Universitatea “Politehnica” din Bucureşti
Facultatea de Electronică, Telecomunicaţii şi Tehnologia Informaţiei

Implementarea pe o placă Altera Cyclone I a unui decodor LDPC şi analiza performanţelor

Proiect de diplomă

prezentat ca cerinţă parţială pentru obţinerea titlului de
Inginer în domeniul *domeniul Calculatoare si Tehnologia Informaţiei*
programul de studii de licenţă *program Ingineria Informaţiei*

Conducător ştiinţific
Conf. Dr. Ing. Ştefan STĂNCESCU

Absolvent
George CRĂCIUN

Anul 2014

Anexa 1

Universitatea "Politehnica" din Bucureşti
Facultatea de Electronică, Telecomunicaţii şi Tehnologia Informaţiei
Departamentul: Electronica Aplicata si Ingineria Informatiei

Aprobat Director de Departament :

prof. dr. ing. Sever PAŞCA

TEMA PROIECTULUI DE DIPLOMĂ
a studentului Craciun N. George, 443A

1. Titlul temei: Implementarea pe o placa Altera Cyclone I a unui decodor LDPC şi analiza performanţelor

2. Contribuţia practică, originală a studentului va consta în: Studiul teoretic al algoritmilor de codare/decodare LDPC. Implementarea se va face pe un chip FPGA de tip Altera Cyclone I (EP1C6Q240C8) folosind software-ul de dezvoltare Quartus II în 2 etape: scrierea codului HDL şi compilarea acestuia într-un bloc funcţional; sinteza şi implementarea blocului pe chip-ul FPGA. Prin intermediul simulatorului CAMAG6 se vor compara datele de codat cu datele decodate, se va calcula frecvenţa de erori in raport cu frecvenţa de transmisie si raportul semnal-zgomot.

3. Proiectul se bazează pe cunoştinţe dobândite în principal la următoarele 3 discipline:

- Circuite Integrate Digitale
- Teoria Transmisiunii Informaţiei
- Inginerie Software

4. Realizarea practică/proiectul rămân în proprietatea: Universitatea Politehnica din Bucureşti

5. Proprietatea intelectuală asupra proiectului aparţine: Universitatea Politehnica din Bucureşti

6. Locul de desfăşurare a activităţii: Complex Leu, corp B, sala 234

7. Data eliberării temei: 10.12.2013

CONDUCĂTOR LUCRARE:

COORDONATOR:
Conf. Dr. Ing. Ştefan STĂNCESCU

STUDENT:

George CRĂCIUN

Anexa 5

Declaraţie de onestitate academică

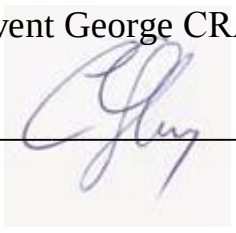
Prin prezenta declar că lucrarea cu titlul "*Implementarea pe o placă Altera Cyclone I a unui decodor LDPC şi analiza performanţelor*", prezentată în cadrul Facultăţii de Electronică, Telecomunicaţii şi Tehnologia Informaţiei a Universităţii "Politehnica" din Bucureşti ca cerinţă parţială pentru obţinerea titlului de *Inginer* în domeniul *Calculatoare si Tehnologia Informaţiei*, programul de studii *Ingineria Informaţiei* este scrisă de mine şi nu a mai fost prezentată niciodată la o facultate sau instituţie de învăţământ superior din ţară sau străinătate.

Declar că toate sursele utilizate, inclusiv cele de pe Internet, sunt indicate în lucrare, ca referinţe bibliografice. Fragmentele de text din alte surse, reproduse exact, chiar şi în traducere proprie din altă limbă, sunt scrise între ghilimele şi fac referinţă la sursă. Reformularea în cuvinte proprii a textelor scrise de către alţi autori face referinţă la sursă. Înţeleg că plagiatul constituie infracţiune şi se sancţionează conform legilor în vigoare.

Declar că toate rezultatele simulărilor, experimentelor şi măsurătorilor pe care le prezint ca fiind făcute de mine, precum şi metodele prin care au fost obţinute, sunt reale şi provin din respectivele simulări, experimente şi măsurători. Înţeleg că falsificarea datelor şi rezultatelor constituie fraudă şi se sancţionează conform regulamentelor în vigoare.

Bucureşti, 02.07.2014

Absolvent George CRĂCIUN



Cuprins

Lista figurilor

Lista acronimelor

AWGN - Additive white Gaussian noise
BER - Bit Error Rate
CPU - central processing unit (Unitatea centrală de prelucrare)
DVB - Digital Video Broadcasting (televiziunea digitală)
FET - field-effect transistor (Tranzistorul cu efect de câmp)
FER - Frame Error Rate
FPGA - Field Programmable Gate Array (circuit integrat digital configurabil)
HDL - Hardware Design Language
IDE - Integrated Drive Electronics
IEEE - Institute of Electrical and Electronics Engineers
JTAG - Joint Test Action Group
LAN - Local Area Network
LCD - Liquid Crystal Display
LDPC - Low-Density Parity Check
LLRs - Log Likelihood Ratios
LUT - Look-Up Table
ML - Maximum-Likelihood
MIMO - Multiple Input Multiple Output
MSB - Most Significant Bit
PCI - Peripheral Component Interconnect
PROM - Programmable Read-Only Memory
PWL - Piece-wise Linear
RAM - Random Access Memory
ROM - Read-Only Memory
SDRAM - Standard Dynamic Random Access Memory
SNR - Signal-to-Noise Ratio (raportul semnal zgomot)
SRAM - Static Random Access Memory
VLSI - Very Large Scale Implementation
VHDL - Very High Speed Integrated Circuit Hardware Description Language (abrevierea VHSIC HDL)
VGA - Video Graphics Array
USB - Universal Serial Bus

Prezentare

Acest proiect are ca scop implementarea unui decodor LDPC (Low Density Parity Check) pe un FPGA (Field Programmable Gate Array). Caracteristica de concepție, cheie, a decodorului este flexibilitatea, fiind capabil să suporte diferite coduri și calcule. Avantajul capabilității de a testa o varietate de coduri, precum cele găsite în standardele de comunicare inclusiv Wireless LAN (IEEE 802.lln), WIMAX (IEEE 801.16e) și DVB-S2, pe o singură platformă hardware va face ca trecerea de la teorie la practică, să fie una mult mai fluidă.

Conceptul decodorului LDPC este flexibil, poate să accepte un cod nou fără un nou calcul, singurul lucru care se schimbă este configurație ROM. Testarea se face prin intermediul interfeței seriale Matlab. Decodorul a fost, de asemenea, făcut să funcționeze, corect, pe coduri mari și, deoarece cerințele de performanță cresc, a fost nevoie de o implementare la scală mai mare.

Contributie

Acest proiect implementează hardware, un decodor flexibil iterativ bazat pe o publicație a lui Alberto Tarable [1]. Autorul a adus următoarele contribuții la acest proiect:

1. Înțelege, dezvoltă și implementează în MATLAB, un algoritm de alocare a resurselor unui sistem care permite flexibilitatea în decodorul LDPC.
2. Înțelege, dezvoltă și implementează în MATLAB, un algoritm care configureaza comutatoarele crossbar Benes.
3. Dezvoltă si implementează în VHDL operațiunea comutatoarelor crssbar Benes.
4. Concepe o arhitectură complet flexibilă a decodorului LDPC.
5. Dezvoltă și simulează buna funcționare a decodorului LDPC în VHDL.

Capitolul 1

1. Introducere

1.1 Scopul lucrării

Acest proiect detaliază designul, punerea în aplicare și testarea unui decodor Low Density Parity Check (LDPC) flexibil, construit pe un circuit integrat digital configurabil (FPGA). Scopul proiectului este acela de a implementa un decodor LDPC flexibil, care poate să suporte diferite tipuri de coduri, care permite testarea unei varietăți de coduri, fără să fie nevoie reproiectarea decodorului. Unul dintre scopurile de lungă durată al proiectului, este să integreze un decodor Multiple Input Multiple Output (MIMO).

1.1.1 Condiţii impuse de realizare

Un decodor flexibil LDPC a fost conceput în VHDL folosind software-ul Altera Quartus II 7.0. Programul și configurația ROM au fost implementate în MATLAB. Decodorul a fost implementat pe o placă Altera Cyclone I (EP1C6Q240C8). Conceptul se poate verifica printr-o interfață serială MATLAB a testbench-ului, pentru a confirma operațiile corecte ale decodorului.

Kit-ul UP3 oferă un sprijin educațional puternic și, de asemenea, o soluție low-cost pentru prototipuri și o rapidă dezvoltare a produselor. Placa servește ca un mijloc excelent pentru realizarea de prototipuri de sistem, emulare și hardware, precum și dezvoltarea de software. Oferă domeniul de aplicare pentru proiectare hardware, folosind HDL, cum ar fi Verilog sau VHDL.

Întregul mediul de dezvoltare ajuta să implementeze rapid orice procesor, precum și orice sistem de operare în timp real pe kit. Placa are interconexiuni standard, subsistem de memorie, ceasuri multiple pentru proiectarea sistemului, JTAG de configurare, antete de expansiune pentru o mai mare flexibilitate și capacitate. Placa poate fi folosit pentru aplicații DSP cu interfață directă la un procesor DSP sau de implementarea funcțiilor DSP în interiorul FPGA-ului. Pe scurt, este un kit cu dublu scop, care pot fi utilizat pentru realizarea de prototipuri și dezvoltarea modelelor VLSI precum și proiectarea și dezvoltarea de sisteme embedded.

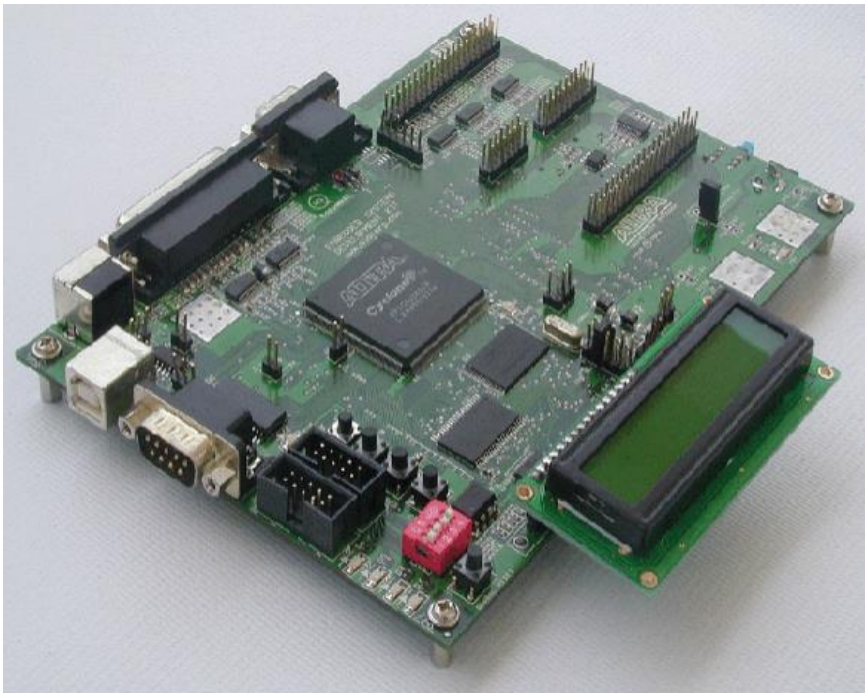


Figura 1.2: UP3 Education Kit

Câteva din caracterisiticile kitului educațional UP3:

- USB 1.1 (Full speed & Low speed)
- port RS 232 (Full Modem)
- port paralel (IEEE1284)
- port PS/2
- port VGA
- IDE (Integrated Drive Electronics)
- SRAM de 128 KB (64Kx16)
- FLASH de 2MB (1Mx16)
- SDRAM de 8MB (4Mx16)
- IC2 PROM de 8MB (Expandabilă)
- suportă diferite ceasuri de peferat ceasul PCI,USB și CPU
- capabilitatea de descărcare JTAG și Active Serial
- patru leduri
- un display LCD 16x2

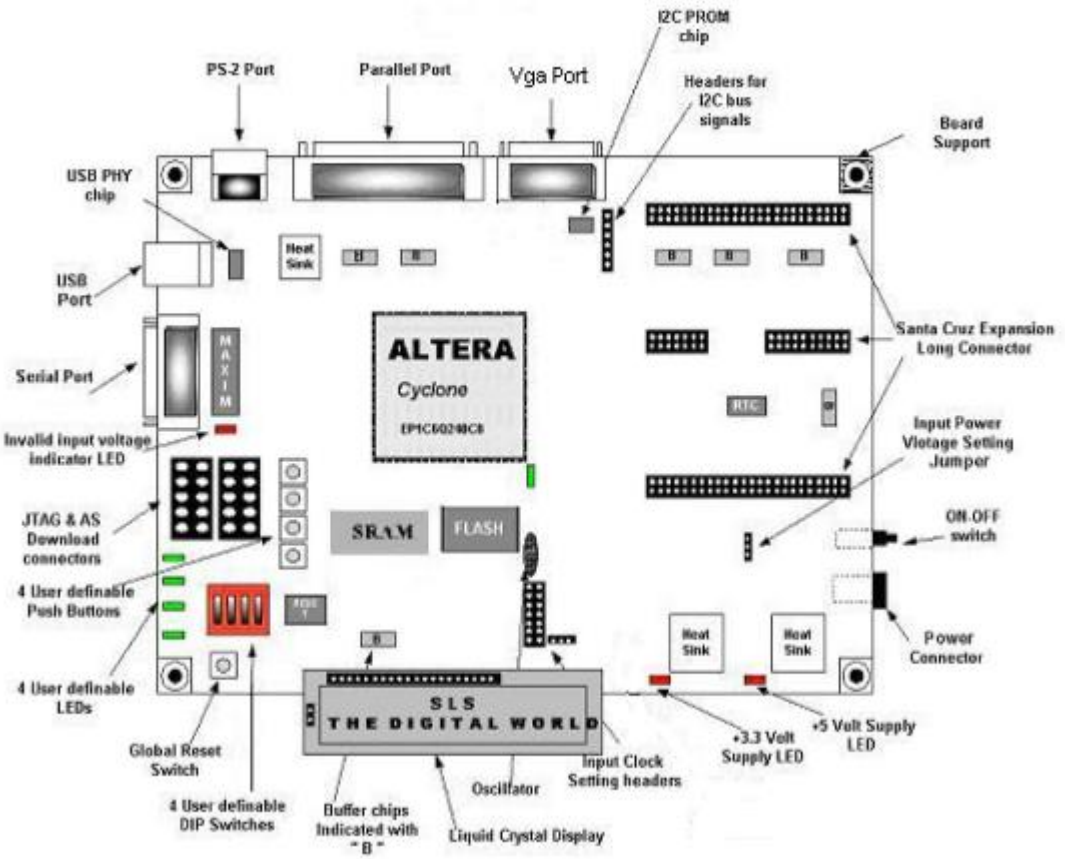


Figura 1.3: Placă Altera Cyclone I (EP1C6Q240C8)

1.1.2 Coduri corectoare de erori

Toate codurile corectoare de erori se bazează pe acelaşi principiu: informaţiei îi este adăugată redundanţă, pentru a încerca corectarea eventualelor erori care pot apărea în procesul de stocare sau de transmisie. În principiu, simboluri redundante sunt ataşate simbolurilor de informaţie, pentru a se obţine o secvenţă codată sau un cuvânt de cod. În figura 1.1 este prezentat un cuvânt de cod obţinut prin codare cu un bloc cod. O astfel de codare se numeşte sistematică.

Asta înseamnă că simbolurile de informaţie vor apărea întotdeauna în primele k poziţii ale cuvântului de cod. Cele (n-k) simboluri rămase din cuvântul de cod sunt anumite funcţii de simboluri de informaţie si asigură redundanţă, care poate fi folosită pentru detectare/corectare erorilor. Mulţimea tuturor secvenţelor de cod de acest fel se numeşte cod corector de erori, şi va fi notat cu C.

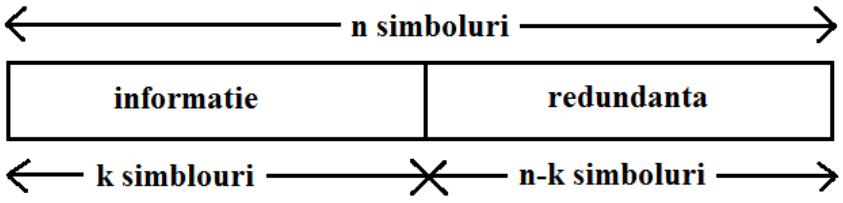


Figura 1.1: Codare bloc sistematică pentru corecţia erorilor

Codurile corectoare de erori reprezintă un mod cu ajutorul căruia erorile introduse într-un sistem de comunicații, pot fi detectate și corectate de receptor. Este o parte importantă a sistemului de comunicare, esențială în baza de informații a societății noastre moderne. Compact Disks, CD-ROM's, DVD's, telefoanele mobile și driverele hard-disk, toate folosesc coduri corectoare de erori pentru a permite transferul de informații sigure.

În 1948 Shannon a publicat un articol, care a revoluționat comunicatile. A aflat că există o limită fundamentală a cât de multe informații, corecte, pot fi trimise pe un canal, cu referire la zgomotul prezent. Deși poate fi definită o limită, dovadă este non-constructiva, arată doar cât de bine poate performa cel mai bun cod posibil. Pentru a demonstra capacitatea canalului, Shannon a folosit diferite coduri, de lungimi infinite; în timp ce acestea nu sunt practice, el sugerează că, pentru a găsi un sistem de codificare care se apropie de această limită fundamentală, ar trebui să folosim coduri aleatoare de lungimi mari.

Codurile Low-Density Parity-Check (LDPC) sunt o clasă de coduri bloc care îndeplinesc ambele condiții, având o lungime mare și fiind aleatoare. Aceste coduri au cele mai bune performanțe până în prezent, cu capacitatea lor de 0.0045dB, între limita lui Shannon [3]. Performanța lor excelentă, împreună cu complexitatea lor redusă de decodare, care au văzut-o codurile LDPC incluse în standardele pentru transmiterea datelor, precum WIMAX (IEEE 802.16e), Digital Video Broadcasting (DVB-S2) și ca un candidat pentru noul standard de Wireless LAN (IEEE 802.n) și în noile standarde ale telefoanelor mobile (3G-Long Term Evolution). În timp ce codurile LDPC își găsesc drumul în standardele de transmisie de date, în prezent nu există niciun hardware universal pe care un producător poate să-l cumpere pentru a dezvolta astfel de dispozitive care poat utiliza codurile LDPC.

Unul dintre țelurile principale a fost de a realiza un decodor LDPC hardware, flexibil (abilitatea de decodifica diferite coduri fără a reorganiza hardware-ul), punând teoria în practică.

1.2 Coduri bloc şi coduri convoluţionale

În funcţie de metoda prin care redundanţa este adaugată mesajului, codurile corectoare de erori se împart în două clase: coduri bloc şi coduri convoluţionale. Ambele tipuri de codări au aplicaţii practice. De-a lungul timpului, codurile convoluţionale au fost preferate, aparent datorită posibilităţii implementării algoritmului de decodare Viterbi, şi datorită părerii conform căreia codurile bloc nu vor putea fi decodate eficient cu decizii soft. Totuşi, descoperirile recente în teoria şi dezvoltarea algoritmilor de decodare cu decizii soft pentru codurile bloc liniare au înlăturat această neîncredere. Mai mult, cele mai bune coduri corectoare de erori cunoscute astăzi sunt blocurile cod (coduri neregulate cu densitate mică şi control al parităţii).

1.2.1 Coduri convoluţionale

Codurile convoluţionale diferă de codurile bloc atât prin faptul că există memorie pentru codor, cât si prin dependenţă celor n ieşiri, la fiecare unitate de timp, nu numai de cele k intrări, ci şi de cele m blocuri anterioare de intrare.

Un cod convoluţional (n,k,m) poate fi implementat cu un circuit secvenţial care are n ieşiri liniare, k intrări şi m intrări de memorie. Uzual, n şi k sunt valori întregi cu $k < n$, dar valoarea lui m trebuie să fie mai mare pentru a avea probabilitate de eroare mică. În cazul în care k=1, secvenţa de informaţie nu mai este divizată în blocuri şi poate fi procesată în mod continuu. Codurile convoluţionale au fost introduse în 1955 de către Elias ca o alternativă a codurilor bloc. La scurt timp după aceea, Wozencraft a propus decodarea secvenţială ca fiind o metodă eficientă pentru codurile convoluţionale. În 1963, Massey a propus o metodă de decodare mai puţin eficientă, dar mai simplu de implementat , numită decodarea cu prag. Aceasta a dat naştere numeroaselor aplicaţii ale codurilor convoluţionale în cadrul transmisiilor digitale prin cablu sau unde radio. În 1967, Viterbi a propus o metodă de decodare de plauzibilitate maximă, uşor de implementat pentru codurile cu memorie mică. Această schemă, denumită decodorul Viterbi, împreună cu versiunea îmbunătăţită a decodării secvenţiale, au lărgit şi mai mult domeniul de aplicaţie al codurilor convoluţionale spre comunicaţiile prin satelit, la începutul anilor 1970.

1.2.2 Coduri bloc

Codurile bloc procesează informaţia bloc-cu-bloc, tratând fiecare bloc de biţi de informaţie separat de celelalte. Cu alte cuvinte, codarea bloc este o operaţie fără memorie, în sensul că toate cuvintele de cod sunt independente unul de celălalt. În schimb, rezultatele codărilor convoluţionale depind nu numai de informaţia de intrare în momentul actual, dar şi de intrări si rezultate precedente, fie într-un mod bloc-cu-bloc, fie bit-cu-bit. Trebuie menţionat că şi codurile bloc au memorie, când sunt proiectate ca un proces bit-cu-bit, în cadrul cuvântului de cod. În ultima perioadă, în schimb, diferenţele dintre codurile bloc şi cele convoluţionale sunt din ce în ce mai mici, mai ales în urma recentelor avansuri în înţelegerea structurilor trellis şi a unor structuri circulare convoluţionale. Într-adevăr, unii cercetători care lucrează cu coduri convoluţionale le numesc uneori coduri cu structură trellis, variabilă în timp, în timp ce unii cercetători care lucrează cu coduri bloc le denumesc pe cele din urmă - coduri cu structură trellis regulată.

1.3 Coduri LDPC

Codurile Low Density Parity Check sunt o formă a unor coduri premature de corectare a erorilor, descoperite prima oară de Gallager în 1962 [4]. Ele sunt coduri bloc a căror matrice de verificare a parității (H) conține foarte puține entități non-zero. Densitatea redusă a lui H care garantează complexitatea decodificării și distanța minimă, ambele crescând liniar cu lungime blocului.

Cea mai mare diferență între codurile LDPC și alte coduri bloc este despre cum sunt decodificate. Codurile bloc tradiționale sunt decodate în mod normal folosind anumite forme de algoritmi Maximum-Likelihood (ML). Codurile LDPC sunt decodate iterativ, folosind proprietățile grafice ale unei matrice de verificare a parității H.

1.3.1 Prezentarea unui model al sistemelor de comunicaţie

Un sistem de comunicaţie complet include numeroase părţi componente care pot ridica probleme interesante şi greu de rezolvat. Cele mai multe sisteme de comunicaţie moderne lucrează în domeniul digital, acesta oferind o mulţime de posibilităţi de procesare a semnalului. În general, pentru obţinerea de performanţe optime, codorul sursei, codorul de canal şi modulatorul ar trebui considerate ca făcând parte din aceeaşi funcţie a sistemului şi nu ca blocuri separate. Shannon[13] a demonstrat că partea de codare a sursei şi partea de codare de canal pot fi separate în unităţi individuale fără a se pierde la capitolul optimizare. Aceeaşi regulă însă nu se poate aplica şi pentru despărţirea funcţiilor de codare de canal şi modulare. De exemplu, modulaţia bazată pe codare trelli s (TCM), care este o tehnică ce combină codarea de canal şi modulaţia, poate obţine performanţe mai ridicate decât metodele care separă codarea de canal şi modulaţia.

1.3.1.1 Modele de canal

Pentru studierea şi compararea codurilor de canal este folositor să considerăm modulatorul şi demodulatorul ca fiind părţi componente ale canalului. Rezultatul este un canal cu intrări şi ieşiri în timp discret, caracterizat de intrările şi ieşirile posibile şi de probabilităţile de tranziţie care fac legătura dintre intrări şi ieşiri. De asemenea, ieşirea poate să nu depindă numai de intrarea curentă ci şi de cele anterioare. În aceste cazuri vorbim despre canale cu memorie. Acest proiect are ca scop înţelegerea şi simularea codurilor LDPC. Îacest scop va fi folosit canalul cu zgomot aditiv Gaussian alb (AWGN), acesta fiind un tip de canal cu o bună relevanţă practică. Aşa cum este sugerat şi de numele canalului ieşirea Y este modelată prin adăugarea unei variabile aleatoare distribuite Gaussian (G) la intrarea în canal X.

$$Y=X+G \tag{1.1}$$

G este o variabilă aleatoare distribuită Gaussian de medie zero şi varianţă a_2. Pentru un anumit simbol X= x ieşirea din canal Y este o variabilă aleatoare distribuită Gaussian de medie x şi varianţă σ^2 , dată de formula:

$$P_{X|Y}(x \mid y) = \frac{1}{\sqrt{2\pi\sigma}} e^{-\frac{(y-x)^2}{2\sigma^2}} \tag{1.2}$$

unde P reprezintă densitatea de probabilitate a unei funcţii de o variabilă aleatoare.

1.3.1.2 Coduri de canal

Scopul codării de canal este de a face mesajul transmis mai puţin susceptibil la zgomotul introdus de canal. Acest lucru este realizat prin adaugarea de informaţie redundantă structurată sevenţei transmise, deci crescând numărul de biţi transmişi. Există două tipuri primare de coduri folosite pentru a introduce această redundanţă: coduri bloc şi coduri convoluţionale. Un cod bloc transformă un bloc de informaţie de lungime k într-un bloc cod de dimensiune n, unde n>k. Un codor convoluţional este în schimb un codor ce introduce în mod continuu redundanţă unui şir de biţi de informaţie primiţi în mod continuu. Pentru ambele tipuri de coduri, rata de cod R este dată de numărul

de simboluri de informaţie transmise pentru un simbol iniţial. Pentru coduri bloc aceasta este R=k/n . În acest proiect sunt folosite coduri binare deci fiecare simbol va putea lua numai valorile 0 sau 1. Datorită redundanţei adăugate de către codorul de canal, receptorul poate detecta şi corecta erorile introduse de canal.

Codurile de canal care sunt realizate în acest scop se numesc coduri corectoare de erori. Codurile LDPC fac parte din această categorie. Performanţa unui cod corector de erori poate fi masurată, de exemplu, în raportul semnal zgomot(SNR) necesar pentru a obţine comunicaţia cu o anumită eroare de rată maximă. De obicei SNR-ul necesar pentru a obţine o anumită rată de eroare este cu mult mai mic când sunt folosite coduri corectoare decât atunci când informaţia este transmisă fără codare. Totuşi există şi două dezavantaje date de folosirea codurilor corectoare de erori: măreşte numărul de accesări ale canalului, deci lărgimea de bandă, şi conduce la o complexitate crescută a implementării transmiţătoarelor şi receptoarelor. În secţiunea următoare sunt prezentate câteva limite şi restricţii fundamentale care sunt folositoare pentru analizarea performanţelor unui cod în perspectiva folosirii lui în cadrul unui sistem.

1.3.1.3 Caracteristici de performanţă ale codurilor corectoare de erori

Ca parte a teoriei matematice a comunicaţiei, Shannon a definit conceptul de capacitate de canal. Capacitatea unui canal reprezintă o măsura a cantităţii de informaţie ce poat e fi transportată între intrarea X şi ieşirea Y a unui canal. Definiţia capacităţii este legată de definiţia matematică a informaţiei; informaţia medie mutuală între variabilele aleatoare continue X şi Y , în biţi, definite astfel:

$$I(X \mid Y) = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} p_{x,y}(x,y)\log_2 \frac{p_{x,y}(x,y)}{p_x(x)p_y(y)}\,dxdy \tag{1.3}$$

unde $p_{x,y}(x,y)$, $p_x(x)$ şi $p_y(y)$ sunt densitatile de probabilitate pt X si Y.

Capacitatea canalului C este definită ca maximul valorii I(X,Y) pentru distribuţia de intrare $p_x(x)$ măsurată în biţi/canal:

$$C = \max I(X,Y) \tag{1.4}$$

Teorema codării (Shanon) de canal asigură o relaţie între capacitatea canalului C şi rata de transmisie a informaţiei R: Dacă avem o sursă cu un debit de informaţie R biţi/s şi un canal de capacitate C biţi/s şi dacă R < C, există un cod cu cuvinte de lungime n astfel încât probabilitatea unei erori de decodare să fie:

$$P_E \le 2^{-nE(R)} \tag{1.5}$$

unde n este lungimea cuvântului de cod şi E(R) o funcţie nenegativă numită exponentul erorii.

Deci, indiferent de nivelul perturbaţiilor din canal se poate face transmisiunea cu o probabilitate a erorii oricât de mică.

Performanţele codorului LDPC sunt foarte bune tinzând spre limita teoremei lui Shannon. Se defineşte eficienţa spectrală: ES=r_b bit/sec/hz, unde r_b este rata de transmisie şi B banda .

Eficienţa spectrală maximă

$$ES_{\max} = \log_2 MR \tag{1.6}$$

Raportul semnal/zgomot

$$\frac{S}{N} = \log_2 MR \frac{E_b}{N_0} \tag{1.7}$$

unde M este modulaţia, R rata codorului şi $\frac{E_b}{N_0}$ reprezintă raportul dintre energia de bit şi densitatea spectrală de putere a zgomotului. Teorema lui Shannon spune că se poate realiza o probabilitate de eroare de bit oricât de mică prin codare-decodare dacă $r_b < C$, unde C este capacitatea canalului

$$C = B\log_2\left(1+\frac{S}{N}\right) \tag{1.8}$$

Daca r_b=C, rezultă că eficienţa spectrală maximă este:

$$ES_{\max} = \log_2\left(1+ES_{\max}\frac{E_b}{N_0}\right) \Rightarrow \frac{E_b}{N_0} = \frac{2^{ES_{\max}}-1}{ES_{\max}} \tag{1.9}$$

şi reprezintă raportul minim dintre energia de bit şi densitatea spectrală de putere a zgomotului necesar pentru transmisie. Pentru canalul discret AWGN cu intrări şi ieşiri continue capacitatea canalului este dată de relaţia :

$$C = B\log_2\left(1+\frac{\sigma^2 x}{\sigma^2}\right) \text{ biti/canal} \tag{1.10}$$

unde puterea la intrarea canalului este limitată de $E\left[X^2\right] \le \sigma^2 x$ si σ^2 este varianţa zgomotului. În practică forma de undă folosită este limitată de bandă. Deci rata de transmisie nu poate fi crescută oricât prin accesarea oricât de des a canalului. O cerinţă fundamentală pentru comunicaţie este dată de:

$$\frac{E_b}{N_0} > \frac{1}{\log_2 e} = \ln 2 \qquad (1.11)$$

deci nu este posibil să se conceapă un sistem cu o rată arbitrară scăzută de erori dacă E_b/N_0<0.7 dB. Pe de altă parte, atât timp cât condiţia este îndeplinită, teoretic se pot atinge probabilităţi de eroare scăzute folosind coduri suficient de mari. Această restricţie impune o limită inferioară în ceea ce priveşte raportul energie per bit pe densitatea spectrală a zgomotului (E_b/N_0), limită ce trebuie respectată pentru a putea obţine o comunicaţie fără erori. Oricum, presupunerile folosite pentru a obţine aceasta limită sunt prea optimiste pentru cele mai multe situaţii reale. Cea mai importantă diferenţă între descriererea teoretică şi realizarea practică este dată de faptul că banda canalului este, în realitate, limitată. Din această cauză raportul E_b/N_0 necesar pentru o comunicaţie bună, chiar folosind coduri foarte mari, este mai mare decât 0.7 dB.

1.3.2 Codurile bloc liniare

Codurile de control al parităţii de densitate mică fac parte din categoria codurilor bloc liniare. Pentru o mai bună înţelegere vom defini proprietăţile codurilor cu diagrame liniare:

Sursa informaţiei. Împărţim o secvenţă cu sursa primară într-un şir de vectori de lungime k înainte de codificare. Un astfel de vecor se notează cu u.

Codarea se face pe baza lungimii unei diagrame aplicând u pe un şir de vectori cifrat cu x. Un cod (n,k) are lungimea de cod n.

Cod sistematic. Un cod în care informaţia vector apare ca o parte a cuvântului de cod sistematic. Segmentul de informaţie din cuvântul de cod este numit $x_p = u$ şi lungimea n-k biţi de paritate adăugaţi x_p.

Proprietatea de liniaritate. Codul C este un cod liniar binar dacă şi numai dacă C formează un subspaţiu vectorial peste +. Deci, suma oricăror două cuvinte de cod trebuie să fie în sine un cuvânt de cod.

Dimensiunea codului este dimensiunea spaţiului vectorial care îi corespunde. Un cod binar (n,k) are dimensiunea k, şi astfel are un total de 2^k cuvinte de cod, fiecare de lungime n.

Cuvânt de cod diferit de zero. O consecinţă directă a proprietăţii de liniaritate este aceea că numele de cod total=zero , x=0, este un membru al oricărui cod liniar.

Coeficientul codului. Coeficientul codului este r=k/n şi indică proporţia de informaţie transferată pe canal utilizat.

Matricea Generatoare. Proprietatea de liniaritate implică existenţa unei baze pentru cod. Să construim o matrice generatoare pentru cod, denumită G, folosind baza independentă de cuvântul de cod k pe post de şir de vectori G. Matricea generatoare are dimensiunea k x n şi poate fi folosită pentru a codifica vectorul de informaţie. Să considerăm forma sistematică $G_{sys}=[P|I_k]$ în aşa fel încât biţii de paritate ai cuvântului de cod să preceadă biţii de informaţie. Aici $[P|I_k]$ denotă legatura matricei unitare k x k cu P. Aşadar, codificarea este realizată respectând $x=[x_p|x_u]=uG$. Matricea generatoare descrie structura codului, totuşi nu este definită numai pentru cod.

Matricea Controlului – Parităţii. Un alt mod de a descrie structura codului, este furnizat de matricea de control al parităţii, denumită H. În general, H are dimensiunea m x n, unde m=n-k. Toate cuvintele de cod trebuie să respecte condiţia $Hx^T=0$. Matricea de control al parităţii nu este definită numai pentru cod şi este legată de G prin $HG^T=0$. Forma sistematica a lui H poate fi obţinută din G, şi viceversa, folosind $Hsys=[I_m|P^T]$.

Distribuţia greutăţii. Definim vectorul de greutate x, W(x), ca numărul elementelor diferite de zero pe care le conţine. Distribuţia greutăţii unui cod este o listă cu numărul de cuvinte la fiecare greutate, pentru toate greutăţile cuvântului cod.

Distanţa Hamming dintre două cuvinte cod este numărul poziţiilor în care ele diferă.

Distanţa minimă se notează cu d_{min} şi este cea mai mică distanţa Hamming dintre două cuvinte de cod din mulţimea tuturor cuvintelor. Având în vedere că un cuvânt de cod nul (cu toti biţii zero) este un membru al fiecarui cod liniar, distanţa minimă a unui cod liniar este egală cu greutatea cuvântului de cod care are cea mai mică greutate. În general, un cod cu greutatea minimă d_{min} poate corecta $[(d_{min}-1)/2]$ greşeli.

În timp ce şirurile matricei generatoare sunt cuvinte de cod, şirul cu cea mai mică greutate reprezintă o legatură superioară simplă sau o distanţă minimă. Teorema următoare se leagă de structura codului prin matricea de control al parităţii[14].

Teorema (Massey). Distanţa minimă a unui cod cu diagrama liniară binară este egală cu numărul minim de coloane diferite de zero în matricea sa de control al parităţii care însumează zero.

1.3.3 Coduri LDPC

Un cod LDPC este un cod care are o matrice de control al parității slab populată (matrici de control rară – *sparse matrix*). În acest capitol vom prezenta evoluția codurilor LDPC, începând de la codurile prezentate prima dată de Gallager până la posibilitatea recentă de abordare a structurii.

Un cod bloc (n,k) ia k biți la un moment dat și produce n biți. Codurile bloc introduc redundanță, în așa fel încât erorile pot fi corectate. Fie u totalitatea mesajelor de k biți, denotând un mesaj cuvânt , iar c totalitatea codificărilor de n biți, denotând un cuvânt de cod.

$$u=[u_0u_1\ldots u_{k-1}] \qquad (1.12)$$
$$c=[c_0c_1\ldots c_{k-1}]$$

Un parametru important al codurilor bloc este rata codului (r). Este o măsură a cantității de informațe (mesaje) trimise pe cuvântul de cod. Mai multe informații, mai puțină redundanță și mai puține erori pe care codul le poate corectă, invers, cu o cantitate mare de redundanță, transmițătorul petrece mai puțin timp să trimită informațiile.

Cuvintele de cod (*codewords*) sunt codate printr-o matrice generatoare G prin următoarea identitate:

$$c=u*G \qquad (1.13)$$

Unde G este o matrice binară (n,k).

De exemplu un cod (7,4):

$$G = \begin{pmatrix} 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \tag{1.14}$$

Pentru orice matrice generatoare G, există mai multe *matrici de verificare a parității* (H), care satisfac condiția :

$$G * H^T = 0 \tag{1.15}$$

Continuând exemplul arătat în ecuația 1.14, o matrice de verificare a parității este dată de:

$$H = \begin{pmatrix} 1 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 \end{pmatrix} \tag{1.16}$$

Matricea de verificare a parității are următoarea proprietate importantă, c este un cuvânt de cod (codeword) numai dacă satisface :

$$c * H^T = 0 \tag{1.17}$$

Condiția în ecuația 1.17, impune constrângeri liniare între biții c, numită ecuație de verificare a parității. Matricea de verificare a parității 1.16, dă naștere la următoarele ecuații de verificare a parității:

$$c_0 \oplus c_3 \oplus c_5 \oplus c_6 = 0$$
$$c_1 \oplus c_3 \oplus c_2 = 0$$
$$c_2 \oplus c_4 \oplus c_5 = 0$$

unde operația $\oplus$ este o sumă în modulo-2 sau pur si simplu o operație XOR.

Pentru cuvinte bloc scurte poate fi folosită o schemă decodificată, cunoscută sub numele de sindromul decodificării, care folosește H. Această formă de, decodare nu este practică pentru blocuri de o lungime vastă, pentru ele se folosesc alți algoritmi de decodare.

1.3.4 Generarea de matrici de paritate ale codurilor LDPC

În comparație cu alte coduri bloc, codurile LDPC sunt construite folosind o matrice de verificare a parității, nu o matrice generatoare. Codurile sunt concepute folosind matricea în sine ori reprezentarea ei grafică, graficul Tanner.

O matrice de verificare a parității, LDPC, este considerată (w_c; w_r)- regulată dacă fiecare bit de cod este cuprins într-un număr fix de verificare a parității w_c și fiecare ecuație de verificare a parității conține un număr fix, w_r, de biți de cod [5].

Daca restricțiile pentru (w_c, w_r)- regulată sunt relaxate, atunci un cod neregulat poate fi proiectat. Pentru o matrice de verificare a paritatii nergulată, fracțiunea de coloane cu greutata *i* este marcată V_i , iar fracțiunea de linii cu greutatea *i* prin h_i. Impreuna *h* și *i* reprezintă gradul de distribuție al codului. În general codurile neregulate funcționează mult mai bine decât codurile regulate, la valoare complexității de decodare hardware. [6]

1.3.5 Reprezentarea grafica a mecanismului de decodare LDPC

Matricea de verificare a parității poate fi de asemenea reprezentată printr-o formă grafică, numită reprezentarea grafică a lui Tanner. O reprezentare grafică Tanner conține două seturi de noduri: n noduri pentru biții cuvântului de cod și m noduri pentru ecuațiile de verificare a parității. O muchie se alătură unui nod bit la un nod de verificare, dacă acel bit este inclus într-o ecuație corespunzătoare de verificare a parității. Numărul de muchii în reprezentarea grafică a lui Tanner este egal cu numărul celor din matricea de verificarea a parității. Figura 1.4, arată o reprezentare grafică Tanner a următoarei matrice de verificare a parității:

$$H = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \end{pmatrix} \tag{1.18}$$

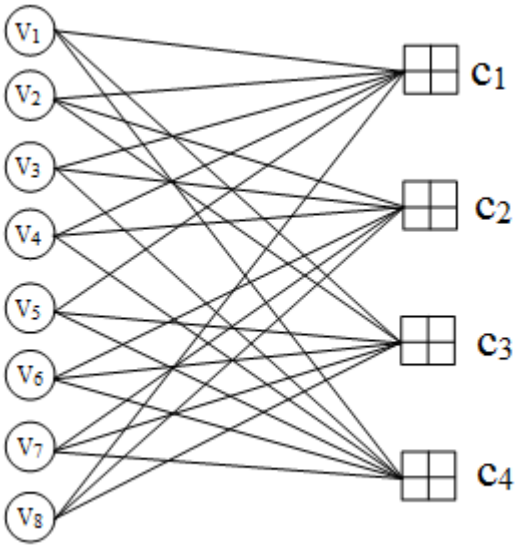


Figura 1.4: Graficul Tanner al unei matrici de verificare a parității

1.3.6 Codarea LDPC

Codurile LDPC reprezintă o clasă a codurilor bloc liniare. Astfel ele pot fi codificate folosind matricea generatoare, însă pe măsura ce dimensiunea blocului de date creşte această metoda impune decodorului stocare mare şi cerinţe legate de procesare. Deşi H este construit ca sa fie slab populat, matricea generatoare a codului, G, este încărcată în general. De fapt,daca G ar fi rar populat atunci nu ne-am fi aşteptat ca acel cod să aibe o distanţă minimă bună. Să ne amintim că şirul cu cea mai mică greutate din G e limitat de d_{min}. Aşadar codoarele pot deveni considerabil mai lente şi mai mari când lungimea blocului n este crescută, având în vedere că înmulţirea matricei-vector are complexitatea $O(n^2)$. Acest lucru a dirijat cercetarea spre găsirea unor codoare eficiente din punct de vedere al calculului şi a codurilor structurate care necesită implementarea unui codificator de complexitate redusă.

1.3.7 Decodarea LDPC– *Message Passing*

Așa cum am spus mai devreme, în general codurile bloc sunt decodate folosind o formă de decodare Maximum Likelihood (ML). Această problemă este clasată ca NP-Complete, ceea ce înseamnă că acolo nu există nicio soluție cunoscută care funcționează în timpul polinomial. Pentru blocurile de o lungime vastă, folosite de codurile LDPC, decodarea ML nu este posibilă, în schimb este folosit *Message Passing* .

Se numește *message passing* un mesaj care trece înainte și înapoi pe muchia graficului lui Tanner. Mesajele sunt informații probabilistice veridice despre bitul emis, exprimat ca *Log Likelihood Ratios* (LLRs). Exprimându-i în acest fel, permite produsului de probabilitate să fie redus la sume pentru reducerea complexității implementării LLR.

Scopul decodificarii sumă-produs este să calculeze valoarea maximă a probabilității a posteriori MAP pentru fiecare bit al cuvântului de cod $P_i = P\{c_i = 1 \mid N\}$, care reprezintă probabilitatea ca al *i*-lea bit al cuvântului de cod să fie optimă condiționată de evenimentul N, și că toate ecuațiile de verificare a parității să fie satisfăcute. Informațiile extra despre bitul *i*, primite de la verificarea parității, este numită informație extrinsecă despre bitul *i*.

Suma tuturor algoritmilor iterativi calculează cu aproximație valoarea MAP pentru fiecare bit de cod. Algoritmul se apropie de probabilitatea MAP doar dacă graficul Tanner este un ciclu liber. Dacă graficul conține cicluri atunci informația extrinsecă devine corelată cu probabilitatea bitului a priori, prevenind probabilitatea b a posteriori să fie exactă. Mesajul extrinsec de la nodul de verificare (check node) *j* până la nodul *i*, $E_{j,i}$, este probabilitatea LLR, aceea că bitul *i* face că ecuația de verificare a parității să fie îndeplinită. Probabilitatea ca ecuația de verificare a parității să fie îndeplinită există în cazul în care bitul *i* este 1.

$$E_{i,j} = 2\tanh^{-1}\left(\prod_{\alpha \in V[j] \alpha \neq i} \tanh\left(\frac{M_{\alpha j}}{2}\right)\right) \qquad [5] \qquad (1.19)$$

unde $M_{\alpha j}$ este LLR curent estimat disponibil să verifice ecuația *j*, a probabilității ca bitul α este 1. Vj este setul de noduri bit conectate la nodul de verificare j. Reține că mesajele trimise de la nodurile de verificare către nodurile de bit nu include informația pe care nodul bit deja o are.

Mesajul care este trimis de la nodul de bit către nodurile de verificare este dat de

$$Q_{i,j} = \lambda_j + \sum_{\alpha \in C[j] \alpha \neq i} R_{\alpha j} \qquad [5] \qquad (1.20)$$

unde C_j reprezintă setul de noduri de verificare conectate la nodul bit *j* și λ_i este LLR al probabilității a priori a mesajului. Încă o dată mesajul trimis de la nodul bit la nodurile de verificare nu include și informația pe care nodul de verificare deja o are.

Suma rezultată a algoritmului conține următorii pași cu LLR impus pentru probabilitatea a priori a mesajului, matricea de verificarea a parității H și numărul maxim de iterații permise I_{max}:

- Inițializarea
  Setul $Q_{ij} = \lambda_j$ inițializează nodurile de verificare cu probabilitatea a priori a mesajului
- Actualizarea mesajului de verificare
  Pentru fiecare nod de verificare (check node) *j* și pentru fiecare nod de bit (node bit) asociat calculului *j* :

$$R_{i,j} = 2 \tanh^{-1} \prod_{\alpha \in V[j] \alpha \neq i} \tanh\left(\frac{Q_{\alpha j}}{2}\right) \qquad (1.21)$$

- Test pentru validarea cuvantului de cod
  Face o decizie experimentală pe cuvântul de cod

$$L_i = \lambda_j + \sum_{j \in C[j]} Q_{ij} \qquad (1.22)$$

$$z_i = \begin{cases} 1, L_i \leq 0, \\ 0, L_i > 0 \end{cases}$$

  Dacă numarul de iterații este I_{max} sau un cuvant de cod a fost gasit $(H_z^T = 0)$, atunci este gata.
- Actualizarea mesajelor de bit pentru fiecare nod de bit *j*, și pentru fiecare nod de verificare asociat calculului *j*:

$$Q_{i,j} = \lambda_j + \sum_{\alpha \in C[j] \alpha \neq i} R_{\alpha j}[k-1] \qquad (1.23)$$

  Se reia pentru actualizarea mesajelor de verificare (Update Check Messages).

Capitolul 2

2 Implementari privind decodoarele LDPC

Obiectivul principal al acestei lucrări a fost de a cerceta literatura de specialitate pentru a afla cum sunt implementate în hardware decodoarele LDPC. După o căutare amănunțită au fost găsite multe documente, care detaliază implementările hardware. Au fost categorizate după configurația în care au fost folosite și apoi clasificate într-o bază importantă pe o prezentare grafică a unui decodor flexibil, devenind capabile să fie implementate pe o platformă FPGA. O scurtă discuție despre configurațiile găsite, inclusiv diagramele lor, viteza, flexibilitatea și potrivirea pentru FPGA, urmează.

2.1 Implementarea unui decodor LDPC serial

Un decodor serial este cel mai simplu decodor în termeni evaluativi hardaware. Este format dintr-un singur nod de verificare, un singur nod variabilă și memorie. Nodurile variabile sunt actualizate pe rând, apoi nodurile de verificare sunt actualizate în aceeași formă serială. Acesta este tipul de implementare care ia naștere din codul scris care doar implementează suma rezultată a algoritmului, pe un calculator sau pe FPGA. Avantajul primordial, a unei implementații în serie, este flexibilitatea foarte mare, suportând blocuri de mărimi diferite și rate de cod, cu doar o singură matrice de verificare introdusă în memorie. Din păcate această abordare este prea lentă pentru orice altceva, exceptând simulările. [7] . Numărul de cicluri de ceas, cerute pentru fiecare iterație a decodorului de serie, este aproximativ de două ori mai mare decât muchile. Figura 2.1 prezintă configurație pentru un decodor de serie.

Exemplu

Codul propus pentru wireless LAN, IEEE 802.11n este codul (1944; 792) cu 6803 muchii.

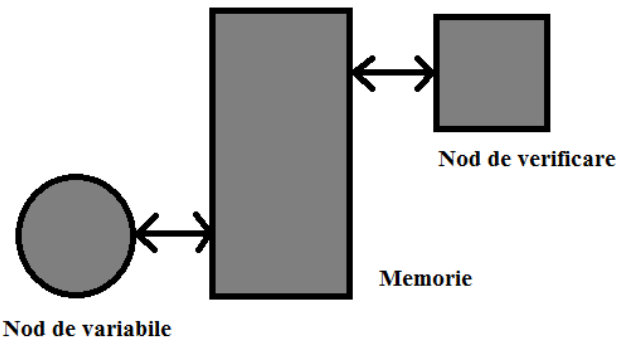


Figura 2.1: Configurația unui decodor serial

Ar fi nevoie de aproximativ 2x6803 sau 13606 cicluri de ceas ca o singură iterație a decodorului să fie completă. Faptul ca nimic nu a putut fi gasit, in literatura de specialitate, despre

implementarea unui decodor serial LDPC arată că nu este posibil pentru orice aplicație, unde, poate, ai vrea să folosești coduri LDPC.

2.2 Implementare analogică a unui decodor LDPC

În timp ce majoritatea documentelor studiate foloseau semnale digitale pentru a implementa configurația decodorului, o serie de lucrări foloseau metode analogice. Din moment ce este imposibil să implementezi un decodor analogic pe un FPGA (dispozitivele FPGA pot procesa doar semnale digitale), cu toate acestea este interesant să observi aplicată o metodă analogică pe o problemă pur digitală, în mod special când performează comparabil cu copiile lor digitale.

Au fost găsite două abordări ale implementării algoritmului sumă-produs, una folosea încărcarea non-liniară a Tranzistorului cu Efect de Câmp (FET) [8,9], iar cealaltă folosea combinații între rezistori și condensatori, creând ecuații diferențiale RC [10], prezentate în figura 2.2.

2.3 Implementare decodor LDPC paralel

Din Figura 1.14 și din suma rezultată a algoritmului, prezentată în secțiunea 1.3.7, reiese existența unui paralelism într-un decodor LDPC, care poate fi exploatat. Un decodor complet paralel într-un ciclu ceas actualizează faza mesajului de verificare, iar apoi completează actualizarea fazei

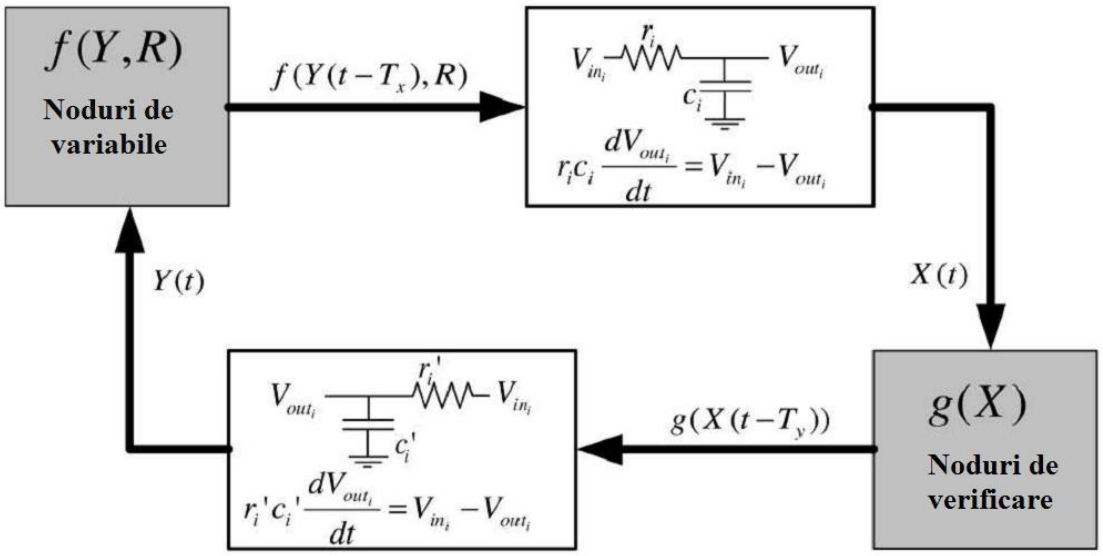


Figura 2.2: Implementarea analogică [10]

mesajului bit. Acesta permite o creștere apreciabilă în viteză decodării, comparativ cu decodorul serial. Lucrări de specialitate care aruncă o privire la configurațiile paralele sunt [11,12,13,14] .

Exemplu

Continuând exemplul din cazul decodorului serial, ar fi nevoie de două cicluri ceas pentru a completa iterația unui decodor.

Figura 2.3 arată configurația unui decodor paralel propus de Blanskby și Howland.

Oricum, această viteză provine atât din costul complexității implementării cât și din cel al flexibilității. Un decodor paralel implementează fiecare nod de variabilă și verificare o singură dată în hardaware și sunt trimise împreună, exact cum sunt explicate de graficul de cod al lui Tanner [11]. În timp ce nodul de verificarea și nodul de variabilă sunt simple, interconexiunea contextului nu este. Numărul de conductori este dat de :

fire de masaj = numarul de muchii ale graficului x numarul de biti/mesaj x 2

Factorul 2 provine de la Blanksby și Howland care au decis să folosească 2 seturi de mesaje conductoare unidirecționale ca opoziție la un singur set de conductoare bidirecționale, pentru a reduce cererile logice și să elimine nevoia de zona tampon a memoriei interne bidirecționale. Implementarea paralelă care a fost investigată [11,12,13], folosește 4 biți pe mesaj, unul pentru semn și 3 biți pentru magnitudine, în timp ce configurația lui Nagarajan folosește și 4 sau 5 biți pe mesaj [14].

În contrast a fost descoperit că pentru o performanță bună a decodorului este nevoie între 4 și 8 biți pe mesaj [15,16,17], ceea ce sugerează că în configurația decodoarelor paralele au fost sacrificate erori de performanță pentru a da rezultate. Cu toate acestea, configurația paralel totală poate fi potrivită pentru decodoare cu rata mare de biți care operează comparativ la un nivel înalt SNR.

Numărul de fire de mesaj necesare de un decodor paralel exclude FPGA-ul ca dispozitiv țintă, în schimb este nevoie de personalizare.

Pentru a ușura dificultatea în călăuzirea semnalului conductor, a fost propusă o configurație tridimensională cu 7 nivele [11], o abordare tridimensională pe 3 nivele a fost deasemenea propusă [13].

Niciuna dintre implementările de decodoare paralele studiate [11,12,13,14] nu au flexibilitate în comparație cu o configurație de cod. Traseul dintre nodurile de verificare și cele de bit trebuie să fie reconfigurat pentru un nou cod realizabil, în timp ce un cod de coeficent diferit sau un exponent de distribuție n-ar putea fi posibile fără o reconfiguratie completă. Aceasta se adaugă la faptul că decodoarele paralele, menționate mai sus, sunt toate configurate personalizat, care exclude orice perspectivă de reprogramare.

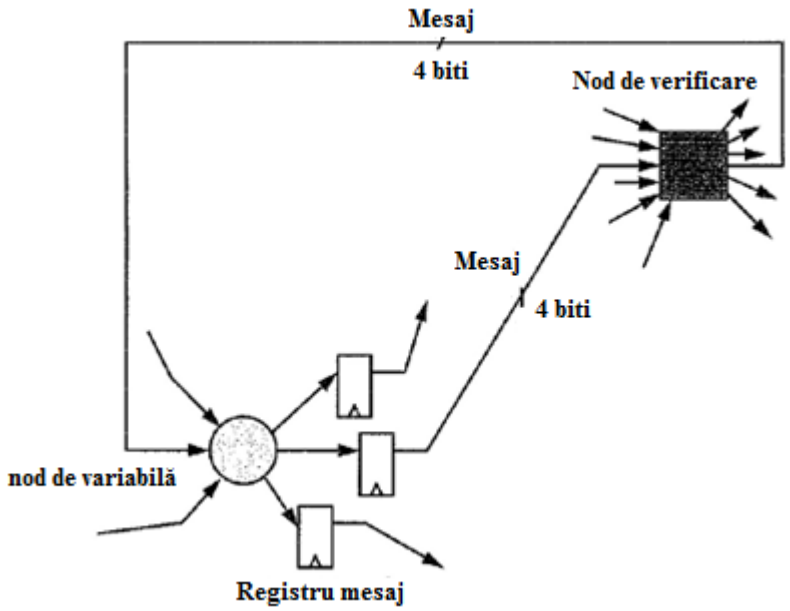


Figura 2.3: Configuratia unui decodor paralel [11]

2.4 Implementare decodor LDPC partial paralel

Între extremele dintre o implementare în serie și una paralelă stă implementarea parțial paralelă. În configurație parțial paralelă câteva noduri de verificare și de biți sunt realizate în hardware, asta implică împărțirea ca într-un caz în serie. Necesită de asemenea folosirea memoriei pentru a stoca mesajele între ciclurile de ceas. Oferă în același timp beneficile paralelismului, găsit într-un decodor paralel, deoarece nodurile de biți și de verificare pot rula și în paralel.

Este un compromis între viteză și complexitate, înmulținduse numărul rodurilor de biți și de verificare realizate în hardware crește și viteză pentru fiecare iterație a decodorului, dar se amplifică și complexitatea.Configuratile în serie și paralele pot fi considerate clase derivate ale decodorului parțial paralel (cel în serie ar avea doar un singur nod bit și de verificare, iar cel paralel ar avea toate nodurile bit și de verificare realizate în hardware).

Marea problemă cu această configurație sunt coliziunile de memorie. Dacă două noduri încearcă să acceseze aceeași memorie în timpul aceluiași ciclu de ceas are loc o coliziune, și unul dintre noduri trebuie să se oprească pentru un ciclu, ceea ce diminuează produsul decodorului. Frecvența acestor coliziuni poate fi redusă prin două metode folosite de implementarea lui Masersa, o altă diagrama de nivel înalt poate fi văzută în figura 2.4. Interconexiunea a fost implementată folosind o formă de comutator crossbar, cunoscut sub numele de rețeaua lui Benes; acesta permite o flexibilitate completă conectând nodurile împreună. Pentru fiecare cod o configurație de direcție, este generată offline folosind un algoritm descris în [1] și în această expunere, în capitolul 4. Acest algoritm configurează rețeaua lui Benes pentru fiecare ciclu de ceas al decodorului, implementând o iterație.

Un alt mod de a reduce coloziunea de memorie este de a modifica algoritmul de convingere al propagării. Bitul de verificare și bitul de verificare a mesajelor ale unui nod individual sunt foarte similare, singura diferență fiind mesajul pe care îl propagă de-a lungul unei muchii, care ignoră efectul mesajului care provine din aceea muchie. Aceste mesaje pot fi modifcate din unicast în broadcast dacă

nodul mesajului este trimis să păstreze o copie a mesajului trimis în jumătatea de iterație anterioară. Acest lucru reduce semnificativ numărul de mesaje care trebuie să fie trimise între noduri, cu singura condiție ca nodurile să aibă memorie și să păstreze mesajele anterior trimise.

Lee și Ryu au folosit deja rețeaua lui Benes în decodorul lor LDPC parțial paralel [16]. Configurația lor fiind foarte complexă și profund conductoare, este potrivită doar pentru implementarea în Aplicație Specifică a Circuitelor Integrate (ASIC). Schema mea este în mare bazată pe a cea a lui Masera, folosind programul descris în [1]. Diferența mea esențială , este bazată pe mesajele unicast, în timp ce a lui Masera se bazează pe multicast. Pentru configurația mea, aceasta face intercalarea mult mai complexă, la fel și procesoarele nodului de verificare și de bit, atâta timp cât programul este simplu.

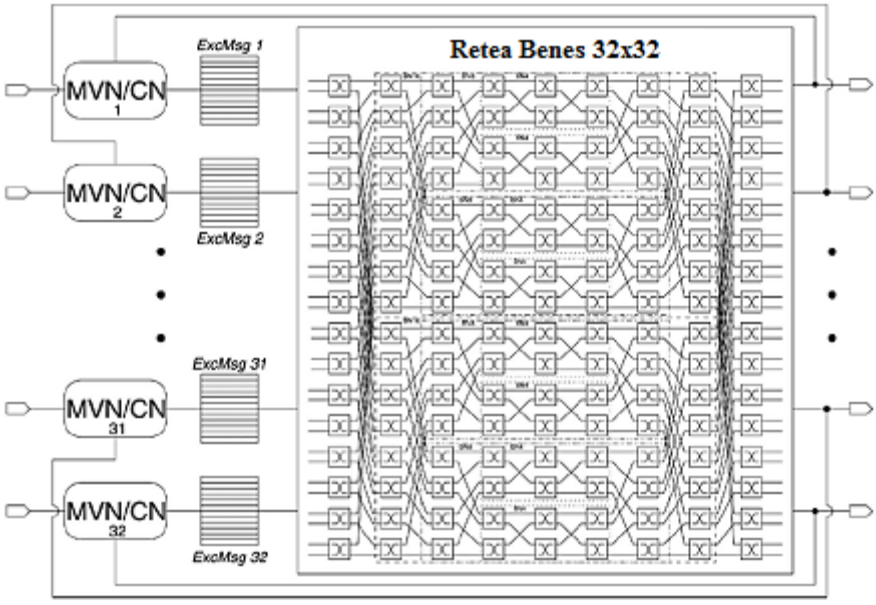


Figura 2.4: Configuratia unui decodor partial paralel [7]

2.5 Specificatii pentru proiectarea decodorului LDPC

Un număr de lucrări de specialtitate detaliază metode pentru a simplifica procesul de decodare a codurilor LDPC, după o configurație specifică a unei matrice de verificare a parității. Impunând restricții la tipurile de coduri suportate, decodorul poate fi simplificat exploatând proprietățile codului, așa cum matricea de verificare a parității regulată reduce stocarea cerințelor necesare pe decodor [17].

Hardware-ul poate fi, de asemenea, proiectat pentru a elimina coliziunile găsite în arhitecturi parțial paralele și atingă capacități ridicate [18].

Scopul țintă al proiectul este acela de a implementa un decodor LDPC flexibil; conceperea unui decodor bazat pe tehnici specifice de proiectare și restricționarea tipurilor de coduri acceptate la un subset redus foarte mult care nu ar îndeplini specificațiile proiectului.

Capitolul 3

3 Implementarea unui decodor LDPC flexibil

Investigând diferite tipuri de implementări ale decodoarelor LDPC în literatura de specialitate, o decizie a fost luată asupra construcției configurației. O configurație parțial paralelă a fost aleasă, pentru că are avantajul unei configurații în serie, fiind flexibilă, cât timp are câteva avantaje de viteză oferite de o configurație paralelă. Poate fi făcut un compromis între viteza decodării și complexitatea implementării prin diversificarea numărului de noduri de procesare paralelă în proiectare.

Scopul principal al conceptului îl reprezintă flexibilitatea, decodorul ar trebui să fie capabil să implementeze orice cod dat (până la constrângerile maxime de proiectare), fără să-și schimbe configurația.

3.1 Prezentare generala

Structura este compusă dintr-un număr P de procesoare, un bloc de permutare a mesajului și un bloc de control logic, exact cum se vede în figura 3.1.

Prezintă un număr mai mic de procesoare bit (verificare) decât noduri bit (verificare) din graficul lui Tanner, ceea ce înseamnă că fiecare procesor bit (verificare) este atribuit unui subset al acestor noduri. Procesoarele însăși sunt responsabile pentru stocarea fiecărui mesaj sosit, efectuând operațiunile nodurilor și transmiterea mesajelor de ieșire, în timp ce atribuirea nodurilor la procesoare este tratată de către unitatea de control.

Procesul de decodare urmărește patru părți distincte văzute în figura 3.2. Bitul de verificare și bitul verificat de jumătate de iterație se repetă de un număr prestabilit de ori înainte de ieșirea cuvântului de cod decodat. Numărul de iterații este mai mic, aproape 10 ca să păstreze timpul de decodare mic. Sunt efectuate verificări pentru un cuvânt de cod valid.

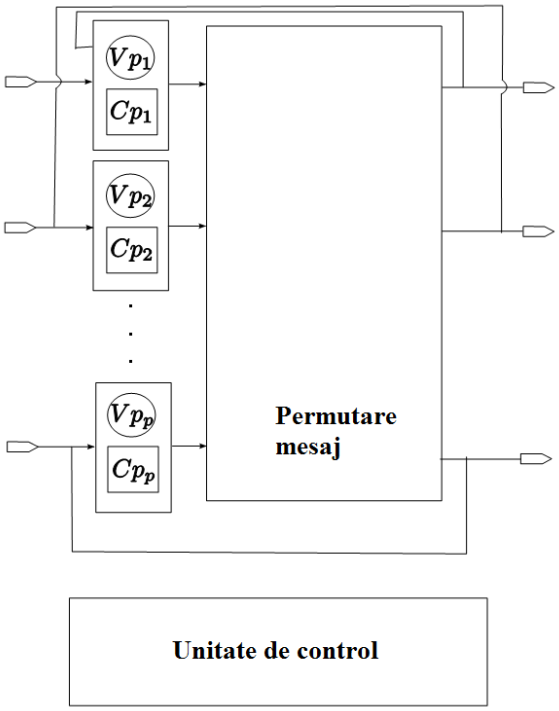


Figura 3.1: Schema bloc pentru decodor

- Inițializare

În timpul inițializării măsurătorile de canal sunt încărcate în blocurile procesoarelor bit.

- Bit de verificare

În timpul bitului de verificare a jumătății de iterație, procesorul nodului bit efectuează funcția nodului bit, cum este descrisă în ecuația 1.10. În prima iterație nu sunt mesaje primite și procesorul emite măsurarea canalului.

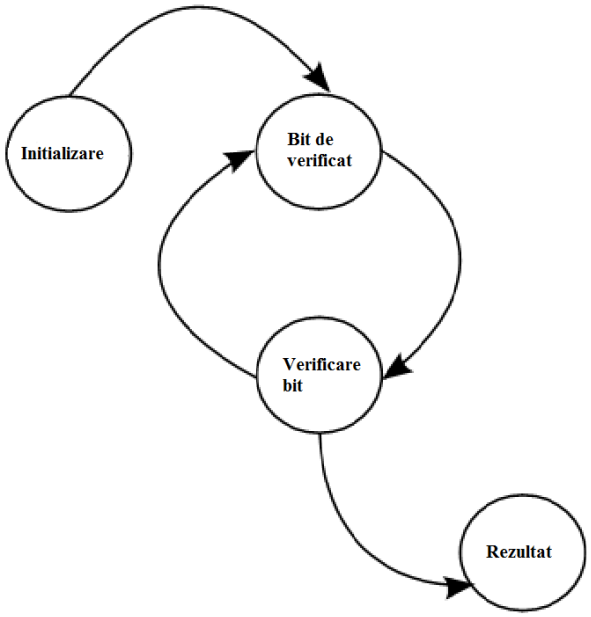


Figura 3.2:Diagrama de stari pentru decodor

Mesajul circula de la procesorul nodului bit la blocul de permutare a mesajului și către procesorul nodului de verificare în ordine.

- Verifcare bit

În timpul bitului de verificare de jumătate de iterație, procesorul nodului de verificare efectuează funcția nodului bit, cum este descrisă în ecuația 1.8. Mesajul circula de la procesorul nodului bit la blocul permutării mesajului și către procesorul nodului de verificare în ordine.

- Rezultat

După un număr de iterații bit de verificat și cicluri de verificare bit, decodorul se mută spre faza de ieșire.

3.2 Blocul de permutare a mesajelor

Blocul de permutare a mesajelor stă în centrul flexibilității decodorului. Scopul său este să conecteze procesorul nodului bit la la procesorul nodului de verificare (și vice versa), în așa fel încât procesorul nodului bit să primească mesajele primite, în ordinea graficului lui Tanner.

Blocul de permutare a mesajelor conține două comutatoare crossbar Benes cu o distribuție intercalată între ele, precum se vede în figura 3.3. Mesajele primite sunt în primul rând permutate în spațiu de partea stângă a rețelei lui Benes și apoi stocate în bancul de distribuție intercalată. Apoi distribuția intercalată permutează mesajele trimise în timp, înainte să fie din nou permutate în spațiu de rețeaua dreaptă a lui Benes. Aceste permutări de timp și spațiu oferă flexibilitate completă, îngăduind mesajelor primite să sosească la ieșirea corectă în ordinea cerută. Configurația pentru rețeaua lui Benes și pentru distribuția intercalată este variată pentru fiecare ciclu de ceas conform programului configurației stocată în unitatea de control. Algoritmul pentru a determina programul este descris în detaliu în capitolul 4.

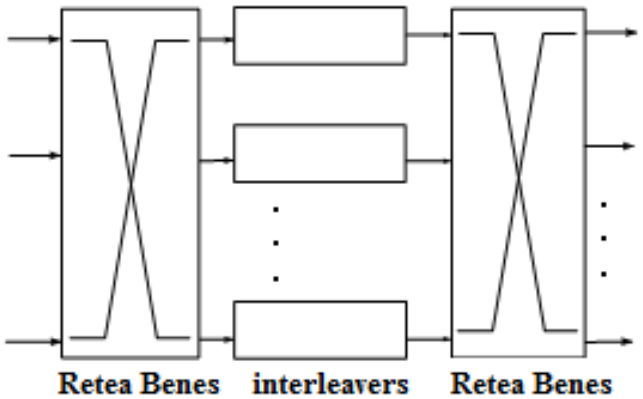


Figura 3.3: Bloc de permutare a mesajelor

3.3 Configurarea setărilor rețelei de comutare Benes

O rețea Benes este o rețea de comutare pe mai multe etape, formalizată de Benes [19], pentru circuite comutatoare de telefon. Elimina nevoia de un comutator crossbar de mare capacitate, înlocuindu-l cu un număr simplu de 2 intrări, 2 ieșiri, conectând elementele precum în figura 3.4. *N(unde N= 2^r*) o rețea de intrare poate fi construită recursiv folosind celule elementare și rețele de intrare 2 $\frac{N}{2}$, așa cum se observă în figura 3.5. I_i sunt comutatoare elementare conectate la intrări, fiecare comutator se conectează la partea superioară (P_A) și cea inferioară (P_B) $\frac{N}{2}$ a rețelei Benes. O_i sunt comutatoare elementare conectate la ieșirile rețelei Benes, fiecare comutator se conectează la partea superioară (P_A) și cea inferioară (P_B) a sub-blocului de permutat. Acest proces de dezintegrare se repetă până când rămâne doar o celulă elementară.

Pentru N intrari din rețeaua Benes o să fie $N \log_2 N - 1$ etape, și un $\frac{N}{2}$ comutatoare pe etapă. Waksman a dovedit că unul dintre comutatoare, în fiecare nivel, este redundant și pot fi setat arbitrar [20]. Comutatorul din dreapta sus va fi folosit și o să fie (arbitrar) în stare de resetare, așa cum este în figura 3.5. Numărul de comutatoare cerute o să fie $N \log_2 N - N + 1$.

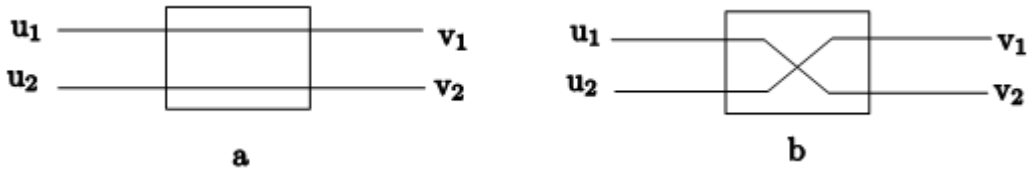


Figura 3.4: Elemente de comutație elementare (a) Resetate (b) Setate

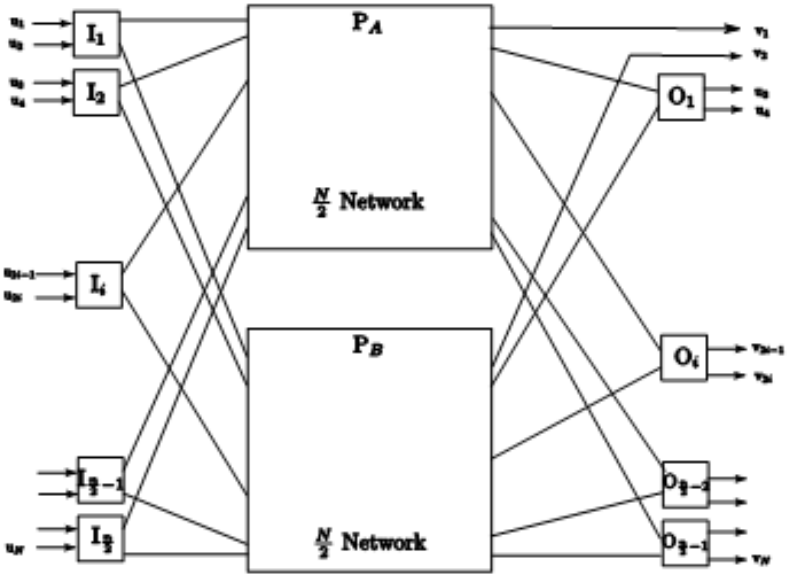


Figura 3.5: N intrări din rețeaua Benes

3.3.1 Prezentare generală a setărilor rețelei de comutare Benes

Procesul de a găsi setările de comutare, date de setările de intrare și ieșire, este împărțit în 3 grupuri de acțiuni și este bazat pe munca lui Waksman [20].

- Obținerea matricei de permutare

O matrice de permutare este obținută începând cu setarea comutatorului din dreapta sus, acesta detaliază care intrări și ieșiri sunt conectate sus $\frac{N}{2}$ la permutarea sub-bloc (P_A) și care sunt conectate jos $\frac{N}{2}$ la permutarea sub-bloc (P_B).

- Găsirea setărilor comutatorului

Matricea de permutare este folosită pentru a găsi setările comutatoarelor elementare.

- Găsirea aplicației de permutare pentru sub-blocurile inferioare și superioare

Știind care intrări și ieșiri sunt conectate la permutările inferioare și superioare sub-bloc, procesul poate fi descompus recursiv. Începem procesul, din nou, pe blocul superior și apoi pe cel inferior. Acest proces se repetă până când ajungem la comutatoarele elementare sub-bloc.

3.3.2 Exemplu pentru configurarea setărilor rețelei de comutare Benes

O explicație a algoritmului pentru stabilirea comutatoarelor într-o rețea Benes, este cel mai bine redată printr-un exemplu.

Luați în considerare o rețea de 8 intrări cerute de permutarea π = {5; 4; 7; 1; 2; 8; 6; 3}. Cu referire la figura 3.5, asta înseamnă că o intrare, u_1 este conectată la a 5 ieșire v_5, i_2 este conectată la v_4 și tot asa.

Ca să găsim matricea de permutare, observăm că v_1 este conectat la P_A , și din permutatia dată v_1 este conectat la u_4. Așa că amplasam un „A” în coloana 1, rândul 4. De asemenea stim că v_2 este conectat la u_5 si ambele sunt conectate la P_B (vezi figura 3.5), asa că plasam un „B” în randul 5, coloana 2. Pe partea de input u_4 este conectat la P_A așa și u_3 trebuie să fie conectat la P_B asa că plasam un „B” în coloana 7, randul 3 (deci u_3 este v_7). Pe partea de output știm ca v_7 este conectat la P_B, deci v_8 trebuie să fie conectat la P_A, așa că plasam un „A” în coloana 8, randul 6. Stiind că u_6 este conectat la P_A, u_5 trebuie să fie conectat la P_B.

Acum încercăm să plasăm un „B” în coloana 2, randul 5, dar el este deja acolo. Sunt 4 intrari i_1, i_2, i_7 si i_8, de asemenea 4 iesiri v_3, v_4, v_5 si v_6 care sunt nerepartizate la P_A si P_B. Asta înseamnă că putem alege arbitrat unde să trimitem una dintre intrări/ieșiri, de exemplu i_1 la P_A. Pentru a găsi

comutatoarele pe parte de intrare trebuie să ne uităm pe coloana matricei de permutare. Așa că setăm rândul 1, coloana 5 la „A” și repetăm acest proces până când avem matricea de permutare, ca în tabelul 3.1.

Acum putem trece la al doilea pas, și anume găsirea setărilor de comutare pentru comutatoarele elementare. Pentru a găsi comutatoarele pe partea de intrare trebuie să ne uităm la coloana matricei de permutare. I_1 este dat de rândurile 1 si 2, I_2 este dat de rândurile 3 si 4 și tot așa. Pentru a-l găsi pe I_1, observăm ca „A” se află într-o coloană neobișnuită (1), așa că este în poziția de resetare. Pentru I_2 , „A” se află în coloană exactă, așa că este în poziția setată. Similar, I_3 si I_4, sunt aceeași poziție.

Pentru comutatoarele din partea de ieșire ne uităm la coloanele matricei de permutare. O_1 este dat de coloanele 3 și 4, O_2 de coloanele 4 și 5, iar O_3 este dat de coloanele 6 și 7

u\v	1	2	3	4	5	6	7	8
1					A			
2				B				
3							B	
4	A							
5		B						
6								A
7						B		
8			A					

Tabelul 3.1: Matricea de permutare

(de coloanele 1 și 2 nu este nevoie, deoarece nu există niciun comutator). „A” apare în coloanele 3,5 și 8 care, în același mod ca și intrările, dă O_1 și O_2 resetate, iar O_3 fiind setat. Configurația rețelei se observă în figura 3.6.

Al treilea și ultimul pas este acela de a găsi aplicația de permutare pentru sub-blocurile inferioare și superioare. Se începe prin gruparea matricelor de permutare în 2x2 pătrate. Aplicația de permutare pentru P_A este dată de poziția lui „A” în aceste pătrate, în timp ce P_B este dat de locația lui „B”, precum în tabelul 3.2. Acesta oferă $\pi_{PA=}$ {3,1,4,2} și $\pi_{PB=}$ {2,4,2,1} . Acum repetăm pașii pentru a afla setările comutatorului pentru permutarea sub-blocurilor inferioare și superioare.

u\v	1	2	3	4
1			1	
2				
3	1			1
4		1		

u\v	1	2	3	4
1		1		
2				1
3	1			
4			1	

Tabelul 3.2: Mapari de permutare pentru P_A si P_B

Procesul de dezintegrare recursivă se încheie când ajungem ca un sub-bloc să conțină o singură celulă elementară (2x2 blocuri). Dacă permutarea este π= {1,2}, atunci comutatorul este în poziția de resetare, altfel este setat. Figura 3.6 ne arată setările comutatorului finalizate. Rețineți că, dacă decizia de a seta i_1 a fost făcută diferit, o să avem un set diferit de setări ale comutatorului, dar vor avea în continuare configurația validă.

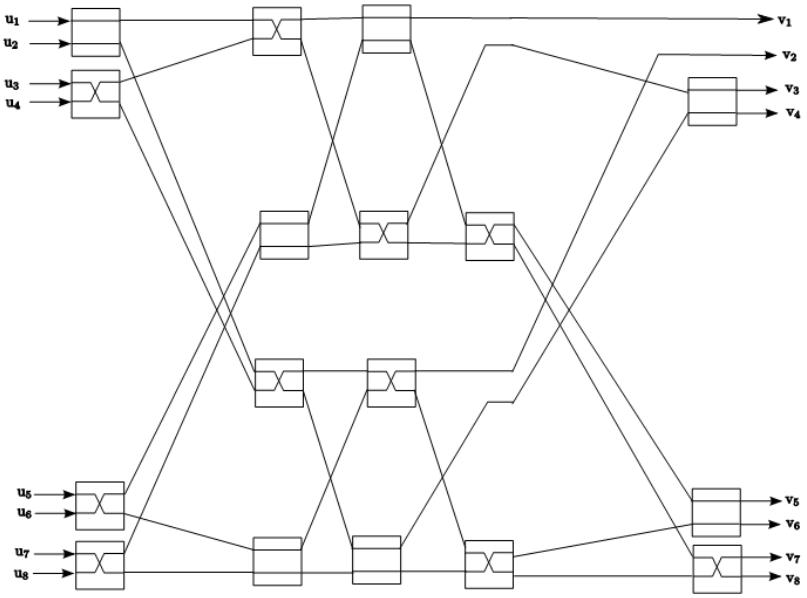


Figura 3.6 N intrari din rețeaua Benes

3.4 Unitate de control a diagramei de stari

Unitatea de control este responsabilă pentru implementarea diagramei de stari exact ca în figura 3.2, care implică setarea linilor de control relevante pentru procesoarele de bit și de verificare, precum și pentru permutarea mesajului bloc. Configurația pentru cod este păstrată într-un bloc ROM în interiorul unității de control, care conține setările de timp diferite pentru rețelele Benes și bancurile intercalate din unitatea permutări mesajului, precum și asignarea nodurilor bit și de verificare pentru procesoarele respective. Deși această implementare folosește ROM pentru fișierul de configurație, un alt sistem nevolatil de stocare cum ar fi memoria flash, poate fi folosit.

3.5 Flexibilitatea decodorului LDPC

Structura acestui decodor, prin utilizarea blocului de permutare a mesajului, este flexibil și permite oricărui cod particular să fie folosit. Pentru a folosi un nou cod, configurația este generată offline și apoi descărcată în blocul ROM, în interiorul blocului de control, fără să schimbe configurația decodorului. Blocul ROM conține configurația variată de timp a rețelelor Benes, bancurile intercalate și asignarea nodurilor bit și de verificare pentru procesoare. Structura suporta și coduri neregulate, cu anumite constrângeri.

Capitolul 4

4 Planificarea construcţiei decodorului LDPC

Codurile LDPC au un grad înalt de paralelism, fiecare nod poate lucra în paralel cu alte noduri de același fel. Cu un grad înalt de paralelism are loc o decodare mai rapidă, cu costul unei complexități mai mari.

Într-o configurație parțial paralelă, există procesoare nod de verificare P și de biți P, fiecare lucrând în paralel. Principala problemă cu această structură este modul în care este accesată memoria. Consider iterația jumătate, atunci când nodurile bit sunt active. Pentru a fi sigur că fiecare procesor nod de verificare primește mesajul corect la fiecare ciclu, mesajul expediat de la nodurile bit trebuie să fie permutat în același timp (intercalate) și spațiu (comutatoare crossbar).

Dacă sunt P procesoare atunci va trebui să fie P acesari simultane la memorie pentru a citi intrarea și a scrie ieșirea. Așadar trebuie să fie P bancuri de memorie în acest fel memoria poate fi accesată fără coliziuni. O imagine a acestuia se observă în figura 4.1. O planificare corectă asigură faptul că nu există coliziuni în memoria de acces pentru scris și citit. Dacă sunt L muchii în total, ansamblul de procesoare P va trebui să fie accesat de [L/P] ori pentru a finaliza o jumătate de iterație. Fiecare accesare cere o planificare care să definească care muchie variabilă este necesară pentru fiecare procesor și în ce ordine aceste muchii variabile sunt citite.

Dificultatea cu codurile LDPC este aceea că planificarea pentru scrierea și citirea în memorie și nodul variabil al jumătății de iterație este diferit de planificarea pentru nodul de verificare al jumătății de iterație. Asta înseamnă că apare un nou set diferit de constrângeri de planificare pentru cele două jumătăți de iterații. Planificarea definește ce variabile sunt citite simultan, și în același timp decide ce variabile trebuiesc plasate în diferite bancuri de memorie.

Planificarea oferă timpul diferitelor configurații pentru bancul crossbar și intercalat, ca în figura 4.3.

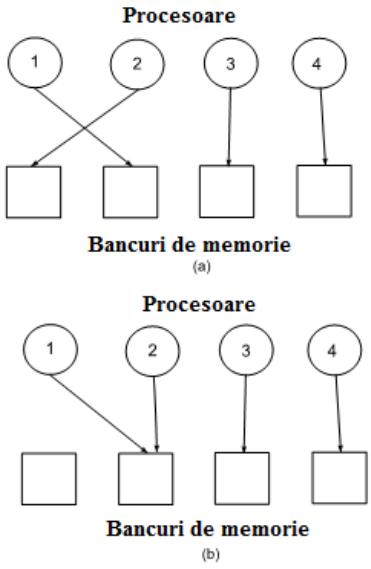


Figura 4.1: a) Fara coliziuni. b) Au loc coliziuni.

4.1 Algoritmul de mapare a graficului Tanner

4.1.1 Prezentarea algoritmului

Primul pas este acela de a decide ce noduri bit va opera în paralel. Matricea de verificare a parității H este reprezentată ca în graficul lui Tanner, expus în figura 4.2. Acesta permite muchilor să fie numărate. Graficul lui Tanner este împărțit, plasând nodurile biți în seturi P, cu un număr egal de muchii în fiecare set, iar nodurile de verificare într-un set P, din nou cu un număr egal de muchii în fiecare set. Seturile reprezintă grupul de noduri bit care sunt asignate fiecărui procesor bit. Un nod de la fiecare set va funcționa în paralel. Figura 4.5 prezintă un grafic Tanner împărțit.

Opțiunea de a împărți va afecta performanța decodorului, astfel încât un pachet de partiționare grafic, METIS [21], este folosit pentru a minimiza numărul de intersecții între partiții. Rezultate experimentale au arătat că graficul partiționat poate diminua ciclul de timp al jumătăți de iterație, în medie, cu 20% [7], reducând numărul potențialelor coliziuni de memorie.

Al doilea punct este la variabilele muchiilor grafice și la bancurile de memorie. O funcție de mapare este apoi aplicată, maparea muchiilor din graficul lui Tanner la bancurile de memorie. Scopul funcției de mapare este acela de a evita coliziunile de memorie, pentru că atunci când fiecare dintre nodurile bit, acționează toate în paralel, accesează memoria în același timp.

Ultimul și al treilea pas este acela de a utiliza funcția de mapare pentru a determina configurația cerută pentru comutatoarele crossbar și pentru bancurile intercalate. Acești pași sunt conturați, mai în detaliu, în cele ce urmează.

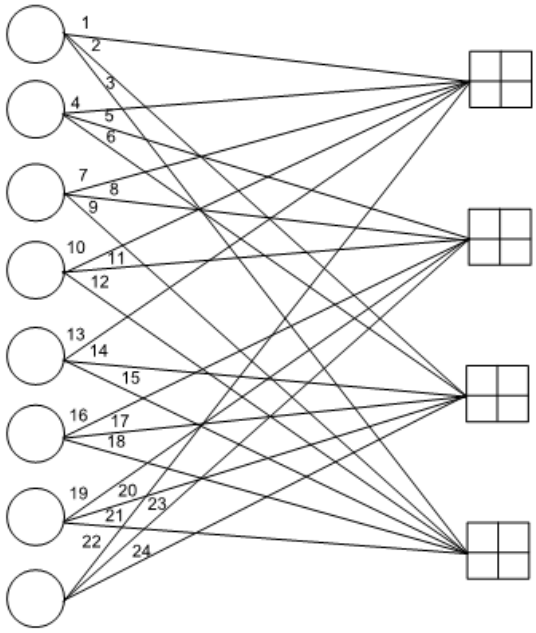


Figura 4.2: Graficul lui Tanner numerotat

4.1.2 Notații folosite pentru maparea graficului Tanner

De-a lungul restului capitolului, următoarele simboluri sunt folosite:
H, denotă matricea de verificare a parității codului
L, denotă numărul de muchii din cod
P, denotă numărul de procesoare care operează în paralel
$M \triangleq \left[\frac{L}{P}\right]$, denotă numărul de operații paralele pentru a completa jumătate de ciclu
P_1, maparea setului de noduri de biți pentru procesoare
P_2, maparea setului de noduri de verificare pentru procesoare
V_i, un subset P_1 care conține muchii care sunt procesate în paralel
V'_i, un subset P_2 care conține muchii care sunt procesate în paralel
M, funcția de mapare a muchilor la bancurile de memorie
Vp_j, procesorul nodului bit j
Cp_j, procesorul nodului de verificare j
β_j (i), configurația de timp care variază de la comutatorul crossbar cel mai stâng
β'_j(i), configurația de timp care variază de la comutatorul crossbar cel mai drept
δ_i(j), configurația de timp care variază de la bancul intercalat i

4.1.3 Procedura formală a funcției de mapare

Procedura următoare este bazată pe munca lui Tarable[1].Se consideră un set de elemente L (muchii) V = { v_1,....v_L}. Apoi avem două partiții (să nu se confunde cu graficul partionat)

$$P_1 = \{ V_1,...V_M \}$$

și

$$P_2 = \{ V'_1,....V'_M\}$$

cu P_1 conținând asignatiunile muchiilor la procesoarele nod de bit și P_2 conținând asignatiunile muchilor la procesoarele nod de verificare. Unde V_i(j) conține muchia care este procesată de procesorul nodului bit j în timpul i. π(k), $k \in \{1,\ldots,L\}$ este matricea de permutare de la P_1 la P_2. Toate subseturile V_i, V'_i, i = 1,...,M, având același număr de elemente $M \triangleq L/M$, unde P este numărul de biți paraleli și procesoare nod, iar L este numărul de muchii din cod. Subseturile conțin muchii care sunt accesate în același timp.

În cazul în care M nu este un divizor al lui L, muchiile fictive sunt introduse în V, P_1 si P_2. Deși, în final, aceste muchii fictive vor fi instalate pe decodor, și nu vor afecta operațiunile sale, deoarece sunt pur și simplu ignorate. Dat fiind P_1 și P_2, o funcție de mapare,este definită pentru a planifica muchiile variabile L la bancurile de memorie P, în așa fel încât, pentru toate i-urile din M, fiecare muchie din V_i este separată de bancul de memorie.

O funcție de mapare M : {1,...,L } → {1,...,P} este definită ca o funcție de mapare pentru (P_1, P_2) dacă îndeplinește următoarele condiții: j, j' =1 ,....,L, $j \neq j'$ [1].

$v_j, v_{j'} \in V_i$ pentru $i => M(j) \neq M(j')$ (4.1)

$v_j, v_{j'} \in V_i'$ pentru $i => M(j) \neq M(j')$ (4.2)

sau echivalent, elemente aparținând aceluiași subset în orice partiție sunt planificate în bancuri diferite de memorie.

Procedura pentru a găsi o funcție validă de mapare M, este obținută din mai multe cicluri, fiecare începând cu un loc liber în funcția de mapare și terminându-se când nicio coliziune nu are loc (coliziunile sunt depistate într-o funcție de mapare care nu îndeplinește funcțile 4.1 și 4.2). La finalul unui ciclu ,dacă numai există locuri libere, procedura este terminată, altfel un alt loc liber este ales.

Îl lăsăm pe L(V_i) (respectiv L(V'_i)) să aibă setul de valori {1,...,P } care nu este asignat la V_i (respectiv V'_i).

La începutul unui ciclu, o funcție, preliminară, de mapare $M^{(0)}$ hărți {1,..,L} este în setul {1,...., P}∪{∅}. $(M^{(0)})^{-1}(\emptyset)$ este setul de locuri libere. La pasul i al ciclului, o funcție, preliminară, de mapare actualizată $M^{(i)}$ este produsă. Unde părțile (0),(1)... se referă la variabilele în pasul corespunzător ciclului[1].

La începutul unui ciclu, un loc liber este identificat în poziția $j^{(0)} \epsilon\, V_{i\,(0)} \cap V'_{i\,(1)}$. Dacă intersectarea dintre $L(V_{i\,(0)})$ si $L(V'_{i\,(1)})$ nu este goală, atunci există o valoare care umple locul liber, fără a crea coliziuni. $M^{(1)}(j^{(0)})$ se va afla la intersecție , în timp ce pentru $j \neq j^{(0)}, M^{(1)}(j) = M^{(0)}(j)$.

Dacă, în orice caz, intersectarea dintre $L(V_{i\,(0)})$ și $L(V'_{i\,(1)})$ este goală, alege $m \epsilon L(V_{i\,(0)})$ și $n \epsilon L(V_{i\,(0)})$. Fixează $M^{(1)}\left(j^{(0)}\right) =$ m , în timp ce $j \neq j^{(0)}$, $M^{(1)}(j) = M^{(0)}(j)$. Apoi repetă pașii următori pentru k=1,2, ... :

1. Caută pentru
$j^{(2k-1)} \epsilon V'_{(2k-1)} \cap V_{(2k)}$
astfel în cat $M^{(2k-1)}\left(j^{(2k-1)}\right) = m$ și$j^{(2k-1)} \neq j^{(2k-2)}$.
În cazul în care $j^{(2k-1)}$ este găsit,
$M^{(2k-1)}\left(j^{(2k-1)}\right) = n$ și $M^{(2k-1)}(j) = M^{(2k-2)}(j)$, pentru $j \neq j^{(2k-1)}$ si trecem la pasul 2. Altfel oprim ciclul. [1]

2. Caută pentru
$j^{(2k)} \epsilon V_{i^{(2k)}} \cap V'_{i^{(2k)}}$
astfel în cat $M^{(2k-1)}\left(j^{(2k)}\right) = n$ si $j^{(2k)} \neq j^{(2k-1)}$.
În cazul în care $j^{(2k)}$ este găsit,
$M^{(2k)}\left(j^{(2k)}\right) = m$ și$M^{(2k)}(j) = M^{(2k-1)}(j)$, pentru $j \neq j^{(2k)}$ și trecem la pasul 1. Altfel oprim ciclul. [1]

În concluzie, pentru un ciclu, seturile $L(V_i)$ si $L(V'_i)$ $\forall i = 1, \dots, M$, trebuiesc actualizate.

4.1.4 Configurația circuitului a blocului de permutare a mesajelor

Funcția de mapare este apoi folosită pentru a determina configurația a trei blocuri, folosite pentru nodurile variabile ale jumătății de iterație, precum în figura 4.3.

Blocul din stânga este un crossbar (Rețeaua lui Benes), cu P pini de intrare și P pini de ieșire, folosiți pentru a implementa funcția de mapare, găsită mai devreme. Pinul i de intrare, la timpul j, este conectat la pinul de ieșire$\beta_j(i)$. Unde $\beta_j(i) = $ M((i-1)M + j) [1].

Blocurile de mijloc sunt bancuri intercalate care aranjează informația în ordinea cerută de procesoarele, de pe partea dreaptă, bazate pe, permutarea π. δ_i este definit, considerând setul M de indici k în $\{1,\dots,L\}$ în asa fel incat $M\ (k)\ =\ i$, denumit $M^{-1}(i)$. Dăm ecuația $k\epsilon M^{-1}(i)$, îl lăsăm pe j să fie număr întreg în $\{1,\dots,M\}$ pentru care j= $_{\text{M}}{}^{\text{k}}$, iar j' este numar întreg în $\{1,\dots,M\}$ pentru care $j' =\ {}_M\pi^{-1}(k)$. Unde M este egal cu modulo M. Apoi $\delta_i(j') = j$.[1]

Blocul din dreapta este un alt comutator crossbar, identic cu cel din stânga. Pinul de ieșire i, la timpul j, este conectat la pinul de intrare $\beta'_j(i)$. Unde $\beta'_j(i) = M(\pi((i-1)M + j))$.

În cealaltă jumătate de iterație, unde informația călătorește de la nodurile de verificare la nodurile variabile, ordinea blocurilor este inversată. Funcțiile blocurilor sunt de asemenea inversate, precum în figura 4.4. Pentru crosbarul din stânga , pinul de ieșire i la timpul j este conectat la pinul de intrare $\beta'_j(i)$. Intercalarea este dată de $\delta_i{}^{-1}$, iar pentru crossbarul din dreapta, pinul de intrare i, la timpul j, este conectat la pinul de ieșire $\beta_j(i)$.

Ecuația de mai sus este validă, atunci când P_1 este într-o ordine numerică. Când METIS este folosit, este foarte improbabil ca acesta să fie cazul, așadar muchiile sunt renumărate pentru a ne asigura că P_1 este într-o ordine numerică.

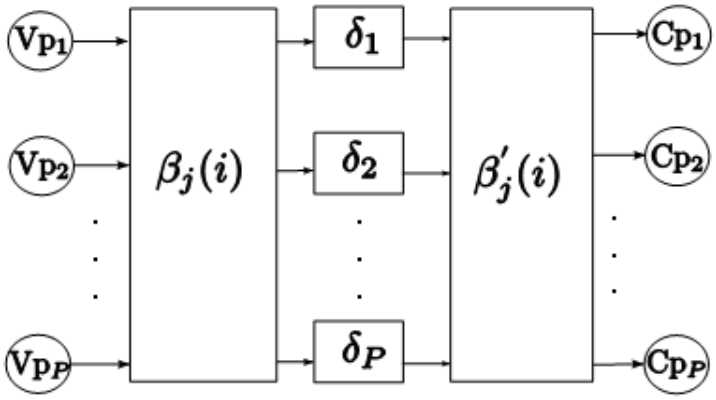


Figura 4.3: Configurația pentru nodul de variabile al jumătății de iterație

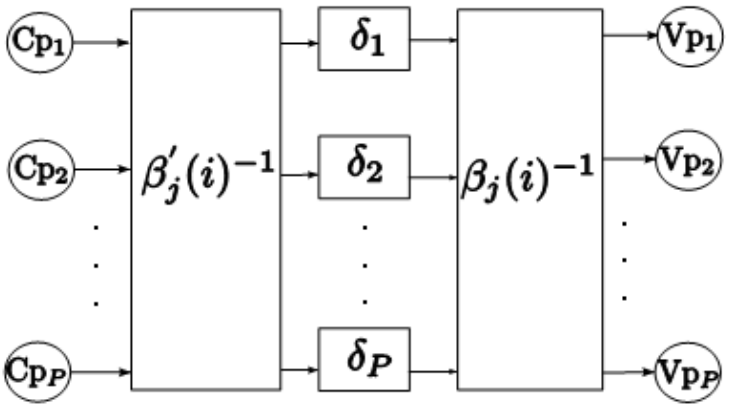


Figura 4.4: Configurația pentru nodul de verificare al jumătății de iterație

4.1.5 Exemplu

Considerăm (3,6) – regulată (8,4) şi rata codului 0.5, data de matricea de verificare a parității 4.3. În acest exemplu o să folosim 4 procesoare și sunt 24 de muchii, așa că rezultă:

L= 24

P= 4

$M = \frac{L}{P} = 6$

$$H = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \end{pmatrix} \quad (4.3)$$

Există L= 24 de muchii și P= 4 procesoare, rezultă M= 6 muchii pe procesor. Figura 4.5 prezintă graficul lui Tanner, reprezentat în 4.3, partiționat de mână.

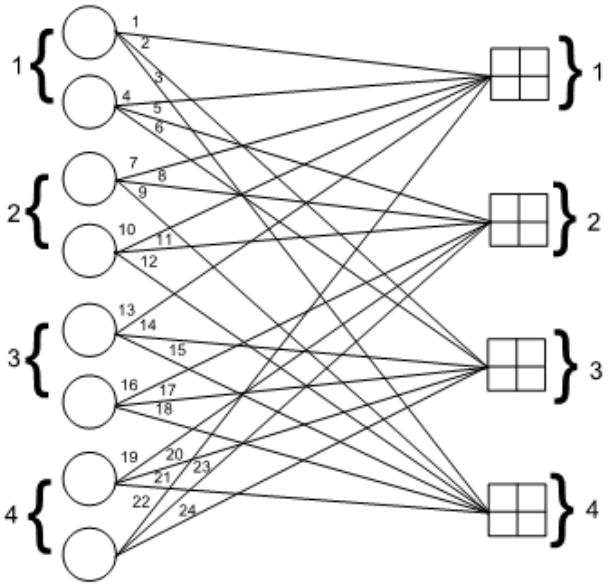


Figura 4.5: Graficul Tanner partiționat

Pentru a-l găsi pe $P_1 = \{V_1, \ldots, V_M\}$, observăm ca $V_i(j)$ conține muchia care este procesată de procesorul nodului bit j, la timpul i, adică i=1, procesorul 1 folosește muchia 1, procesorul 2 folosește muchia 7, procesorul 3 folosește muchia 13, iar procesorul 4 folosește muchia 19. Așadar V_1 = (1,7,13,19), continuând pentru toate i-urile date,

$$P_1 = \{(1,7,13,19), (2,8,14,20), (3,9,15,21), (4,10,16,22), (5,11,17,23), (6,12,18,24)\}. \quad (4.4)$$

Pentru a-l găsi pe $P_2 = \{V'_1, \ldots, V'_M\}$, observăm ca $V'_i(j)$ conține muchia care este procesată de procesorul nodului de verificare j'. la timpul i, adică i=1, procesorul 1 folosește muchia 1, procesorul 2 muchia 5, procesorul 3 muchia 2, iar procesorul 4 folosește muchia 3. Asta ne dă

$$P_1 = \{(1,5,2,3), (4,8,6,9), (7,11,14,12), (10,16,17,15), (13,19,20,18), (22,23,24,21)\}. \quad (4.5)$$

Acum putem observa că permutarea (π) pentru mapare de la P_1 la P_2 este :

$$\pi = (1,4,7,10,13,22,5,8,11,16,19,23,2,6,14,17,20,24,3,9,12,15,18,21), \quad (4.6)$$

și inversul π^{-1} este

$$\pi^{-1} = (1,13,19,2,7,14,3,8,20,4,9,21,5,15,22,10,16,23,11,17,24,6,12,18). \quad (4.7)$$

Este folositor să reprezentăm funcția de mapare M ca o matrice P x M, a cărei element (i,j) este:

$$M = \begin{pmatrix} M(1) & M(1) & \dots & M(M) \\ M(M+1) & M(M+2) & \dots & M(2M) \\ \dots & \dots & \dots & \dots \\ M(3M+1) & M(3M+2) & \dots & M(4M) \end{pmatrix} \tag{4.8}$$

Începem cu M compus din locuri libere, $L(V)$ și $L(V')$ conținând $\{1,\dots,4\}$ pentru toate i-urile. Începând $j^{(0)} = 1$ în prima poziție, putem observa că muchia 1 există în V_1 și în V'_1. $L(V_1) \cap L(V_1') = \{1,2,3,4\}$, alegând primul element avem M(1) =1 și

$$L(V) = \{(2,3,4),(1,2,3,4),(1,2,3,4),(1,2,3,4),(1,2,3,4),(1,2,3,4)\}.$$

$$L(V') = \{(2,3,4),(1,2,3,4),(1,2,3,4),(1,2,3,4),(1,2,3,4),(1,2,3,4)\}.$$

Continuând cu următorul loc liber, $j^0 = 2\epsilon V_2 \cap V'_1$. Avem
$L(V_1) \cap L(V_1') = \{2,3,4\}$, și din nou luând primul element avem $M(2) = 2$ și

$$L(V) = \{(2,3,4),(1,3,4),(1,2,3,4),(1,2,3,4),(1,2,3,4),(1,2,3,4)\}.$$

$$L(V') = \{(3,4),(1,2,3,4),(1,2,3,4),(1,2,3,4),(1,2,3,4),(1,2,3,4)\}.$$

Acest proces se repetă până când obținem următoarele :

$$M = \begin{pmatrix} 1 & 2 & 3 & 1 & 4 & 2 \\ 2 & 3 & 4 & 2 & 1 & 3 \\ 3 & 4 & 1 & 3 & - & - \\ - & - & - & - & - & - \end{pmatrix}$$

cu,

$$L(V) = \{(4),(1),(2),(4),(2,3),(1,4)\}.$$

$$L(V') = \{\emptyset,\emptyset,\emptyset,(4),(1,2,4),(1,2,3,4)\}.$$

Cu $j^{(0)} = 17\epsilon V_5 \cap V'_4$, avem $L(V_5) \cap L(V'_4) = \emptyset$. Așa că luând primul element din $L(V_5)$, avem $m = 2$, și similar $n = 4$. Îl setăm pe $M(17) = 2$ și continui la primul pas din partea iterativă din algoritmul descris în secțiunea 4.1.3.

Căutăm pentru $j^{(1)}\epsilon V'_4 \cap V_{i^{(2)}}$ așadar $M\left(j^{(1)}\right) = m$ si $j^{(1)} \neq j^{(0)}$. Candidații posibili sunt (10,16,15) și aflăm că $j^{(1)} = 10$ satisface condițiile cu $i^{(2)} = 4$. Apoi setăm $M(10) = n$. Acum continuăm cu al doilea pas din algoritm.

Căutăm pentru $j^{(2)}\epsilon V_4 \cap V'_{i^{(3)}}$ așadar $M\left(j^{(2)}\right) = n$ si $j^{(2)} \neq j^{(1)}$. Candidații posibili sunt (4,16,22) și aflăm că niciunul dintre ei nu satisface condițiile. Așa că ciclul se termină și continuăm cu

următorul loc liber aflat în funcția de mapare, după reactualizarea M, $L(V)$ si $L(V')$. Dacă am fi găsit un candidat $(j^{(2)})$, am fi setat $M(j^{(2)}) = m$ și am continua din nou cu pasul unu al algoritmului.

Acum avem :

$$M = \begin{pmatrix} 1 & 2 & 3 & 1 & 4 & 2 \\ 2 & 3 & 4 & 2 & 1 & 3 \\ 3 & 4 & 1 & 3 & 2 & - \\ - & - & - & - & - & - \end{pmatrix}$$

și

$$L(V) = \{(4), (1), (2), (2), (3), (1,4)\}.$$

$$L(V') = \{\emptyset, \emptyset, \emptyset, \emptyset, (1,2,4), (1,2,3,4)\}.$$

Urmând acest algoritm până la producția lui completă:

$$M = \begin{pmatrix} 1 & 2 & 3 & 1 & 4 & 2 \\ 2 & 3 & 4 & 2 & 1 & 3 \\ 3 & 4 & 1 & 3 & 2 & 2 \\ 4 & 1 & 2 & 1 & 3 & 4 \end{pmatrix}$$

$\beta_j(i)$ este dat de (i, j) elementul lui M. Pentru a găsi $\beta'_j(i)$ construim matricea ale cărei coloane corespund subseturilor V' al P_2 (ecuatia 4.5) :

$$B'_j(i) = \begin{pmatrix} M(1) & M(4) & M(7) & M(10) & M(13) & M(22) \\ M(5) & M(8) & M(11) & M(16) & M(19) & M(23) \\ M(2) & M(6) & M(14) & M(17) & M(20) & M(24) \\ M(3) & M(9) & M(12) & M(15) & M(18) & M(21) \end{pmatrix} = \begin{pmatrix} 1 & 2 & 2 & 4 & 3 & 1 \\ 4 & 3 & 1 & 3 & 4 & 3 \\ 2 & 1 & 4 & 2 & 1 & 4 \\ 3 & 4 & 3 & 1 & 2 & 2 \end{pmatrix}$$

Pentru a determina δ_1 găsim că $k \epsilon M^{-1}(1)$, dând $k = \{1,6,11,15,20,22\}$. Amintind ecuația 4.7 și începând cu k=1, avem

j= $_M1$ =1

j'= $_M\pi^{-1}(1)$ = $_M1$=1

așa că avem $\delta_1(j') = j$. continuăm cu $k = 6$ avem

j= $_M6$ =6

j'= $_M\pi^{-1}(6)$ = $_M14$=2

așa că avem $\delta_1(2) = 6$. Urmând acest proces vom completă:

$$\delta_1 = (1,6,5,3,2,4)$$

$$\delta_2 = (2,4,1,5,6,3)$$

$$\delta_3 = (3,2,6,4,1,5)$$
$$\delta_4 = (5,3,2,4,1,6)$$

Așadar acum avem parametrii necesari pentru blocuri, când nodurile variabile sunt active precum în figura 4.3. Când nodurile de verificare (vezi figura 4.4) sunt active și δ_i este înlocuit cu $\delta_i{}^{-1}$. $\beta_j(i)$ și $\beta'_j(i)$ sunt înlocuite cu această diferență importantă. Când nodurile de verificare sunt active, $\beta_j(i)$ specifică portul de ieșire i este conectat la $\beta'_j(i)$ portul de intrare la timpul j. Acesta este opusul a ceea ce se întâmplă când nodurile variabile sunt active, deci pentru a menține configurația crossbarurilor consistentă input-ului este i și output-ul este $\beta(i)$, spunem că parametrul pentru crossbarul din stânga este $\beta'_j(i)^{-1}$, iar parametrul pentru crossbarul din dreapta este $\beta_j(i)^{-1}$, precum este arătat mai jos.

$$\beta_j(i)^{-1} = \begin{pmatrix} 1 & 4 & 3 & 4 & 2 & 1 \\ 2 & 1 & 4 & 1 & 3 & 3 \\ 3 & 2 & 1 & 3 & 4 & 2 \\ 4 & 3 & 2 & 2 & 1 & 4 \end{pmatrix}$$

Capitolul 5

5 MATLAB Software

MATLAB a fost folosit pentru a dezvolta softului care acceptă un cod definit pentru matricea de verificare a parității, și produce fișierele ROM ce definesc programul pentru decodor. Fișierele ROM pot fi apoi descărcate pe decodor, permițând implementarea unui nou decodor LDPC.

MATLAB a fost, de asemenea, folosit pentru a dezvolta o bancă de testare pentru verificarea software a operațiunilor decodorului.

5.1 Matricea graficului Tanner

Codul explicat în Anexa A.1, matrix2graph.m convertește o matrice de verificare a parității într-un grafic Tanner și salvează rezultatele pe discul în pregătire pentru partiționarea graficului.

5.1.1 Partiționarea graficului

Fișierele graficului, rezultate din secțiunea anterioară, sunt pasate către METIS [21], un pachet de partiționare grafic. Graficul este partiționat în P partiții (numărul de procesoare care funcționează în paralel pe decodor), pentru a micșora numărul de intersecții dintre partiții. Acesta poate reduce potențialul număr de coliziuni din program și crește performanța decodorului , în medie, cu 20% [7]. Fișierele de ieșire din METIS trebuie să fie convertite în două seturi (P_1 și P_2) care conțin asignarea nodurilor la procesoare, în timpul jumătății de iterație bit și respectiv verificare. Anexa A.2 arată plist2part.m care este responsabilă pentru acest proces. METIS nu este optim pentru graficele bipartite și va produce partiții care nu conțin un număr egal de muchii atât în jumătățile de iterație bit cât și în cele de verificare. Ca o consecință plist2part.m încearcă să mențină neschimbate partițiile până la costul unei mici creșteri a intersectiunilor dintre partiții.

Dacă numărul de muchii din grafic nu este un divizor întreg al numărului de procesoare paralele, atunci muchii false trebuiesc introduse pentru a satisface condițiile. Aceste muchii vor fi puse pe decodor (vor fi așezate între nodurile bit și cele de verificare, prin permutarea mesajului bloc), dar nu vor afecta operațiunile sale, din moment ce decodorul știe exact câte muchii sunt conectate direct la fiecare nod prin blocurile de configurație ROM. În acest pas sunt adaugate (dacă este nevoie) muchii false.

5.2 Maparea

Următoarea sarcină este aceea de a găsi o funcție de mapare că în capitolul 4. Anexa A.3 arată codul responsabil pentru acest lucru. Funcția de mapare reprezintă fiecare muchie a codului din graficul lui Tanner la o bancă intercalată, asigurându-se că pot fi accesate în paralel, fără coliziuni. În momentul în care o funcție, validă, de mapare este găsită (satisfăcând ecuațiile 4.1 și 4.2) este folosită pentru a determina configurația pentru comutatoarele crossbar și băncile intercalate.

Programul care a fost găsit în această secțiune, trebuie procesat în plus. Este o implementare neutră, prin faptul că definește crossbarurile de intrare și de ieșire fără a ține seama de cum s-ar putea implementa. De asemenea ține seama doar de nodul bit al jumătății de iterație.

Este o relație simplă între nodul bit al jumătății de iterație și programarea nodului de verificare al jumătății de iterație care este explicat în secțiunea 4.1.4. Pentru a genera ieșirea unui decodor, că în

ecuația 1.9, mesajul bloc de permutare este folosit din nou pentru a obține o decizie grea a cuvântului de cod. Acesta cere un nou program, explicat în cele ce urmează.

Ieșirea unui decodor va fi trimisă către procesoarele nodurilor bit, grație rețelei Benes și spre bancurile intercalate. În timpul acestui proces rețeaua Benes din stânga este setată astfel încat sa pemute mesajele (intrarea n este conectat la ieșirea n). Asta înseamnă că banca intercalată n conține biți de cod asociati cu procesorul nodului bit n. Acum tot ce rămâne este să setăm băncile intercalate la comutatorul din dreapta, în așa fel încât codurile bit să apară în prima ieșire a mesajului bloc de permutare, în scopul de a produce cuvântul de cod decodat. Celelalte ieșire ale blocului de permutare a mesajelor sunt nefolosite. Aceste setări de comutare sunt aplicate la cele deja găsite.

5.3 Configurația rețelei Benes

Fișierele benes-main.m, benes.m, forward.m, backward.m și inverse.m în anexa A.4 implementează setările de comutare are rețelei Benes, descrise în secțiunea 3.3. Setările ieșiri de comutare sunt convertite în hexadecimal pentru a elimina orice neconcordanță potențială de precizie atunci când convertești din binar în decimal.

Sunt două componente recurente în setarea comutatoarelor, forward.m setează o matrice de intrare pe partea de input și solicită backward.m pentru a seta matricea de ieșire,corespunzătoare, părții de output. Dacă există un alt comutator pentru a seta forward.m solicită forward.m și procesul se repetă. A doua parte recurență este benes.m, care descompune comutatorul în sub-blocuri și se solicită repede în mod repetat, până când se ajunge la un comutator elementar, în timp ce se setează comutatoarele din afară.

5.4 Definirea fisierelor ROM

După ce programarea este definită și comutatoarele sunt setate ultima sarcină rămasă în obținerea fișierelor de configurare, pentru un cod particular, este aceea de a le salva în fisierel de definiție ROM pentru a fi utilizate de către programatorul hardware. Anexa A.5 arată cum print-files.m realizează acest lucru. Codul salvează configurația în formatul folosit de software-ul Altera's QUARTUS II.

5.5 Banca de test

În scopul de a verifica ieșirea decodorului, o bancă test Matlab a fost creată care implementează algoritmul sumă-produs, vezi Anexa A.6. Folosește același tabel de căutare ca și decodorul, singura diferență dintre decodor și versiunea Matlab este ordinea operațiilor de adunare. Aceasta poate produce o diferență între hardware și rezultatele Matlab, dar se așteaptă să fie una mică.

Interfața serială Matlab este folosită pentru a comunica cu decodorul pe o legătură în paralel, comparând rezultatele simulări cu hardware-ul.

Capitolul 6

6 Decodorul flexibil LDPC hardware

Decodorul flexibil LDPC a fost conceput într-un amestec de VHDL și captare schematică în Altera's Quartus II 7.0. Proiectul a fost directionat pe o placa de dezvoltare Altera's Cyclone I. Acest proiect poate implementa orice cod regulat (3,6) LDPC până la mărimea 480 x 240 și suportă 8 unități de procesare paralelă.

Proiectul complet la care se vrea a ajunge este direcționat către placa de dezvoltare Stratix II, și este gândit să suporte coduri cu dimensiune nominală de 1920x960 și coduri iregulate. Având de patru ori procesoarelor nodurilor bit și de verificare, va fi de patru ori mai rapid pe cicluri ceas ca proiectul actual.

Flexibilitatea conceptului, atât pentru proiectul complet cât și pentru proiectul actual, permite unui cod să fie decodat, schimbând configurația fișierelor ROM.

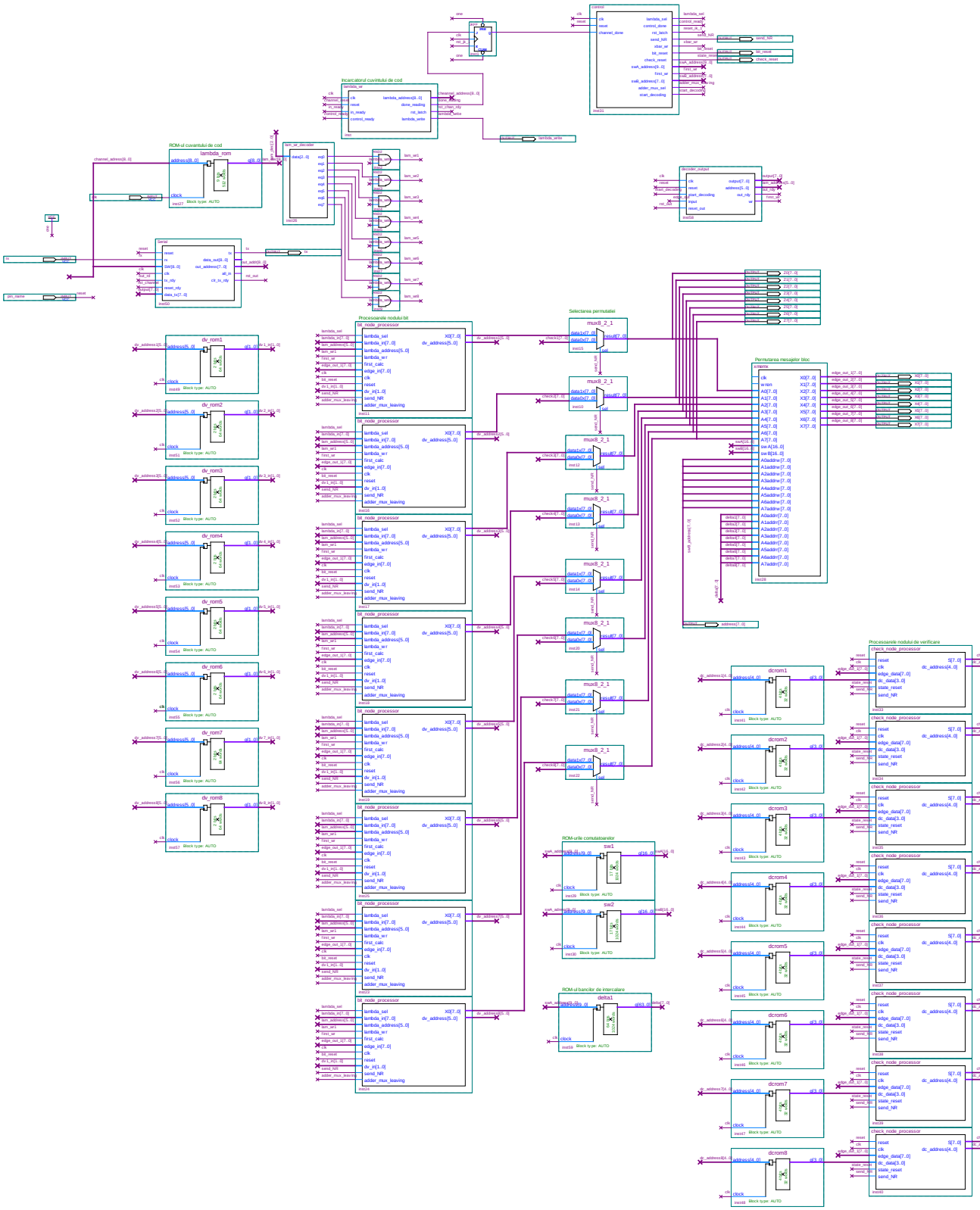


Figura 6.1: Diagrama bloc a decodorului.

6.1 Potentiale probleme de implementare

Primul pas în conceperea implementării a decodorului LDPC, a fost de a investiga potențialele probleme de implementare. Primul pas a fost de a investiga și de a decide asupra reprezentării numerelor din decodor, deoarece reprezentarea lor va influența direct fiecare componentă a conceptului decodorului.

6.1.1 Reprezentarea numărului

Numerele care sunt reprezentate pe hardware sunt mesajele care se deplasează prin suma rezultată a algoritmului. În algoritm numerele sunt reale, dar instanțierea multiplă a unor unități de puncte mobile, de a administra aceste sume, nu este practică, deci a fost făcut un compromis între acuratețea numerică și complexitatea conceptului, și au fost decise puncte fixe. Mulțimea numerelor este și pozitivă și negativă, așadar este folosit sistemul complet. Acest lucru înseamnă că numai un singur tip de bloc de adunare este necesar să adauge un amestec de numere pozitive și negative.

Acuratețea numerică afectează operațiunile decodorului, mai ales că este iterativ, cu toate că erorile pot fi mici pe un număr de iterații mai mare, acestea au potențialul de a se acumula. S-a constat că între 5 și 8 biți de precizie sunt necesari pentru ca un decodor LDPC să întâlnească normele standard BER [15,16]. Reprezentarea pe 8 biți cu partea fracționată 4 biți a fost decisă. Această oferă o gamă de 8.000 - 7.9375 cu trepte de 0,0625. Gama va fi restricționată pe partea negativă a -7.9375, pentru a elimina o posibilă confuzie în decodor pentru numerele negative.

6.1.2 Funcții de multiplicare și trigonometrice

Cele două operații ale algoritmului sumă- produs văzut în secțiunea 1.3.3, sunt reproduse aici pentru clarificare.

$$Q_{i,j} = \lambda_j + \sum_{\alpha \in C[j] \alpha \neq i} R_{\alpha j}[k-1] \tag{6.1}$$

$$R_{i,j} = 2\tanh^{-1} \prod_{\alpha \in V[j] \alpha \neq i} \tanh\left(\frac{Q_{\alpha j}}{2}\right) \tag{6.2}$$

Se poate observa că ecuația 6.1 conține numai sume care se pot implementa cu ușurință în hardware, pe când ecuația 6.2 conține atât funcții de multiplicare cât și trigonometrice. În timp ce funcțiile trigonometrice pot fi implementate pe hardware [22], numărul care ar fi cerut pentru acest proces este prohibitiv. Numărul de multiplicatori necesari este de asemenea prohibitiv. Din fericire există o soluție atât pentru tangentă hiperbolică cât și pentru problema multiplicatorilor. Introducem o funcție în așa fel încât :

$$\psi(x) = -\ln\left(\tanh\left|\frac{x}{2}\right|\right) = \ln\frac{\left(1+e^{-|x|}\right)}{\left(1-e^{-|x|}\right)} \tag{6.3}$$

Acum că tanh (x) este o funcție pară ,

$$\tanh\frac{x}{2} = \tanh\left|\frac{x}{2}\right| \cdot \delta_{ij} \tag{6.4}$$

Unde δ_{ij} este produsul mesajelor ,

Substituind Ecuația 6.3 în 6.2 obținem:

$$R_{i,j}[k] = 2\tanh^{-1}\left(\exp\left(-\sum_{\alpha\in V[j]\alpha\neq i}\psi\left(Q_{\alpha j}[k-1]\right)\right)\cdot\delta_{ij}\right) \tag{6.5}$$

$$\tanh^{-1}(x) = \frac{1}{2}\ln\left(\frac{1+x}{1-x}\right)$$

putem trage δ_{ij} afară și să se simplifice.

$$R_{i,j}[k] = \ln\frac{1+\exp\left(-\sum_{\alpha\in V[j]\alpha\neq i}\psi\left(Q_{\alpha j}[k-1]\right)\right)}{1-\exp\left(-\sum_{\alpha\in V[j]\alpha\neq i}\psi\left(Q_{\alpha j}[k-1]\right)\right)}\cdot\delta_{ij}$$

$$R_{i,j}[k] = \psi\left(\sum_{\alpha\in V[j]\alpha\neq i}\psi\left(Q_{\alpha j}[k-1]\right)\right) \tag{6.6}$$

Așa că acum am eliminat multiplicarea în decodor, înlocuindu-l cu sumatoare și introducând o nouă funcție ψ. Detalii cu privire la punerea în aplicare a implementării vor fi discutate în secțiunea următoare.

6.1.3 Funcții non liniare

Funcția ψ este extrem de neliniară, nelimitată din cauza asimptotei verticale la x = 0. Trei tipuri de implementații care au fost discutate în literatură au fost găsite [7,23], o implementare Piece-wise Linear (PWL), o aproximare folosind baza 2 aritmetică și un direct Lookup Table (LUT). S-a descoperit că implementarea PWL a fost capabilă să funcționeze mult mai bine cu 3 biți mantisă și 4 biți fractionali.[7].

Am comparat două opțiuni, o implementare PWL și una LUT. Coeficienții au fost concepuți pentru a fi ușor reprezentați în notații de puncte fixe, pentru a evita multiplicările, în schimb o serie de comutări și sumatoare au fost folosite.

LUT a fost bazat pe [23] cu 256 de intrări în tabel. La construirea și compararea implementărilor PWL și LUT s-a aflat că PWL este foarte scump în termeni de resurse aproape prohibitivi, așa că, comparativ cu o implementare directă LUT, PWL ocupă 76% mai mult spațiu și era cu 8% lent [23].

S-a decis să se pună în aplicare o funcție cu LUT, precum în B.1.

6.2 Sumatoare

Pentru decodorul LDPC, există o nevoie de a adaugă, în paralel, multiple mesaje de intrare. Numărul de intrări la sumatoare dictează gradul de greutate suportate de nodurile de bit și de verificare. Gradul de greutate maximă suportat de nodurile de verificare este numărul de intrări în sumator, în timp ce pentru nodurile bit este cu unul mai puțin, datorită uneia dintre intrările utilizate pentru cananul de intrate ,conform ecuației 6.1.

Pentru proiect s-a decis să se susțină o greutate maximă de grad 3 pentru nodurile bit și de 6 pentru nodurile de verificare. S-a decis să se ocupe de intrările paralele arborele sumatoarelor (tree adder), spre deosebire de transferul sumatoarelor salvate (carry save adders), datorită complexității acestuia din urmă și a numărului mic de operanzi. Sumatoarele, ripple, carry-look a head și carry select au fost investigate și clasificate pe baza complexității și a performanței lor. S-a găsit că sumatorul, carry select, a fost cel mai rapid și complex (și consumă cea mai multă putere), în timp ce blocul ripple era cel mai încet, dar mai puțin complex [24].Blocul carry look ahead era cea mai bună combinație dintre viteza și complexitate, așa că a fost ales pentru decodor.

Sumatorul carrry look ahead calculează, în avans, semnalele carry, bazate pe semnalele input. Implementarea blocului de adunare carrry look ahead se bazează pe codul VHDL, oferit în [24], precum în Anexa B.2.

În acest concept nu există niciun procesor tradițional, deci nu există nicio verificare a condiților de eroare, precum cele de overflow. Asta înseamnă că erorile din sumatoare vor trece neverificate, de exemplu 4.5 + 4.5 = -7, asta este o eroare serioasă. Pentru a elimina această eroare output-ul sumatorului este extrem de limitat, dacă merge peste șirul lui ori în direcția pozitivă ori în cea negativă. Output-ul fiecărui sumator este verificat pentru overflow și dacă s-a întâmplat, output-ul este setat la limita pozitivă sau negativă, aproximativ, 7,9375. Figura 6.2 prezintă 8 input-uri tree adders folosind blocuri de adunare carry look ahead.

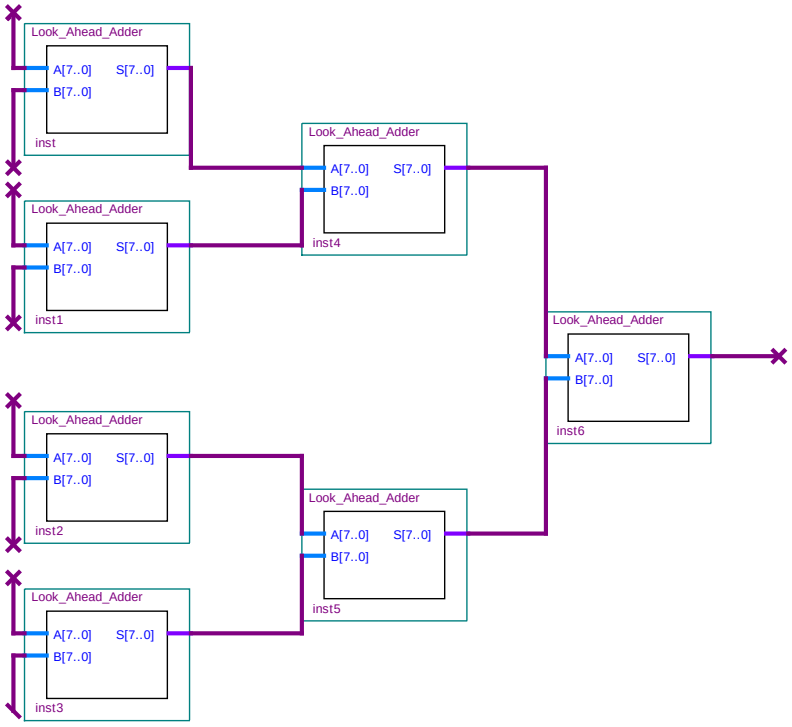


Figura 6.2: Intrarile sumatorului

6.3 Procesorul nodului bit

Procesorul nodului bit este responsabil pentru receptarea mesajelor de la nodul de verificare către blocul mesajului de permutare, procesând mesajele și trimițându-le în blocul mesajului de permutare. Este important să reținem că procesorul nodului bit este responsabil pentru păstrarea urmei nodurilor bit, individuale, în cod. Din perspectiva unității de control LDPC, nodurile bit trimit un flux de mesaje, fără nicio distincție care să arate cărui nod de aparțin.

Procesorul nodului bit are un semnal de control, care îi controlează funcția, citind mesajele în mesajul RAM, sau trimițând mesaje. Semnalul are efect doar atunci când procesorul este proaspăt resetat. Figura 6.3 arată diagrama bloc a procesorului nodului bit.

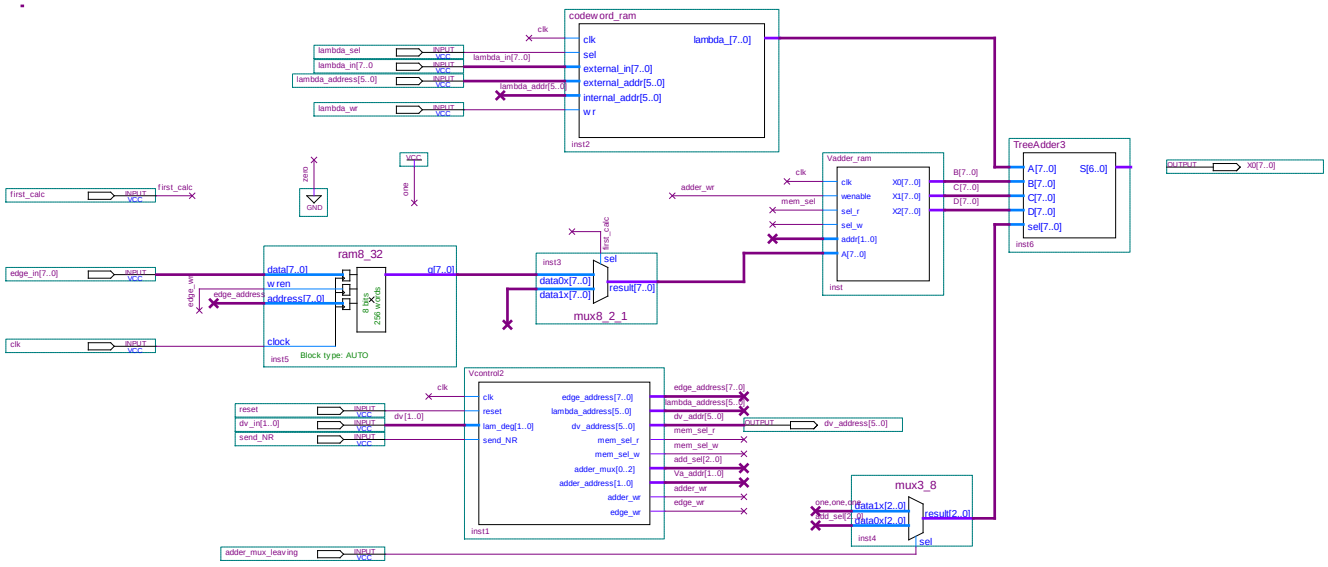


Figura 6.3: Diagrama bloc a procesorului nodului bit

6.3.1 Unitatea RAM a sumatorului

Pentru a procesa un mesaj pe ciclu de ceas, blocul trebuie să aibă toate mesajele asociate unui nod bit disponibil. Pentru a face posibil acest lucru unitatea RAM a sumatorului, implementează un convertor serial la paralel, făcând ca mesajele asociate cu nodul curent, să fie procesat de cel disponibil. Când vine timpul de a procesa un nou nod bit, procesorul, normal, va aștepta convertorul serial la paralel să încarce toate mesajele necesare. Pentru a evita oprirea procesorului pentru fiecare nod bit ,convertorul are două memorii. Încarcă în serie mesajele pentru următorul nod bit, într-o memorie, iar pe cealaltă o încarcă cu mesajele nodului bit curent, fiind disponibil în paralel pentru blocul de adunare. Unitatea de control este responsabilă cu comutarea memoriilor către linia de control. Codul care implementează acest bloc poate fi observat în anexa B.3.

6.3.2 Unitatea RAM a cuvantului de cod

Cuvântul de cod care încarcă procesul (Secțiunea 6.5) durează atâtea cicluri de ceas cât lungimea unui cuvânt de cod, care este un timp lung pentru a opri decodorul. Procesul de decodare are nevoie de mai multe cicluri de ceas decât numărul de biţi de cod, așa că este timp, cât decodorul procesează mesajele, să încarce următorul cuvânt de cod (dacă este disponibil) .Lucrează cam în același fel ca unitatea RAM a sumatorului, numai că este controlat de unitatea principală de control.

Sunt două RAM-uri ale cuvântului de cod, cu linia de control care selectează care este disponibil pentru blocul nodului bit și care pentru a încărca nodul bit. Cu această configurație decodorul este capabil să încarce următorul cuvânt de cod, în timp ce îl procesează pe cel curent. La concluzia de decodare a unui cuvânt de cod, unitatea de control semnalează încărcătorul cuvântului de cod ca un nou cuvânt de cod poate fi încărcat. Figura 6.4 arată o diagramă bloc a unității.

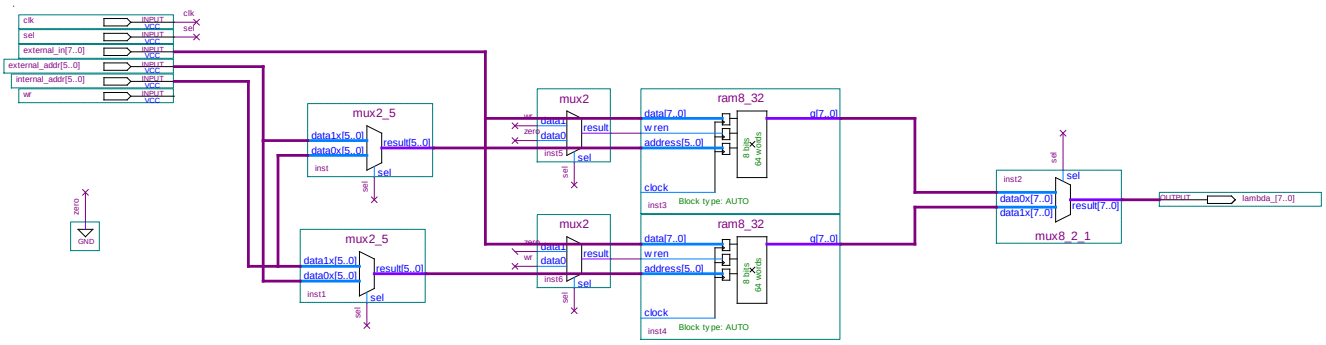


Figura 6.4: Diagrama bloc a cuvantului de cod unitati RAM

6.3.3 Unitatea de control a procesorului nodului bit

Când procesorul nodului bit primește mesajele, unitatea de control setează scrisul posibil pentru mesajul RAM și crește adresa, asigurându-se că mesajele care sosesc sunt stocate în locația corectă.

Când procesorul procesează și trimite mesajele, unitatea de control încarcă adresa pentru următorul cod bit în unitatea RAM a sumatorului. Acesta introduce un singur nod bit, latent, în decodor, înainte ca primul mesaj al nodului bit să fie trimis. Sumatoarele pentru procesor au input-uri, în acest fel mesajele trimise de-a lungul unei muchii a graficului Tanner nu conține mesajul care vine din verificarea ,anterioară, a nodului de jumătate de iterație. Acest lucru este atins cu multiplexoare care trimit zero la sumatoare, în loc să trimită mesajul.

Unitatea de control incrementează adresa canalelor de măsurare ale blocului RAM, astfel încât canalele de măsurare asociate cu nodul bit, procesat curent, este disponibil pentru sumator. De asemenea unitatea de control crește adresa pentru gradul de greutate a blocului RAM , care este folosită de unitatea de control pentru a determina câte muchii sunt încărcate în unitatea blocului RAM pentru fiecare nod bit. Deși conceptul folosește un grad de greutate fix (coduri regulate), se speră că conceptul final să aibă variabil gradul de greutate (coduri iregulare), așa că unitatea de control pentru procesoarele nod bit (și de verificare) au fost create mergând pe această idee.

Procesorul nodului bit încarcă numărul corect de mesaje pentru următorul nod bit pentru a fi procesat, citindu-i gradul de greutate din gradul de greutate RAM. În același timp procesează nodul bit curent , setând adresa codului de cuvânt RAM, așa că sumatorul deține codul bit curent. Când mesajele nodului bit sunt calculate, unitatea de control dezactivează blocul input corespunzător, eliminând efectul mesajului de primire de la mesajul de trims.

La prima iterație mesajul RAM este neinițializat, așa că unitatea principală de control revendică un semnal de control care evită mesajul RAM către un multiplexor. Codul care implementează unitatea de control a procesorului nodului bit poate fi găsită în anexa B.3.

6.4 Procesorul nodului de verificare

Procesorul nodului de verificare este identic cu procesorul nodului bit, cu exceptia sumatoarelor, neexistând cuvânt de cod RAM asociat cu unitatea de control. Pentru procesorul nodului de verificare, blocul de adunare acceptă 8 input-uri (doar 6 sunt folosite), La fiecare bloc intrarea are un tabel de căutare $\psi(x)$, output-ul blocului este trecut către un alt tabel de căutare $\psi(x)$ care de asemenea performează corectatea semnului descris în secțiunea 6.1.2. Corectarea de semn este făcută de bitul de semn (MSB) al XOR-ului, împreună cu input-urile și dacă rezultatul este '1', atunci rezultatul LUT este făcut negativ. Figura 6.5 arată o diagramă bloc a procesorului nodului de verificare, în timp ce anexă B.4 prezintă codul pentru unitatea de control.

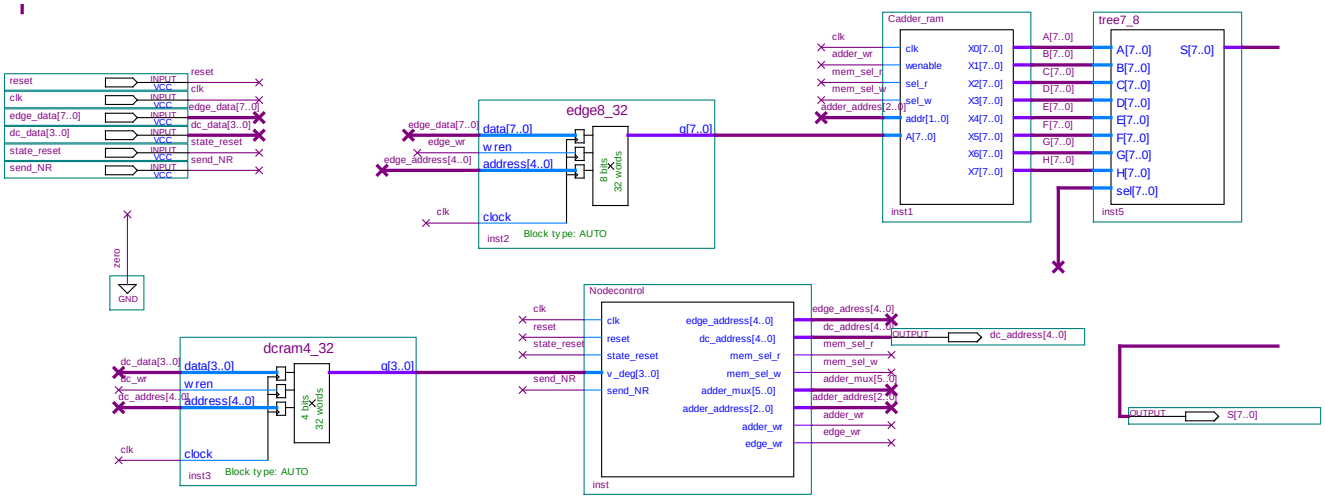


Figura 6.5: Diagrama bloc a procesorului nodului de verificare

Când un input nu este necesar pentru sumatorul procesorului nodului de verificare, input-ul lui este înlocuit cu o reprezentație binară a 7.9375 (01111111), aceasta pentru că , iar în acest fel input-ul nefolosit nu va afecta suma. Figura 6.6 arată blocul procesorului nodului de verificare.

6.5 Incarcatorul cuvantului de cod

Pentru că decodorul flexibil LDPC să fie capabil să decodeze un cuvânt de cod, trebuie să fie capabil să încarce un cuvânt de cod, această sarcină îi este desemnată încărcătorului cuvântului de cod. Acesta așteaptă un semnal de la unitatea IO, însemnând că un nou cuvânt decod este disponibil. ROM-ul cuvântului de cod conține fiecare procesor nod bit, fiecare bit de cod fiind desemnat în ordine. Încărcătorul cuvântului de cod folosește ROM-ul ca să selecteze procesorul nodului bit desemnat și adresa fiecărui bit de cod. Figura 6.7 prezintă o diagramă bloc a încărcătorului cuvântului de cod, în timp ce anexă B.5 arată codul pentru implementarea blocului de control.

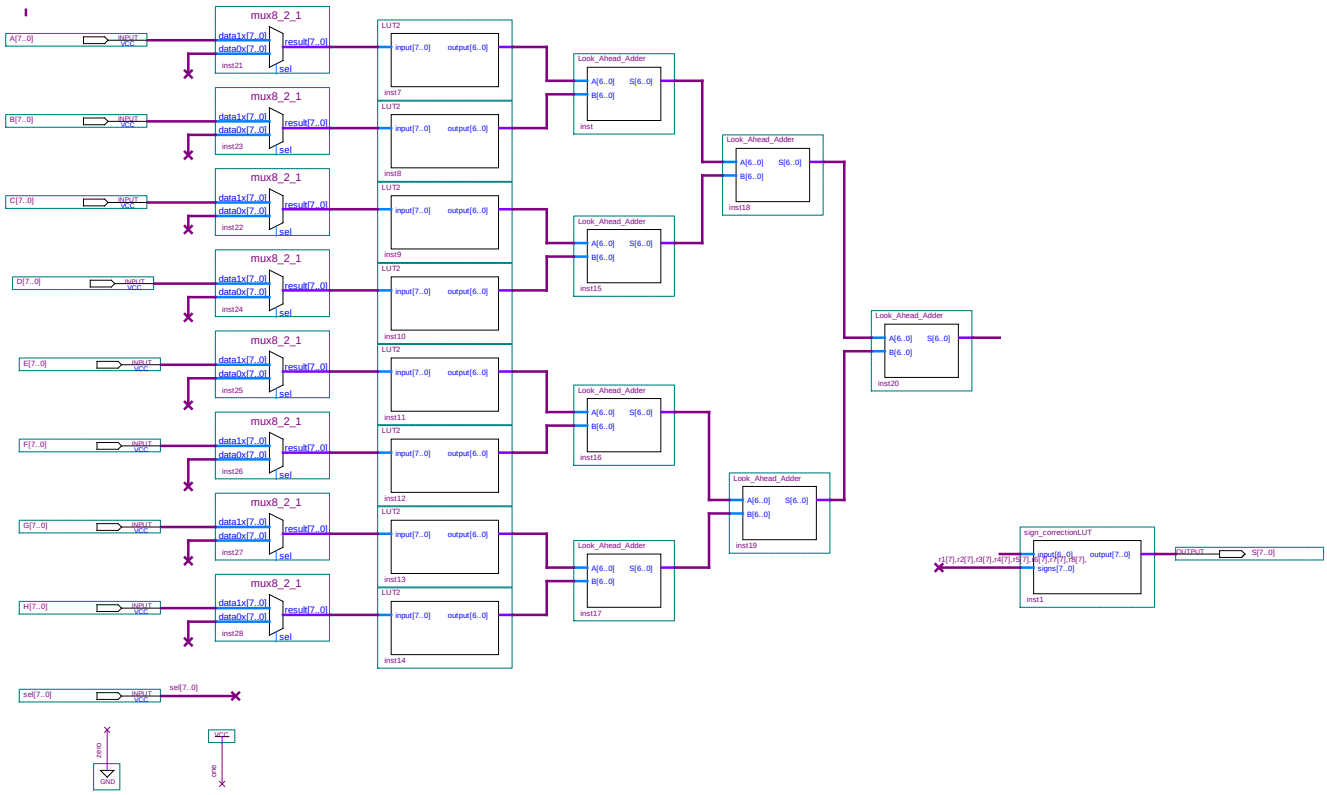


Figura 6.6: Diagrama bloc a sumatorului procesorului nodului de verificare

6.5.1 Rețeaua de permutare a mesajului

Rețeaua de permutare a mesajului stă la baza flexibilității decodorului LDPC. Mesajele sosite sunt primele permutate în spațiu prin rețeaua Benes, apoi în timp prin băncile intercalate și în final în spațiu din nou prin a doua rețea Benes. Dând programul corect, definit în capitolul 4, orice cod poate fi suportat de rețeaua de permutare. Figura 6.8 arată o diagramă bloc a rețelei mesajului de permutare.

6.5.2 Reteaua Benes

Rețele Benes folosesc comutatoarele crossbar deoarece au cea mai scăzută complexitate a implementării și cerințele logice de control ale oricărei alternative [16,7]. Rețeua Benes este contruită recursiv începând de la comutatoare elementare 2x2 și folosindu-le pentru a crea un comutator 4x4 apoi folosindu-l pentru a crea un comutator 8x8 și tot așa.

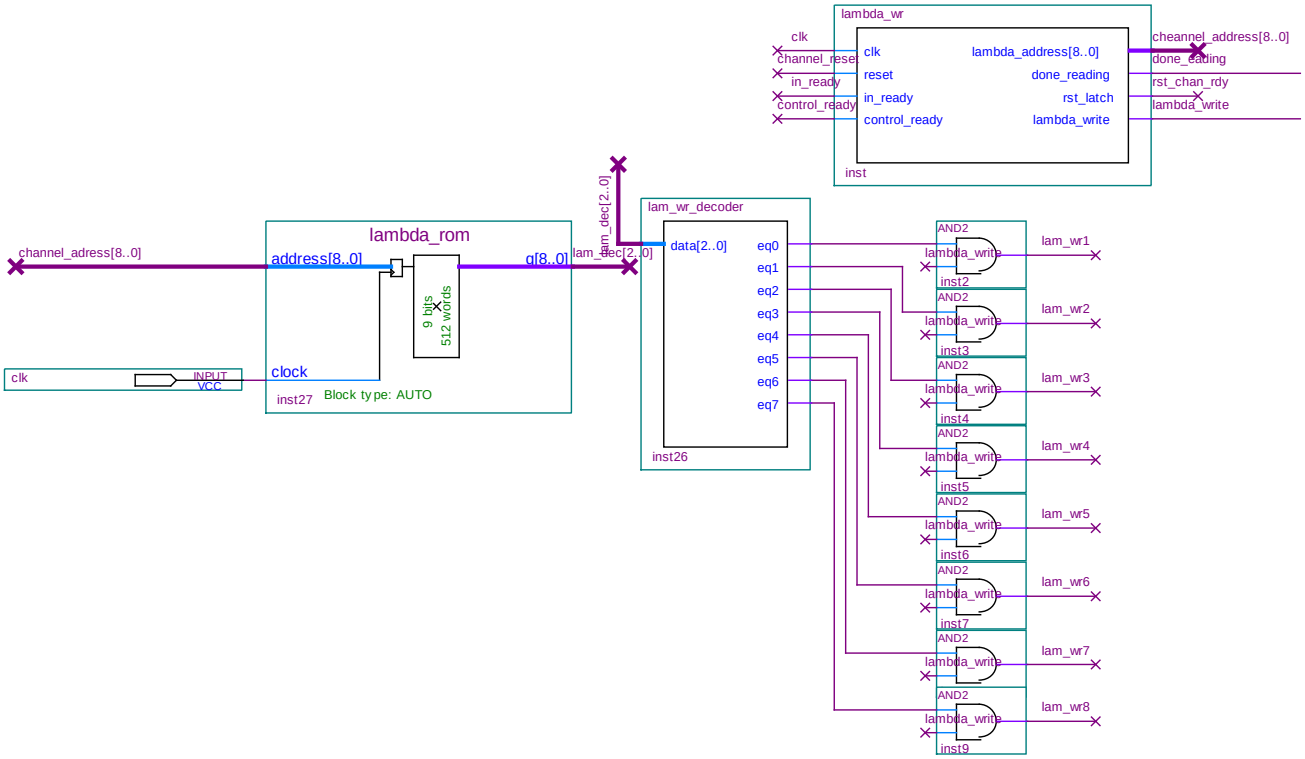


Figura 6.7: Diagrama bloc a incarcatorului cuvantului de cod

Rețeaua este controlată setând și resetând fiecare celulă a comutatorului elementar 2x2, folosind metodologia explicată în secțiunea 3.3. Input-urile de control sunt citite de pe input-ul și output-ul filelor comutatorului ROM. Adresa de la fiecare configurație este controlată de unitatea de control. Există mai multe configurații output-ul filelor comutatorului ROM, deoarece este folosit de output pentru a decoda cuvântul de cod. Figura 6.9 arată o diagramă bloc a rețelei Benes.

6.5.3 Bancile intercalate

Băncile intercalate sunt implementate în blocurile RAM ca un port dual. Adresa scrisă este sporită , stocând mesajele primite în ordinea în care sunt primite, în timp ce adresă citită reprezintă ordinul de permutare prin care sunt citite la iesire. Ordinul de citire al băncii intercalate i este setat , definit în secțiunea 4.1.4. Diagrama blocului intercalată este reprezentată în figura 6.10.

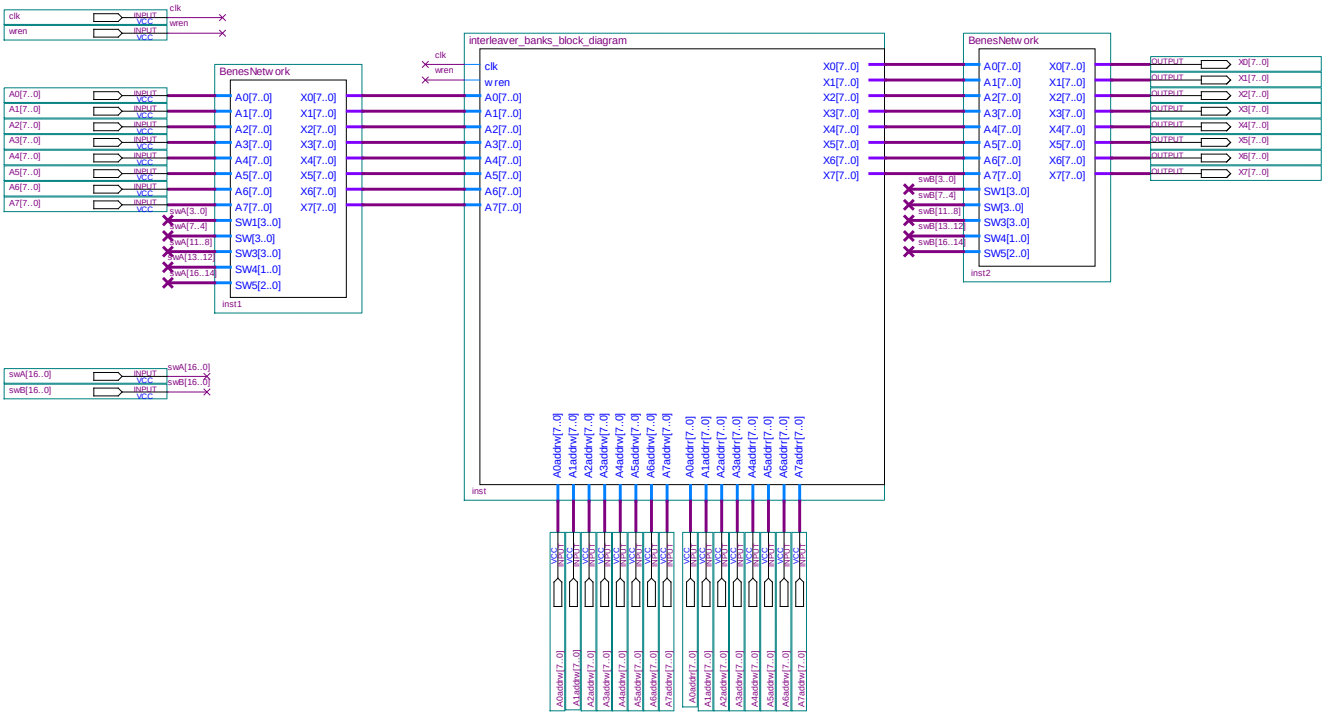


Figure 6.8: Schema bloc a retelei de permutare a mesajelor

6.6 Iesirea decodorului

Blocul de iesire al decodorului citește cuvântul de cod decodat, de la primul ouput al blocului mesajului de permutare. Performează o decizie grea asupra fiecărui bit de cod, stocând semnul. Dacă bitul de cod este negativ stochează ’1’, dacă este pozitiv stochează ’0’. Produce o conversie paralela-serie de 8 biți pe timp, stocând rezultatul în RAM-ul decodat. Astfel fiecare cuvânt al RAM-ului decodat are 8 decizii grele ale codului de biți decodat.

6.7 Input si Output

Input-ul sin output-ul de pe placă pentru acest proiect este portul serial RS-232. Un UART a fost creat, bazat pe codul lui Juin [25] . Parametrii de comunicare sunt 28800 de biți/ secundă, 8 biți de date, 1 bit de stop cu paritate chiar. Input-ul este primit de la banca de test MATLAB și când un cod de cuvânt complet este primit, un semnal este trimis la încarcatorul cuvântului de cod, astfel poate fi încărcat în procesorul nodului bit corect.

Când codul de cuvânt este decodat, la concluzia unității de control in stadiul final, blocul IO trimite conținutul RAM-ului decodat prin UART, înapoi la banca de test MATLAB.

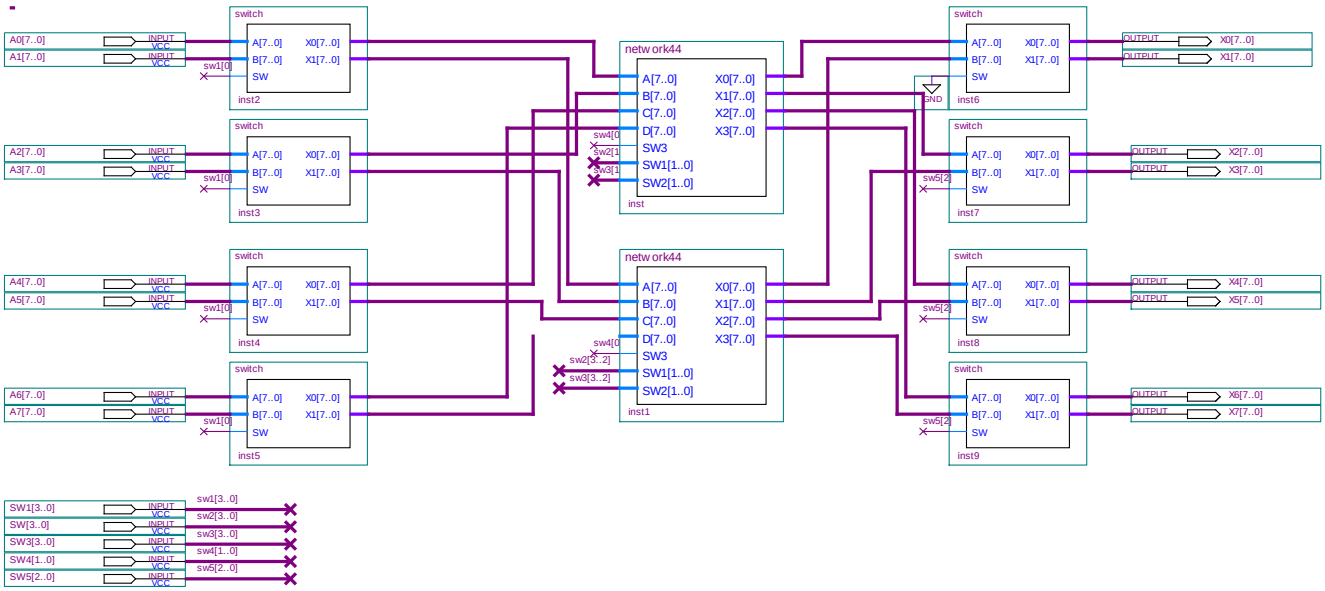


Figura 6.9: Shema bloc a rețelei Benes

6.8 Unitatea de control

Unitatea de control principală implementează 12 nivele ale stări mașinii, controlând toate părțile decodorului. Unitatea de control citește comutatorul și ROM-ul intercalat pentru a controla rețeaua de permutare, care reprezintă inima flexibilității decodorului. Diagrama stării de tranziție este prezentată în figura 6.11.

Starea idle

Decodorul pornește în acest stat și rămâne aici până când un nou cod de cuvânt este disponibil pentru decodare. În acestă stare scrisul pentru rețeaua de permutare este dezactivat. Când un cuvânt de cod este disponibil, decodorul continuă cu starea mesaj nou, selectând semnalul cuvântului de cod al RAM-ului, astfel noul mesaj este disponibil pentru procesorul nodului bit. Procesoarele de verificare și bit sunt ținute în starea de resetare.

Starea new_message

În această stare, unitatea de control stabilește un semnal pentru a ocoli blocul mesajului RAM al nodului bit când acesta este neinițializat. Decodorul purcede direct în starea bit de așteptare și renunta la resetarea procesoarele nodului bit.

Starea bit_wait

Renunțând la resetarea procesoarelor nodurilor bit, începe transmiterea mesajelor, dar există o latență introdusă pe un nod bit (în proiect aceasta este 3 cicluri de ceas), de sumatoarele nodurilor bit, astfel încât decodorul trebuie să aștepte până la ieșirea mesajelor procesorului nodului bit. Cum decodorul continuă la starea de bit_in, se incrementează contorul de iterații.

Starea bit_in

În această stare scrierea este activă în blocul de permutare și mesajele de la procesorul nodului bit sunt scrise acolo. Blocul de control incrementează adresa comutatorului ROM, astfel mesajele nodului bit sunt stocate în blocurile intercalate corecte. Când toate mesajele de la procesorul nodului bit au fost scrise în blocul de permutare a mesajelor, decodorul continuă la starea de bit_out.

Starea bit_out

Resetarea procesoarelor nodurilor de verificare este lăsată, în timp ce scrierea blocului mesajelor de permutare este dezactivată. Decodorul incrementează adresa comutatorului și ROM-ului intercalat, astfel nodul de verificare primește mesajul corect. Când toate mesajele au fost încărcate în mesajul RAM al procesorului nodului de verificare, decodorul trece la starea de bit2check.

Starea bit2check

In aceasta stare functiile procesoarelor nod bit si de verificare sunt inversate. Nodurile de verificare vor trimite mesaje si nodul bit le va primi. Input-ul in blocul mesajelor de permutare este setat la procesorul nodului de verificare. Procesoarele nodurilor de verificare sunt resetate pentru un ciclu de ceas pentru a fi comutate de la receptarea mesajelor trimise. Decodorul trece la starea check_wait.

Starea check_wait

Ca în starea bit_wait, decodorul trebuie să accepte un nod de verificare (în proiect asta înseamnă 6 cicluri pe ceas) pentru procesoarele nodurilor de verificare să înceapă să trimită mesaje. Decodorul trece la starea check_in.

Starea check_in

În această stare, scrisul este activat pentru blocul mesajelor de permutare și mesajele din procesoarele nodurilor de verificare sunt stocate în blocurile intercalate. Blocul de control incrementează adresa comutatorului ROM pentru a se asigura că mesajele nodurilor de verificare sunt stocate corect în blocurile intercalate. Când toate mesajele provenite din procesoarele nodurilor de verificare au fost scrise în blocul de permutare a mesajelor, decodorul trece la starea de check_out.

Starea check_out

Resetarea procesoarelor nod bit este lasată în timp ce scrisul pe blocul de permutare a mesajelor este dezactivat. Decodorul incrementează adresa comutatorului și a ROM-ului intercalat, astfel încât procesoarele nodului bit pot să primească mesajele corect. Când toate mesajele au fost încărcate în mesajul RAM al procesoarelor nodului bit, decodorul trece la starea check2bit.

Starea check2bit

În această stare funcțiile procesoarelor nodului bit și a nodului de verificare sunt inversate. Procesoarele nodului bit vor trimite mesaje, iar procesoarele nodului de verificare vor primi mesaje. Input-ul în blocul mesajului de permutare este setat la procesoarele nodului bit. Procesoarele nod bit

sunt resetate pentru un ciclu de ceas, pentru a le comuta din a primi la a trimite mesaje. Dacă calculul de iterații este mai mic decât numărul predeterminat (10 în proiect) , decodorul trece la starea de bit_wait, dacă nu trece la starea final_wait.

Starea final_wait

Această stare este similară cu starea bit_wait, resetarea fiind lasata la procesoarele nodului bit, decodorul așteaptând până când procesoarele nodului bit încep să trimită mesaje. Unitatea de control setează un semnal care cauzează ca toate input-urile în blocul de adunare al procesorului nodului bit, să fie folosite. Pentru a face acest lucru procesoarele nodului bit calculează următoarele pentru fiecare nod bit, implementând ecuația 1.10, calculând astfel mesajele gata pentru decizie grea de decodare.

$$L_j = \lambda_j + \sum_{\alpha \in C[j]} R_{\alpha j}[k] \tag{6.7}$$

Decodorul trece la stare final_in.

Starea final_in

În această stare mesajele de la nodurile de verificare bit sunt stocate în băncile intercalate. Comutatorul de intrare al ROM-ului este nefolosit, cu toate acestea, intrarea de la fiecare procesor este stocata în respectivele banci de intercalare. Când toate mesajele au fost trimise către băncile intercalate, decodorul trece la starea final_out.

Starea final_out

Decodorul incrementează adresa output-ului comutatorului și ROM-ului intercalat. Acest lucru produce cuvantul de cod decodat pe primul output al rețelei de permutare. Decodorul blocului de iesire efectuează o decizie grea asupra cuvântului de cod și depozitează rezultatele în RAM-ul decodat. Acum, decodorul a decodat un cod de cuvânt și o ia de la capăt, cu starea inactivă, pentru a decoda un altul.

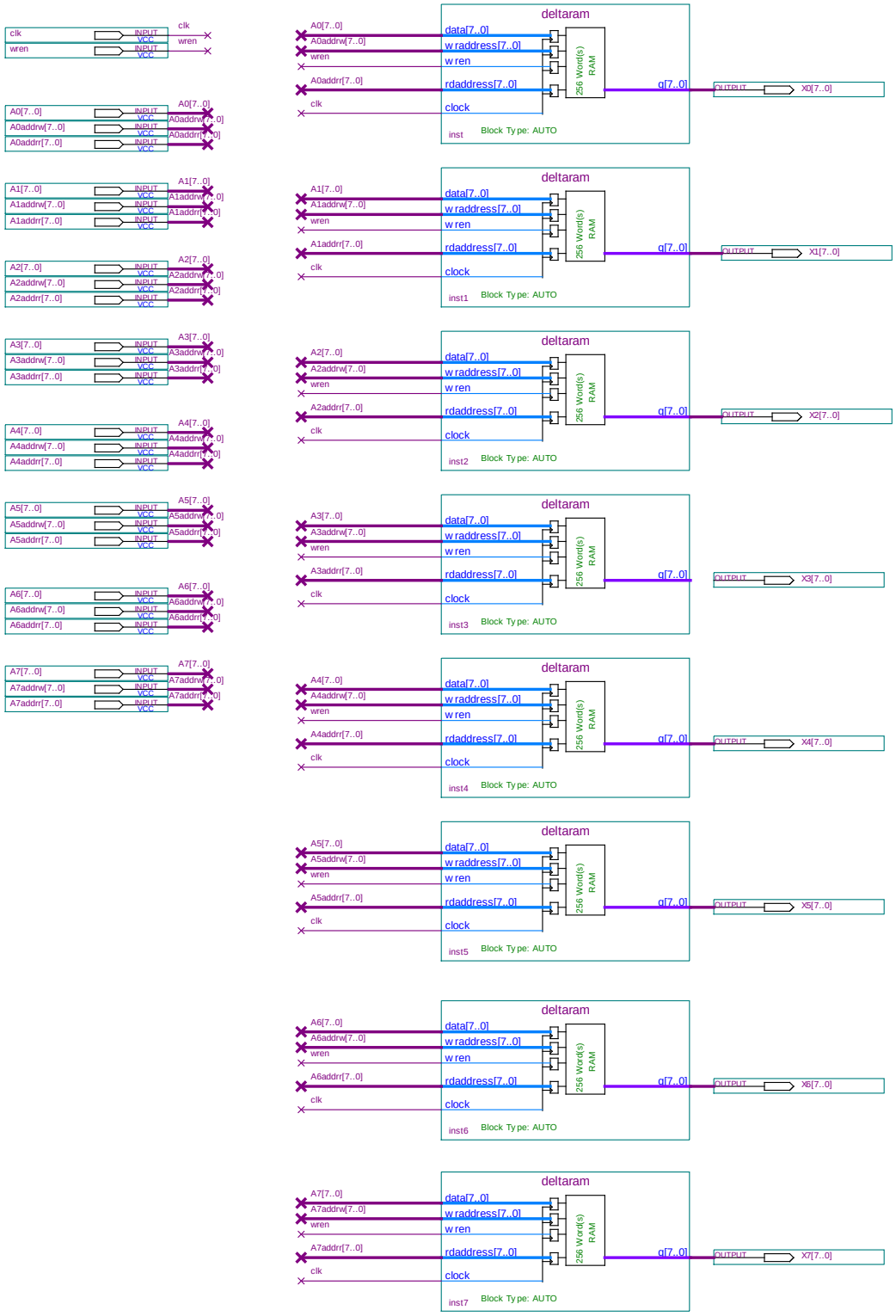


Figura 6.10: Diagrama bloc a bancilor intercalate

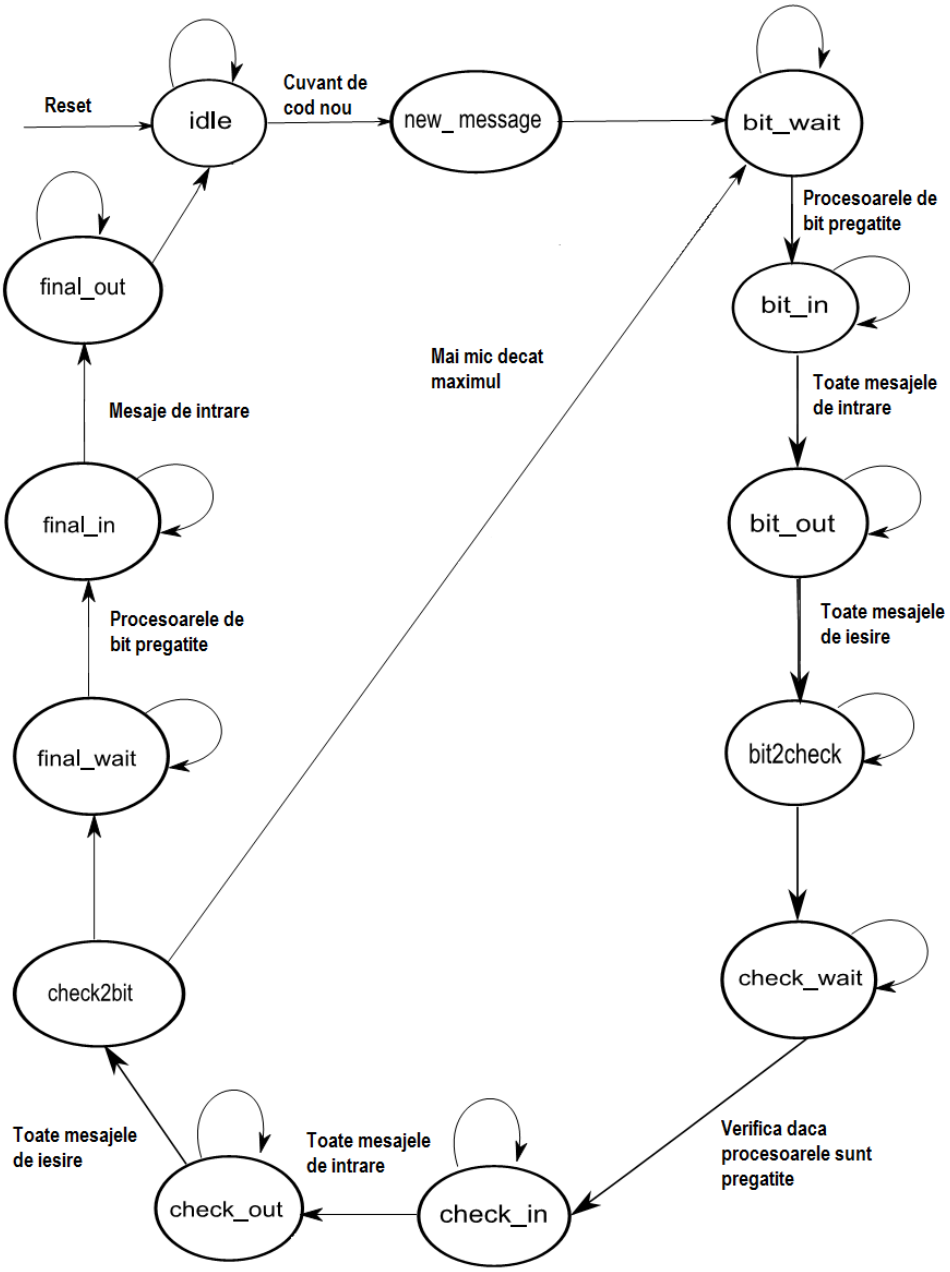


Figura 6.11: Unitatea de control a masinii de stare

Capitolul 7

7 Rezultate

Implementarea hardware a decodorului a avut ca scop compararea rezultatelor cu simulările Matlab. Banca de test Matlab a fost folosit pentru a simula transmisia BPSK (multiplexare binară a fazelor) peste un canal AWGN (zgomot aditiv alb gaussian) folosind o legătură în serie RS-232. Un cuvânt de cod decodat este comparat între decodorul implementat și banca de test. În cele ce urmează sunt măsurate resursele folosite pentru implementare.

7.1 Resurse utilizate pentru pentru implementarea proiectului

Decodorul a fost implementat inițial pe o placă Altera Cyclone I (EP1C6Q240C8) dar pe parcurs s-au dovedit insuficiente resursele hardware ale acestei FPGA. Pentru continuarea proiectului s-a decis testarea pe un model mai nou de la Altera, și anume Cyclone II (EP2C35F672C6N) DE2, aceasta având resursele necesare. Utilizarea resurselor în ceea ce privește tabelele de căutare, este arătată în tabelul 7.1.

După cum se poate observa, procesoarele nodului de verificare domină utilizarea resurselor. Tabelele de căutare ψ(x) sunt responsabile pentru acest lucru, ele având 100 de elemente logice, fiecare fiind de instantiat de 9 ori pentru fiecare procesor al nodului de verificare.

	Celule logice (%)	Total
Permutarea mesajelor bloc	4.7	556
Procesoarele nodului bit	20.6	2240
Procesoarele nodului de verificare	73.4	8648
Unitatea de control	1.3	136

Tabelul 7.1: Resurse utilizate pentru proiect

În timp ce procesoarele nodului de verificare ocupă cea mai pare parte a proiectului, ele nu sunt exagerat de complexe, putându-se mări numărul de procesoare paralele la 32. Această implementare ar putea fi realizată pe o placă Altera Stratix II (EP2S60F672C5E2). Decodorul actual ocupă aproximativ 31% din resursele disponibile.

Atunci când numărul de procesoare paralele este dublat, codul maxim suportat este de 60 de ori mai mare, dar și complexitatea conceptului crește de două ori. Complexitatea va crește liniar cu numărul de noduri de procesare paralelă, prin urmare creste și scalabilitatea decodorului.

7.2 Simularea MATLAB

Banca de test Matlab a fost folosită pentru a simula transmisia BPSK pe un canal AWGN. Matricea 128 x 256 Regular (3,6) de verificare a parității a fost folosită pe decodor și simulată pentru 10 iterații (max 30 cw errors) cu lungimea cuvântului decodat egal cu 256.

Simulari BER şi FER pentru diferite valori ale SNR-ului.

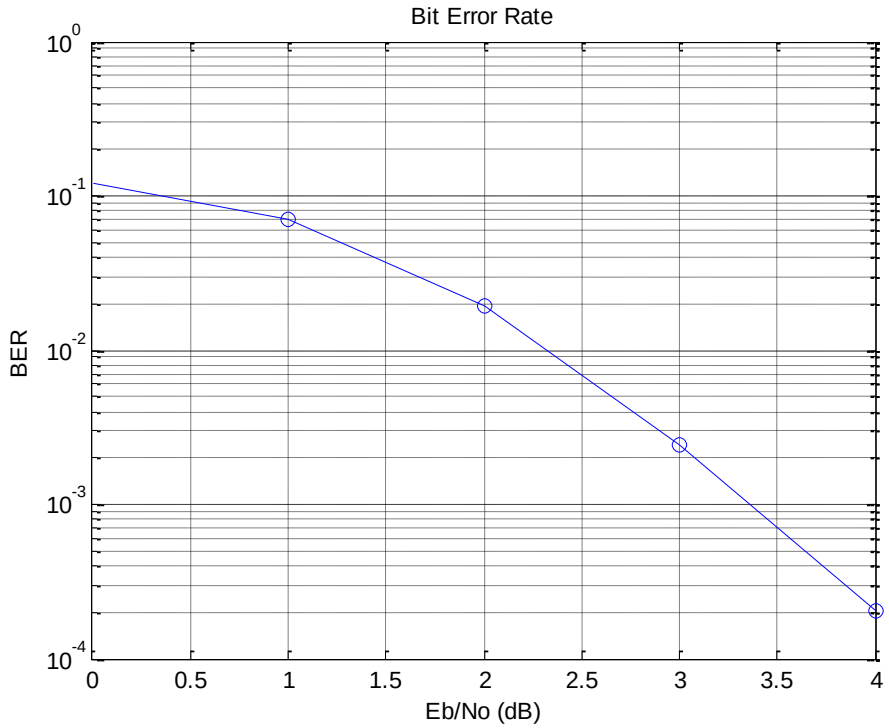


Figura 7.1: Reprezentarea grafica BER pentru SNR [0:1:4]

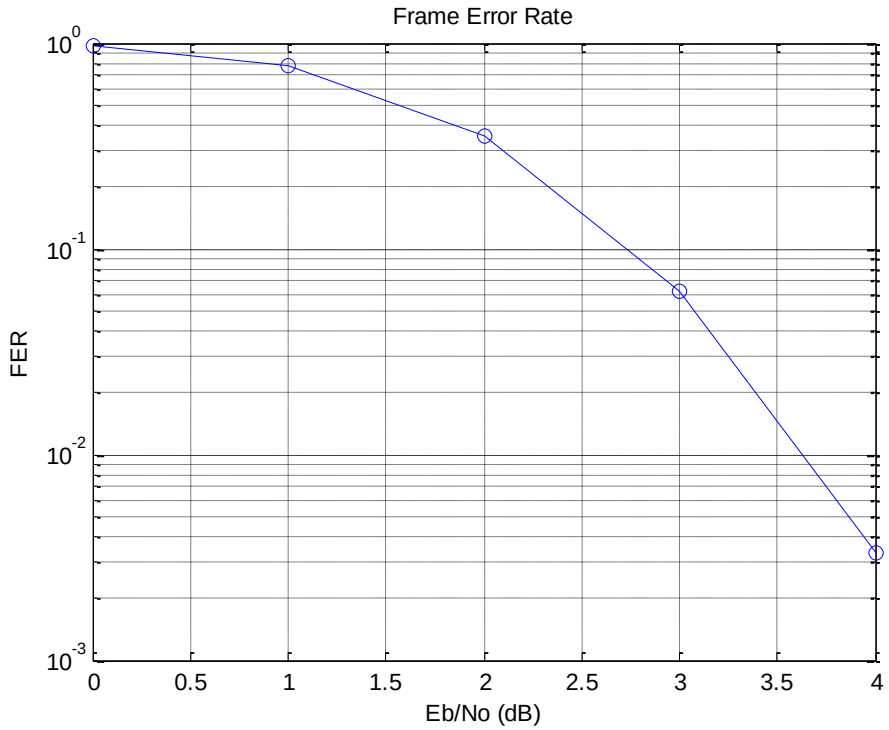


Figura 7.1: Reprezentarea grafica FER pentru SNR [0:1:4]

Timpul necesar a fost de 32.06 secunde.

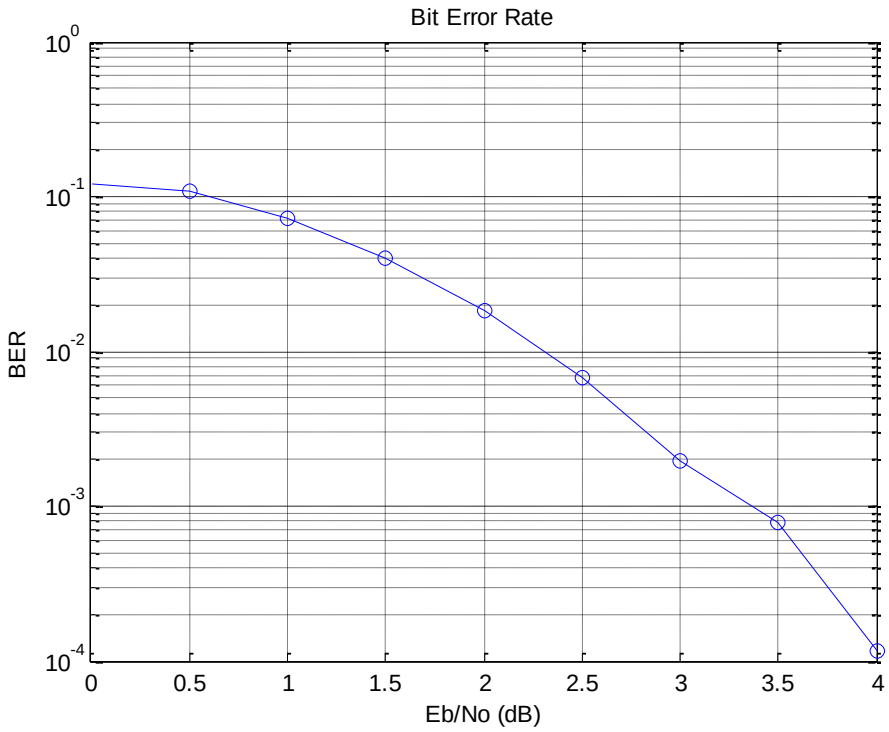


Figura 7.1: Reprezentarea grafica BER pentru SNR [0:0.5:4]

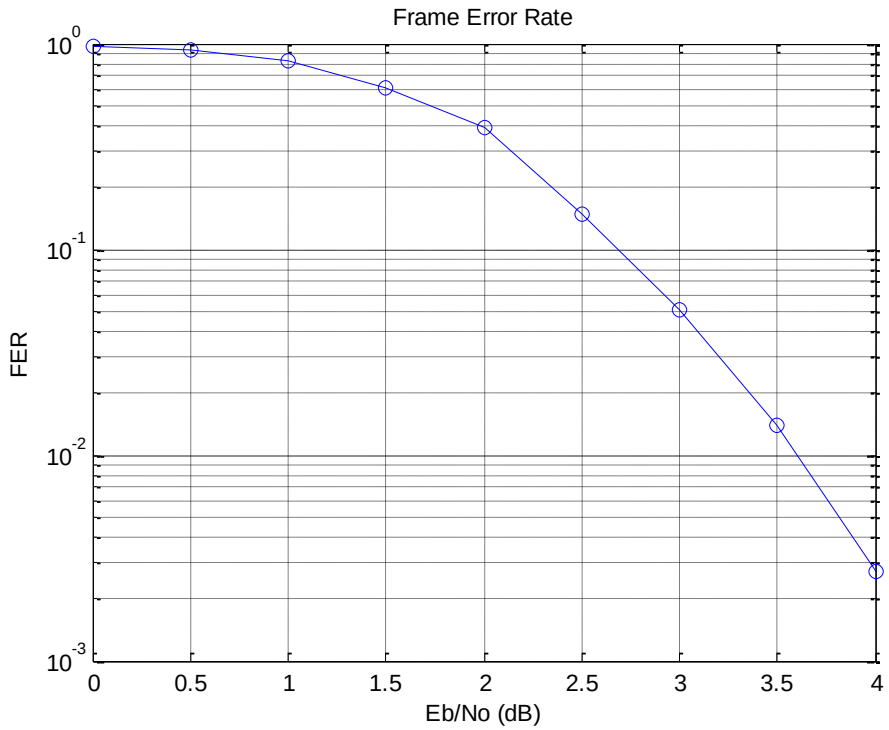


Figura 7.1: Reprezentarea grafica FER pentru SNR [0:0.5:4]

Timpul necesar a fost de 45.66 secunde.

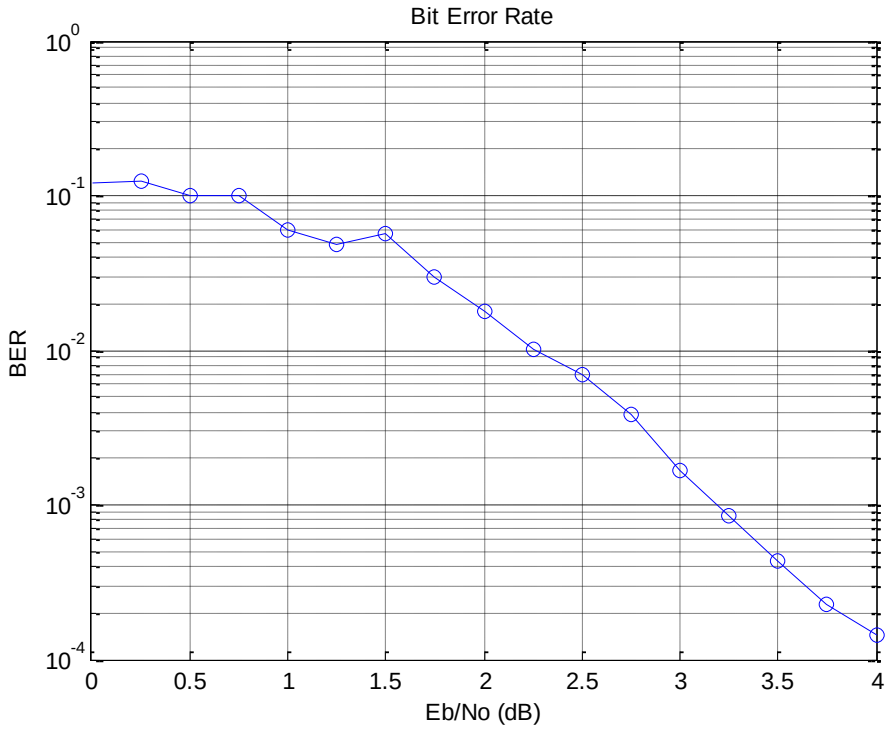


Figura 7.1: Reprezentarea grafica BER pentru SNR [0:0.25:4]

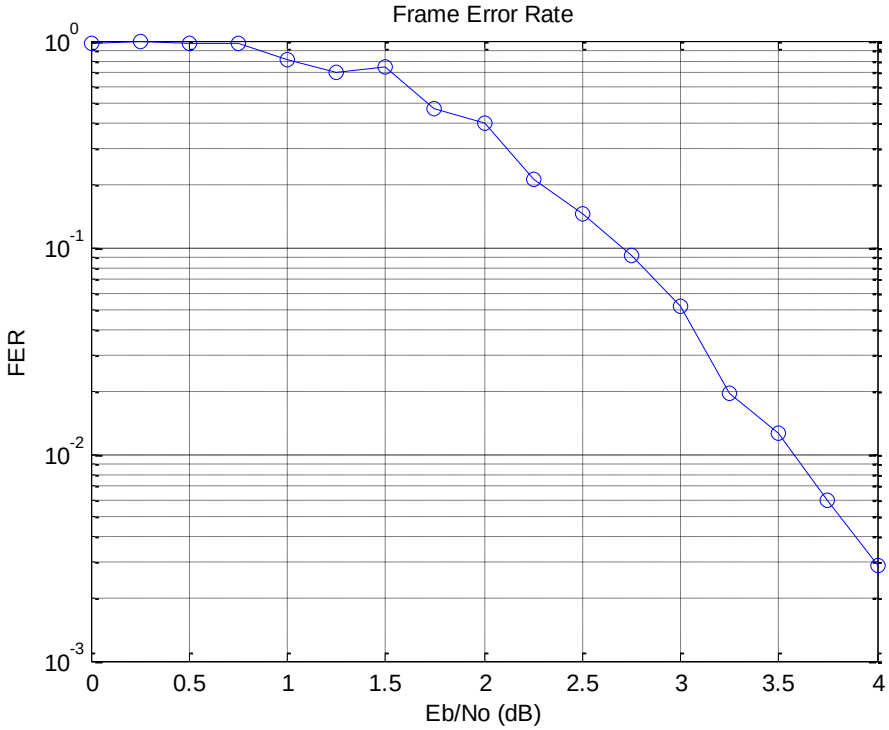


Figura 7.1: Reprezentarea grafica BER pentru SNR [0:0.25:4]

Timpul necesar a fost de 80.06 secunde.

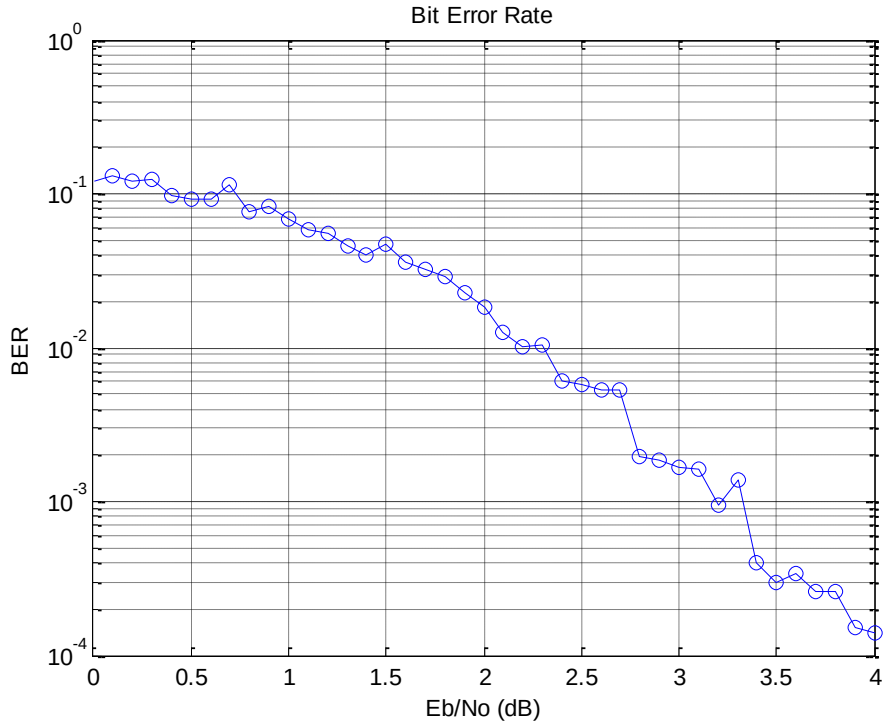


Figura 7.1: Reprezentarea grafica BER pentru SNR [0:0.1:4]

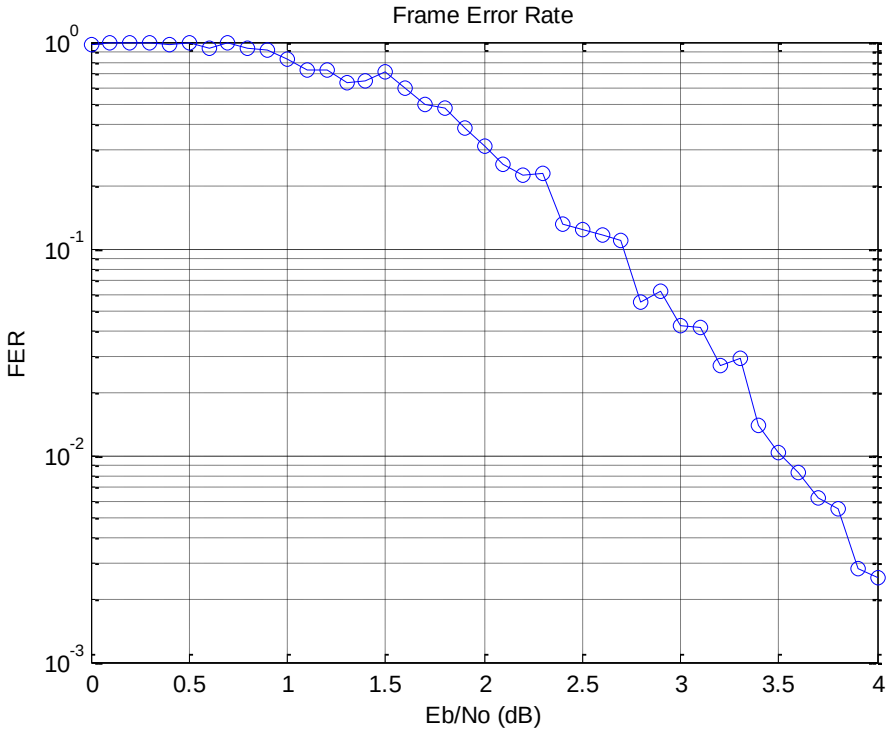


Figura 7.1: Reprezentarea grafica FER pentru SNR [0:0.1:4]

Timpul necesar a fost de 187.65 secunde.

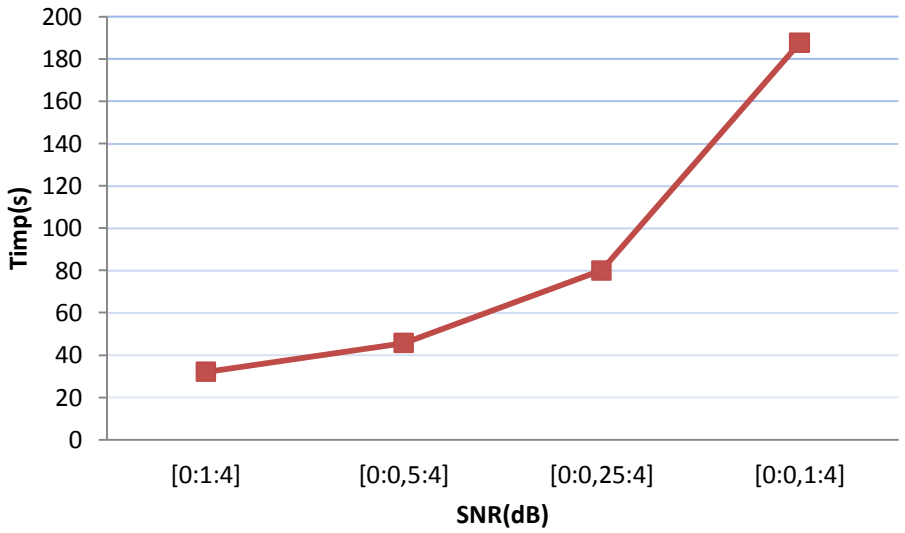


Figura 7.1: SNR si timpul necesar decodarii

7.3 Banca de test Matlab

Banca de test Matlab a fost folosită pentru a simula transmisia BPSK pe un canal AWGN. Matricea 480 x 240 regulată (3,6) de verificare a parității a fost folosită pe decodor și simulată pentru 10 iterații.

Raportul semnal-zgomot (SNR) a fost variat de la 0 la 10 dB, în creștere creștere cu 1dB, cu 200,000 cuvinte de cod transmise pe punctul SNR. Acesta dă 1,600,000 biți de cod pe SNR și 17,600,000 biți de cod pe pe întreaga simulare. Fiecare decodor funcționează pentru 10 iterati.

Sub 6dB SNR, decodorul hardware ar trebui sa se afle în completă concordanță cu simularea matlab în virgulă mobilă și punct fix. Decodorul hardware are o precizie superioară în reprezentarea virgulei mobile.

La peste 7dB SNR decodorul hardware și simularea Matlab a punctului fix încep a diferi, deși în cel mai rău caz diferența este de mai puțin de 0.5dB. Motivul pentru discrepanță se datorează ordinii în care testul de banca Matlab efectuează operații de adunare.

Capitolul 8

8 Concluzii şi muncă adiţională

Având în vedere codul LDPC (3,6)-regulat, software-ul Matlab creat, definește un program, care configurează comutatoarele rețelei Benes și generează fișiere ROM pentru pentru a fi utilizate atunci când se descarcă pe placa de dezvoltare Cyclone DE2. Ce este necesar pentru a utiliza un nou cod pe decodorul LDPC este de a informa Quartus II că fișierele de definiție ROM sunt modificate și să le descărcați iar pe placă, nefiind necesară altă configurație.

Un nou pas în proiectarea unui decodor LDPC ar fi creșterea numărului de unități de procesare paralele la 32. Acest lucru o să-i ofere o viteză mărită de 400% în comparație cu proiectul actual. Proiectul nu va mai potrivit pentru placa Cyclone II DE2, așa că următorul pas este de a îl transferă pe o placă Stratix II (EP2S60F672C5E2). Odată cu transferarea pe noua placă va crește și output-ul decodorului, având nevoie de o interfață mai rapidă input-output. Scopul final este acela de a implementa o interfață ethernet între FPGA și PC.

Bibliografie

[1] Alberto Tarable, „Mapping Interleaving Laws to Parallel Turbo and LDPC Decoder Architectures”, *IEEE TRANSACTIONS ON INFORMATION THEORY*, VOL. 50, NO. 9, SEPTEMBER 2004.

[2] C. E. SHANNON, „A Mathematical Theory of Communication”, *The Bell System Technical Journal*, Vol. 27, pp. 379–423, 623–656, July, October, 1948.

[3] Sae-Young Chung, „On the Design of Low-Density Parity-Check Codes

within 0.0045 dB of the Shannon Limit”, *IEEE COMMUNICATIONS LETTERS*, VOL. 5, NO. 2, FEBRUARY 2001.

[4] R. G. GALLAGER, „Low-Density Parity-Check Codes”, *IRE TRANSACTIONS ON INFORMATION THEORY*,pp.21-28,Jan 1962.

[5] Sarah J. Johnson, „Introducing Low-Density Parity-Check Codes”, 2006.

[6] Thomas J. Richardson and Rüdiger L. Urbanke, „The Capacity of Low-Density Parity-Check Codes Under Message-Passing Decoding”, *IEEE TRANSACTIONS ON INFORMATION THEORY*, VOL. 47, NO. 2, FEBRUARY 2001.

[7] Guido Masera, „Implementation of a Flexible LDPC Decoder”, *IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS*—II: EXPRESS BRIEFS, VOL. 54, NO. 6, JUNE 2007

[8] Chris Winstead, „Low-Voltage CMOS Circuits for Analog Iterative Decoders”, *IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS—I: REGULAR PAPERS*, VOL. 53, NO. 4, APRIL 2006.

[9] Chhay Kong, „Analog Iterative LDPC Decoder Based on Margin Propagation”, *IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS—II: EXPRESS BRIEFS*, VOL. 54, NO. 12, DECEMBER 2007.

[10] Saied Hemati, „Dynamics and Performance Analysis of Analog Iterative Decoding for

Low-Density Parity-Check (LDPC) Codes”

[11] Kun Guo, „A Parallel-Layered Belief-Propagation Decoder for Non-layered LDPC Codes”, *JOURNAL OF COMMUNICATIONS*, VOL. 5, NO. 5, MAY 2010.

[12] Luca Fanucci, „Design of a fully-parallel high-throughput de-coder for turbo gallager codes", *IEICE Transactions on Fundamentals of Electronics*, VOL.E89-A, NR.7, JULY 2006.

[13] Lili Zhoum, „CASCADE: A Standard Supercell Design Methodology With Congestion-Driven Placement for Three-Dimensional Interconnect-Heavy Very Large-Scale Integrated Circuits", *IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS*, VOL. 26, NO. 7, JULY 2007

[14] VIJAY NAGARAJAN, „High-throughput VLSI Implementations of Iterative Decoders and Related Code Construction Problems*", Journal of VLSI Signal Processing 49*, 185–206, 2007

[15] G. Masera, „Finite precision implementation of LDPC decoders", *Communications IEEE Proceedings*, vol. 152, no. 6, pp. 1098-1102, 9 Dec. 2005.

[16] Jong-Yeol Lee, „A 1-Gb/s Flexible LDPC Decoder Supporting Multiple Code Rates and Block Lengths", *Consumer Electronics, IEEE Transactions on*, vol. 54,pp. 417-424, May 2008.

[17] Sarah J. Johnson, „Resolvable 2-Designs for Regular Low-Density Parity-Check Codes", *IEEE TRANSACTIONS ON COMMUNICATIONS*, VOL. 51, NO. 9, SEPTEMBER 2003

[18] Mohammad M. Mansour, „High-Throughput LDPC Decoders", *IEEE TRANSACTIONS ON VERY LARGE SCALE INTEGRATION SYSTEMS*, VOL. 11, NO. 6, DECEMBER 2003

[19] V. E. Benes, „Permutation groups, complexes and rearrangeable connecting network"

Bell System Technical Journal, vol. 43, no. 4, pp. 1619-1640, 1964.

[20] ABRAHAM WAKSMA, „A Permutation Network", J. ACM, vol. 15, no. 1,

pp. 159-163, 1968.

[21] „METIS serial graph partitioning and fill reducing matrix ordering"

http://glaros.dtc.umn.edu/gkhome/metis/metis/overview

[22] R. Andraka, „A survey of cordic algorithms for fpga based computers",

Proceedings of the 1998 ACM/SIGDA sixth international symposium on Field

programmable gate arrays, pp. 191-200, 1998.

[23] Federico Quaglio, „Low Complexity, Flexible LDPC Decoders", *14th*

IST Mobile & Wireless Communications Summit, 2005.

[24] „Digital Design and Synthesis Lecture 2 Slides, Concordia University Canada"

http://users.encs.concordia.ca/»asim/COEN 6501/Lecture Notes/Lecture Notes.htm

[25] „Implementation of a digital UART by VHDL

http://www.ee.ualberta.ca/~elliott/ee552/studentAppNotes/1999f/UART/uart.html

Anexa A

Listarea codului MATLAB

partNmap.m

```
% Acest fisier accepta o matrice de verificare a paritatii
% Scopul sau final este acela de a genera
% blocuri de configuratie ROM pentru un cod dat, folosit de Altera's
% Quartus II in dispozitivul de programare.
% Acesta converteste matricea de verificare a paritãtii într-un grafic
% Tanner, trece prin softaware-ul matricei de partitionare, verifica daca
% partitiile sunt egale atat pe nodul bit cat si pe noldul de verificarea de
% jumatate de iteratie, urmatã de gãsirea functiei de programare.
% Urmatorul pas este de a genera setarile de comutare ale retelei Benes,
% urmat de generararea fisierelor ROM si si scrierea lor pe disc.


function partNmap(H,parts,fn)
matrix2Graph(H,fn); % Converteste matricea H intr-un grafic Tanner, salvarea
% pe disc a celulelor METIS pentru partitionarea graficului
str = ['metis\pmetis ',fn, ' ',num2str(parts)];
system(str);
% incarca partitionara
[p1,p2,Vn,Cn] = plist2part(H,[fn,'.part.',num2str(parts)],parts);
% creaza o functie de programare pentru comutatorul crossbar si
% a bancilor intercalate
[b1,b2,delta] = mapping(H,p1,p2,parts);
% genereaza setarile de comutare ale retelei Benes, inclusiv iesirea
[sw1a,sw2a] = generateSwitches(parts,b1,b2,Vn);
% genereaza setarile pentru intercalare, inclusiv iesirea
delta_f = generateDelta(H,delta);
print_files(H,Vn,Cn,delta_f,sw1a,sw2a);
```

A.1 Matricea graficului Tanner

matrix2Graph.m

```
% acest fisier accepta o matrice de verificare a paritatii si ouput-urile
% reprezentarii ei,grafice,corespunzatoare, cu prima linie continand
% numarul de noduri si muchii, iar urmãtoarele contin muchiile conectate
% la nodul n (linia n-1).
% Nodurile bit apar inaintea nodurilor de verificare.

function H = matrix2Graph(H,fn)
[M,N] = size(H);
[I,J]=ind2sub([M,N],find(H)); % cauta prima pozitie
edges=size(I,1);
F = fopen(fn,'w');
% printeaza numarul de noduri si muchii
fprintf(F,'%d %d\n',(M+N),edges);
```

```
% printeaza nodurile de bit
for i=1:N
fprintf(F,'%d ',N+I(find(J==i))');
fprintf(F,'\n');
end
% printeaza nodurile de verificare
for i=1:M
fprintf(F,'%d ',J(find(I==i))');
fprintf(F,'\n');
end
fclose(F);
```

A.2 Partitionarea graficului

plist2part.m

```
% Acest fisier converteste output-ul din graficul partitionat METIS si
% produce 2 seturi P1 si P2 care contin asignarea nodurilor bit si de
% verificare pentru procesoare, in timpul jumatatii de iteratie bit si
% de verificare.
% De asemenea, se încearcã a echilibreze partitiile, dand o eroare atunci
% când nu se poate.


function [P1,P2,Vn,Cn] = plist2part(H,fn,partitions)
rand('twister',5489) % repetarea generarii de numere aleatoare
edges=sum(sum(H));
M=ceil(edges/partitions);
% incarca partitia
partition = load(fn);
 [Mh,Nh] = size(H);
[I,J] = find(H);
% creeaza P1 (prima jumatate a iteratiei bit)
for z=1:1000
plist = zeros(partitions,2*M);
for i=1:Nh
ind = find(plist(partition(i),:)==0,1);
edge = find(J==i)';
plist(partition(i),ind:ind+size(edge,2)-1)=edge;
end
% verifica care partitii sunt în curs sau depasite
overfull = find(plist(:,M+1)~=0);
underfull = find(plist(:,M)==0);
% in cazul în care nici una nu este depasita, atunci am terminat
if(size(overfull,1)==0)
break;
end
% muta la întamplare un nod dintr-o partitie depasita la una nedepasita
over = overfull(ceil(rand()*size(overfull,1)));
under = underfull(ceil(rand()*size(underfull,1)));
lst = find(partition(1:Nh)==over);
partition(lst(ceil(rand()*size(lst,1))))=under;
end
% acum ne mutam la verificarea jumatati de iteratie
Vn = partition(1:Nh);
P1=plist(:,1:M);
```

```
if size(overfull,1)~=0
error('unbalanced','unbalanced');
end
partition = partition(Nh+1:Nh+Mh);
for z=1:1000
plist = zeros(partitions,2*M);
for i=1:Mh
ind = find(plist(partition(i),:)==0,1);
edge = find(I==i)';
plist(partition(i),ind:ind+size(edge,2)-1)=edge;
end
% verifica care partitii sunt în curs sau complete
overfull = find(plist(:,M+1)~=0);
underfull = find(plist(:,M)==0);
% in cazul în care nici una nu este depasita, atunci am terminat
if(size(overfull,1)==0)
break;
end
% muta la întamplare un nod dintr-o partitie depasita la una nedepasita
over = overfull(ceil(rand()*size(overfull,1)));
under = underfull(ceil(rand()*size(underfull,1)));
lst = find(partition(1:Mh)==over);
partition(lst(ceil(rand()*size(lst,1))))=under;
end
% face verificarea jumatati de iteratie
Cn = partition(1:Mh);
if size(overfull,1)~=0
error('unbalanced','unbalanced');
end
P2=plist(:,1:M);
% introduce marginile fictive daca M/L nu este un numar intreg
if size(underfull)~=0
under1 = find(P1==0);
under2 = find(P2==0);
for i = 1:size(under2)
P1(under1(i)) = i+edges;
P2(under2(i)) = i+edges;
end
end
```

A.3 Programarea

mapping.m

```
% Acest fisier produce o functie de mapare definind programul decodorului.
% Functia de mapare este apoi folosita pentru a determina configuratia
% comutatoarelor crossbar si bancilor intercalate.

% p - numãrul de procesoare paralele
% p1 - setul care contine nodurile asociate cu procesoarele bit P in
% jumatatea de iteratie a nodului bit
% p2- setul care contine nodurile asociate cu procesoarele P in jumatatea
% de iteratie a nodului de verificare
% b1, b2 – configuratia comutatoarelor crossbar de stanga si dreapta
% delta- configuratie bancilor intercalate
```

```
function [b1,b2,delta] = mapping(p1,p2,P)
% definirea unor constante
L=size(p1,1)*size(p1,2);
M=L/P;
% algoritmul functioneaza daca p1 este intr-o ordine numerica , asa ca
% rearanjam muchiile, numarandu-le pentru a se asigura de acest lucru
p11=zeros(P,M);
p21=zeros(P,M);
p22=p2';
for i=1:L
p11(ceil(i/M),myMod(i,M)) = i;
p21(ceil(i/M),myMod(i,M)) = find(p1'==p22(i));
end
p1 = p11;
p2 = p21;
% creaza permutarea pi
pi = zeros(1,size(p2,2)*size(p2,1));
for i = 1:P
pi((i-1)*M+1:i*M)=p2(i,:);
end
% creaza functia de mapare si variabila L
Map = zeros(1,L);
Lv = zeros(P,M);
Lv1= Lv;
for i = 1:P
Lv1(i,:)=i;
end
Lv2 = Lv1;
Lv = Lv1;
% pentru fiecare margine
for j=1:L
J=j;
jv1=myMod(find(p1'==J),M);
jv2=myMod(find(p2'==J),M);
a = nonzeros(intersect(Lv1(:,jv1),Lv2(:,jv2)));
if size(a,1)>0
% avem o atribuire fara coliziuni
Map(j) = a(1);
Lv1(a(1),jv1)=0;
Lv2(a(1),jv2)=0;
else
% nici o atribuire fara coliziuni, cream coliziune
m = nonzeros(Lv1(:,jv1));
n = nonzeros(Lv2(:,jv2));
m = m(1);
n = n(1);
Map(J) = m;
% acum, incercam sa o remediem
while 1
a = p2(find(p2(:,jv2)~=J),jv2); % lista de elemente în V', care ar putea
%avea nevoie de actualizare
found = false;
for i=1:length(a)
if (Map(a(i))==m)
found=true;
J=a(i);
```

```
Map(J) = n;
jv1=myMod(find(p1'==J),M);
end
end
% am remediat, dar ar putea fi cauzata alta
if found==true
a = p1(find(p1(:,jv1)~=J),jv1); % Lista de elemente în V
for i=1:length(a)
if (Map(a(i))==n)
found = true;
J=a(i);
Map(J) = m;
jv2=myMod(find(p2'==J),M);
end
end
if found==false
% fara coliziuni
break
end
else break % fara coliziuni
end
end
% variabile actualizate
Lv1 = Lv;
Lv2 = Lv;
for i=1:j
Lv1(Map(i),myMod(find(p1'==i),M)) = 0;
Lv2(Map(i),myMod(find(p2'==i),M)) = 0;
end
end
end
% am terminat cu maparea, cream b1,b2 si delta
b1 = zeros(P,M);
b2 = zeros(P,M);
delta = zeros(P,M);
piinv = zeros(1,L);
% cream permutarea inversa piinv
for i=1:L
piinv(pi(i))=i;
end
for i = 1:P
a = find(Map==i);
for j = 1:M
delta(i,myMod(piinv(a(j)),M))= myMod(a(j),M);
b1(i,j) = Map((i-1)*M + j);
b2(i,j) = Map(pi((i-1)*M + j));
end
end
```

A.4 Reteaua Benes

benes-main.m

```
% Aceasta functie configureaza comutatoarele pentru reteaua Benes data
% de permutare piT.
% Comutatoarele sunt convertite in format hexadecimal pentru
% printarea fisierelor.

function sw = benes_main(piT)
global s;
global N;
clear sw;
N = 4;
piT = [16,15,14,13,12,11,10,9,8,7,6,5,4,3,2,1];
s = zeros(1,N*log2(N)-N+1);
benes(piT,N,1);
for i = 1:(N*log2(N)-N)/4
sw(i) = dec2hex(bi2de(s((i-1)*4+1:i*4)));
end
sw((N*log2(N)-N)/4+1) = dec2hex(bi2de(s(N*log2(N)-N+1)));
sw = fliplr(sw)
```

benes.m

```
% Acesta stabileste recursiv comutatoarele asociate cu reteaua Benes,
% se acceptã o permutare, o dimensiune si un parametru d, care se refera
% la nivelul de sub-bloc din partea de sus de exemplu
% Nivelul este folosit pentru a determina care comutator seteaza în
% vectorul de comutare.

function b = benes(newPi,n,d)
global P;
global pi;
global piinv;
global s
global N
% Formule pentru pozitii ale comutatorului pentru partea de intrare
% si de iesire
switch_start = (N/2)*log2(N/n)+ (d-1)*n/2;
switch_end = (N/2)*log2(N)+(N/2)*(log2(n)-3+4/n)+(d-1)*(n/2-1)-1;
pi = newPi;
piinv = inverse(pi);
% acesta este cazul de terminarein care am gasit pozitia elementara
if (n==2)
s(switch_start+1) = pi(1)-1;
else
P = zeros(n,n);
% 1 - Partitia A, -1 Partitia B
% umple matricea
while (sum(sum(abs(P)))<n)
a = find(sum(P,1)==0); %gaseste toate intrarile care nu au fost umplute
a = a(1); % o ia pe prima
P(piinv(a),a) = 1;
forward(piinv(a),-1);
```

```
end
% seteaza comutatoarele
[a,b] = ind2sub([n,n],find(P'==1));
for i=1:n/2
s(switch_start+i) = abs(rem(b(i),2)-1);
end
[a,b] = ind2sub([n,n],find(P==1));
for i=2:n/2
s(switch_end+i) = abs(rem(b(i),2)-1);
end
% gaseste permutari pentru sub-blocuri si cauta recursiv
Pa = inverse(ceil(a/2)');
[a,b] = ind2sub([n,n],find(P==-1));
Pb = inverse(ceil(a/2)');
benes(Pa,n/2,d*2-1);
benes(Pb,n/2,d*2);
end;
b = 0;
```

forward.m

```
% seteaza pozitia comutatoarelor pe partea de intrare si daca apelurile
% sunt goale, backward.m seteaza comutatorul corespunzător pe partea
% de iesire

function A = forward(indx,entry)
global P;
global pi;
if (rem(indx,2))
x = indx+1;
else
x = indx-1;
end
if (P(x,pi(x)) ==0)
P(x,pi(x)) = entry;
backward(pi(x),entry*-1);
end
```

backward.m

```
% Seturi de comutatoare pe partea de iesire si daca apelurile sunt goale
% la forward.m seteaza comutatorul corespunzător pe partea de intrare

function A = backward(indx,entry)
global P;
global piinv;
if (rem(indx,2))
x = indx+1;
else
x = indx-1;
end
if (P(piinv(x),x) ==0)
P(piinv(x),x) = entry;
forward(piinv(x),entry*-1);
end
```

inverse.m

```
% Găseste inversa unei functii pi(x), se presupune ca pi(x) are o
% singura valoare.

function inv = inverse(pi)
inv = zeros(1,size(pi,2));
for i = 1:size(pi,2)
inv(pi(i))=i;
end
```

A.5 Definirea fisierelor ROM

print-files.m

```
% imprima variabilele in fisierele ROM

function print_files(H,Vn,Cn,delta_f,sw1a,sw2a)
a = sum(H,1);
% imprima gradul pentru fiecare variabila si verifica nodurile asociate
% pentru fiecare procesor
for i=1:N
print_ram_dec(['variable_deg',num2str(i),'.mif'],a(find(Vn==i)),2,64);
end
a = sum(H,2);
for i=1:N
print_ram_dec(['check_deg',num2str(i),'.mif'],a(find(Cn==i)),8,32);
end
% printeaza bancile de intercalare si comutatoarele
print_ram_hex('delta1.mif',delta_f,64,1024)
print_ram_hex('sw1.mif',sw1a,17,1024);
print_ram_hex('sw2.mif',sw2a,17,1024);
% cauta si printeaza fiecare procesor
lambda_rom = zeros(1,size(H,2));
for i=1:size(H,2)
lambda_rom(i) = (Vn(i)-1)*2^6+find((find(Vn==(Vn(i))))'==i)-1;
end
print_ram_dec('lambda_rom.mif',lambda_rom',9,512)
```

print-ram-dec.m

```
% Scrie variabilele in fisierul ROM, valorile sunt în zecimal.

function print_ram_dec(filename,values,width,depth)
F = fopen(filename,'w');
fprintf(F,'WIDTH = %d;\n',width);
fprintf(F,'DEPTH = %d;\n',depth);
fprintf(F,'ADDRESS_RADIX = UNS;\n');
fprintf(F,'DATA_RADIX = UNS;\n');
fprintf(F,'CONTENT BEGIN\n');
for i = 1 : size(values)
fprintf(F,'\t%d : %d;\n',i-1,values(i));
end
for i=size(values,1):depth-1
fprintf(F,'\t%d : 0;\n',i);
```

```
end
fprintf(F,'END;');
```

print-ram-hex.m

```
%  Scrie variabilele in fisierul ROM, valorile sunt în hexazecimal.

function print_ram_hex(filename,values,width,depth)
F = fopen(filename,'w');
fprintf(F,'WIDTH = %d;\n',width);
fprintf(F,'DEPTH = %d;\n',depth);
fprintf(F,'ADDRESS_RADIX = UNS;\n');
fprintf(F,'DATA_RADIX = HEX;\n');
fprintf(F,'CONTENT BEGIN\n');
for i = 1 : size(values,1)
fprintf(F,'\t%d : %s;\n',i-1,values(i,:));
end
for i=size(values,1):depth-1
fprintf(F,'\t%d : 0;\n',i);
end
fprintf(F,'END;');
```

A.6 Banca de test

LDPCSimF.m

```
% Calculează numărul de erori pentru LDPC, produse de decodare folosind
% phi LUT
% Matrice H
% SNR
% numarul de iteratii ale decodorului
% masurarea canalului Y

function [numerr] = LDPCSimF(H,EbNo,niter,Y)
% Y trimite cuvintele de cod in pretenta AWGN
% si calculeaza numarul de erori de bit pentru fiecare iteratie
% initializare aici
[I,J]=ind2sub(size(H),find(H ~=-0));
[M,N]=size(H);
L=zeros(1,N);
% PHI LUT
phi =
[127,79,47,40,32,30,28,26,24,22,20,18,16,15,14,13,12,12,11,11,10,10,9,9,8,8,7,7,5,
5,3,2,2,1];
E=zeros(M,N);
Col=sortrows([I,J]);
Row=[I,J];
% Y=[-0.1,0.5,-0.8,1,-0.7,0.5];
% Y=[-0.63,-0.83,-0.73,-0.04,0.1,0.95,-0.76,0.66,-0.55,0.58];
% niter=3;
% Y = Y + sig*randn(1,N); % trimite toate 0 cu AWGN
% calcule
r=EbNo*4*Y/2; % rata
r = round(r*16)/16;
r(r>7.9375) = 7.9375;
r(r<-7.9375) = -7.9375;
```

```
% r= [2,1.875,1.75,2.1875,2.375,-.3125,2.1875,1.875];
% incarca r;
Ma=zeros(M,N);
for i = 1:length(J)
Ma(I(i),J(i))=r(J(i));
end
numerr = zeros(1,niter);
incorrect = 0;
dv=sum(H,1);
dc=sum(H,2);
for i=1:niter
index=0;
% verifica mesajele
for j=1:M % peste randuri
for k = 1:dc(j)
E(j,Col(index+k,2)) = 0;
sg = 1;
for l = 1:dc(j)
if (k ~= l)
E(j,Col(index+k,2)) = E(j,Col(index+k,2)) + phi(abs(16*Ma(j,Col(index+k,2))));
a = sign(Ma(j,Col(index+l,2)));
if (a == 0)
a=1;
end
sg = sg*a;
if (E(j,Col(index+k,2)) > 127)
E(j,Col(index+k,2)) = 127;
end
end
end
E(j,Col(index+k,2)) = phi(E(j,Col(index+k,2))+1)*sg/16;
end
index = index + dc(j);
end
L=r+sum(E,1);
L(L==0) = 1;
z = -sign(L)/2 +0.5;
numerr(1,i) = sum(z);
%syndrome=mod(H*z',2)';
if (sum(z)==0)
% pauza
end
index = 0;
for j=1:N % coloanele de jos
for k = 1:dv(j) % pentru fiecare mesaj
Ma(Row(index+k,1),j) = r(j);
for l = 1:dv(j) % suma pe alte mesaje
if (k ~= l)
Ma(Row(index+k,1),j) = Ma(Row(index+k,1),j) +E(Row(index+l,1),j);
if (Ma(Row(index+k,1),j) > 7.9375)
Ma(Row(index+k,1),j) = 7.9375;
end
if (Ma(Row(index+k,1),j) < -7.9375)
Ma(Row(index+k,1),j) = -7.9375;
end
end
end
end
index = index + dv(j);
```

```
end
end
```

LDPCTest.m

```
% test LDPC
% Genereaza o matrice de verificare a paritãtii si trimite toate 0 în
% prezenta zgmomotului AWGN
 clear all;
 close all;
 clc;
global E Ma
% initializarea
randn('state',0);
niter_decoder = 10; % numarul de iterarii de decodare
n = 1e6; % numarul de iteratii de mesaje
dv=3;   %(3x6)
dc=6;
N=480;  %(482x240)
M=240;
%Setarile seriale
s = serial('com1');
s.BaudRate = 28800;
s.Parity = 'even';
s.FlowControl = 'none';
fopen(s);
%H = genH(N,M,dv,dc);
% incarca H;
EbNodB=linspace(2,10,9)
EbNo=10.^(EbNodB/10);
ber=zeros(length(EbNo),niter_decoder);
ber_mine = ber;
incorrect = 0;
NotInf=zeros(length(EbNo));
rate=1/2;
Eb=1;
a = zeros(1,8);
for i = 1:length(EbNo)
N0=Eb/(EbNo(i)*rate);
sig = sqrt(N0/2);
for j=1:n
Y = ones(1,length(N)) + sig*randn(1,N); %trimite toate 0 cu awgn
r=EbNo(i)*4*Y/2; % rata
% ne asiguram ca raza de actiune a canalului se încadreaza în
% gama de decodoare
r = round(r*16)/16;
r(r>7.9375) = 7.9375;
r(r<-7.9375) = -7.9375;
r = r*16;
% conversia la zecimal(+-)xxx.xxxx
for ij = 1:8
a(ij) = r(ij);
if (r(ij) <0)
a(ij) = 256 + a(ij);
end
end
fwrite(s,a);
% asteapta pana cand decodorul termina
```

```
while(s.BytesAvailable == 0)
end
result = fread(s,s.BytesAvailable);
t = 0;
% converteste raspunsul in binar
b = dec2bin(result(1),8);
for ij = 1:8
if b(ij) == '1'
t = t+1;
end
end
% comparare cu simularea
tmp=LDPCSimF(H,EbNo(i),niter_decoder,Y);
if (sum(isnan(tmp)) ==0)
ber(i,:)= ber(i,:) +tmp;
NotInf(i) = NotInf(i) +1;
end
ber_mine(i,:)= ber_mine(i,:) +t;
end
% ber(i,:)=ber(i,:)/(j*N);
ber_mine(i,10)=ber_mine(i,10)/(j*N)
i
end
```

Appendix B

Codul VHDL

B.1 ψ Lookup Table (LUT)

LUT.vhdl

```
library ieee;
use ieee.std_logic_1164.all;
USE ieee.numeric_std.all;
entity LUT2 is
port(
input : IN std_logic_vector(7 downto 0);
output : OUT std_logic_vector(6 downto 0));
end LUT2;
architecture behavior of LUT2 is
begin
process(input)
VARIABLE inp : INTEGER range -128 to 127;
begin
inp := TO_INTEGER(SIGNED(input));
case inp is
when -128 =>
output <= "0000000";
when -127 =>
output <= "0000000";
when -126 =>
output <= "0000000";
```

```
when -125=>
output <= "0000000";
when -124=>
output <= "0000000";
when -123=>
output <= "0000000";
when -122=>
output <= "0000000";
when -121=>
output <= "0000000";
when -120=>
output <= "0000000";
when -119 =>
output <= "0000000";
when -118 =>
output <= "0000000";
when -117 =>
output <= "0000000";
when -116 =>
output <= "0000000";
when -115 =>
output <= "0000000";
when -114 =>
output <= "0000000";
when -113 =>
output <= "0000000";
when -112 =>
output <= "0000000";
when -111 =>
output <= "0000000";
when -110 =>
output <= "0000000";
when -109 =>
output <= "0000000";
when -108 =>
output <= "0000000";
when -107 =>
output <= "0000000";
when -106 =>
output <= "0000000";
when -105 =>
output <= "0000000";
when -104 =>
output <= "0000000";
when -103 =>
output <= "0000000";
when -102 =>
output <= "0000000";
when -101 =>
output <= "0000000";
when -100 =>
output <= "0000000";
when -99 =>
output <= "0000000";
when -98 =>
output <= "0000000";
when -97 =>
output <= "0000001";
when -96=>
```

```
output <= "0000001";
when -95=>
output <= "0000001";
when -94=>
output <= "0000001";
when -93=>
output <= "0000001";
when -92=>
output <= "0000001";
when -91=>
output <= "0000001";
when -90=>
output <= "0000001";
when -89=>
output <= "0000001";
when -88=>
output <= "0000001";
when -87=>
output <= "0000001";
when -86=>
output <= "0000001";
when -85=>
output <= "0000001";
when -84 =>
output <= "0000001";
when -83 =>
output <= "0000001";
when -82 =>
output <= "0000001";
when -81 =>
output <= "0000001";
when -80 =>
output <= "0000001";
when -79 =>
output <= "0000001";
when -78 =>
output <= "0000001";
when -77 =>
output <= "0000001";
when -76 =>
output <= "0000001";
when -75 =>
output <= "0000001";
when -74 =>
output <= "0000001";
when -73 =>
output <= "0000001";
when -72 =>
output <= "0000001";
when -71 =>
output <= "0000001";
when -70 =>
output <= "0000001";
when -69 =>
output <= "0000001";
when -68 =>
output <= "0000001";
when -67 =>
output <= "0000001";
```

```
when -66 =>
output <= "0000001";
when -65 =>
output <= "0000001";
when -64 =>
output <= "0000001";
when -63 =>
output <= "0000001";
when -62 =>
output <= "0000001";
when -61 =>
output <= "0000001";
when -60 =>
output <= "0000001";
when -59 =>
output <= "0000001";
when -58 =>
output <= "0000001";
when -57 =>
output <= "0000001";
when -56 =>
output <= "0000010";
when -55 =>
output <= "0000010";
when -54 =>
output <= "0000010";
when -53 =>
output <= "0000010";
when -52 =>
output <= "0000010";
when -51 =>
output <= "0000010";
when -50 =>
output <= "0000010";
when -49 =>
output <= "0000010";
when -48 =>
output <= "0000010";
when -47 =>
output <= "0000010";
when -46 =>
output <= "0000010";
when -45 =>
output <= "0000010";
when -44 =>
output <= "0000011";
when -43 =>
output <= "0000011";
when -42 =>
output <= "0000011";
when -41 =>
output <= "0000011";
when -40 =>
output <= "0000011";
when -39 =>
output <= "0000011";
when -38 =>
output <= "0000011";
when -37 =>
```

```
output <= "0000011";
when -36 =>
output <= "0000100";
when -35 =>
output <= "0000100";
when -34 =>
output <= "0000100";
when -33 =>
output <= "0000100";
when -32 =>
output <= "0000100";
when -31 =>
output <= "0000101";
when -30 =>
output <= "0000101";
when -29 =>
output <= "0000110";
when -28 =>
output <= "0000110";
when -27 =>
output <= "0000111";
when -26 =>
output <= "0000111";
when -25 =>
output <= "0001000";
when -24 =>
output <= "0001000";
when -23 =>
output <= "0001001";
when -22 =>
output <= "0001001";
when -21 =>
output <= "0001010";
when -20 =>
output <= "0001010";
when -19 =>
output <= "0001011";
when -18 =>
output <= "0001011";
when -17 =>
output <= "0001100";
when -16 =>
output <= "0001100";
when -15 =>
output <= "0001101";
when -14 =>
output <= "0001110";
when -13 =>
output <= "0001111";
when -12 =>
output <= "0010000";
when -11 =>
output <= "0010010";
when -10 =>
output <= "0010100";
when -9 =>
output <= "0010110";
when -8 =>
output <= "0011000";
```

```
when -7 =>
output <= "0011010";
when -6 =>
output <= "0011100";
when -5 =>
output <= "0011110";
when -4 =>
output <= "0100000";
when -3 =>
output <= "0101000";
when -2 =>
output <= "0101111";
when -1 =>
output <= "1001111";
when 0 =>
output <= "1111111";
when 127 =>
output <= "0000000";
when 126 =>
output <= "0000000";
when 125=>
output <= "0000000";
when 124=>
output <= "0000000";
when 123=>
output <= "0000000";
when 122=>
output <= "0000000";
when 121=>
output <= "0000000";
when 120=>
output <= "0000000";
when 119 =>
output <= "0000000";
when 118 =>
output <= "0000000";
when 117 =>
output <= "0000000";
when 116 =>
output <= "0000000";
when 115 =>
output <= "0000000";
when 114 =>
output <= "0000000";
when 113 =>
output <= "0000000";
when 112 =>
output <= "0000000";
when 111 =>
output <= "0000000";
when 110 =>
output <= "0000000";
when 109 =>
output <= "0000000";
when 108 =>
output <= "0000000";
when 107 =>
output <= "0000000";
when 106 =>
```

```
output <= "0000000";
when 105 =>
output <= "0000000";
when 104 =>
output <= "0000000";
when 103 =>
output <= "0000000";
when 102 =>
output <= "0000000";
when 101 =>
output <= "0000000";
when 100 =>
output <= "0000000";
when 99 =>
output <= "0000000";
when 98 =>
output <= "0000000";
when 97 =>
output <= "0000001";
when 96=>
output <= "0000001";
when 95=>
output <= "0000001";
when 94=>
output <= "0000001";
when 93=>
output <= "0000001";
when 92=>
output <= "0000001";
when 91=>
output <= "0000001";
when 90=>
output <= "0000001";
when 89=>
output <= "0000001";
when 88=>
output <= "0000001";
when 87=>
output <= "0000001";
when 86=>
output <= "0000001";
when 85=>
output <= "0000001";
when 84 =>
output <= "0000001";
when 83 =>
output <= "0000001";
when 82 =>
output <= "0000001";
when 81 =>
output <= "0000001";
when 80 =>
output <= "0000001";
when 79 =>
output <= "0000001";
when 78 =>
output <= "0000001";
when 77 =>
output <= "0000001";
```

```
when 76 =>
output <= "0000001";
when 75 =>
output <= "0000001";
when 74 =>
output <= "0000001";
when 73 =>
output <= "0000001";
when 72 =>
output <= "0000001";
when 71 =>
output <= "0000001";
when 70 =>
output <= "0000001";
when 69 =>
output <= "0000001";
when 68 =>
output <= "0000001";
when 67 =>
output <= "0000001";
when 66 =>
output <= "0000001";
when 65 =>
output <= "0000001";
when 64 =>
output <= "0000001";
when 63 =>
output <= "0000001";
when 62 =>
output <= "0000001";
when 61 =>
output <= "0000001";
when 60 =>
output <= "0000001";
when 59 =>
output <= "0000001";
when 58 =>
output <= "0000001";
when 57 =>
output <= "0000001";
when 56 =>
output <= "0000010";
when 55 =>
output <= "0000010";
when 54 =>
output <= "0000010";
when 53 =>
output <= "0000010";
when 52 =>
output <= "0000010";
when 51 =>
output <= "0000010";
when 50 =>
output <= "0000010";
when 49 =>
output <= "0000010";
when 48 =>
output <= "0000010";
when 47 =>
```

```
output <= "0000010";
when 46 =>
output <= "0000010";
when 45 =>
output <= "0000010";
when 44 =>
output <= "0000011";
when 43 =>
output <= "0000011";
when 42 =>
output <= "0000011";
when 41 =>
output <= "0000011";
when 40 =>
output <= "0000011";
when 39 =>
output <= "0000011";
when 38 =>
output <= "0000011";
when 37 =>
output <= "0000011";
when 36 =>
output <= "0000100";
when 35 =>
output <= "0000100";
when 34 =>
output <= "0000100";
when 33 =>
output <= "0000100";
when 32 =>
output <= "0000100";
when 31 =>
output <= "0000101";
when 30 =>
output <= "0000101";
when 29 =>
output <= "0000110";
when 28 =>
output <= "0000110";
when 27 =>
output <= "0000111";
when 26 =>
output <= "0000111";
when 25 =>
output <= "0001000";
when 24 =>
output <= "0001000";
when 23 =>
output <= "0001001";
when 22 =>
output <= "0001001";
when 21 =>
output <= "0001010";
when 20 =>
output <= "0001010";
when 19 =>
output <= "0001011";
when 18 =>
output <= "0001011";
```

```
when 17 =>
output <= "0001100";
when 16 =>
output <= "0001100";
when 15 =>
output <= "0001101";
when 14 =>
output <= "0001110";
when 13 =>
output <= "0001111";
when 12 =>
output <= "0010000";
when 11 =>
output <= "0010010";
when 10 =>
output <= "0010100";
when 9 =>
output <= "0010110";
when 8 =>
output <= "0011000";
when 7 =>
output <= "0011010";
when 6 =>
output <= "0011100";
when 5 =>
output <= "0011110";
when 4 =>
output <= "0100000";
when 3 =>
output <= "0101000";
when 2 =>
output <= "0101111";
when 1 =>
output <= "1001111";
end case;
end process;
end behavior;
```

B.2 Carry Lookahead Adder

look_ahead_adder.vhd

```
LIBRARY ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;


ENTITY  Look_Ahead_Adder IS
PORT( A, B : IN std_logic_vector( 6 DOWNTO 0 );
--carry_in : IN std_logic;
--carry_out : OUT std_logic;
S : OUT std_logic_vector( 6 DOWNTO  0 ) );
END Look_Ahead_Adder;

ARCHITECTURE RTL OF Look_Ahead_Adder IS
COMPONENT carry_generator
```

```
PORT( P, G : in std_logic_vector(7 DOWNTO 0);
--C1 : in std_logic;
C : OUT std_logic_vector(8 DOWNTO 0));
END COMPONENT;

COMPONENT half_adder
PORT( A, B : IN std_logic_vector( 7 DOWNTO 0 );
P, G : OUT std_logic_vector( 7 DOWNTO 0) );
END COMPONENT;
For CG : carry_generator Use entity work.carry_generator(RTL);
For HA : half_adder Use entity work.half_adder(RTL);
SIGNAL tempG, tempP : std_logic_vector( 7 DOWNTO 0 );
SIGNAL tempC : std_logic_vector( 8 DOWNTO 0 );
begin
HA: half_adder PORT MAP( A=>A, B=>B, P =>tempP, G=>tempG );
CG: carry_generator PORT MAP( P=>tempP, G=>tempG, C=>tempC );
with tempC(8 DOWNTO 7) SELECT S <=
"01111111" WHEN "01",
"10000001" WHEN "10",
(tempC(7 DOWNTO 0 ) xor tempP) WHEN OTHERS;
--S <= tempC( 7 WHEN 0 ) xor tempP;
--carry_out <= tempC(8);
END;
```

carry generator.vhd

```
library IEEE;
USE ieee.std_logic_1164.all;
ENTITY carry_generator IS
port( P , G : IN std_logic_vector(7 DOWNTO 0);
--C1 : IN std_logic;
C : OUT std_logic_vector(8 DOWNTO 0));
END carry_generator;
ARCHITECTURE RTL OF carry_generator IS
BEGIN
PROCESS(P, G)
VARIABLE tempC : std_logic_vector(8 DOWNTO 0);
BEGIN
tempC(0) := '0';
FOR i IN 0 to 7 LOOP
tempC(i+1) := G(i) OR (P(i) AND tempC(i));
END LOOP;
C <= tempC;
END PROCESS;
END;
```

half_adder.vhd

```
library IEEE;
use ieee.std_logic_1164.all;
ENTITY half_adder IS
PORT( A, B : IN std_logic_vector( 7 DOWNTO 0 );
P, G : out std_logic_vector( 7 DOWNTO 0 ) );
END half_adder;

ARCHITECTURE RTL OF half_adder IS
```

```
begin
P <= A xor B;
G <= A and B;
END;
```

B.3 Procesorul nodului bit

Vadder_ram.vhd

```
library ieee;
use ieee.std_logic_1164.all;
USE ieee.numeric_std.all;
entity Vadder_ram is
port(
clk : IN std_logic;
wenable : IN std_logic;
sel_r : IN std_logic;
sel_w : IN std_logic;
addr : IN integer range 0 to 2;
A : IN std_logic_vector(7 DOWNTO 0);
X0 : OUT std_logic_vector(7 DOWNTO 0);
X1 : OUT std_logic_vector(7 DOWNTO 0);
X2 : OUT std_logic_vector(7 DOWNTO 0));
end Vadder_ram;
architecture behavior of Vadder_ram is
TYPE vector_array IS ARRAY (0 TO 2) OF STD_LOGIC_VECTOR (7 DOWNTO 0);
SIGNAL memory0 : vector_array;
SIGNAL memory1 : vector_array;
begin
process(clk,sel_r,sel_w,addr,A,wenable,memory0,memory1)
begin
if (clk'event AND clk='1') then
if (wenable='1') then
if (sel_w='0') then
memory0(addr) <= A;
else
memory1(addr) <= A;
end if;
end if;
end if;
if (sel_r='1') then
X0 <= memory1(0);
X1 <= memory1(1);
X2 <= memory1(2);
else
X0 <= memory0(0);
X1 <= memory0(1);
X2 <= memory0(2);
end if;
end process;
end behavior;
```

Vcontrol.vhd

```
library ieee;
```

```
use ieee.std_logic_1164.all;
USE ieee.numeric_std.all;
entity Vcontrol2 is
port(
clk : IN std_logic;
reset : IN std_logic;
lam_deg : IN integer range 0 to 3;
send_NR : IN std_logic; --1 SEND EDGES, 0 RECIEVE EDGES
edge_address : OUT std_logic_vector(7 downto 0);
lambda_address : OUT std_logic_vector(5 downto 0);
dv_address : OUT std_logic_vector(5 downto 0);
mem_sel_r : BUFFER std_logic;
mem_sel_w : BUFFER std_logic;
adder_mux : OUT std_logic_vector(0 to 2);
adder_address : OUT std_logic_vector(1 downto 0);
adder_wr : OUT std_logic;
edge_wr : OUT std_logic);
end Vcontrol2;
architecture behavior of Vcontrol2 is
type state_type is (rst,zero,one,two,recv);
signal state_w : state_type;
signal state_r : state_type;
begin
process(clk,reset)
VARIABLE edg_address_base : integer range 0 to 255;
VARIABLE lam_address : integer range 0 to 63;
VARIABLE dv_addres : integer range 0 to 63;
VARIABLE dv_previous : integer range 0 to 3;
VARIABLE dv_current : integer range 0 to 3;
VARIABLE dv_current_r : integer range 0 to 3;
begin
if (reset='1') then
state_r <= rst;
state_w <= rst;
elsif (clk'event AND clk='1') then
case state_w is
when rst =>
lambda_address <= "111111";
lam_address := 63;
dv_address <= "000000";
dv_addres := 0;
mem_sel_r <= '1';
mem_sel_w <= '0';
adder_mux <= "000";
adder_address <= "00";
dv_current := 3;
dv_previous :=3;
dv_current_r :=3;
if (send_NR = '1') then
state_w <= zero;
edg_address_base := lam_deg -1;
edge_address <= std_logic_vector(to_unsigned(edg_address_base,8));
adder_wr <= '1';
edge_wr <= '0';
else
state_w <= recv;
edg_address_base :=0;
edge_address <= std_logic_vector(to_unsigned(edg_address_base,8));
adder_wr <= '0';
```

```
edge_wr <= '1';
end if;
when zero =>
dv_previous := dv_current;
dv_current := lam_deg;
state_w <= one;
mem_sel_w <= NOT mem_sel_w;
if (edg_address_base = 2) then
dv_addres := dv_addres+1;
end if;
edge_address <= std_logic_vector(to_unsigned(edg_address_base-1,8));
adder_address <="00";
dv_address <=std_logic_vector(to_unsigned(dv_addres,6));
adder_wr <= '1';
edge_wr <= '0';
when one =>
adder_address <="01";
if (dv_current = 2) then
adder_mux(2) <='0';
state_w <= zero;
edg_address_base := edg_address_base+lam_deg;
edge_address <= std_logic_vector(to_unsigned(edg_address_base,8));
dv_addres := dv_addres+1;
else
state_w <= two;
edge_address <= std_logic_vector(to_unsigned(edg_address_base-2,8));
end if;
dv_address <=std_logic_vector(to_unsigned(dv_addres,6));
adder_wr <= '1';
edge_wr <= '0';
when two =>
adder_address <="10";
edg_address_base := edg_address_base+lam_deg;
edge_address <= std_logic_vector(to_unsigned(edg_address_base,8));
state_w <= zero;
dv_addres := dv_addres+1;
dv_address <=std_logic_vector(to_unsigned(dv_addres,6));
adder_wr <= '1';
edge_wr <= '0';
when recv =>
adder_wr <= '0';
edge_wr <= '1';
edg_address_base := edg_address_base +1;
edge_address <= std_logic_vector(to_unsigned(edg_address_base,8));
lambda_address <= std_logic_vector(to_unsigned(edg_address_base,6));
dv_address <= "000000";
mem_sel_r <= '1';
mem_sel_w <= '0';
adder_mux <= "000";
adder_address <= "00";
state_w <=recv;
end case;
case state_r is
when rst =>
if (send_NR = '1') then
state_r <= zero;
else
state_r <= recv;
end if;
```

```
when zero =>
mem_sel_r <= NOT mem_sel_r;
adder_mux <= "110";
if (dv_current_r = 2) then
adder_mux(0) <= '0';
end if;
lambda_address <= std_logic_vector(to_unsigned(lam_address,6));
state_r <= one;
when one =>
adder_mux <= "101";
if (dv_current_r = 2) then
adder_mux(0) <= '0';
lam_address := lam_address +1;
dv_current_r := dv_previous;
state_r <= zero;
else
state_r <= two;
end if;
lambda_address <= std_logic_vector(to_unsigned(lam_address,6));
when two =>
adder_mux <="011";
lam_address := lam_address +1;
dv_current_r := dv_previous;
state_r <= zero;
lambda_address <= std_logic_vector(to_unsigned(lam_address,6));
when recv =>state_r <=recv;
end case;
end if;
end process;
end behavior;
```

B.4 Procesorul nodului de verificare

nodecontrol.vhd

```
library ieee;
use ieee.std_logic_1164.all;
USE ieee.numeric_std.all;
entity Nodecontrol is
port(
clk : IN std_logic;
reset : IN std_logic;
state_reset : IN std_logic;
v_deg : IN integer range 0 to 8;
send_NR : IN std_logic; --1 SEND EDGES, 0 RECIEVE EDGES
edge_address : OUT std_logic_vector(4 downto 0);
dc_address : OUT std_logic_vector(4 downto 0);
mem_sel_r : BUFFER std_logic;
mem_sel_w : BUFFER std_logic;
adder_mux : OUT std_logic_vector(5 to 0);
adder_address : OUT std_logic_vector(2 downto 0);
adder_wr : OUT std_logic;
edge_wr : OUT std_logic);
end Nodecontrol;
architecture behavior of Nodecontrol is
type state_type1 is (rst,zero,one,two,three,four,five,recv);
type state_type2 is (rst,zero,one,two,three,four,five,recv);
```

```
signal state_w : state_type1;
signal state_r : state_type2;
begin
process(clk,reset,send_NR,v_deg)
VARIABLE edg_address_base : integer range 0 to 255;
VARIABLE dc_addres : integer range 0 to 31;
VARIABLE dc_previous : integer range 0 to 8;
VARIABLE dc_current : integer range 0 to 8;
VARIABLE dc_current_r : integer range 0 to 8;
begin
if (reset='1') then
dc_address <= "00000";
dc_addres := 0;
mem_sel_r <= '1';
mem_sel_w <= '0';
adder_mux <= "000000";
adder_address <= "000";
dc_current := 6;
dc_previous :=6;
dc_current_r :=6;
state_w <= rst;
state_r <= rst;
edg_address_base := v_deg -1;
edge_address <= std_logic_vector(to_unsigned(edg_address_base,5));
adder_wr <= '1';
edge_wr <= '0';
elsif (clk'event AND clk='1') then
case state_w is
when rst =>
edg_address_base := v_deg -1;
adder_wr <= '1';
edge_wr <= '0';
dc_current := 6;
dc_previous :=6;
dc_current_r :=6;
if (send_NR = '1') then
state_w <= zero;
else
state_w <= recv;
edg_address_base := 255;
end if;
edge_address <= std_logic_vector(to_unsigned(edg_address_base,5));
if (state_reset = '1') then
state_w <= rst;
end if;
when zero =>
dc_previous := dc_current;
dc_current := v_deg;
state_w <= one;
mem_sel_w <= NOT mem_sel_w;
if (edg_address_base = 7) then
dc_addres := dc_addres+1;
end if;
edge_address <= std_logic_vector(to_unsigned(edg_address_base-1,5));
adder_address <="000";
dc_address <=std_logic_vector(to_unsigned(dc_addres,5));
adder_wr <= '1';
edge_wr <= '0';
if (state_reset = '1') then
```

```
state_w <= rst;
end if;
when one =>
adder_address <="001";
if (dc_current = 2) then
state_w <= zero;
edg_address_base := edg_address_base+v_deg;
edge_address <= std_logic_vector(to_unsigned(edg_address_base,5));
dc_addres := dc_addres+1;
else
state_w <= two;
edge_address <= std_logic_vector(to_unsigned(edg_address_base-2,5));
end if;
dc_address <=std_logic_vector(to_unsigned(dc_addres,5));
adder_wr <= '1';
edge_wr <= '0';
if (state_reset = '1') then
state_w <= rst;
end if;
when two =>
adder_address <="010";
if (dc_current = 3) then
state_w <= zero;
edg_address_base := edg_address_base+v_deg;
edge_address <= std_logic_vector(to_unsigned(edg_address_base,5));
dc_addres := dc_addres+1;
else
state_w <= three;
edge_address <= std_logic_vector(to_unsigned(edg_address_base-3,5));
end if;
dc_address <=std_logic_vector(to_unsigned(dc_addres,5));
adder_wr <= '1';
edge_wr <= '0';
if (state_reset = '1') then
state_w <= rst;
end if;
when three =>
adder_address <="011";
if (dc_current = 4) then
state_w <= zero;
edg_address_base := edg_address_base+v_deg;
edge_address <= std_logic_vector(to_unsigned(edg_address_base,5));
dc_addres := dc_addres+1;
else
state_w <= four;
edge_address <= std_logic_vector(to_unsigned(edg_address_base-4,5));
end if;
dc_address <=std_logic_vector(to_unsigned(dc_addres,5));
adder_wr <= '1';
edge_wr <= '0';
if (state_reset = '1') then
state_w <= rst;
end if;
when four =>
adder_address <="100";
if (dc_current = 5) then
state_w <= zero;
edg_address_base := edg_address_base+v_deg;
edge_address <= std_logic_vector(to_unsigned(edg_address_base,5));
```

```
dc_addres := dc_addres+1;
else
state_w <= five;
edge_address <= std_logic_vector(to_unsigned(edg_address_base-5,5));
end if;
dc_address <=std_logic_vector(to_unsigned(dc_addres,5));
adder_wr <= '1';
edge_wr <= '0';
if (state_reset = '1') then
state_w <= rst;
end if;
when five =>
adder_address <="101";
state_w <= zero;
edg_address_base := edg_address_base+v_deg;
edge_address <= std_logic_vector(to_unsigned(edg_address_base,5));
dc_addres := dc_addres+1;
dc_address <=std_logic_vector(to_unsigned(dc_addres,5));
adder_wr <= '1';
edge_wr <= '0';
if (state_reset = '1') then
state_w <= rst;
end if;
when recv =>
adder_wr <= '0';
edge_wr <= '1';
edg_address_base := edg_address_base +1;
edge_address <= std_logic_vector(to_unsigned(edg_address_base,5));
dc_address <= std_logic_vector(to_unsigned(edg_address_base,5));
mem_sel_r <= '1';
mem_sel_w <= '0';
adder_mux <= "000000";
adder_address <= "000";
state_w <=recv;
if (state_reset = '1') then
state_w <= rst;
end if;
end case;
case state_r is
when rst =>
if (send_NR = '1') then
state_r <= zero;
else
state_r <= recv;
end if;
if (state_reset = '1') then
state_r <= rst;
end if;
when zero =>
adder_mux <= "111110";
state_r <= one;
mem_sel_r <= NOT mem_sel_r;
if (state_reset = '1') then
state_r <= rst;
end if;
when one =>
adder_mux <= "111101";
if (dc_current_r = 2) then
dc_current_r := dc_previous;
```

```
state_r <= zero;
else
state_r <= two;
end if;
if (state_reset = '1') then
state_r <= rst;
end if;
when two =>
adder_mux <= "111011";
if (dc_current_r = 3) then
dc_current_r := dc_previous;
state_r <= zero;
else
state_r <= three;
end if;
if (state_reset = '1') then
state_r <= rst;
end if;
when three =>
adder_mux <= "110111";
if (dc_current_r = 4) then
dc_current_r := dc_previous;
state_r <= zero;
else
state_r <= four;
end if;
if (state_reset = '1') then
state_r <= rst;
end if;
when four =>
adder_mux <= "101111";
if (dc_current_r = 5) then
dc_current_r := dc_previous;
state_r <= zero;
else
state_r <= five;
end if;
if (state_reset = '1') then
state_r <= rst;
end if;
when five =>
adder_mux <= "011111";
dc_current_r := dc_previous;
state_r <= zero;
if (state_reset = '1') then
state_r <= rst;
end if;
when recv =>
state_r <=recv;
if (state_reset = '1') then
state_r <= rst;
end if;
end case;
if (dc_current_r = 2) then
adder_mux(5 to 0) <="000000";
elsif (dc_current_r = 3) then
adder_mux(4 to 0) <="00000";
elsif (dc_current_r = 4) then
adder_mux(3 to 0) <="0000";
```

```
elsif (dc_current_r = 5) then
adder_mux(2 to 0) <="000";
end if;
end if;
end process;
end behavior;
```

c_adder.vhd

```
library ieee;
use ieee.std_logic_1164.all;
USE ieee.numeric_std.all;
entity Cadder_ram is
port(
clk : IN std_logic;
wenable : IN std_logic;
sel_r : IN std_logic;
sel_w : IN std_logic;
addr : IN integer range 0 to 2;
A : IN std_logic_vector(7 downto 0);
X0 : OUT std_logic_vector(7 downto 0);
X1 : OUT std_logic_vector(7 downto 0);
X2 : OUT std_logic_vector(7 downto 0);
X3 : OUT std_logic_vector(7 downto 0);
X4 : OUT std_logic_vector(7 downto 0);
X5 : OUT std_logic_vector(7 downto 0);
X6 : OUT std_logic_vector(7 downto 0);
X7 : OUT std_logic_vector(7 downto 0));
end Cadder_ram;
architecture behavior of Cadder_ram is
TYPE vector_array IS ARRAY (0 TO 2) OF STD_LOGIC_VECTOR (7 DOWNTO 0);
SIGNAL memory0 : vector_array;
SIGNAL memory1 : vector_array;
begin
process(clk,sel_r,sel_w,addr,A,wenable,memory0,memory1)
begin
if (clk'event AND clk='1') then
if (wenable='1') then
if (sel_w='0') then
memory0(addr) <= A;
else
memory1(addr) <= A;
end if;
end if;
end if;
if (sel_r='1') then
X0 <= memory1(0);
X1 <= memory1(1);
X2 <= memory1(2);
X3 <= memory1(3);
X4 <= memory1(4);
X5 <= memory1(5);
X6 <= memory1(6);
X7 <= memory1(7);
else
X0 <= memory0(0);
X1 <= memory0(1);
X2 <= memory0(2);
```

```
X3 <= memory0(3);
X4 <= memory0(4);
X5 <= memory0(5);
X6 <= memory0(6);
X7 <= memory0(7);
end if;
end process;
end behavior;
```

B.5 Incarcatorul cuvantului de cod

lambda_write.vhd

```
library ieee;
use ieee.std_logic_1164.all;
USE ieee.numeric_std.all;
entity lambda_wr is
port(
clk : IN std_logic;
reset : IN std_logic;
in_ready : IN std_logic; --cand datele sunt gata pentru a fi trimise
control_ready : IN std_logic; --pregatit sa accepte una sau mai multe intrari
lambda_address : OUT std_logic_vector(8 downto 0);
done_reading : OUT std_logic; --am terminat de citit datele
rst_latch : OUT std_logic; --merge mai departe dupa ce am terminat de citit datele
lambda_write : OUT std_logic);
end lambda_wr;
architecture behavior of lambda_wr is
type state_type is (idle,processing,finish);
signal state : state_type;
begin
process(clk,reset)
VARIABLE lam_address : integer range 0 to 511;
begin
if (reset = '1') then
lambda_write <='0';
lambda_address <= "000000000";
lam_address := 0;
done_reading <= '0';
rst_latch <= '0';
state <= idle;
elsif (clk'event AND clk='1') then
case state is
when idle =>
lambda_write <='0';
rst_latch <= '0';
lambda_address <= "000000000";
lam_address :=0;
if (in_ready = '1' AND control_ready = '1') then
state <= processing;
else
state <=idle;
end if;
done_reading <= '0';
when processing =>
lambda_write <='1';
done_reading <= '0';
```

```
rst_latch <= '0';
if (lam_address = 480) then
state <= finish;
else
state <=processing;
end if;
lambda_address <= std_logic_vector(to_unsigned(lam_address,9));
lam_address := lam_address + 1;
when finish =>
done_reading <= '1';
rst_latch <= '1';
lambda_write <='0';
lambda_address <="000000000";
state <= idle;
end case;
end if;
end process;
end behavior;
```