitate, flexibilitate, uşurinţa implementării şi rapiditate. Toate acestea m-au determinat să aleg cifrul RC6 pentru realizarea criptochip-ului.

e. **Contribuţii privind analiza de oportunitate pentru aplicarea tehnicilor BIST în criptochip-uri.**

   În acest sens am făcut în paragrafele 3.1.1 şi 3.1.2 o sistematizare a proprietăţilor necesare stimulilor pseudoaleatori de testare şi pentru propagarea modelelor aleatoare în sistemele de cifrare, am enunţat un număr de cinci definiţii şi am prezentat un algoritm (paragraful 3.1.3), toate fiind necesare pentru definirea unui graf potrivit pentru a descrie diagrama fluxului de date a unui cifru, respectiv pentru construirea listei predecesorilor. Am enunţat apoi, în paragraful 3.3 un număr de cinci teoreme originale şi un corolar, necesare pentru întocmirea listei predecesorilor şi demonstrarea caracterului aleator al cifrului RC6. În scopul realizării unei arhitecturi cât mai restrânse, în paragraful 3.2 am întocmit graful fluxului de date a cifrului RC6 şi diagrama fluxului de date

a arhitecturii cifrului. Am determinat de asemenea structura care transformă partea corespunzătoare a cifrului RC6 într-un cifru de involuție în scopul folosirii acesteia pentru autotestul concurent (paragraful 3.4.2.1).

f. **Prezentarea în detaliu a procesului de proiectare cu circuite VLSI utilizând sistemul Foundation Express.**

Am descris pe baza a numeroase diagrame toate etapele de proiectare care se parcurg şi procesele de transformare care au loc, începând de la etapa de introducere a proiectului până la verificarea acestuia. Am prezentat de asemenea structura internă a circuitelor FPGA şi facilitățile de testare „boundary scan" ale circuitelor din familia Virtex

g. **Contribuții la implementarea unui circuit de criptare folosind limbaje de descriere hardware.**

Am prezentat în capitolul 5 două implementări ale cifrului RC6, una realizată în VHDL, şi care a stat la baza criptochip-ului realizat de mine şi denumit CRIPTOR, şi cea de a doua în limbajul Verilog. Această a doua implementare este rezultatul colaborării mele, în cadrul unui program Tempus, cu unul dintre colectivele de cercetare în domeniul securității datelor a Universității Catolice din Leuven - Belgia, colectiv condus de prof. dr. ing. Bart Prenel.

h. **Descrierea originală prin FSM-uri a algoritmului de criptare RC6.**

Implementarea algoritmului am făcut-o pe baza unor FSM-uri care descriu succesiunea stărilor din cifrul bloc RC6. Acestea sunt prezentate în paragraful 5.2. Prezentarea prin intermediul FSM-urilor, în paralel cu modul de implementare al acestora în VHDL şi Verilog (paragraful 5.4) reprezintă o abordare originală şi detaliată a proceselor care au loc în timpul criptării, nerealizată până în momentul de față.

i. **Descrierea detaliată a etapelor de proiectare şi de testare ale circuitului CRIPTOR.**

Am realizat prima implementare hardware a algoritmului RC6. Am implementat circuitul de criptare utilizând limbaje de descriere hardware de nivel înalt, limbaje care oferă o serie de avantaje, aşa cum am prezentat în paragrafele 5.1 şi 5.2. Pentru scurtarea timpului necesar criptării, respectiv decriptării, precum şi pentru reducerea circuitului parcurgerea celor 20 de runde necesare obținerii unei criptări sigure am realizat-o în cadrul aceluiaşi modul. Am utilizat tehnici de codificare *one-hot* care oferă numeroase avantaje privind viteza de codificare - este independentă de numărul de stări şi depinde doar de numărul de tranziții într-o stare anume. Partea principală a criptării, reprezentată de bucla *for* din procedura de criptare am divizat-o în patru stări realizând în acest mod economie de timp şi de spațiu.

j. Contribuţii originale la implementarea hardware cu facilităţi BIST a algoritmului RC6.

În paragraful 5.3.1 am prezentat o schemă originală de implementare hardware cu facilităţi BIST a algoritmului de criptare RC6, schemă care prezintă o serie de avantaje:

- deşi intrarea magistralei este de 256 biţi pentru cheie şi 128 de biţi pentru text, intrarea portului este o magistrală doar de 32 biţi,
- un model pseudo-aleator de lungime de 32 biţi poate fi generat rapid, timpul total rezultat fiind cu puţin peste cel al generării unuia 256 biţi, dar cu o mare economie de hardware,
- toate intrările, subcheile generate şi programate şi circuitele de ieşire sunt incluse în procesul de autotestare,
- necesarul suplimentar de hardware este de aproximativ 20% din numărul de porţi necesare pentru circuitul fără BIST,
- operaţia de test off-line BIST durează aproximativ 10.040 de cicluri de tact, adică aproximativ 0,1 ms la o frecvenţă de tact de 100 MHz.

k. Evidenţierea performanţelor superioare ale circuitului CRIPTOR.

În urma comparaţiei performanţelor circuitului CRIPTOR cu cele ale implementărilor software existente şi prezentate până în momentul de faţă în literatura de specialitate, aşa cum se poate observa şi în graficul 5.1, performanţele criptochip-ului realizat de mine sunt mult superioare.

Importanţa domeniului şi rezultatele experimentale obţinute în urma realizării circuitului CRIPTOR, mă îndreptăţesc să afirm că acest proiect trebuie continuat prin trecerea de la circuite FPGA la realizarea unui circuit specializat care să poată fi produs pe scară largă, la un preţ scăzut.

Anexa 1

Vectorii de test pentru criptarea cu RC6

Text clar	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
Cheia utilizator	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
Text cifrat	8f	c3	a5	36	56	b1	f7	78	c1	29	df	4e	98	48	a4	1e
Text clar	02	13	24	35	46	57	68	79	8a	9b	ac	bd	ce	df	e0	f1
Cheia utilizator	01	23	45	67	89	Ab	cd	ef	01	12	23	34	45	56	67	78
Text cifrat	52	4e	19	2f	47	15	c6	23	1f	51	f6	36	7e	a4	3f	18
Text clar	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
Cheia utilizator	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
					00	00	00	00	00	00	00	00				
Text cifrat	6c	d6	1b	cb	19	0b	30	38	4e	8a	3f	16	86	90	ae	82
Text clar	02	13	24	35	46	57	68	79	8a	9b	ac	bd	ce	df	e0	f1
Cheia utilizator	01	23	45	67	89	ab	cd	ef	01	12	23	34	45	56	67	78
					89	9a	ab	bc	cd	de	ef	f0				
Text cifrat	68	83	29	d0	19	e5	05	04	1e	52	e9	2a	f9	52	91	d4
Text clar	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
Cheia utilizator	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
Text cifrat	8f	5f	bd	05	10	d1	5f	a8	93	fa	3f	da	6e	85	7e	c2
Text clar	02	13	24	35	46	57	68	79	8a	9b	ac	bd	ce	df	e0	f1
Cheia utilizator	01	23	45	67	89	ab	cd	ef	01	12	23	34	45	56	67	78
	89	9a	ab	bc	cd	de	ef	f0	10	32	54	76	98	ba	dc	fe
Text cifrat	c8	24	18	16	f0	d7	e4	89	20	ad	16	a1	67	4e	5d	48

Anexa 2

Implementarea hardware în VHDL a modulului de criptare din algoritmul RC6

```
--entitate care realizeaza criptarea utilizand entitatea de criptare pentru o runda  m_cript

    library ieee;
    use ieee.std_logic_1164.all;
    use ieee.std_logic_arith.all;
    use ieee.std_logic_unsigned.all;

    entity criptare is
       port (
            reset:          in std_logic;
            clk:            in std_logic;
            init:           in std_logic;
            text:           inout std_logic_vector (127 downto 0)
           );
    end criptare;

    architecture criptare_arch of criptare is
      type states is (asteapta, gen_subchei, apel_criptare, runda_noua, final);
      signal current_s, next_s:     states;
      signal count20 :              integer range 0 to 20;
      type tabel is array (43 downto 0) of std_logic_vector(31 downto 0);
      signal s  :                   tabel;

--semnale pentru comunicarea cu entitatea de criptare pentru o runda m_cript

      signal crip:                                  std_logic;
      signal a_in1,  b_in1, c_in1, d_in1:           std_logic_vector(31 downto 0);
      signal cheie1, cheie2:                        std_logic_vector(31 downto 0);
      signal a_out1, b_out1, c_out1, d_out1:        std_logic_vector(31 downto 0);
      signal rezolvat   :                           std_logic;

      component m_cript
        port (
--inputs
            reset:                  in std_logic;
            clk:                    in std_logic;
            cript:                  in std_logic;
            a_in:                   in std_logic_vector(31 downto 0);
            b_in:                   in std_logic_vector(31 downto 0);
            c_in:                   in std_logic_vector(31 downto 0);
            d_in:                   in std_logic_vector(31 downto 0);
            s0:                     in std_logic_vector(31 downto 0);
            s1:                     in std_logic_vector(31 downto 0);
--outputs
            a_out:                  out std_logic_vector(31 downto 0);
            b_out:                  out std_logic_vector(31 downto 0);
            c_out:                  out std_logic_vector(31 downto 0);
            d_out:                  out std_logic_vector(31 downto 0);
            rezolvat:               out std_logic
            );
       end component;
```

```
   begin
--realizarea interfatarii cu entitatea m_cript
      u1:      m_cript
      port map (reset, clk, crip, a_in1, b_in1, c_in1, d_in1, cheie1, cheie2, a_out1,
                b_out1, c_out1, d_out1, rezolvat);

      syncronous_p: process(reset, clk, next_s)
      begin
        if reset='1' then
          current_s<=asteapta;
        elsif (clk'event and clk='1') then
          current_s<= next_s;
        end if;
      end process;

      async_p: process (current_s, init, count20, rezolvat, clk)
      begin
        case current_s is
            when asteapta =>
                if init='1' then
                    next_s <=gen_subchei;
                else
                     next_s <=asteapta;
                end if;
            when gen_subchei =>
                     next_s <=apel_criptare;
            when apel_criptare =>
                     next_s <=runda_noua;
            when runda_noua =>
                if rezolvat='1' then
                          if count20 = 20  then
                              next_s <=final;
                          else
                              next_s <=apel_criptare;
                          end if;
                else
                      next_s<=runda_noua;
                end if;
           when final =>
                          next_s <=asteapta;
        end case;
      end process;

      generarea_cheilor_p:process(next_s,  clk)
      begin
          if (clk'event and clk='1') then
          if next_s=gen_subchei then
                s(0)<="01100101111000000001101100011011";
                s(1)<="10110101100000000010000000101110";
                s(2)<="01011000111011011001100000001110";
```

```
                s(3)<="00000011111111011110010101001111";
                s(4)<="01100000111000011110101011111001";
                s(5)<="11110010100000000010011110000000";
                s(6)<="00001011111011010111101111011111";
                s(7)<="10110111011111001110000101101001";
                s(8)<="11101011001111000110010101101001";
                s(9)<="10001100111011101011100001110100";
                s(10)<="11101011000011110011000010101000";
                s(11)<="10011101011101011011001110101111";
                s(12)<="10100110100111001011111111011000";
                s(13)<="10111111000011000110110011111010";
                s(14)<="01000111111111111101110011000101";
                s(15)<="00111001000100100010000001011010";
                s(16)<="00000010010001011000010011011000";
                s(17)<="11001101100110010101100100111001";
                s(18)<="01010011011010000110101001101010";
                s(19)<="11010000001111111100110111001001";
                s(20)<="00100011000011101111010100010011";
                s(21)<="10010001000111111110110000100110";
                s(22)<="01110111001100100111001110101011";
                s(23)<="11001110100001000101011110101100";
                s(24)<="11001100011101011110001011010011";
                s(25)<="01000010011100000001101011101101";
                s(26)<="10001100010010010100001000110011";
                s(27)<="00100110111010000110110001111010";
                s(28)<="00011101011010001011110111011000";
                s(29)<="00111000011011011000011010110011";
                s(30)<="10101011000000100111000101111001";
                s(31)<="10100010001011100000000000010001";
                s(32)<="00101011000100101000011000100101";
                s(33)<="10001111000000000111110010101110";
                s(34)<="10000011110111110110100101110010";
                s(35)<="00100001010101111001100111100100";
                s(36)<="10000010110011111010011110010100";
                s(37)<="00100110110011001100111111111111";
                s(38)<="01010010010110101110111110011101";
                s(39)<="10000100011101001001011110101100";
                s(40)<="00100011010100100110001001001001";
                s(41)<="11111110011100110001101110101000";
                s(42)<="00111011100010100100101110100111";
                s(43)<="01011101010111111001101111010110";
--subchei generate de modului gen_chei
        end if;
        end if;
    end process;

    criptare_p: process (clk, current_s, next_s, s, text, a_out1, b_out1, c_out1,
                         d_out1, count20)
    variable a_in:          std_logic_vector(31 downto 0);
    variable b_in:          std_logic_vector(31 downto 0);
    variable c_in:          std_logic_vector(31 downto 0);
```

```
variable d_in:          std_logic_vector(31 downto 0);
variable c0, c1:        integer range 4 to 43;
begin
  if clk'event and clk='1' then
        if current_s=gen_subchei and next_s=apel_criptare then
              a_in:=text(103 downto 96) & text(111 downto 104)
                    & text(119 downto 112) & text(127 downto 120);
              b_in:=text(71 downto 64) & text(79 downto 72)
                    & text (87 downto 80) & text(95 downto 88);
              c_in:=text(39 downto 32) & text(47 downto 40)
                    & text (55 downto 48) & text(63 downto 56);
              d_in:=text(7 downto 0) & text(15 downto 8)
                    & text (23 downto 16) & text(31 downto 24);
              a_in1<=a_in;
              b_in1<=b_in+s(0);
              c_in1<=c_in;
              d_in1<=d_in+s(1);
              cheie1<=s(2);
              cheie2<=s(3);
        elsif next_s=apel_criptare then
              c0:=2*count20+2;
              c1:=c0+1;
              a_in1<=a_out1;
              b_in1<=b_out1;
              c_in1<=c_out1;
              d_in1<=d_out1;
              cheie1<=s(c0);
              cheie2<=s(c1);
        end if;
  end if;
end process;

counter20_p: process(clk, next_s)
begin
  if (clk'event and clk='1') then
     if next_s=apel_criptare then
              count20<=count20+1;
      elsif next_s=gen_subchei then
              count20<=0;
     end if;
   end if;
end process;

crip_signal_p: process (clk, next_s)
begin
  if clk'event and clk='1' then
        if next_s=apel_criptare then
              crip<='1';
        else
              crip<='0';
        end if;
```

```
        end if;
      end process;

      final_p: process(clk, next_s, a_out1, b_out1, c_out1, d_out1,s)
      variable a_out:          std_logic_vector(31 downto 0);
      variable b_out:          std_logic_vector(31 downto 0);
      variable c_out:          std_logic_vector(31 downto 0);
      variable d_out:          std_logic_vector(31 downto 0);
      begin
        if (clk'event and clk='1') then
        if next_s=final then
              a_out:=a_out1+s(42);
              b_out:=b_out1;
              c_out:=c_out1+s(43);
              d_out:=d_out1;
              text(127 downto 96)<=a_out(7 downto 0) & a_out(15 downto 8)
                                   & a_out(23 downto 16) & a_out(31 downto 24);
              text(95 downto 64)<=b_out(7 downto 0) & b_out(15 downto 8)
                                  & b_out(23 downto 16) & b_out(31 downto 24);
              text(63 downto 32)<=c_out(7 downto 0) & c_out(15 downto 8)
                                  & c_out(23 downto 16) & c_out(31 downto 24);
              text(31 downto 0)<=d_out(7 downto 0) & d_out(15 downto 8)
                                  & d_out(23 downto 16) & d_out(31 downto 24);

        end if;
        end if;
      end process;
end criptare_arch;
```

Anexa 3

Implementarea hardware în VHDL a modulului de criptare pentru o rundă

```
--varianta adaptata la functionarea ca si componenta a entitatii de criptare
--realizeza criptarea pentru o runda

    library ieee;
    use ieee.std_logic_1164.all;
    use ieee.std_logic_arith.all;
    use ieee.std_logic_unsigned.all;
    entity m_cript is
        port (
          reset: in std_logic;
          clk:          in std_logic;
          cript:        in std_logic;
          a_in:         in std_logic_vector(31 downto 0);
          b_in:         in std_logic_vector(31 downto 0);
          c_in:         in std_logic_vector(31 downto 0);
          d_in:         in std_logic_vector(31 downto 0);
          s0:           in std_logic_vector(31 downto 0);
          s1:           in std_logic_vector(31 downto 0);
          a_out:        out std_logic_vector(31 downto 0);
          b_out:        out std_logic_vector(31 downto 0);
          c_out:        out std_logic_vector(31 downto 0);
          d_out:        out std_logic_vector(31 downto 0);
          rezolvat:     out std_logic
            );
    end m_cript;

    architecture m_cript_arch of m_cript is
        type states is (idle_s, criptare_interior, criptare_final);
        signal current_s, next_s:       states;
        signal a:                       std_logic_vector(31 downto 0);
        signal b:                       std_logic_vector(31 downto 0);
        signal c:                       std_logic_vector(31 downto 0);
        signal d:                       std_logic_vector(31 downto 0);
        type vec is array (31 downto 0) of std_logic_vector(31 downto 0);
    begin
        async_p: process (clk, current_s, cript)
        begin
          case current_s is
              when idle_s =>
                  if cript='1' then
                          next_s <=criptare_interior;
                  else
                          next_s <=idle_s;
                  end if;
            when criptare_interior =>
                          next_s <=criptare_final;
            when criptare_final =>
                          next_s <=idle_s;
          end case;
        end process;
```

```
syncronous_p: process(reset, clk, next_s)
begin
   if reset='1' then
     current_s<=idle_s;
   elsif (clk'event and clk='1') then
      current_s<= next_s;
   end if;
end process;

criptare_p:process(current_s, next_s, clk, s0, s1, a_in, b_in, c_in, d_in)
variable tempb:                    std_logic_vector(31 downto 0);
variable tempd :                   std_logic_vector(31 downto 0);
variable vb, vd:                   vec;
variable vb0,vb1,vb2,vb3,vb4,vb5:  std_logic_vector(31 downto 0);
variable vb6,vb7,vb8,vb9,vb10,vb11: std_logic_vector(31 downto 0);
variable vb12,vb13,vb14,vb15,vb16: std_logic_vector(31 downto 0);
variable vb17,vb18,vb19,vb20,vb21: std_logic_vector(31 downto 0);
variable vb22,vb23,vb24,vb25,vb26: std_logic_vector(31 downto 0);
variable vb27,vb28,vb29,vb30:      std_logic_vector(31 downto 0);
variable vd0,vd1,vd2,vd3,vd4,vd5:  std_logic_vector(31 downto 0);
variable vd6,vd7,vd8,vd9,vd10,vd11: std_logic_vector(31 downto 0);
variable vd12,vd13,vd14,vd15,vd16: std_logic_vector(31 downto 0);
variable vd17,vd18,vd19,vd20,vd21: std_logic_vector(31 downto 0);
variable vd22,vd23,vd24,vd25,vd26: std_logic_vector(31 downto 0);
variable vd27,vd28,vd29,vd30:      std_logic_vector(31 downto 0);
variable i, intu, intt :           integer range 0 to 31;
variable t, u, xora, xorc:         std_logic_vector(31 downto 0);
begin
  if (clk'event and clk='1') then
      if next_s=criptare_interior then
       for i in 0 to 30 loop
          vb(i):= b_in(i)&b_in(i)&b_in(i)&b_in(i)&b_in(i)&b_in(i)&b_in(i)&b_in(i)
                  &b_in(i)&b_in(i)&b_in(i)&b_in(i)&b_in(i)&b_in(i)&b_in(i)&b_in(i)
                  &b_in(i)&b_in(i)&b_in(i)&b_in(i)&b_in(i)&b_in(i)&b_in(i)&b_in(i)
                  &b_in(i)&b_in(i)&b_in(i)&b_in(i)&b_in(i)&b_in(i)&b_in(i)&b_in(i);
          vb(i):=vb(i) and b_in;
          vd(i):= d_in(i)&d_in(i)&d_in(i)&d_in(i)&d_in(i)&d_in(i)&d_in(i)&d_in(i)
                  &d_in(i)&d_in(i)&d_in(i)&d_in(i)&d_in(i)&d_in(i)&d_in(i)&d_in(i)
                  &d_in(i)&d_in(i)&d_in(i)&d_in(i)&d_in(i)&d_in(i)&d_in(i)&d_in(i)
                  &d_in(i)&d_in(i)&d_in(i)&d_in(i)&d_in(i)&d_in(i)&d_in(i)&d_in(i);
          vd(i):=vd(i) and d_in;
  end loop;
  vb0:=vb(0);                                   vb8:=vb(8);
  vb1:=vb(1);                                   vb9:=vb(9);
  vb2:=vb(2);                                   vb10:=vb(10);
  vb3:=vb(3);                                   vb11:=vb(11);
  vb4:=vb(4);                                   vb12:=vb(12);
  vb5:=vb(5);                                   vb13:=vb(13);
  vb6:=vb(6);                                   vb14:=vb(14);
  vb7:=vb(7);                                   vb15:=vb(15);
```

```
vb16:=vb(16);
vb17:=vb(17);
vb18:=vb(18);
vb19:=vb(19);
vb20:=vb(20);
vb21:=vb(21);
vb22:=vb(22);
vb23:=vb(23);
vb24:=vb(24);
vb25:=vb(25);
vb26:=vb(26);
vb27:=vb(27);
vb28:=vb(28);
vb29:=vb(29);
vb30:=vb(30);

vd0:=vd(0);
vd1:=vd(1);
vd2:=vd(2);
vd3:=vd(3);
vd4:=vd(4);
vd5:=vd(5);
vd6:=vd(6);
vd7:=vd(7);
vd8:=vd(8);
vd9:=vd(9);
vd10:=vd(10);
vd11:=vd(11);
vd12:=vd(12);
vd13:=vd(13);
vd14:=vd(14);
vd15:=vd(15);
vd16:=vd(16);
vd17:=vd(17);
vd18:=vd(18);
vd19:=vd(19);
vd20:=vd(20);
vd21:=vd(21);
vd22:=vd(22);
vd23:=vd(23);
vd24:=vd(24);
vd25:=vd(25);
vd26:=vd(26);
vd27:=vd(27);
vd28:=vd(28);
vd29:=vd(29);
vd30:=vd(30);

tempb:=b_in+(vb0(30 downto 0)&"0")+(vb1(29 downto 0)&"00")
        +(vb2(28 downto 0)&"000")
        +(vb3(27 downto 0)&"0000")+(vb4(26 downto 0)&"00000")
        +(vb5(25 downto 0)&"000000")
        +(vb6(24 downto 0)&"0000000")+(vb7(23 downto 0)&"00000000")
        +(vb8(22 downto 0)&"000000000")+(vb9(21 downto 0)&"0000000000")
        +(vb10(20 downto 0)&"00000000000")
        +(vb11(19 downto 0)&"000000000000")
        +(vb12(18 downto 0)&"0000000000000")
        +(vb13(17 downto 0)&"00000000000000")
        +(vb14(16 downto 0)&"000000000000000")
        +(vb15(15 downto 0)&"0000000000000000")
        +(vb16(14 downto 0)&"00000000000000000")
        +(vb17(13 downto 0)&"000000000000000000")
        +(vb18(12 downto 0)&"0000000000000000000")
        +(vb19(11 downto 0)&"00000000000000000000")
        +(vb20(10 downto 0)&"000000000000000000000")
        +(vb21(9 downto 0)&"0000000000000000000000")
        +(vb22(8 downto 0)&"00000000000000000000000")
        +(vb23(7 downto 0)&"000000000000000000000000")
        +(vb24(6 downto 0)&"0000000000000000000000000")
        +(vb25(5 downto 0)&"00000000000000000000000000")
        +(vb26(4 downto 0)&"000000000000000000000000000")
        +(vb27(3 downto 0)&"0000000000000000000000000000")
```

```
        +(vb28(2 downto 0)&"00000000000000000000000000000000")
        +(vb29(1 downto 0)&"00000000000000000000000000000000")
        +(vb30(0)&"00000000000000000000000000000000");
tempd:=d_in+(vd0(30 downto 0)&"0")+(vd1(29 downto 0)&"00")
        +(vd2(28 downto 0)&"000")+(vd3(27 downto 0)&"0000")
        +(vd4(26 downto 0)&"00000")+(vd5(25 downto 0)&"000000")
        +(vd6(24 downto 0)&"0000000")+(vd7(23 downto 0)&"00000000")
        +(vd8(22 downto 0)&"000000000")+(vd9(21 downto 0)&"0000000000")
        +(vd10(20 downto 0)&"00000000000")
        +(vd11(19 downto 0)&"000000000000")
        +(vd12(18 downto 0)&"0000000000000")
        +(vd13(17 downto 0)&"00000000000000")
        +(vd14(16 downto 0)&"000000000000000")
        +(vd15(15 downto 0)&"0000000000000000")
        +(vd16(14 downto 0)&"00000000000000000")
        +(vd17(13 downto 0)&"000000000000000000")
        +(vd18(12 downto 0)&"0000000000000000000")
        +(vd19(11 downto 0)&"00000000000000000000")
        +(vd20(10 downto 0)&"000000000000000000000")
        +(vd21(9 downto 0)&"0000000000000000000000")
        +(vd22(8 downto 0)&"00000000000000000000000")
        +(vd23(7 downto 0)&"000000000000000000000000")
        +(vd24(6 downto 0)&"0000000000000000000000000")
        +(vd25(5 downto 0)&"00000000000000000000000000")
        +(vd26(4 downto 0)&"000000000000000000000000000")
        +(vd27(3 downto 0)&"0000000000000000000000000000")
        +(vd28(2 downto 0)&"00000000000000000000000000000")
        +(vd29(1 downto 0)&"000000000000000000000000000000")
        +(vd30(0)&"0000000000000000000000000000000");

t:=tempb(26 downto 0) &tempb(31 downto 27);
u:=tempd(26 downto 0) & tempd(31 downto 27);
a<=b_in;
        xorc:=c_in xor u;
        xora:=a_in xor t;
        intu:=conv_integer(u(4 downto 0));
        intt:=conv_integer(t(4 downto 0));
        case intt is
                when 0 =>
                        b<=xorc(31 downto 0)+s1;
                when 1 =>
                        b<=xorc(30 downto 0) & xorc(31 downto 31)+s1;
                when 2 =>
                        b<=xorc(29 downto 0) & xorc(31 downto 30)+s1;
                when 3 =>
                        b<=xorc(28 downto 0) & xorc(31 downto 29)+s1;
                when 4 =>
                        b<=xorc(27 downto 0) & xorc(31 downto 28)+s1;
                when 5 =>
                        b<=xorc(26 downto 0) & xorc(31 downto 27)+s1;
                when 6 =>
```

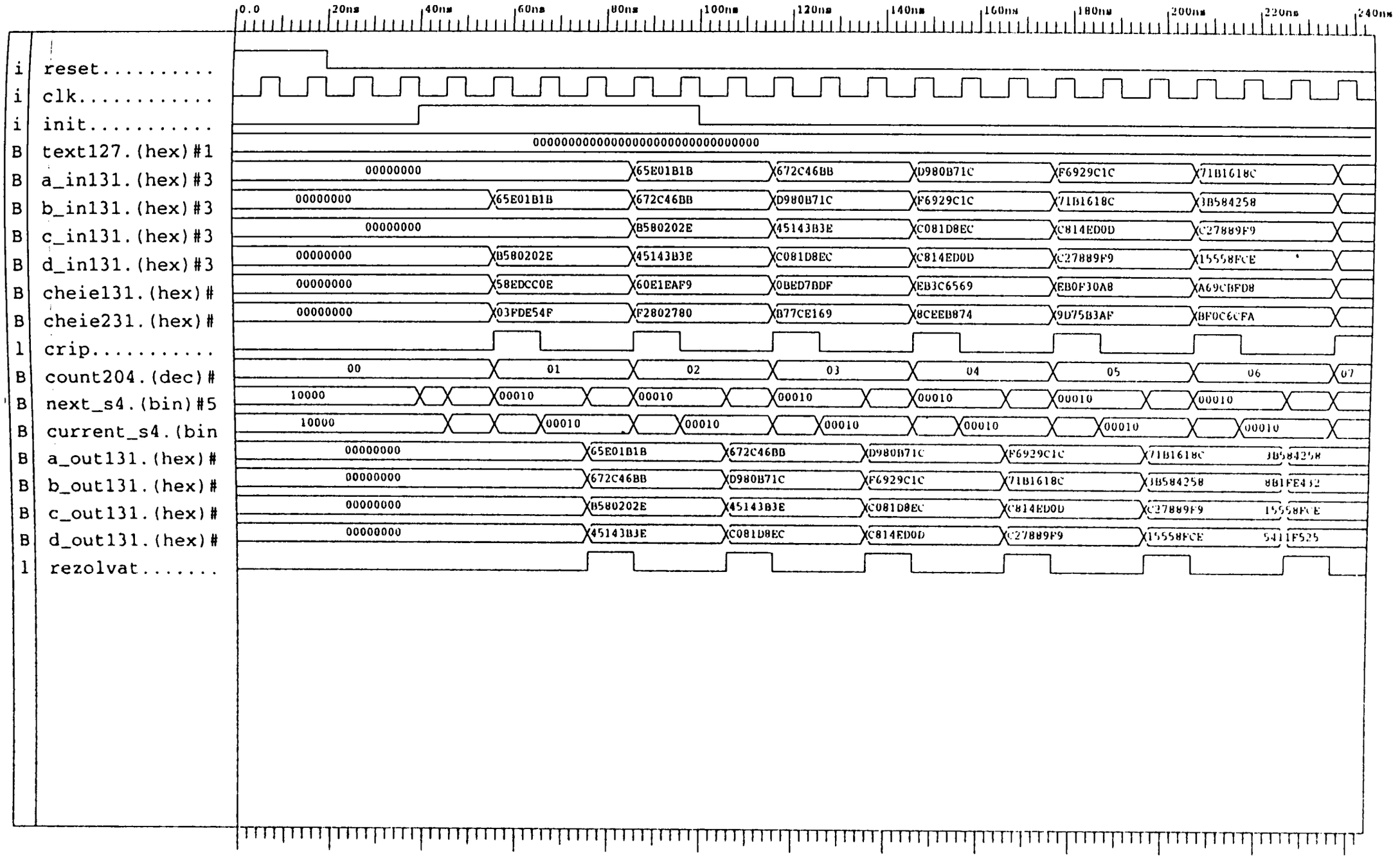
```
        b<=xorc(25 downto 0) & xorc(31 downto 26)+s1;
when 7 =>
        b<=xorc(24 downto 0) & xorc(31 downto 25)+s1;
when 8 =>
        b<=xorc(23 downto 0) & xorc(31 downto 24)+s1;
when 9 =>
        b<=xorc(22 downto 0) & xorc(31 downto 23)+s1;
when 10 =>
        b<=xorc(21 downto 0) & xorc(31 downto 22)+s1;
when 11 =>
        b<=xorc(20 downto 0) & xorc(31 downto 21)+s1;
when 12 =>
        b<=xorc(19 downto 0) & xorc(31 downto 20)+s1;
when 13 =>
        b<=xorc(18 downto 0) & xorc(31 downto 19)+s1;
when 14 =>
        b<=xorc(17 downto 0) & xorc(31 downto 18)+s1;
when 15 =>
        b<=xorc(16 downto 0) & xorc(31 downto 17)+s1;
when 16 =>
        b<=xorc(15 downto 0) & xorc(31 downto 16)+s1;
when 17 =>
        b<=xorc(14 downto 0) & xorc(31 downto 15)+s1;
when 18 =>
        b<=xorc(13 downto 0) & xorc(31 downto 14)+s1;
when 19 =>
        b<=xorc(12 downto 0) & xorc(31 downto 13)+s1;
when 20 =>
        b<=xorc(11 downto 0) & xorc(31 downto 12)+s1;
when 21 =>
        b<=xorc(10 downto 0) & xorc(31 downto 11)+s1;
when 22 =>
        b<=xorc(9 downto 0) & xorc(31 downto 10)+s1;
when 23 =>
        b<=xorc(8 downto 0) & xorc(31 downto 9)+s1;
when 24 =>
        b<=xorc(7 downto 0) & xorc(31 downto 8)+s1;
when 25 =>
        b<=xorc(6 downto 0) & xorc(31 downto 7)+s1;
when 26 =>
        b<=xorc(5 downto 0) & xorc(31 downto 6)+s1;
when 27 =>
        b<=xorc(4 downto 0) & xorc(31 downto 5)+s1;
when 28 =>
        b<=xorc(3 downto 0) & xorc(31 downto 4)+s1;
when 29 =>
        b<=xorc(2 downto 0) & xorc(31 downto 3)+s1;
when 30=>
        b<=xorc(1 downto 0) & xorc(31 downto 2)+s1;
when 31 =>
        b<=xorc(0) & xorc(31 downto 1)+s1;
```

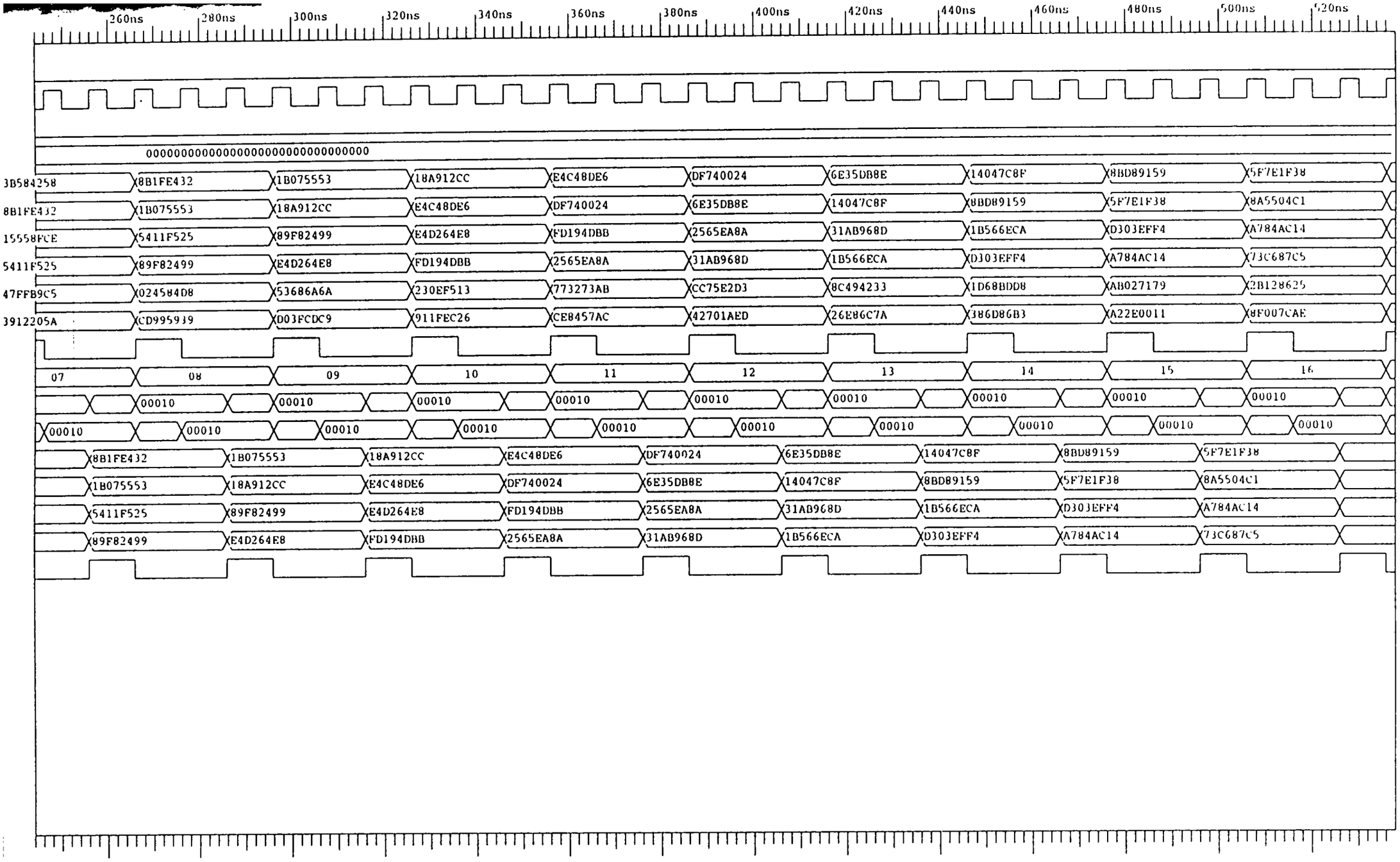
```
        end case;
    c<=d_in;
        case intu is
            when 0=>
                d<=xora(31 downto 0)+s0;
            when 1 =>
                d<=xora(30 downto 0) & xora(31 downto 31)+s0;
            when 2 =>
                d<=xora(29 downto 0) & xora(31 downto 30)+s0;
            when 3 =>
                d<=xora(28 downto 0) & xora(31 downto 29)+s0;
            when 4 =>
                d<=xora(27 downto 0) & xora(31 downto 28)+s0;
            when 5 =>
                d<=xora(26 downto 0) & xora(31 downto 27)+s0;
            when 6 =>
                d<=xora(25 downto 0) & xora(31 downto 26)+s0;
            when 7 =>
                d<=xora(24 downto 0) & xora(31 downto 25)+s0;
            when 8 =>
                d<=xora(23 downto 0) & xora(31 downto 24)+s0;
            when 9 =>
                d<=xora(22 downto 0) & xora(31 downto 23)+s0;
            when 10 =>
                d<=xora(21 downto 0) & xora(31 downto 22)+s0;
            when 11 =>
                d<=xora(20 downto 0) & xora(31 downto 21)+s0;
            when 12 =>
                d<=xora(19 downto 0) & xora(31 downto 20)+s0;
            when 13 =>
                d<=xora(18 downto 0) & xora(31 downto 19)+s0;
            when 14 =>
                d<=xora(17 downto 0) & xora(31 downto 18)+s0;
            when 15 =>
                d<=xora(16 downto 0) & xora(31 downto 17)+s0;
            when 16 =>
                d<=xora(15 downto 0) & xora(31 downto 16)+s0;
            when 17 =>
                d<=xora(14 downto 0) & xora(31 downto 15)+s0;
            when 18 =>
                d<=xora(13 downto 0) & xora(31 downto 14)+s0;
            when 19 =>
                d<=xora(12 downto 0) & xora(31 downto 13)+s0;
            when 20 =>
                d<=xora(11 downto 0) & xora(31 downto 12)+s0;
            when 21 =>
                d<=xora(10 downto 0) & xora(31 downto 11)+s0;
            when 22 =>
                d<=xora(9 downto 0) & xora(31 downto 10)+s0;
            when 23 =>
                d<=xora(8 downto 0) & xora(31 downto 9)+s0;
```

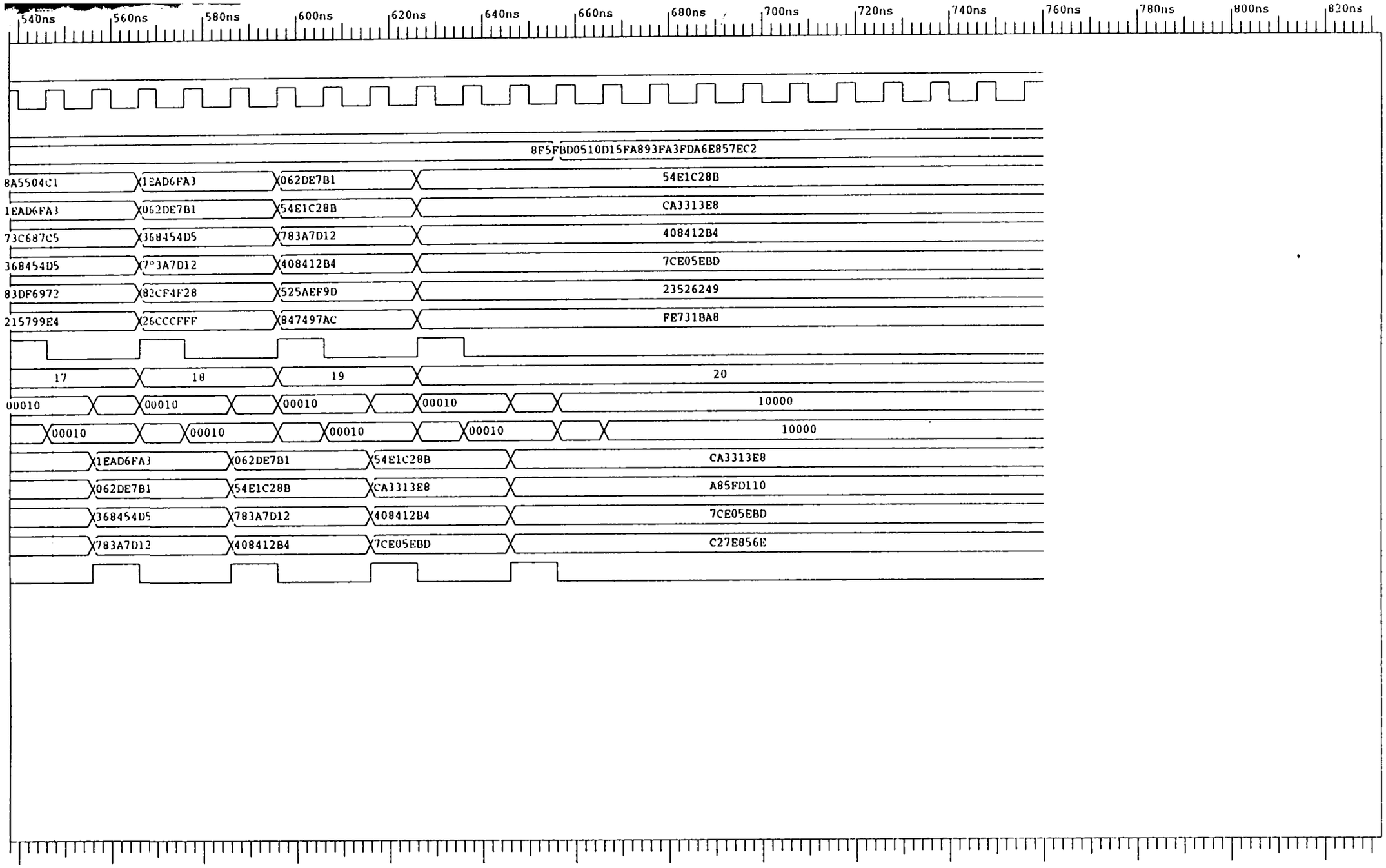
```
                    when 24 =>
                        d<=xora(7 downto 0) & xora(31 downto 8)+s0;
                    when 25 =>
                        d<=xora(6 downto 0) & xora(31 downto 7)+s0;
                    when 26 =>
                        d<=xora(5 downto 0) & xora(31 downto 6)+s0;
                    when 27 =>
                        d<=xora(4 downto 0) & xora(31 downto 5)+s0;
                    when 28 =>
                        d<=xora(3 downto 0) & xora(31 downto 4)+s0;
                    when 29 =>
                        d<=xora(2 downto 0) & xora(31 downto 3)+s0;
                    when 30=>
                        d<=xora(1 downto 0) & xora(31 downto 2)+s0;
                    when 31 =>
                        d<=xora(0) & xora(31 downto 1)+s0;
            end case;
      elsif next_s=criptare_final then
                    a_out<=a;
                    b_out<=b;
                    c_out<=c;
                    d_out<=d;
      end if;
      end if;
    end process;

    rezolvat_p: process(clk, next_s)
    begin
    if (clk'event and clk='1') then
      if next_s=criptare_final then
            rezolvat<='1';
      else
            rezolvat<='0';
      end if;
    end if;
    end process;
end m_cript_arch;
```

Anexa 4

Diagrama de timp pentru criptare

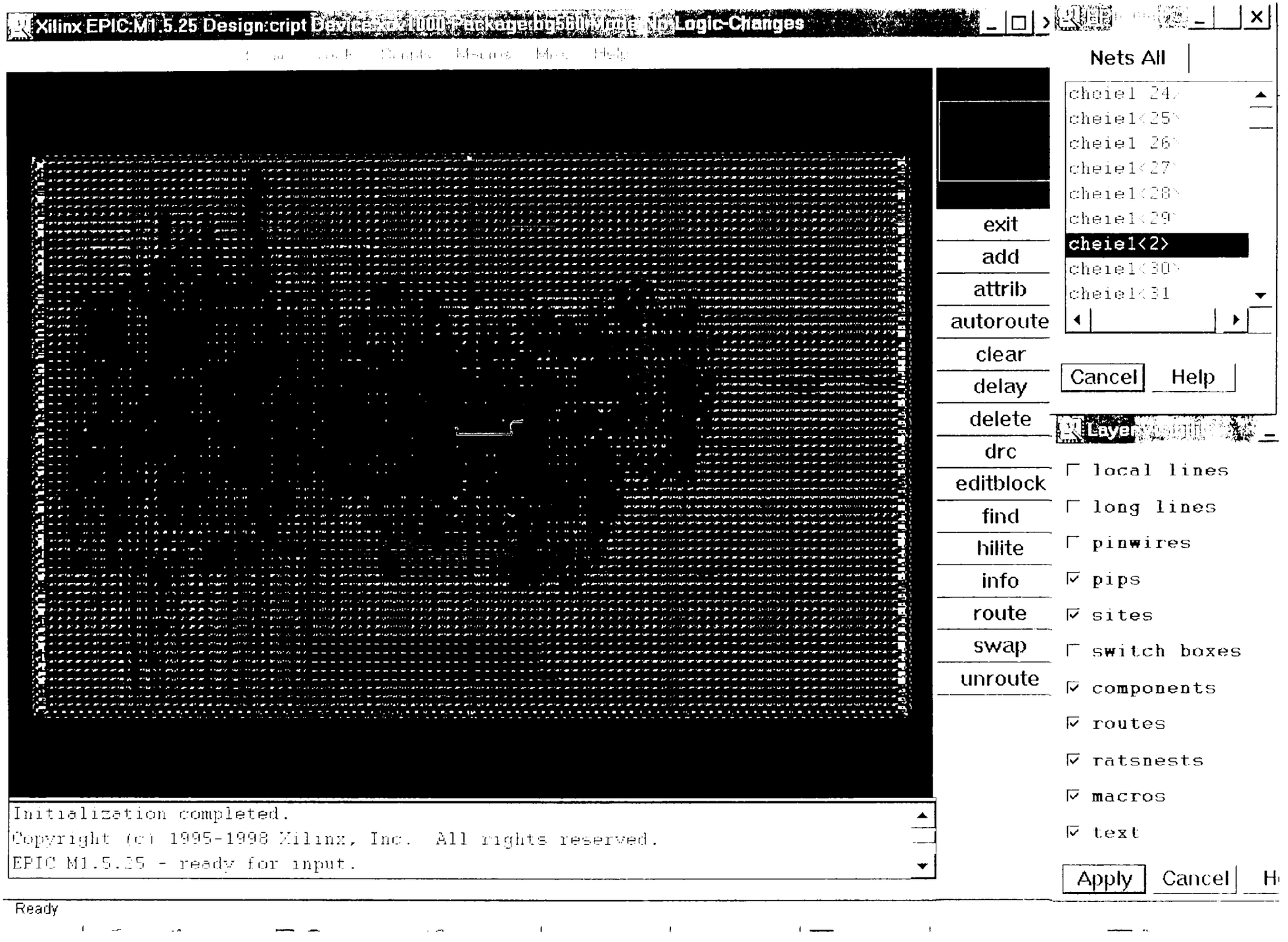

0.0
20ns
40ns
60ns
80ns
100ns
120ns
140ns
160ns
180ns
200ns
220ns
240ns
i reset.....
i clk.....
i init.....
B text127.(hex)#1
B a_in131.(hex)#3
B b_in131.(hex)#3
B c_in131.(hex)#3
B d_in131.(hex)#3
B cheie131.(hex)#
B cheie231.(hex)#
l crip.....
B count204.(dec)#
B next_s4.(bin)#5
B current_s4.(bin
B a_out131.(hex)#
B b_out131.(hex)#
B c_out131.(hex)#
B d_out131.(hex)#
l rezolvat.....
00000000000000000000000000000000
00000000
65E01B1B
672C46BB
D980B71C
F6929C1C
71B1618C
3B584258
B580202E
45143B3E
C081D8EC
C814ED0D
C27889F9
15558FCE
58EDCC0E
60E1EAF9
0BED7BDF
EB3C6569
EB0F30A8
A69CBFD8
03FDE54F
F2802780
B77CE169
8CEEB874
9D75B3AF
BF0C6CFA
00
01
02
03
04
05
06
07
10000
00010
8B1FE432
5411F525

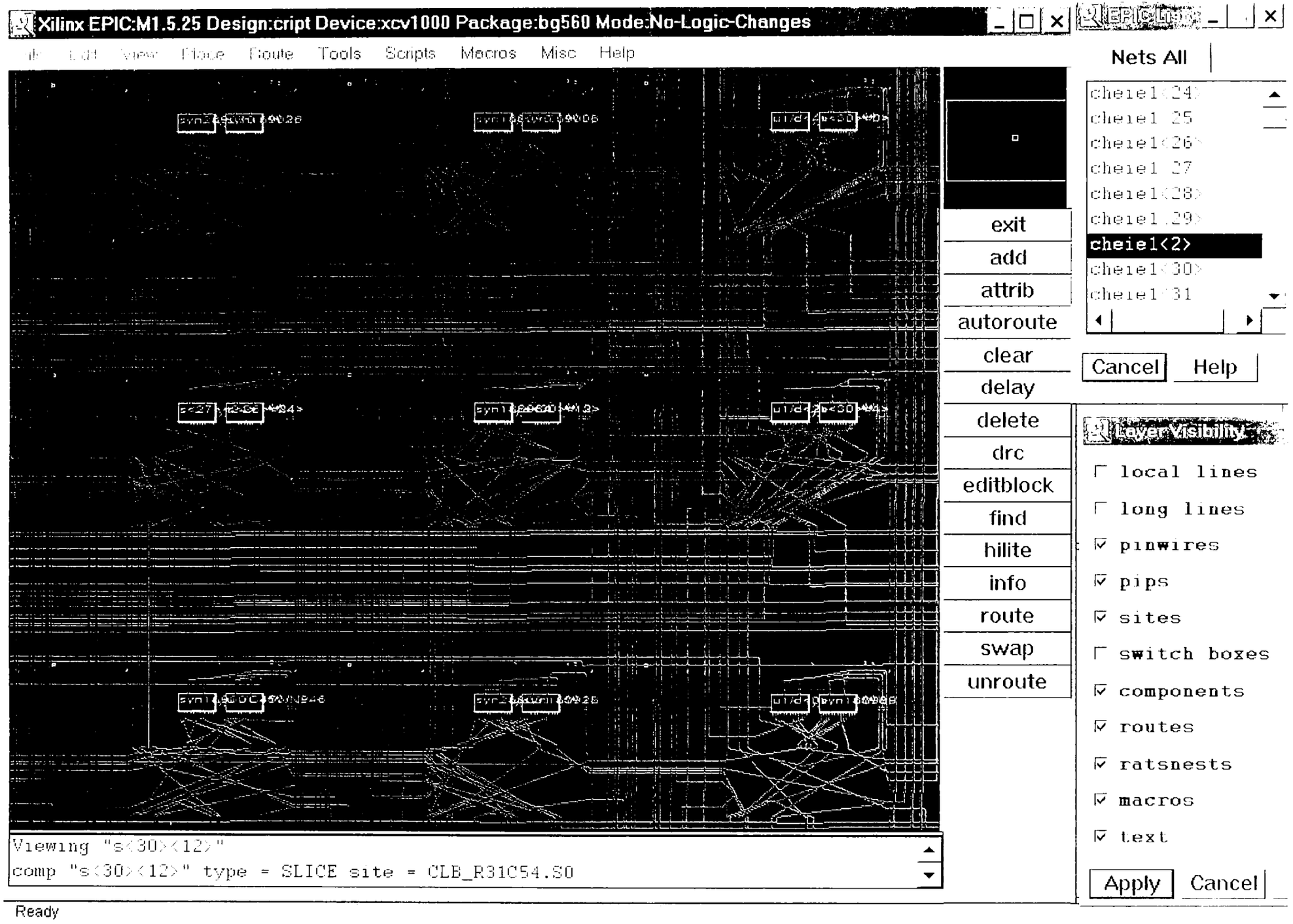

260ns
280ns
300ns
320ns
340ns
360ns
380ns
400ns
420ns
440ns
460ns
480ns
500ns
520ns
00000000000000000000000000000000
3B584258
8B1FE432
1B075553
18A912CC
E4C48DE6
DF740024
6E35DB8E
14047C8F
8BD89159
5F7E1F38
8B1FE432
1B075553
18A912CC
E4C48DE6
DF740024
6E35DB8E
14047C8F
8BD89159
5F7E1F38
8A5504C1
15558FCE
5411F525
89F82499
E4D264E8
FD194DBB
2565EA8A
31AB968D
1B566ECA
D303EFF4
A784AC14
5411F525
89F82499
E4D264E8
FD194DBB
2565EA8A
31AB968D
1B566ECA
D303EFF4
A784AC14
73C687C5
47FFB9C5
024584D8
53686A6A
230EF513
773273AB
CC75E2D3
8C494233
1D68BDD8
AB027179
2B128625
3912205A
CD995939
D03FCDC9
911FEC26
CE8457AC
42701AED
26E86C7A
386D86B3
A22E0011
8F007CAE
07
08
09
10
11
12
13
14
15
16
00010
00010
00010
00010
00010
00010
00010
00010
00010
00010
00010
00010
00010
00010
00010
00010
00010
00010
00010
00010
8B1FE432
1B075553
18A912CC
E4C48DE6
DF740024
6E35DB8E
14047C8F
8BD89159
5F7E1F38
1B075553
18A912CC
E4C48DE6
DF740024
6E35DB8E
14047C8F
8BD89159
5F7E1F38
8A5504C1
5411F525
89F82499
E4D264E8
FD194DBB
2565EA8A
31AB968D
1B566ECA
D303EFF4
A784AC14
89F82499
E4D264E8
FD194DBB
2565EA8A
31AB968D
1B566ECA
D303EFF4
A784AC14
73C687C5

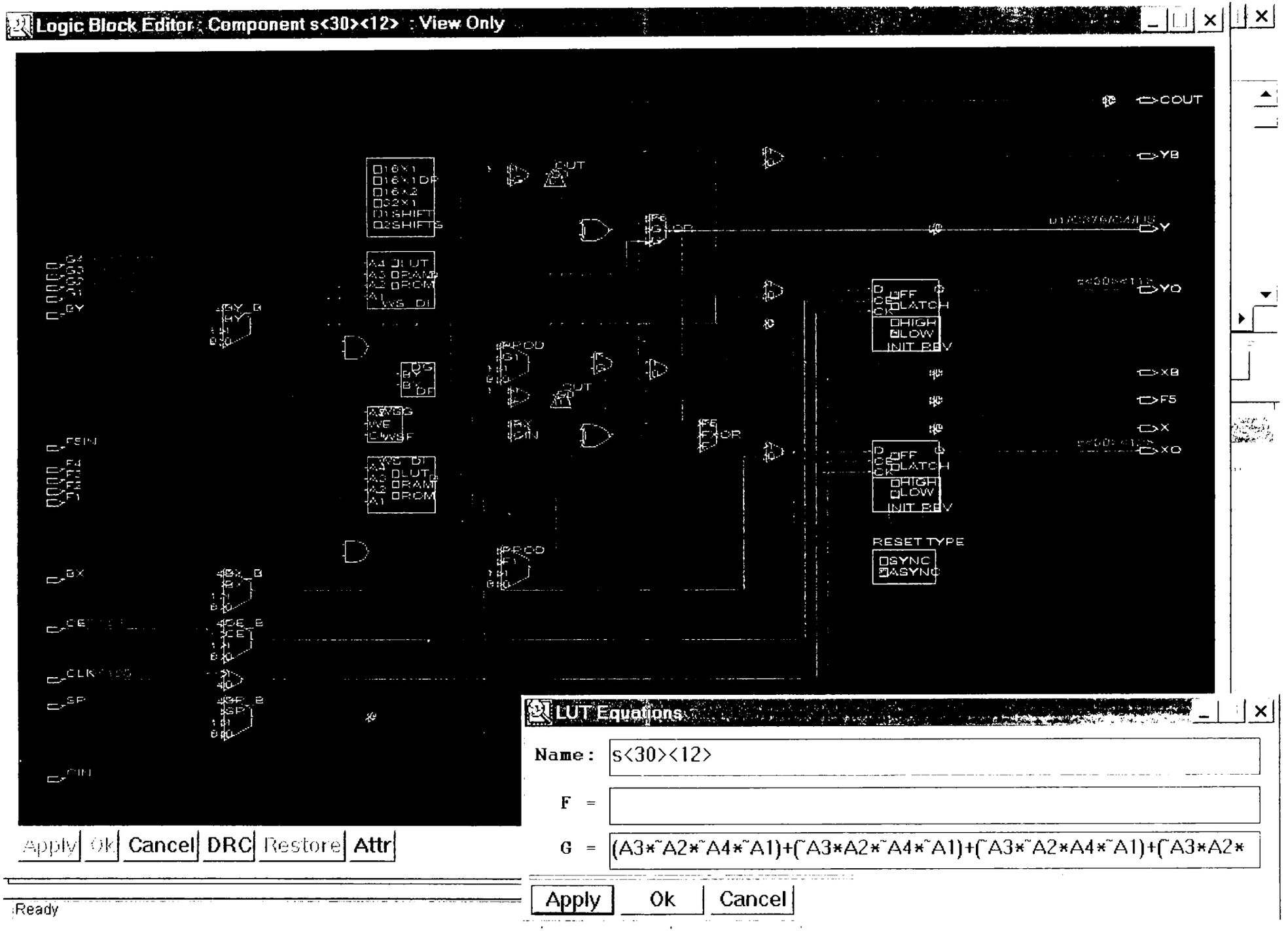

540ns
560ns
580ns
600ns
620ns
640ns
660ns
680ns
700ns
720ns
740ns
760ns
780ns
800ns
820ns
8F5FBD0510D15FA893FA3FDA6E857EC2
8A5504C1
1EAD6FA3
062DE7B1
54E1C28B
1EAD6FA3
062DE7B1
54E1C28B
CA3313E8
73C687C5
368454D5
783A7D12
408412B4
368454D5
783A7D12
408412B4
7CE05EBD
83DF6972
82CF4F28
525AEF9D
23526249
215799E4
26CCCFFF
847497AC
FE731BA8
17
18
19
20
00010
00010
00010
00010
10000
00010
00010
00010
00010
10000
1EAD6FA3
062DE7B1
54E1C28B
CA3313E8
062DE7B1
54E1C28B
CA3313E8
A85FD110
368454D5
783A7D12
408412B4
7CE05EBD
783A7D12
408412B4
7CE05EBD
C27E856E

Anexa 5

Harta distribuției circuitelor din modulul de criptare


Xilinx EPIC:M1.5.25 Design:cript
Logic-Changes
Nets All
cheie1<24>
cheie1<25>
cheie1<26>
cheie1<27>
cheie1<28>
cheie1<29>
cheie1<2>
cheie1<30>
cheie1<31
exit
add
attrib
autoroute
clear
delay
delete
drc
editblock
find
hilite
info
route
swap
unroute
Cancel
Help
local lines
long lines
pinwires
pips
sites
switch boxes
components
routes
ratsnests
macros
text
Apply
Cancel
Initialization completed.
Copyright (c) 1995-1998 Xilinx, Inc. All rights reserved.
EPIC M1.5.25 - ready for input.
Ready


Xilinx EPIC:M1.5.25 Design:cript Device:xcv1000 Package:bg560 Mode:No-Logic-Changes
File Edit View Place Route Tools Scripts Macros Misc Help
Nets All
cheie1<24>
cheie1<25>
cheie1<26>
cheie1<27>
cheie1<28>
cheie1<29>
cheie1<2>
cheie1<30>
cheie1<31>
exit
add
attrib
autoroute
clear
delay
delete
drc
editblock
find
hilite
info
route
swap
unroute
Cancel
Help
Layer Visibility
local lines
long lines
pinwires
pips
sites
switch boxes
components
routes
ratsnests
macros
text
Apply
Cancel
Viewing "s<30><12>"
comp "s<30><12>" type = SLICE site = CLB_R31C54.S0
Ready


Logic Block Editor : Component s<30><12> : View Only
COUT
YB
Y
YO
XB
F5
X
XO
RESET TYPE
SYNC
ASYNC
Apply
Ok
Cancel
DRC
Restore
Attr
LUT Equations
Name: s<30><12>
F =
G = (A3*~A2*~A4*~A1)+(~A3*A2*~A4*~A1)+(~A3*~A2*A4*~A1)+(~A3*A2*
Apply
Ok
Cancel
Ready

Anexa 6

Implementarea hardware în VHDL a modulului de generare al cheilor din algoritmul RC6

```
--      modul pentru generarea subcheilor

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity gen_chei is
   port (
        reset:                  in std_logic;
        clk:                    in std_logic;
        init:                   in std_logic;
--      chei:                   in std_logic_vector (255 downto 0);
         );
end gen_chei;

architecture gen_chei_arch of gen_chei is
      constant p :    std_logic_vector:= "10110111111000010101000101100011";
      constant q :    std_logic_vector:= "10011110001101110111100110111001";
      signal a_ch:    std_logic_vector(31 downto 0);
      signal b_ch:    std_logic_vector(31 downto 0);
      signal i :      integer range 0 to 43;
      signal j :      integer range 0 to 7;
--    stari
      type states is (asteapta, initializare, gen_subchei,final);
      signal current_s, next_s:       states;
      signal count132 :               integer range 0 to 132;
      type tabel is array (43 downto 0) of std_logic_vector(31 downto 0);
      signal s  :                     tabel;
      type tabel_chei_init is array (7 downto 0) of std_logic_vector(31 downto 0);
      signal l:                       tabel_chei_init;
      signal a_in:                    std_logic_vector(31 downto 0);
      signal b_in:                    std_logic_vector(31 downto 0);
      signal c_in:                    std_logic_vector(31 downto 0);
      signal d_in:                    std_logic_vector(31 downto 0);

begin
      syncronous_p: process(reset, clk, next_s)
      begin
       if reset='1' then
              current_s<=asteapta;
       elsif (clk'event and clk='1') then
              current_s<= next_s;
       end if;
      end process;


      async_p: process (current_s, init, count132, clk)
      begin
        case current_s is
          when asteapta =>
```

```
            if init='1' then
                    next_s <=initializare;
            else
                    next_s <=asteapta;
            end if;
        when initializare=>
                    next_s <=gen_subchei;
        when gen_subchei =>
            if count132 = 132  then
                    next_s <=final;
            else
                    next_s <=gen_subchei;
            end if;
        when final =>
                    next_s <=asteapta;
     end case;
    end process;

    counter_p: process(clk, next_s)
    begin
     if (clk'event and clk='1') then
          if next_s=gen_subchei then
            count132<=count132+1;
          elsif (next_s=initializare or next_s=asteapta) then
            count132<=0;
          end if;
     end if;
    end process;

    generarea_cheilor_p: process(current_s, next_s,  clk, l, a_ch, b_ch, i, j,s)
    variable interm  :        std_logic_vector(31 downto 0);
    variable interm1:         std_logic_vector(31 downto 0);
    variable interm2:         std_logic_vector(31 downto 0);
    variable temp:            std_logic_vector(31 downto 0);
    variable intab:           integer range 0 to 31;
    begin
     if (clk'event and clk='1') then
          if  next_s=initializare then
--valorile de initializare sunt introduse pentru simplificarea logicii
            s(0)<="10110111111000010101000101100011";
            s(1)<="01010110000110001100101100011100";
            s(2)<="11110100010100000100010011010101";
            s(3)<="10010010100001111011111010001110";
            s(4)<="00110000101111110011100001000111";
            s(5)<="11001110111101101011001000000000";
            s(6)<="01101101001011100010101110111001";
            s(7)<="00001011011001011010010101110010";
            s(8)<="10101001100111010001111100101011";
            s(9)<="01000111110101001001100011100100";
            s(10)<="11100110000011000001001010011101";
            s(11)<="10000100010000111000110001010110";
```

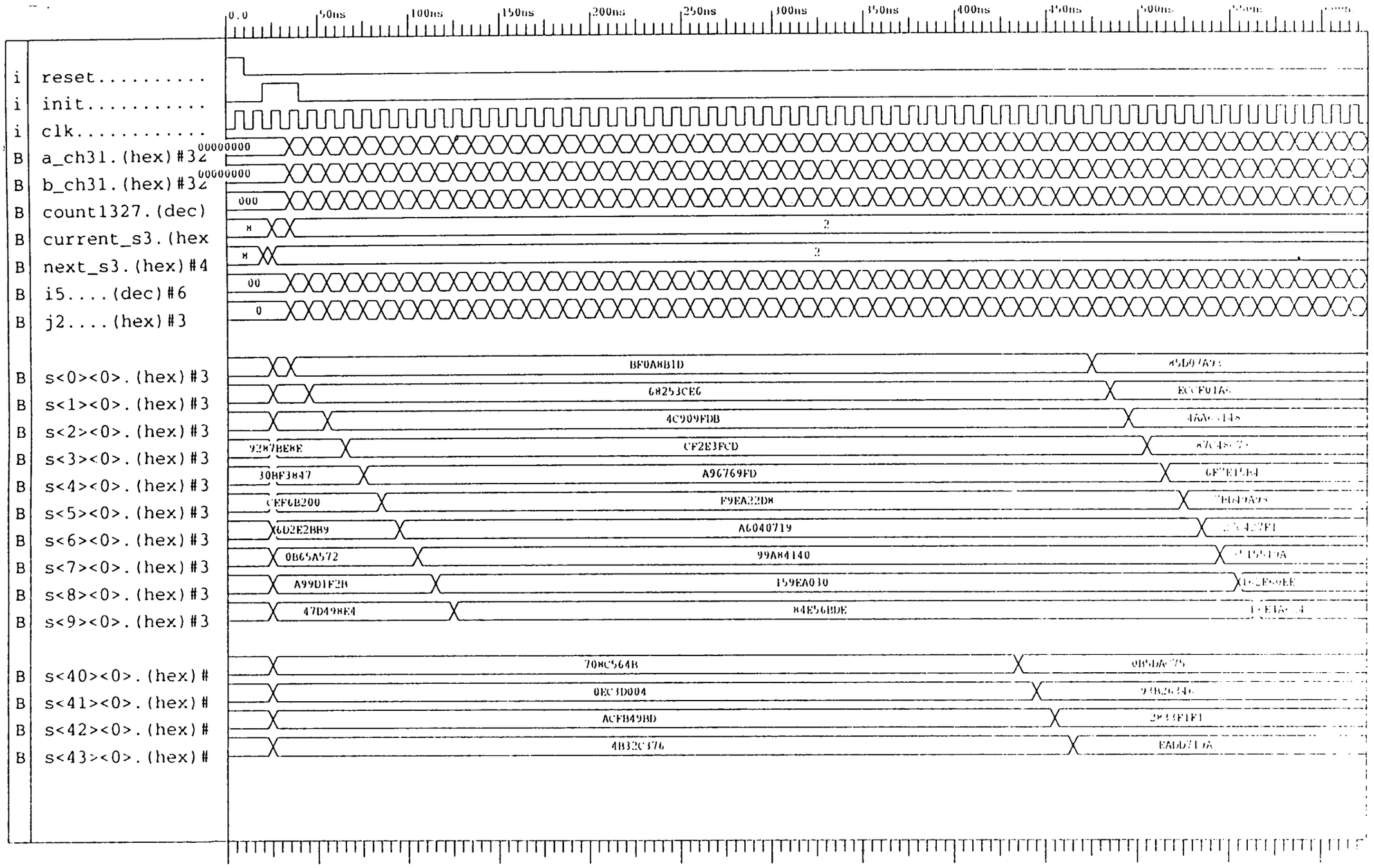
```
            s(12)<="00100010011110110000011000001111";
            s(13)<="11000000101100100111111111001000";
            s(14)<="01011110111010011111100110000001";
            s(15)<="11111101001000010111001100111010";
            s(16)<="10011011010110001110110011110011";
            s(17)<="00111001100100000110011010101100";
            s(18)<="11010111110001111110000001100101";
            s(19)<="01110101111111110101101000011110";
            s(20)<="00010100001101101101001111010111";
            s(21)<="10110010011011100100110110010000";
            s(22)<="01010000101001011100011101001001";
            s(23)<="11101110110111010100000100000010";
            s(24)<="10001101000101001011101010111011";
            s(25)<="00101011010011000011010001110100";
            s(26)<="11001001100000111010111000101101";
            s(27)<="01100111101110110010011111100110";
            s(28)<="00000101111100101010000110011111";
            s(29)<="10100100001010100001101101011000";
            s(30)<="01000010011000011001010100010001";
            s(31)<="11100000100110010000111011001010";
            s(32)<="01111110110100001000100010000011";
            s(33)<="00011101000010000000001000111100";
            s(34)<="10111011001111110111101111110101";
            s(35)<="01011001011101101111010110101110";
            s(36)<="11110111101011100110111101100111";
            s(37)<="10010101111001011110100100100000";
            s(38)<="00110100000111010110001011011001";
            s(39)<="11010010010101001101110010010010";
            s(40)<="01110000100011000101011001001011";
            s(41)<="00001110110000111101000000000100";
            s(42)<="10101100111110110100100110111101";
            s(43)<="01001011001100101100001101110110";
            a_ch<="00000000000000000000000000000000";
            b_ch<="00000000000000000000000000000000";
            i<=0;
            j<=0;
--caz particular, cheia utilizator fiind "0...0"
            l(0)<="00000000000000000000000000000000";
            l(1)<="00000000000000000000000000000000";
            l(2)<="00000000000000000000000000000000";
            l(3)<="00000000000000000000000000000000";
            l(4)<="00000000000000000000000000000000";
            l(5)<="00000000000000000000000000000000";
            l(6)<="00000000000000000000000000000000";
            l(7)<="00000000000000000000000000000000";
        elsif next_s=gen_subchei then
            interm:= s(i)+a_ch+b_ch;
            interm1:=interm(28 downto 0) & interm(31 downto 29);
            s(i)<=interm1;
            a_ch<=interm1;
            interm1:=interm1+b_ch;
```

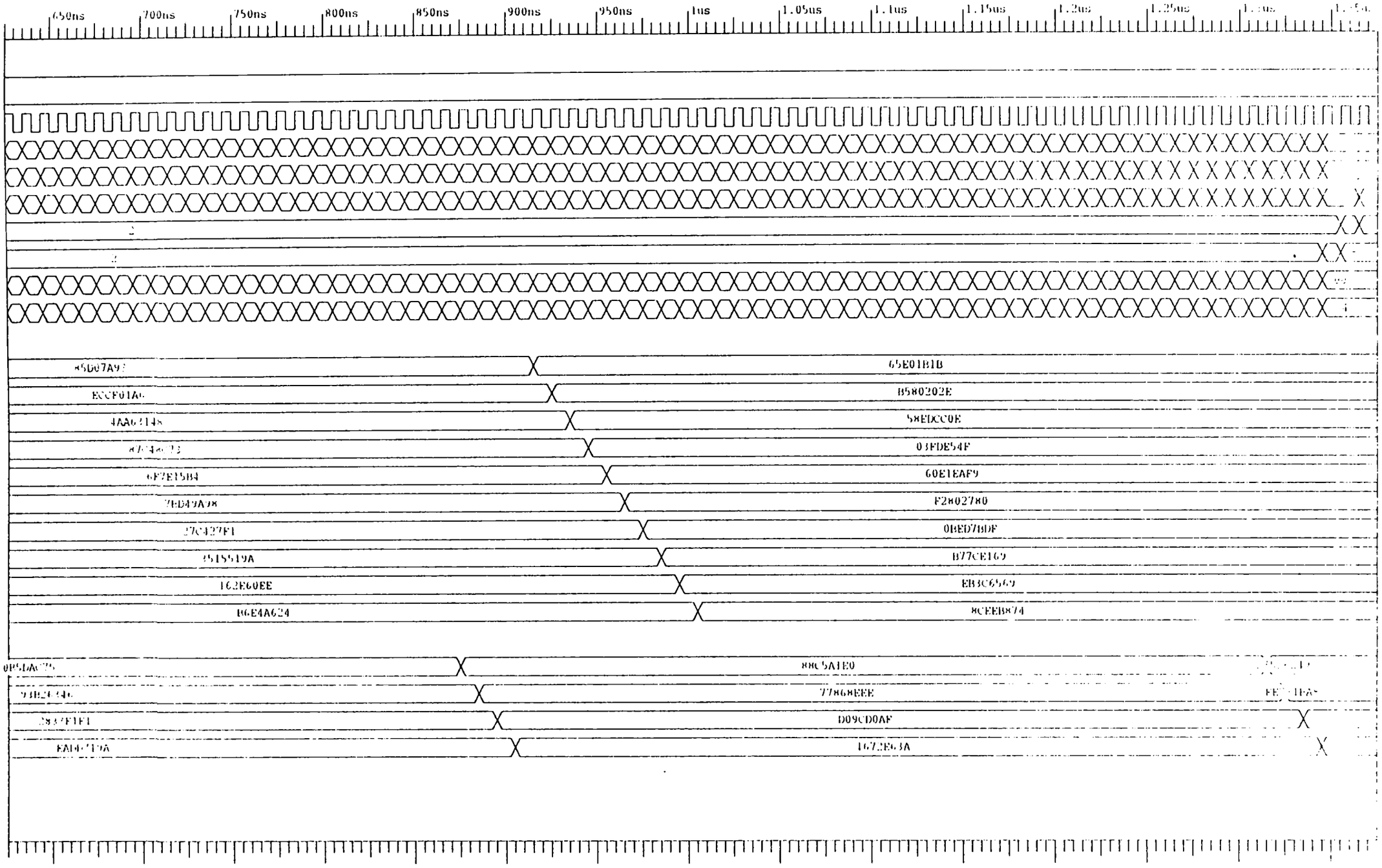
```
interm2:=l(j)+interm1;
intab:=conv_integer(interm1(4 downto 0));
case intab is
when 0 =>
        temp:=interm2;
when 1  =>
        temp:=interm2(30 downto 0) & interm2(31);
when 2 =>
        temp:=interm2(29 downto 0) & interm2(31 downto 30);
when 3 =>
        temp:=interm2(28 downto 0) & interm2(31 downto 29);
when 4 =>
        temp:=interm2(27 downto 0) & interm2(31 downto 28);
when 5 =>
        temp:=interm2(26 downto 0) & interm2(31 downto 27);
when 6 =>
        temp:=interm2(25 downto 0) & interm2(31 downto 26);
when 7  =>
        temp:=interm2(24 downto 0) & interm2(31 downto 25);
when 8 =>
        temp:=interm2(23 downto 0) & interm2(31 downto 24);
when 9 =>
        temp:=interm2(22 downto 0) & interm2(31 downto 23);
when 10 =>
        temp:=interm2(21 downto 0) & interm2(31 downto 22);
when 11 =>
        temp:=interm2(20 downto 0) & interm2(31 downto 21);
when 12 =>
        temp:=interm2(19 downto 0) & interm2(31 downto 20);
when 13 =>
        temp:=interm2(18 downto 0) & interm2(31 downto 19);
when 14 =>
        temp:=interm2(17 downto 0) & interm2(31 downto 18);
when 15 =>
        temp:=interm2(16 downto 0) & interm2(31 downto 17);
when 16 =>
        temp:=interm2(15 downto 0) & interm2(31 downto 16);
when 17 =>
        temp:=interm2(14 downto 0) & interm2(31 downto 15);
when 18 =>
        temp:=interm2(13 downto 0) & interm2(31 downto 14);
when 19 =>
        temp:=interm2(12 downto 0) & interm2(31 downto 13);
when 20 =>
        temp:=interm2(11 downto 0) & interm2(31 downto 12);
when 21 =>
        temp:=interm2(10 downto 0) & interm2(31 downto 11);
when 22  =>
        temp:=interm2(9 downto 0) & interm2(31 downto 10);
when 23 =>
        temp:=interm2(8 downto 0) & interm2(31 downto 9);
```

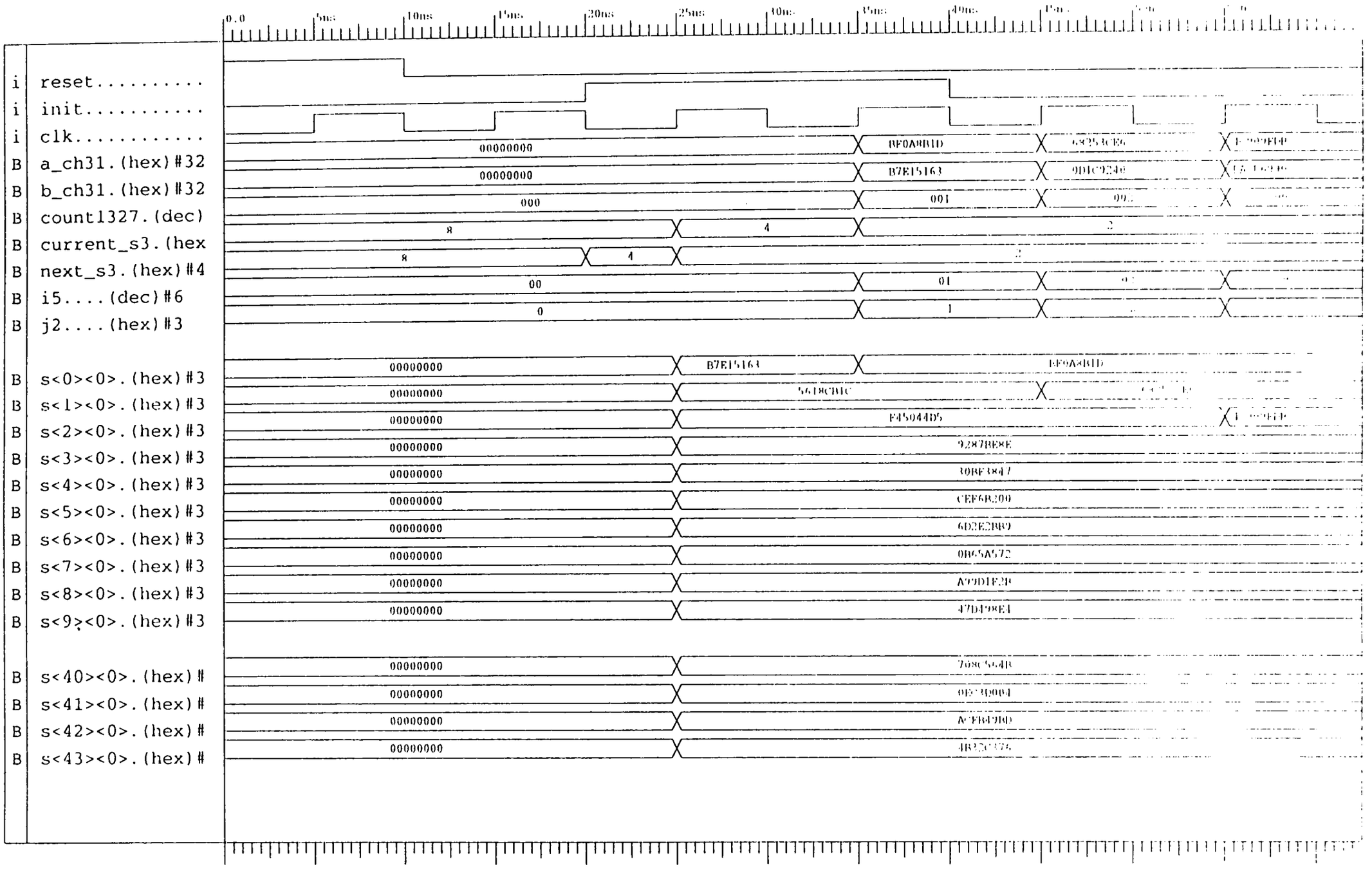
```
                when 24 =>
                        temp:=interm2(7 downto 0) & interm2(31 downto 8);
                when 25 =>
                        temp:=interm2(6 downto 0) & interm2(31 downto 7);
                when 26 =>
                        temp:=interm2(5 downto 0) & interm2(31 downto 6);
                when 27 =>
                        temp:=interm2(4 downto 0) & interm2(31 downto 5);
                when 28 =>
                        temp:=interm2(3 downto 0) & interm2(31 downto 4);
                when 29 =>
                        temp:=interm2(2 downto 0) & interm2(31 downto 3);
                when 30 =>
                        temp:=interm2(1 downto 0) & interm2(31 downto 2);
                when 31 =>
                        temp:=interm2(0) & interm2(31 downto 1);
                end case;
                b_ch<=temp;
                l(j)<=temp;
                if i+1>43 then
                        i<=i-43; --   i<=i+1-44; adica i<=(i+1) mod 44
                else
                        i<=i+1;
                end if;
                j<=(j+1) mod 8;
          end if;
        end if;
        end process;

        final_p : process(next_s, s)
        begin
          if (clk'event and clk='1') then
              if  next_s=final then
                  s0<=s(0);
              end if;
          end if;
        end process;
end gen_chei_arch;
```

Anexa 7

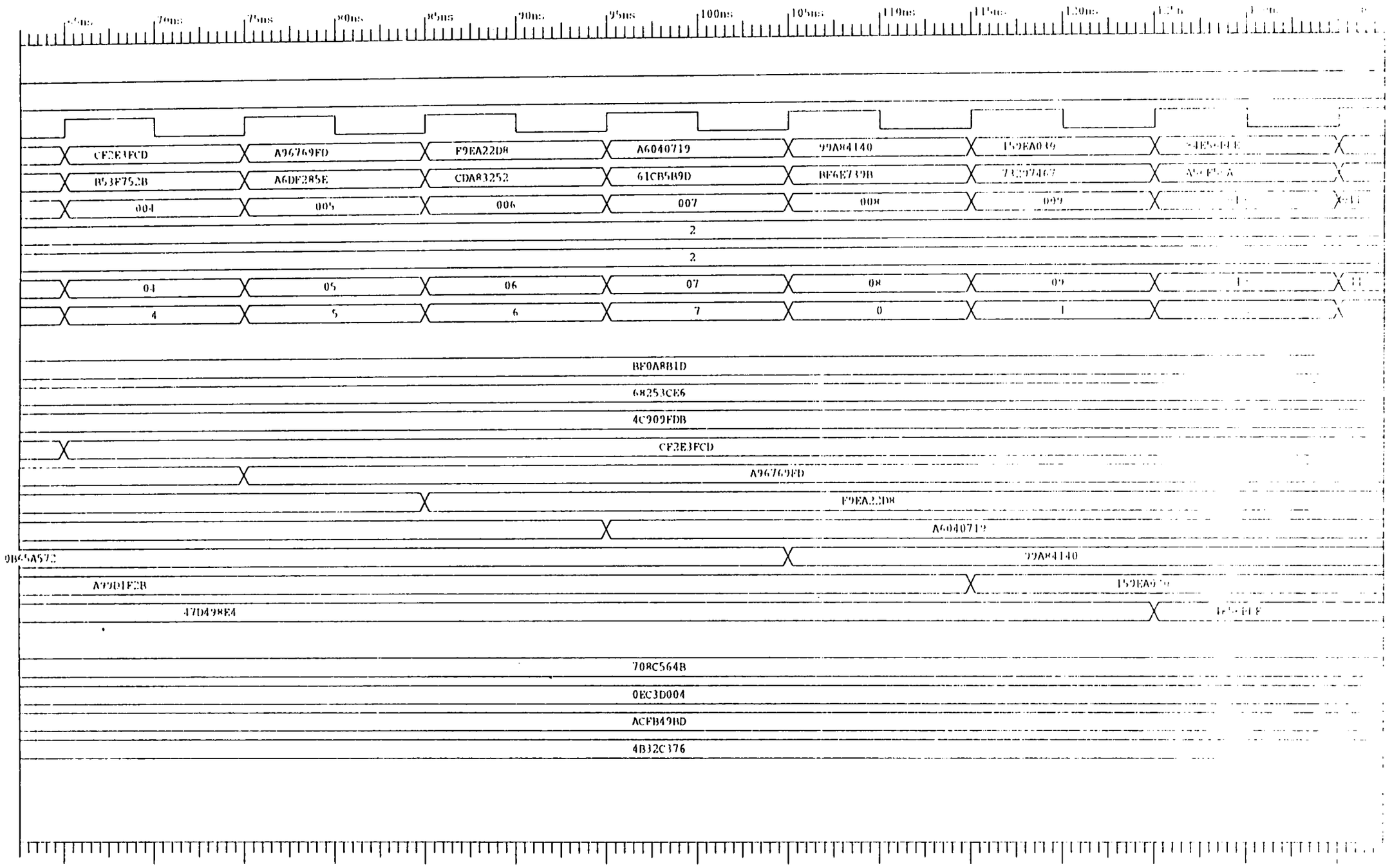
Diagrama de timp pentru generarea cheilor

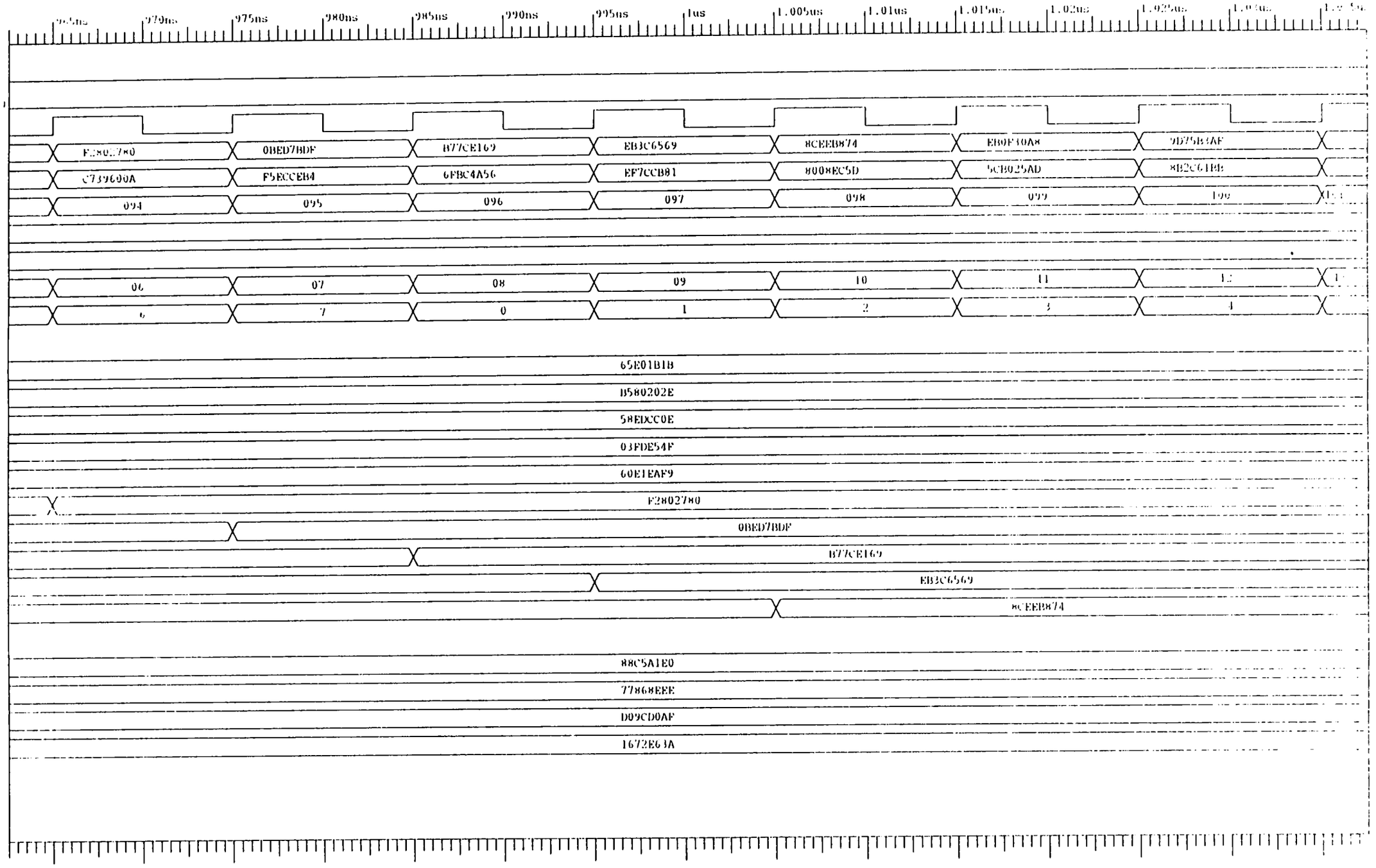

0.0
50ns
100ns
150ns
200ns
250ns
300ns
350ns
400ns
450ns
500ns
i reset.....
i init.....
i clk.....
B a_ch31.(hex)#32
B b_ch31.(hex)#32
B count1327.(dec)
B current_s3.(hex
B next_s3.(hex)#4
B i5....(dec)#6
B j2....(hex)#3
B s<0><0>.(hex)#3
B s<1><0>.(hex)#3
B s<2><0>.(hex)#3
B s<3><0>.(hex)#3
B s<4><0>.(hex)#3
B s<5><0>.(hex)#3
B s<6><0>.(hex)#3
B s<7><0>.(hex)#3
B s<8><0>.(hex)#3
B s<9><0>.(hex)#3
B s<40><0>.(hex)#
B s<41><0>.(hex)#
B s<42><0>.(hex)#
B s<43><0>.(hex)#
00000000
00000000
000
8
3
8
3
00
0
BF0A8B1D
68253CE6
4C909FDB
9287BE8E
CF2E3FCD
30BF3847
A96769FD
CEF6B200
F9EA22D8
6D2E2BB9
A6040719
0B65A572
99A84140
A99D1F2B
159EA030
47D498E4
84E56BDE
708C564B
0EC3D004
ACFB49BD
4B32C476

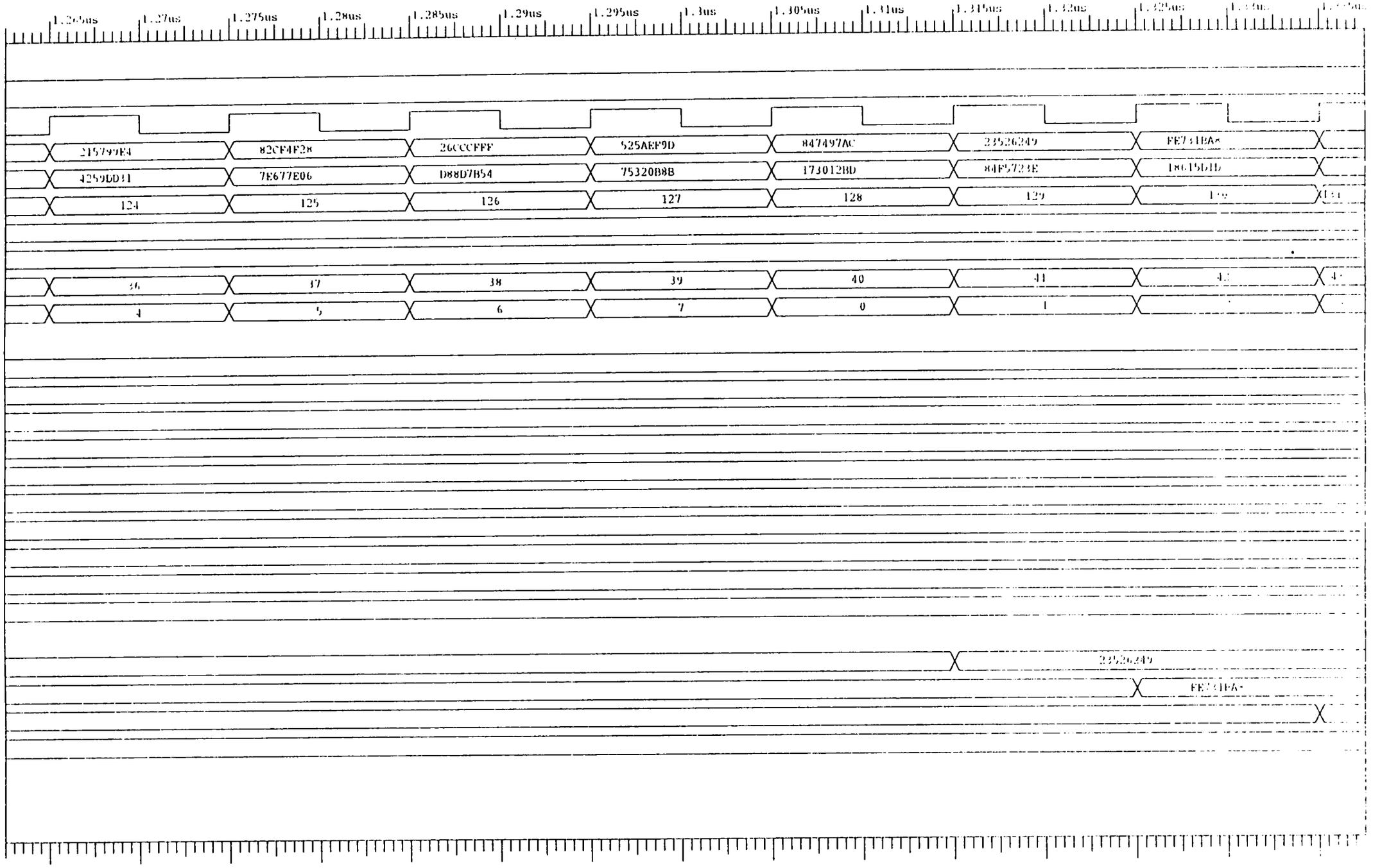

650ns
700ns
750ns
800ns
850ns
900ns
950ns
1us
1.05us
1.1us
1.15us
1.2us
1.25us
65E01B1B
B580202E
58EDCC0E
03FDE54F
60E1EAF9
F2802780
0BED7BDF
B77CE169
EB3C6569
8CEEB874
88C5A1E0
77868EEE
D09CD0AF

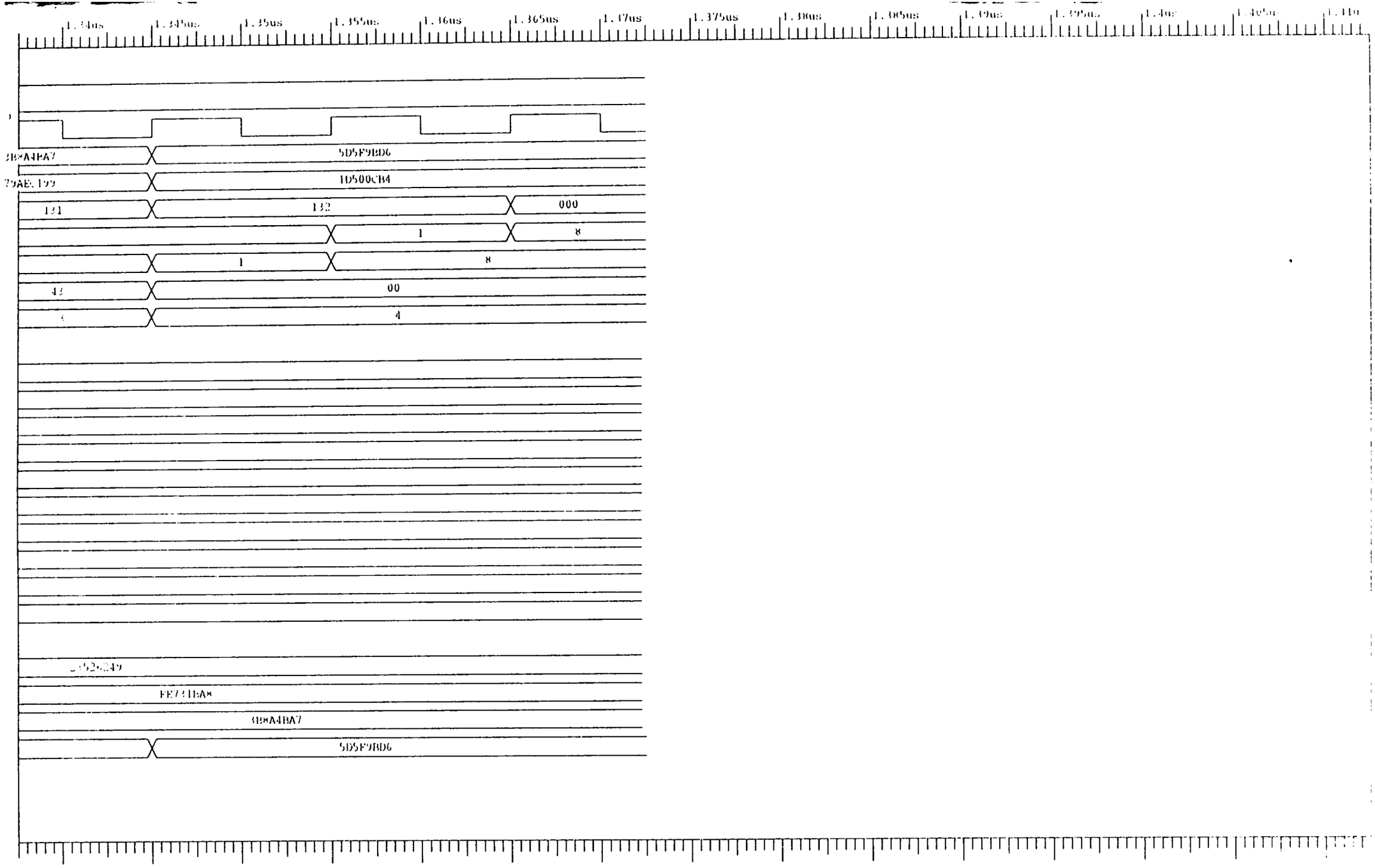

0.0
5ns
10ns
15ns
20ns
25ns
30ns
35ns
40ns
i reset.....
i init.....
i clk.....
B a_ch31.(hex)#32
B b_ch31.(hex)#32
B count1327.(dec)
B current_s3.(hex
B next_s3.(hex)#4
B i5....(dec)#6
B j2....(hex)#3
B s<0><0>.(hex)#3
B s<1><0>.(hex)#3
B s<2><0>.(hex)#3
B s<3><0>.(hex)#3
B s<4><0>.(hex)#3
B s<5><0>.(hex)#3
B s<6><0>.(hex)#3
B s<7><0>.(hex)#3
B s<8><0>.(hex)#3
B s<9><0>.(hex)#3
B s<40><0>.(hex)#
B s<41><0>.(hex)#
B s<42><0>.(hex)#
B s<43><0>.(hex)#
00000000
BF0A8B1D
00000000
B7E15163
000
001
8
4
2
8
4
2
00
01
0
1
00000000
B7E15163
BF0A8B1D
00000000
5618CB1C
00000000
F45044D5
00000000
9287BE8E
00000000
30BF3847
00000000
CEF6B200
00000000
6D2E2BD9
00000000
0B65A572
00000000
A79D1F2B
00000000
47D479E4
00000000
00000000
00000000
00000000
00000000

BUPT


CF2E3FCD
A96769FD
F9EA22D8
A6040719
99A84140
B53F752B
A6DF285E
CDA83252
61CB5B9D
004
005
006
007
008
2
2
04
05
06
07
08
4
5
6
7
0
BF0A8B1D
68253CE6
4C909FDB
CF2E3FCD
A96769FD
F9EA22D8
A6040719
99A84140
A79D1F2B
47D498E4
708C564B
0EC3D004
ACFB49BD
4B32C376

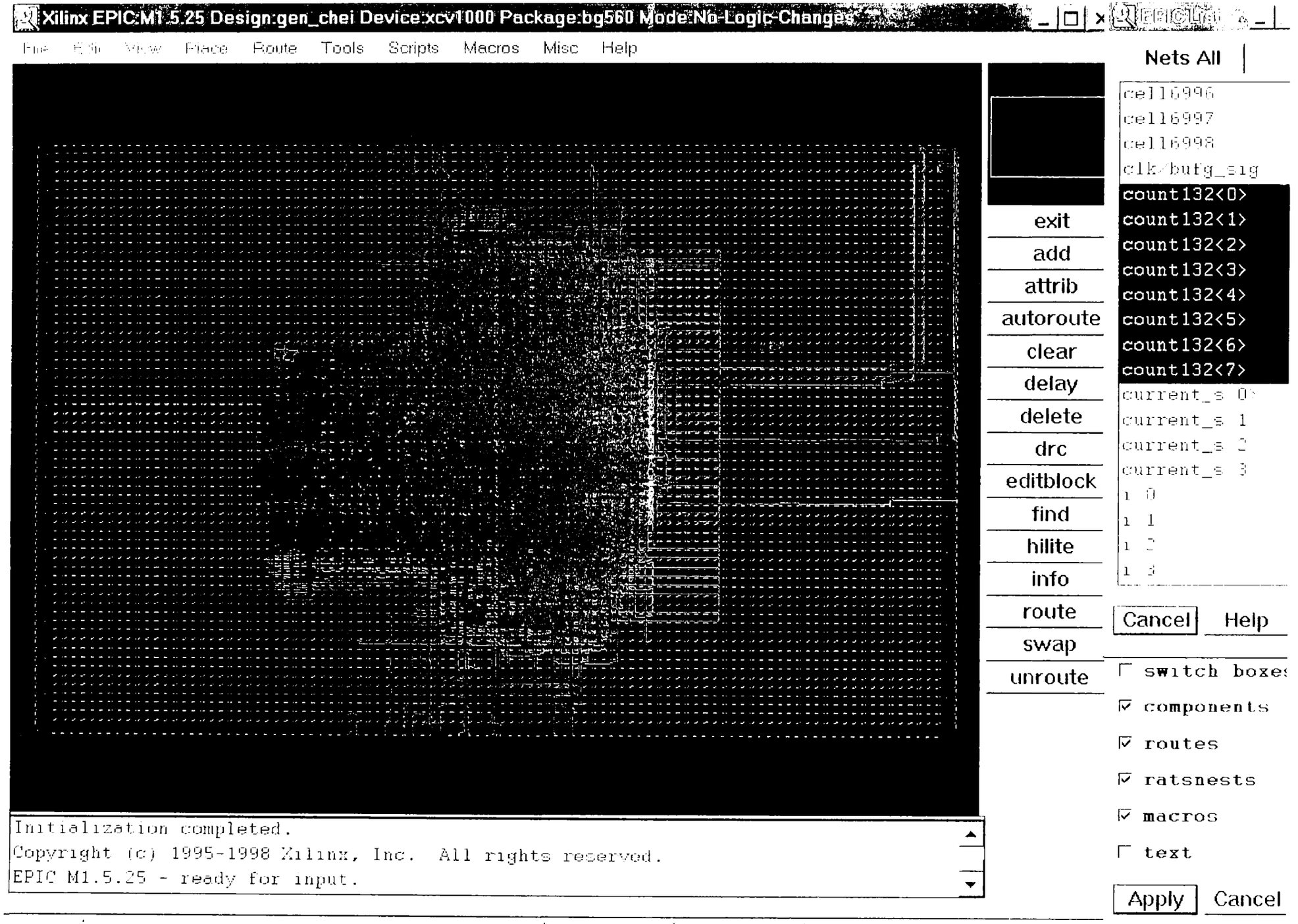

970ns
975ns
980ns
985ns
990ns
995ns
1us
1.005us
1.01us
1.015us
1.02us
1.025us
F2802780
0BED7BDF
B77CE169
EB3C6569
8CEEB874
EB0F30A8
9D75B7AF
C739600A
F5ECCEB4
6FBC4A56
EF7CCB81
8008EC5D
5CB025AD
8B2C61BB
094
095
096
097
098
099
100
06
07
08
09
10
11
12
6
7
0
1
2
3
4
65E01B1B
B580202E
58EDCC0E
03FDE54F
60E1EAF9
F2802780
0BED7BDF
B77CE169
EB3C6569
8CEEB874
88C5A1E0
77868EEE
D09CD0AF
1672E63A

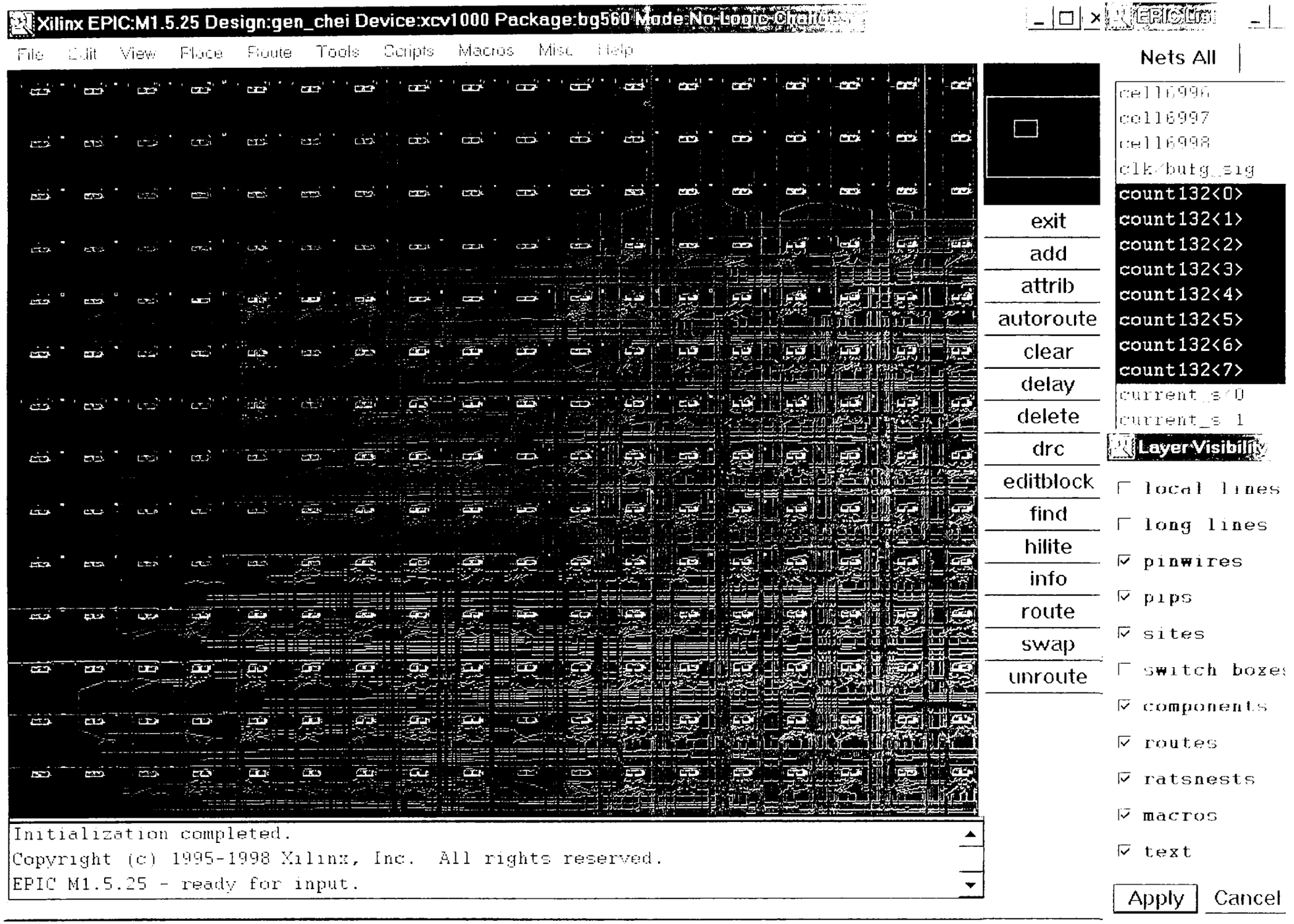

1.265us
1.27us
1.275us
1.28us
1.285us
1.29us
1.295us
1.3us
1.305us
1.31us
1.315us
1.32us
1.325us
215799E4
82CF4F28
26CCCFFF
525AEF9D
847497AC
23526249
4259DD11
7E677E06
D88D7B54
75320B8B
17301281
84F5724E
124
125
126
127
128
129
36
37
38
39
40
41
4
5
6
7
0
1
23526249

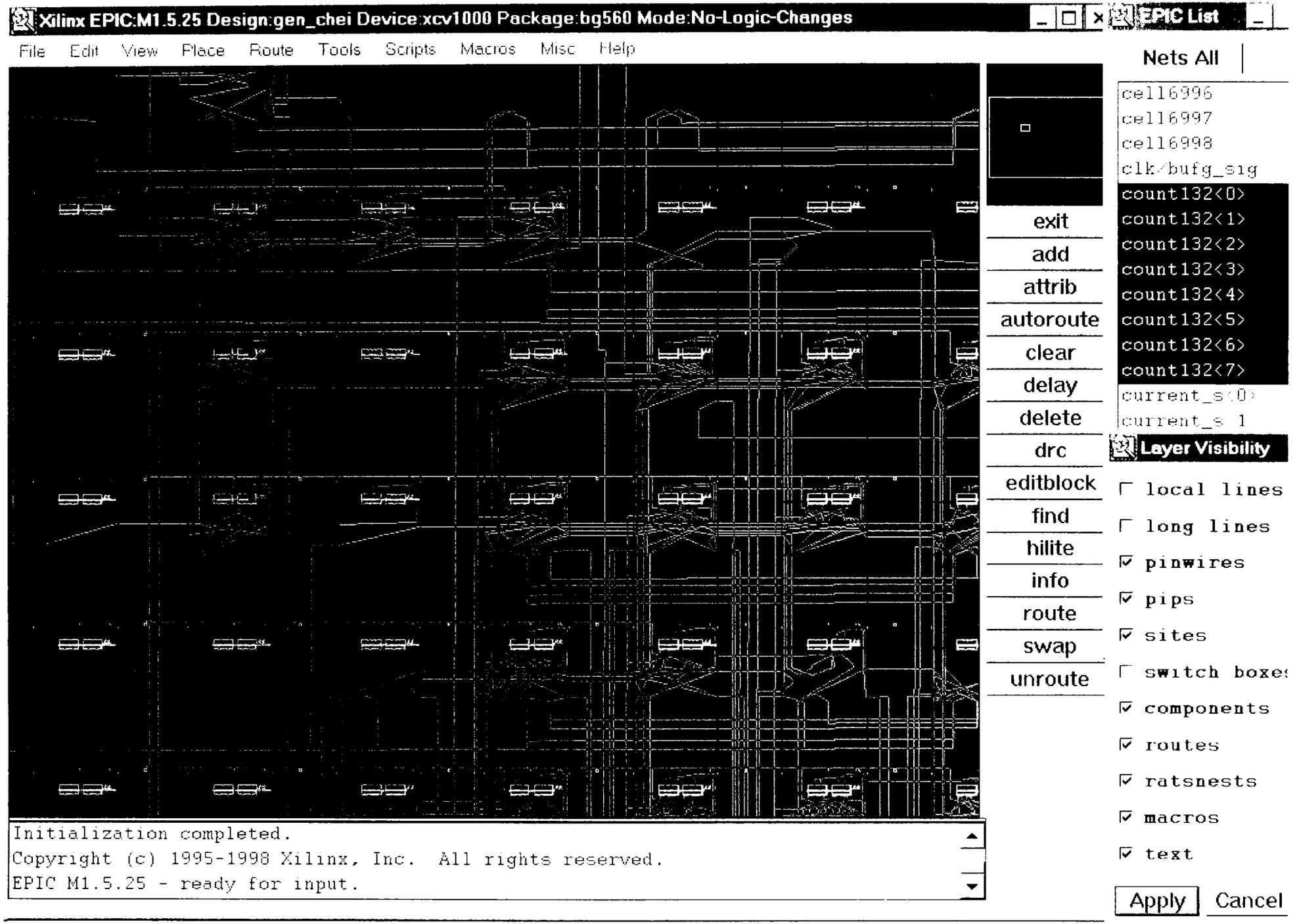

1.34us
1.345us
1.35us
1.355us
1.36us
1.365us
1.37us
1.375us
1.38us
1.385us
1.39us
1.395us
1.4us
1.405us
5D5F9BD6
1D500CB4
000
8
00
4
FE741BA8
3B8A4BA7
5D5F9BD6

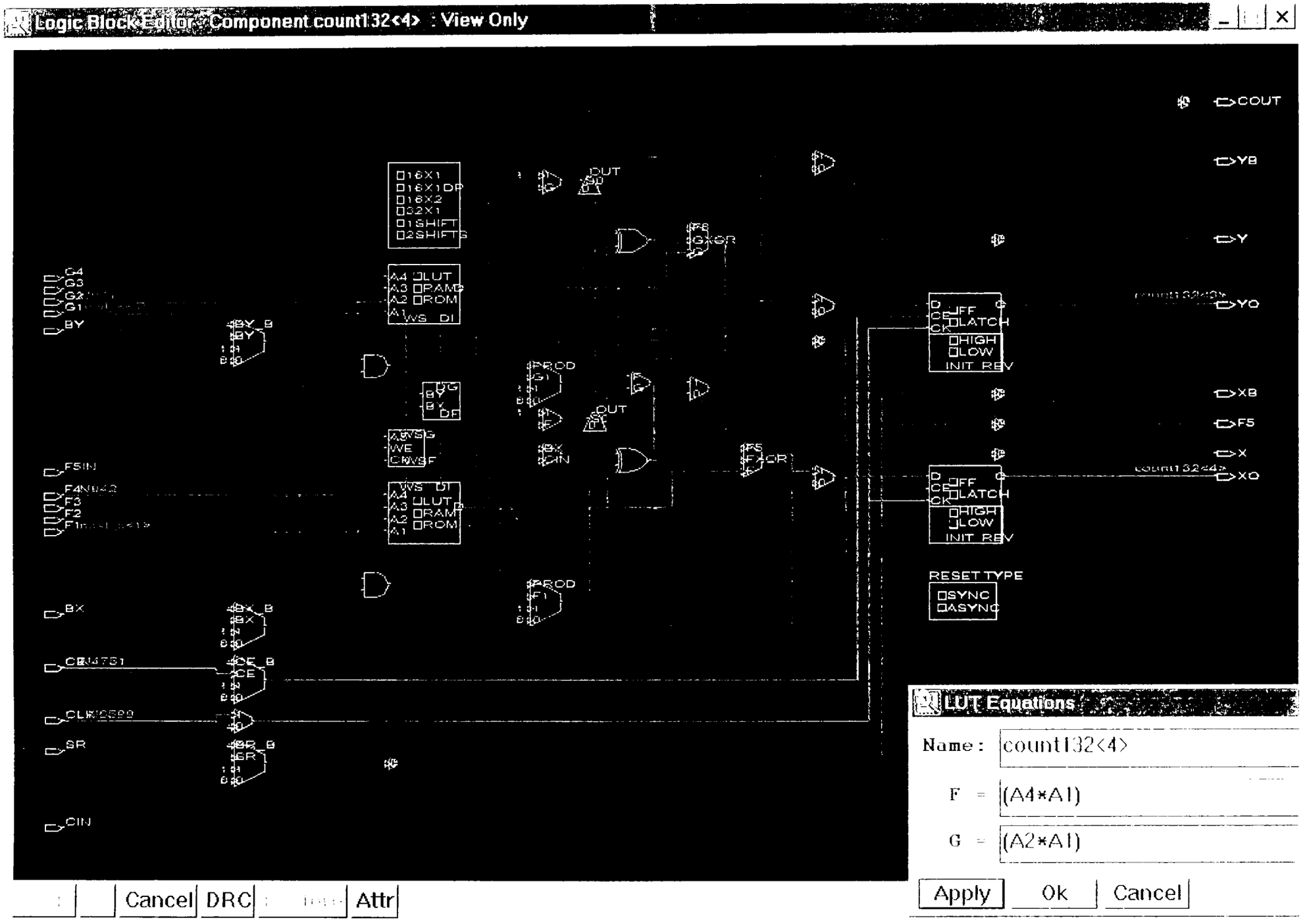
Anexa 8

Harta distribuției circuitelor din modulul de generare al cheilor


Xilinx EPIC:M1.5.25 Design:gen_chei Device:xcv1000 Package:bg560 Mode:No-Logic-Changes
File Edit View Place Route Tools Scripts Macros Misc Help
exit
add
attrib
autoroute
clear
delay
delete
drc
editblock
find
hilite
info
route
swap
unroute
Nets All
cell6996
cell6997
cell6998
clk/bufg_sig
count132<0>
count132<1>
count132<2>
count132<3>
count132<4>
count132<5>
count132<6>
count132<7>
current_s 0
current_s 1
current_s 2
current_s 3
1 0
1 1
1 2
1 3
Cancel
Help
switch boxes
components
routes
ratsnests
macros
text
Apply
Cancel
Initialization completed.
Copyright (c) 1995-1998 Xilinx, Inc. All rights reserved.
EPIC M1.5.25 - ready for input.


Xilinx EPIC:M1.5.25 Design:gen_chei Device:xcv1000 Package:bg560 Mode:No-Logic-Changes
File Edit View Place Route Tools Scripts Macros Misc Help
exit
add
attrib
autoroute
clear
delay
delete
drc
editblock
find
hilite
info
route
swap
unroute
Nets All
cell6996
cell6997
cell6998
clk/butg_sig
count132<0>
count132<1>
count132<2>
count132<3>
count132<4>
count132<5>
count132<6>
count132<7>
current_s<0
current_s<1
Layer Visibility
local lines
long lines
pinwires
pips
sites
switch boxes
components
routes
ratsnests
macros
text
Apply
Cancel
Initialization completed.
Copyright (c) 1995-1998 Xilinx, Inc. All rights reserved.
EPIC M1.5.25 - ready for input.


Xilinx EPIC:M1.5.25 Design:gen_chei Device:xcv1000 Package:bg560 Mode:No-Logic-Changes
File
Edit
View
Place
Route
Tools
Scripts
Macros
Misc
Help
exit
add
attrib
autoroute
clear
delay
delete
drc
editblock
find
hilite
info
route
swap
unroute
Initialization completed.
Copyright (c) 1995-1998 Xilinx, Inc. All rights reserved.
EPIC M1.5.25 - ready for input.
EPIC List
Nets All
cell6996
cell6997
cell6998
clk/bufg_sig
count132<0>
count132<1>
count132<2>
count132<3>
count132<4>
count132<5>
count132<6>
count132<7>
current_s<0>
current_s 1
Layer Visibility
local lines
long lines
pinwires
pips
sites
switch boxes
components
routes
ratsnests
macros
text
Apply
Cancel


Logic Block Editor - Component count132<4> : View Only
COUT
YB
Y
YO
XB
F5
X
XO
RESET TYPE
SYNC
ASYNC
LUT Equations
Name: count132<4>
F = (A4*A1)
G = (A2*A1)
Apply
Ok
Cancel
Cancel
DRC
Attr

Anexa 9

Implementarea hardware în VHDL a modulului de decriptare din algoritmul RC6

```
--definirea entitatii care realizeaza decriptarea.
--aceasta entitate apeleaza componenta de decriptare pentru o runda m_decript

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity decript is
   port (
        reset:          in std_logic;
        clk:            in std_logic;
        init:           in std_logic;
        text:           inout std_logic_vector (127 downto 0)
         );
end decript;

architecture decript_arch of decript is
      type states is (asteapta, gen_subchei, apel_decriptare, runda_noua, final);
      signal current_s, next_s:                 states;
      signal count20:                           integer range 0 to 20;
      type tabel is array (43 downto 0) of std_logic_vector(31 downto 0);
      signal s:                                 tabel;
      signal decrip:                            std_logic;
      signal a_in1, b_in1, c_in1, d_in1:        std_logic_vector(31 downto 0);
      signal cheie1, cheie2:                    std_logic_vector(31 downto 0);
      signal a_out1, b_out1, c_out1, d_out1: std_logic_vector(31 downto 0);
      signal rezolvat:                          std_logic;

component m_decript
   port (
        reset:                  in std_logic;
        clk:                    in std_logic;
        decrip:                 in std_logic;
        a_in:                   in std_logic_vector(31 downto 0);
        b_in:                   in std_logic_vector(31 downto 0);
        c_in:                   in std_logic_vector(31 downto 0);
        d_in:                   in std_logic_vector(31 downto 0);
        s0:                     in std_logic_vector(31 downto 0);
        s1:                     in std_logic_vector(31 downto 0);
        a_out:                  out std_logic_vector(31 downto 0);
        b_out:                  out std_logic_vector(31 downto 0);
        c_out:                  out std_logic_vector(31 downto 0);
        d_out:                  out std_logic_vector(31 downto 0);
        rezolvat_d:             out std_logic
         );
end component;

begin
-- declararea si realizarea corespondentei intre modulul de decriptare si entitatea principala
```

```
comp1: m_decript
port map (reset, clk, decrip, a_in1, b_in1, c_in1, d_in1, cheie1, cheie2, a_out1,
          b_out1,  c_out1,  d_out1, rezolvat);

syncronous_p: process(reset, clk, next_s)
 begin
     if reset='1' then
          current_s<=asteapta;
     elsif (clk'event and clk='1') then
          current_s<= next_s;
     end if;
 end process;

async_p: process (current_s, init, count20, rezolvat, clk)
 begin
  case current_s is
      when asteapta =>
          if init='1' then
                    next_s <=gen_subchei;
          else
                    next_s <=asteapta;
           end if;
       when gen_subchei =>
                    next_s <=apel_decriptare;
      when apel_decriptare =>
                    next_s <=runda_noua;
      when runda_noua =>
          if rezolvat='1' then
                    if count20 = 20  then
                              next_s <=final;
                    else
                              next_s <=apel_decriptare;
                    end if;
           else
                    next_s<=runda_noua;
          end if;
      when final =>
                    next_s <=asteapta;
  end case;
 end process;

 generarea_cheilor_p: process(next_s, clk)
 begin
   if (clk'event and clk='1') then
      if  next_s=gen_subchei then
--din motive ce tin de capacitatea placii, cheile sunt date implicit
            s(0)<="01100101111000000001101100011011";
            s(1)<="10110101100000000010000000101110";
            s(2)<="01011000111011011100110000001110";
            s(3)<="00000011111111011110010101001111";
            s(4)<="01100000111000011110101011111001";
```

```
            s(5)<="11110010100000000010011110000000";
            s(6)<="00001011111011010111101111011111";
            s(7)<="10110111011111001110000101101001";
            s(8)<="11101011001111000110010101101001";
            s(9)<="10001100111011101011100001110100";
            s(10)<="11101011000011110011000010101000";
            s(11)<="10011101011101011011001110101111";
            s(12)<="10100110100111001011111111011000";
            s(13)<="10111111000011000110110011111010";
            s(14)<="01000111111111111011100111000101";
            s(15)<="00111001000100100010000001011010";
            s(16)<="00000010010001011000010011011000";
            s(17)<="11001101100110010101100100111001";
            s(18)<="01010011011010000110101001101010";
            s(19)<="11010000001111111100110111001001";
            s(20)<="00100011000011101111010100010011";
            s(21)<="10010001000111111111011000010110";
            s(22)<="01110111001100100111001110101011";
            s(23)<="11001110100001000101011110101100";
            s(24)<="11001100011101011110001011010011";
            s(25)<="01000010011100000001101011101101";
            s(26)<="10001100010010010100001000110011";
            s(27)<="00100110111010000110110001111010";
            s(28)<="00011101011010001011110111011000";
            s(29)<="00111000011011011000011010110011";
            s(30)<="10101011000000100111000101111001";
            s(31)<="10100010001011100000000000010001";
            s(32)<="00101011000100101000011000100101";
            s(33)<="10001111000000000111110010101110";
            s(34)<="10000011110111111011010010111010";
            s(35)<="00100001010101111001100111100100";
            s(36)<="10000010110011111010011110010100";
            s(37)<="00100110110011001100111111111111";
            s(38)<="01010010010110101110111110011101";
            s(39)<="10000100011101001001011110101100";
            s(40)<="00100011010100100110001001001001";
            s(41)<="11111110011100110001101110101000";
            s(42)<="00111011100010100100101110100111";
            s(43)<="01011101010111111001101111010110";
  end if;
end if;
end process;

decrip_signal_p: process (clk, next_s)
begin
  if clk'event and clk='1' then
            if next_s=apel_decriptare then
                        decrip<='1';
            else
                        decrip<='0';
            end if;
```

```
  end if;
end process;

decriptare_p: process (clk, current_s, next_s, text, s, a_out1, b_out1, c_out1,
                       d_out1, count20)
variable a_in:          std_logic_vector(31 downto 0);
variable b_in:          std_logic_vector(31 downto 0);
variable c_in:          std_logic_vector(31 downto 0);
variable d_in:          std_logic_vector(31 downto 0);
variable c0, c1:        integer range 0 to 43;
begin
 if clk'event and clk='1' then
   if current_s=gen_subchei and next_s=apel_decriptare then
        a_in:=text(103 downto 96) & text(111 downto 104)
                       & text(119 downto 112) & text(127 downto 120);
        b_in:=text(71 downto 64) & text(79 downto 72)
                       & text (87 downto 80) & text(95 downto 88);
        c_in:=text(39 downto 32) & text(47 downto 40)
                       & text (55 downto 48) & text(63 downto 56);
        d_in:=text(7 downto 0) & text(15 downto 8)
                       & text (23 downto 16) & text(31 downto 24);
   a_in1<=a_in-s(42);
        b_in1<=b_in;
        c_in1<=c_in-s(43);
        d_in1<=d_in;
        cheie1<=s(40);
        cheie2<=s(41);
   elsif next_s=apel_decriptare then
        c0:=2*(20-count20);
        c1:=c0+1;
        a_in1<=a_out1;
        b_in1<=b_out1;
        c_in1<=c_out1;
        d_in1<=d_out1;
        cheie1<=s(c0);
        cheie2<=s(c1);
  end if;
 end if;
end process;

counter20_p: process(clk, next_s)
begin
      if (clk'event and clk='1') then
         if next_s=apel_decriptare then
               count20<=count20+1;
         elsif next_s=gen_subchei then
               count20<=0;
         end if;
       end if;
end process;
```

```
    final_p: process(clk, next_s, a_out1, b_out1, c_out1, d_out1)
    variable a_out, b_out, c_out, d_out:   std_logic_vector(31 downto 0);
    begin
      if (clk'event and clk='1') then
          if next_s=final then
             a_out:=a_out1;
             b_out:=b_out1-s(0);
             c_out:=c_out1;
             d_out:=d_out1-s(1);
             text(127 downto 96)<=a_out(7 downto 0) & a_out(15 downto 8)
                          & a_out(23 downto 16) & a_out(31 downto 24);
             text(95 downto 64)<=b_out(7 downto 0) & b_out(15 downto 8)
                          & b_out(23 downto 16) & b_out(31 downto 24);
             text(63 downto 32)<=c_out(7 downto 0) & c_out(15 downto 8)
                          & c_out(23 downto 16) & c_out(31 downto 24);
             text(31 downto 0)<=d_out(7 downto 0) & d_out(15 downto 8)
                          & d_out(23 downto 16) & d_out(31 downto 24);
          end if;
      end if;
    end process;
end decript_arch;
```

Modulul de decriptare pentru o rundă

```
--varianta adaptata la functionarea ca si componenta a entitatii de decriptare
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
entity alg_dec is
   port (
        reset:                  in std_logic;
        clk:                    in std_logic;
        decrip:                 in std_logic;
        a_in:                   in std_logic_vector(31 downto 0);
        b_in:                   in std_logic_vector(31 downto 0);
        c_in:                   in std_logic_vector(31 downto 0);
        d_in:                   in std_logic_vector(31 downto 0);
        s0:                     in std_logic_vector(31 downto 0);
        s1:                     in std_logic_vector(31 downto 0);
        a_out:                  out std_logic_vector(31 downto 0);
        b_out:                  out std_logic_vector(31 downto 0);
        c_out:                  out std_logic_vector(31 downto 0);
        d_out:                  out std_logic_vector(31 downto 0);
        rezolvat_d:             out std_logic
         );
```

```
end alg_dec;
architecture alg_dec_arch of alg_dec is
        type states is (asteapta, decripare_interior, decripare_final);
        signal current_s, next_s:       states;
--internal  registers
        signal a:                       std_logic_vector(31 downto 0);
        signal b:                       std_logic_vector(31 downto 0);
        signal c:                       std_logic_vector(31 downto 0);
        signal d:                       std_logic_vector(31 downto 0);
        type vec is array (31 downto 0) of std_logic_vector(31 downto 0);
begin

        async_p: process (current_s, decrip, clk)
        begin
          case current_s is
             when asteapta =>
                 if decrip='1' then
                         next_s <=decripare_interior;
                 else
                         next_s <=asteapta;
                 end if;
             when decripare_interior =>
                         next_s <=decripare_final;
             when decripare_final =>
                         next_s <=asteapta;
          end case;
        end process;

        syncronous_p: process(reset, clk, next_s)
        begin
          if reset='1' then
                  current_s<=asteapta;
          elsif (clk'event and clk='1') then
                  current_s<= next_s;
          end if;
        end process;

        decripare_p:process(current_s, next_s, clk, s0, s1, a_in, b_in, c_in, d_in)
        variable tempa   :                      std_logic_vector(31 downto 0);
        variable tempc   :                      std_logic_vector(31 downto 0);
        variable va:                            vec;
        variable vc:                            vec;
        variable va0, va1, va2,  va3, va4, va5:     std_logic_vector(31 downto 0);
        variable va6, va7, va8, va9, va10, va11:    std_logic_vector(31 downto 0);
        variable va12, va13,  va14, va15, va16:     std_logic_vector(31 downto 0);
        variable va17, va18, va19, va20, va21:      std_logic_vector(31 downto 0);
        variable va22,  va23, va24, va25, va26:     std_logic_vector(31 downto 0);
        variable va27, va28, va29, va30:            std_logic_vector(31 downto 0);
        variable vc0, vc1, vc2,  vc3, vc4, vc5:     std_logic_vector(31 downto 0);
        variable vc6, vc7, vc8, vc9, vc10, vc11:    std_logic_vector(31 downto 0);
```

```
variable vc12, vc13,  vc14, vc15, vc16:      std_logic_vector(31 downto 0);
variable vc17, vc18, vc19, vc20, vc21:       std_logic_vector(31 downto 0);
variable vc22,  vc23, vc24, vc25, vc26:      std_logic_vector(31 downto 0);
variable vc27, vc28, vc29, vc30:             std_logic_vector(31 downto 0);
variable i, intu, intt:                      integer range 0 to 31;
variable t, u, difb, difd:                   std_logic_vector(31 downto 0);
begin
  if (clk'event and clk='1') then
     if next_s=decripare_interior then
      for i in 0 to 30 loop
          vc(i):= c_in(i)&c_in(i)&c_in(i)&c_in(i)&c_in(i)&c_in(i)&c_in(i)&c_in(i)
                  &c_in(i)&c_in(i)&c_in(i)&c_in(i)&c_in(i)&c_in(i)&c_in(i)&c_in(i)
                  &c_in(i)&c_in(i)&c_in(i)&c_in(i)&c_in(i)&c_in(i)&c_in(i)&c_in(i)
                  &c_in(i)&c_in(i)&c_in(i)&c_in(i)&c_in(i)&c_in(i)&c_in(i)&c_in(i);
          vc(i):=vc(i) and c_in;
          va(i):= a_in(i)&a_in(i)&a_in(i)&a_in(i)&a_in(i)&a_in(i)&a_in(i) &a_in(i)
                  &a_in(i)&a_in(i)&a_in(i)&a_in(i)&a_in(i)&a_in(i)&a_in(i)&a_in(i)
                  &a_in(i)&a_in(i)&a_in(i)&a_in(i)&a_in(i)&a_in(i)      &a_in(i)&a_in(i)
                  &a_in(i)&a_in(i)&a_in(i)&a_in(i)&a_in(i)&a_in(i)&a_in(i)&a_in(i);
          va(i):=va(i) and a_in;
  end loop;
  va0:=va(0);                                  va16:=va(16);
  va1:=va(1);                                  va17:=va(17);
  va2:=va(2);                                  va18:=va(18);
  va3:=va(3);                                  va19:=va(19);
  va4:=va(4);                                  va20:=va(20);
  va5:=va(5);                                  va21:=va(21);
  va6:=va(6);                                  va22:=va(22);
  va7:=va(7);                                  va23:=va(23);
  va8:=va(8);                                  va24:=va(24);
  va9:=va(9);                                  va25:=va(25);
  va10:=va(10);                                va26:=va(26);
  va11:=va(11);                                va27:=va(27);
  va12:=va(12);                                va28:=va(28);
  va13:=va(13);                                va29:=va(29);
  va14:=va(14);                                va30:=va(30);
  va15:=va(15);

  vc0:=vc(0);                                  vc11:=vc(11);
  vc1:=vc(1);                                  vc12:=vc(12);
  vc2:=vc(2);                                  vc13:=vc(13);
  vc3:=vc(3);                                  vc14:=vc(14);
  vc4:=vc(4);                                  vc15:=vc(15);
  vc5:=vc(5);                                  vc16:=vc(16);
  vc6:=vc(6);                                  vc17:=vc(17);
  vc7:=vc(7);                                  vc18:=vc(18);
  vc8:=vc(8);                                  vc19:=vc(19);
  vc9:=vc(9);                                  vc20:=vc(20);
  vc10:=vc(10);
```

```
vc21:=vc(21);
vc22:=vc(22);
vc23:=vc(23);
vc24:=vc(24);
vc25:=vc(25);
vc26:=vc(26);
vc27:=vc(27);
vc28:=vc(28);
vc29:=vc(29);
vc30:=vc(30);

tempa:=a_in+(va0(30 downto 0)&"0")+(va1(29 downto 0)&"00")
+(va2(28 downto 0)&"000")+(va3(27 downto 0)&"0000")
+(va4(26 downto 0)&"00000")+(va5(25 downto 0)&"000000")
+(va6(24 downto 0)&"0000000")+(va7(23 downto 0)&"00000000")
+(va8(22 downto 0)&"000000000")+(va9(21 downto 0)&"0000000000")
+(va10(20 downto 0)&"00000000000")+(va11(19 downto 0)&"000000000000")
+(va12(18 downto 0)&"0000000000000")
+(va13(17 downto 0)&"00000000000000")
+(va14(16 downto 0)&"000000000000000")
+(va15(15 downto 0)&"0000000000000000")
+(va16(14 downto 0)&"00000000000000000")
+(va17(13 downto 0)&"000000000000000000")
+(va18(12 downto 0)&"0000000000000000000")
+(va19(11 downto 0)&"00000000000000000000")
+(va20(10 downto 0)&"000000000000000000000")
+(va21(9 downto 0)&"0000000000000000000000")
+(va22(8 downto 0)&"00000000000000000000000")
+(va23(7 downto 0)&"000000000000000000000000")
+(va24(6 downto 0)&"0000000000000000000000000")
+(va25(5 downto 0)&"00000000000000000000000000")
+(va26(4 downto 0)&"000000000000000000000000000")
+(va27(3 downto 0)&"0000000000000000000000000000")
+(va28(2 downto 0)&"00000000000000000000000000000")
+(va29(1 downto 0)&"000000000000000000000000000000")
+(va30(0)&"0000000000000000000000000000000");
tempc:=c_in+(vc0(30 downto 0)&"0")+(vc1(29 downto 0)&"00")
+(vc2(28 downto 0)&"000")  +(vc3(27 downto 0)&"0000")
+(vc4(26 downto 0)&"00000")+(vc5(25 downto 0)&"000000")
+(vc6(24 downto 0)&"0000000")+(vc7(23 downto 0)&"00000000")
+(vc8(22 downto 0)&"000000000")+(vc9(21 downto 0)&"0000000000")
+(vc10(20 downto 0)&"00000000000")+(vc11(19 downto 0)&"000000000000")
+(vc12(18 downto 0)&"0000000000000")
+(vc13(17 downto 0)&"00000000000000")
+(vc14(16 downto 0)&"000000000000000")
+(vc15(15 downto 0)&"0000000000000000")
+(vc16(14 downto 0)&"00000000000000000")
+(vc17(13 downto 0)&"000000000000000000")
+(vc18(12 downto 0)&"0000000000000000000")
+(vc19(11 downto 0)&"00000000000000000000")
+(vc20(10 downto 0)&"000000000000000000000")
+(vc21(9 downto 0)&"0000000000000000000000")
+(vc22(8 downto 0)&"00000000000000000000000")
+(vc23(7 downto 0)&"000000000000000000000000")
+(vc24(6 downto 0)&"0000000000000000000000000")
+(vc25(5 downto 0)&"00000000000000000000000000")
```

```
+(vc26(4 downto 0)&"00000000000000000000000000000")
+(vc27(3 downto 0)&"00000000000000000000000000000")
+(vc28(2 downto 0)&"00000000000000000000000000000")
+(vc29(1 downto 0)&"00000000000000000000000000000")
+(vc30(0)&"00000000000000000000000000000");

t:=tempa(26 downto 0) &tempa(31 downto 27);
u:=tempc(26 downto 0) & tempc(31 downto 27);
difb:=b_in -s1;
difd:=d_in -s0;
intu:=conv_integer(u(4 downto 0));
intt:=conv_integer(t(4 downto 0));
        case intu is
                when 0 =>
                        a<=difd xor t;
                when 1 =>
                        a<=difd(0) & difd(31 downto 1) xor t;
                when 2 =>
                        a<=difd(1 downto 0) & difd(31 downto 2) xor t;
                when 3 =>
                        a<=difd(2 downto 0) & difd(31 downto 3) xor t;
                when 4 =>
                        a<=difd(3 downto 0) & difd(31 downto 4) xor t;
                when 5 =>
                        a<=difd(4 downto 0) & difd(31 downto 5) xor t;
                when 6 =>
                        a<=difd(5 downto 0) & difd(31 downto 6) xor t;
                when 7 =>
                        a<=difd(6 downto 0) & difd(31 downto 7) xor t;
                when 8 =>
                        a<=difd(7 downto 0) & difd(31 downto 8) xor t;
                when 9 =>
                        a<=difd(8 downto 0) & difd(31 downto 9) xor t;
                when 10 =>
                        a<=difd(9 downto 0) & difd(31 downto 10) xor t;
                when 11 =>
                        a<=difd(10 downto 0) & difd(31 downto 11) xor t;
                when 12 =>
                        a<=difd(11 downto 0) & difd(31 downto 12) xor t;
                when 13 =>
                        a<=difd(12 downto 0) & difd(31 downto 13) xor t;
                when 14 =>
                        a<=difd(13 downto 0) & difd(31 downto 14) xor t;
                when 15 =>
                        a<=difd(14 downto 0) & difd(31 downto 15) xor t;
                when 16 =>
                        a<=difd(15 downto 0) & difd(31 downto 16) xor t;
                when 17 =>
                        a<=difd(16 downto 0) & difd(31 downto 17) xor t;
                when 18 =>
                        a<=difd(17 downto 0) & difd(31 downto 18) xor t;
```

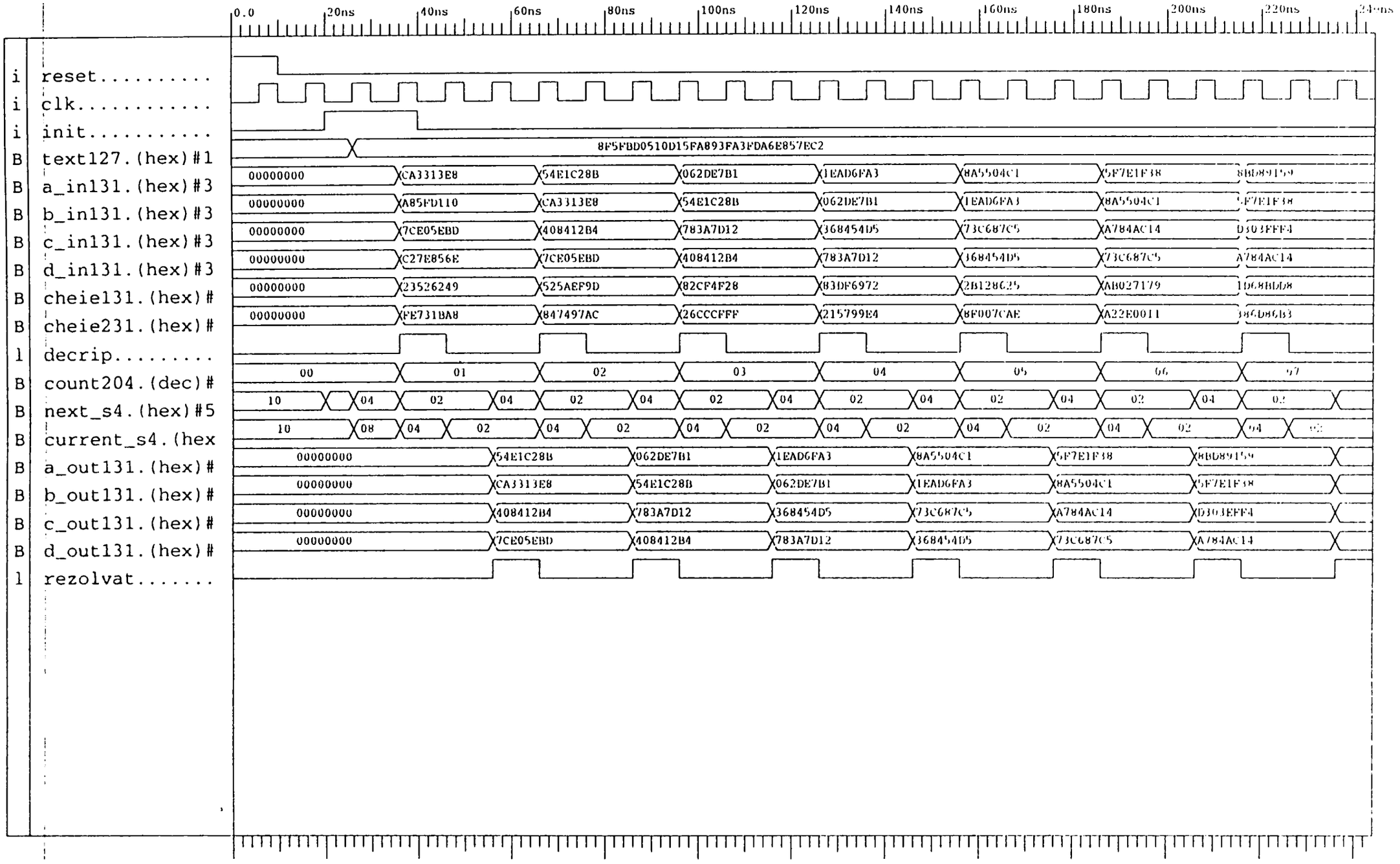
```
            when 19 =>
                a<=difd(18 downto 0) & difd(31 downto 19) xor t;
            when 20 =>
                a<=difd(19 downto 0) & difd(31 downto 20) xor t;
            when 21 =>
                a<=difd(20 downto 0) & difd(31 downto 21) xor t;
            when 22 =>
                a<=difd(21 downto 0) & difd(31 downto 22) xor t;
            when 23 =>
                a<=difd(22 downto 0) & difd(31 downto 23) xor t;
            when 24 =>
                a<=difd(23 downto 0) & difd(31 downto 24) xor t;
            when 25 =>
                a<=difd(24 downto 0) & difd(31 downto 25) xor t;
            when 26 =>
                a<=difd(25 downto 0) & difd(31 downto 26) xor t;
            when 27 =>
                a<=difd(26 downto 0) & difd(31 downto 27) xor t;
            when 28 =>
                a<=difd(27 downto 0) & difd(31 downto 28) xor t;
            when 29 =>
                a<=difd(28 downto 0) & difd(31 downto 29) xor t;
            when 30=>
                a<=difd(29 downto 0) & difd(31 downto 30) xor t;
            when 31 =>
                a<=difd(30 downto 0) & difd(31) xor t;
    end case;
    b<=a_in;
    case intt is
            when 0=>
                c<=difb xor u;
            when 1 =>
                c<=difb(0) & difb(31 downto 1) xor u;
            when 2 =>
                c<=difb(1 downto 0) & difb(31 downto 2) xor u;
            when 3 =>
                c<=difb(2 downto 0) & difb(31 downto 3) xor u;
            when 4 =>
                c<=difb(3 downto 0) & difb(31 downto 4) xor u;
            when 5 =>
                c<=difb(4 downto 0) & difb(31 downto 5) xor u;
            when 6 =>
                c<=difb(5 downto 0) & difb(31 downto 6) xor u;
            when 7 =>
                c<=difb(6 downto 0) & difb(31 downto 7) xor u;
            when 8 =>
                c<=difb(7 downto 0) & difb(31 downto 8) xor u;
            when 9 =>
                c<=difb(8 downto 0) & difb(31 downto 9) xor u;
            when 10 =>
                c<=difb(9 downto 0) & difb(31 downto 10) xor u;
```

```
                when 11 =>
                        c<=difb(10 downto 0) & difb(31 downto 11) xor u;
                when 12 =>
                        c<=difb(11 downto 0) & difb(31 downto 12) xor u;
                when 13 =>
                        c<=difb(12 downto 0) & difb(31 downto 13) xor u;
                when 14 =>
                        c<=difb(13 downto 0) & difb(31 downto 14) xor u;
                when 15 =>
                        c<=difb(14 downto 0) & difb(31 downto 15) xor u;
                when 16 =>
                        c<=difb(15 downto 0) & difb(31 downto 16) xor u;
                when 17 =>
                        c<=difb(16 downto 0) & difb(31 downto 17) xor u;
                when 18 =>
                        c<=difb(17 downto 0) & difb(31 downto 18) xor u;
                when 19 =>
                        c<=difb(18 downto 0) & difb(31 downto 19) xor u;
                when 20 =>
                        c<=difb(19 downto 0) & difb(31 downto 20) xor u;
                when 21 =>
                        c<=difb(20 downto 0) & difb(31 downto 21) xor u;
                when 22 =>
                        c<=difb(21 downto 0) & difb(31 downto 22) xor u;
                when 23 =>
                        c<=difb(22 downto 0) & difb(31 downto 23) xor u;
                when 24 =>
                        c<=difb(23 downto 0) & difb(31 downto 24) xor u;
                when 25 =>
                        c<=difb(24 downto 0) & difb(31 downto 25) xor u;
                when 26 =>
                        c<=difb(25 downto 0) & difb(31 downto 26) xor u;
                when 27 =>
                        c<=difb(26 downto 0) & difb(31 downto 27) xor u;
                when 28 =>
                        c<=difb(27 downto 0) & difb(31 downto 28) xor u;
                when 29 =>
                        c<=difb(28 downto 0) & difb(31 downto 29) xor u;
                when 30=>
                        c<=difb(29 downto 0) & difb(31 downto 30) xor u;
                when 31 =>
                        c<=difb(30 downto 0) & difb(31) xor u;
        end case;
        d<=c_in;
elsif next_s=decripare_final then
        a_out<=a;
        b_out<=b;
        c_out<=c;
        d_out<=d;
end if;
end if;
```

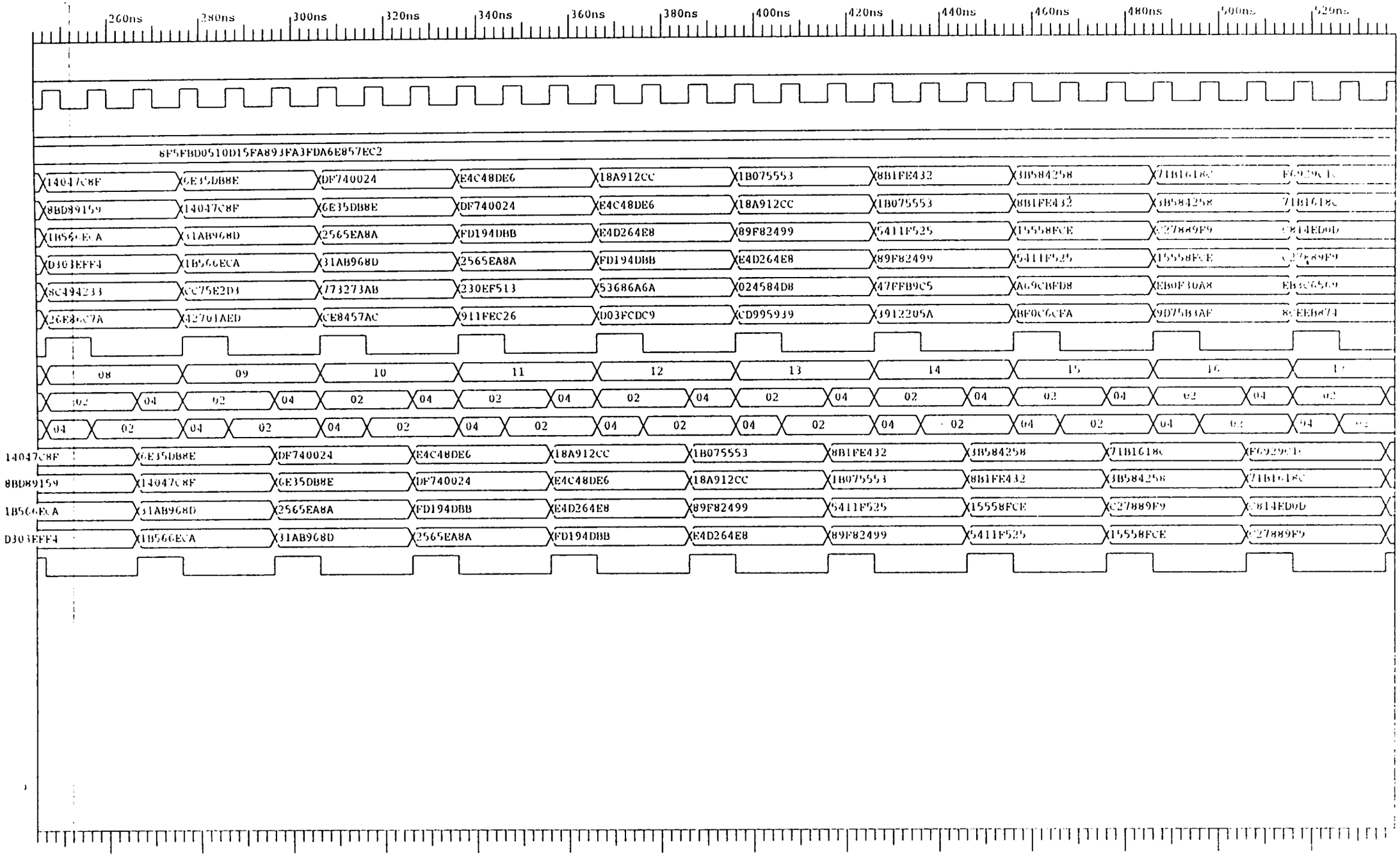
```
    end process;

    rezolvat_d_p: process(clk, next_s)
    begin
      if (clk'event and clk='1') then
            if next_s=decripare_final then
                  rezolvat_d<='1';
            else
                  rezolvat_d<='0';
            end if;
      end if;
    end process;
end alg_dec_arch;
```

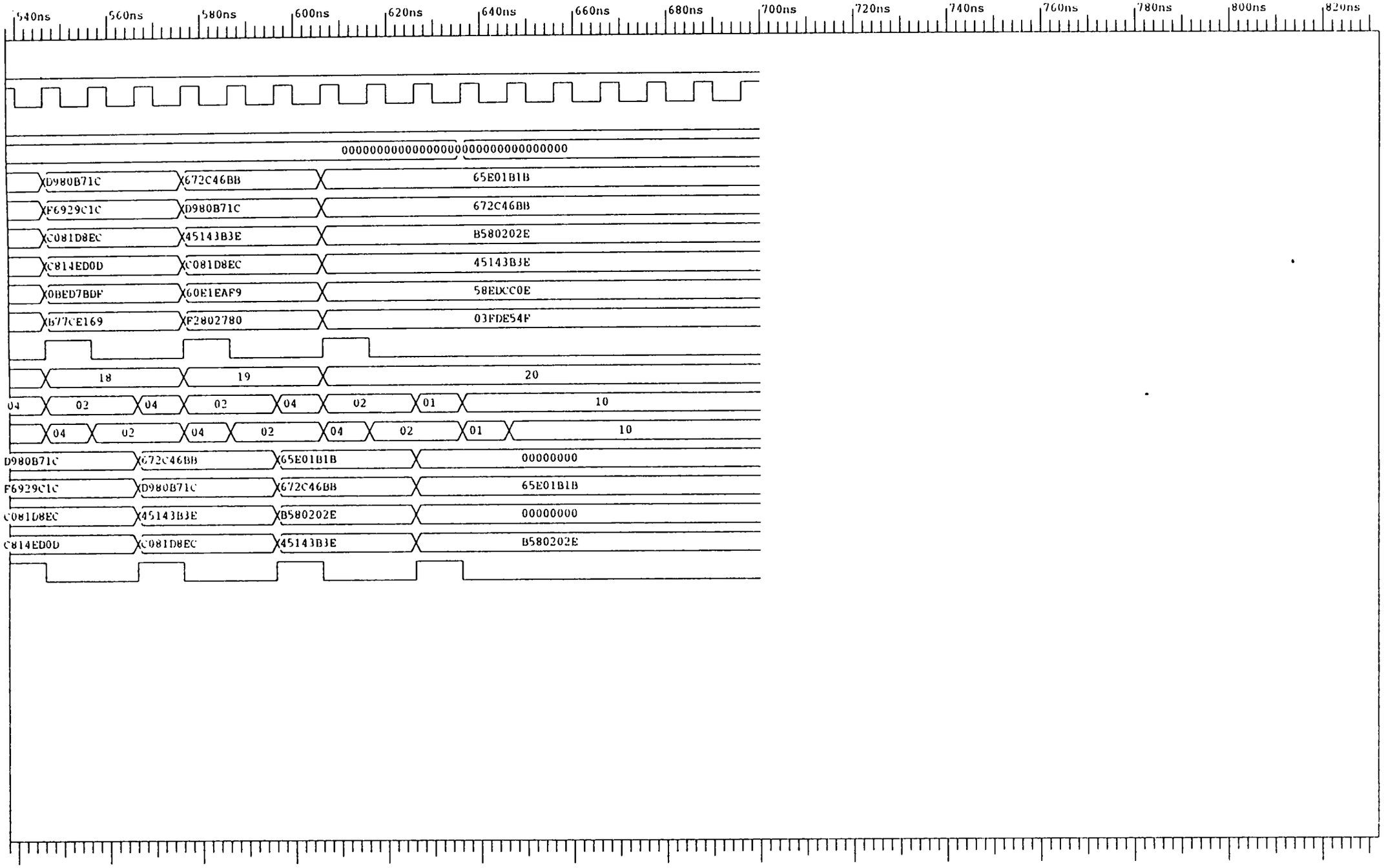
Anexa 10

Diagrama de timp pentru decriptare


0.0
20ns
40ns
60ns
80ns
100ns
120ns
140ns
160ns
180ns
200ns
220ns
240ns
i reset.....
i clk.....
i init.....
B text127.(hex)#1
B a_in131.(hex)#3
B b_in131.(hex)#3
B c_in131.(hex)#3
B d_in131.(hex)#3
B cheie131.(hex)#
B cheie231.(hex)#
l decrip.....
B count204.(dec)#
B next_s4.(hex)#5
B current_s4.(hex
B a_out131.(hex)#
B b_out131.(hex)#
B c_out131.(hex)#
B d_out131.(hex)#
l rezolvat.....
8F5FBD0510D15FA893FA3FDA6E857EC2
00000000 CA3313E8 54E1C28B 062DE7B1 1EAD6FA3 8A5504C1 5F7E1F38
00000000 A85FD110 CA3313E8 54E1C28B 062DE7B1 1EAD6FA3 8A5504C1
00000000 7CE05EBD 408412B4 783A7D12 3684545D5 73C687C5 A784AC14
00000000 C27E856E 7CE05EBD 408412B4 783A7D12 3684545D5 73C687C5
00000000 23526249 525AEF9D 82CF4F28 83DF6972 2B128625 AB027179
00000000 FE731BA8 847497AC 26CCCFFF 215799E4 8F007CAE A22E0011
00 01 02 03 04 05 06 07
10 04 02 04 02 04 02 04 02 04 02 04 02 04 02 04
10 08 04 02 04 02 04 02 04 02 04 02 04 02 04 02
00000000 54E1C28B 062DE7B1 1EAD6FA3 8A5504C1 5F7E1F38
00000000 CA3313E8 54E1C28B 062DE7B1 1EAD6FA3 8A5504C1
00000000 408412B4 783A7D12 3684545D5 73C687C5 A784AC14
00000000 7CE05EBD 408412B4 783A7D12 3684545D5 73C687C5

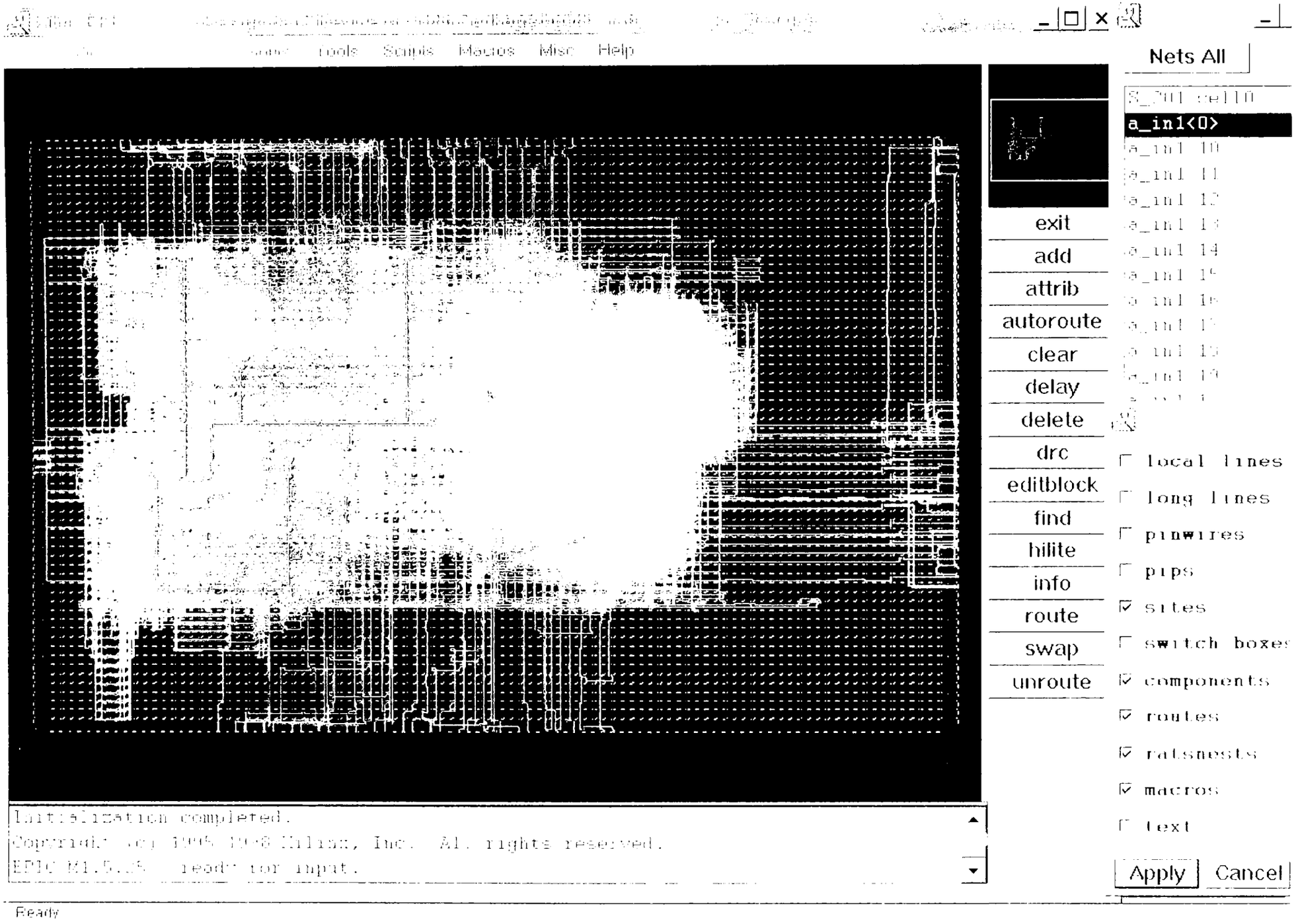
BUPT

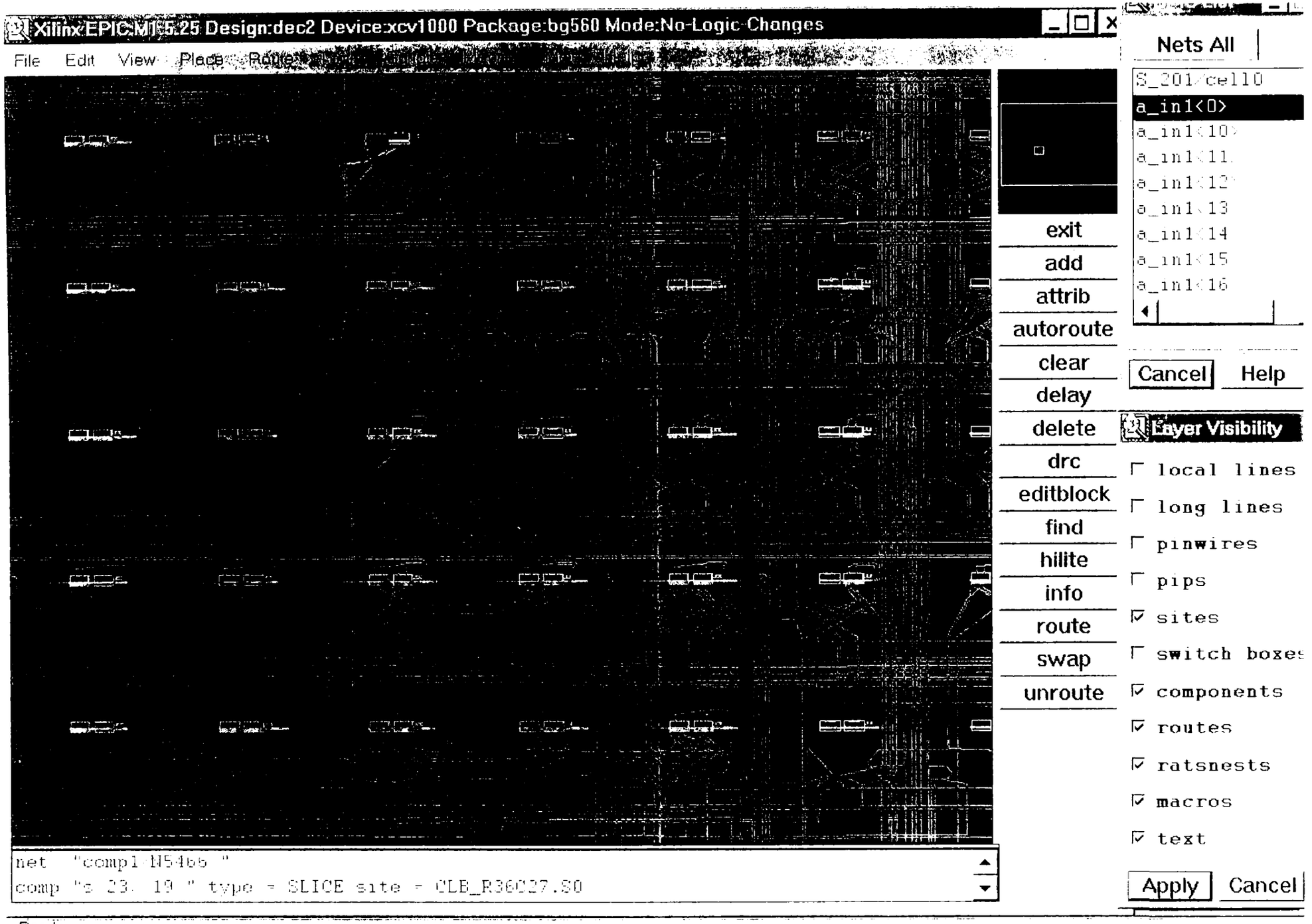

260ns
280ns
300ns
320ns
340ns
360ns
380ns
400ns
420ns
440ns
460ns
480ns
500ns
520ns
8F5FBD0510D15FA893FA3FDA6E857EC2
14047C8F
6E35DB8E
DF740024
E4C48DE6
18A912CC
1B075553
8B1FE432
3B584258
71B1618C
8BD89159
1B566ECA
31AB968D
2565EA8A
FD194DBB
E4D264E8
89F82499
5411F525
15558FCE
C27889F9
D303EFF4
8C494233
CC75E2D3
773273AB
230EF513
53686A6A
024584D8
47FFB9C5
A69CBFD8
EB0F30A8
26E8CC7A
42701AED
CE8457AC
911FEC26
D03FCDC9
CD995939
3912205A
BF0C6CFA
9D75B1AF
08
09
10
11
12
13
14
15
16
02
04
03

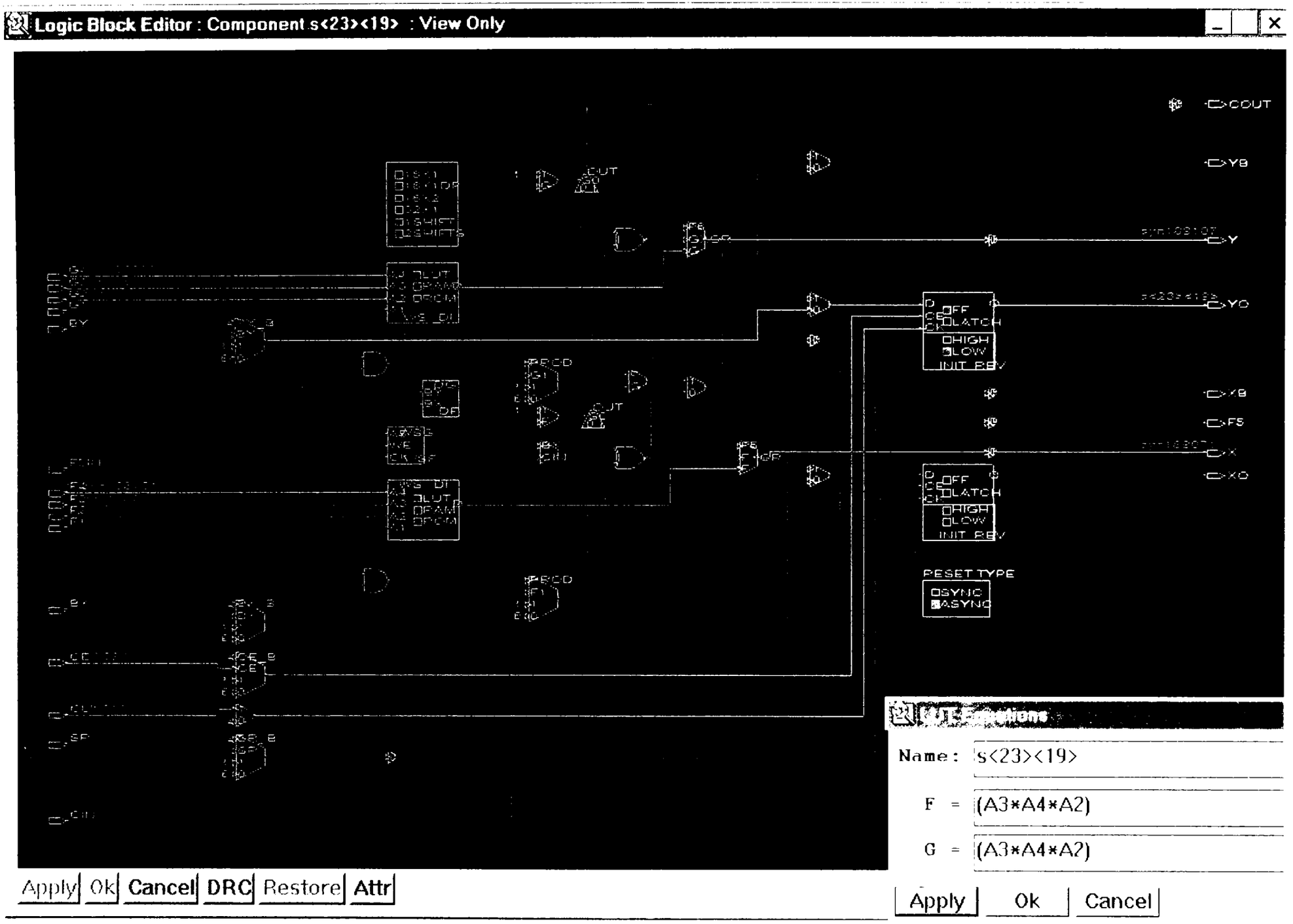

540ns
560ns
580ns
600ns
620ns
640ns
660ns
680ns
700ns
720ns
740ns
760ns
780ns
800ns
820ns
00000000000000000000000000000000
D980B71C
672C46BB
65E01B1B
F6929C1C
D980B71C
672C46BB
C081D8EC
45143B3E
B580202E
C814ED0D
C081D8EC
45143B3E
0BED7BDF
60E1EAF9
58EDCC0E
B77CE169
F2802780
03FDE54F
18
19
20
04
02
04
02
04
02
01
10
04
02
04
02
04
02
01
10
D980B71C
672C46BB
65E01B1B
00000000
F6929C1C
D980B71C
672C46BB
65E01B1B
C081D8EC
45143B3E
B580202E
00000000
C814ED0D
C081D8EC
45143B3E
B580202E

Anexa 11

Harta distribuției circuitelor din modulul de decriptare


Tools
Scripts
Macros
Misc
Help
Nets All
a_in1<0>
exit
add
attrib
autoroute
clear
delay
delete
drc
editblock
find
hilite
info
route
swap
unroute
local lines
long lines
pinwires
pips
sites
switch boxes
components
routes
ratsnests
macros
text
Apply
Cancel
Initialization completed.
Ready


Xilinx EPIC M1.5.25 Design:dec2 Device:xcv1000 Package:bg560 Mode:No-Logic-Changes
File
Edit
View
Place
Route
exit
add
attrib
autoroute
clear
delay
delete
drc
editblock
find
hilite
info
route
swap
unroute
Nets All
S_201/cell0
a_in1<0>
a_in1<10>
a_in1<11>
a_in1<12>
a_in1<13>
a_in1<14>
a_in1<15>
a_in1<16>
Cancel
Help
Layer Visibility
local lines
long lines
pinwires
pips
sites
switch boxes
components
routes
ratsnests
macros
text
Apply
Cancel
net "comp1_N54b5 "
comp "s_23__19_" type = SLICE site = CLB_R36C27.S0
Ready


Logic Block Editor : Component s<23><19> : View Only
COUT
YB
Y
YQ
XB
FS
X
XQ
RESET TYPE
SYNC
ASYNC
Name: s<23><19>
F = (A3*A4*A2)
G = (A3*A4*A2)
Apply
Ok
Cancel
DRC
Restore
Attr
Apply
Ok
Cancel

Anexa 12

Rapoarte

```
                    Xilinx Mapping Report File for Design "criptare"
                Copyright (c) 1995-1998 Xilinx, Inc.  All rights reserved.

Design Information
------------------
Command Line   : map -p xcv1000-4-bg560 -o map.ncd cript.ngd cript.pcf
Target Device  : xv1000
Target Package : bg560
Target Speed   : -4
Mapper Version : virtex -- M1.5.29i
Mapped Date    : Wed Feb 10 11:56:51 1999

Design Summary
--------------
   Number of errors: 0
   Number of warnings: 0
   Number of CLB slices: 3669
   Number of function generators: 6637
   Number of flip flops or latches: 1999
   Number of external Iobs: 130
   Number of Bufgs: 1
   Number of Bufgpads: 1
Total equivalent gate count for design: 68666

Table of Contents
-----------------
Section 1 - Errors
Section 2 - Warnings
Section 3 - Design Attributes
Section 4 - Removed Logic Summary
Section 5 - Removed Logic
Section 6 - Added Logic
Section 7 - Expanded Logic
Section 8 - Signal Cross-Reference
Section 9 - Symbol Cross-Reference
Section 10 - IOB Properties
Section 11 - RPMs
Section 12 - Guide Report

Section 1 - Errors
------------------
Section 2 - Warnings
--------------------
Section 3 - Design Attributes
-----------------------------
Section 4 - Removed Logic Summary
---------------------------------
   2 block(s) optimized away
Section 5 - Removed Logic
-------------------------
Optimized Block(s):
TYPE          BLOCK
VCC           C17525
GND           C17526

To enable printing of redundant blocks removed and signals merged, set the
environment variable MAP_REPORT_DETAIL to TRUE and rerun map.

Section 6 - Added Logic
-----------------------
Section 7 - Expanded Logic
--------------------------
To enable this section, set the environment variable MAP_REPORT_DETAIL to TRUE
and rerun MAP.
```

```
Section 8 - Signal Cross-Reference
----------------------------------
To enable this section, set the environment variable MAP_REPORT_DETAIL to TRUE
and rerun MAP.

Section 9 - Symbol Cross-Reference
----------------------------------
To enable this section, set the environment variable MAP_REPORT_DETAIL to TRUE
and rerun MAP.

Section 10 - IOB Properties
---------------------------
text<127> (IOB - LVTTL) : SLEW=FAST DRIVE=12
text<126> (IOB - LVTTL) : SLEW=FAST DRIVE=12
text<125> (IOB - LVTTL) : SLEW=FAST DRIVE=12
text<124> (IOB - LVTTL) : SLEW=FAST DRIVE=12
text<123> (IOB - LVTTL) : SLEW=FAST DRIVE=12
text<122> (IOB - LVTTL) : SLEW=FAST DRIVE=12
text<121> (IOB - LVTTL) : SLEW=FAST DRIVE=12
text<120> (IOB - LVTTL) : SLEW=FAST DRIVE=12
text<119> (IOB - LVTTL) : SLEW=FAST DRIVE=12
text<118> (IOB - LVTTL) : SLEW=FAST DRIVE=12
text<117> (IOB - LVTTL) : SLEW=FAST DRIVE=12
text<116> (IOB - LVTTL) : SLEW=FAST DRIVE=12
text<115> (IOB - LVTTL) : SLEW=FAST DRIVE=12
text<114> (IOB - LVTTL) : SLEW=FAST DRIVE=12
text<113> (IOB - LVTTL) : SLEW=FAST DRIVE=12
text<112> (IOB - LVTTL) : SLEW=FAST DRIVE=12
text<111> (IOB - LVTTL) : SLEW=FAST DRIVE=12
text<110> (IOB - LVTTL) : SLEW=FAST DRIVE=12
.
.
.
text<9> (IOB - LVTTL) : SLEW=FAST DRIVE=12
text<8> (IOB - LVTTL) : SLEW=FAST DRIVE=12
text<7> (IOB - LVTTL) : SLEW=FAST DRIVE=12
text<6> (IOB - LVTTL) : SLEW=FAST DRIVE=12
text<5> (IOB - LVTTL) : SLEW=FAST DRIVE=12
text<4> (IOB - LVTTL) : SLEW=FAST DRIVE=12
text<3> (IOB - LVTTL) : SLEW=FAST DRIVE=12
text<2> (IOB - LVTTL) : SLEW=FAST DRIVE=12
text<1> (IOB - LVTTL) : SLEW=FAST DRIVE=12
text<0> (IOB - LVTTL) : SLEW=FAST DRIVE=12
clk (GCLKIOB)

Section 11 - RPMs
-----------------
Section 12 - Guide Report
-------------------------
Guide not run on this design.


                    PAR: Xilinx Place And Route M1.5.25.
          Copyright (c) 1995-1998 Xilinx, Inc.  All rights reserved.

Wed Feb 10 11:59:58 1999
par -w -ol 2 -d 0 map.ncd cript.ncd cript.pcf
Constraints file: cript.pcf
Loading device database for application par from file "map.ncd".
   "criptare" is an NCD, version 2.27, device xcv1000, package bg560, speed -4
Loading device for application par from file 'v1000.nph' in environment
C:/fndtn.
Device speed data version:  x1_0.69 1.75 Advanced.
Device utilization summary:
```

```
   Number of External GCLKIOBs          1 out of 4      25%
   Number of External IOBs            130 out of 404    32%

   Number of SLICEs                  3669 out of 12288  29%

   Number of GCLKs                      1 out of 4      25%

Overall effort level (-ol):   2 (set by user)
Placer effort level (-pl):    2 (default)
Placer cost table entry (-t): 1
Router effort level (-rl):    2 (default)
Starting initial Placement phase.  REAL time: 1 mins 24 secs
Finished initial Placement phase.  REAL time: 1 mins 36 secs
Writing design to file "cript.ncd".
Starting the placer. REAL time: 1 mins 39 secs
Placer score = 5483055
Placer score = 5354070
Placer score = 4697110
Placer score = 4291775
Placer score = 4162315
Placer score = 3965670
.
.
.
Placer score = 1269570
Placer score = 1262615
Placer score = 1255980
Placer score = 1250535
Placer score = 1246365
Placer score = 1243610
Placer completed in real time: 5 mins 30 secs
Writing design to file "cript.ncd".
Starting Optimizing Placer.  REAL time: 5 mins 34 secs
Optimizing
Swapped 793 comps.
Xilinx Placer [1]   1221425   REAL time: 5 mins 44 secs
Finished Optimizing Placer.  REAL time: 5 mins 44 secs
Writing design to file "cript.ncd".
Starting IO Improvement.  REAL time: 5 mins 48 secs
Placer score = 1207235
Finished IO Improvement.  REAL time: 5 mins 48 secs
Total REAL time to Placer completion: 5 mins 58 secs
Total CPU time to Placer completion: 0 secs
0 connection(s) routed; 23996 unrouted.
Starting router resource preassignment
Completed router resource preassignment. REAL time: 9 mins 49 secs
Starting iterative routing.
Routing active signals.
.....................................End of iteration 1
23996 successful; 0 unrouted; (0) REAL time: 11 mins 57 secs
Constraints are met.
Routing PWR/GND nets.
Power and ground nets completely routed.
Total REAL time: 11 mins 57 secs
Total CPU  time: 0 secs
End of route.  23996 routed (100.00%); 0 unrouted.
No errors found.
Completely routed.

This design was run without timing constraints.  It is likely that much better
circuit performance can be obtained by trying either or both of the following:

  - Enabling the Delay Based Cleanup router pass, if not already enabled
```

```
  - Supplying timing constraints in the input design

Total REAL time to Router completion: 12 mins 10 secs
Total CPU time to Router completion: 0 secs

Generating PAR statistics.

   The Delay Summary Report
   The Score for this design is: 569
The Number of signals not completely routed for this design is: 0
   The Average Connection Delay for this design is:         3.344 ns
   The Average Connection Delay on critical nets is:        0.000 ns
   The Average Clock Skew for this design is:               0.632 ns
   The Maximum Pin Delay is:                               12.453 ns
   The Average Connection Delay on the 10 Worst Nets is:   11.733 ns

   Listing Pin Delays by value: (ns)

    d <= 10    < d <= 20   < d <= 30   < d <= 40   < d <= 50    d > 50
   ---------   ---------   ---------   ---------   ---------   ---------
       23865         131           0           0           0           0

Writing design to file "cript.ncd".

All signals are completely routed.

Total REAL time to PAR completion: 12 mins 41 secs
Total CPU time to PAR completion: 0 secs
PAR done.

                          Xilinx TRACE, Version M1.5.25
            Copyright (c) 1995-1998 Xilinx, Inc.  All rights reserved.

Design file:              cript.ncd
Physical constraint file: cript.pcf
Device,speed:             xcv1000,-4 (x1_0.69 1.75 Advanced)
Report level:             error report, limited to 3 items per constraint
--------------------------------------------------------------------------------
WARNING:bastw:170 - No timing constraints found, doing default enumeration.
================================================================================
Timing constraint: Default period analysis
 2021965521 items analyzed, 0 timing errors detected.
 Minimum period is 321.025ns.
--------------------------------------------------------------------------------
Timing constraint: Default net enumeration
 9333 items analyzed, 0 timing errors detected.
 Maximum net delay is  12.453ns.
--------------------------------------------------------------------------------
All constraints were met.
Timing summary:
---------------
Timing errors: 0  Score: 0

Constraints cover 2021965521 paths, 9333 nets, and 23996 connections (100.0%
coverage)

Design statistics:
   Minimum period: 321.025ns (Maximum frequency:   3.115MHz)
   Maximum net delay:  12.453ns

Analysis completed Wed Feb 10 12:18:43 1999
```

```
                    Xilinx Mapping Report File for Design "decript"
                Copyright (c) 1995-1998 Xilinx, Inc.  All rights reserved.

Design Information
------------------
Command Line   : map -p xcv1000-4-bg560 -o map.ncd dec2.ngd dec2.pcf
Target Device  : xv1000
Target Package : bg560
Target Speed   : -4
Mapper Version : virtex -- M1.5.29i
Mapped Date    : Wed Feb 10 10:54:23 1999

Design Summary
--------------
   Number of errors: 0
   Number of warnings: 0
   Number of CLB slices: 3369
   Number of function generators: 5982
   Number of flip flops or latches: 1999
   Number of external Iobs: 130
   Number of Bufgs: 1
   Number of Bufgpads: 1
Total equivalent gate count for design: 64736

Table of Contents
-----------------
Section 1 - Errors
Section 2 - Warnings
Section 3 - Design Attributes
Section 4 - Removed Logic Summary
Section 5 - Removed Logic
Section 6 - Added Logic
Section 7 - Expanded Logic
Section 8 - Signal Cross-Reference
Section 9 - Symbol Cross-Reference
Section 10 - IOB Properties
Section 11 - RPMs
Section 12 - Guide Report

Section 1 - Errors
------------------
Section 2 - Warnings
--------------------
Section 3 - Design Attributes
-----------------------------
Section 4 - Removed Logic Summary
---------------------------------
   2 block(s) optimized away
Section 5 - Removed Logic
-------------------------
Optimized Block(s):
TYPE         BLOCK
VCC          C17017
GND          C17018

To enable printing of redundant blocks removed and signals merged, set the
environment variable MAP_REPORT_DETAIL to TRUE and rerun map.
Section 6 - Added Logic
-----------------------
Section 7 - Expanded Logic
--------------------------
To enable this section, set the environment variable MAP_REPORT_DETAIL to TRUE
and rerun MAP.
Section 8 - Signal Cross-Reference
```

```
---------------------------------
To enable this section, set the environment variable MAP_REPORT_DETAIL to TRUE
and rerun MAP.
Section 9 - Symbol Cross-Reference
---------------------------------
To enable this section, set the environment variable MAP_REPORT_DETAIL to TRUE
and rerun MAP.
Section 10 - IOB Properties
---------------------------
text<127> (IOB - LVTTL) : SLEW=FAST DRIVE=12
text<126> (IOB - LVTTL) : SLEW=FAST DRIVE=12
text<125> (IOB - LVTTL) : SLEW=FAST DRIVE=12
text<124> (IOB - LVTTL) : SLEW=FAST DRIVE=12
text<123> (IOB - LVTTL) : SLEW=FAST DRIVE=12
text<122> (IOB - LVTTL) : SLEW=FAST DRIVE=12
text<121> (IOB - LVTTL) : SLEW=FAST DRIVE=12
text<120> (IOB - LVTTL) : SLEW=FAST DRIVE=12
.
.
.
text<7> (IOB - LVTTL) : SLEW=FAST DRIVE=12
text<6> (IOB - LVTTL) : SLEW=FAST DRIVE=12
text<5> (IOB - LVTTL) : SLEW=FAST DRIVE=12
text<4> (IOB - LVTTL) : SLEW=FAST DRIVE=12
text<3> (IOB - LVTTL) : SLEW=FAST DRIVE=12
text<2> (IOB - LVTTL) : SLEW=FAST DRIVE=12
text<1> (IOB - LVTTL) : SLEW=FAST DRIVE=12
text<0> (IOB - LVTTL) : SLEW=FAST DRIVE=12
clk (GCLKIOB)

Section 11 - RPMs
-----------------
Section 12 - Guide Report
-------------------------
Guide not run on this design.


                    PAR: Xilinx Place And Route M1.5.25.
          Copyright (c) 1995-1998 Xilinx, Inc.  All rights reserved.

Wed Feb 10 10:57:16 1999

par -w -ol 2 -d 0 map.ncd dec2.ncd dec2.pcf

Constraints file: dec2.pcf

Loading device database for application par from file "map.ncd".
   "decript" is an NCD, version 2.27, device xcv1000, package bg560, speed -4
Loading device for application par from file 'v1000.nph' in environment
C:/fndtn.
Device speed data version:  x1_0.69 1.75 Advanced.

Device utilization summary:

   Number of External GCLKIOBs          1 out of 4      25%
   Number of External IOBs            130 out of 404    32%
   Number of SLICEs                  3369 out of 12288  27%
   Number of GCLKs                      1 out of 4      25%

Overall effort level (-ol):   2 (set by user)
Placer effort level (-pl):    2 (default)
Placer cost table entry (-t): 1
Router effort level (-rl):    2 (default)
```

```
Starting initial Placement phase.  REAL time: 47 secs
Finished initial Placement phase.  REAL time: 58 secs
Writing design to file "dec2.ncd".

Starting the placer. REAL time: 1 mins 1 secs
Placer score = 5172170
Placer score = 5055815
Placer score = 4383970
Placer score = 4042265
Placer score = 3848575
.
.
.
Placer score = 1125985
Placer score = 1121640
Placer score = 1117550
Placer score = 1114420
Placer score = 1111915
Placer completed in real time: 4 mins 8 secs

Writing design to file "dec2.ncd".

Starting Optimizing Placer.  REAL time: 4 mins 11 secs
Optimizing
Swapped 670 comps.
Xilinx Placer [1]   1093510   REAL time: 4 mins 19 secs
Finished Optimizing Placer.  REAL time: 4 mins 19 secs

Writing design to file "dec2.ncd".

Starting IO Improvement.  REAL time: 4 mins 23 secs
Placer score = 1066540
Finished IO Improvement.  REAL time: 4 mins 23 secs

Total REAL time to Placer completion: 4 mins 33 secs
Total CPU time to Placer completion: 0 secs

0 connection(s) routed; 22143 unrouted.
Starting router resource preassignment
Completed router resource preassignment. REAL time: 6 mins 57 secs
Starting iterative routing.
Routing active signals.
.................................End of iteration 1
22143 successful; 0 unrouted; (0) REAL time: 8 mins 48 secs
Constraints are met.
Routing PWR/GND nets.
Power and ground nets completely routed.
Total REAL time: 8 mins 48 secs
Total CPU  time: 0 secs
End of route.  22143 routed (100.00%); 0 unrouted.
No errors found.
Completely routed.

This design was run without timing constraints.  It is likely that much better
circuit performance can be obtained by trying either or both of the following:

  - Enabling the Delay Based Cleanup router pass, if not already enabled
  - Supplying timing constraints in the input design

Total REAL time to Router completion: 9 mins
Total CPU time to Router completion: 0 secs

Generating PAR statistics.
```

```
   The Delay Summary Report

   The Score for this design is: 554

The Number of signals not completely routed for this design is: 0

   The Average Connection Delay for this design is:         3.208 ns
   The Average Connection Delay on critical nets is:        0.000 ns
   The Average Clock Skew for this design is:               0.601 ns
   The Maximum Pin Delay is:                               14.556 ns
   The Average Connection Delay on the 10 Worst Nets is:   11.663 ns

   Listing Pin Delays by value: (ns)

    d <= 10    < d <= 20   < d <= 30   < d <= 40   < d <= 50    d > 50
   ---------   ---------   ---------   ---------   ---------   ---------
       21975         168           0           0           0           0

Writing design to file "dec2.ncd".

All signals are completely routed.

Total REAL time to PAR completion: 9 mins 31 secs
Total CPU time to PAR completion: 0 secs
PAR done.


                        Xilinx TRACE, Version M1.5.25
            Copyright (c) 1995-1998 Xilinx, Inc.  All rights reserved.

Design file:              dec2.ncd
Physical constraint file: dec2.pcf
Device,speed:             xcv1000,-4 (x1_0.69 1.75 Advanced)
Report level:             error report, limited to 3 items per constraint
--------------------------------------------------------------------------------
WARNING:bastw:170 - No timing constraints found, doing default enumeration.
================================================================================
Timing constraint: Default period analysis
 956686493 items analyzed, 0 timing errors detected.
 Minimum period is 310.947ns.
--------------------------------------------------------------------------------
Timing constraint: Default net enumeration
 8614 items analyzed, 0 timing errors detected.
 Maximum net delay is  14.556ns.
--------------------------------------------------------------------------------
All constraints were met.
Timing summary:
---------------
Timing errors: 0  Score: 0
Constraints cover 956686493 paths, 8614 nets, and 22143 connections (100.0%
coverage)

Design statistics:
   Minimum period: 310.947ns (Maximum frequency:   3.216MHz)
   Maximum net delay:  14.556ns

Analysis completed Wed Feb 10 11:08:26 1999


     EDIF netlist 'c:\FNDTN\ACTIVE\PROJECTS\DEC2\DEC2.EDF' loading v.2.99.1.10
EDIF parser completed successfully
Loading - c:\fndtn\active\projects\dec2\dec2.edf
================================
```

```
EDIF Netlist Loading Statistics
===============================
Number of hierarchical blocks: 1
Total number of cell types: 15
Total number of cell instances: 10480
Total number of EDIF blocks: 0
Total number of models taken from libraries: 11
Total number of automatic/internal models: 3
Total number of missing models: 0
Total number of signals (block ports and labels): 10483
Total number of labels: 10352
Total number of nets: 10483
Total number of net elements: 51706
Total number of generic values: 0
Total number of port generic values: 0
Memory usage statistics:
    Total number of allocated memory blocks: 110769 (10484)
    Total number of memory reallocations: 3
    Total length of memory used to load the netlist: 9012 kB (225)
    Total length of memory used by EDIF parser: 4352 kB
    Number of system memory blocks (total/long): 20/3
Netlist loaded in: 20 seconds
Parsing time: 4 seconds
===============================
```

Xilinx Mapping Report File for Design "gen_chei"


```
Design Information
------------------
Command Line   : map -p xcv1000-4-bg560 -o map.ncd gen_chei.ngd gen_chei.pcf
Target Device  : xv1000
Target Package : bg560
Target Speed   : -4
Mapper Version : virtex -- M1.5.29i
Mapped Date    : Wed Feb 10 17:38:59 1999

Design Summary
--------------
   Number of errors: 0
   Number of warnings: 0
   Number of CLB slices: 2329
   Number of function generators: 4658
   Number of flip flops or latches: 1781
   Number of external Iobs: 34
   Number of Bufgs: 1
   Number of Bufgpads: 1
Total equivalent gate count for design: 42997
Table of Contents
-----------------
Section 1 - Errors
Section 2 - Warnings
Section 3 - Design Attributes
Section 4 - Removed Logic Summary
Section 5 - Removed Logic
Section 6 - Added Logic
Section 7 - Expanded Logic
Section 8 - Signal Cross-Reference
Section 9 - Symbol Cross-Reference
Section 10 - IOB Properties
Section 11 - RPMs
Section 12 - Guide Report
```

```
Section 1 - Errors
------------------
Section 2 - Warnings
--------------------
Section 3 - Design Attributes
-----------------------------
Section 4 - Removed Logic Summary
---------------------------------
   2 block(s) optimized away
Section 5 - Removed Logic
-------------------------
Optimized Block(s):
TYPE          BLOCK
VCC           C14115
GND           C14116

To enable printing of redundant blocks removed and signals merged, set the
environment variable MAP_REPORT_DETAIL to TRUE and rerun map.

Section 6 - Added Logic
-----------------------
Section 7 - Expanded Logic
--------------------------
To enable this section, set the environment variable MAP_REPORT_DETAIL to TRUE
and rerun MAP.

Section 8 - Signal Cross-Reference
----------------------------------
To enable this section, set the environment variable MAP_REPORT_DETAIL to TRUE
and rerun MAP.

Section 9 - Symbol Cross-Reference
----------------------------------
To enable this section, set the environment variable MAP_REPORT_DETAIL to TRUE
and rerun MAP.

Section 10 - IOB Properties
---------------------------
s0<31> (IOB - LVTTL) : SLEW=FAST DRIVE=12
s0<30> (IOB - LVTTL) : SLEW=FAST DRIVE=12
s0<29> (IOB - LVTTL) : SLEW=FAST DRIVE=12
s0<28> (IOB - LVTTL) : SLEW=FAST DRIVE=12
s0<27> (IOB - LVTTL) : SLEW=FAST DRIVE=12
s0<26> (IOB - LVTTL) : SLEW=FAST DRIVE=12
s0<25> (IOB - LVTTL) : SLEW=FAST DRIVE=12
s0<24> (IOB - LVTTL) : SLEW=FAST DRIVE=12
.
.
.
s0<5> (IOB - LVTTL) : SLEW=FAST DRIVE=12
s0<4> (IOB - LVTTL) : SLEW=FAST DRIVE=12
s0<3> (IOB - LVTTL) : SLEW=FAST DRIVE=12
s0<2> (IOB - LVTTL) : SLEW=FAST DRIVE=12
s0<1> (IOB - LVTTL) : SLEW=FAST DRIVE=12
s0<0> (IOB - LVTTL) : SLEW=FAST DRIVE=12
clk (GCLKIOB)

Section 11 - RPMs
-----------------
Section 12 - Guide Report
-------------------------
Guide not run on this design.
```

```
                    PAR: Xilinx Place And Route M1.5.25.
          Copyright (c) 1995-1998 Xilinx, Inc.  All rights reserved.

Wed Feb 10 17:41:23 1999

par -w -ol 2 -d 0 map.ncd gen_chei.ncd gen_chei.pcf


Constraints file: gen_chei.pcf

Loading device database for application par from file "map.ncd".
   "gen_chei" is an NCD, version 2.27, device xcv1000, package bg560, speed -4
Loading device for application par from file 'v1000.nph' in environment
C:/fndtn.
Device speed data version:  x1_0.69 1.75 Advanced.

Device utilization summary:

   Number of External GCLKIOBs          1 out of 4      25%
   Number of External IOBs             34 out of 404     8%
   Number of SLICEs                  2329 out of 12288  18%
   Number of GCLKs                      1 out of 4      25%

Overall effort level (-ol):   2 (set by user)
Placer effort level (-pl):    2 (default)
Placer cost table entry (-t): 1
Router effort level (-rl):    2 (default)

Starting initial Placement phase.  REAL time: 33 secs
Finished initial Placement phase.  REAL time: 41 secs

Writing design to file "gen_chei.ncd".

Starting the placer. REAL time: 43 secs
Placer score = 3008625
Placer score = 2962015
Placer score = 2543995
.
.
.
Placer score = 382940
Placer score = 379470
Placer score = 376815
Placer score = 374785
Placer score = 372625
Placer completed in real time: 4 mins 16 secs

Writing design to file "gen_chei.ncd".

Starting Optimizing Placer.  REAL time: 4 mins 18 secs
Optimizing
Swapped 507 comps.
Xilinx Placer [1]   359375   REAL time: 4 mins 29 secs
Finished Optimizing Placer.  REAL time: 4 mins 29 secs

Writing design to file "gen_chei.ncd".

Starting IO Improvement.  REAL time: 4 mins 31 secs
Placer score = 355085
Finished IO Improvement.  REAL time: 4 mins 32 secs

Total REAL time to Placer completion: 4 mins 38 secs
Total CPU time to Placer completion: 0 secs
```

```
0 connection(s) routed; 19404 unrouted active, 7 unrouted PWR/GND.
Starting router resource preassignment
Completed router resource preassignment. REAL time: 11 mins 23 secs
Starting iterative routing.
Routing active signals.
.........................End of iteration 1
19404 successful; 0 unrouted active,
   7 unrouted PWR/GND; (0) REAL time: 16 mins 58 secs
End of iteration 2
19404 successful; 0 unrouted active,
   7 unrouted PWR/GND; (0) REAL time: 17 mins 5 secs
Constraints are met.
Routing PWR/GND nets.
Power and ground nets completely routed.
Total REAL time: 19 mins 13 secs
Total CPU  time: 0 secs
End of route.  19411 routed (100.00%); 0 unrouted.
No errors found.
Completely routed.

This design was run without timing constraints.  It is likely that much better
circuit performance can be obtained by trying either or both of the following:

  - Enabling the Delay Based Cleanup router pass, if not already enabled
  - Supplying timing constraints in the input design

Total REAL time to Router completion: 19 mins 28 secs
Total CPU time to Router completion: 0 secs

Generating PAR statistics.
   The Delay Summary Report
   The Score for this design is: 774
The Number of signals not completely routed for this design is: 0
   The Average Connection Delay for this design is:        4.610 ns
   The Average Connection Delay on critical nets is:       0.000 ns
   The Average Clock Skew for this design is:              0.387 ns
   The Maximum Pin Delay is:                              22.184 ns
   The Average Connection Delay on the 10 Worst Nets is:  15.676 ns
   Listing Pin Delays by value: (ns)

    d <= 10    < d <= 20   < d <= 30   < d <= 40   < d <= 50    d > 50
   ---------   ---------   ---------   ---------   ---------   ---------
       17634        1659         118           0           0           0

Writing design to file "gen_chei.ncd".
All signals are completely routed.
Total REAL time to PAR completion: 19 mins 56 secs
Total CPU time to PAR completion: 0 secs
PAR done.


                        Xilinx TRACE, Version M1.5.25
            Copyright (c) 1995-1998 Xilinx, Inc.  All rights reserved.

Design file:              gen_chei.ncd
Physical constraint file: gen_chei.pcf
Device,speed:             xcv1000,-4 (x1_0.69 1.75 Advanced)
Report level:             error report, limited to 3 items per constraint
                  ------------------------------------------------------------
WARNING:bastw:170 - No timing constraints found, doing default enumeration.

Timing constraint: Default period analysis
 1101132069 items analyzed, 0 timing errors detected.
```

```
 Minimum period is  98.900ns.
--------------------------------------------------------------------------------
Timing constraint: Default net enumeration
 4764 items analyzed, 0 timing errors detected.
 Maximum net delay is  22.184ns.
--------------------------------------------------------------------------------
All constraints were met.

Timing summary:
---------------
Timing errors: 0  Score: 0

Constraints cover 1101132069 paths, 4764 nets, and 19404 connections (100.0%
coverage)

Design statistics:
   Minimum period:  98.900ns (Maximum frequency:  10.111MHz)
   Maximum net delay:  22.184ns

Analysis completed Wed Feb 10 18:28:43 1999
--------------------------------------------------------------------------------
```

Anexa 13

Implementarea hardware a algoritmului RC6 utilizând Verilog

```
module rc6(clock, reset, enable, encrypt, new_key, new_text, text_valid, key_valid,
key_type, key, text_in, text_out, ready);

input clock, reset, enable, encrypt, new_key, new_text, text_valid, key_valid;
input [1:0] key_type;                    // 00- 128bit; 01- 192bit; 10,11- 256bit
input [0:255] key;
input [0:127] text_in;

output [0:127] text_out;
output ready;

reg [31:0] L [0:7];                      // user supplied key
reg [31:0] S [0:43];                     // 32bit round keys
reg [31:0] A, B, C, D;                   // plain (cipher) text stored in 4x32bit
registers
reg [31:0] t, u;
reg [5:0] i;
reg [2:0] j,k;
reg [2:0] c;                             // <key type> (c+1)x32bit_words = key_length
reg [16:0] state, next_state;
reg [0:127] text_out;
reg ready, flag, next_flag;

parameter [4:0] START=0,

                SK0 = 1,
                SK1 = 2,
                SK2 = 3,
                SK3 = 4,
                SK4 = 5,

                SE0 = 6,
                SE1 = 7,
                SE2 = 8,
                SE3 = 9,
                SE4 = 10,

                SD0 = 11,
                SD1 = 12,
                SD2 = 13,
                SD3 = 14,
                SD4 = 15,

               FINISH = 16;

function [31:0] rotl3;                   // rotate to the left by 3
 input [31:0] rot;
 reg [2:0] z;
 begin
  z=rot[31:29];
  rot=rot<<3;
  rot[2:0]=z;
  rotl3=rot;
 end
endfunction

function [31:0] rotl5;                   // rotate to the left by 5
 input [31:0] rot;
 reg [4:0] z;
 begin
  z=rot[31:27];
  rot=rot<<5;
  rot[4:0]=z;
```

```
  rotl5=rot;
 end
endfunction

function [31:0] rotl;                    // rotate to the left by r
input [31:0] rot;
input [4:0] r;
reg z;
begin
 if(r<16) begin
           repeat(r)
            begin
             z=rot[31];
             rot=rot<<1;
             rot[0]=z;
            end
           rotl=rot;
          end
 else begin
           repeat(32-r)
            begin
             z=rot[0];
             rot=rot>>1;
             rot[31]=z;
            end
           rotl=rot;
          end
end
endfunction

always @(state or enable or text_valid or key_valid or new_text or flag) begin

 next_state=17'b0;                       // default value
 case(1'b1)
  state[START]:
   begin
    ready=1'b0;
//  text_out=0;
    if(enable && text_valid)
      begin
       A<={text_in[24:31],text_in[16:23],text_in[8:15],text_in[0:7]};
       B<={text_in[56:63],text_in[48:55],text_in[40:47],text_in[32:39]};
       C<={text_in[88:95],text_in[80:87],text_in[72:79],text_in[64:71]};
       D<={text_in[119:127],text_in[112:119],text_in[104:111],text_in[96:103]};
    if(new_key) begin
                  next_state[SK0]=1'b1;
                end
    else if(encrypt) begin
                       next_state[SE0]=1'b1;
                     end
         else begin
                next_state[SD0]=1'b1;
              end
     end
    else begin
           next_state[START]=1'b1;
         end

   end

  state[SK0]:
   begin
    if(key_valid) begin
                  if(key_type[1]==1'b1)
```

```
                    begin
                     j<=7;
                     c<=7;
                    L[0]<={key[24:31],key[16:23],key[8:15],key[0:7]};
                    L[1]<={key[56:63],key[48:55],key[40:47],key[32:39]};
                    L[2]<={key[88:95],key[80:87],key[72:79],key[64:71]};
                    L[3]<={key[119:127],key[112:119],key[104:111],key[ 96:103]};
                    L[4]<={key[152:159],key[144:151],key[136:143],key[128:135]};
                    L[5]<={key[184:191],key[176:183],key[168:175],key[160:167]};
                    L[6]<={key[216:223],key[208:215],key[200:207],key[192:199]};
                    L[7]<={key[248:255],key[240:247],key[232:239],key[224:231]};
                     end
                    else if(key_type[0]==1'b1)
                           begin
                            j<=5;
                            c<=5;
                    L[0]<={key[24:31],key[16:23],key[8:15],key[0:7]};
                    L[1]<={key[56:63],key[48:55],key[40:47],key[32:39]};
                    L[2]<={key[88:95],key[80:87],key[72:79],key[64:71]};
                    L[3]<={key[119:127],key[112:119],key[104:111],key[ 96:103]};
                   L[4]<={key[152:159],key[144:151],key[136:143],key[128:135]};
                   L[5]<={key[184:191],key[176:183],key[168:175],key[160:167]};
                           end
                         else begin
                                j<=3;
                                c<=3;
                    L[0]<={key[24:31],key[16:23],key[8:15],key[0:7]};
                    L[1]<={key[56:63],key[48:55],key[40:47],key[32:39]};
                    L[2]<={key[88:95],key[80:87],key[72:79],key[64:71]};
                    L[3]<={key[119:127],key[112:119],key[104:111],key[ 96:103]};
                                end
                  next_state[SK1]=1'b1;
                 end
    else begin
          next_state[SK0]=1'b1;
         end
 end

state[SK1]:
 begin
  S[0]=rotl3(32'hb7e15163);
  u=S[0];
  L[0]=rotl(L[0] + S[0], u[4:0]);
  i=0;
  k=1;
  next_state[SK2]=1'b1;
 end

state[SK2]:
 begin
  flag=1'b0;
  if(i<43) begin
            i=i+1;
            S[i] = rotl3(32'hb7e15163 + i*32'h9e3779b9 + S[i-1] + L[c-j]);
            u = S[i] + L[c-j];
            L[k] = rotl(L[k] + S[i] + L[c-j], u[4:0]);
            j=j-1;
            if(key_type==2'b00 && j==7) begin
                                        j=3;
                                       end
            if(key_type==2'b01 && j==7) begin
                                        j=5;
                                       end
            k=c-j+1;
```

```
            if(key_type[0]==1'b0 && j==0) begin
                                            k=0;
                                           end
            next_state[SK2]=1'b1;
            next_flag=~flag;
           end
   else begin
         S[0] = rotl3(S[0] + S[43] + L[c-j]);
         u = S[0] + L[c-j];
         L[k] = rotl(L[k] + S[0] + L[c-j], u[4:0]);
         i=0;
         next_state[SK3]=1'b1;
        end
 end

state[SK3]:
 begin
  if(i<43) begin
           i=i+1;
           j=j-1;
           if(key_type==2'b00 && j==7) begin
                                        j=3;
                                       end
           if(key_type==2'b01 && j==7) begin
                                        j=5;
                                       end
           k=c-j+1;
           if(key_type[0]==1'b0 && j==0) begin
                                           k=0;
                                          end
           S[i] = rotl3(S[i] + S[i-1] + L[c-j]);
           u = S[i] + L[c-j];
           L[k] = rotl(L[k] + S[i] + L[c-j], u[4:0]);
           next_state[SK3]=1'b1;
           next_flag=~flag;
          end
   else begin
         j=j-1;
         if(key_type==2'b00 && j==7) begin
                                      j=3;
                                     end
         if(key_type==2'b01 && j==7) begin
                                      j=5;
                                     end
         k=c-j+1;
         if(key_type[0]==1'b0 && j==0) begin
                                         k=0;
                                        end
         S[0] = rotl3(S[0] + S[43] + L[c-j]);
         u = S[0] + L[c-j];
         L[k] = rotl(L[k] + S[0] + L[c-j], u[4:0]);
         i=0;
         next_state[SK4]=1'b1;
        end
 end

state[SK4]:
 begin
  if(i<43) begin
           i=i+1;
           j=j-1;
           if(key_type==2'b00 && j==7) begin
                                        j=3;
                                       end
```

```
            if(key_type==2'b01 && j==7) begin
                                        j=5;
                                       end
            k=c-j+1;
            if(key_type[0]==1'b0 && j==0) begin
                                           k=0;
                                          end
            S[i] = rotl3(S[i] + S[i-1] + L[c-j]);
            u = S[i] + L[c-j];
            L[k] = rotl(L[k] + S[i] + L[c-j], u[4:0]);
            next_state[SK4]=1'b1;
            next_flag=~flag;
           end
   else if(encrypt) begin
                     next_state[SE0]=1'b1;
                    end
         else begin
               next_state[SD0]=1'b1;
              end
end

state[SE0]:                              // begin encryption
 begin
  B=B+S[0];
  D=D+S[1];
  i=1;
  next_state[SE1]=1'b1;
 end

state[SE1]:
 begin
  t=rotl5(B*(2*B+1));
  u=rotl5(D*(2*D+1));
  A=rotl(A^t, u[4:0]) + S[2*i];
  C=rotl(C^u, t[4:0]) + S[2*i+1];
  i=i+1;
  next_state[SE2]=1'b1;
 end

state[SE2]:
 begin
  t=rotl5(C*(2*C+1));
  u=rotl5(A*(2*A+1));
  B=rotl(B^t, u[4:0]) + S[2*i];
  D=rotl(D^u, t[4:0]) + S[2*i+1];
  i=i+1;
  next_state[SE3]=1'b1;
 end

state[SE3]:
 begin
  t=rotl5(D*(2*D+1));
  u=rotl5(B*(2*B+1));
  C=rotl(C^t, u[4:0]) + S[2*i];
  A=rotl(A^u, t[4:0]) + S[2*i+1];
  i=i+1;
  next_state[SE4]=1'b1;
 end

state[SE4]:
 begin
  t=rotl5(A*(2*A+1));
  u=rotl5(C*(2*C+1));
  D=rotl(D^t, u[4:0]) + S[2*i];
```

```
   B=rotl(B^u, t[4:0]) + S[2*i+1];
   i=i+1;
   if(i<18) begin
             next_state[SE1]=1'b1;
            end
    else begin
          A=A+S[42];
          C=C+S[43];
          next_state[FINISH]=1'b1;
         end
 end

state[SD0]:                                  // begin decryption
 begin
  C=C-S[43];
  A=A-S[42];
  i=20;
  next_state[SD1]=1'b1;
 end

state[SD1]:
 begin
  u=rotl5(C*(2*C+1));
  t=rotl5(A*(2*A+1));
  B=rotl(B-S[2*i+1], 32-t[4:0])^u;
  D=rotl(D-S[2*i], 32-u[4:0])^t;
  i=i-1;
  next_state[SD2]=1'b1;
 end

state[SD2]:
 begin
  u=rotl5(B*(2*B+1));
  t=rotl5(D*(2*D+1));
  A=rotl(A-S[2*i+1], 32-t[4:0])^u;
  C=rotl(C-S[2*i], 32-u[4:0])^t;
  i=i-1;
  next_state[SD3]=1'b1;
 end

state[SD3]:
 begin
  u=rotl5(A*(2*A+1));
  t=rotl5(C*(2*C+1));
  D=rotl(D-S[2*i+1], 32-t[4:0])^u;
  B=rotl(B-S[2*i], 32-u[4:0])^t;
  i=i-1;
  next_state[SD4]=1'b1;
 end

state[SD4]:
 begin
  u=rotl5(D*(2*D+1));
  t=rotl5(B*(2*B+1));
  C=rotl(C-S[2*i+1], 32-t[4:0])^u;
  A=rotl(A-S[2*i], 32-u[4:0])^t;
  i=i-1;
  if(i>0) begin
           next_state[SD1]=1'b1;
          end
   else begin
         D=D-S[1];
         B=B-S[0];
         next_state[FINISH]=1'b1;
```

```
          end
   end

  state[FINISH]:
   begin
    text_out={A[7:0],A[15:8],A[23:16],A[31:24],B[7:0],B[15:8],B[23:16],B[31:24],
              C[7:0],C[15:8],C[23:16],C[31:24],D[7:0],D[15:8],D[23:16],D[31:24]};
    ready=1;
    if(new_text) begin
                  next_state[START]=1'b1;
                 end
     else begin
           next_state[FINISH]=1'b1;
          end
   end
endcase
end
always @(posedge clock or negedge reset) begin
  if(reset==0) begin
                state <= #1 17'b0;
                state[START] <= #2 1'b1;
               end
  else begin
        #1 state <= next_state;
        flag <= next_flag;
        $display("state=%b  i=%d", state, i);
       end
end
endmodule
```

Anexa 14

Implementarea software a algoritmului RC6 utilizând limbajul C

```
// RC6 - AES candidate
#include <stdio.h>
#include <stdlib.h>
typedef unsigned char   u1byte; /* an 8 bit unsigned character type */
typedef unsigned short  u2byte; /* a 16 bit unsigned integer type   */
typedef unsigned long   u4byte; /* a 32 bit unsigned integer type   */
FILE *f1,*f2,*f3; /* f1 - Plain_Text, f2 - User_Key, f3 - Cipher_Text */
/* Circular rotate of 32 bit values                                  */
#define rotr(x,n)   (((x) >> ((n)&31)) | ((x) << (32 - ((n)&31))))
#define rotl(x,n)   (((x) << ((n)&31)) | ((x) >> (32 - ((n)&31))))

#define f_rnd(i,a,b,c,d)     u = rotl(d*(d+d+1),5);      \
                             t = rotl(b*(b+b+1),5);      \
                             a = rotl(a^t,u)+l_key[i];   \
                             c = rotl(c^u,t)+l_key[i+1]
#define i_rnd(i,a,b,c,d)     u = rotl(d*(d+d+1),5);      \
                             t = rotl(b*(b+b+1),5);      \
                             c = rotr(c-l_key[i+1],t)^u; \
                             a = rotr(a-l_key[i],u)^t
#define bswap(x)    (rotl(x,8) & 0x00ff00ff | rotr(x,8) & 0xff00ff00)
u4byte  l_key[44];  /* storage for the key schedule (20 rounds) */
/* initialise the key schedule from the user supplied key   */
u4byte *set_key(u4byte in_key[8], u4byte key_len)
{   u4byte  i, j, k, a, b, l8, t, l[8];
    l_key[0] = 0xb7e15163;    /* P_32 */
    for(k = 1; k < 44; k++)   /* S[0..2r+3] vector */
        l_key[k] = l_key[k-1] + 0x9e3779b9; /* Q_32 */
    for(k = 0; k < key_len / 32; ++k)
{
      l[k] = bswap(in_key[k]);       /* L[0..c-1] vector */
printf("L[%i] = %08x\n",k,l[k]);}printf("\n");
    t = (key_len / 32) - 1;   /* c-1 */
    a = b = i = j = 0;
    for(k = 0; k < 132; k++)  /* 3*max{c,2r+4} = 3*44 = 132 */
    {
        a = rotl(l_key[i]+a+b,3);
        l_key[i] = a;
        b += a;
        b = rotl(l[j]+b,b);
        l[j] = b;
        i = (i == 43 ? 0 : i + 1);
        j = (j == t ? 0 : j + 1);
    }
printf("subkeys: \n");
 for (i = 0; i < 44; i ++) {
  printf("%08x ", l_key[i]);
  if (i % 4 == 3) printf("\n");
 }
 printf("\n");
    return l_key;
}; /* set_key */
/* encrypt a block of text  */
void encrypt(u4byte in_blk[4], u4byte out_blk[4])
{   u4byte  a, b, c, d, t, u;
    a = bswap(in_blk[0]);
    b = bswap(in_blk[1]) + l_key[0];
    c = bswap(in_blk[2]);
    d = bswap(in_blk[3]) + l_key[1];
printf("\n Round[0] : %08x %08x %08x %08x\n",a,b,c,d);
    /* 20 rounds */
    f_rnd(2,a,b,c,d);
printf(" Round[1] : %08x %08x %08x %08x\n",b,c,d,a);
    f_rnd(4,b,c,d,a);
printf(" Round[2] : %08x %08x %08x %08x\n",c,d,a,b);
```

```
    f_rnd(6,c,d,a,b);
printf(" Round[3] : %08x %08x %08x %08x\n",d,a,b,c);
    f_rnd(8,d,a,b,c);
printf(" Round[4] : %08x %08x %08x %08x\n",a,b,c,d);
    f_rnd(10,a,b,c,d);
printf(" Round[5] : %08x %08x %08x %08x\n",b,c,d,a);
    f_rnd(12,b,c,d,a);
printf(" Round[6] : %08x %08x %08x %08x\n",c,d,a,b);
    f_rnd(14,c,d,a,b);
printf(" Round[7] : %08x %08x %08x %08x\n",d,a,b,c);
    f_rnd(16,d,a,b,c);
printf(" Round[8] : %08x %08x %08x %08x\n",a,b,c,d);
    f_rnd(18,a,b,c,d);
printf(" Round[9] : %08x %08x %08x %08x\n",b,c,d,a);
    f_rnd(20,b,c,d,a);
printf("Round[10] : %08x %08x %08x %08x\n",c,d,a,b);
    f_rnd(22,c,d,a,b);
printf("Round[11] : %08x %08x %08x %08x\n",d,a,b,c);
    f_rnd(24,d,a,b,c);
printf("Round[12] : %08x %08x %08x %08x\n",a,b,c,d);
    f_rnd(26,a,b,c,d);
printf("Round[13] : %08x %08x %08x %08x\n",b,c,d,a);
    f_rnd(28,b,c,d,a);
printf("Round[14] : %08x %08x %08x %08x\n",c,d,a,b);
    f_rnd(30,c,d,a,b);
printf("Round[15] : %08x %08x %08x %08x\n",d,a,b,c);
    f_rnd(32,d,a,b,c);
printf("Round[16] : %08x %08x %08x %08x\n",a,b,c,d);
    f_rnd(34,a,b,c,d);
printf("Round[17] : %08x %08x %08x %08x\n",b,c,d,a);
    f_rnd(36,b,c,d,a);
printf("Round[18] : %08x %08x %08x %08x\n",c,d,a,b);
    f_rnd(38,c,d,a,b);
printf("Round[19] : %08x %08x %08x %08x\n",d,a,b,c);
    f_rnd(40,d,a,b,c);
printf("Round[20] : %08x %08x %08x %08x\n",a,b,c,d);
    out_blk[0] = bswap(a + l_key[42]);
    out_blk[1] = bswap(b);
    out_blk[2] = bswap(c + l_key[43]);
    out_blk[3] = bswap(d);
printf("    Final : %08x %08x %08x
%08x\n",out_blk[0],out_blk[1],out_blk[2],out_blk[3]);
}; /* encrypt */
/* decrypt a block of text  */
void decrypt(u4byte in_blk[4], u4byte out_blk[4])
{   u4byte  a,b,c,d,t,u;
    d = bswap(in_blk[3]);
    c = bswap(in_blk[2]) - l_key[43];
    b = bswap(in_blk[1]);
    a = bswap(in_blk[0]) - l_key[42];
    i_rnd(40,d,a,b,c); i_rnd(38,c,d,a,b);
    i_rnd(36,b,c,d,a); i_rnd(34,a,b,c,d);
    i_rnd(32,d,a,b,c); i_rnd(30,c,d,a,b);
    i_rnd(28,b,c,d,a); i_rnd(26,a,b,c,d);
    i_rnd(24,d,a,b,c); i_rnd(22,c,d,a,b);
    i_rnd(20,b,c,d,a); i_rnd(18,a,b,c,d);
    i_rnd(16,d,a,b,c); i_rnd(14,c,d,a,b);
    i_rnd(12,b,c,d,a); i_rnd(10,a,b,c,d);
    i_rnd(8,d,a,b,c);  i_rnd(6,c,d,a,b);
    i_rnd(4,b,c,d,a);  i_rnd(2,a,b,c,d);
    out_blk[3] = bswap(d - l_key[1]);
    out_blk[2] = bswap(c);
    out_blk[1] = bswap(b - l_key[0]);
    out_blk[0] = bswap(a);
```

```
}; /* decrypt */
main(int argc, char *argv[])
{
 u4byte  klength,key[8],in[4],out[4], blk[4],j,i,l,m,k, k_bytes;
 if(argc!=4) { printf("the name of a file is missing\n");
            exit(0);
           }
if((f1=fopen(argv[1],"r"))==NULL) { printf("cannot open file f1 (Plain_Text file)");
                                    exit(0);
                                    }
if((f2=fopen(argv[2],"r"))==NULL) { printf("cannot open file f2 (User_Key file)");
                                    exit(0);
                                   }
f3=fopen(argv[3],"w");
klength = 256;
       /*Introducerea datelor in fisiare (exemplu):\n");
       0x00000000, 0x00000000, 0x00000000,0x00000000 - Plain_Text in f1
       0x00000000, 0x00000000, 0x00000000,0x00000000 - User_Key in f2 */
       for (i= 0; i < 4; i++)
              fscanf(f1,"%x",&in[i]);
       for (i= 0; i < klength/32; i++)
              fscanf(f2,"%x",& key[i]);
       printf("Plain_Text : ");
              for (i= 0; i < 4; i++)
                     printf("%08x ",in[i]);
       printf("User_Key : ");
              for (i = 0; i< klength/32; i ++)
                     printf("%08x ",key[i]);
              printf("\n\n");
       set_key(key, klength);
encrypt(in,out);
       printf("\nciphertext: ");
       for (i= 0; i < 4; i++)
              printf("%08x ",out[i]);
       for (i= 0; i < 4; i++)
              fprintf(f3,"%08x ",out[i]);
decrypt(out,in);
       printf("\nplaintext : ");
       for (i= 0; i < 4; i++)
              printf("%08x ",in[i]);
printf("\n");
fclose(f1);
fclose(f2);
fclose(f3);
exit(0);
}
```

BIBLIOGRAFIE

1. [Acte-92] Actel Coorporation. „ACT Family Field Programmable Gate Array Data Book", 1992.
2. [AbBF-90] M. Abramovici, M. Breuer, A. Friedman: "Digital Systems Testing and Testable Design", Computer Science Press, 1990.
3. [Adam-98] C. Adams, „The CAST-256 Encription Algorithm", NIST AES Proposal, 1998.
4. [AFSPL-99] Akyman Financial Services Pty Ltd., letter dated February 19,1999.
5. [Alte-95] Altera Coorporation. „Altera Data Book", 1995.
6. [AnBK-98a] R. Anderson, E. Biham, L. Knudsen. „Serpent: A Proposal for the Advanced Encription Standard", NIST AES Proposal, 1998.
7. [AnBK-98b] R. Anderson, E. Biham, L. Knudsen. „Serpent and Smart Cards", Third Smart Card Research and Advanced Applications Conference Proceedings, 1998.
8. [ANSI-85] ANSI X9.17. „Financial institution key management", Technical report American Bankers Association, April 1985.
9. [Aviz-82] A. Avizienis. „Design diversity – the Challenge of the eighties", Proceedings of the 12th Annual International Symposium on Fault Tolerant Computing, Santa Monica, California, pag. 44-45, June 1982.
10. [Baue-97] F.L. Bauer. „Decrypted Secrets – Methods and Maxims of Cryptology", Springer Verlag, 1997.
11. [BCDG-98] C. Burwick, D. Coppersmith, E. D'Avignon, R. Gennaro, S. Halevi, C. Jutla, S.M. Matyas, L. O'Connor, M. Peyravian, D. Safford, N. Zunic, „MARS – A Candidate Cipher for AES", NIST Proposal, 1998.
12. [BCKF-91] H. Bonnenberg, A. Curiger, H. Kaeslin, N. Felber, X. Lai. „VLSI implementation of a New Block Cipher", In Proc. of the International Conference on Computer Design ICCD 91, Cambridge, 1991.

13. [BCFK-93] H. Bonnenberg, A. Curiger, N. Felber, H. Kaeslin. „VINCI: Secure Test of a High-speed Encryption System", In Proc. of the International Test Conference (ITC 93), Baltimore, MD, USA, October 1993.

14. [BCZF-93] H. Bonnenberg , A. Curiger, R. Zimmermann, N. Felber: „A New and Secure Selftest Scheme for Block Cipher Implementations". In Proc. of the 3rd European Test Conference (ETC) 1993.

15. [Beck-88] B. Becker. „Efficient Testing of Optimal Time Addres", IEEE Trans. On Computers, vol 37, nr. 9, pag. 1113-1121, 1988.

16. [Beck-97] B. Beckett. „Introduction to Cryptology and PC Security", McGraw-Hill Publishing Company, 1997.

17. [BePi-82] H. Becker, F. Piper. „Cipher Systems – The Protection of Communications". Northwood Books London, 1982.

18. [Biha-98] E. Biham. „A Note Comparing the AES Candidates", Israel Institute of Technology, 1998.

19. [Biha-99] E. Biham. „A note on comparing the AES candidates", The Second AES Conference, March 22-23, pag 85-92; 1999.

20. [BiKu-98] A. Biryukov, E. Kushilevitz. „Improved Cryptanalysis of RC5". Proceedings of Advances in Cryptology – Eurocrypt'98, Lecture Notes in Computer Science, Springer Verlag, 1998.

21. [BiSh-90] E. Biham, A. Shamir. „Differential Cryptanalysis of DES-like Crypto-systems", Advances in Cryptology – CRYPTO'90 Proceedings, 1990.

22. [BiSh-93] E. Biham, A. Shamir. „Differential Cryptanalysis of Data Encription, Standard", Sringer-Verlag, New York, 1993.

23. [BoCK-90] H. Bonnenberg, A. Curiger, H. Kaeslim. LEONARDO. „Design aspects of the VLSI implementation of a new secret key block cipher". Tehnical Report 05/90, Integrated Systems Laboratory , ETH Zurich , April 1990.

24. [Bout-94] Dave Van den Bout, "The Practical Xilinx Designer Lab Book", Prentice Hall, 1994.

25. [Brya-91] R. E. Bryant. "On the Complexity of VLSI Implementations and Graph Representation of Boolean Functions with Applications to Integer Multiplication", IEEE Trans. On Computers, vol. 40, nr. 2, pag. 205-213, 1991.

26. [BrPi-98] L. Brown, J. Pieprzyk. „Introducing the New LOKI97 Block Cipher", NIST Proposal,1998.

27. [BrSt-88] T. J. Brossman, N. R. Strader II. „Modular Error Detection for Bit-Serial Multiplication", IEEE Trans. On Computers, vol. 37, nr. 9, pag. 1043-1052, 1988.

28. [CaDN-99] G. Carter, E. Dawson şi L. Nielsen. „Key Schedule Clasification of the AES Candidates", Second Advanced Encryption Standard Candidate Conference, National Institute of Standard and Technology (NIST), March 1999.

29. [CăBa-85] V. Cătuneanu, I. Bacivarof, "Fiabilitatea sistemelor de telecomunicaţii", Ed. Militară, 1985.

30. [CăBa-89] V.M. Cătuneanu, A. Bacivarof, "Structuri electronice de înaltă fiabilitate.Toleranțe la defectări", Ed. Militară, Bucureşti, 1989.

31. [CăMi-83] V. Cătuneanu, A. Mihalache, "Bazele teoretice ale fiabilității", Ed. Academiei, 1983.

32. [ChGu-96] C.A. Chen, S.K. Gupta. „BIST Test Pattern Generators for Two-Pattern Testing – Theory and Design Algorithms", IEEE Transactions on Computers, vol. 45. no. 3, pag. 257-268, 1996.

33. [CMKK-98] L. Chen, J.L. Massey, G.H. Khachatrian, M.K. Kuregian. „Nomination of SAFER+ as Candidate Algorithm for the Advanced Encryption Standard (AES)", NIST AES Proposal, 1998.

34. [CoYi-98] S. Contini, Yiqun Lisa Yin. „On Differential Properties of Data-Dependent Rotations and their Use in MARS and RC6", RSA Laboratories, 1998.

35. [CRRY-98] S. Contini, R.L. Rivest, M.J.B. Robshaw şi Y.L. Yin. „The Security of the RC6 Block Cipher – version 1.0", M.I.T. Laboratory for Computer Science, Cambridge şi RSA Laboratories, San Mateo. August 20,1998.

36. [CRRY-99] S. Contini, R.L. Rivest, M.J.B. Robshaw şi Y.L.Yin. „Some Comments on the First Round AES Evaluation of RC6", AES-Submmision, 1999.

37. [CoQu-83] C. Couvreur, J.J. Quisquarter. „An introduction to fast generation of large prime numbers", Philips Journal of Research, Vol. 37, pag.231-264, 1982, (errata: id, Vol.38, pag. 77, 1983).

38. [CuBK-90] A. Curiger, H. Bonnenberg, H. Kaeslin. „VLSI-tailored architectures for multiplication modulo (2^n+1)". In Sixteenth European Solid-State Circuites Conference, pag. 249-252, Grenoble, France, 1990.

39. [DaRi-98a] J. Daemen, V. Rijmen. „AES Proposal: Rijndael", AES Proposal, 1998.

40. [DaRi-98b] J. Daemen, V. Rijmen. „Third Smart Card Research and Advanced Applications", Conference Proceedings, 1998.

41. [DaRi-99] J. Daemen şi V. Rijmen. „Resistance Implementation Attacks. A comparative Study of the AES Proposals". Second Advanced Encryption Standard Candidate Conference, National Institute of Standard and Technology (NIST), March 1999.

42. [Fede-82] Federal Information Processing Standards Publication (FIPS PUB) 140. „Telecomunications: General Security Requirements for Equipment using the Data Encryption Standard", 1982.

43. [Fede-77] Federal Information Processing Standards Publication (FIPS PUB) 46. "The Data Encryption Standard (DES)", 1977.

44. [Fuji-85] H. Fujiwara. „Logic Design and Testing for Testability", MIT Press Series in Computer Systems. The MIT Press, Cambridge/MA, London/UK, 1985.

45. [FuRi-88] W. Fumy, H. P. Rieß: „Kryptographie: Entwurf und Analyse symmetrischer Kryptosysteme". R. Oldenbourg Verlag, München, Wien, 1988.

46. [Glad-99] B. Gladman. „Implementation experience with AES Candidate Algoritms". Second Advanced Encryption Standard Candidate Conference, National Institute of Standard and Technology (NIST), March 1999.

47. [GeLC-98] D. Georgoudis, D. Lerous, B.S. Chaves. „The "Frog" Encription Algorithm", NIST Proposal, 1998.

48. [Goor-91] A.J. van de Goor. „Testing Semiconductor Memeories – Theory and Practice", John Wiley & Sons, 1991.

49. [Goli-96] J.D. Golic. „Linear Models for Keystream Generators", IEEE Transactions on Computers, vol. 45. no. 1, pag. 41-48, 1996.

50. [GoPe-96] J.D. Golic, S. Petrovic. „Correlation Attacks on Clock-Controlled Shift Registers in Keystream Generators", IEEE Transactions on Computers, vol. 45. no. 4, pag. 482-486, 1996.

51. [HaDu-98] F. Hanchek, S. Dutt. „Methodologies for Tolerating Cell and Interconnect Faults in FPGAs". IEEE Transactions on Computers, vol. 47, No. 1, Jan.1998.

52. [HRSW-91] K. Hafner, H. C. Ritter, T. M. Schwair, S. Wallstab, M. Deppermann, J. Gessner, S. Koesters, W.-D. Moeller, G. Sandweg: „Design and test of an integrated cryptochip". IEEE Design and Test , pag 6-17 , 1991.

53. [Intel-94] Intel Corporation. „MCS 51 Microcontroller Family User's Manual". Feb. 1994.

54. [ITSE-91] Information Technology Security Evaluation Criteria, June 1991.

55. [ITSE-93] Information Technology Security Evaluation Criteria, September 1993.

56. [JaHu-98] M.J. Jacobson, K. Huber. „The MAGENTA Block Cipher Algorithm", NIST AES Proposal, 1998.

57. [JaKn-97] T. Jakobsen şi L.R. Knudsen. „The interpolation attacks on block ciphers". In E.Biham, editor, Fast Software Encryption, volume 1267 of Lecture Notes in Computer Science, pag. 28-40, Springer-Verlag, 1997.

58. [John-84] B.W. Johnson. „Fault Tolerant microprocessor-based systems", IEEE Micro, vol. 4, nr. 6, pag. 6-21, 1984.

59. [John-89] B.W. Johnson. „Design and Analisys of Fault Tolerant Digital Systems", Edison-Wesley Publishing Company, 1989.

60. [Kaes-90] H. Kaeslin. „Sicherheitsprobleme Mikroelektronischer Schaltungen", Internal Report ETH Zürich, Designezentrum für Mikroelektronik, 1990.

61. [Kant-93] V. Kantabura. „A Recursive Carry-Lookahead/Carry-Select Hybrid Adder", IEEE Trans. on Computers, vol. 42, nr. 12, pag. 1496-1499, 1993.

62. [KaRo-94] B.S. Kaliski, M.J.B. Robshaw. „Linear Cryptanalysis Using Multiple Approximations". In Y.G.Desmedt, editor, Advances in Cryptology/Crypto'94,

volume 839 of Lecture Notes in Computer Science, pag. 26-39, New York, Springer-Verlag, 1994.

63. [KaRo-95] B.S. Kaliski, M.J.B. Robshaw. „Linear Cryptanalysis Using Multiple Approximations and FEAL". In B.Preneel, editor, Fast Software Encryption, volume 1008 of Lecture Notes in Computer Science, pag. 249-264, Springer-Verlag, 1995.

64. [Katt-96] R.S. Katti. „A new Residue Arithmetic Error Correction Scheme", IEEE Transactions on Computers, vol. 45, no. 1, pag. 13-19, 1996.

65. [KaYi-95a] B.S. Kaliski, Y.I. Yin. „On Differential and Liniar Cryptanalysis of the RC5 Encryption Algorithm". In D. Coppersmith, editor, Advances in Cryptology – Crypto '95, volume 963 of Lecture Notes in Computer Science, pagers 171-184, Springer Verlag, 1995.

66. [KaYi-95b] B.S. Kaliski, Y.I. Yin. „On the Security of the RC5 Encription Algorithm". RSA Laboratories Technical Report TR-602, 1995.

67. [KeSW- 96] J. Kelesey, B. Schneier, D. Wagner. In N.Kobliz, editor, „Advances in Cryptology – Crypto'96", volume 1109 of Lecture Notes in Computer Science, pag. 237-251, New York, Springer-Verlag, 1996.

68. [Konh-81] A.G. Kohnheim: "Cryptography: A Primer", New-York, John Wiley & Sons, Inc., 1981.

69. [KnBe-95] L.R. Knudsen, T. Berson. „Truncated differentials of SAFER". In D.Gollman, editor, Fast Software Encryption, volume 1008 of Lecture Notes in Computer Science, pag. 196-211, Springer-Verlag, 1995.

70. [KnMe-96] L.R. Knudsen, W. Meier. „Improved differential attacks on RC5". In N.Koblitz, editor, Advances in Cryptology – Crypto'96, vol. 1109 of Lecture Notes in Computer Science, pag. 216-228, Springer-Verlag ,1996.

71. [KnMe-99] L.R. Knudsen, W. Meier. „Corelations in RC6", Technical Report 177, Department of Informatics, University of Bergen, Norway, 1999.

72. [KnRo-96] L.R. Knudsen, M.J.B. Robshaw. „Non-linear approximations in linear cryptanalysis". In U. Maurer, editor, Advances in Cryptology - Eurocrypt'96, volume 1070 of Lecture Notes in Computer Science, pag. 224-236, Springer-Verlag, 1996.

73. [Knud-94] L.R. Knudsen. „Block Ciphers – Analysis, Design and Applications". PhD thesis, Aarhus University, 1994.

74. [Knud-95] L.R. Knudsen. „Applications of higher order differentials and partial differentials". In B. Preneel, editor, Fast Software Encryption, volume 1008 of Lecture Notes in Computer Science, pag. 196-211, Springer-Verlag, 1995.

75. [Knud-98] L.R. Knudsen. „DEAL – A 128-bit Block Cipher", NIST AES Proposal, 1998.

76. [Lai-92] X. Lai. „On the design and security of Block Ciphers". PhD thesis, ETH, Zurich, 1992.

77. [Lapr-85] J.C. Laprie. „Dependable computing and fault tolerance: concepts and terminology", Proceedings of the 15th Anual International Symposium on Fault Tolerant Computing, Ann Arbor, Michingan, pag. 2-11, June 1985.

78. [LaHe-94] S.K. Langford, M.E. Hellman. „Differențial-linear cryptanalysis". In Y.G.Desmedt, editor, Advances in Cryptology – Crypto'94, volume 839 of Lecture Notes in Computer Science, pag. 17-25, Springer-Verlag, 1994.

79. [LaMa-90] X. Lai, J.L. Massey. "A Proposal for a New Block Encryption Standard". Proc. EuroCrypt'90, Aarhus, Denmark, 1990.

80. [LaMa-91] X. Lai si J. L. Massey. „On the security of a proposed encryption standard against differential cryptanalysis". Proc. EuroCrypt '91, Brighton, UK, 1991.

81. [LaMM-94] X. Lai, J.L. Massey, S. Murphy. „Markov ciphers and differential crypt-analysis". In D.W.Davies, editor, Advances in Cryptology-Eurocrypt'91, volume 547 of Lecture Notes in Computer Science, pages 17-38, Berlin, Springer-Verlag, 1992.

82. [LeSh-96] Chung-Len Lee, Meng-Lieh Sheu. „A Multiple-Sequence Generator Based on Inverted Nonlinear Autonomus Machines", IEEE Transactions on Computers, vol. 45, no. 9, pag. 1079-1083, 1996.

83. [Leib-99] L. Leibrock. „Exploratory Candidate Algorithm Performance Characteristics in Commercial Symmetric Multiprocessing (SMP) Environments for the advanced Encryption Standard (AES)", 1999.

84. [Lim-98] C.H. Lim. „"CRYPTON": A New 128-bit Block Cipher", NIST AES Proposal, 1998.

85. [Lipm-99] H. Lipmaa. „AES Candidates: A survey of Implementations". Second Advanced Encryption Standard Candidate Conference, National Institute of Standard and Technology (NIST), March 1999.

86. [Lo-96] Jien-Chung Lo. „A Hyper Optimal Encoding Scheme for Self-Checking Circuits", IEEE Transactions on Computers, vol. 45, no. 9, pag. 1022-1029, 1996.

87. [Lo-97] Jien-Chung Lo. „A Fast Binary Adder with Conditional Carry Generation", IEEE Transactions on Computers, vol. 46, no. 2, pag. 248-253, 1997.

88. [Lo-98] Jien-Chung Lo. Correction to "A Fast Binary Adder with Conditional Carry Generation", IEEE Transactions on Computers, vol. 47, no. 12, pag. 1425, 1998.

89. [MMMC-83] I. Mang, E. Mang, A. Mezei, C. Chioreanu. „Unitate de comandă microprogramată cu microprocesorul I3000, pentru un testor de plachete echipate". A XXIV-a sesiune de comunicări ştiințifice studențeşti - Timişoara, 27-28 mai 1983.

90. [MaMa-93] E. Mang, I. Mang. „Implementarea operațiilor booleene prin diagrame ordonate de decizie binară", Analele Universității din Oradea, pag. 459-464, Editura Universității din Oradea, 1993.

91. [MaMa-94] E. Mang, I. Mang. "Programmable Logic Array", FEI'25 Conference on electronic, computers and informatics, pag. 120-125, Kosice -Slovakia, 22-23 sept., 1994.

92. [Mang-94a] E. Mang. „Algoritm de minimizare logică a PLA", Analele Universității din Oradea, pag. 543-546, Editura Universității din Oradea, 1993.

93. [Mang-94b] E. Mang. „Analiza si sinteza circuitelor logice" – curs. Editura Uiversitatii Oradea, 1994.

94. [MaMa-95a] E. Mang, I. Mang. „Diophantine equations and public-key cipher systems", Proceding of 113th Panonian Applied Mathematical Meeting", pag.125-129, Slovakia, 1995.

95. [MaMa-95b] E. Mang, I. Mang. „Circuitul sumator-comparator fără propagarea transportului", A III-a conferință internațională de ingineria sistemelor moderne în electrotehnică - Oradea, publicată în "Analele Universității Oradea, fascicola Electrotehnică", pag. 91-96, 2-4 iunie 1995.

96. [MaMa-95c] I. Mang, E. Mang. „Metodă de protecţie a fişierelor în calculator", A III-a conferință internațională de ingineria sistemelor moderne în electrotehnică - Oradea, publicată în "Analele Universității Oradea, fascicola Electrotehnică", pag. 97-101, 2-4 iunie 1995.

97. [MaMa-96] E. Mang, I. Mang. „Prime numbers and secure public-key cryptographic parameters", publicată în Proceedings of Workshop, pag. 43-53, 25-27 aprilie, Kosice-Slovakia, 1996.

98. [MaMa-97] E. Mang, I. Mang. „Encryption algorithm with LFSR", EMES'97, Oradea, "Analele Universității Oradea, fascicola Electrotehnică", pag. 267-271, mai 1997.

99. [MaMȚ-99a] E. Mang, I. Mang, R. Țirtea. „Proiectarea logică cu FPGA - lucrări practice", Universitatea din Oradea, 1999.

100. [MaMȚ-99b] E. Mang, I. Mang, R. Țirtea. „Tehnica securității datelor" – îndrumător de laborator, Universitatea din Oradea, 1999.

101. [MaMȚ-00] E. Mang, I. Mang, R. Țirtea. „A Comparative Analysis of the Fifteen Advanced Encryption Standard Candidates", lucrare acceptată la sesiunea de comunicări SINTES'10, Craiova, 2000.

102. [Mang-97] E. Mang. „Analiza stadiului actual al tehnicilor de securizare a informatiei", Referat de doctorat, Universitatea "Politehnica" Timişoara, 1997.

103. [Mang-98] E. Mang. „Analiza metodelor de criptare/decriptare a informației şi elemente de sinteză pentru implementarea securizării informatiei", Referat de doctorat, Universitatea "Politehnica" Timişoara, 1998.

104. [Mang-00] E. Mang. „Analysis of Suitability for pseudorandom BIST of the RC6 Cipher". Lucrare acceptată la Simpozionul ECI 2000, Kosice-Slovakia, sept. 2000.

105. [MaȚi-00b] E. Mang, R. Țirtea. „CRIPTOR - VLSI implementation of the RC6 Block Cipher", Lucrare acceptată la Simpozionul ECI 2000, Kosice-Slovakia, sept. 2000.

106. [MaȚi-00a] E. Mang, R. Țirtea. „Proiectarea în VHDL – lucrări practice", Universitatea din Oradea, 2000.

107. [Mang-99] I. Mang. „Introducere în tehnica securității datelor", Editura Universității din Oradea, 1999.

108. [Mano-84] M. Mano. "Digital Design", Prentice Hall, 1984.

109. [MaPo-88] R. Marlett, S. R. Pollock. "Guaranteeing ASIC testability, VLSI System Design", August 1988.

110. [Mass-88] J.L. Massey. "An Introduction to Contemporary Cryptology". Proceedings of the IEEE, pag. 533-549, May 1988.

111. [Mass-93] J.L. Massey. „Cryptography: Fundamentals and Applications", Advanced Technology Seminars, Zürich, Switzerland, 1993.

112. [Mass-94] J. Massey, SAFER K-64: „A Byte-Oriented Block-Ciphering Algorithm". R.Anderson, editor, Fast Software Encription, volume 809 of Lecture Notes in Computer Science, pag 1-17, Springer Verlag, 1994.

113. [Mats-94] M. Matsui. „The First Experimental Cryptanalysis of the Data Encryption Standard". In Y.G.Desmedt, editor, Advances in Cryptology/ Crypto'94, volume 839 of Lecture Notes in Computer Science, pag. 1-11, New York, Springer-Verlag, 1994.

114. [Maur-90] U.M.Maurer. „Fast generation of secure RSA-moduli with almost maximal diversity", Advances in Cryptology - EUROCRYPT '89, Lecture Notes in Computer Science, Vol.434, pag. 636-647, Berlin: Springer-Verlag, 1990.

115. [Maur-92] U.M.Maurer. „Some number-theoretic conjectures and their relation to the generation of cryptographic primes, in Cryptography and Coding II", C. Mitchell (ed.), pag.173-191, Oxford Univeristy Press, 1992.

116. [MaYa-91] U.M.Maurer, Y. Yacobi. „Non-interactive public-key cryptography, Advances in Cryptology - EUROCRYPT '91", Lecture Notes in Computer Science, Vol.547, pag.498-507, Berlin: Springer-Verlag, 1991.

117. [McCl-85] E. J. McCluskey. „Built-in-self-test techniques". IEEE Design & Test, pag 21-28, April 1985.

118. [McCu-90] K.McCurley. „The discrete logarithm problem, in Cryptology and computational number theory", C.Pomerance (ed.), Proc.of Symp. in Applied Math., Vol.42, pag.49-74, American Mathematical Society, 1990.

119. [Mokr-99] P. Mokros. „Comparative Analysis of the Advanced Encryption Standard Candidate Algorithms", Comments on the First Round AES Evaluation, April, 1999.

120. [MoYa-91] H. Morita, M. Yamane. „Hardware approach to fast encipherment processing and its implementation". IEICE Transactions, vol. E 74, nr. 8, pag. 83-93, 1991.

121. [NaAb-90] V. S. S. Nair, J. A. Abraham. „Real-Numbers Codes for Fault-Tolerant Matrix Operations on Processor Arrays", IEEE Trans. on Computers, vol. 39, nr. 4, pag. 426-435, 1990.

122. [NBDDFR-99] J. Nechvatal, E. Barker, D. Dodson, M. Dworkin, J. Foti, E. Roback – „Status Report on the First Round of the Development of the Advanced Encryption Standard", 1999.

123. [NIST-97] Federal Register, Department of Commerce: National Institute of Standards and Technology. Volume 62, Number 177, September 12, 1997.

124. [NTT-98] Nipon Telephone and Telegraph. „Specification of E2 – a 128-bit Block Cipher", NIST AES Proposal, 1998.

125. [Parh-90] B. Parhami. Generalized Signed-Digit Number Systems: „Unifying Framework for Redundant Number Representations", IEEE Trans. On Computers, vol. 39, nr. 1, pag. 89-98, 1990.

126. [Patr-94] V.V. Patriciu. „Criptologia şi securitatea reţelelor de calculatoare, cu aplicaţii în C şi Pascal", Editura Tehnică, Bucureşti, 1994.

127. [PhSe-96] Philips Semiconductors. „80C51 Family Programmer's Guide and Instruction Set", Nov., 1996.

128. [PhVa-94] J.Phillips, S. Vassiliadis. „High Performance 3-1 Interlock Collapsing ALU's", IEEE Trans. on Computers, vol. 43, nr. 3, pag. 257-268, 1994.

129. [Pfit-96] B. Pfitzmann. „Digital Signatures Schemes – General Framework and Fail-Stop Signatures", Lecture Notes in Computer Science, Springer Verlag, 1996.

130. [Prad-86] D.K. Pradhan. „Fault-Tolerant Computing: Theory and Techniques", Prentice Hall Publishing Company, New-York, 1986.

131. [RaFu-89] T.R.N.Rao, C. Fujiwara. „Error-Control Coding for Computer Systems", Prentice Hall International Inc., 1989.

132. [RaTy-96] J.Rajski, J.Tyszer. „On Linear Dependencies in Subspaces of LFSR-Generated Sequences", IEEE Transactions on Computers, vol. 45, no. 10, pag. 1212-1216, 1996.

133. [Rive-96] R.L. Rivest. „The RC5 Encription Algorithm". In B.Preneel, editor, Fast Software Encryption, volume 1008 of Lecture Notes in Computer Science, pag. 86-96, Springer Verlag, 1996.

134. [RoBo-91] R. Rogenmoser, H. Bonnenberg. „Entwurf eines Kryptographie Ics mit 8-bit Architektur". Master ' s thesis , ETH Zürich, 1991.

135. [RRSY-98] R. Rivest, M.J.B. Robshaw, R. Sidney, Y.L..Yin, „The RC6 Block Cipher", M.I.T.Laboratory for Computer Science, RSA Laboratories, 1998.

136. [RuGa-91] D. Russell, G.T. Gangemi Sr. „Computer Security Basics", O'Reilly & Associates, Inc., 1991.

137. [Salo-93] A. Salomaa. „Criptografie cu chei publice", Editura Militară, Bucureşti, 1993.

138. [Saar-98] M.J.O. Saarinen. „A Note Regarding the Hash Function Use of MARS and RC6". AES-Submission, 1998.

139. [Schn-94] B. Schneier. „Applied Cryptography". John Wiley & Sons, New York, 1994.

140. [Schn-96] B. Schneier. „Applied Cryptography. Protocols Algorithms and Source Code in C", John Wiley & Sons Inc., 1996.

141. [Selc-98] A.A. Selcuk. „New Results in Linear Cryptanalysis of RC5". S.Vaudenay, editor, Fast Software Encryption, volume 1372 of Lecture Notes in Computer Science, pag. 1-16, Springer-Verlag, 1998.

142. [Shan-49] C.E. Shannon. "Communication Theory of Secrecy Systems". Bell System Technical Journal, vol. 28, pag. 656-715, 1949.

143. [SKWW-99a] B. Schneier, J. Kelsey, D. Whiting, D. Wagner, C. Hall, N. Ferguson, „Performance Comparision of the AES Submisions", version 2.0, february 1999.

144. [SKWW-99b] B. Schneier, J. Kelsey, D. Whiting, D. Wagner, C. Hall, N. Ferguson. "Twofish: A 128-bit Block Cipher", NIST AES Proposal, 1998.

145. [SKWWHF-99] B. Schneier, J. Kelsey, D. Whiting, D. Wagner, C. Hall, N. Ferguson, „Performance Comparision of the AES Submisions", The second AES Conference, pag 15-34; 1999.

146. [Spec-91] „Specifications for a digital signature standard", US Federal Register, Vol. 56, No. 169, August 30, 1991.

147. [Stin-96] D.R. Stinson. „Cryptography – Theory and Practice". CRC Press, 1996.

148. [StFK-93] C. H. Stapper, J. A. Fifield, H.L.Kalter, W.A. Klassen. „High-Reliability 16-Mbit Memory Chip", IEEE Trans. on Reliability, vol. 42, nr. 4, pag. 596-603, 1993.

149. [Tsui-87] Fr. Tsui. LSI/VLSI Testability Design, Mc. Graw Hill Book Company, 1987.

150. [Vasa-98] Vasarhely J. „Proiectarea cu circuite logice programabile", Casa de Editura Albastra, 1998.

151. [Vasi-95] L. Vasiu. „Configurarea nucleului hardware de fiabilitate sporită pentru echipamente de calcul tolerante la defecte", Referat de doctorat, 1995, Universitatea Tehnică Timişoara.

152. [Verb-91] I. Verbauwhede. „VLSI Design Methodologies for Application-specific Cryptographic and Algebraic Systems". PhD thesis, Katholieke Universiteit Leuven, 1991.

153. [Vlăd-86] M. Vlăduțiu, "Tehnica testării echipamentelor de calcul şi evaluarea performanțelor", Curs litografiat, IPTV Timişoara, 1986.

154. [Vlăd-89] M. Vlăduțiu, "Tehnologie de ramură şi fiabilitate", I.P.T.V. Timişoara, curs litografiat, 1989.

155. [VlCr-89] M. Vlăduțiu, M. Crişan, "Tehnica testării echipamentelor automate de prelucrare a datelor", Ed. Facla, Timişoara, 1989.

156. [VlGr-81] M. Vlăduțiu, V. Groza, "Tehnica testării echipamentelor de calcul şi evaluarea performanțelor", Îndrumător de lucrări, Litografia IPTV, 1981.

157. [VlMa-83] M. Vlăduțiu, I. Mang: "Structura unui sistem de testare a plachetelor echipate Al 2-lea Simpozion de tehnologii şi echipamente de testare automată" - Cluj Napoca, pag. 273-281, 11-12 nov. 1983.

158. [WoGo-94] W. F. Wong, E. Goto. "Fast Hardware-Based Algorithms for Elementary Function Computations Using Rectangular Moltipliers", IEEE Trans. on Computers, vol. 43, nr. 3, pag. 278-294, 1994.

159. [WrMi-93] A. Wrzyszcz si D. Milford. „A New Modulo 2^a+1 Multiplier". Proc. IEEE Int'l Conf. Computer Design: VLSI in Computers and Processors, 1993.

160. [Xili-92] Xilinx Coorporation. „XACT, Logic Cell Array Macro Library", 1992.

161. [Xili-98a] Xilinx Coorporation. „Foundation Series User Guide", 1998.

162. [Xili-98b] Xilinx Coorporation. „Development System User Guide", 1998.

163. [Xili-98c] Xilinx Coorporation. „The Programmable Logic Data Book", 1998.

164. [Xili-98d] Xilinx Coorporation. „Hardware User Guide", 1998.

165. [Xili-98e] Xilinx Coorporation. „Foundation Series Quick Start Guide 1.5", 1998.

166. [Xili-98f] Xilinx Coorporation. „Verilog Reference User Guide",1998.

167. [Xili-98g] Xilinx Coorporation. „VHDL Reference Guide", 1998.

168. [Xili-98h] Xilinx Coorporation. „Timing Analizer Reference User Guide", 1998.

169. [Xili-98i] Xilinx Coorporation. „Design Manager/Flow Engine Reference/User Guide", 1998.

170. [Yarb-97] J. Yarbrough. "Digital Logic, Applications and Design", Oregon Institute of Technology, West Publishing Company, 1997.

171. [Zimm-92] R. Zimmermann. „Full custom 8×64-bit shift register". Technical Report 92/24, Integrated Systems Laboratory, ETH Zürich, December 1992.