Accelerare hardware a explorării în spațiul stărilor pentru jocul Reversi

Marius M. TIVADAR
Universitatea „POLITEHNICA” Timișoara, România
Conducător științific: conf. dr. ing. Doru TODINCĂ
- Lucrare de diplomă -

21 iunie 2009

Cuprins

Capitolul 1

Introducere

Ideea de „gândire artificială" în jocurile precum Șah, Go, Reversi a fascinat lumea încă din cele mai vechi timpuri în momentul în care nu existau calculatoare moderne, iar posibilitatea rezolvării unui joc de către o mașină era un mister total. În anul 1770, mașina *The Turk* era cunoscută ca și „mașina automată de șah", a fost probabil una din cele mai importante invenții ale omului care a răspândit întrebarea, „Oare este capabilă o mașină să gândească?". Bineînteles, *The Turk* era o farsă extraordinar de bine realizată, o mașinărie construită de către *Wolfgang von Kempelen* care părea capabilă de un joc de șah foarte competent, precum și de rezolvarea problemei *Turul Cavalerului*.[1] Cel puțin așa se credea inițial, pentru că mașinăria avea de fapt în ea ascuns un om, care printr-un mecanism complicat, instruia operatorul ce să facă. Farsa a bucurat oamenii timp de 84 de ani, însuși Napolean Bonaparte jucând o partidă cu mașinăria. A fost un pas mare din punct de vedere filosofic, întrebarea a rămas, frământând mințile oamenilor mult timp.
În 1927, matematicianul și filosoful John von Neumann, a enunțat în lucrarea sa „Zur Theorie der Gesellschaftsspiele" teoria *minmax*, iar în 1944 împreună cu Oskar Morgenstern a fundamentat domeniul Teoria Jocurilor. Din acest moment exista posibilitatea algoritmizării unui joc de șah.[1][2]
Claude Shannon, a publicat în 1949 lucrarea intitulată „Programming a Computer for Playing Chess" în care a descris funcționalitatea unui program de șah pe un calculator, folosind teoriile lui Neumann. Shannon a observat imposibilitatea unui calculator de a explora întreg spațiul al stărilor și a propus metoda de explorare parțială, evaluând starea tablei de joc printr-o funcție euristică propusă. Programul, nefiind realizat, era considerat capabil să joace împotriva unui adversar începător.[3]
În 1997 IBM a proiectat calculatorul *Deep Blue*, un adversar comparabil cu campionul mondial *Garry Kasparov*. Mașinăria era un monstru[4], o arhitectură masiv paralelă conținând 30 de procesoare IBM RS/6000 care funcționau la frecvența de $120MHz$, la care se adăugau 480 de chipuri VLSI specializate pentru jocul de șah. Deep Blue era capabil sa analizeze 200 milioane poziții/secundă. Rezultatul a fost $3.5 - 2.5$ pentru *Deep Blue*.[2]

[1] Problema acoperirii tablei de șah cu un cal, astăzi o problemă trivială rezolvabilă prin forță brută, existând și metode mai evoluate.

[2] 0.5 reprezintă remiză.

1.1 Tema Proiectului

Ce îmi propun eu în acest proiect, este de a proiecta un calculator capabil să joace Reversi cu o accelerare puternică hardware, în dorința de a fi mai rapid ca un PC. Funcționalitatea este implementată în FPGA, iar mașina este autonomă, fiind legată doar la un monitor pentru a vizualiza tabla. Pentru explorare în spațiul stărilor folosesc algoritmul clasic *minmax* cu optimizare *alpha-beta* ce va reduce complexitatea algoritmului clasic. Voi proiecta practic un chip specializat, prototip implementat în FPGA, capabil să joace acest joc. Modulele din care este compus, sunt specializate și fiecare au funcții specifice jocului. De exemplu, modulul de generare a mutărilor valide, modulul de tranziționare de la o stare la alta, modulul de evaluare euristică a tablei de joc. Fiecare modul este gândit încât să exploateze avantajul hardware în fața software-ului pe un CPU general. Mașina are un semnal de ceas de $50MHz$. Voi identifica modulele considerate cele mai costisitoare computațional și am să încerc o paralelizare internă a acestora. Modulul de evaluare a stării de joc, conține o serie de parametrii determinați empiric, iar proiectarea lui va exploata puternic capabilitățile hardware. Totuși, pentru creșterea frecvenței va fi nevoie de și mai multă proiectare, dar nu este scopul lucrării de față. În final, voi compara cu același algoritm, aceleași euristici, dar implementate pe un PC. Rezultatul este pe departe în favoarea jocului implementat în FPGA.

Capitolul 2

Jocul Reversi

2.1 Istoric

Varianta modernă a jocului este bazată pe jocul Reversi inventat în anul 1883 de către englezul Lewis Waterman și a fost foarte popular în Anglia la sfârșitul secolului XIX.
Regulile moderne, acum universal acceptate, își au originea în Japonia de prin anii 1970. Jocul a fost redenumit în numele Othello[1] și a devenit o marcă înregistrată a companiei japoneze *Tsukuda Original*. Campionate mondiale de Othello se țin din anul 1977, iar majoritatea campionilor sunt japonezi. În 1997, Logistello [5] a câștigat în fața campionului mondial Takeshi Murakami cu scorul de 6:0. În ziua de azi, cel puțin după regulile standard, adică o tablă de joc 8x8, omul este considerat învins de către calculator [6].

2.2 Regulile jocului

2.2.1 Notația

În figura 2.1 avem notația standard în joc. Coloanele sunt numerotate de la *a* la *h* de la stânga spre dreapta, iar liniile sunt numerotate de la *1* la *8* de sus în jos. O poziție va fi referită prin notația $a1$, de exemplu pentru colțul stânga sus. Jucătorii sunt în general considerați *Negru* și *Alb*, dar se mai folosește și *Albastru* și *Roșu* cum am folosit și eu la afișarea pe monitor.

2.2.2 Reguli

Jocul începe cu discuri negre poziționate pe poziția d5 și e4 și cu discuri albe poziționate pe poziția d4 și e5, precum în figura 2.1. Jucătorul negru va fi primul care va face o mutare, după care jucătorii vor alterna. O mutare legală constă în a pune un disc pe tabla de joc pe o poziție liberă, astfel încât să se captureze unul sau mai multe discuri inamice. Orice disc adversar care este flancat de către discul tocmai pus și un alt disc de aceași culoare pe o direcție oarecare, vor fi capturate. Capturarea se face pe direcțiile orizontală, verticală și pe ambele diagonale pe fiecare direcție în ambele sensuri. Capturarea

[1] În lucrare voi folosi ambele denumiri.

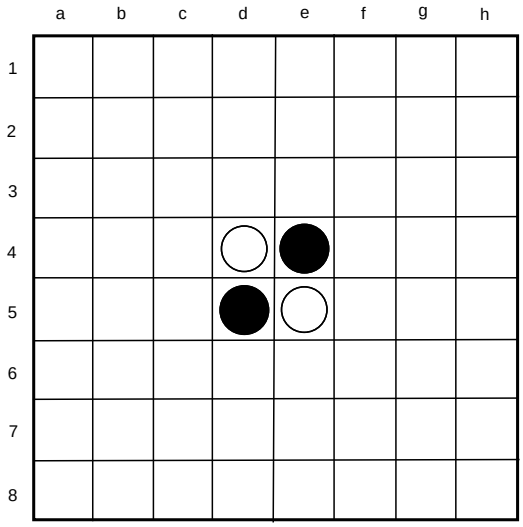

Figura 2.1: Poziția de început în Reversi/Othello.

este posibilă doar dacă între cele două discuri care flanchează, vor exista numai discuri adversare și toate pozițiile vor fi ocupate. O capturare constă în transformarea discurilor adversare în culoarea discului adăugat. Discurile pot fi întoarse (capturate) pe mai multe direcții la aceași mutare. Toate discurile flancate trebuie capturate, jucătorul nu poate alege care dintre acestea să fie. Dacă un jucător nu are mutări legale, adică oriunde ar poziționa un disc pe o poziție liberă nu va captura nici un disc adversar, atunci va ceda mutarea adversarului care va face mutări consecutive până când va avea și celălalt jucător o mutare legală. Un jucător nu poate ceda mutarea din propria lui inițiativă. Jocul se va termina când nici unul dintre jucător nu vor mai avea mutări legale. Dacă se joacă pe o tablă fizică, discurile vor avea pe o parte culoarea *neagră*, iar pe cealaltă parte culoarea *albă*. [7, 8]

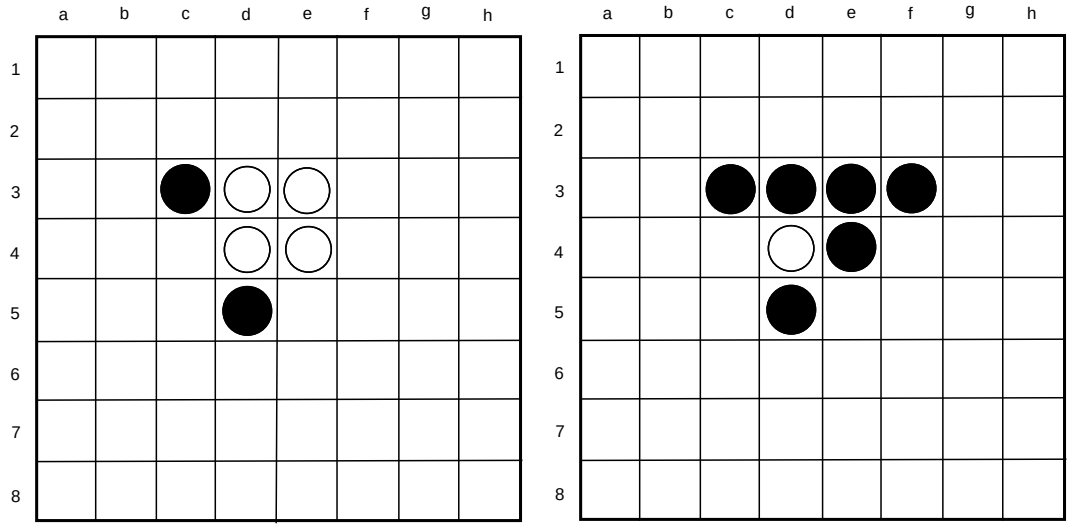

Figura 2.2: (a) O stare oarecare de joc. (b) După ce negru pune un disc la „f3”.

La sfârșitul jocului va câștiga jucătorul care are cele mai multe discuri de culoarea lui, pe tabla de joc. Puțin contează numărul de discuri diferență.

2.3 Particularități, principii

2.3.1 Colțurile

Pentru jucătorii începători și intermediari [7], bătălia se dă pentru cucerirea colțurilor. Motivul este foarte clar: un disc poziționat în colț, nu va mai putea fi cucerit de către adversar, pentru că adversarul nu va avea cum să-l flancheze pentru a-l captura. Un colț devine un disc *stabil*, stabilitatea fiind singurul factor care se traduce direct în câștig în Othello.

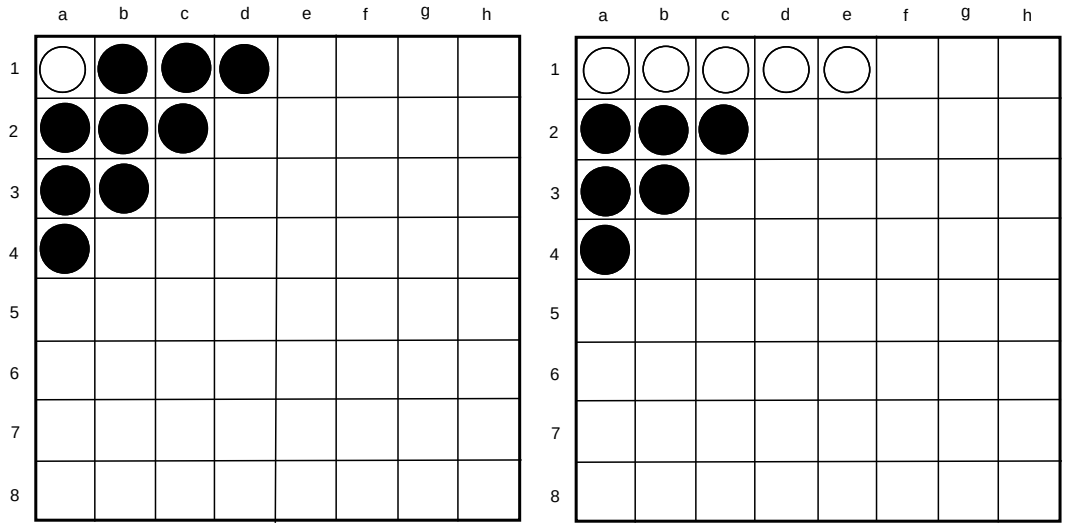

Figura 2.3: (a) O stare oarecare de joc. (b) După ce albul pune un disc la „e1”, va genera 5 discuri stabile.

În figura 2.3 se vede cum jucătorul alb construiește discuri stabile după ce a cucerit colțul. Cele 5 discuri de pe linie, nu vor mai putea fi capturate de adversar, deci vor contribui direct la scorul final. În general, pozițiile din jurul colțurilor sunt periculoase, adică pot permite adversarului să cucerească colțul.

2.3.2 Concepte intermediare

„Greed is one of the deadly seven sins”[2] [7] se aplică și în Othello. Ce se poate face în începutul jocului, când mai e mult până să ajungem în zona colțurilor? S-a observat că numărul mare de piese capturate nu este deloc un lucru bun. Motivul ar fi pentru că scade *mobilitatea*, adică dacă pe tablă vom avea doar discuri de culoarea noastră și unul de culoarea adversarului, probabil nu vom mai avea mutări valide, iar adversarul va avea multe opțiuni. O tehnică bună, se pare că e să capturăm cât mai puține discuri la o mutare. [7] O mutare *tăcută*

[2]Lăcomia este unul dintre cele 7 păcate.

în Othello, este o mutare care nu va schimba cu mult configurația tablei de joc. Dacă de exemplu capturăm doar un disc adversar, acea mutare este o mutare tăcută. Acest concept este preluat din Șah, unde o mutare tăcută înseamnă cam același lucru. De obicei, o mutare tăcută este întotdeauna de preferat.

Zugzwang

Este un concept preluat din Șah, ce se aplică și la Othello și la multe alte jocuri. Un jucător este în *Zugzwang*[3] dacă este rândul lui la mutare, dar ar prefera să nu facă acea mutare pentru că i-ar aduce un dezavantaj. În Othello acest lucru se poate întâmpla atunci când singurele mutări valide ale jucătorului sunt cele de a ceda un colț. Acest lucru face diferența dintre un jucător amator/intermediar și unul avansat. Un jucător avansat va reuși să aducă în zugzwang pe adversar.

Mobilitatea

Mobilitatea este cel mai important lucru în Othello. O mobilitate mică, va duce la o dominare din partea adversarului, care ne va forța în *zugzwang*. Mobilitatea se poate crește prin capturarea a cât mai puține discuri la o mutare. Funcțiile euristice construite în programele de Reversi se bazează foarte mult pe acest concept. Acest lucru ar părea la prima vedere în contradicție cu faptul că trebuie să câștigăm prin a avea mai multe discuri pe tablă decât adversarul. Regula este de a avea mai multe discuri ca adversarul *la sfârșitul* jocului, nu în mijlocul lui.

2.3.3 Fazele jocului

Jocul se împarte în 3 faze: jocul de început, jocul de mijloc, jocul de sfârșit.[4]
În faza de început, un sfat bun [7] este să rămânem cu discurile în pătratul de 4x4 din interiorul tablei de joc, având colțurile opuse „c3” și „f6”. Mutările în acest pătrat va împiedica adversarul să poziționeze discuri pe laturi, lucru ce îi poate oferi un avantaj deoarece mutăriile pe laturi au șanse mai mari să fie mutări tăcute care vor scădea mobilitatea celuilalt jucător.
În faza de mijloc, cel mai important lucru este creșterea mobilității. În acest fel se va forța adversarul să cedeze colțurile.
În faza de sfârșit, urmează să construim pe ce am acumulat până acum. Dacă avem colțuri, le putem folosi ca ancore pentru a construi mai multe discuri stabile. În această fază a jocului se vor încălca toate regulile descrise anterior, cel mai important lucru fiind capturarea de cât mai multe discuri. Discurile stabile ar trebui construite sistematic, fără a lăsa discuri adversare între ele, care ar putea în final să întoarcă situația scorului. O carte foarte bună despre jocul Othello este „Othello: A Minute to Learn, A Lifetime to Master.” de Brian Rose [8] sau alta ar fi „Othello: From Beginner to Master” de Randy Fang [7].

[3]Din lb. germană, s-ar traduce prin cuvântul „forțat”.
[4]Voi mai folosi și termenul în engleză: end-game.”

Capitolul 3

Fundamente teoretice pentru jocul Reversi

3.1 Noțiuni de Teoria Jocurilor

Teoria jocurilor este o ramură a matematicii aplicate, care se ocupă cu studiul situațiilor competitive. Problemele de interes implică participanți multipli, fiecare având obiective independente referitoare la o anumită resursă sau un anumit sistem, iar succesul unui agent implicat în joc depinde și de alegerea celorlalți agenți. Teoria jocurilor descrie un limbaj care formalizează matematic, analizează și structurează situațiile competitive.[9]

Pentru că teoria jocurilor studiază scenariile competitive, problemele se numesc *jocuri* iar agenții se numesc *jucători*[1]. În jocuri, acțiunile jucătorilor se numesc *mutări*, iar rolul analizei este de a identifica secvența de mutări ce trebuie *jucată*. Secvența de mutări a unui jucător, se numește *strategie*, iar o *strategie optimă* este o secvența de mutări care duce la cel mai bun rezultat. O strategie optimă nu este neaparat unică, pot exista mai multe strategii care să aibă același rezultat și toate sunt optime, atâta timp cât nici o altă strategie nu aduce un rezultat mai bun.
Una dintre ipoteze în teoria jocurilor, este că jucătorii sunt *rționali*. Un jucător rațional va alege întotdeauna o mutare care îi va aduce cel mai mare venit, care în mod rațional este cea mai bună alegere pentru el, iar mulțimea mutărilor alese formează o strategie optimă. Ne putem gândi la venit ca fiind o *funcție de utilitate*, care diferă de la joc la joc. O funcție de utilitate este o corespondență între o stare curentă a jocului și un număr real, iar valoarea este interpretată ca nivelul de mulțumire al jucătorului. Considerăm A fiind mulțimea stărilor unui joc oarecare, $\forall a, b \in A$, funcția $u : A \to \mathbb{R}$ este o funcție de utilitate, atunci [10]:

$$u_i(a) > u_i(b) \qquad \text{dacă și numai dacă jucătorul } i \text{ preferă } a \text{ în favoarea lui } b. \tag{3.1}$$

În jocuri, se folosește termenul *cunoștințe comune*[9]. Cunoștințele comune sunt informații deținute de către toți participanții la joc. Raționalitatea ad-

[1]În continuare voi folosi aceste două denumiri.

versarilor este o cunoștință comună, precum și structura jocului și regulile jocului.[2] În aceste condiții, scopul analizei este de a prezice modul în care jocul va fi jucat de către jucători raționali, sau de a găsi cea mai bună strategie împotriva jucătorilor raționali.
Un important criteriu de împărțire a jocurilor, este modul în care se desfășoară ele în timp. Există două mari clase: *Jocuri simultane*, în care jucătorii acționează simultan[3] (în această clasă intră majoritatea jocurilor care modelează situații competitive reale din diferite domenii) și *jocuri secvențiale* în care jucătorii acționează secvențial, după o anumită regulă. În continuare mă voi referi la jocurile secvențiale, ele fiind fundamentul teoretic al acestei lucrări.
Pentru reprezentarea unui joc secvențial, se folosește *forma extinsă*. Forma extinsă presupune desenarea unui arbore, în care nodurile sunt stări ale jocului și este o descriere completă asupra evoluției jocului. Descrierea pornește de la rădăcina arborelui, care este starea inițială a jocului, iar fiii sunt stările viitoare ale jocului. Un nod va avea un număr egal de fii cu numărul de mutări permise de regulile jocului pentru jucătorul care e la rând în starea curentă a jocului. Nodurile terminale ale arborelui, reprezintă *stări finale* ale jocului. O stare finală a jocului, presupune că jocul s-a terminat și nu există nici o tranziție de la starea finală la altă stare. [1]
În continuare vom defini proprietăți ale jocurilor, care sunt necesare pentru această lucrare.

3.1.1 Informația perfectă

Această proprietate împarte jocurile în două mulțimi disjuncte [2]: Jocurile cu *informație perfectă* și jocurile cu *informație imperfectă*. Informație perfectă presupune că în orice moment t_0 în timpul jocului, fiecare jucător are acces la toate informațiile definitorii ale jocului. Informații definitorii înseamnă că: în orice moment t_0 fiecare jucător cunoaște toate mutările ce au avut loc până în momentul t_0. În această categorie de jocuri, intră jocuri precum șah-ul, go, reversi, table, etc. La șah de exemplu, fiecare jucător vede în orice moment tabla de joc și cunoaște mutările anterioare, piesele ce au fost capturate, poziția pieselor pe tablă. Singurul lucru necunoscut ramâne strategia adversarului. Categoria de jocuri cu *informație imperfectă* cuprinde jocuri precum poker, unde un jucător nu cunoaște cărțile adversarului. Jocurile care prezintă un element nedeterminist, precum aruncarea unor zaruri, intră în categoria jocurilor cu informație perfectă cu *șansă*. Elementul nedeterminist poate fi considerat ca făcând parte din regulile jocului, o funcție care în orice moment din joc va genera mulțimea mutărilor posibile. Acest tip de jocuri se mai numesc și jocuri cu *informație incompletă*.

Strategia optimă

Strategia optimă într-un joc cu informație perfectă, întotdeauna constă într-o *strategie pură*[2]. O strategie pură este o definiție completă, deterministă,

[2]Ideea de raționalitate poate fi relaxată, de exemplu în psihologie, unde scopul este de a studia comportamentul uman în situații competitive.

[3]Simultan se referă la faptul că un jucător nu așteaptă ca un alt jucător să facă o mutare. Toți jucătorii vor acționa în momentul în care se decid, neexistând o regulă

a strategiei alese de către un jucător. Mulțimea strategiilor unui jucător, în acest tip de joc, este formată din strategii pure. O *strategie mixtă* implică nedeterminism și asignează fiecărei strategii pure existente o probabilitate, iar jucătorul va selecta aleator una dintre ele fiind luate în calcul probabilitățile. Probabilitatea fiind un element continuu, vor exista o infinitate de strategii mixte pentru un joc, chiar dacă setul de strategii pure este finit. În cazul unui joc cu informație imperfectă, jocul optim va necesita o strategie mixtă.

Evaluarea stării jocului

Pentru a restrânge și mai mult categoria în care se află jocul studiat, Reversi intră în categoria jocurilor de *două persoane* cu *sumă zero*[4]. Un joc se spune că este de tipul sumă-zero, dacă suma utilităților jucătorilor referitor la o stare de joc, este zero. În acest tip de jocuri, interesele jucătorilor sunt diametral opuse, câștigul unui jucător va însemna pierderea celuilalt jucător. Șah, X și 0 sunt astfel de jocuri unde rezultatul este câștig, remiză, pierdere. Reversi fiind un joc cu scor, intră în aceași categorie, punctele acumulate de un jucător sunt pierderea celuilalt jucător.

Joc perfect

Noțiunea de joc perfect înseamnă strategia unui jucător care va duce la cel mai bun rezultat pentru acel jucător, indiferent de strategia adversarului. Dacă de exemplu s-a determinat că prin joc perfect, pornind din starea s_0 a jocului, jucătorul p_i ajunge la remiză, atunci rezultatul va fi o remiză sau un câștig, niciodată nu va pierde pentru că determinarea jocului perfect ia în calcul jocul perfect și din partea adversarului.

3.1.2 Istoric

Analiza jocurilor cu doi jucători cu informație perfectă, este doar un domeniu de aplicare a teoriei jocurilor. Sunt foarte multe domenii în care se aplică aceste fundamente matematice, multe fiind mai complicate unde jucătorii sunt mai mult de doi și mută simultan, de exemplu. Această știință s-a dezvoltat și a devenit un domeniu de studiu odată cu publicațiile din 1927 ale lui John von Neumann, fiind considerat părintele teoriei jocurilor.[5] În 1927 el a enunțat teoria minmax, în lucrarea denumită *„Zur Theorie der Gesellschaftsspiele"*, care stă la baza rezolvării multor jocuri de azi. Teoria jocurilor ocupându-se cu studiul situaților competitive la modul general, se poate aplica în domenii precum: economie, politică, natură, filosofie, psihologie, biologie, etc. Jucătorii pot deveni firme competitive, iar regulile jocului pot deveni regulile pieței, candidații politici fiind în competiție pentru voturi, ofertanții fiind în competiție pentru o licitație. Înarmarea nucleară a țărilor este tot o situație competitivă, unde e mai bine să ai decât să nu pentru că nu cunoști alegerea adversarului, problemă asemănatoare cu *Dilema Prizionerilor*, cu toate că soluția cea mai bună ar fi

[4] În engl. zero-sum game

[5] Primele discuții despre ceea ce numim azi teoria jocurilor, au avut loc în anul 1713, când James Waldegrave a scris o scrisoare în care a prezentat ceea ce azi numim algoritmul minmax cu strategie mixtă pentru jocul de cărți „le Her".

ca nimeni să nu aibă. Toate aceste situații necesită gândire strategică, folosind informațiile existente pentru a concepe cel mai bun plan pentru a-și împlini fiecare jucător obiectivul, care nu înseamnă neaparat a câștiga. O altă piatră de temelie în teoria jocurilor, a fost publicarea de către John von Neumann și Oscar Morgenstern a lucrării *„Theory of Games and Economic Behavior"*. Această scriere a dus la o intensă aplicare a cunoștințelor de teoria jocurilor în economie. În 1950, John Nash a demosntrat că jocurile finite întotdeauna au un punct de echilibru, unde nici un jucător nu își va putea îmbunătăți rezultatul dacă va devia de la strategia aleasă, demonstrație care i-a adus și un premiu Nobel. Multe probleme și analize ale situaților competitive, presupun rezolvarea unei astfel de probleme, adică găsirea unui echilibru Nash. Jocruile nu sunt neaparat de tipul sumă-zero, precum jocurile de societate, iar găsirea unui echilibru Nash devine o problemă complicată. Analiza jocurilor cu sumă-zero și cu informație perfectă prin explorarea arborelui de joc printr-o metodă minmax, este un echilibru Nash pentru că nici un jucător nu poate devia de la strategia aleasă pentru a obține un rezultat mai bun, pentru că minmax ia în considerare un joc perfect din partea jucătorilor. În rezolvarea jocului Reversi, soluția minmax este echivalentă cu echilibrul Nash.[1, 9, 10]

3.2 Teoria jocurilor combinatorială

Teoria jocurilor combinatorială, este o teorie matematică ce se ocupă cu studiul jocurilor cu doi jucători, unde jucătorii joacă pe ture, adică secvențial, cu scopul de a ajunge la o situație de câștig. Această teorie se rezumă la jocurile cu informație perfectă, precum Reversi, Șah, Go și nu acoperă jocurile cu informație imperfectă precum Poker, Table, acestea din urmă fiind studiul pentru teoria jocurilor clasică.

3.3 Arbori AND-OR

Un arbore AND-OR poate să conțină trei tipuri de noduri: noduri *terminale*, noduri *AND* și noduri *OR* [11]. Un nod *OR* reprezintă o decizie luată de cel ce rezolvă problema, alegerea fiind făcută după o anumită logică, cu scopul de a avansa în spațiul stărilor.
Un nod *AND* reprezintă anumite evenimente ce sunt independente în raport cu cel care rezolvă problema. Grafic se deosebește un nod *AND* de un nod *OR* prin marcarea arcelor ce ies din nod cu un arc de cerc.

Un nod terminal este fie un nod *soluție*, fie un nod *eșec*. Un nod soluție reprezintă o rezolvare posibilă a problei, iar un nod eșec reprezintă un nod în care problema nu e rezolvabilă. Nodurile AND și OR au succesori, iar nodurile terminale sunt noduri frunză, fără succesori.
Soluția unei probleme al cărei spațiu de stări este un arbore AND-OR, nu este un simplu nod soluție ci este un lanț AND-OR din arbore. Soluția problemei și un nod asociat unei stări soluție, sunt concepte diferite [11]. O soluție este o strategie de rezolvare a problemei, strategie care arată metoda de rezolvare a problemei în prezența unor factori independenți de rezolvator. Arborele poate fi construit înaintea începerii rezolvării problemei, dar soluția va fi determinată

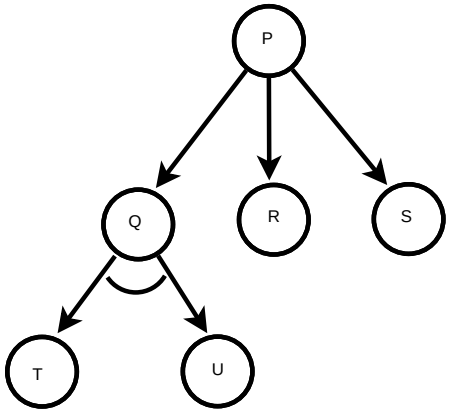

Figura 3.1: Exemplu arbore AND-OR.

în momentul explorării lui.
Pentru explorare, există diferite strategii, arborelee AND-OR oferă doar o reprezentare a stărilor problemei.

3.4 Arbori de joc

În teoria jocurilor combinatorială, un arbore de joc este un arbore AND-OR, unde nodurile sunt stări ale jocului iar arcele sunt mutări posibile dintr-o stare în alta. Arborele complet este format din poziția inițială, fiind rădăcina arborelui și toate stările posibile ale jocului, reprezentate prin noduri. Nodurile frunză vor reprezenta sfârșiturile posibile ale jocului.
Arborii de joc sunt un concept important în AI, pentru că o metodă de a alege cea mai bună mutare este de a căută în spațiul stărilor, adică în arbore. Pentru jocuri triviale precum „X și 0" căutarea se poate face cu resurse rezonabile și timp rezonabil, dar când e vorba de un joc mai complex precum Reversi sau Șah, căutarea este imposibilă în timp rezonabil. Pentru astfel de jocuri, apare conceptul de arbore de joc parțial, care va avea atâtea niveluri încât căutarea să se încadreze într-o limită de timp. În afară de jocurile patologice [12] care se pare că sunt rare, căutarea mai adâncă în arbore va genera un rezultat mai bun.

3.5 Numerotarea nodurilor

O posibilă numerotare a arborilor este descrisă în [13] și se numește sistemul decimal Dewey.

De la rădăcină, nodurile într-un arbore de căutare se notează cu o secvență de întregi de forma $a.b.c\cdots$. Rădăcina este considerată o secvență goală, iar în rest, pentru orice nod, pornind de la rădăcină se vor lua pe rând ramurile a, apoi de la nivelul următor b, apoi c, etc.

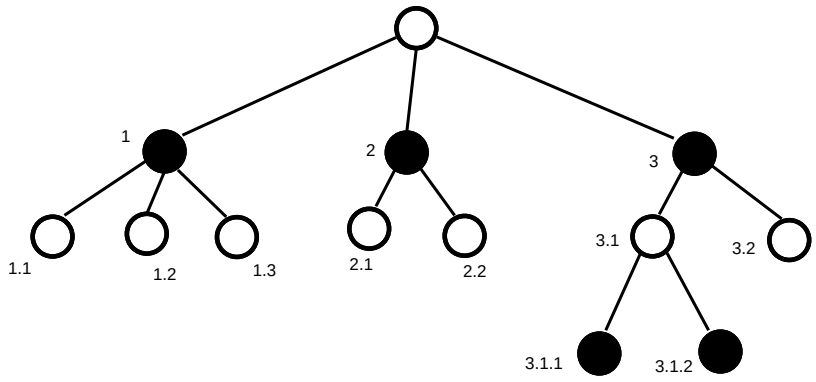

Figura 3.2: Numerotare Dewey.

3.6 Metode de rezolvare a jocurilor

Aici voi prezenta diferite metode de rezolvare a jocurilor la care mă refer în această lucrare, adică jocuri cu informație perfectă cu doi jucători și sumă-zero.

3.6.1 Metode bazate pe forță brută

Metodele bazate pe forță brută sunt cele mai folosite în rezolvarea jocurilor. Motivul este că această metodă este singura care poate alege cea mai bună mutare la un moment dat, având în vedere că nu se cunoaște un model matematic pentru joc. Cele mai multe rezolvatoare de jocuri folosesc metode precum $\alpha - \beta$ sau metode îmbunătățite într-un fel sau altul.

Analiza retrogradă

Analiza retrogradă[6] este o metodă în care pentru orice poziție a unui joc specific, sau unui joc din partea de sfârșit [7], numărul de mutări până la cel mai bun câștig posibil, este memorat [14]. De exemplu în jocul de șah, dintr-o anumită poziție, mutările care le va face primul jucător luând în vedere joc perfect din partea adversarului, sunt memorate. Se construiește o bază de date pornind de la nodurile terminale și mergând înapoi în arbore.
După ce baza de date a fost construită, jocul perfect este garantat. Jucătorul MAX va alege mutările cu cel mai scurt drum spre o poziție de mat, iar jucătorul MIN va alege mutările cu drumul cel mai lung spre mat. În ziua de azi această tehnică este des folosită în partea de *end-game* [14]. De exemplu, pentru jocul Șah, au fost analizate toate posibilitățile din *end-game* cu maxim 6 piese de joc pe tablă. Un exemplu extrem, arată că cea mai lungă poziție care în final duce la mat, dacă amândoi jucători joacă perfect, necesită 262 de mutări [14]. Acest exemplu a arătat că cel mai complicat lucru nu este de a genera aceste poziții, ci de a le înțelege și de a extrage din ele anumite reguli, informații ce să poată apoi fi asimilate de creierul uman.

[6] În engl. retrograde analysis.
[7] În engl. end-game.

3.6.2 Metode bazate pe bază de cunoștiințe

Ca o adăugare la metodele bazate pe forță brută, este de multe ori benefică incorporarea unei metode de rezolvare care implică o bază de cunoștiințe. Principalul lor avantaj este de a oferi o ordonare a mutărilor ce urmează a fi explorate [14].

3.7 Explorarea MINMAX

Algoritmii de explorare a grafurilor (în cazul particular, al arborilor) au o variantă particulară numită explorare MINMAX pentru navigarea în spațiul stărilor a jocurilor prezentate. Algoritmul fiind enunțat prima dată de către John von Neumann, a fost folosit în forma actuală de către Turing în 1950, și Shannon [3]. Cei doi jucători sunt numiți convențional *MIN* și *MAX*. Jucătorul MIN încearcă să minimizeze câștigul lui MAX, adică să-i maximizeze pierderea, iar jucătorul MAX va încerca să-și maximizeze câștigul, adică să-și minimizeze pierderea. Fiind vorba de jocuri cu sumă zero, înseamnă că MAX maximizându-și câștigul, va maximiza și pierderea lui MIN, rolul se inversează pentru MIN. Scopul algoritmului este de a determina cea mai bună mutare ce poate fi efectuată de jucătorul MAX în starea curentă a jocului. Amândoi jucători sunt considerați raționali (vezi 3.1) și vor juca perfect. Orice abatere a lui MIN de la jocul perfect, este în câștigul lui MAX și orice abatere de la jocul perfect a lui MAX, este în favoarea lui MIN.
Spațiul stărilor jocului este un arbore AND-OR numit și arbore MIN-MAX, sau cel mai comun, arbore de joc. Nodul rădăcină este considerat nod MAX, deci toate nodurile OR sunt considerate noduri MAX și toate nodurile AND sunt considerate noduri MIN. Rădăcina P este considerată problema lui MAX, rezolvarea ei înseamnă rezolvarea problemei lui MAX, adică de a găsi cea mai bună mutare. Nodurile terminale desemnează câștigul (eventual pierderea)[8] lui MAX. Arcele ce ies din noduri reprezintă mutări posibile, MAX va încerca să găsească o strategie ca să ajungă într-un nod terminal cu câștig maxim, iar MIN contrariul lui MAX. Dacă spațiul stărilor poate fi parcurs în totalitate, atunci strategia este completă [11]. Dacă spațiul stărilor este prea mare, atunci arborele este explorat până la o adâncime limitată, pornind de la nodul MAX curent, va rezulta o strategie incompletă formată doar din câteva mutări în avans. Nodurile devenite terminale prin limitarea căutării, vor fi considerate ca fiind noduri finale, iar un evaluator va spune care dintre jucători este în câștig. În acest caz, bineînțeles, rezultatul jocului este incert, nu se mai poate vorbi de un joc perfect din partea lui MAX decât în cazul în care estimarea scorului este perfectă. Rezultatul va depinde deci de jocul lui MIN. La începutul unui joc, problema P este starea inițială a jocului, iar pe parcursul lui, problema P devine starea curentă a jocului. Mutările sunt irevocabile, putem spune deci că din punctul de vedere al algoritmului MINMAX, este neimportant ce s-a întâmplat în trecut.
Valoarea minmax a unui nod a fost enunțată de către David McAllester în felul următor [15]:

[8] Mai corect spus: utilitatea.

Definiția 1. Valoarea minmax pentru nodul de pe nivelul 0 este definit ca fiind o valoare statică a acelui nod. Valoarea minmax de pe nivelul d a unui nod max, este maximul dintre valorile de pe nivelul $d-1$ a fiilor lui. Valoarea minmax de pe nivelul d a unui nod min, este minimul dintre valorile de pe nivelul $d-1$ a fiilor lui.

Pseudocod 3.1 Algoritmul MINMAX

```
funcție MINMAX(s, adâncime)
{
....returnează MAX(s, adâncime)
}

funcție MAX(s, adâncime)
{
....Dacă succ(s) = {∅} SAU adâncime = 0:
........returnează valoarea lui s

....v = −∞
....pentru_fiecare nod ∈ succ(s):
........v := max(v, MIN(nod, adâncime - 1))
....returnează v
}

funcție MIN(s, adâncime)
{
....Dacă succ(s) = {∅} SAU adâncime = 0:
........returnează valoarea lui s

....v = ∞
....pentru_fiecare nod ∈ succ(s):
........v := min(v, MAX(nod, adâncime - 1))
....returnează v
}
```

Algoritmul are complexitatea exponențială $O(b^d)$ în raport cu înălțimea d a arborelui MINMAX și cu numărul b de arce ce ies din noduri, fiind inutilizabil pentru un număr mare de stări [2, 11, 14, 16]. Ca o metodă de compromis, se poate limita adâncimea de explorare, adică pe d și folosirea unei funcții euristice h pentru evaluarea nodurilor neexpandate. Bineînțeles, dacă euristica h nu este perfectă, atunci algoritmul nu va fi optim.
Este posibil, ca o mutare mai puțin bună a lui MAX să poată duce într-o stare viitoare dincolo de orizontul de explorare, fiind o stare de unde jucătorul MAX nu va mai avea scăpare. Acest fenomen este denumit „efect de orizont” [11] și nu are încă soluție.

Jocurile cu șansă

Un caz aparte, este cel al jocurilor cu șansă. Și aceste jocuri se pot transpune într-un arbore de joc, unde sunt adăugate noduri suplimentare, numite no-

duri șansă [11]. Nodurile șansă au o probabilitate și corespund momentului aruncării zarului de exemplu. Fiecare nod șansă va avea o probabilitate, ce reprezintă probabilitatea ca acea combinație de zaruri să se nimerească. Evident, algoritmul de explorare nu poate garanta optimalitatea soluției într-un joc bazat pe șansă [11].

3.8 Tăiere $\alpha\beta$

Algoritmul $\alpha - \beta$ a fost descoperit individual de mai multe persoane [17]. Donald Knuth și Ronald W. Moore au rafinat algoritmul în anul 1975 enunțând și demonstrația lui.
Minmax este un algoritm de forță brută, nici o optimizare nu se face asupra căutării. Complexitatea lui fiind exponențială, îl face de nefolosit în forma actuală. Exponentul d nu e tocmai mic în jocuri, 58 pentru Reversi (vezi 4.3). O optimizare evidentă ar fi să nu explorăm toate ramurile, dar cum se va face selecția? S-a observat că anumite noduri nu contribuie la valoarea minmax a problemei, deci nu au de ce să fie explorate. Algoritmul $\alpha - \beta$ face parte din algoritmii de tip *branch-and-bound*. Sunt folosite două limite în procesul de căutare, α și β, ele fiind transmise în arbore în momentul explorării lui în adâncime. Perechea $\alpha - \beta$ se mai numește și fereastră de căutare, pentru că în orice nod, α reprezintă cea mai mică valoare ce poate afecta valoarea minmax deasupra nodului curent în arbore, iar β reprezintă cea mai mare valoare ce poate afecta valoarea minmax. Algoritmul $\alpha - \beta$ se folosește de nodurile explorate pentru a decide tăierea altor noduri. Dacă de exemplu știm sigur că valoarea determinată până acum este mai mult de 10, iar căutarea curentă a unei ramuri până acum arată ca poate returna un scor de cel mult 9, atunci nu mai este nici un motiv pentru a continua căutarea pe acea ramură.

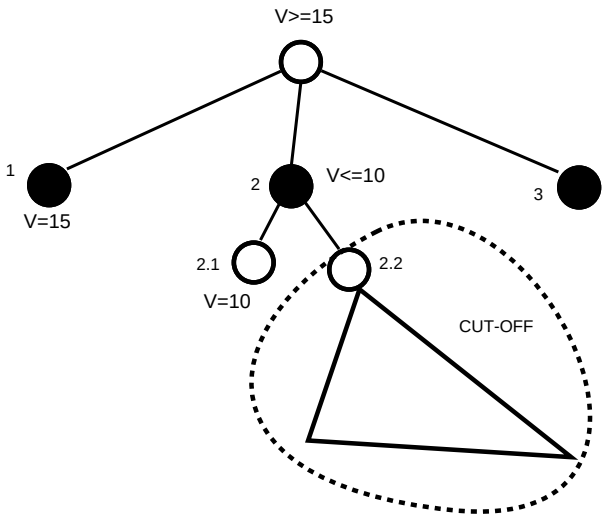

Figura 3.3: Tăiere α.

Presupunem (exemplificare în figura 3.3) că prima ramură a fost deja explorată, iar valoarea nodului 1 este 15. Reamintim că la rădăcina arborelui este jucătorul MAX, care va maximiza câștigul. Înseamnă că pentru a se

schimba valoarea nodului rădăcină, va trebui ca valoarea nodului de pe ramura următoare să fie mai mare decât valoarea nodului 1, adică 15. Vom explora în continuare nodul 2, pentru că încă nu cunoaștem suficient din arbore. Presupunem că nodul 2.1 returnează o valoare $< limita$, să zicem 10. Nu este nevoie să evaluăm în continuare alt nod succesor nodului doi. Motivul, este că acum ne aflăm la un nivel MIN, unde se va minimiza câștigul, deci cea mai bună alegere a jucătorului MIN până în acest moment este 10, iar o valoare mai bună pentru el trebuie să fie mai mică ca 10. Dar cea mai buna valoare posibilă pentru nodul 2 este $< limita$, deci nici nod succesor a lui 2 nu va putea schimba valoarea minmax pentru rădăcină. Deci, în continuare pentru a explora nodul 3, se va considera prima ramură ca fiind cea mai bună soluție.

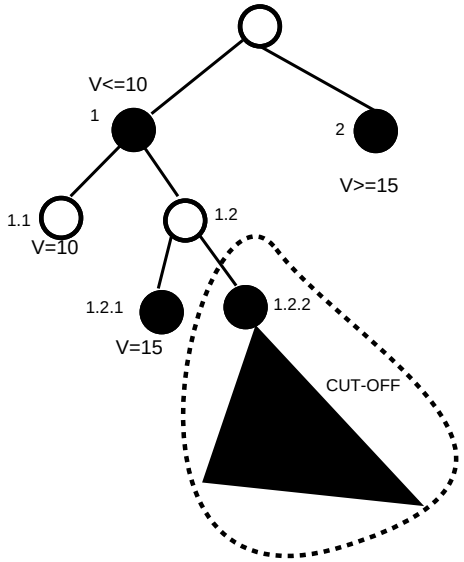

Figura 3.4: Tăiere β.

În cazul unui nod MIN, avem (figura 3.4). Presupunem că s-a explorat nodul 1.1 a nodului MIN 1 și s-a obținut cea mai bună valoare, 10 ca și limită. Algoritmul va explora în continuare nodul 1.2 și presupunem că nodul 1.2.1 va returna o valoare $> limita$, de exemplu 15. Nu mai este nevoie să explorăm nodul 1.2.2 și nici orice alt nod care ar mai exista ca descendent a lui 1.2, pentru că deja cea mai bună valoare a lui 1.2 este 15 care e $> limita$. Prin urmare, vom lua prima ramură ca fiind cea mai bună soluție curentă [13]. În timpul căutării, pentru un nod MAX, valoarea curentă cea mai bună este cel puțin α, iar pentru un nod MIN, valoarea curentă cea mai bună este cel puțin β.

O enunțare a algoritmului $\alpha - \beta$, o adaptare după [18].

Folosind tehnicile descrise, tăierea $\alpha - \beta$ poate fi folosită pentru a computa valoarea minmax a unui arbore de joc într-un timp $O(\sqrt{b^d})$ față de $O(b^d)$ cât e la minmax. Diferența este foarte mare, având în vedere că de exemplu la Reversi, factorul $d = 58$. Un alt mod de a privi [15], este că algoritmul $\alpha - \beta$ permite o explorare a unui număr dublu de noduri în față de minmax.
Algoritmul este corect în momentul în care se pornește explorarea cu fereastra

Pseudocod 3.2 Algoritmul MINMAX cu $\alpha\beta$

```
funcție MINMAXαβ(s, adâncime, α, β)
{
....returnează MAX(s, adâncime, α, β)
}

funcție MAX(s, adâncime, α, β)
{
....Dacă succ(s) = {∅} SAU adâncime = 0:
.....returnează valoarea lui s

....v = −∞
....pentru_fiecare nod ∈ succ(s):
.....v := max(v, MIN(nod, adâncime - 1, α, β))
.....Dacă v ≥ β:
.....returnează v
.....α := max(α, v)
....returnează v
}

funcție MIN(s, adâncime, α, β)
{
.... La fel ca la MAX, dar cu rolurile α, β inversate.
}
```

($\alpha = -\infty, \beta = \infty$). Domonstrația corectitudinii algoritmului, putem găsi în [11, 15, 13].
Se observă că același arbore, dar cu ordonări diferite ale tăierii $\alpha-\beta$. In cazuri diferite de ordonare, se vor explora un număr diferit de noduri pentru a calcula valoarea minmax. Se crede că dacă pornind de la un nod, se vor explora mai întâi ramurile cu cea mai bună *utilitate*, atunci și numărul de tăieri va fi maxim, deci explorarea ar dura mai puțin. Asta ar însemna ca pentru un nod MAX să evaluăm prima dată ramura cu valoarea cea mai mare, iar într-un nod MIN, să evaluăm mai întâi ramura cu valoarea cea mai mică.
În [13], avem o analiză pentru cazul optim de ordonare a arborelui pentru a atinge limita de $\sqrt{b^d}$. Tot în [13], se ajunge la concluzia că ordonarea perfectă (adică ordonare după utilitatea fiecărui nod) nu e neaparat cea mai bună, cum s-ar crede intuitiv, iar $\alpha - \beta$ ar avea un timp optim pentru o permutare a arborelui, care nu e chiar evidentă.
Complexitatea algoritmului este deci $\Omega(\sqrt{b^d})$, având în vedere că limita este atinsă doar în cazul unei ordonări optime.
În cazul în care arborele de joc ar fi o orodnat aleator, complexitatea explorării $\alpha - \beta$ ar fi $\Theta((\frac{b}{log_2 b})^d)$ pentru $b > 1000$, iar pentru valori moderate ale lui b, complexitatea este $O(b^{\frac{3 \cdot d}{4}})$, ceea ce permite mărirea cu 25% a adâncimii de explorare față de algoritmul MINMAX, la același număr de noduri explorate [11].

3.9 Îmbunătățiri ale algoritmului $\alpha\beta$

Cu toate că $\alpha\beta$ oferă un timp de explorare mult îmbunătățit față de MINMAX și prin asta devenind algoritmul standard de explorare în spațiul stărilor, un timp de explorare $b^{\frac{d}{2}}$ nu este suficient. În 3.8 am discutat despre importanța ordonării mutărilor înainte de explorare. S-au dezvoltat o serie de algoritmi bazați pe această observație, vom descrie pe scurt câțiva.

3.9.1 Adâncire iterativă

Idea acestui algoritm[9] este de a explora la o adâncime k înainte de a explora la o adâncime d, unde $k < d$. Dupa ce s-a atins nivelul k, se forțează o evaluare a stărilor obținute, după care se reia o explorare până la nivelul $k + s$. În practică, de cele mai multe ori, valorile k și s sunt egale cu 1. Astfel, se va explora doar un nivel, după care se va explora 2 niveluri, etc. Motivul, este de a folosi informația acumulată la explorarea până la nivelul k, pentru a genera o bună ordonare a mutărilor înainte de a explora la nivelul $k + 1$. Astfel, s-a arătat în practică [17] că se va genera un arbore puternic-ordonat, deci algoritmul $\alpha\beta$ va tinde spre optim. Metoda are avantaje și în competiții, unde timpul de gândire pentru o mutare este fix. Din moment ce nu se poate prezice când se va opri algoritmul $\alpha\beta$, ar putea rezulta o mutare catastrofică. Cu o adâncire iterativă, avem tot timpul rezultatul de la pasul anterior, care poate fi luat în considerare în cazul în care timpul a expirat iar explorarea până la nivelul d nu s-a terminat. În practică, numărul de noduri explorate la adâncimea d, folosind această tehnică, este semnificativ mai mic comparat cu o explorare standard până la nivelul d.
Dacă euristica construită va fluctua puțin la diferențe mici de adâncire, atunci într-o explorare iterativă, am putea îngusta fereastra de căutare, din nou optimizând algoritmul.

3.9.2 Tehnici bazate pe ordonarea mutărilor

Ordonarea statică

Prima metodă de ordonare a mutărilor pentru a obține o îmbunătățire a algoritmului $\alpha\beta$ presupune ordonarea lor după valoarea nodurilor succesoare. Nodul care a generat valoarea cea mai mare este explorat primul, iar nodul care a generat valoarea cea mai mică, este pus la coadă. De exemplu în Reversi, implementat și în FPGA, am putea considera o ordonare bună mutările în care se adaugă un disc în colț, pentru că devine stabil[10]. Ordonarea statică a funcționat bine [17] pentru anumite jocuri, chiar și Othello, dar nu a funcționat bine în Șah. Problema este că metoda este statică și nu se folosește de informațiile adunate pe parcurs.

[9] În engl. iterative deeping.
[10] Metodă cunoscută sub numele de *killer response*.

Tabele de transpoziție

Informații specifice pentru o căutare ar putea fi salvate în tabele de transpoziție. În algoritmul $\alpha\beta$ avem diferite informații despre un nod, precum cel mai bun scor, cea mai bună mutare de la acea poziție, adâncimea la care a fost căutat. Toate aceste informații ar putea fi memorate în tabele de transpoziție. Aceste tabele sunt în mod normal implementate ca tabele *hash*, cu funcții de *hash* ușor de implementat. Dimensiunea tabelei se poate alege după resursele disponibile. Acest tip de tabele pot fi apoi folosite pentru a ordona mutările sau a găsi duplicate de stări de joc [17]. În timpul jocului, o anumită stare de joc poate fi întâlnită prin selectarea mai multor strategii diferite. Astfel de poziții sunt și în Reversi. În acest joc, precum și în șah, s-ar putea exploata si mai mult, pentru că putem avea simetrii pe tabla de joc. Având tabela de transpoziție, putem căuta acea stare de joc în tabelă într-un timp $O(1)$. Dacă este găsită, atunci înseamnă că știm și valoarea minmax a acelui nod și nu mai avem motiv să continuăm explorarea.
Un mod de folosință și mai bun pentru aceste tabele, este în momentul în care se folosesc împreună cu o explorare iterativă. De obicei, rezultatul funcțiilor euristice de evaluare a stărilor de joc vor varia puțin de la nivelul k la nivelul $k+1$. Putem deci să pornim o explorare la nivelul $k+1$ folosind informația de la nivelul k. In tabelă, la indexul dat de nodul respectiv[11] se va afla mutarea cea mai bună. Astfel, o vom considera ca cea mai bună ordonare și explorarea va porni cu acea ramură.

3.9.3 Fereastră de căutare nulă

O îmbunătățire a algoritmului $\alpha\beta$ s-a arătat că se poate obține folosindu-ne de o căutare la o adâncime mai mică decât d. Dacă avem și o bună ordonare a mutărilor, atunci explorarea se va apropia și mai mult de cazul optim. Având în vedere că primul nod explorat e considerat ca fiind cel mai bun, înseamnă că restul le considerăm inferioare. Metoda ferestrei nule[12] se folosește de această presupunere. Presupunem că valoarea minmax obținută pentru prima mutare este *gamma*. În mod normal ar urma o explorare a nodului următor cu fereastra (*gamma*, beta). Dar cum prima mutare e considerată cea mai bună, algoritmul ferestrei nule va porni explorarea nodului următor cu fereastra (*gamma*, *gamma+1*). Dacă mutarea este într-adevăr inferioară, atunci rezultatul va cel mult egal cu *gamma* și nu mai trebuie făcut nimic. Dacă valoarea returnată va fi mai mare ca și *gamma*, atunci înseamnă ca mutarea e superioară și se va explora din nou cu o fereastră mai mare.
Bineînțeles, acest algoritm va da rezultate bune doar dacă avem o bună ordonare a mutărilor [17]. *NegaScout* este un algoritm ce folosește căutare cu fereastră nulă pentru a demonstra dacă mutarea care e testată este mai bună decât cea care a fost deja aleasă la o căutare normală. În practică, algoritmul NegaScout obține pentru jocul Șah, un plus de performanță de 10% față de o căutare cu fereastra completă.

[11] De fapt e indexul dat de funcția de *hash* aplicată nodului.
[12] În engl. *null window*.

3.9.4 Căutare selectivă

Am văzut cum un algoritm va căuta exhaustiv în spațiul stărilor pentru a determina o mutare bună. Dar se pune problema, cum sunt capabili oamenii de un joc bun? Bineînțeles, campionii de Reversi nu vor analiza toate mutările precum face un algoritm, fie el cu $\alpha\beta$. Răspunsul ar fi că mintea umană este cel mai bun evaluator existent și are capacitatea de a tăia anumite ramuri din arborele de joc fără a fi măcar explorate. Algoritmii prezentați până acum sunt corecți, valoarea determinată de ei poate fi demonstrată și nu poate fi alta.[13]

Algoritmul *Multi-ProbeCut* [19, 20] încearcă să facă exact acest lucru, de a tăia anumite ramuri înainte de a fi explorate. Diferența mare, este că $\alpha\beta$ standard va tăia ramuri după ce a explorat și a văzut că poate să facă acest lucru. MPC[14] va tăia ramuri înainte de a fi explorate. Procesul este probabilistic, un nod nu va mai fi explorat dacă probabilitatea de a fi o mutare proastă este suficient de mare. Ca mod de folosire, se va defini o adâncime $d' < d$ care este adâncimea la care se va face tăierea. Până la d', algoritmul va face o explorare $\alpha\beta$ obișnuită. Obținând valoarea minmax v pentru un nod, se va estima care ar putea fi valoarea v' dacă nodul ar fi explorat până la adâncimea d. Dacă valoarea estimată v pare o mutare proastă, atunci explorarea va înceta.

Această metodă presupune existența unui evaluator foarte bun, care să fie cât de cât uniform, să nu varieze brusc. Valorile d' și d se determină cel mai bine în practică. Rezultate bune se obțin și dacă se fac explorări selective multiple, de exemplu d', d'', d.

Pentru jocul Othello, MPC are rezultate remarcabile [20]. Algoritmul implementat cu MPC de către Michael Buro, a folosit doar 4% din timpul aceluiași algoritm fără MPC, amândoi fiind la fel de puternici. Dacă implementarea cu MPC este lăsată să exploreze un timp comparativ cu cel al implementării fără MPC, atunci cel fără MPC va câștiga aproximativ 78% din partide, pentru că va putea explora mai adânc.

[13] Dacă considerăm un evaluator perfect.

[14] Multi-ProbeCut.

Capitolul 4

Analiza jocului Reversi

Jocul Reversi este un joc în prezent nerezolvat. Pe o tablă 6x6 a fost rezolvat „ultra-ușor", al doilea jucător fiind câștigător. Reversi face parte din jocurile considerate campioane, alături de Șah și Go, care în prezent nu au rezolvare. Dintre acestea, Go este cel mai complex și se pare că nu va fi rezolvat încă zeci de ani de acum în colo [2, 14].

4.1 Rezolvarea jocurilor

Rezolvarea unui joc se referă în general la determinarea rezultatului în cazul în care ambii jucători joacă perfect. Se consideră 3 tipuri de rezolvare a jocurilor [2, 14].

Jocuri rezolvate *ultra-ușor*

Reprezintă acele jocuri pentru care s-a determinat rezultatul pentru starea inițială a jocului s_0.

Jocuri rezolvate *ușor*

Reprezintă acele jocuri, pentru care s-a determinat, pentru poziția inițială s_0, o strategie de joc pentru a câștiga *cel puțin* cu valoarea minmax, folosind resurse rezonabile.[1]

Jocuri rezolvate *puternic*

Reprezintă acele jocuri pentru care s-a determinat o strategie optimă de câștig pentru orice stare a jocului, folosind resurse rezonabile. De exemplu cazul jocului Nim demonstrat de Knuth.

4.2 Tipul jocului

Reversi/Othello face parte din clasa de jocuri cu informație perfectă cu doi jucători și sumă zero. Jocurile se mai împart în două mari categorii, necesare pentru determinarea complexității și analiza lor.

[1] În engl. game-theoretic value.

4.2.1 Jocurile convergente

Un joc este considerat convergent aunci când spațiul stărilor descrește odată cu înaintarea în joc. În general, jocurile convergente pornesc cu multe piese pe tabla de joc, iar în moment ce jocul progresează, adică înaintează în timp, piesele sunt treptat scoase de pe tabla de joc [14]. Acest aspect este foarte important, pentru că jocurile convergente pot avea bază de date pentru *end-game*.[2] Șah-ul de exemplu este un joc convergent. Fiind un joc convergent, spre sfârșitul jocului putem mult mai ușor cunoaște finalul. Jocul de Șah este rezolvat în momentul în care mai sunt 6 piese pe tabla de joc, oricare combinație ar fi.[3] În [14] avem descrisă poziția de Șah din 6 piese cu cel mai lung drum până la mat, dacă ambii jucători joacă perfect.[4]

4.2.2 Jocurile divergente

Jocurile divergente de obicei pornesc cu foarte puține piese, iar pe parcursul jocului piesele se adaugă pe tabla de joc, rezultând într-o explozie a spațiului stărilor spre final. Acest tip de jocuri sunt imune analizei retro sî a contrucției de baze de date cu jocurile de final, datorită numărului de combinații foarte mare [14]. În [2] avem o definiție formalizată matematic a convergenței/divergenței jocurilor, considerând n clase disjuncte, fiecare clasă formată din toate stările jocului formate din m piese. Reversi/Othello face parte din clasa jocurilor divergente, pentru că la fiecare mutare se adugă discuri pe tabla de joc, în final spațiul stărilor fiind în expansiune. În afară de faza de sfârșit a jocului, adăugând un disc pe tabla de joc, numărul de mutări legale este în creștere.

4.3 Complexitatea

Complexitatea în jocuri, determină două mărimi: complexitatea pentru spațiul stărilor și complexitatea arborelui de joc.

4.3.1 Complexitatea pentru spațiul stărilor

Complexitatea spațiului stărilor este definită ca numărul de stări posibile de joc la care se pot ajunge pornind dintr-o oarecare stare, de exemplu starea inițială s_0. De exemplu, pentru un joc ca și „X și 0”, această complexitate se poate determina ușor matematic. O limită superioară ar fi dacă calculăm 3^9, observând că fiecare pătrat poate conține un X sau un 0 sau să fie liber. Din acest număr se scad pozițile ilegale de joc și rezultă 5.478 stări de joc posibile [2].

4.3.2 Complexitatea arborelui de joc

Complexitatea arborelui de joc, este definită ca și numărul total al nodurilor frunză ce formează arborele de joc, pornind de la starea inițială s_0 a jocului.

[2]Din engl., faza de sfârșit a jocului.
[3]Cei doi regi sunt luați în calcul.
[4]Poziția necesită 262 de mutări până la mat.

De exemplu pentru un joc, J fiind starea inițială, dacă din J avem 20 de mutări posibile, iar apoi din fiecare mutare posibilă avem încă 30 de mutări posibile, înseamnă ca la cel mai jos nivel al arborelui am avea 600 de noduri terminale. Complexitatea arborelui de joc în acest exemplu este 600 [2]. Dacă dorim calcularea complexității arborelui de joc pentru un joc complicat precum Șah sau Reversi, este mult mai complicat și vom putea doar spune o limită superioară a acestuia. Complexitatea se aproximează folosind o medie a numărului de noduri ce pot porni dintr-un nod. De exemplu pentru jocul „X și 0”, avem o adâncime a arborelui de maxim 9 niveluri, iar la fiecare nivel i, factorul de lațime a arborelui este de $9 - i$, calculând complexitatea ne va da $9! = 362880$. Se observă că această complexitate este mai mare ca și spațiul stărilor, pentru că în arborele de joc se vor repeta multe stări ale jocului.

4.4 Determinarea complexității prin analiză „Monte Carlo”

O limită superioară pentru spațiul stărilor în jocul Reversi, ar fi 3^{64} adică aproximativ 10^{30}. Acest număr este obținut prin observația că tabla de joc are mărimea 8x8, adică 64, iar pe fiecare poziție se poate afla un disc negru, alb sau nimic. Bineînțeles, o parte din acest număr reprezintă stări de joc invalide, adică la care nu se poate ajunge prin nici o combinație. De exemplu pentru fiecare stare de joc trebuie să avem cele 4 poziții din mijloc ocupate, pentru că asta este starea inițială a jocului. O analiză Monte Carlo, ar presupune generarea unui număr de stări, de exemplu 100.000, complet aleatoare. Considerăm existența unei funcții $\pi(s)$ care are ca rezultat *adevărat* sau *fals*, indicând dacă starea de joc generată este validă sau nu. Funcția este construită după mai multe observații ale jocului precum și regulile lui. De exemplu nu putem avea grupuri de discuri care să nu aibă legătură între ele, deoarece în Othello trebuie adaugat un disc în vecinătatea unui alt disc. Victor Allis [2] a făcut o astfel de analiză și a arătat că pentru Reversi/Othello, complexitatea spațiului stărilor este de aproximativ 10^{28}.
Complexitatea arborelui de joc este aproximativ 10^{58}, 10 fiind considerat numărul mediu de mutări posibile la fiecare stare. Adâncimea arborelui, 58, a fost determinată prin analiza multor partide care în medie durează 58 de mutări. [2]

Capitolul 5

Placa de dezvoltare Spartan3E

5.1 Logică programabilă, FPGA

5.1.1 Scurtă descriere

Un FPGA[1] este un dispozitiv ce permite programarea expresiilor logice. Ca și capacitate, un FPGA este extrem de mare comparativ cu circuitele logice programabile de genul PLD, CPLD. În mod obișnuit, un FPGA este folosit pentru a construi un prototip hardware ce apoi va urma să fie implementat într-un ASIC[2]. Totuși, tot mai des FPGA-urile sunt folosite ca produs final, datorită flexibilității lor foarte mari.
La începutul anilor 1980, cele mai multe circuite logice erau implementate în tehnologie LSI[3]. Aceste circuite erau: microprocesoare, diferite *controlere*, etc. Bineînțeles, fiecare sistem avea nevoie de o logică suplimentară[4] pentru comunicarea între dispozitive. Pentru acest lucru erau construite circuite integrate specializate care îndeplineau funcția respectivă, dar procesul era foarte scump și inflexibil, logica diferind mult de la un produs finit la altul. Având această problemă, Xilinx, o fompanie de startup, a introdus în anul 1984 tehnologia FPGA ca o alternativă la circuitele integrate personalizate pentru implementarea logicii de alipire. Datorită programelor de tip CAD, astăzi circuitele logice pot fi implementate foarte ușor fără diferite procese fizice complicate. [21]

5.1.2 Arhitectura FPGA

Arhitectura de baza a FPGA-ului constă într-o matrice bidimensională de blocuri logice și bistabile. Omul va trebui să descrie logica pentru fiecare bloc, să descrie intrările/ieșirile blocurilor precum și conexiunile între blocuri.

Programarea

Tipul de programare folosit de FPGA-urile de la Xilinx (cele mai răspândite), este bazat pe memoria SRAM. Conexiunile FPGA se fac folosind porți de

[1] Field Programmable Gate Array.
[2] Application Specific Integrated Circuit.
[3] Large Scale of Integration.
[4] În engl. glue logic.

transmisie, multiplexoare, toate fiind controlate de celule SRAM. Singurul dezavantaj al acestei metode ar fi memoria SRAM care va ocupa cel mai mult. Un astfel de circuit va putea fi configurat de un număr infinit de ori. [21]

Look-Up Table

Felul în care funcțiile logice sunt implementate în blocurile logice este specific FPGA-urilor. Tabelele de LookUp sunt implementate ca o memorie, sau multiplexoare și memorie. O memorie de dimensiunea 2^nx1 poate implementa orice funcție logică cu n intrări. În practică, valorile folosite pentru n sunt $2, 3, 4, 5$. O tabelă n-LUT, adică implementată cu o memorie de mărimea 2^n, este implementarea directă a unei tabele de adevăr.

Bloc logic

Cel mai simplist, un bloc logic este implementat folosind o tabelă LUT cu 4 intrări, care va putea implementa o logică combinațională de 4 variabile și un bistabil care opțional va memora ieșirea funcției. Un FPGA este compuns dintr-o matrice foarte mare de astfel de blocuri logice.

Resurse FPGA Xilinx

Resursele de bază într-un FPGA de la Xilinx sunt:

- Blocuri logice configurabile, care vor conține logica combinațională și bistabile.
- Blocuri I/O care vor face legătura FPGA-ului cu exteriorul.
- PI, adică interconexiuni programabile.
- Blocuri de memorie RAM (BRAM).
- Alte resurse: circuite tri-state, circuite speciale pentru propagarea semnalului de ceas, etc.

Arhitecturile noi, chiar și Spartan3, pe lângă aceste resurse standard au în dotare și multiplicatoare hardware dedicate, DCM-uri [5], un circuit specializat cu care se poate modifica semnalul de ceas: multiplica, divide, defaza, etc. VirtexII are înclus chiar și un procesor PowerPC care poate fi folosit.

Slice-urile

Un CLB (Configurable Logic Block) dintr-un FPGA este compus din slice-uri. Spartan3 are de exemplu 4 slice-uri per CLB. Fiecare slice conține două LUT-uri și două elemente de memorare. Pe lângă operația de generator de funcții, un LUT poate fi folosit ca o memorie de 16x1 biți. Mai mult, cele două LUT-uri pot fi combinate pentru a forma o memorie de 16x2 biți sau 32x1 biți memorie sincronă, sau 16x1 biți memorie dual-port. LUT-urile pot fi folosite și ca registre de deplasare pentru capturarea datelor.

[5] Digital Clock Manager.

Elementele de stocare

Elementele de stocare dintr-un LUT pot fi folosite ca bistabile D sincrone, sau senzitive pe nivel.

Resurse aritmetice

FPGA-urile moderne au circuistică suplimentară pentru a optimiza operațiile aritmetice. De exemplu, linii dedicate pentru propagarea transportului la adunare.

5.1.3 BRAM

FPGA-urile Xilinx au incorporate memorii RAM denumite „Block RAM". Aceste memorii sunt organizate în coloane în chip. Placa ce o dețin, Spartan3E cu un FPGA XC3S500E are două coloane de BRAM a câte 10 blocuri fiecare, în total 20 de blocuri RAM care în total sunt 360K biți de memorie RAM incorporată în FPGA, foarte rapidă.[22]
Fizic, fiecare bloc de RAM are două porturi de acces complet independente, fiecare port suportă operația de citire și de scriere. Fiecare port este sincron cu propriul lui semnal de ceas și fiecare port deține semnalele lui de control. Astfel, memoria poate fi folosită ca și memorie dual port. Diferite programe de sintetizare, precum și XST[6] sunt capabile să infereze memorie BRAM pe baza descrierii hardware. Astfel, o descriere hardware care respectă regulile inferării de memorie block RAM, va duce la generarea unui anume *bitstream* care va folosi resursa de block RAM din FPGA. [22]

5.2 Spartan3E

Placa de dezvoltare „Spartan3E Starter Kit" conține un FPGA XC3S500E. Resursele FPGA-ului sunt [23]:

- 1164 blocuri logice.
- în total $4 \cdot 1164$ adică 4656 de slice-uri.
- 360K biți de memorie bloc RAM.
- 20 de multiplicatoare dedicate (două intrări a câte 18bit per multiplicator).
- 4 DCM-uri.

Placa de dezvoltare mai conține [24]:

- O memorie 4Mbit Platform Flash, memorie ROM configurabilă.
- un CPLD de 4 macrocelule, folosibil.
- 64MB DDR SDRAM, linia de date fiind de 16 biți, la o frecvență de 100+ Mhz.

[6] Xilinx Synthesis Tool.

- 16MB memorie Flash accesabilă paralel. (Poate memora configurația FPGA-ului)
- 16Mbit memorie Flash serială.
- Ecran LCD 2x16 caractere.
- interfață PS/2.
- interfață VGA.
- 10/100 Ethernet PHY.
- două interfețe RS232 9 pini.
- oscilator 50Mhz.
- patru convertoare digital-analogice.
- două convertoare analogice-digitale.
- LED-uri, butoane.

Capitolul 6

Proiectare hardware

6.1 HDL, Verilog, scurtă introducere

Proiectarea hardware a acestui proiect, a fost făcută în limbajul Verilog HDL [25]. Cu astfel de limbaje de descriere hardware, se poate descrie logica unor circuite complexe. Se pot scrie procese combinaționale și procese secvențiale. Ca și mod de gândire, toate procesele secvențaile se vor întâmpla în paralel, la apariția semnalului de sincronizare. Există atribuiri blocante și non-blocante. Ca și mod de gândire, cele non-blocante pot fi considerate că se întâmplă în paralel, iar cele blocante sunt practic o cascadare a nivelurilor logice, intrarea unui nivel fiind rezultatul unui nivel anterior. Un subset din aceste declarații în Verilog este sintetizabil. Asta înseamnă că un program de sinteză va putea transforma acea descriere hardware într-un *netlist*. Un netlist este o ierarhie de structuri logice, porți logice, interconexiuni între ele. Mai departe, acea structură de porți logice, se poate transforma într-o listă de interconexiuni pentru un FPGA. (vezi 5.1)

6.2 Paradigme de proiectare

Proiectarea hardware în Verilog se poate face la 4 niveluri de abstractizare.

- Comportamental, unde se va descrie efectiv comportamentul funcției logice.
- RTL[1] folosește registre conectate prin expresii booleene.
- La nivel de porți logice. Adică interconexiuni directe între primitive logice.
- La nivel de tranzistor. Tranzistoare MOS în interiorul porților.

În acest proiect, s-a proiectat comportamental și RTL.
Ca și în orice proiectare, sunt anumite paradigme, reguli de aur pentru o bună proiectare. O bună proiectare va permite ca circuitul să funcționeze la o frecvență mai mare. O proiectare proastă va funcționa la o frecvență mică, vor

[1] Register Transfer Level.

apărea erori de sincronizare, latențe prea mari și diferite între nivelurile logice, etc. [26, 25, 27] Cu toate că scopul meu a fost ca funcționarea acestui proiect să fie la ofrecvență de 50MHz, lucru care probabil nu este chiar greu de realizat, voi aminti aici câteva reguli care m-au ajutat foarte mult în proiectare:

6.2.1 Identificarea blocurilor critice

Este foarte important de a determina blocurile cu niveluri logice cascadate. Aceste blocuri vor încetini tot circuitul, datorită propagării semnalului. În general trebuie evitată construirea a astfel de blocuri. Pentru a scăpa de astfel de ierarhii, este recomandată introducerea stagiilor de *pipeline*, astfel încât circuitul să fie secvențial iar rezultatul să fie obținut în mai multe faze care vor fi memorate în registre. La fiecare ciclu de ceas se va calcula o fază a rezultatului, dar toate vor fi în paralel. În acest mod, la fiecare impuls de ceas vom avea un rezultat final. Astfel se poate crește frecvența circuitului, pentru că cea mai lungă cale de propagare va fi împărțită de registre. [26]

6.2.2 Memorarea ieșirilor

O altă regulă este de a memora ieșirile blocurilor, adică de a avea un nivel de registre la ieșiri în care să fie memorat rezultatul. Acest lucru va crește mult flexibilitatea și ușurința proiectării. În acest proiect fiecare bloc respectă această regulă. [28]

6.2.3 Semnale de ceas, reset

Semnalul de reset sincron

În FPGA este de recomandat utilizarea semnalului de reset sincron pentru module în loc de semnal de reset asincron. De exemplu, conform [28] un multiplicator implementat cu stagii de pipeline într-un FPGA Virtex4 și cu reset asincron, va funcționa la o frecvență de 200MHz, iar dacă descrierea hardware se modifică pentru a funcționa cu un reset sincron, circuitul va funcționa la 500MHz. Acest lucru se întâmplă pentru că blocurile de multiplicare sunt construite cu un reset sincron, iar dacă se folosește un reset asincron, atunci logica va trebui să folosească alte resurse din FPGA, totul devenind mai lent. Folosirea resetului sincron va reduce și numărul de slice-uri folosite (vezi 5.1).

Semnalul de sincronizare

O regulă de aur este folosirea unui semnal *enable* pentru a valida/invalida un modul în loc de compunerea semnalului de sincronizare cu alte semnale. Compunerea semnalului de sincronizare prin porți logice va duce la întârzieri a semnalului, rezultând probleme de sincronizare, etc.

6.2.4 Sumatoare arborescente sau înlănțuite

O problemă foarte importantă cu care m-am confruntat, a fost adunarea mai multor termeni. Un sumator de n termeni proiectat într-o structură liniară,

adică un lanț, va avea performanțe foarte scăzute pentru că rezultatul va apărea abia după o propagare după n niveluri. Dacă proiectăm sumatorul sub ca o structură arborescentă, complexitatea lui va fi doar logaritmică. Dar, un sumator arborescent va consuma mai multă putere și mai multe resurse.
Totuși, un lucru interesant [28], este că în familia de FPGA-uri Virtex4, sunt anumite blocuri speciale denumite DSP48, proiectate pentru prelucrarea datelor, care au o structură lanț, dar foarte optimizate și vor avea performanțe mult mai bune ca un sumator arborescent standard.

6.2.5 Evitarea latch-urilor

Este întotdeauna de dorit evitarea inferării de latch-uri. Latch-urile sunt sensibile pe palier, nu pe fron ca și bistabilele, generând multe probleme. Pentru a evita latch-urile proiectarea trebuie să fie atentă, petntru a nu necesita elemente de memorie în plus și fără voia noastră. De exemplu într-o structură *if* trebuie să exprimăm valoarea semnalului și în cazul în care nu ar fi satisfăcută condiția, altfel se va genera un latch pentru că trebuie memorată valoarea anterioară.

6.3 XST, Macro-uri

XST este programul de sintetizare de la Xilinx. Macro-urile sunt anumite descrieri funcționale recunoscute de XST pentru a genera o circuistică optimă pentru ele. În timpul rulării, XST va încerca să recunoască cât mai multe macro-uri posibile. Macro-urile pot fi: numărătoare, memorii RAM, registre, automate finite, decodificatoare prioritare, etc. [29]
Avem practic un set de reguli care ne spun cum să descriem hardware un bistabil, numărător, FSM pentru a fi recunoscute de programul de sinteză.
De exemplu, pentru a folosi memoria BRAM, în [29] avem niște reguli care vor fi folosite pentru a spune programului de sinteză ce fel de memorie vrem să folosim: citire sincronă/asincronă, dual-port, etc. La rularea lui, XST va încerca să recunoască cât mai multe astfel de descrieri. În cazul în care nu recunoaște, va încerca să contruiască funcțional circuitul.
Pentru automatele finite [29, 25], sunt mai multe tipuri de descriere. Cel mai recomandat este de a separa logica secvențială, adică de tranziție a stării la apariția frontului crescător (de exemplu) al semnalului de ceas și logica combinațională de construire a stării viitoare și eventual logica semnalelor de ieșirie în fiecare stare. Sunt mai multe metode de implementare a automatelor, iar programul de sinteză poate fi instruit care mod să folosească. Există moduri optimizate pentru consum minim, resurse minime, viteză maximă, etc.

Capitolul 7

Descrierea proiectării hardware

7.1 Informații generale despre designul hardware

7.1.1 Reprezentarea logică a tablei de joc

Un lucru foarte important în momentul în care se proiectează un procesor pentru un joc, este alegerea reprezentării tablei de joc. Importanța acestui lucru vine din faptul că o stare de joc este elementul de bază în rezolvarea jocului. Algoritmii de rezolvare, vor explora stări de joc. Putem considera această reprezentare ca un atom, iar proiectarea hardware constă într-o *atentă* explorare a acestor atomi.
Din regulile jocului observăm că Reversi se joacă pe o tabla cu dimensiunile 8x8. Știm că există doi jucători, fiecare adăugând pe tabla de joc discuri de culoare albastră respectiv roșie. O analiză mai atentă a acestor reguli, vor conduce la ideea de reprezentare a tablei de joc în procesor. Dimensiunea unei table fiind de 64 de căsuțe, observăm că putem reprezenta această informație pe un memorie de 64 de celule, dacă privim la modul abstract. Fiind într-un sistem digital, aceste căsuțe le constrângem sa aibă doar două valori, 0 sau 1.[1] Nouă ne trebuie 3 valori într-o căsuță, pentru că putem avea discuri albastre, roșii, sau căsuță goală. O soluție elegantă este să folosim 64 de celule pentru fiecare jucător. Vom părăsi denumirea de celule și ne vom referi la un registru pe 64 de biți. Considerăm 2 registre pe 64 de biți ca fiind informația despre configurația tablei de joc, în orice stare din spațiul complet al stărilor s-ar afla jocul. Notăm acele doua registre cu B și R.[2] Fiecare registru va avea un bit de 1 pe poziția unde jucătorul are pus un disc cu culoarea lui. Se observă imediat că este interzis ca registrele să aibă amândouă un bit de 1 pe aceași poziție, pentru că în joc nu se poate ca pe o căsuță să existe două discuri. Nici un disc pe căsuța i, va însemna un bit de 0 pe poziția i în ambele registre R, B și avem expresia logică $R_i \vee B_i = 0$. Având această reprezentare, putem genera și alte expresii ce ne vor ajuta în proiectare. De exemplu, pentru a avea un registru în care să avem 1 logic pe fiecare poziție liberă din tabla de joc, putem scrie $M = \neg(R \vee B)$.

[1] „0" se va înțelege ca și FALS, iar "1„ ca și ADEVĂRAT.
[2] Din engleză Blue and Red (Albastru și Roșu).

7.1.2 Leagea lui Amdahl

O consecință a legii lui Amdahl presupune favorizarea cazurilor frecvente în detrimentul cazurilor mai puțin frecvente. Voi respecta și în acest proiect această lege, mai ales că avem bine definiți pașii algoritmului, iar unele operații ies clar în evidența ca fiind executate frecvent. *A executa* aici are sensul de a evalua o expresie, sau a aștepta răspunsul de la un modul care în interior face anumite calcule. Algoritmul presupune majoritatea timpului explorări de stări de joc, trecerea dintr-o stare de joc în alta, generarea mutărilor legale pentru un anumit jucător. Fiind luată în considerare complexitatea arborelui de joc și o analiză atentă a jocului, reiese că procesorul va face calcule precum *generează lista de mutări valide*, *fă mutarea i pentru jucătorul p*, *generează un scor pentru această stare de joc*, foarte des.[3] Prin urmare, proiectarea optimă a acestor tipuri de module, va duce la un timp de execuție minim.

7.1.3 Modul de proiectare

Procesorul de Reversi este scris modular, în secțiunile următoare voi descrie fiecare modul în parte. Modulul de top le va avea pe toate cele descrise conectate între ele. Un modul are *intrări* și *ieșiri* și funcționalitatea din interior. După această tipologie vor fi descrise în continuare modulele. Ca și intrări comune, vom avea în general semnalul de ceas *clk* și semnaul de resetare *RST*, fiind vorba de obicei de un circuit secvențial.

7.2 Modulul de generare a mutărilor valide

Tranziția de la o stare curentă a jocului la o stare viitoare, este posibilă doar prin existența unei funcții de tranziție între cele două stări pentru jucătorul care e la rând. În orice stare a jocului ne aflăm, pentru a continua jocul, avem nevoie de întreaga mulțime $M = \{m_1, m_2 \dots m_n\}$ unde m_i reprezintă perechea (i, j) , o pereche de coordonate în tabla de joc, care reprezintă o mutare validă pentru jucătorul curent. Jucătorul va alege una din aceste mutări posibile, conform strategiei. În momentul în care mulțimea $M = \{\emptyset\}$ pentru ambii jucători, atunci și numai atunci jocul s-a terminat.

7.2.1 Observații

Acest modul, cu numele *moves_map*, face parte din modulele considerate *critice* unde vom urma regulile Amdahl. (7.1.2)
O observație atentă, ne arată că pentru fiecare stare de joc explorată, pentru a explora următoarea stare, trebuie să avem o tranziție din starea s_i în starea s_{i+1}, conform unei strategii. Cum această operație, de tranziție, va fi executată de procesorul Reversi majoritatea timpului, obiectivul nostru fiind viteza, vrem ca timpul alocat acestei operații să fie minim. Vom descrie ce înseamnă matematic o mutare validă în jocul Reversi, după care voi reveni cu o observație de implementare.

[3] Pentru o adâncime în arbore de 6 niveluri, se vor explora aproximativ 10^6 stări de joc, nefiind luată în considerare nici o optimizare.

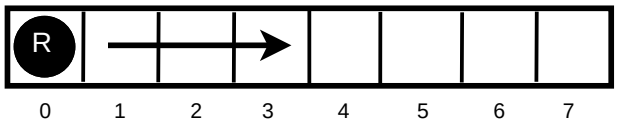

Figura 7.1: Pentru exemplificare considerăm o singură direcție, orizontal spre dreapta, discul marcat cu „R" pe poziția 0 înseamnă intenția jucătorului „R" de a muta acolo, căsuța fiind liberă.

7.2.2 Expresiile logice pentru o mutare validă

Din regulile jocului 2.2.2, observăm că jucătorii adaugă discuri, de culoarea corespunzătoare fiecăruia, alternativ. Prima regulă evidentă, este că un disc poate fi poziționat doar pe un loc liber, avem deci expresia $R[i] \vee B[i] = 0$. (7.1.1)
Existența condiției de *flancare*, înseamnă că un rând de discuri adversare trebuie să se termine cu un disc de-al jucătorului curent. Numărul de discuri capturat trebuie să fie cel putin 1, iar capturarea se face pe toate cele 8 direcții. Asta înseamnă că pentru fiecare poziție din tablă i, putem scrie o ecuație logică unde rezultatul va fi *FALS* dacă regulile jocului nu permit ca jucătorul curent să poziționeze un disc pe poziția i făcând o tranziție într-o stare viitoare si *ADEVĂRAT* dacă regulile jocului permit acest lucru. Ecuațiile vor lua în considerare toate posibilitățile ce se pot întâlni pe o anumită direcție, adică, jucătorul poate captura pe direcția d $1, 2 \ldots$ până la maxim 6 discuri ale adversarului. Mutarea fiind validă dacă este permisă capturarea pe cel puțin o direcție din cele opt, putem construi un sistem de expresii logice, iar reuniunea lor va reprezenta rezultatul.
Considerăm în continuare existența unei singure direcții, cea orizontală spre dreapta pentru simplitate. Coordonatele vor fi cuprinse între 0 și 7. De exemplu, pentru colțul stânga-sus, presupunând că jucătorul roșu e la rând, vom avea ecuația:

$$
\begin{aligned}
M_0 =& (\neg R_0 \wedge \neg B_0) \wedge && (7.1)\\
& \big((B_1 \wedge R_2) \vee && (7.2)\\
& (B_1 \wedge B_2 \wedge R_3) \vee && (7.3)\\
& (B_1 \wedge B_2 \wedge B_3 \wedge R_4) \vee && (7.4)\\
& (B_1 \wedge B_2 \wedge B_3 \wedge B_4 \wedge R_5) \vee && (7.5)\\
& (B_1 \wedge B_2 \wedge B_3 \wedge B_4 \wedge B_5 \wedge R_6) \vee && (7.6)\\
& (B_1 \wedge B_2 \wedge B_3 \wedge B_4 \wedge B_5 \wedge B_6 \wedge R_7)\big). && (7.7)
\end{aligned}
$$

În acest caz avem cele mai multe expresii pentru că pornim de la M_0. Observăm din figura 7.1 că pentru M_6, nu avem expresii pentru această direcție pentru că are ca vecin doar pe M_7, iar conform (7.2.2) trebuie să flancăm cel puțin un disc adversar. Pentru M_5 avem expresiile:

$$
\begin{aligned}
M_5 =& (\neg R_5 \wedge \neg B_5) \wedge && \text{(condiția ca poziția 5 să fie liberă)}\\
& (B_6 \wedge R_7). && \text{(condiția să flancăm poziția 6)}
\end{aligned}
$$

Din figura 7.2, observăm că pentru pozițiile $(3,3), (3,4), (4,3), (4,4)$ nu avem ecuații pentru că acestea nu vor fi niciodată libere, conform regulilor

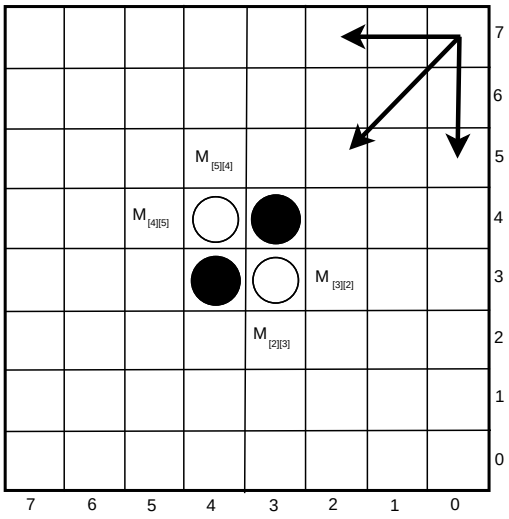

Figura 7.2: Starea inițială a jocului. $M_{[i][j]}$ înscrise pe grafic, reprezintă mutările legale ale jucătorului *negru*.

jocului. Rămân deci 60 de expresii logice, pentru fiecare poziție de pe tablă. Fiecare poziție având cel mult 8 expresii precum în (7.1) și cel puțin 3 dacă observăm colțurile, unde avem doar 3 direcții. Se observă diferite simetrii în tabla de joc, de exemplu $M_{[4][5]}$ cu $M_{[3][2]}$ sau colțurile opuse fiind simetrice din punct de vedere a expresiilor.

7.2.3 Descrierea modulului

Denumire

moves_map

Dependințe

N/A

Intrări

Modulul primește la intrare semnalele:

- Semnalul *clk* este tactul la care va funcționa acest modul și va fi legat la tactul global în acest proiect.
- Semnalul *RST*, semnal de resetare legat la resetul global.
- $B\_$, $R\_$ în total $2 \cdot 64$ biți de date, reprezentând o stare a jocului. (7.1.1)
- *player* intrare 1 bit care reprezintă jucătorul pentru care trebuie construită harta mutărilor valide.

Ieșiri

- $M\_$, ieșire pe 64 biți, ce reprezintă harta mutărilor valide, scopul acestui modul.

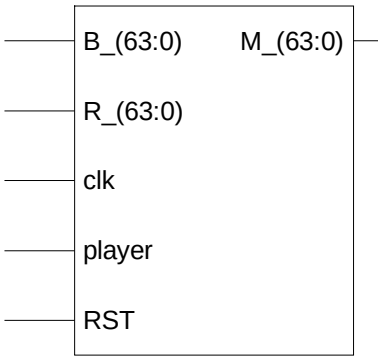

Figura 7.3: Modulul *moves_map*, schemă black-box generată de programul Xilinx ISE.
[4]

7.2.4 Detalii de implementare

Pentru a avea o proiectare bună a modulului, s-a luat în considerare principiul de a avea un registru la ieșirea modulului pentru a păstra ieșirile până la următorul *clock cycle*[5] (6.2). Prin urmare, modulul este împarțit într-o parte combinațională și o parte secvențială care va memora ieșirile. Ieșirea se schimbă pe frontul crescător al semnalului *clk*.
Vom declara registrul:

```
reg [63:0] RES_Q;
reg [63:0] RES_D;
```

Ieșirea M este de tipul *wire*, este conectată direct la *RES_Q*, ieșirea registrului *RES* implementat cu bistabile de tipul *D*.

Partea secvențială a modulului:

În *VERILOG* avem următoarea descriere:

Fragment sursă 7.1 Procesul sincron

```
always @(posedge clk) begin
    if ( RST ) begin
        RES_Q <= 64'b0;
    end
    else begin
        RES_Q <= RES_D;
    end
end
```

Operația îndeplinită este de a bascula la ieșirea Q, intrarea D a bistabilelor ce formează registrul *RES*, pe frontul crescător al semnalului de sincronizare.

[5] Ciclu de ceas.

Resetul acestui modul este sincron cu semnalul *clk*, un principiu bun de proiectare hardware în FPGA (6.2). Ieșirea Q va avea toți biții *0* în cazul în care linia *RST* este activată.

Partea combinațională a modulului

Am descris în (7.2.2) că fiecare poziție de pe tabla de joc va fi o mutare validă dacă o serie de propoziții logice vor returna *adevărat*. Tot în (7.2.2) am văzut că o poziție poate să aibă maxim 8 expresii logice și minim 3. Ca și structură, am definit în modulul *moves_map* un registru pe 8 biti pentru fiecare poziție din tabla de joc și 64 astfel de registre. În total $64 \cdot 8$ bistabile. Cei 8 biți ai unui registru, corespund celor 8 direcții în care un disc poate captura discuri adversare:

```
parameter ZEROP   = 3'd0;
parameter DOWN45P = 3'd1;
parameter DOWN    = 3'd2;
parameter DOWN45M = 3'd3;
parameter ZEROM   = 3'd4;
parameter UP45M   = 3'd5;
parameter UP      = 3'd6;
parameter UP45P   = 3'd7;
```

În acești parametrii, care sunt un alias pentru valorile numerice care înseamnă direcția, „45" se referă la direcțiile oblice, „UP" la direcțiile pozitive, dacă considerăm un sistem de axe în poziția de pe tablă unde dorim să punem discul, „ZERO" înseamnă orizontal, adică coordonata $y = 0$, iar „M" și „P" vine de la stânga/dreapta[6] pe tabla de joc.
Un alt lucru ce am să-l menționez aici, este vorba de intrarea *player*[7] a modulului. Singura diferență între expresiile (7.1) luând în considerare jucătorul, este că se va interschimba „B" cu „R". Jucătorii având aceleași reguli, contează doar culoarea lor. Această observație este foarte importantă pentru că nu vom generea o circuistică în plus pentru al doilea jucător. Determinarea mutărilor pentru orice jucător vor folosi aceleași expresii, singura diferență este că se va inversa „B" cu „R" pentru adversar. Mă refer relativ la adversar, adică consider jucătorii de genul p și $\neg p$, pentru că nu contează aici cine e *roșu* și cine e *albastru*, singurul lucru important este această observație de care am vorbit și expresiile (7.1).
În descrierea hardware, multiplexăm intrările „R" și „B", intrarea *player* fiind intrarea de selecție. În *VERILOG* vom avea următorul cod:

```
assign R = (player) ? B_ : R_;
assign B = (player) ? R_ : B_;
```

În continuare se va lucra cu „R" și „B" care reprezintă intrările „R_" și „B_" în această formă, sau inversate.
Expresiile logice vor fi evaluate continuu, la fiecare modificare a lui „R" și „B" și în paralel pentru fiecare direcție și pentru fiecare poziție de pe tablă. În acest

[6] De fapt vine de la Minus/Plus.
[7] Jucător.

mod, întarzierea maximă va fi dată de etajele logice din cea mai complicată expresie. Exemplificăm printr-o expresie din cod:

Fragment sursă 7.2 Expresia logică pentru $M_{[0][0][UP45M]}$

```
M[0][0][UP45M] = (!R[0*8 + 0] && !B[0*8 + 0]) &&
 (
  (R[1*8 + 0] && B[2*8 + 0]) ||
  (R[1*8 + 0] && R[2*8 + 0] && B[3*8 + 0]) ||
  (R[1*8 + 0] && R[2*8 + 0] && R[3*8 + 0] && B[4*8 + 0]) ||
  (R[1*8 + 0] && R[2*8 + 0] && R[3*8 + 0] && R[4*8 + 0] && B[5*8 + 0]) ||
  (R[1*8 + 0] && R[2*8 + 0] && R[3*8 + 0] && R[4*8 + 0] && R[5*8 + 0] && B[6*8 + 0]) ||
  (R[1*8 + 0] && R[2*8 + 0] && R[3*8 + 0] && R[4*8 + 0] && R[5*8 + 0] && R[6*8 + 0] && B[7*8 + 0])
 );
```

Expresia din fragmentul de sursă (7.2) aparține poziției $M_{[0][0]}$, vezi figura 7.2 pentru a vizualiza pe tablă această poziție, pe direcția diagonală sus-stânga. Este un exemplu de cea mai lungă expresie, pentru că pot fi capturate până la 6 discuri, ce reprezintă numărul maxim pe o tablă $8 * 8$.
La fiecare intrare D_i din registrul pe 64 biți *RES*, va fi o expresie ce va face operația „SAU" între valorile expresiilor determinate pentru fiecare direcție din cele 8. În final, ca și asignare continuă vom avea M legat de ieșirea Q a registrelor.

Schema internă

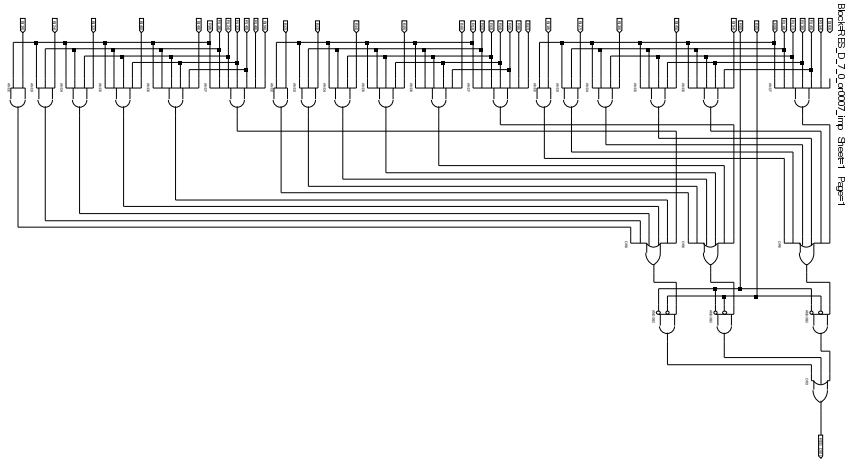

Figura 7.4: Această schemă reprezintă intrarea D_0 a registrului *RES*.

În 7.4 este expresia completă pentru intrarea D_0 a registrului de ieșire. Programul de sinteză minimizează dacă este cazul orice expresie legică. De exemplu, pentru pozițiile unde o anumită direcție nu există, pozițiile de margine sau colțurile, valoarea expresiei este cunoscută de la început ca fiind valoarea 0 pentru direcția inexistentă. Programul de sintetizare va folosi acest lucru pentru generarea unui număr minim de porți logice. Se observă în figură, poarta *SAU* cu 3 intrări, fiind cele 3 direcții.

7.2.5 Observații

Numărul de expresii fiind foarte mare, mai exact $60 \cdot 8$, scrierea manuală ar fi fost extrem de complicată și foarte posibil ar fi apărut multe greșeli. Dar, toate aceste expresii pot fi foarte ușor descrise algoritmic. Am dezvoltat și eu un script *Python* pentru a descrie pentru fiecare poziție expresiile logice, iar ieșirea acestui script reprezintă codul sursă *Verilog*. Ce se vede de exemplu în fragmentul de sursă 7.2, este cod sursă generat de acest script (*map.py*). Descrierea algoritmică a expresiilor a necesitat 77 linii de cod, sursa generată fiind de 2392 linii de cod.

7.2.6 Concluzii

După sintetizare, conform raportului generat de către *xst*[8], modulul *moves_map* ocupă 7% din resursele (*slice-uri*) FPGA-ului. Acest număr poate varia în momentul în care acest modul va fi integrat cu alte module, datorită resintetizării de către program, împreună cu alte module. Numărul de *LUT-uri* (vezi 5.1.2) ocupate este de 594, aproximativ 6% din resurse.
Observăm că s-a consumat o parte semnificativă din resurse cu acest modul, acesta fiind compromisul acceptat dorind ca la ieșirile modulului să avem noi date pe fiecare front crescător al impulsului de sincronizare. Modulul curent se încadrează constrângerii globale de $50MHz$ atribuită semnalului *clk*. Dacă dorim să creștem frecvența, probabil va trebui să facem o serie de modificări în design datorită întârzierilor pentru fiecare nivel de porți logice.

7.3 Modulul de tranziționare între stări

În primul modul prezentat, dând la intrare o stare de joc reprezentată prin „R" și „B" și prin „player", semnal pe 1 bit însemnând jucătorul pentru care se calculează ieșirile, obținem o hartă de biți unde prin biții de „1" avem reprezentate mutările valide. Fiind determinată această hartă, avem oarecum în acel modul exprimate o parte din regulile jocului. Ele sunt exprimate în acele expresii logice de care am vorbit, dar lipsește a doua parte a regulilor, adică. „Cum trecem la o stare viitoare?".
Modulul prezentat în această secțiune, complementează modulul anterior implementând restul de reguli ale jocului. Reglulile neimplementate până acum sunt acelea referitoare la starea viitoare a jocului, adică răspunsul la întrebarea „Ce se întâmplă când am pus discul de culoarea mea pe această poziție?". Matematic putem exprima:

[8]Programul de sintetizare de la Xilinx.

$$M = \{m_1, m_2, \ldots, m_k\} \tag{7.8}$$
$$H = \{h_1, h_2, \ldots, h_q\} \tag{7.9}$$
$$S = \{s_1, s_2, \ldots, s_n\} \tag{7.10}$$
$$H \subset S \tag{7.11}$$
$$\delta(s_0) : S \backslash H \rightarrow S \backslash \{H, s_0\} \tag{7.12}$$
$$H = H \cup \{s_0\} \tag{7.13}$$

H reprezintă stările care au fost jucate, în Reversi nu este posibil ca o stare a jocului din trecut să se repete, pentru că jocul se termină maxim în 60 de mutări.
În acest modul ne interesează funcția δ, unde ca rezultat vom avea o nouă stare a jocului. Având în vedere algoritmul de explorare a spațiului stărilor, vom comanda foarte des acest modul, pentru fiecare tranziție din starea s_i în s_{i+1}. Este evident că timpul total de explorare, timpul de „gândire" a procesorului este direct proporțional cu timpul de răspuns al acestui modul. Din nou, ca și o consecință a legilor lui Amdahl (7.1.2), va trebui să proiectăm acest modul făcând alte compromisuri în favoarea timpului mic de răspuns.
Algoritmic, ce trebuie să facă acest modul, este de a captura piesele adversarului. Se presupune că la intrare primește o coordonată de pe tabla de joc, validă, adică unde jucătorul care e la rând are voie să pună disc, iar rezultatul constă într-o tablă de joc modificată, cu discurile adversarului transformate în culoarea jucătorului curent.

7.3.1 Metoda greșită

Dacă ar fi să exprimăm printr-un pseudocod acest algoritm, considerând un procesor obișnuit, secvențial, cel mai probabil vom exprima tabela de joc ca o matrice $8 * 8$ și pentru fiecare direcție din cele 8, vom face un cilcu *while* unde vom schimba culoarea discurilor adversare în culoarea noastră. Vom merge în acel ciclu până când am ajuns la un disc de culoarea noastră, marginea tablei de joc, sau o casuță liberă. Dacă nu am întâlnit un disc de culoarea noastră (adică a jucătorului ce e la rând), atunci nu se face nici o modificare.
La început, s-a încercat și această soluție în hardware. Ciclul a fost înlocuit cu un automat cu stări finite.
Această soluție presupunea un automat finit pentru fiecare direcție, astfel inspectându-se în paralel fiecare direcție pentru a întoarce discurile adversarului. Problema mare aici ar fi fost memoria de lucru. Ar fi fost prea complicat ca aceste automate să opereze toate pe aceași matrice de bistabile, care sunt reprezentarea tablei de joc. Era pus în dificultăți și programul de sinteză, care nu a găsit altă soluție decât conectarea ieșirilor mai multor porți la intrarea unui bistabil, total greșit.
Dacă aș fi folosit câte o matrice de bistabile pentru fiecare direcție din cele 8, s-ar fi consumat mult prea multe resurse, necâștigându-se nimic în loc. Mai mult, circuistica pentru a compune rezultatul celor 8 direcții, adică de a face operația SAU între ele, ar fi fost iar complicată. Un alt lucru esențial, ar fi fost sincronizarea între procesele care inspectează paralel toate cele 8 direcții

în care se pot captura discuri în Reversi. Pentru o anumită poziție, câteva dintre direcții se vor termina mai repede de inspectat, fiind mai puține discuri de capturat, iar altele mai târziu, fiind mai multe discuri. Această așteptare după toate cele 8 direcții, ar fi necesitat o circuistică suplimentară.
Având în vedere că automatele finite folosite vor face o tranziție la fiecare impuls de sincronizare, fiind vorba de aproximativ 10 stări, nu are sens să fie prezentate, timpul de răspuns al modulului în acest caz ar fi fost foarte mare comparativ cu soluția prezentată în secțiunea următoare.

7.3.2 Implementarea corectă

După multe încercări de proiectare si de debarasare de o gândire secvențială și de tipologii de programare *software* limitate de capacitățile unui procesor secvențial, implementarea a părut evidentă. Inspirat din ideea de sumator cu transport propagat[9], putem considera tabla de joc ca un întreg circuit, unde toate pozițiile tablei sunt legate între ele, fiecare cu toți cei 8 vecini. Avem ca rezultat o rețea puternic interconectată formată din bistabile, 8 pentru fiecare poziție. Pentru a captura discurile pe o direcție, programatic am căuta discul de culoarea noastră iar dacă e găsit, toate discurile adversare până la discul nostru, adică cele flancate, se vor întoarce.[10] Putem considera un val care se propagă de-a lungul direcției, iar în momentul în care s-a lovit de un disc de culoarea jucătorului curent, valul este *reflectat*. Poziția de unde s-a pornit și toate pozițile de pe direcție prin care a trecut, vor sesiza această reflexie și își vor *complementa* culoarea. Acest lucru se va întâmpla în paralel pe toate direcțiile. În final se va compune la ieșirea modulului tabla de joc ce reprezintă o stare viitoare a jocului, adică rezultatul funcției $\delta(s_0)$, s_0 fiind starea de la intrarea modulului.

7.3.3 Descrierea modulului

Denumire

b_move

Dependințe

move_cell

Intrări

Modulul primește la intrare semnalele:

- *clk*, semnalul de tact la care va funcționa acest modul.
- *RST*, semnalul de reset al modulului, conectat la resetul global.
- *player*, intrare 1 bit care reprezintă jucătorul pentru care se va face mutarea.

[9] În engl. ripple carry adder.
[10] Asta înseamnă că vor deveni de culoarea jucătorului curent.

- *X*, intrare 3 biți care reprezintă coordonata x pe tabla de joc, adică coloana. (vezi 7.2)
- *Y*, intrare 3 biți care reprezintă coordonata y pe tabla de joc, adică linia. (vezi 7.2)
- $B\_$, $R\_$ în total $2 \cdot 64$ biți de date, reprezentând o stare a jocului. (7.1.1)

Ieșiri

- R_OUT ieșire 64 biți reprezentând matricea R pentru starea viitoare. (vezi 7.1.1)
- B_OUT ieșire 64 biți reprezentând matricea B pentru starea viitoare. (vezi 7.1.1)

7.3.4 Detalii de implementare

Am vorbit mai înainte despre faptul că ne vom închipui tabla de joc formată dintr-o rețea puternic conectată, adică fiecare căsuță conectată cu toți cei 8 vecini, unde e cazul. Mă voi referi în continuare la termenul de *celulă*, având 64 de celule pentru o astfel de reprezentare. Fiind prea complicat circuistic să construim o celulă care să poată fi conectată cu 8 astfel de celule, ne vom închipui că există 8 straturi. în fiecare strat valul se propagă într-o singură direcție și se reflectă dacă e cazul, asta însemnând că o celulă va fi conectată cu maxim 2 celule vecine. Fiind o singură direcție de propagre pe strat, celulele de pe două linii diferite nu vor fi legate între ele, dacă considerăm cele două direcții orizontale. La fel și pentru verticală, coloanele nu vor fi legate între ele. Pentru cele 4 diagonale la fel, vor fi despărțite.
Figura 7.5 prezintă un astfel de lanț de propagare. Este irelevantă direcția pe care se exemplifică. Toate lanțurile de propagare, în total $8 \cdot 8 \cdot 8$ sunt la fel, contează doar legăturile între ele, adică liniile *fw* și *bw*.[11]

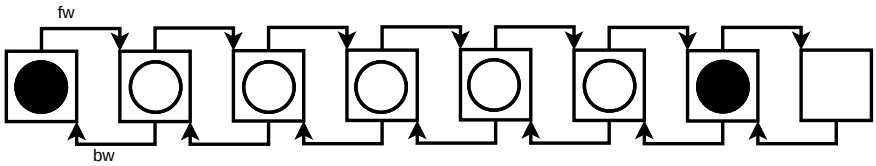

Figura 7.5: Considerăm propagarea valului în acest exemplu, de la stânga spre dreapta, până la penultima celulă unde se află un disc de culoarea considerată curentă, negru. *fw* marchează direcția de propagare, iar *bw* semnalul ce se întoarce în cazul în care a fost găsit un disc de culoare neagră.

Definiția unei celule. (modulul *move_cell*)

Fiind un număr mare de astfel de celule, $8 \cdot 8 \cdot 8$ trebuie să avem mare grijă la proiectare încât circuitul gândit să ocupe minim de resurse.[12] Orice circuistică

[11] În engl. forward și backward. În rom. înainte și înapoi.
[12] 8 celule pentru fiecare lanț, 8 lanțuri per strat, 8 straturi.

în plus, va fi multiplicată cu numărul total de celule. Trebuie avut grijă pentru a nu avea secvențe *if* prea multe, pentru că se vor traduce în multiplexoare care sunt destul de costisitoare. O celulă este un modul pur combinațional.

Intrări

- r semnal 1 bit ce reprezintă existența unui disc roșu pe poziția atribuită celulei.
- b semnal 1 bit ce reprezintă existența unui disc albastru pe poziția atribuită celulei.
- *pulse* semnal 1 bit reprezintă sursa propagării semnalului fw.
- *fw_in* semnale 1 bit ce reprezintă semnalul de propagare primit de la vecinul anterior.
- *bw_in* semnale 1 bit ce reprezintă semnalul de întoarcere primit de la vecinul următor.

Ieșiri

- *fw_out* semnal 1 bit ce reprezintă semnalul de propagare trimis la vecinul următor.
- *bw_out* semnal 1 bit ce reprezintă semnalul de întoarcere trimis la vecinul anterior.
- *r_out, b_out* semnale 1 bit ce reprezintă noile valori pentru *r, b*.

Schema bloc

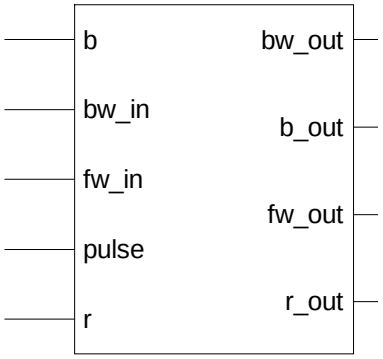

Figura 7.6: Schema bloc a modulului *move_cell*.

În figura 7.5 am considerat discul inițial fix la început dar nu trebuie neaparat să fie așa. În tot lanțul de celule trebuie să spunem unde inițiem propagarea. Pentru acest lucru este semnalul *pulse*. O singură celulă din lanț va avea semnalul *pulse* pus pe 1 logic, pentru că un singur disc se poate pune pe tablă la o mutare. *Pulse* este inițiatorul „valului" și va fi pe poziția pe care jucătorul va pune un disc.

Fragment sursă 7.3 Procesul combinațional pentru *move_cell*

```
always @( * ) begin
   /* if backward signal and forward signal, flip the discs.
      Otherwise, r,b bits remain the same */
    if ( bw_in && fw_in ) begin
        b_out_reg = 1;
        r_out_reg = 0;
    end
    else begin
        b_out_reg = b;
        r_out_reg = r;
    end
end
```

Logica unei celule este foarte simplă, lucru absolut necesar. Se vor considera *b_out_reg* și *r_out_reg* ca fiind *r_out* și *b_out*, numele inițial având sens doar în descrierea comportamentală în *VERILOG*.
Pentru interpretare, celula consideră jucătorul albastru ca fiind la rând. Un lucru important tot pentru a simplifica circuistica, vom vedea ulterior cum se calculează pentru jucătorul roșu.
În descrierea comportamentală, avem: dacă semnalul *bw_in* și *fw_in* au valoarea 1 logic, atunci înseamnă ca a fost primit semnalul de întoarcere și cel de propagare, ceea ce înseamnă că s-a întâlnit un disc de culoarea jucătorului, albastru în acest caz. Rezultatul, va fi setarea componentei *b* a celului pe 1 și a componentei *r* pe 0. S-a întors practic discul.
În cazul în care rezultatul porții *ȘI* dintre cele două semnale de propagare nu este 1, componentele *r* și *b* rămân egale cu cele inițiale, adică discul nu trebuie întors.

Fragment sursă 7.4 Expresiile logice a semnalelor de propagare

```
assign  fw_out = pulse || (r && fw_in);

/* backward signal is 1 if we received
   backward signal from neighbour cell,
   or if we flanc the disc (b bit is 1)
   and we propagate the bw signal */

assign bw_out = bw_in || (b && fw_in);
```

Semnalul de întoarcere *bw_out* se va propaga în cazul în care avem la intrarea modulului semnalul *bw_in* activ, asta înseamnă ca discul albastru căutat pentru a flanca, a fost găsit, iar valul este reflectat, sau, celula reprezintă un disc albastru și am primit semnalul de propagare *fw_in* care înseamnă ca tocmai această celulă e cea căutată. Semnalul *fw_out* va fi 1 logic, dacă a fost primit deja un „val” pe intrarea *fw_in*, iar celula reprezintă un disc roșu, sau, avem semnalul *pulse* activ care înseamnă că tocmai s-a inițiat o propagare,

adică se dorește o mutare pe acea poziție. Din aceste două semnale trebuie observat ca r și b se consideră că nu vor fi simultan 1, pentru că nu pot fi două discuri pe aceași căsuța de pe tablă.

Modulul *b_move*

Respectând una din paradigmele proiectării hardware corecte, adică "întotdeauna reține ieșirile"[13], modulul are și o parte secvențială, care acest lucru va face. La fiecare impuls de ceas, va bascula la ieșirea Q a registrelor, intrarea D. Registrele sunt puse pe fiecare ieșire, adică pe *R_OUT* și *B_OUT*. În total $2 \cdot 64$ bistabile. Dacă linia de reset, *RST*, este activată, atunci registrele vor avea la ieșire intrările nemodificate: Toată logica compusă de către celulele *move_cell*

Fragment sursă 7.5 Procesul sincron pentru *b_move*

```
always  @(posedge clk) begin
    if (RST) begin
        /* prepare outputs */
        R_OUT_Q <= R_;
        B_OUT_Q <= B_;
    end
    else begin
/* output logic, registered, depends of player */
        R_OUT_Q <= R_OUT_D;
        B_OUT_Q <= B_OUT_D;
    end
end
```

vor fi legate la intrarea D a registrului *R_OUT* și *B_OUT*. Este vorba de ieșirile r și b a fiecărei celule. Timpul total al propagărilor prin celule trebuie să fie mai mic decât perioada impulsului de ceas, pentru că la fiecare front crescător, se vor citi intrările registrului, moment în care rezultatul de la celule trebuie să fie stabil. Se poate determina astfel frecvența maximă la care poate funcționa acest modul.

Definirea celulelor

Având structura unei celule, descrisă anterior, va trebui acum să le definim, adică să le legăm între ele. Vor fi în total $8 \cdot 8 \cdot 8$ celule. Avem definite în sursă, o matrice de *fire* pentru fiecare strat cu care vom lega intrările și ieșirile celulelor, semnalele de tip *fw* și *bw*. Pentru celulele de pe linii și de pe coloane, ambele sensuri (stânga, dreapta) pot fi depinite mai ușor, utilizând definiția *for* a limbajului *VERILOG*. For-ul va primi o variabilă de tip *genvar*, care este un iterator, iar în interior vom descrie celulele pentru un lanț, iar în for se va multiplica acel lanț de 8 ori, fiind schimbate indicii din matricea de fire pentru semnalele *fw* și *bw*. Pentru diagonale, e puțin mai complicat, fiind necesare tratarea câtorva cazuri speciale. În (7.6) *i1* este variabila din ciclul *for*. Se

[13] „Always register your outputs." [28]

Fragment sursă 7.6 Definirea unei celule în cod

```
move_cell m0(.r(R[0 + i1*8]), .b(B[0 + i1*8]),
             .fw_in(wfw_out0[1 + i1*8]), .bw_in(1'b0),
             .bw_out(wbw_out0[0 + i1*8]), .fw_out(wfw_out0[0 + i1*8]),
             .pulse((X == 0) && (Y == i1)),
             .r_out(R_OUT_D0[0 + i1*8]), .b_out(B_OUT_D0[0 + i1*8]) );
```

vede deci din parametrii *r* și *b*, că se definește celula 0, pentru fiecare linie.[14] La intrările și ieșirile de tipul *fw* și *bw*, sunt matricile care sunt definite ca fire.[15] Intrarea pentru *bw_in* adică semnalul de întoarcere, este legat la 0 logic, pentru că în acest exemplu este definită celula 0, corespunzătoare poziției $M_{[0][0]}$ (vezi 7.2), care nu are de la cine să primească semnal de întoarcere, pentru că este în marginea tablei de joc.[16] La fel și semnalul *bw_out* este redundant definit aici, va fi practic un fir care nu va fi legat niciunde, pentru ca nu are unde să se propage semnalul în dreapta, dar a fost definit pentru simetrie. Programul de sinteză va ignora acest lucru. Semnalul *pulse* va avea valoarea 1 dacă și numai dacă coordonata *X* și coordonata *Y*, intrări ale modulului, sunt egale cu coordonatele celulei. În acest caz, fiind definită celula 0, pentru fiecare coloană a tablei, *X* trebuie să fie 0, iar *Y* trebuie să fie egal cu coloana. Ieșirile *r_out* și *b_out* vor fi legate la intrarea unor bistabile *D*. Va fi definită o matrice de bistabile pentru fiecare strat. Ciclul *for* în *VERILOG* nu face altceva decât să repete o declarație în funcție de un parametru, în acest caz *i1*, altfel ar fi fost necesară scrierea de mână a tuturor celor $8 \cdot 8 \cdot 8$ celule.

Procesul combinațional

La fiecare impuls de ceas, rezultatul celulelor vor fi încărcate în registru. Având 8 straturi, va trebui să compunem rezultatul de la fiecare strat de celule. Intrarea *D* a registrelor de ieșire va fi o funcție *SAU* între toate cele 8 straturi pentru culoarea albastră și funcția *ȘI* pentru roșu.

Fragment sursă 7.7 Definirea intrărilor *D* pentru *R_OUT*, *B_OUT*

```
always @( * ) begin
    if ( player ) begin
        R_OUT_D = B_OUT_D0 | B_OUT_D1 | B_OUT_D2 | B_OUT_D3 | B_OUT_D4 | B_OUT_D5 | B_OUT_D6 | B_OUT_D7;
        B_OUT_D = R_OUT_D0 & R_OUT_D1 & R_OUT_D2 & R_OUT_D3 & R_OUT_D4 & R_OUT_D5 & R_OUT_D6 & R_OUT_D7;
        R_OUT_D[Y*8 + X] = 1;
        B_OUT_D[Y*8 + X] = 0;
    end
    else begin
        R_OUT_D = R_OUT_D0 & R_OUT_D1 & R_OUT_D2 & R_OUT_D3 & R_OUT_D4 & R_OUT_D5 & R_OUT_D6 & R_OUT_D7;
        B_OUT_D = B_OUT_D0 | B_OUT_D1 | B_OUT_D2 | B_OUT_D3 | B_OUT_D4 | B_OUT_D5 | B_OUT_D6 | B_OUT_D7;
        R_OUT_D[Y*8 + X] = 0;
        B_OUT_D[Y*8 + X] = 1;
    end
end
```

În sursă, 7.7, albastru este considerat ca fiind jucătorul pentru care $player = 0$. La intrarea registrului *B_OUT* se va face funcția *SAU*, pentru că ne interesează rezultatul fiecărui strat. Dacă de exemplu un strat nu a schimbat nimic

[14] 0 reprezintă coordonata *y*, iar *i1*8* reprezintă coordonata *x*.

[15] În *VERILOG* sunt *wires*.

[16] Este considerată direcția de propagare dreapta-stânga.

în tabla de joc, adică în acea direcție nu se puteau captura piese, atunci valorile celulelor vor fi tot 0 pentru matricea B. Noi dorim să luăm în considerare fiecare strat, discurile întoarse de pe fiecare strat, ne trebuie practic reuniunea lor $S_0 \cup S_1 \cup S_2 \cup S_3 \cup S_4 \cup S_5 \cup S_6 \cup S_7$. Pentru matricea R, avem nevoie exact invers, de a fi luat în considerare orice disc capturat, de pe orice strat. Un disc capturat va fi reprezentat prin 0 pentru ieșirea *r_out* din celulă. Avem nevoie practic de $S_0 \cap S_1 \cap S_2 \cap S_3 \cap S_4 \cap S_5 \cap S_6 \cap S_7$. Dacă roșu este jucătorul care a mutat, atunci rolulile R și B se schimbă, atâta tot. Este observația valabilă și la modulul *moves_map* (vezi 7.1). Pe poziția pe care s-a aplicat semnalul *pulse*, nu se pune nici un disc. Motivul este că s-ar fi inferat mult mai multe porți în total pentru toate celulele. Pentru a pune totuși discul pe tablă, se pune direct intrarea D corespunzătoare poziții pe 1, iar pentru culoarea adversarului, pe 0. Și acest lucru va genera o circuistică destul de mare, pentru că programul de sinteză va aplica un multiplexor pe fiecare poziție din cele 64. Se observă că discul se adaugă necondiționat, înseamnă că acest modul nu face nici un test dacă intrările X și Y sunt valide. Este și normal, cu validarea mutărilor se ocupă modulul anterior, cel de față presupune că intrările primite sunt valide.

7.3.5 Concluzii

Timpii de propagare prin toate nivelurile de porți logice este mai mic decât perioada tactului, funcționarea la $50MHz$ este asigurată. Dacă dorim să creștem frecvența, vor apărea noi probleme de proiectare care presupun un studiu destul de aprofundat. Resursele ocupate de acest modul, sunt semnificativ mai mari decât la modulul anterior, dar este un compromis bun, având în vedere că o tranziție de la o stare la alta se va executa într-un ciclu de tact. După sinteză, raportul arată că modulul ocupă 1515 de LUT-uri, ce înseamnă 16% din resursele totale, la care se adugă încă câteva blocuri logice. Numărul de linii de cod este de 378.
Cu *moves_map* și *b_move* pot să spun că există „armata” necesară pentru a „lupta” într-un joc precum Reversi, fiind luată în considerare complexitatea lui.

7.4 Controlerul VGA

Ca și metodă de comunicare cu un utilizator, am ales varianta afișării tablei de joc pe un monitor. De obicei, un controler VGA, va dispune de o memorie video unde se află informațiile de culoare pentru fiecare pixel. În cazul nostru, nu este nevoie de așa ceva, pentru că tabla de joc se poate descrie algoritmic, nu este nevoie să memorăm o hartă de biți ce compune tabla de joc. Vom genera deci, informațiile de culoare, componentele R, G, B în timp real.

7.4.1 Generatorul de semnale H/V

Ca și dependința, la controlerul VGA, avem un generator pentru impulsurile de sincronizare orizontale și verticale. Scopul acestui modul, este de a genera la portul VGA de pe placă, aceste semnale. Calculul valoriilor R, G, B se va face în controlerul VGA.

Schema bloc pentru modulul *hvsync_gen*

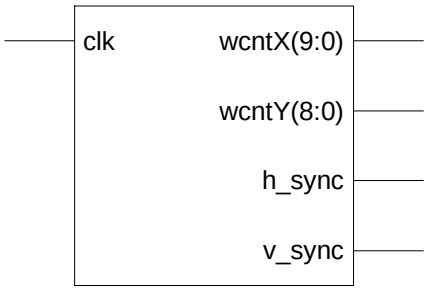

Figura 7.7: Schema bloc a modulului *hvsync_gen*.

Intrări

- *clk* tactul la care va funcționa acest modul, este tactul global $50Mhz$.

Ieșiri

- *wcntX* semnal 10 biți ce reprezintă zona vizibilă din ecran pe orizontală. (0-639)
- *wcntY* semnal 9 biți ce reprezintă zona vizibilă din ecran pe verticală. (0-479)
- *h_sync* impulsul de sincronizare orizontală.
- *v_sync* impulsul de sincronizare verticală.

7.4.2 Detalii de implementare

Informațiile necesare vizualizării pe monitor în modul 640x480@60Hz, au fost luate din standardul VESA, precum și alte documentații.[30]

Informațiile generale

- frecvența de reîmprospătare a monitorului: $60Hz$.
- frecvența de reîmprospătare verticală: $31.46875kHz$.
- impulsul de bază (pixel): 25.175 MHz.[17]

[17] În engl. pixel clock.

Informațiile de timp pentru orizontală

- aria vizibilă: 640 pixeli.
- timpul de întoarcere 64 pixeli.[18]
- Durata impulsului de sincronizare: 96 pixeli.
- Întreaga linie, 800 pixeli.

Informațiile de timp pentru verticală

- aria vizibilă: 480 linii.
- timpul de întoarcere 43 linii.
- Durata împulsului de sincronizare: 2 linii.
- Întregul cadru: 525 linii.

Precum se spune în [31], "standardul este că nu există nici un standard", aceste valori diferă de la monitor la monitor, dar se pot ajusta din setări.
În modulul *hvsync_gen*, aceste valori au fost înmulțite cu 2, pentru că frecvența de bază de a pixelului, este de $50Mhz$. *Designul* este format dintr-un circuit secvențial, care va incrementa un numărator pentru impulsul de sincronizare orizontală și un alt numărator pentru impulsul de sincronizare verticală. În momentul în care numărătoarele au atins valorile stabilite, se va genera la ieșire, impulsul în logică negativă. Primul numărător funcționează la frecvența de $50MHz$, iar cel pentru sincronizarea verticală se va incrementa o dată cu generarea unui impuls orizontal, deci se va incrementa o dată la fiecare linie. În momentul în care a ajuns la maxim, adică s-au desenat cele 480 de linii, acesta se va reseta. În ambele cazuri se iau în considerare timpii de întoarcere a tunului de electroni. Rolul ferestrei date de $wcntX$ și $wcntY$ este a cunoaște coordonatele pixelului ce trebuie afișat.

7.4.3 Detalii de implementare pentru modulul *vga_controller*

Dependințe

hvsync_gen

Schema bloc

Intrări

- *clk* tactul la care va funcționa acest modul, este tactul global $50Mhz$.
- *boardR* semnal 64 biți reprezentând matricea R, va fi afișat cu roșu pe monitor.

[18] În engl. front & back porch (timpii sunt exprimați în pixeli, se pot calcula cei reali folosind frecvența de baza).

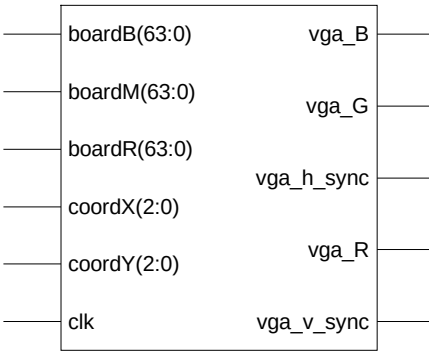

Figura 7.8: Schema bloc a modulului *vga_controller*.

- *boardB* semnal 64 biți reprezentând matricea B, va fi afișat cu albastru pe monitor.
- *boardM* semnal 64 biți reprezentând matricea M, mutările posibile pentru jucător, va fi afișat cu galben pe monitor.
- *coordX* semnal 3 biți ce reprezintă coordonata X curentă în tabla de joc, pătratul va fi colorat în alb.
- *coordY* semnal 3 biți ce reprezintă coordonata Y curentă în tabla de joc, pătratul va fi colorat în alb.

Ieșiri

- *vga_R* semnal 1 bit ce reprezintă informația de culoare, componenta R.
- *vga_G* semnal 1 bit ce reprezintă informația de culoare, componenta G.
- *vga_B* semnal 1 bit ce reprezintă informația de culoare, componenta B.
- *vga_v_sync* impulsul de sincronizare verticală, preluat de la modulul *hvsync_gen*.
- *vga_h_sync* impulsul de sincronizare orizontală, preluat de la modulul *hvsync_gen*.

În realitate, semnalul analogic aplicat intrărilor R, G, B unui monitor, vor genera toată paleta de culori, atât cât permit materialele chimice folosite. În această placă de dezvoltare pe care o folosesc, ieșirile din FPGA sunt legate la mufă fiecare în serie cu o rezistență de 270Ω. În acest fel, este asigurată plaja de $0V - 0.7V$ acceptată. Cum din FPGA putem genera două valori, 0 sau 1 logic, vom avea astfel cele 8 culori de bază.
Considerăm un pătrat de pe tabla de joc cu latura de 32 pixeli. Astfel, folosind ieșirile *wcntX* și *wcntY* de la modulul *hvsync_gen*, vom ști în orice moment ce pătrat din tabla de joc trebuie desenat, calculând câtul împărțirii lui *wcntX* respectiv *wcntY* la 32. În acest proces sincron, vom folosi și un comparator pentru a determina dacă ne aflăm în partea de început a ecranului, pentru a evita desenarea multiplă. Pentru a genera culoarea adecvată, coordonatele

aflate după împărțirea la latura pătratului ales, vor fi folosite pentru a afla valoarea poziției în tablourile *R, B, M*. În acest fel, culoarea aleasă va fi *roșu*, *albastru*, *galben* sau *alb* pentru a evidenția pătratul pe care utilizatorul s-a poziționat. În orice alt caz, se va genera valoarea zero la ieșirile *R, G, B*, deci pe monitor va apărea *negru*.

7.4.4 Concluzii

În acest modul constantele se pot modifica, astfel încât să deplasăm pe monitor la stânga, dreapta, sus, jos tabla de joc. S-a testat pe un model de monitor marca *DELL* și pe două modele de monitoare marca *LG*. Rezultatele mai bune au fost obținute pe monitorul *DELL*.

7.5 Miniclaviatura

Sunt puține comenzi necesare unui joc precum Reversi. Avem nevoie de a selecta poziția, avem două axe *OX* și *OY*, rezultă deci că patru butoane sunt ideale pentru a selecta poziția și un al cincelea pentru adăugarea discului la poziția aleasă. Poziția pe care ne aflăm va fi colorată în alb (vezi 7.4).
Placa de dezvoltare (vezi 5.2) prezintă 4 butoane cu contact metalic, care sunt legate printr-un rezistor de pull-down la intrarea FPGA-ului. Butonul 5 care reprezintă adăugarea discului pe tabla, va fi butonul rotativ, vom folosi doar funcția de buton, nu și cea de rotație. La fel ca la celălalte, butonul este legat la $3.3V$, iar la intrarea în FPGA vom avea o rezistență legată la masă. Adăugarea discului va fi posibilă doar dacă există acea tranziție posibilă (vezi 7.8).
Se știe că orice contact metalic suferă de o instabilitate în momentul apăsării. Datorită contactului metalic, fiind elastic, forma de undă va avea tranziții dese de la 0 la 1 și invers. Perioada de stabilizare este scurtă, dar este suficient ca acele tranziții parazite să fie interpretate greșit de către modulul de claviatură. Ca metodă de combatere, voi folosi o metodă simplă, pe departe optimă, dar functionează pentru situația dată. Voi folosi un divizor de frecvența, intrarea butonului fiind citită cu o frecvență foarte mica, mult mai mică decât perioada de instabilitate. Va fi folosit un numărător pe 23 de biți. Valoarea 23 a fost aleasă empiric, încât dacă butonul este continuu apăsat, deplasarea poziției pe tablă să se facă suficient de lent pentru a o stăpâni și a putea fixa o poziție. Frecvența cu care se va mișca poziția selectată în cazul în care butonul este continuu apăsat, va fi de aproximativ $7Hz$, valoare ușor de controlat de către om.

7.5.1 Descrierea modulului

Schema bloc

Intrări

- *clk* tactul la care va funcționa acest modul, este tactul global $50Mhz$.

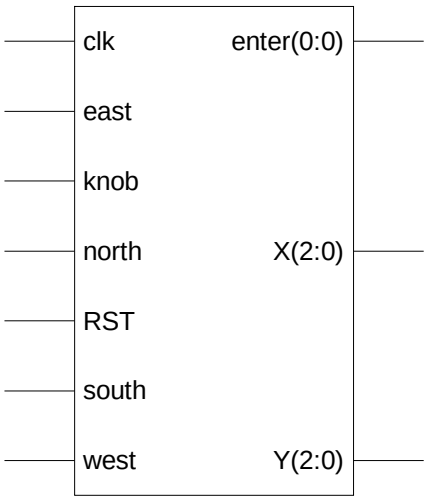

Figura 7.9: Schema bloc a modulului *xy_calculate*.

- *east*, *west*, *north*, *south* semnalele primite de la butoane, reprezintă cele 4 direcții de deplasare pe tablă.
- *knob* semnalul butonului rotativ, doar cel de apăsare, reprezintă poziționarea unui disc la poziția selectată pe tablă.
- *RST* semnalul de resetare pentru acest modul, va fi legat la resetul global.

Ieșiri

- *enter* semnalul ce va comanda adăugarea unui disc pe tablă.
- X semnal pe 3 biți ce reprezintă poziția curentă, coordonata X pe tablă.
- Y semnal pe 3 biți ce reprezintă poziția curentă, coordonata Y pe tablă.

7.5.2 Detalii de implementare

Modulul este format dintr-un proces sincron cu semnalul *clk*, la fiecare tact incrementându-se un numărător. În cazul în care numărătorul ajunge la va-

Fragment sursă 7.8 Incrementarea registrului Z

```
always @(posedge clk) begin
    if ( RST ) begin
        Z <= 0;
        X <= 0;
        Y <= 0;
        enter <= 0;
    end
    else begin
        Z <= Z + 1;
        ...
         end
...
end
```

loarea maximă, se vor testa semnalele de intrare care vin de la butoane. Pentru

fiecare semnal, dacă are valoarea 1, se vor face modificările corespunzătoare registrelor *X, Y* și *enter* în felul următor: Dacă semnalul *south* este activat, atunci registrul *Y*, coordonata Y pe tablă, se va decrementa, iar dacă e semnalul *north* activ, atunci se va incrementa.[19] Dacă semnalul *east* este activ, atunci din registrul *X* se va scădea o unitate, iar dacă semnalul activ este *west*, se va adăuga o unitate. Dacă semnalul *knob* este găsit activ, atunci registrul *enter* va avea valoarea 0. În cazul în care semnalul *RST* este 1, atunci toate registrele vor fi setate pe 0. (vezi 7.8)
Dacă vor fi active mai multe semnale, atunci doar unul se va lua în considerare. Prioritatea este *east, west, north, south, knob.*

7.5.3 Concluzii

Modulele *vga_controller* și *xy_calculate* au rolul de interfața I/O pentru acest proiect. În acest fel un jucător uman poate să joace Reversi interactiv cu această placă. Bineînțeles, monitorul și miniclaviatura se pot înlocui cu un ecran LCD cu touchscreen,[20] dar nu a fost scopul acestei lucrări.

7.6 Modulul pentru transmitere serială

A fost de multe ori nevoie de a vizualiza anumite informații, de a construi statistici. Placa de dezvoltare are în dotare două porturi seriale RS232. Nivelurile de tensiune necesare sunt ajustate pe placă, singurul efort rămâne de a construi protocolul. Modul folosit este simplu, linia Tx este ținută activă în modul *idle*[21] iar în momentul transmiterii, se va trimite un bit de start, având valoarea 0, după care urmează cei 8 biți de date, după care urmează un singur bit de stop, reprezentat prin 1 logic. După transmitere linia intră din nou în *idle* și va fi continuu activă. Viteza de transmitere va fi de 115200 baud. În general, pentru a genera un astfel de tact, se folosește un oscilator de $1.8432MHz$. Acesta divizat cu valoarea 16 ne va da exact $115200Hz$. Asta înseamnă că la fiecare 16 impulsuri de la oscilator, se va genera un impuls pentru interfața serială. Această placă având un oscilator de $50MHz$, va trebui să găsim un factor cu care să dividem această frecvență. Nefiind o prioritate în acest proiect, modulul a fost preluat din [32].

7.7 Transmiterea serială a unui număr 32 biți, în format ASCII

Acest modul e necesar pentru o vizualizare ușoară a datelor trimise. Astfel, în loc să trimitem un număr 32 de biți în forma lui binară, vom trimite același număr în format ASCII. Asta înseamnă că în loc să trimitem pe interfața serială 4 octeți, se vor trimite 8 octeți. Fiecare octet din cei 8 reprezintă o cifră hexadecimală a numărului pe 32 biți pe care dorim să-l transmitem.

[19] În realitate e invers datorită orientării plăcii.
[20] Ecran sensibil la atingere.
[21] Adică când nu se transmit date.

După transmiterea unui număr, se va transmite octetul 0x0D care reprezintă *line feed*.

7.7.1 Descrierea modulului

Dependințe

Modulul *rs232* (vezi 7.6).

Schema bloc

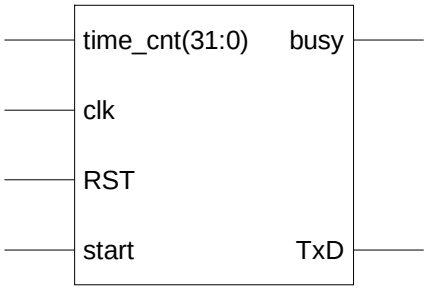

Figura 7.10: Schema bloc a modulului *time_analysis*.

Intrări

- *clk* tactul la care va funcționa acest modul, este tactul global $50Mhz$.
- *time_cnt* valoarea de 32 biți pe care dorim să o transmitem.
- *start* semnal 1 bit care reprezintă că se dorește începerea unei noi transmisii seriale.
- *RST* semnalul de resetare pentru acest modul, va fi legat la resetul global.

Ieșiri

- *busy* semnal 1 bit reprezintă starea modulului, dacă este 1 înseamnă ca modulul este deja ocupat cu o transmisie. Modulul ce comandă acest modul, va trebui să aștepte ca linia *busy* să fie 0.
- *TxD* semnalul care va fi legat la portul serial, linia Tx. Pe această linie se serializează informația transmisă.

7.7.2 Detalii de implementare

Modulul de transmitere serială este construit să trimită 8 biți de date la o transmisie. Dorind să trimitem 32 de biți, va trebui să-i trimitem pe rând. Proiectarea va fi în felul următor: Vom selecta pe rând din *time_cnt* câte 4

biți, care vor parcurge un automat fint care va face transformarea în ASCII și va seta corespunzător semnalele de așteptare.

Automatul finit

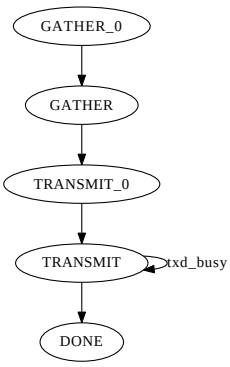

Figura 7.11: Automatul finit care reprezintă transformarea a 4 biți într-o reprezentare pe 8 biți, după care va genera semnalele corespunzătoare modulului rs232 pentru serializarea lui.

În starea GATHER_0 se va seta doar semnalul *busy_nibble*, semnal ce va deveni activ în starea următoare. Rolul este de a informa procesul care alimentează cu bucăți de câte 4 biți, că există deja o bucată în curs de transmitere.
În starea GATHER, se va face transformarea din 4 biți în 8 biți fiind reprezentarea ASCII.

Fragment sursă 7.9 Translatarea din binar (4 biți), în ASCII (8 biți).

```
case (nibble)
   4'b0000 : hex_byte_d = 8'h30;
   4'b0001 : hex_byte_d = 8'h31;
   4'b0010 : hex_byte_d = 8'h32;
   4'b0011 : hex_byte_d = 8'h33;
   4'b0100 : hex_byte_d = 8'h34;
   4'b0101 : hex_byte_d = 8'h35;
   4'b0110 : hex_byte_d = 8'h36;
   4'b0111 : hex_byte_d = 8'h37;
   4'b1000 : hex_byte_d = 8'h38;
   4'b1001 : hex_byte_d = 8'h39;
   4'b1010 : hex_byte_d = 8'h61;
   4'b1011 : hex_byte_d = 8'h62;
   4'b1100 : hex_byte_d = 8'h63;
   4'b1101 : hex_byte_d = 8'h64;
   4'b1110 : hex_byte_d = 8'h65;
   4'b1111 : hex_byte_d = 8'h66;
endcase
```

Acest cod definește valoarea ASCII corespunzătoare oricărei valori posibile de 4 biți. În circuistică, de obicei programul de sintetizare va genera multiplexoare. Ca și observație, programul de sinteză va observa că în acest *case* sunt acoperite toate cazurile și nu se poate întâmpla ca două cazuri să fie valide în același timp. Astfel, nu se vor genera *latch-uri* sau codificatoare prioritare.
În starea TRANSMIT_0, se va activa linia de *TxD_start* de la modulul RS232, care va începe serializarea datelor.
În starea TRANSMIT se va aștepta până cei 8 biți de date vor fi serializați,

adică semnalul *txd_busy* va fi 0. În momentul în care linia *busy* a modulului RS232 indică că datele au fost serializate, se va trece în starea DONE. În acest fel, s-au transmis 4 biți din cei 32 ai intrării *time_cnt*.
În paralel, un proces sincron cu semnalul *clk* va comanda acest automat și va pregăti cei 4 biți care urmează să fie transformați și serializați. Registrul

Fragment sursă 7.10 Pregătirea a 4 biți de date.

```
if ( (how_many_q < 5'b01000) && (~busy_nibble_q) ) begin
    busy_q <= 1;
    data <= { data[28:0], data[31:28] };
    how_many_q <= how_many_q + 1;
    state_q <= GATHER_0;
end
```

how_many memorează câte grupuri de 4 biți au fost serializate. Acest lucru e necesar pentru a trimite în final încă un octet, acela de *line feed*. Observăm în acest fragment de cod că se așteaptă ca linia *busy_nibble* să fie 0. În caz că nu se trimite nimic, se va face o rotire pe biți a registrului *data* ce conține cei 32 de biți ce trebuie transmiși, după care se va seta automatul finit în starea de start, adică GATHER_0.

7.7.3 Concluzii

Acest modul a fost folosit pentru a trimite date precum: în cât timp se gândește o mutare, sau ce scor au generat anumite stări ale jocului. Datele transmise pot fi folosite pentru statistici, fiind și resursa graficelor de comparație între FPGA și implementarea pe un CPU general.
Placa a fost conectată prin cablul serial la un computer, unde s-au capturat datele transmise.

7.8 Modulul de explorare

În continuare, urmează implementarea efectivă a algoritmului minmax. Vom folosi aici toate modulele considerate critice prezentate. Va fi nevoie de a afla mulțimea M, avem funcția de tranziție δ, urmează proiectarea modulului care să facă închegarea. Minmax, fiind un algoritm de tip *depthfirst* 3.7, va fi nevoie construirea unei stive în care se vor memora stările jocului de la fiecare nivel în arbore. Este nevoie de un număr mic de structuri memorate, pentru că este o explorare în adâncime. Considerăm momentan memorarea de stări ca fiind ceva abstract. În secțiunea 7.9 voi prezenta metoda folosită. Momentan considerăm o memorie adresabilă prin nivelul arborelui, care ne va furniza structura ce definește starea jocului memorată la care se adaugă scorul, adică valoarea minmax[22].
Algoritmului de explorare în hardware, i se potrivește foarte bine ca model, un automat finit. Avem nevoie de stări precum: generează mulțimea M, alege strategia $m_i \in M$, execută tranziția δ pentru starea curentă și strategia m_i, scrie în memorie structura aferentă stării curente, treci la nivelul următor în

[22]Se mai spune și *game-theoretic value*.

arbore, obtine $u(s_k)$ care reprezintă scorul stării s_k, sau utilitatea, mergi în sus cu un nivel în arbore, etc. Toate aceste acțiuni se vor întâmpla secvențial și sunt foarte bine definite. Ce putem optimiza la acest modul, este de a avea

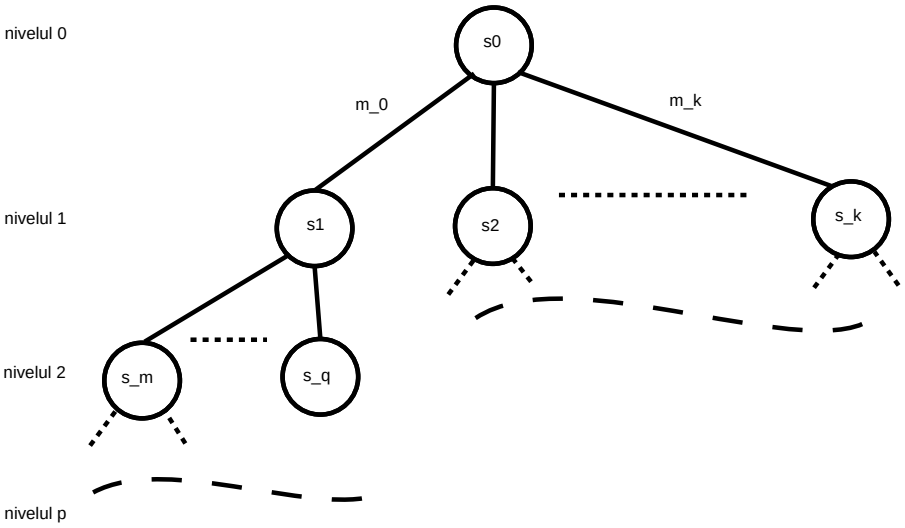

Figura 7.12: Exemplu arbore de joc. s_0, s_1, s_k, s_m, s_q reprezintă stări de joc. m_0, m_1 reprezintă strategii posibile.

cât mai puține stări în automat. Se vede acum de ce era necesar ca modulele folosite să aibă răspuns într-un timp foarte scurt, adică într-un ciclu de ceas în cazul nostru, pentru că astfel vom avea un număr minim de stări de așteptare. Dacă am dori să folosim altfel de algoritmi, de exemplu cei care presupun o ordonare a strategiilor înainte de a fi explorate stările în adâncime, s-ar putea să fie în final foarte ineficient, ordonarea fiind costisitoare.

7.8.1 Descrierea modulului

Comparativ cu restul modulelor prezentate, acesta este mult mai stufos. Complexitatea se datorează și faptului că este nevoie comunicaera cu alte module. De exemplu modulul de tranziționare *b_move* se va afla în exterior, pentru că este folosit și de alt modul. Modulul de tranziționare nu se duplică pentru că deja ocupă o parte considerabilă din resurse și este mai bine atunci să fie un acces partajat. Nu apare aici nici un deficit de viteză, pentru că nimeni nu va dori folosirea simultană a modulului *b_move*. Prima dată este folosit în explorare, iar în rest este folosit în momentul în care jucătorul uman a ales mutarea și prin apăsarea butonului va poziționa discul. În acel moment procesorul de Reversi nu va folosi modulul.

Denumire

game_ai

Dependințe

Modulul *b_move* (vezi 7.7).
Modulul *moves_map* (vezi 7.2).

Modulul *memory_bram* (vezi 7.9).
Modulul *heuristics* (vezi 7.11).
Modulul *RB_cnt* (vezi 7.10).

Schema bloc

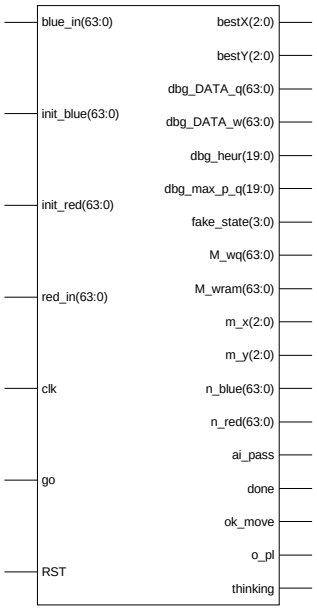

Figura 7.13: Schema bloc a modulului *game_ai*.

Intrări

- *clk* tactul la care va funcționa acest modul, este tactul global $50Mhz$.
- *RST* semnalul de resetare pentru acest modul, va fi legat la resetul global.
- *init_red* semnal 64 biți ce reprezintă matricea R, reprezentând starea s_0 a jocului, de unde va începe explorarea.
- *init_blue* semnal 64 biți ce reprezintă matricea B, reprezentând starea s_0 a jocului, de unde va începe explorarea.
- *red_in* semnal 64 biți ce reprezintă matricea$R$ primită de la modulul *b_move*.
- *blue_in* semnal 64 biți ce reprezintă matricea$B$ primită de la modulul *b_move*.
- *go* semnal 1 bit ce reprezintă inițierea unei noi explorări.

Ieșiri

O parte din ieșiri sunt semnale folosite pentru depanare și nu vor fi prezentate aici, pentru că nu influențează cu nimic.

- *n_red* semnal 64 de biți ce reprezintă matricea R care va fi conectată la intrarea modulului *b_move*.
- *n_blue* semnal 64 de biți ce reprezintă matricea B care va fi conectată la intrarea modulului *b_move*.
- *m_x* semnal 3 biți ce reprezintă coordonata X a unei mutări. Semnalul este conectat la modulul *b_move*.
- *m_y* semnal 3 biți ce reprezintă coordonata Y a unei mutări. Semnalul este conectat la modulul *b_move*.
- *o_pl* semnal 1 bit ce va fi conectat la *b_move* la intrarea *player* și reprezintă jucătorul care e la rând.
- *bestX, bestY* reprezintă mutarea cea mai bună aleasă de algoritm. Semnalele sunt pe 3 biți fiecare.
- *thinking* semnal 1 bit care va fi 1 logic cât timp modulul explorează în spațiul stărilor.
- *ai_pass* semnal 1 bit care va fi 1 logic dacă nu există nici o mutare posibilă, $M_{s_0} = \{\emptyset\}$, adică trebuie să mute tot adversarul.
- *done* semnal 1 bit care va fi activat în momentul în care s-a terminat explorarea.

7.8.2 Detalii de implementare

Căutarea în arbore este implementată folosind un automat finit cu 15 stări. Pentru a proiecta mult mai ușor, s-a considerat că intotdeauna se „operează" asupra unor registre interne care memorează matricile R, B, M, cele 3 dimensiuni prezente în această reprezentare. Întotdeauna când avem noi valori pentru aceste matrici, fie că sunt citite din memoria RAM, sau că sunt primite de la modulul *b_move* (adică a avut loc o tranziție), valorile vor fi memorate în aceste registre interne. La fel se procedează și cu alte valori, există un singur registru intern pentru stocarea celei mai bune mutări.[23] În acest fel este mult mai ușor să gândim, pentru că știm că trebuie să modificăm într-un singur loc, si vom lua o singură decizie, de a memora sau nu noile valori primite.
Circuitul este sincron cu semnalul *clk*, iar starea următoare a automatului, precum și orice semnal de ieșire, va fi calculat continuu, adică într-un circuit combinațional și luat în considerare sincron, adică doar când avem un front crescător al semnalului *clk*. Mai multe despre această paradigmă folosită se poate citi în secțiunea 6.2.
Dacă semnalul *RST* este activ, atunci toate registrele interne se vor reseta, valoarea lor devenind *0*, iar starea curentă a automatului va deveni starea *RESET*. Dacă *RST* nu este activ, registrele interne își vor păstra starea, iar dacă semnalul *go* este găsit activ, atunci registrele interne corespunzătoarea matricilor *R, B* se vor încărca cu valorile de la intrare *init_red*, *init_blue*.

[23] De fapt sunt două, unul pentru X și altul pentru Y.

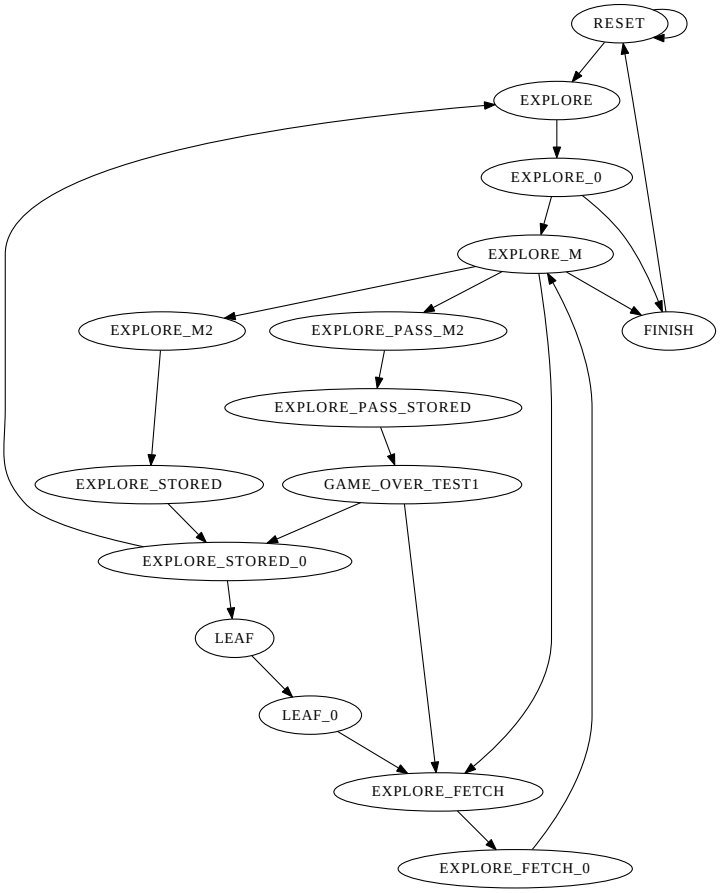

Figura 7.14: Automatul de explorare. Este de fapt doar un graf de stări, nefiind scrise și ieșirile/condițiile de tranziție.

În starea *RESET* avem două tranziții posibile. Automatul rămâne în această stare, până când primește semnalul *go*. Dacă semnalul de intrare *go* este activ, se va trece în starea *EXPLORE*, semnalul *thinking* va deveni 1 iar registrul intern pentru reprezentarea jucătorului, *pl*, se poziționează pe 1, reprezentând *roșu*. Starea *EXPLORE* tranziționează necondiționat în starea *EXPLORE_0*. Rolul acestei tranziții suplimentare, este de a aștepta rezultatul de la modulul *moves_map*. La ieșirile conectate la modulul *b_move* avem direct conectate ieșirile registrelor interne care reprezintă *R, B* și coordonatele mutării. Astfel, după o încărcare a registrelor interne, la ciclul următor de ceas, bistabilele vor bascula valorile de la intrare. Modulul *moves_map* este de asemenea conectat direct la reprezentarea internă pentru *R, B*, ceea ce înseamnă ca el va primi intrările un impuls mai târziu după ce au fost încărcate, de aceea este necesară starea de așteptare.
În starea *EXPLORE_0* putem culege deja ieșirile de la modulul *moves_map* și le vom încărca în reprezentarea internă a lui *M*, care după cum am spus, se va încărca cu un impuls mai târziu. În această stare se va ajunge tot timpul când se dorește explorarea unei stări noi, niciodată de la revenirea dintr-un nivel superior. Se vede și în 7.14 că există o singură tranziție spre această stare. Fiind un nod nou, vom porni cu o nouă valoare minmax, care se va seta în funcție de jucător. Aceasta poate fi $-\infty$ sau $+\infty$. Voi reprezenta aceste valori printr-un număr de 20 de biți, în care 1 bit reprezintă semnul, numerele fiind

reprezentate în complement de 2. În registrul *best_value* avem cea mai bună valoare minmax pentru nodul respectiv, iar în registrul *max_p* avem cea mai bună valoare primită de la stăriile de pe un nivel superior în arbore. În 7.12 ar însemna că *max_p* este valoarea primită de s_1 de la s_m și s_q, iar *best_value* este cea mai bună valoare pentru jucătorul respectiv în acea stare. Fiind o stare încă neexplorată, vom porni cu aceste valori setate pe $-\infty$ pentru jucatorul *roșu*, considerat algoritmul din FPGA și cu $+\infty$ pentru jucătorul *albastru* care este adversarul și va dori minimizarea scorului. Tot în această stare se face și testul de mutări vide. Dacă ne aflăm la nivelul 0 în arbore, adică starea inițială, iar matricea M este toată 0, adică $M = \emptyset$, înseamnă că nu avem mutări valide și trebuie să lăsăm adversarul să facă o mutare. Tranziția va fi spre starea *FINISH*, iar semnalul de *pass* va fi activat. În starea *EXPLORE_M* este inima algoritmului. Aici are loc selecția unei mutări din setul M determinat, ieșirile din *moves_map* fiind înregistrate deja în această stare, aici are loc și calcularea valorilor α și β ale algoritmului. Selectarea mutării se va face printr-un decodificator prioritar. Acest lucru ne oferă șansa să selectăm ca prima mutare o mutare considerată mai bună după poziționarea ei. Tehnica se numește *killer moves* 3.9.2, iar ca o reprezentare simplă, vom selecta colțurile prima dată.[7, 8] Această metodă de ordonare a mutărilor este foarte simpla, nu ne

Fragment sursă 7.11 Implementarea tehnicii *killer moves*, decodificatorul prioritar.

```
...
if ( M_q[0] ) begin
    X_d = 0;
    Y_d = 0;
end
else
if ( M_q[7] ) begin
    X_d = 7;
    Y_d = 0;
end
else
if ( M_q[63] ) begin
    X_d = 7;
    Y_d = 7;
end
else
if ( M_q[57] ) begin
    X_d = 1;
    Y_d = 7;
end
...
```

constă practic nimic, vine datorită designului. Este singurul tip de ordonare ce este permisă fără a consuma în plus resurse de timp. Îmbunătățirea globală a algoritmului poate fi însă foarte mare, deoarece un disc pus în colț, poate genera un scor foarte mare, devenind astfel optimă tăierea $\alpha - \beta$. Dacă nu avem mutări valide și suntem din nou pe nivelul 0 al arborelui, înseamnă că nu mai avem stări de explorat, iar automatul va tranziționa în starea *FINISH*, semnalul *done* fiind activat. Dacă nu mai avem mutări, dar nu suntem pe nivelul 0, înseamnă că am ajuns într-o stare în care ar trebui să facem *pass*, iar dacă explorăm nodul pentru prima dată, vom tranziționa în starea *EXPLORE_PASS_M2*.[24] Dacă nodul nu este explorat pentru prima dată și

[24] Aici mă refer la oricare jucător. Este vorba de cel ce e la rând pe nivelul p în arbore.

nu mai avem mutări, înseamnă ca nodul curent a fost epuizat și putem să ne întoarcem în arbore la nivelul $p-1$, vom tranziționa deci în starea *EXPLORE_FETCH*, iar jucătorul p va deveni $\neg p$, adică MIN și MAX își schimbă rolurile.[25] Urmează calculul valoriilor *min* și *max*. Considerăm că suntem la un nivel MAX, adică algoritmul trebuie să facă mutarea. Pentru fiecare nod, avem un registru în care am memorat cea mai bună valoare estimată.[26] Dacă o valoare primită de la un fiu din arbore, adică dacă una din strategii este mai bună decât cea aleasă până în acel moment, atunci vom actualiza cea mai bună valoare, precum și mutarea (adică perechea de coordonate (x,y)). Dacă cea mai bună valoare estimată momentan, adică tot *best_value*, este mai mare decât α, atunci putem îngusta fereastra și vom actualiza pe α. Nodurile pe nivelul MIN ce vor urma, au șanse mai mari să fie tăiate, deci timpul de explorare va fi mai mic. Urmează testul pentru β: Dacă estimarea noastră este mai mare decât β, facem tăierea, mergem la nivelul $p-1$, jucătorul p devine $\neg p$ și tranziționăm în starea *EXPLORE_FETCH*. În cazul în care ne aflăm pe un nivel MIN, atunci rolurile se inversează. Se va îngusta fereastra modificând pe β și nodurile vor fi tăiate de către α. Un alt lucru important ce se mai face

Fragment sursă 7.12 Min-Max cu $\alpha - \beta$. (exemplificare pentru MAX)

```
...
/* considerăm nod MAX */
if ( pl_q ) begin
    /* o estimare mai bună decât cea curentă */
    if (max_p_q > best_value_q ) begin
        /* actualizăm */
        best_value_d = max_p_q;
        /* dacă suntem pe nivelul 0, atunci modificăm și mutare aleasă */
        if ( sp_q == 0 ) begin
            best_X_d = last_X_q;
            best_Y_d = last_Y_q;
        end
    end
    else begin
        // we return best_value on upper level
        max_p_d = best_value_q;
    end

    /* modificăm fereastra */
    if ( best_value_q > alfa_q ) begin
       alfa_d = best_value_q;
    end

    /* tăierea beta */
    if ( best_value_q >= beta_q )  begin
        /* nivelul anterior în arbore */
        sp_d = sp_q - 1;
        /* p devine !p */
        pl_d = ~pl_q;
        state_d = EXPLORE_FETCH;
        max_p_d = best_value_q;
    end
end
...
```

în această stare, e că se va poziționa bitul corespunzător mutării selectate, pe 0, astfel modificându-se intrările de la registrul intern corespunzător matricii M. După un impuls de ceas, intrările vor fi preluate de bistabilele ce formează

[25] A nu se face confuzie cu p notația pentru nivelul la care explorăm în arbore.

[26] Adică, game-theoretic value.

registrul intern *M*, iar la ieșire vor fi basculate vechile valori. Orice valoare basculată va fi citită doar dacă e nevoie de ea.
În cazul tranziționării în starea *EXPLORE_M2*, înseamnă că am ales o tranziție și dorim trecerea la un nivel mai jos în arbore.[27] Trebuie deci să salvăm starea curentă. Starea curentă presupune matricile *R, B* și matricea *M* modificată, adică s-a pus pe 0 bitul corespunzător tranziției alese. Salvăm și valorile α, β și *best_value*. Pentru a fi salvate, înseamnă ca în registrele interne corespunzătoare să avem acele valori. Astfel, dacă în starea *EXPLORE_M* am modificat harta mutărilor posibile, modificarea va apărea în registru după un impuls de tact, adică în starea *EXPLORE_M2*. Scrierea în memorie durează un ciclu, astfel în memorie datele vor fi scrise abia în starea următoare. Starea *EXPLORE_M2* este deci o stare de așteptare, va aștepta ca datele să fie scrise în memorie. Adresa la care sunt scrise este dată de registrul în care avem memorat nivelul la care ne aflăm în arbore, adică registrul intern *sp*.[28] Semnalul de scriere în memorie, *we*, este activ în această stare. Un alt lucru important ce se întâmplă, este că toate valorile sunt pregătite și pentru modulul *b_move*, care va transforma date continuu, e datoria noastră să le culegem dacă avem nevoie de ele. În starea următoare deci, vom avea și un răspuns de la *b_move* care ne interesează, fiind o nouă stare a jocului, s-a făcut deci mutarea aleasă de decodificatorul prioritar.
În starea *EXPLORE_STORED* avem datele scrise în memorie. Tot aici, avem și ieșirile valide de la modulul *b_move*, care a necesitat un ciclu de tact. Vom încărca registrele interne corespunzătoare matricilor *R, B* cu valorile primite. Vom trece la alt nivel în arbore, deci jucătorul *p* devine $\neg p$. Avem o singură posibilitate, tranziționăm în starea *EXPLORE_STORED_0*. În 7.13 *sp_q* re-

Fragment sursă 7.13 EXPLORE_STORED_0: Testăm dacă avem un nod frunză sau continuăm explorarea.

```
...
If ( sp_q < MAX_DEPTH - 1) begin
    sp_d = sp_q + 1;
    state_d = EXPLORE;
end
else begin
    state_d = LEAF;
end
...
```

prezintă ieșirea registrului intern în care este memorat nivelul la care am ajuns în arbore. Dacă nivelul curent este cu minim 1 mai mic decât nivelul maximm până la care explorăm, atunci vom incremneta nivelul arborelui, explorăm deci mai adânc, ne întoarcem în starea *EXPLORE* unde se reia tot ciclul. Altfel, înseamnă că avem un nod terminal și tranziționăm în starea *LEAF*.
Starea *LEAF* este doar o stare de așteptare, intrările sunt pregătite pentru modulul *heuristics*. Intrările de care este nevoie, sunt: reprezentarea internă a matricilor *R, B, M*. De *M* este nevoie pentru a calcula *mobilitatea* (vezi 2.3.2). Nivelul la care ne aflăm în momentul în care suntem într-un nod terminal, este întotdeauna unul par, astfel întotdeauna dintr-o stare terminală va urma la

[27] Nivel mai jos, dar *p* va crește.
[28] Vine de la *stack-pointer*, pentru că această versiune de min-max necesită simularea unei stive.

rând jucătorul de pe nivelul 0. Acest lucru contează pentru a ști pentru cine este matricea M. Tranziționăm doar în starea *LEAF_0*.

Fragment sursă 7.14 starea LEAF_0

```
...
LEAF_0: begin
            /* starea viitoare */
            state_d = EXPLORE_FETCH;
            /* culegem valoarea euristicii */
            max_p_d = heur_w;
            /* p devine !p */
            pl_d = ~pl_q;
        end
...
```

În această stare, *LEAF_0*, culegem ieșirea modulului care a calculat euristica și urmează să ne întoarcem în arbore. Pentru asta, va trebui să citim din memorie starea de pe nivelul anterior. Vom tranzitiona deci în starea *EXPLORE_FETCH*. În registrul intern *max_p* se va reține valoarea euristicii și înseamnă cea mai bună estimare a nodului, indiferent de câți fii ar avea[29]. Jucătorul p devine $\neg p$, pentru că trecem la alt nivel în arbore. Se observă că nu decrementăm registrul *sp*, pentru că în 7.13 nu am incrementat în momentul în care urma un nod terminal. Motivul, e pentru că nu are rost, pentru că nu dorim să salvăm starea unui nod terminal.
În starea *EXPLORE_FETCH* în acest caz tranziționăm în *EXPLORE_FETCH_0*.
În starea următoare, vom citi valorile de la ieșirea memoriei, care sunt deja disponibile, prin faptul că nu am modificat registrul *sp*. Se vor încărca toate datele din memorie în registrele interne, adică: $< R, B, M >$, valoarea *best*, α, β, precum și bitul care indică dacă nodul este explorat pentru prima dată. Starea în care tranziționăm este *EXPLORE_M*. Motivul pentru care tranziționăm în această stare și nu în *EXPLORE*, este pentru că acum nu trebuie să așteptăm rezultat de la modulul *moves_map*, pentru că ne întoarcem de la un nivel de mai de jos din arbore, iar harta mutărilor pentru starea respectivă o avem în memorie.
Ajunși din nou în starea *EXPLORE_M*, presupun acum că tranziționăm în *EXPLORE_FETCH*, deci înseamnă cu nu mai sunt posibilități de explorat din această stare de joc și va trebui să mergem la un nivel mai sus în arbore. Acum are sens starea de așteptare, pentru ca trebuie să așteptăm actualizarea registrului *sp*.
În cazul în care nodul ce trebuie explorat nu are mutări posibile, înseamnă că jucătorul respectiv trebuie să cedeze rândul și tranziționăm în starea *EXPLORE_PASS_M2*. Trecem prin *EXPLORE_PASS_STORED*, configurația va fi scrisă în memorie, după care ajungem în starea *GAME_OVER_TEST1*. Testul este necesar pentru a vedea dacă și jucătorul următor are sau nu mutări posibile, iar în cazul în care nu are, înseamnă ca nodul respectiv este unul în care jocul se termină. În această stare, ieșirile de la modulul *RB_cnt* sunt valide, știm deci fiecare jucător ce scor are.
Dacă scorul adversarului este mai mare (vezi 7.15), atunci vom da un scor foarte mic (negativ) stării de joc, pentru că vrem să evităm să ajungem în

[29]Evident, în cazul unui nod terminal, nu va avea fii.

această stare. În caz contrarm, vom da ca și scor o valoare mare, la care se adugă diferența de discuri, astfel se va maximiza și scorul, va avea grijă să câștige cu un scor cât mai mare posibil. Vom trece mai departe în starea *EXPLORE_FETCH* a automatului, pentru că nodul a fost explorat, știm cine va câștiga chiar, ne întoarcem având în registrul *max_p* scorul.

În cazul în care nu e sfârșit de joc, înseamnă că doar un singur jucător are

Fragment sursă 7.15 starea GAME_OVER_TEST1

```
...
GAME_OVER_TEST1: begin
    // no more moves
    if ( M_w[63:0] == 64'b0 )  begin
    // we have game over!
    // maximize the score
        if ( cnt_score_R >= cnt_score_B )
        begin
            max_p_d = 15000 + (cnt_score_R - cnt_score_B);
        end
        // adversarul
        else begin
            // we loose
            max_p_d = -15000;
        end
        sp_d = sp_q - 1;
        state_d = EXPLORE_FETCH;
    end
    else begin
        // nu e game over
        state_d = EXPLORE_STORED_0;
    end
...
```

posibilitatea de a muta, iar explorarea continuă în mod normal, cu tranziție în starea *EXPLORE_STORED_0*.
În starea finală, *FINISH* se va ajunge în momentul în care am ajuns printr-o revenire la nivelul 0 în arbore și nu mai avem mutări posibile. Automatul va tranziționa în starea *RESET*, activând semnalul *done* și dezactivând semnalul *thinking*. Din starea *RESET* se va ieși doar dacă primim un impuls *go*, adică se dorește o nouă explorare.

7.8.3 Concluzii

Modulul *b_move* este în exterior, comunicarea cu el se face prin semnalele specificate în descriere. Modulele *moves_map*, *heuristics*, *RB_cnt*, *memory_bram* sunt inferate în interiorul acestui modul. În designul complet, sunt inferate în total două module de tipul *moves_map*, se putea și cu unul singur, dar nu era necesar, iar resursele ocupate de acestea sunt acceptabile. Fiecare modul va funcționa continuu, având la ieșire un rezultat la fiecare impuls de ceas, în afară de modulul de memorie unde semnalul de scriere, *we*, este activat doar în starea *EXPLORE_M2* și *EXPLORE_PASS_M2*, stările în care se dorește memorarea unei stări din joc s_i la care se adaugă informațiile necesare. Se observă acum cât de mult ajută la performanță modulele descrise anterior ca fiind rapide. Pentru un rezultat se așteaptă maxim 1 impuls de ceas, iar timpul total de explorare este direct influențat de aceste module. În total, modulul *game_ai* cu toate modulele ce le deține, ocupă 33% din resurse (numărul de

slice-uri). Numărul de bistabile ocupate, este de 456, ce reprezintă 4% din total.

7.9 Modulul de memorie

Modulul de memorie este implementat în BRAM (vezi 5.1.3). Din [28] ne este asigurată o foarte bună performanță a acestuia. Flexibilitatea cu care putem construi acest modul ne ajută foarte mult în designul nostru, practic putem construi orice tip de memorie dorim. Pentru acest algoritm, avem nevoie de o memorie adresabilă prin nivelul arborelui, adică maxim 16 celule[30]. Lungimea datelor este însă de 256 de biți. Avem practic o memorie $16x256$ cu un singur port.

7.9.1 Descrierea modulului

Schema bloc

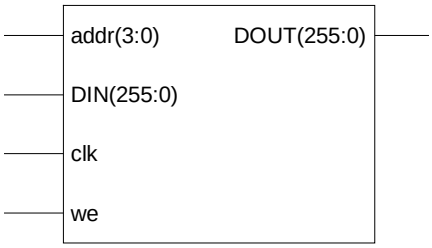

Figura 7.15: Schema bloc a modulului *memory_bram*.

Intrări

- *clk* impulsul de sincronizare, va fi legat la tactul global.
- *addr* linia de adrese, pe 4 biți.
- *we* semnalul de scriere în memorie, activ la 1.
- *DIN* datele de intrare, 256 de biți.

Ieșiri

- *DOUT* datele de ieșire, 256 de biți.

7.9.2 Detalii de implementare

Memoria este construită pentru a scrie în ea, sincron cu semnalul *clk*. Citirea se va face tot sincron. La fiecare front crescător a semnalului *clk*, dacă semnalul

[30]Mai mult de 16, explorarea ar dura prea mult.

Fragment sursă 7.16 Memoria

```
always @(posedge clk) begin
    if ( we ) begin
        bram[addr] <= DIN;
    end

    read_addr <= addr;
end

assign DOUT = bram[read_addr];
```

de scriere *we* este găsit activ, atunci se va scrie în memorie datele de la intrare, reprezentate prin *DIN*. Se va memora într-un registru intern adresa la care se face scrierea sau se accesează memoria.
În final citirea se face sincron, asignare continua, folosind registrul intern în care s-a memorat adresa. Asta va însemna că rezultatele de la ieșire le vom avea cu o mică întârziere, dată de bascularea bistabilelor ce compun registrul *read_addr*.

Reprezentarea datelor

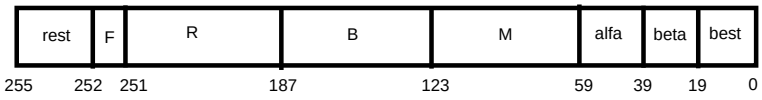

Figura 7.16: Reprezentarea datelor în memorie

Cam așa arată reprezentarea datelor în memorie (vezi 7.16. Valorile *alfa, beta, best* sunt reprezentate pe 20 biți ($19 + 1$ bit de semn). Urmează valorile ce definesc tabla de joc, matricile *M, B, R* fiecare pe câte 64 de biți. Urmează apoi 1 bit ce reprezintă dacă nodul respectiv e explorat prima dată sau nu. Biții rămași sunt rezervați.

7.9.3 Concluzii

Acest modul este de fapt o descriere *macro* a unei memorii. Programul de sintetizare va recunoaște formatul și intern va folosi memoria *BRAM*, nu va infera 16x256 bistabile. Totuși, dacă se dorește, se poate forța acest lucru. O astfel de memorie sintetizată cu bistabile, ar ocupa 77% din resurse! Mai mult decât intreg jocul Reversi implementat. Instruind programul de sinteză să folosească memoria BRAM, această memorie va ocupa 40% din capacitatea totală a BRAM-ului de pe această placă.

7.10 Numărarea discurilor

În contrucția algoritmului, este nevoie pentru diferite stări de joc de a computa scorul. La jocul Reversi, scorul însemnând câte discuri are fiecare jucător. Este greu de estimat pentru câte noduri din arbore va fi nevoie de a calcula scorul, dar făcând parte din algoritmul de explorare voi propune o computație rapidă,

într-un ciclu de ceas. Această nevoie, de a calcula scorul, apare atunci când un nod din arbore este un nod terminal iar algoritmul trebuie să decidă cine a câștigat. În cazul în care a câștigat adversarul, minmax va evita acea stare. Computația scorului, constă în numărarea de discuri din fiecare matrice *R, B*. Procesul este complet paralel, fiind matrici diferite. Bineînțeles, paralelizarea se poate face și la nivel de matrice.

7.10.1 Descrierea modulului

Schema bloc

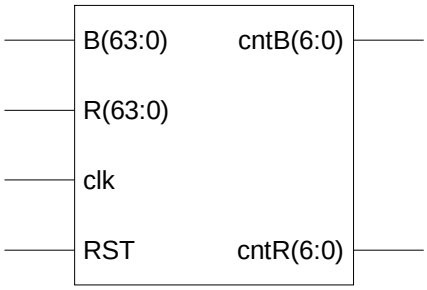

Figura 7.17: Schema bloc a modulului *RB_cnt*.

Intrări

- *clk* impulsul de sincronizare, va fi legat la tactul global.
- *RST* linia de reset, va fi legată la resetul global.
- *R* 64 biți, reprezintă matricea *R*.
- *B* 64 biți, reprezintă matricea *B*.

Ieșiri

- *cntR* 7 biți, reprezintă scorul pentru jucătorul *R* (maxim 64).
- *cntB* 7 biți, reprezintă scorul pentru jucătorul *B* (maxim 64).

7.10.2 Detalii de implementare

Modulul este format dintr-un proces sincron cu semnalul *clk* și un proces asincron. În procesul asincron se calculează suma fiecărui element din fiecare matrice, iar procesul sincron va memora în registrii rezultatul la fiecare front crescător al semnalului *clk*.

Metoda naivă

O metodă naivă constă în adunarea tuturor poziților din matrice formând un lanț.[31] Programatic, acest lucru s-ar face adunând fiecare bit într-un ciclu repetitiv. Hardware, prin aplicarea metodei ar rezulta tot un rezultat la fiecare ciclu de ceas, problema ar apărea la creșterea frecvenței, pentru că cel mai semnificativ bit va apărea după o întârziere mare, dată de întregul lanț de sumatoare, complexitatea fiind $O(n)$.

Metoda paralelă

Metoda paralelă este o implementare hardware a algoritmului descris de Knuth în lucrarea sa „Art of Computer Programming", secțiunea „Bitwise tricks and techniques" [33]. Algoritmul constă în a calcula pe rând sume de biți (u_{2j+1}, u_{2j}), rezultat care va forma un număr în baza 4 unde fiecare termen reprezintă suma a câtor 2 biți de la intrare. Calculul se repetă până când se ajunge la un singur număr care reprezintă suma biților intrării u. Hardware, putem considera o adunare arborescentă. La nivelul cel mai sus, însumăm câte 2 biți, iar la nivelul următor câte 4 biți și tot așa până la rădăcina arborelui care va face ultima însumare. Spre deosebire de varianta naivă aici complexitatea ar fi doar $log_2(n)$, pentru că întârzierea este dată de numărul de niveluri în arbore.
În această lucrare, eu am combinat cele două variante, motivul ar fi economisirea de resurse. Se va face o adunare în lanț pentru fiecare linie din matrice, după care rezultatul celor 8 linii se vor aduna din nou într-un lanț.

Fragment sursă 7.17 Adunarea în lanț a fiecărei linii.

```
...
cntB_p00_d = B[00] + B[01] + B[02] + B[03] + B[04] + B[05] + B[06] + B[07];
cntB_p01_d = B[08] + B[09] + B[10] + B[11] + B[12] + B[13] + B[14] + B[15];
cntB_p02_d = B[16] + B[17] + B[18] + B[19] + B[20] + B[21] + B[22] + B[23];
cntB_p03_d = B[24] + B[25] + B[26] + B[27] + B[28] + B[29] + B[30] + B[31];
cntB_p04_d = B[32] + B[33] + B[34] + B[35] + B[36] + B[37] + B[38] + B[39];
cntB_p05_d = B[40] + B[41] + B[42] + B[43] + B[44] + B[45] + B[46] + B[47];
cntB_p06_d = B[48] + B[49] + B[50] + B[51] + B[52] + B[53] + B[54] + B[55];
cntB_p07_d = B[56] + B[57] + B[58] + B[59] + B[60] + B[61] + B[62] + B[63];
...
```

7.10.3 Concluzii

Nu prea sunt multe de spus la acest modul, o întrebare interesantă ar fi... „Cum se poate face și mai rapid?". Nu am studiat încă bine problema, dar estimez că o variantă mai bună ar fi și folosirea arborilor *Wallace*[32], teoria lor poate fi găsită în [34] [35]. Procesoarele moderne Intel, conțin în setul de instrucțiuni SSE4 instrucțiuni specifice pentru a calcula numărul de biți într-un număr [36], operația fiind foarte folosită în multe domenii deoarece cel mai simplu o mulțime se reprezintă printr-un șir de biți, iar cardinalul mulțimii înseamnă de fapt problema de față. Putem formula problema existentă în

[31] În engl. chain adder.
[32] Arborii Wallace sunt folosiți la adunarea produselor parțiale la înmulțire.

Reversi în felul următor: „Să se afle $|R|$ și $|B|$ unde R,B sunt mulțimi formate din elementele r_i și b_i care reprezintă discurile pentru fiecare jucător aflate pe poziția i." Fiind vorba de Reversi, avem: dacă $r_i \in R \Leftrightarrow b_i \notin B$.

7.11 Euristica

Multe jocuri din lumea noastră precum șah, Go, Reversi dispun de conceptul de evaluare a poziției de joc. În fiecare stare de joc se va putea apreacia cât este de favorabilă pentru fiecare dintre jucători. Acest modul presupune generarea unui scor pentru o anumită stare a jocului. Nu s-a pus accent pe generarea unei funcții de evaluare competitivă, totuși, rezultatele obținute sunt destul de bune.
Funcția de evaluare este foarte dependentăde joc. Rezultatul unui program de Reversi este direct determinat de această funcție, ea înlocuind incapacitatea programelor de a explora complet arborele de joc. Jocul nefiind rezolvat, nu este cunoscută această funcție încât jucătorul să joace perfect. În general, aceste funcții sunt mai mult empirice decât determinate științific, fiecare constând într-o serie de parametrii și de observații. Cercetarea în acest domeniu a variat, de la algoritmi statici la algoritmi dinamici, unde ponderile se schimbă pe parcursul jocului [37], la rețele neuronale, programare genetică sau învățare[38]. Un alt concept, nelipsit din programele de Reversi profesionale, este *cartea de deschidere*[33][39]. Cartea de deschidere presupune o bază de date imensă cu de diferite stări de joc, din începutul jocului. După o partidă jucată, se poate determina retro dacă o mutare de la început a fost una bună sau nu. O astfel de bază de date conține sute de mii de combinații.
Acesta este modulul care implementează funcția u_{s_i} și vom avea nevoie de un rezultat la fiecare nod terminal al arborelui. Pentru acest lucru, este evident că va trebui proiectat încât să furnizeze un rezultat suficient de repede. Este ultimul modul critic prezentat, în care mi-am propus ca un rezultat să poată fi furnizat la fiecare impuls de tact. Algoritmul este bazat pe favorizarea discurilor stabile, care vor contribui direct la scor și mobilitatea, un aspect foarte important în Reversi. Implementarea este statică, fără parametrii ajustabili și gândită astfel încât să poată fi hardware foarte ușor de implementat. Generarea unui scor pe structurile stabile a fost inspirată din lucrările lui Michael Buro [37], el obținând rezultate foarte bune, dar cu modele mult mai sofisticate decât cele prezentate aici.

7.11.1 Descrierea modulului

Denumire

heuristics

Schema bloc

Intrări

- *clk* impulsul de sincronizare, va fi legat la tactul global.

[33] În engl. opening-book.

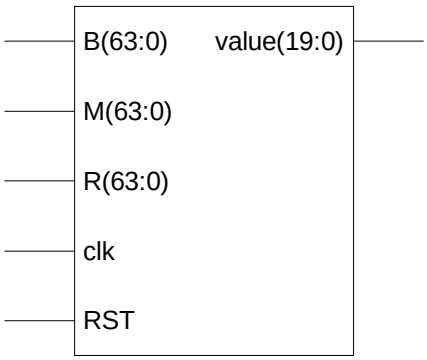

Figura 7.18: Schema bloc a modulului *heuristics*.

- *RST* linia de reset, va fi legată la resetul global.
- *R* 64 biți, reprezintă matricea *R*.
- *B* 64 biți, reprezintă matricea *B*.
- *M* 64 biți, reprezintă matricea *M* pentru jucătorul curent aflat în starea terminală.

Ieșiri

- *value* 20 biți, reprezintă scorul atribuit stării formată din intrările (R, B). Numărul este cu semn.

7.11.2 Detalii de implementare

Un singur lucru este cert la o stare de joc din Reversi: discurile stabile vor contribui direct la scor. Și această euristică își propune acest lucru, de a favoriza construcția de discuri stabile. Reprezentarea foarte bună a tablei ne permite o potrivire de șabloane[34] extrem de rapidă. O singură potrivire este de fapt doar o operație *AND* între starea jocului și șablon, care este reprezentat tot pe 64 de biți pentru a face posibilă această comparare rapidă. Putem exprima astfel:

$$s_i^R \wedge w_0 = w_0 \qquad \text{dacă tabla de joc conține șablonul } w_0. \tag{7.14}$$

În (7.14) s_i^R reprezintă componenta R a stării s_0.

În 7.19 se pot vedea 6 din cele 48 de șabloane implementate. Bineînțeles, un șablon poate eclipsa un alt șablon (adică îl va conține), dar acest lucru nu e neaparat de evitat pentru că scoate în evidență importanța unor discuri. Pentru fiecare șablon se acordă un scor prestabilit. Dacă un anume șablon este conținut de către matricea B, adică de adversar, atunci se va acorda același scor, dar negativ. În acest fel algoritmul de explorare va face tot posibilul să evite mutările unde adversarul ar avea multe discuri stabile.

[34] În engl. patterns.

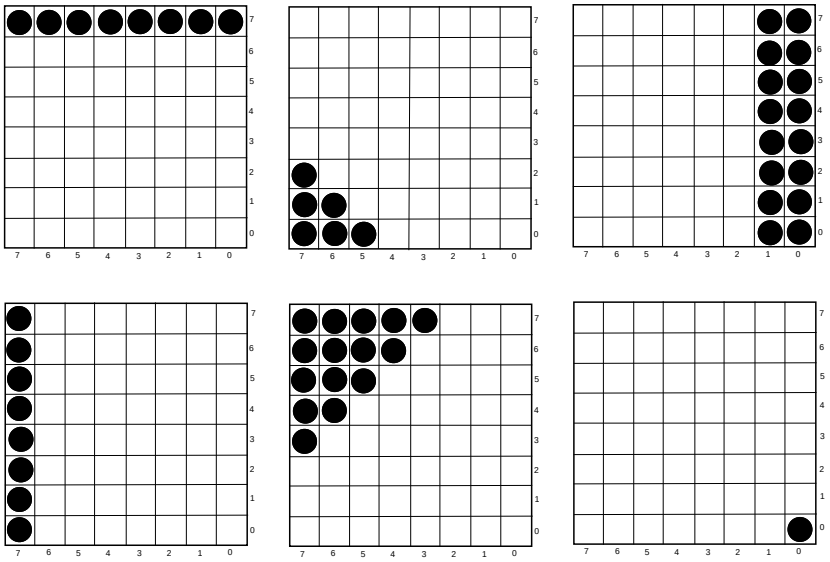

Figura 7.19: Exemple de șabloane: discuri stabile.

$$u_0 = \sum_{i=0}^{n} x_i \qquad x_i = \begin{cases} h_i & \text{dacă } s_0^R \wedge w_i = w_i \\ -h_i & \text{dacă } s_0^B \wedge w_i = w_i \end{cases} \tag{7.15}$$

În (7.15) h_i reprezintă ponderea asociată tiparului i. Metoda este ușor de implementat în hardware și este ușor de paralelizat.

Calcularea x_i

Vom calcula toate scorurile x_i în paralel. Abordarea este în felul următor: Pentru fiecare șablon avem h_i dacă șablon este închis în R, altfel avem 0 iar în același timp avem $-h_i$ dacă este inclus în B, sau 0 altfel. Bineînțeles, nu putem avea pentru aceași stare de joc h_i și $-h_i$. În final urmează să adunăm toate aceste scoruri în număr de $2 \cdot 48$.

Fragment sursă 7.18 Exemplu de potrivire de șabloane.

```
...
assign pattern06_Rd = ((R[63:0] & 64'h000103070F1F3F7F) == 64'h000103070F1F3F7F) ?  7 : 0;
assign pattern06_Bd = ((B[63:0] & 64'h000103070F1F3F7F) == 64'h000103070F1F3F7F) ? -7 : 0;
assign pattern07_Rd = ((R[63:0] & 64'h0103070F1F3F7FFF) == 64'h0103070F1F3F7FFF) ?  8 : 0;
assign pattern07_Bd = ((B[63:0] & 64'h0103070F1F3F7FFF) == 64'h0103070F1F3F7FFF) ? -8 : 0;
...
```

Avantajul extraordinar față de o variantă software, este calcularea în paralel a tuturor tiparelor. Circuistica generată va conține foarte multe porți *AND*, iar apoi scorurile generate x_i vor intra în sumatoare pentru a fi adunate.

Însumarea scorurilor

Ca și cum am spus în 7.10, o soluție naivă ar consta într-o adunare în lanț a valorilor. Algoritmul având o complexitate $O(n)$ ar fi prea lent, constrângerile

de timp nefiind satisfăcute. Metoda arborescentă, care construită într-un mod corect are o complexitate de $log_2(n)$ va da rezultate mult mai bune [34] [35]. În această lucrare am combinat cele două metode rezumându-mă la un sumator în lanț cu 4 termeni. Întârzierea va fi deci $4 \cdot t_A$ pentru fiecare nivel din arbore.

Fragment sursă 7.19 Exemplu de adunare în paralel a scorurilor.

```
...
value_Rp0_d = pattern00_Rd + pattern01_Rd + pattern02_Rd + pattern03_Rd;
value_Rp1_d = pattern04_Rd + pattern05_Rd + pattern06_Rd + pattern07_Rd;
value_Rp2_d = pattern08_Rd + pattern09_Rd + pattern10_Rd + pattern11_Rd;
value_Rp3_d = pattern12_Rd + pattern13_Rd + pattern14_Rd + pattern15_Rd;
...
```

La scorul final, se va adăuga și numărul de mutări valide. Acest număr se obține exact ca la numărarea scorului (vezi 7.10) doar că operația se face asupra matricei M. Scorul final considerat este de forma:

$$scor = p_1 \cdot stabilitate + p_2 \cdot mutabilitate \tag{7.16}$$

Mutabilitatea reprezintă numărul de mutări posibile pentru jucătorul curent. Valorile p_1 și p_2 sunt determinate empiric, momentan ele fiind puteri ale lui 2, pentru un calcul rapid, timp 0 practic, în hardware.

7.11.3 Concluzii

Am prezentat o funcție de evaluare simplă și destul de eficientă, dar cel mai important pentru scopul lucrării, foarte rapidă. Bineînțeles, se poate îmbunătăți și această reprezentare, de exemplu la adunarea scorurilor prin folosirea unor algoritmi rapizi de *multi-operand adders* [35], sau prin folosirea arborilor Wallace [34] [35]. Arborii Wallace se bazează pe observația că: dacă adunarea a două numere $x+y$ necesită un timp $O(n)$, atunci adunarea a 3 numere $x+y+z$ va necesita în plus doar un timp constant [34]. Astfel, trei termeni ce necesită adunați îi putem transforma în problema adunării a 2 termeni, tot așa până la un număr de 2 termeni pe care să-i adunăm cu un sumator complet.
Dezavantajul implementării în hardware a funcțiilor euristice, este că se pierde mult din flexibilitate. Pe un procesor general, se vor putea implementa algoritmi mult mai flexibili, precum cei bazați pe învățare.

7.12 Închegarea modulelor

A mai rămas un singur modul de descris, cel principal. Acest modul este motorul ce va face ca întreg sistemul să funcționeze, el va permite și jocul cu un jucător uman. Nu este nimic critic în acest modul, contează doar funcționalitatea lui. Pot spune că modulul este „arbitrul" jocului, el va da startul modulului de AI, va aștepta după el, după care va lua mutarea calculată și va face schimbarea pe tablă, după care va lăsa jucătorul uman să facă mutarea, va trimite datele pe interfața serială și tot așa. Totul este implementat sub forma unui automat finit.

7.12.1 Descrierea modulului

Denumire

reversi

Dependințe

Modulul *b_move* (vezi 7.7).
Modulul *moves_map* (vezi 7.2).
Modulul *game_ai* (vezi 7.8).
Modulul *xy_calculate* (vezi 7.5).
Modulul *time_analysis* (vezi 7.7).
Modulul *vga_controller* (vezi 7.4).

Schema bloc

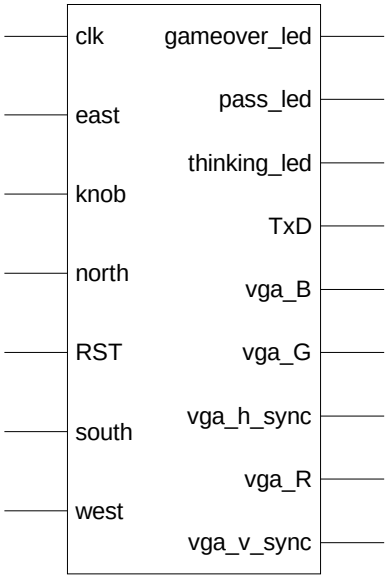

Figura 7.20: Schema bloc a modulului principal.

Intrări

- *clk* impulsul de sincronizare, va fi legat la tactul global.
- *RST* linia de reset, va fi legată la resetul global.
- *east* linia ce este legată în exeterior la butonul pentru o deplasare spre dreapta. (vezi 7.5)
- *west* linia ce este legată în exeterior la butonul pentru o deplasare spre stânga. (vezi 7.5)

- *north* linia ce este legată în exeterior la butonul pentru o deplasare în sus. (vezi 7.5)
- *south* linia ce este legată în exeterior la butonul pentru o deplasare în jos. (vezi 7.5)
- *knob* linia ce este legată în exeterior la butonul pentru poziționarea discului. (vezi 7.5)

Ieșiri

- *gameover_led* semnal ce va fi conectat la un led de pe placă, printr-o rezistență de 390Ω. Aprinderea ledului semnifică sfârșitul unui joc.
- *pass_led* semnal ce va fi conectat la un led de pe placă, printr-o rezistență de 390Ω. Aprinderea ledului semnifică faptul că jucătorul uman nu are mutări posibile și va muta din nou adversarul.
- *thinking_led* semnal ce va fi conectat la un led de pe placă, printr-o rezistență de 390Ω. Aprinderea ledului semnifică faptul că modullul *game_ai* încă explorează.
- *TxD* va fi conectat la pinul Tx la portul serial de pe placă.
- *vga_h_sync* reprezintă impulsul de sincronizare orizontală, va fi conectat la portul VGA.
- *vga_v_sync* reprezintă impulsul de sincronizare verticală, va fi conectat la portul VGA.
- *vga_R* componenta R a controler-ului VGA.
- *vga_B* componenta B a controler-ului VGA.
- *vga_G* componenta G a controler-ului VGA.

7.12.2 Detalii de implementare

Logica jocului este reprezentată printr-un automat finit. La inițializare, adică în momentul în care linia *RST* este activă, tabla de joc va avea poziția standard [7], adică două discuri roșii și două albastre, în centru.

Automatul finit

Imediat după reset, jocul începe în starea *HUMAN*, automatul va rămâne în această stare până când omul va dori poziționarea unui disc și va apăsa butonul pentru poziționare. Dacă poziția selectată de om, este validă (adică există în harta de mutări calculată), atunci se va reține în registre coordonata X și Y selectată și se va trece în starea *HUMAN_MOVE*.
Starea *HUMAN_MOVE* are rolul doar de a aștepta ca la intrarea modulului *b_move* coordonatele memorate în starea anterioară să fie stabile. Se va tranziționa în starea *HUMAN_MOVE_WAIT*. În această stare avem calculată

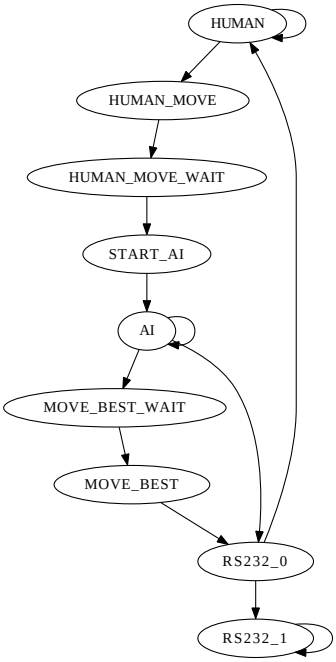

Figura 7.21: Automatul finit ce reprezintă logica de joc. Alternarea dintre mutările omului și a jucătorului artificial.

noua stare de joc în care jucătorul uman a făcut mutarea. Se pune rezultatul în registrele ce reprezintă tabla de joc care e vizualizată pe ecran, se complementează jucătorul curent pentru că urmează ca adversarul să facă o mutare și se trece în starea *START_AI*. Se folosește o singură instanță a modulului *b_move*, iar printr-un multiplexor, în această stare el va fi conecta la modulul *game_ai*. Partajarea acestei resurse este ușoară, pentru că știm că nu se poate ca evenimentele să apară simultan, iar un simplu multiplexor la intrările și ieșirile modulului *b_move*, care va fi conectat când la AI, când la tabla vizibilă pe ecran, este suficient. Având toate intrările pregătite pentru modulul *game_ai*, registrul în care memorăm timpul de explorare este resetat, se va tranziționa în starea *AI*.
În starea *AI*, modulul *game_ai* a început explorarea, în cazul în care nu are mutări posibile, acest lucru va fi semnalat prin linia *pass*, și se va tranziționa în starea *RS232_0*. În cazul în care semnalul *done* devine activ, atunci modulul a terminat de explorat și automatul va tranziționa în starea *MOVE_BEST_WAIT*. Dacă explorarea continuă, atunci automatul rămâne în starea *AI* iar numărătorul în care memorăm timpul de gândire, se va incrementa.
În starea *MOVE_BEST_WAIT* automatul va tranziționa în *MOVE_BEST*, iar modulul *b_move* fiind deja „deconectat" de la modulul *game_ai*, are intrările pregătite pentru a face vizibilă mutarea AI-ului.
În starea *MOVE_BEST* se vor actualiza registrele ce reprezintă tabla de joc cu ieșirile de la modulul *b_move*, după care se va tranziționa în starea *RS232_0* pentru a trimite timpii de gândire pe interfața serială. Tot aici, se complementează jucătorul, pentru că va urma jucătorul uman la rând.
În starea *RS232_0* semnalul de start pentru modulul RS232 devine activ, după care tranziționăm în starea *RS232_1*. În *RS232_1* se va verifica starea semnalului *TxD_busy*, iar dacă este 1, atunci rămânem în aceași stare. Dacă transmisia

serială s-a terminat, adică *TxD_busy* este 0, atunci tranziționăm în starea *HUMAN*, după care se va repeta.
Semnalul *pass* este calculat simplu, este comparată matricea M cu 0. Sfârșitul jocului va fi atunci când vor fi active semnalele *pass* și *ai_pass*.

7.12.3 Concluzii

Versiunea existentă poate avea mici modificări față de ce e descris aici, pentru că pe parcurs s-a dorit transmiterea la un calculator a mai multor informații pentru depanare sau statistică, de exemplu: numărul de noduri evaluate, valoarea minmax, etc.

7.12.4 Testarea modulelor

Testarea modulelor s-a făcut folosind programele ISim și ModelSim. ISim s-a dovedit folositor pentru modulele mai mici, iar la modulele mari a fost imposibil de folosit. Modulele simulate cu ISim au fost simulate comportamental dar s-a simulat și *post-route*. A fost creat pentru fiecare modul, un modul de test corespunzător, cu anumite cazuri de test construite de autor. Cu ModelSim s-a reușit simularea în întregime, și a modulului de top, adică închegarea tuturor modulelor. S-au descoperit unele erori de proiectare în modulul *game_ai* și datorită testării, au fost identificate și rezolvate.

Capitolul 8

Rezultate

Mașina contruită a fost testată dacă e capabilă de un joc corect, cu ajutorul altor programe. Primul ar fi cel contruit de către mine, programul cu care urmează să compar mașina rezultată. Este vorba de un program scris pe platforma .NET iar testele au fost făcute pe un sistem de operare Windows, pe un calculator cu procesor Pentrium IV ce rulează la o frecvență de 3GHz și 1.5GB memorie RAM. Mașina proiectată a fost testată și cu alte programe și s-a dovedit a fi capabilă de un joc corect, lucru rezultat și din testarea modulelor (vezi 7.12.4). Programele cu care s-a mai testat au fost: Zebra, Saio, Pocket PC Reversi, Microsoft Reversi. S-a testat și cu jucători umani într-o serie relativ mare de partide (eu estimez pe la 200-300 de partide). Prin urmare, consider mașina capabilă de un joc corect, respectând regulile jocului fără defecțiuni vizibile momentan.

Comparația cu versiunea pe PC

Scopul imediat al acestei lucrări a fost de a obține un plus de performanță față de varianta jocului programată pe un PC, tot de către autor. Rezultatele s-au obținut după mai multe partide de joc, la diferite adâncimi de explorare, iar în final pentru reprezentare s-a ales adâncimea 6. Această adâncime a fost aleasă pentru că mai mult de atât, varianta pe PC nu ar fi făcut o mutare într-un timp rezonabil. Graficul (8.1) reprezintă un joc între mine și construită și programul pe PC. Răspunsurile circuitului din FPGA și al programului pe PC vor fi identice, pentru că a fost implementați exact aceași algoritmi, aceași adâncime de explorare și aceași euristică, prin urmare, graficul are sens. Tot ce este diferit între implementarea pe PC și FPGA, este timpul de răspuns pentru fiecare mutare. Se observă că diferența este foarte mare, în favoarea implementării în FPGA, chiar mai mare decât se aștepta autorul. Reprezentarea este pe o scară logaritmică pentru că diferențele foarte mari dintre valori nu ar face vizibile concluziile. Făcând mai multe teste, s-a calculat un factor de accelerare de aproximativ 250 de ori pentru implementarea în FPGA. Accelerare aici înseamnă efectiv de câte ori este mai rapidă o implementare decât alta. Această testare trebuie luată ca fiind orientativă, și interpretat exact rezultatul final, adică: implementarea FPGA este mai rapidă ca implementarea curentă pe PC, pentru că testarea este influențată și de încărcarea sistemului de operare și de mulți alți factori. Platforma .NET este de fapt o

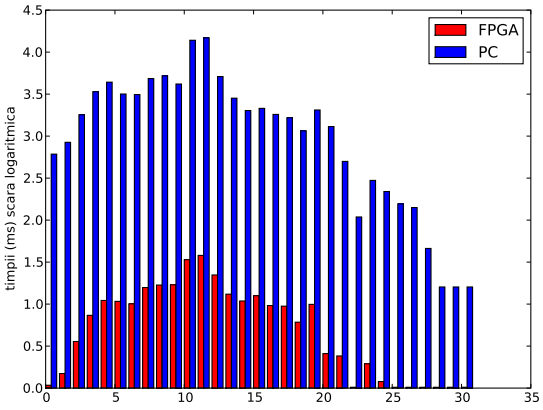

Figura 8.1: FPGA vs PC, reprezentare pe scară logaritmică.

mașină virtuală, care iar încetinește mult aplicația.

Comparația cu alte programe

În graficul (8.2) avem reprezentat numărul de noduri evaluate pentru fiecare mutare în timpul unei partide de joc cu programul Zebra.

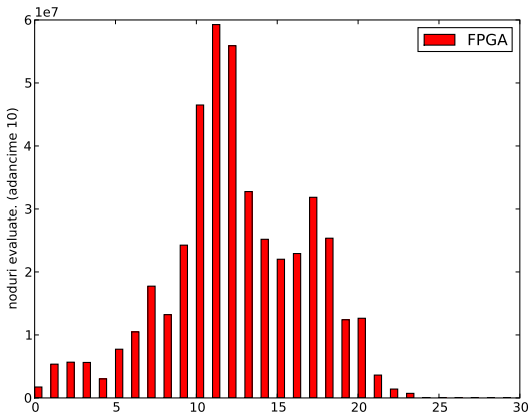

Figura 8.2: Numărul de noduri evaluate la fiecare mutare pentru o partidă de joc. (FPGA)

După ambele două grafice se pot distinge destul de clar fazele de joc. Se observă că în faza de mijloc a jocului, factorul ramificare a arborelui este cel mai mare, pentru că sunt un număr mare de mutări posibile. Spre faza de sfârșit, factorul descrește semnificativ, și numărul de noduri analizate este mai mic și timpii de evaluare sunt mai mici. Prin această observație am putea determina de la ce mutare este fiabilă o căutare *full* până la sfârșitul jocului. În cazul de față, se pare că o adâncime în arbore de 16 niveluri în faza de sfârșit,

ar duce la o determinare a valorii minmax finale în resurse de timp rezonabile. După mai multe teste, s-a calculat că mașina poate evalua aproximativ 5 milioane de noduri/secundă. Programul Zebra poate evalua între 1.5-2M noduri/secundă în faza de mijloc a jocului (conform autorului Gunnar Andersson), ceea ce înseamnă că circuitul implementat poate evalua de aproximativ 3 ori mai multe noduri decât Zebra. Programul Zebra este probabil cel mai rapid program de Othello implementat pe PC. Pentru calcule interne, acesta folosește instrucțiuni SIMD din setul MMX de la Intel pentru optimizare. Totuși, aceste rezultate oarecum în favoarea implementării mele, nu trebuie nicidecum considerate că această implementare este competitivă cu programul Zebra care are o euristică excepțională, cu valori calculate și cântărite după mii de partide de joc. Mai mult decât atât, Zebra are implementați diferiți algoritmi de căutare selectivă (vezi 3.9.4) ceea ce în final îl face mai rapid și mai bun decât implementarea mea. De aceea testul cu Zebra nu este relevant, este doar orientativ.

Capitolul 9

Concluzii

Este minunat cum tehnologia de azi ne permite să contruim anumite mașini, care acum 50 de ani erau gânduri puse pe hârtie a unor oamenii de știință precum Neumann, Turing, Shannon.
O proiectare hardware pentru o anumită aplicație deschide noi orizonturi, gândirea este complet diferită față de o gândire secvențială în momentul în care programăm o mașină obișnuită. Am încercat să arăt o posibilă implementare a joclui Othello pe un FPGA, rezultatele fiind optimiste aș zice, s-ar putea construi o astfel de mașină care să ducă la un joc extrem de rapid și de competitiv. Problema rezolvată este o problemă de căutare în spațiul stărilor, care poate avea o mulțime de aplicații, nu doar acest joc, iar un design hardware s-ar putea să fie o metodă de rezolvare mult mai rapidă și în final chiar mai flexibilă datorită paralelizării, decât programarea unui calculator obișnuit. A fost nevoie de Deep Blue, un adevărat monstru tehnologic, pentru a putea concura cu un campion mondial la Șah. Deep Blue avea o putere de analiză de 200M noduri/secundă. Impresionant, ... impresionant creierul uman care poate face față la o astfel de mașinărie. În 50 de ani, după cum am spus, s-a ajuns unde alții nici nu își imaginau și totuși, suntem foarte departe de adevăr, încât încercăm să copleșim mintea omului doar prin generarea tuturor combinațiilor posibile și prin selectarea unui subset pe baza unei euristici. Mintea omului este capabilă de o astfel de selecție inimaginabilă, campionul de șah nu va gândi n mutări înainte, va avea altfel structurată informația în creier, strategiile și mutările bune fiind în atenția lui. Othello nu a fost rezolvat decât pe o tablă 6x6, jocul de Șah a fost rezolvat decât pe o tablă 3x3 și fazele de final cu oricare 6 piese pe o tablă 8x8. Jocul Go este intangibil din punctul de vedere al mașinilor. Cel mai bun calculator la Go pierde în fața unui jucător novice.[14] Este puțin probabil găsirea unui algoritm ce să funcționeze în timp rezonabil pentru rezolvarea jocurilor, acestea făcând parte din clasa de probleme NP.
Care ar fi scopul rezolvării jocurilor? Fiecare joc crează o nouă lume, cu un spațiu enorm de stări. Lumea este creată de regulile jocului. Un joc rezolvat este o mică lume rezolvată. Adevăratul sens apare după rezolvare, când pe baza rezultatelor obținute se pot observa foarte multe lucruri [14]. Scopul calculatoarelor ar putea fi de a reuși să transpună aceste rezultate obținute în reguli care mai apoi să fie asimilate de creierul uman. De exemplu, pentru jocul „X și 0”, oricine poate dezvolta foarte rapid o strategie pentru a avea cel

puțin remiză. Acest lucru se întâmplă pentru că jocul este rezolvat, este cunoscut. Dacă mintea umană ar putea asimila un set mult mai mare de reguli, am putea dezvolta o astfel de strategie pentru jocuri/situații mai complicate, care ne-ar lărgi apoi orizontul cunoașterii, am fi capabili de a aborda probleme și mai complicate.

Lista fragmentelor

Listă de figuri

Bibliografie

[1] K. L.-B. Yoav Shoham, *MULTIAGENT SYSTEMS: Algorithmic, Game-Theoretic, and Logical Foundations*. 2009.

[2] L. V. Allis, *Searching for Solutions in Games and Artificial Intelligence*. PhD thesis, University of Limburg, Maastricht, 1994.

[3] C. Shannon, "Programming a computer for playing chess," in *Philosophical Magazine*, 1949.

[4] A. J. H. J. Feng-hsiung Hsu, Murray Champbell, "Deep blue," tech. rep., Compaq, IBM Watson Research Center, Sandbridge Technologies, 2001.

[5] M. Buro, "Takeshi murakami vs. logistello," tech. rep., NEC Research Institute, 1997.

[6] M. Buro, "The iwec-2002 man-machine othello match," tech. rep., University of Alberta, 2002.

[7] R. Fang, *Othello: From Beginner to Master*. 2003.

[8] B. Rose, *Othello: A minute to learn... A lifetime to master*. 2005.

[9] B. v. S. Theodore L. Turocy, "Game theory," tech. rep., CDAM Research Report, 2001.

[10] M. J. Osborne, *An Introduction to Game Theory*. Oxford University Press, 2002.

[11] C. A. Giumale, *Introducere în Analiza Algoritmilor*. POLIROM, 2004.

[12] D. S. Nau, "On game graph structure and its influence on pathology," *Computer and Information Sciences*, 1983.

[13] T. sheng Hsu, "An analysis of alpha-beta pruning by d.e knuth and r.w moore," 11 2008.

[14] J. v. R. H. Jaap van den Herik, Jos W.H.M. Uiterwijk, "Games solved: Now and in the future," *Artificial Intelligence*, 2002.

[15] D. McAllester, "Game search," *Artificial Intelligence*, 1992.

[16] R. L. Rivest, "Game tree searching by min/max approximation," tech. rep., MIT Laboratory for Computer Science, 1995.

[17] M. G. Brockington, *Asynchronous Parallel Game-Tree Search.* PhD thesis, University of Alberta, 1998.

[18] D. S. Nau, "Game tree search," 02 2009.

[19] M. Buro, "Multi-probecut search," 2002.

[20] M. Buro, "An effective selective extension of the $\alpha\beta$ algorithm," tech. rep., ICCA Journal, 1995.

[21] J.-P. Deschamps, *Synthesis of Arithmetic Circuits: FPGA, ASIC and Embedded Systems.* John Wiley & Sons, 2006.

[22] Xilinx, *Spartan-3 Generation FPGA User Guide*, 2009.

[23] Xilinx, *Spartan-3 FPGA Family Data Sheet*, 2008.

[24] Xilinx, *Spartan-3E Starter Kit Board User Guide*, 2006.

[25] P. M. Nyasulu, *Introduction to Verilog.* 2001.

[26] N. A. Kartik Subramanian Iyer, *High Performance FPGA Designs.* 2006.

[27] J. Marshall, *RTL Coding and Optimization Guide for use with Design Compiler.* 2002.

[28] P. Garrault and B. Philofsky, *HDL Coding Practices to Accelerate Design Performance*, 2006.

[29] Xilinx, *XST User Guide*, 2005.

[30] *VGA Signal 640 x 480 @ 60 Hz Industry standard timing.* http://microvga.com/vga-timing/640x480@60Hz.

[31] *VGA timing information.* http://www.epanorama.net/documents/pc/vga_timing.html.

[32] *FPGA for Fun: Serial interface (RS-232).* http://www.fpga4fun.com/.

[33] D. E. Knuth, *The Art of Computer Programming*, vol. 4. Addison-Wesley, 2008.

[34] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms.* MIT Press and McGraw-Hill, 2001.

[35] K. Hwang, *Computer Arithmetic: Principles, Architecture, and Design.* John Wiley & Sons, 1979.

[36] Intel, *Intel SSE4 Programming Reference*, 2007.

[37] M. Buro, "Statistical feature combination for the evaluation of game positions," tech. rep., NEC Research Institute, 1995.

[38] M. Buro, "Improving heuristic mini-max search by supervised learning," tech. rep., NEC Research Institute, 2001.

[39] M. Buro, "Toward opening book learning," tech. rep., NEC Research Institute, 1999.