DESIGN & ARCHITECTURE

with Reconfigurable Logic of Data Structures for the Game of

GO

STYLIANOS S. KOKKALIS

The Technical University of Crete

TABLE OF CONTENTS

Chapter 1

Introduction or where one learns why this Thesis was written

Chapter 2

A Brief Review of Computer Games

☞ 2.1 *Classification of Games*

☞ 2.2 *The Game of Go*

2.2.1 *History*

2.2.2 *Rules*

2.2.3 *Computational Complexity*

2.2.4 *History of Computer Go*

Chapter 3

State of the Art in Computer Go

☞ 3.1 *Board Representation*

3.1.1 *The Board itself*

3.1.2 *The stone*

3.1.3 *The move & the capturing*

☞ 3.2 *Move Generation*

3.2.1 *Classical approach: Divide & Conquer*

3.2.2 *New approach: Monte Carlo Tree search*

Chapter 4
A Go Board in Hardware or where the architecture of our design is built from ground-up.

☞ 4.1 *Performance through Concurrency*

4.1.1 *A word on concurrency*

4.1.2 *Erlang*

☞ 4.2 *Design of a Parallel Board*

☞ 4.3 *Design of a Simulator*

Chapter 5
Design & Implementation or where the ideas of Chapter 4 seed on the ground of a true hardware circuit

☞ 5.1 *Towards a Master Slave design*

☞ 5.2 *Move Generating Module*

☞ 5.3 *Circuit Evaluation*

Chapter 6
Future Work

Acknowledgements

Who should I thank first? Vasilis who will be forever my good neighbour, Alex who takes you into his heart and his mind who like the bee collects from everybody and then generously offers it to you, the adorable and kind but with a wicked sense of time Thanasis, Gounis who is still experimenting nowadays with the inventive nature of the man of the caves, Juliana who shows us every day that life is light like the steps of salsa, Anadim who paints life in emotion-coloured poems and then transmits them to the galaxies, Mitsakos who is always there when you need him, Grammatikakis whose generosity is inversely proportional to his weight, the mother of us all Malamo, the most noble but nobly numb Dinos, the stylishly lunatic Miltos, my deeply beautiful and easygoing Natasa, the kind but "how damn kind can he be?" Meletis, my friends Meropi and Tsbaou who walk together in life and Art, Kyriakos who emerges you into his world written in beautiful lines of C and some day for the good of us all he will turn to Man, Myrtia who lets your spirit travel to places of outworldly beauty, the folklore Bilakakos who gets on the stage and with a wave of his hand makes you fly, my dear Danai with whom we walk on parallel paths who is always transforming and always hesitant, my sweet dynamic Kakia who puts everybody in his place and who has surely gone through more organized phases in her life (*for which we are to blame, too*), the sincere and joyful but relationship-experimentalist Niki who sings and we the rest steal from her notes to fill again, the cheerful (*ok Fotini, overcheerful)* but striving to find her path Eleni, Tachos who smiles at everyone so that the very heavens shine down upon us, my beloved Stergios who is constantly flirting with madness and sanity showing us all how to be free, the Lawyer who loves and unites everyone under his wings, my professor who saw the arrow on me and became the bow giving me direction and strength.

This Thesis is finally the result of 24 years of love which my family has given to me, my mother who sees herself and the whole universe in me, my father who quietly and discretely connects all the family, and my sister which only with a volcano eruption can be compared who always leaves me a better human.

Ποιόν να πρωτοευχαριστήσω; Τον Βασίλη που θα είναι ο παντοτινός μου καλός μου γείτονας, τον Άλεξ που σε βάζει στην καρδιά του και στο μυαλό του, σαν την μέλισσα συλλέγει από όλους και μετά σου το προσφέρει απλόχερα, τον αξιαγάπητο και ευγενικό μα με απόλυτα διαστρευλωμένη αίσθηση του χρόνου Θανάση, τον Γκούνη που ακόμα σήμερα πειραματίζεται με την εφευρετική φύση του ανθρώπου των σπηλαίων, την Τζουλιάνα που μας δείχνει κάθε μέρα ότι η ζωή είναι ελαφριά σαν τα βήματα της σάλσα, τον Αναντίμ που ζωγραφίζει τη ζωή με ποιήματα χρωματισμένα με συναισθήματα και μετά τα μεταδίδει στους γαλαξίες, τον Μιτσάκο που είναι εκεί κάθε φορά που τον ζητάς, τον Γραμματικάκη που η γενναιοδωρία του είναι αντιστρόφως ανάλογη του βάρους του, την μητέρα όλων μας Μαλάμω, τον απόλυτα ευγενή μα αριστοκρατικά απαθή Ντίνο, τον παρανοϊκό πλην πάντα με στυλ Μίλτο, την βαθειά όμορφη Νατάσα που η παρέα της είναι σαν το νερό, τον ευγενικό αλλά “καλά πόσο στον π* ευγενικός μπορεί να είναι;” Μελέτη, τους φίλους μου τη Μερόπη και τον Τσμπάου που πορεύονται μαζί στη ζωή και στην Τέχνη, τον Κυριάκο που σε βυθίζει στον κόσμου του γραμμένο με όμορφες γραμμές απο κώδικα C που για το καλό όλων μας θα στραφεί κάποτε στον Άνθρωπο, την Μυρτιά που αφήνει το πνεύμα σου να ταξιδέψει σε μέρη ομορφιάς από άλλον κόσμο, τον λαϊκό Μπιλακάκο που ανεβαίνει στη σκηνή και με ένα νεύμα του χεριού του σε κάνει να πετάς, την αγαπημένη μου Δανάη που περπατάμε μαζί σε παράλληλα μονοπάτια, που πάντα αλλάζει και πάντα διστάζει, την γλυκιά μου δυναμική Κάκια που μας βάζει όλους στη σειρά και θα έλεγα έχει περάσει πιο οργανωτικές φάσεις (για το οποίο μερίδιο ευθύνης ομολογουμένως φέρουμε κι εμείς), την ειλικρινή και πάντα ευδιάθετη μα πειραματιστή των σχέσεων Νίκη που τραγουδάει και οι υπόλοιποι ξεκλέβουμε νότες για να γεμίσουμε ξανά, την πρόσχαρη (εντάξει Φωτεινή, υπερπρόσχαρη) Ελένη που ψάχνει να βρει τον δρόμο της, τον Τάχο που χαμογελά σε όλους και αφήνει τον παράδεισο να λάμπει πάνω μας, τον αγαπημένο μου Στέργιο που διαρκώς φλερτάρει με την τρέλα και την νηφαλιότητα δείχνοντας μας όλους πως να ζούμε ελεύθεροι, τον Δικηγόρο που αγαπάει και ενώνει τους πάντες κάτω απο τα φτερά του, τον καθηγητή μου που είδε σε εμένα το βέλος και έγινε το τόξο δίνοντας μου κατεύθυνση και στήριξη.

Αυτή η διπλωματική είναι τελικά το αποτέλεσμα 24 χρόνων αγάπης που μου έδωσε η οικογένειά μου, η μητέρα μου που βλέπει τον εαυτό της και όλο το Σύμπαν μέσα μου, ο πατέρας μου που σιωπηλά και διακριτικά φροντίζει για την συνοχή της οικογένειας, και η αδερφή μου που μόνο με έκρηξη ηφαιστείου μπορεί να παρομοιαστεί που πάντα με αφήνει πίσω έναν καλύτερο άνθρωπο.

Introduction

Why should Go in hardware interest me? This is the question, which I enquire here. First, what is, that interests me? Certainly, one finds interest in what has to do with him.

Why does Go interest me? I was never really a character of the competitive kind. Playing a game, mental or physical was totally uninteresting to me as the ultimate motive of winning, which is the cornerstone of most games, was practically non-existent in me. Dismantling toys or painting wherever I could was all I desired. Thus, as this artist of whichever art, I entered the Computer Science department of the Technical University of Crete, in the same fashion as probably all the children of the world; listening to parents, teachers, politicians, priests and all the mighty and knowledgeable people inhabiting this by all means crazy planet.

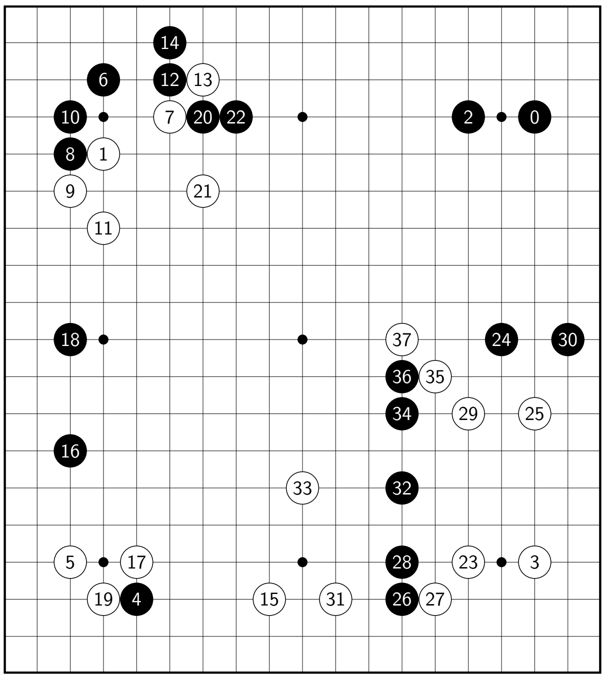
When my pre-graduate studies came to an end and my Thesis was all that was left, destiny with its vast irony had kept a game for me as my Thesis subject. Thus, I approached Go with a moderate interest initially. If one doesn't like games during his childhood, how could he ever like them long after his puberty? And at this point I started playing Go. Attacking fiercely lead me to a loss, defending as best as I could definitely lead me to a loss, and trying to remember good moves by acknowledged Masters of the game always lead me to an utter loss. As much as a cliche it clings nowadays that the East thinks opposite than the West, Go was in fact an apt proof of this.

The most successful game I played during the first month of my Thesis came when despaired, I didn't strive to win, or lose. It was a boring match by all means between me and the computer. Quiet soon I realized that the quality of mind that I had named boredom was in fact only a response to the opponent's moves deprived of any desires whatsoever and with the single goal of neither winning nor losing. It was a game of balance with my ambitions in fact. Playing Go many more times to test my new observation, this quality seemed to lead to quiet consistent, non-negative scores. What I was doing, was acknowledging the opponent's right to move freely on the board and occupy space as much as myself. I still play Go because of this.

Backgammon is a "man vs. fate" contest, with chance playing a strong role in determining the outcome. Chess, with rows of soldiers marching forward to capture each other, embodies the conflict of "man vs. man". Because the handicap system tells Go players where they stand relative to other players, an honestly ranked player can expect to lose about half of their games; therefore, Go can be seen as embodying the quest for self-improvement—"man vs. self".

Why does computer hardware interest me? First of all hardware gives me the sheer joy of creating something apt that I see working afterwards. Inspiring "life" to materials which are considered lifeless (*even though they are no more lifeless than we are*) is a feeling that everyone who has been dismantling and re-soldering his toys as a kid is aware of. Even if programming hardware nowadays is closer than ever before to software programing, where no deep knowledge of the complexities of the world of electrons is needed, still hardware has the aforementioned taste. Furthermore, the inherent parallelism of hardware, provides a refreshing new view for every software programmer as myself.

Thus, examining the game of Go under the prism of, and taking advantage of hardware's own characteristics appeared as a thoroughly interesting subject to me in the end. Constructing a processor, with FPGA technology which was available to me at that time, capable of playing Go competently, was something that oversized the time constraints of a Thesis subject. But, what I ended up with, is an elegant solution to some of the fundamental problems which every Go programmer might have to face. The solutions proposed serve perhaps two roles to you, reader. They are firstly some (*I believe*) elegant examples of how computer science problems in the field of board games can be mapped to hardware designs, and secondly, they provide a new view to the much researched problem of computer Go for a software programmer during these days of multi-core CPUs and the availability of true parallelism in software.



Honinbo Shusaku
本因坊秀策

RELEVANT RESEARCH

A Brief Review of Computer Games

2.1 Classification of Games

Man has been inventing and playing games through centuries as a means of understanding the way the world and himself in it functions. Creating a micro-cosmos, based on a set of predefined rules enables one to observe the development of a miniature of the world he lives in. Success in adapting to these rules, whether it leads to a victory towards another man, luck, or the self, gives the illusion of a god-like power and the psychological security arising from a better understanding of nature's workings.

Game theory, a field of mathematics which came into being after John Von Neumann and Oskar Morgenstern wrote their book "Theory of Games and Economic Behavior" on 1944, has extensively researched the nature of games, and has provided a solid theoretical basis on which one can classify a game. The classification itself has, as a side-effect, enabled computer programmers to develop different families of algorithms, which can be applied to each game category. Thus, an identification of the nature of a game according to what game theory suggests, enables the researcher to prune his search space.

Games are initially classified into one-player and multi-player ones. One-player games are considered those which involve decisions to problems only by a single individual player with, or without, the presence of randomly acting players which make "moves by nature" and serve the role of a dice.

Multiple-player games are classified according to the cooperation-forming criterion. A game is *cooperative* if the players of the game can form groups of common interests, bound under a set of mutual commitments. One might say that cooperative games permit the communication among players of the same group while *non-cooperative* ones don't.

Games are further classified as *symmetric* or *asymmetric* according to a role criterion. In a symmetric game the identities of players can change without altering the payoff to the strategy adopted by each player. A symmetric game's outcome depends on the strategy employed and not by who employs that strategy. The notorious "chicken" game, which often constitutes the climax of 80s teen movies and which involves two cars running towards each other with the aim of proving the defiance towards death of one of the two "players", is an example of a symmetric game. The dictator game on the other hand is non-symmetric (as the dictatorship itself).

The flow of a game's resources classify games into *zero-sum* and *non-zero-sum.* In zero sum games, the cumulative benefit of all players is always constant, or else, one player benefits only at the equal expense of others. Poker, Go and chess are zero-sum games, exactly as most of the classical board games are.

The ability of players to move simultaneously classifies games as *simultaneous* or *sequential.* One quiet non-obvious case of simultaneous games are those, in which actions of each player is unknown to the others. This effectively makes the game simultaneous. At the same time, sequential games imply complete or only some knowledge about other players' actions.

The latter further classifies sequential games in *perfect-information* and *imperfect information* games. A game is one of perfect information if all players know the actions made in the past by the rest of the players. Go and chess are again perfect information sequential games.

Finally, time-wise, games can be either *infinitely long* or *finite.* Most people (except for mathematicians) are interested in games which end in a finite number of moves and which contain a finite number of players and outcomes.

2 . 2 The Game of Go

2 . 2 . 1 *History*

Legends trace the origin of the game to Chinese emperor Yao (2337 - 2258 BC), who had his counselor Shun design it for his son, Danzhu to teach him discipline, concentration and balance. Other theories suggest that the game was derived from Chinese tribal warlords and generals, who used pieces of stone to map out attacking positions, or that Go equipment was originally a fortune-telling device.

The earliest written reference of the game is generally recognized as the historical annal Zuo Zhuan (c. 4th century BC), referring to a historical event of 584 BC. It is also mentioned in Book XVII of the Analects of Confucius (c. 3rd century BC). In all of these works, the game is referred to as *yi* 弈. Today, in China, it is known as *weiqi* (圍棋). Go was originally played on a 17 x 17 line grid but a 19 x 19 grid be-came standard by the Tang Dynasty.

In China, Go was perceived as the popular game of the aristocracy, while Xiangqi (Chinese chess) was the game of the masses. Go was considered one of the four cultivated arts of the Chinese scholar gentleman, along with calligraphy, painting and playing the musical instrument guqin.

Go was introduced to both Japan and Korea - where it is called *baduk* - somewhere between the 5th and 7th centuries AD, and was popular among the higher classes. In Korea, the game evolved into the variant called Sunjang baduk by the 16th century. Sunjang baduk became the main variant played in Korea until the end of the 19th century.

In Japan - where it is called Go (碁) or *igo* (囲碁) the game became popular at the Japanese imperial court in the 8th century, and among the general public by the 13th century. In 1603 Tokugawa Ieyasu reestablished Japan's unified national government. In the same year, he assigned the then-best player in Japan, a Buddhist monk named Nikkai to the post of Godokoro (Minister of Go). Nikkai took the name Honinbo Sansa and founded the Honinbo Go school. Several competing schools were founded soon after. These officially recognized and subsidized Go schools greatly developed the level of play and introduced the dan/kyu style system of ranking players. Players from the four schools (Honinbo, Yasui, Inoue, Hayashi) competed in the annual castle games, played in the presence of the shogun.

Despite its widespread popularity in East Asia, Go has been slow to spread to the rest of the world, unlike other games of ancient Asian origin, such as chess. Go did not start to become popular in the West until the end of the 19th century, when german scientist Oskar Kosrchelt wrote a treatise on the game. By the early 20th century, Go had spread throughout the German and Austro-Hungarian empires. In 1905, Edward Lasker learned the game while in Berlin. When he moved to New York, Lasker founded the New York Go Club. Lasker's book Go and Go-moku healped spread the game throughout US and in 1935 the American Go Association was formed. Two years later the German Go Association was founded. As of 2008, the International Go Federation has a total 71 member countries.

2.2.2 *Rules*

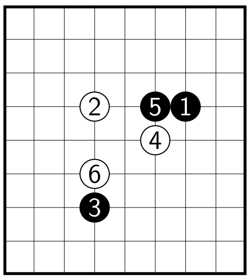
According to Game Theory, Go is a multiple-player, non-cooperative, symmetric, zero-sum, sequential, perfect-information, discrete and finite game. Go is a contest between two people to secure territory. The territory consists of 361 points which are formed by the intersections of 19 vertical and horizontal lines drawn on a wooden board. Players use lens-shaped discs, called stones to mark off their territory. One player plays black, the other white, in alternating turns. The board which is empty in the beginning, gradually fills as players place their stones. Contrary to most western games, motion in Go takes the form of adding to what is already in place rather than moving the position of the pieces. Once put on the board, a Go stone is stationary unless captured. The player controlling the largest total area at the end of the game is the victor.

Quoting Kaoru Iwamoto:

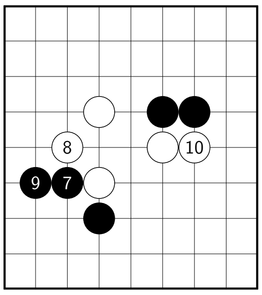
" There is no better way to learn how Go is played than to go through a short demonstration. For the sake of brevity, the game in this chapter takes place on a 9x9 board instead of the usual 19x19, but the rules of Go are the same no matter what the size of the board. You can play through this game on a 9x9 Go board, which can be made by masking off part of an ordinary board, or you can follow it by just looking at the diagrams. Since the stones do not move about, go diagrams are easy to read. In ***Dia. 1,*** *1 is the first stone played, 2 the second, and so on.*

Besides getting a general idea of the game in this chapter you will learn exactly what territory is, how it is formed, and how it is counted. You will also see how stones are captured, but we will be going into the that matter in more detail in Chapter 2.

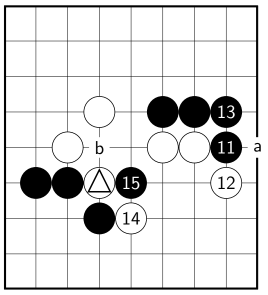
Dia 1 As always in Go, black plays first. You are free to put your stones wherever you like, but notice that the players here do not play right on the edges of the board and will not start to do so until later in the game. This is a good policy.



Dia 2 *With 7 and 9 Black begins to stake out some territory in the lower left corner.*
Dia 3 *The area in the upper right corner starts to fall into Black's hands, too. It is bounded above and to the right by the edges of the board, and is walled off below by a solid row of black stones. It is still open to the left, and there is a small gap at the point marked 'a', but Black will deal with these matters in due time.*



Dia. 2

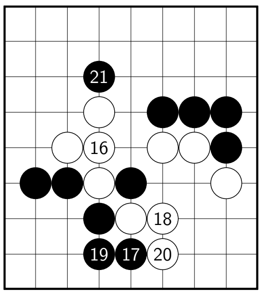


Dia. 3

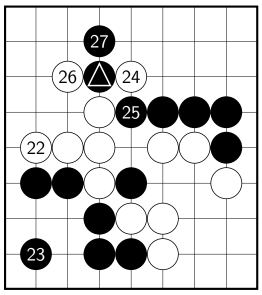
When Black plays 15 we are presented with an example of threatened capture, or atari. White △ has become surrounded on three sides by black stones, and if Black is permitted to play on the fourth side at 'b', White △ will be removed from the board.

Dia 4 *White saves his threatened stone by playing 16. Black 17 is another atari, this time against △, and White 18 is another saving move. The end of this diagram sees Black in control of the lower left corner, White in control of the lower right and Black ambitiously stepping out to 21 on the upper side.*

Dia 5 *White 22 threatens Black's while structure in the lower left, although this is something which is not obvious except to an experienced player, and Black has to strengthen his position with 23. Now White counterattacks with 24 and 26, and the next diagram will witness the capture of black ▲ and 27.*



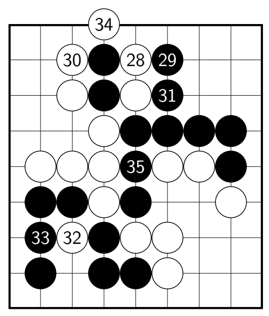
Dia. 4



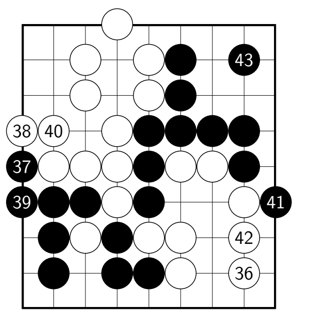
Dia. 5

Dia 6 *White captures the two black stones by playing 28,30 and 34, removing them from the board and putting them in the upturned lid of his bowl when he plays 34. The two points where they rested have now become White's territory, along with a good many more in the upper left corner, while Black has walled off the upper right corner. White 32, which was played just to test Black's defenses, is in atari, being surrounded on the three sides by Black 33, ▲, and ▲, and cannot hope for salvation. Black next plays 35, cutting off the white stones in the lower right corner and making them vulnerable.*

Dia 7 *White defends his position with 36, and now both players begin to complete the walls around their territories by playing at the edges.*

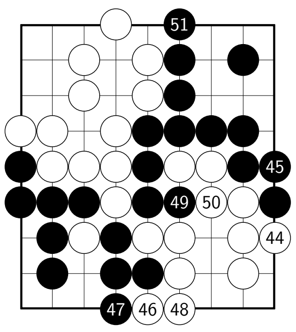


Dia. 6

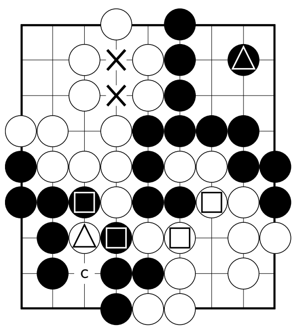


Dia. 7

Dia 8 *At the end of this diagram all the boundaries are completed. Both players recognize that there is nothing more they can do to enlarge their own territories or to reduce their opponent's, so the game is over. Let's count and see who has won.*



Dia. 8



Dia. 9

Counting

Dia 9 *A player's territory consists of the vacant points he has surrounded. In the upper right corner Black has surrounded nine points - see if you agree. The point* ▲ *does not count, even though it is in the middle of Black's territory, because it is occupied by a black stone.*

In the lower right corner White has six points of territory. There is a kind of diagonal break between the two stones marked □*, but that is all right. The territory is surrounded because there is no route leading out of it along the lines on the board that does not run into a white stone.*

What about the lower left corner? This is Black's territory, but there is the white stone △ *left stranded within it. As was said before, there is no saving that stone, and according to the rules of Go Black need not actually capture it by playing inside his own territory at 'c'. When the game is over he just takes it from the board as his prisoner, leaving himself seven vacant points here. Again there is a diagonal break between the two stones marked* ■*, and again that is all right.*

The rules of Go also state that one point is to be subtracted from a player's territory for every stone lost, so we have to figure another point for White △*. Instead of subtracting one point from White's score for this stone, it is easier to add a point to Black's territory, so we shall follow that course and count eight points for Black in the lower left corner: six vacant, one that becomes vacant when white* △ *is removed, and one more 'prisoner point'. That is, Black gets two points for* △*.*

In the upper left corner White has surrounded ten vacant points, but on two of them, the two marked X *, black stones have been captured. As before, we shall add the two points for them to White's territory here instead of subtracting them from Black's, making White's total in this corner twelve. The score is thus:*

Black		*White*	
Upper right	*9 points*	*Lower right*	*6 points*
Lower left	*8 points*	*Upper left*	*12 points*
	17 points		*18 points*

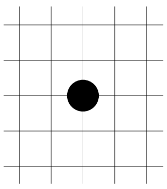
White has won by one point. What about the vacant point in the center of the upper edge? This is a neutral point, a sort of no-man's-land between the black and white lines, and as such it counts for neither player. The method of counting we have just described is the natural one and is used by good players to count the territories during the course of an actual game. However, it gives too much opportunity to human error to be entirely trustworthy, so when a game is played out to the end, the players use the following foolproof procedure..."

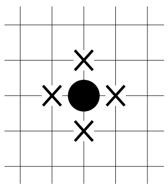
Kaoru Iwamoto continues with a proposition for a safer territory counting method, which is out of this subchapter's main subject, which is to provide a basic introduction to the basics of a Go game to the reader, so that he can follow the rest of the text.

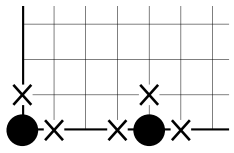
There has been a very deep research by Go professional players of the present and the past on Go tactics and strategy, as mainly due to its large board size, Go is an extremely complex game as far as evaluating the value of a move is concerned. This becomes obvious when one sees the place which Go owns in Japan, where it is considered more than a game. Go is taken very seriously, as demonstrated by the fact that professional Go players, who acquire their distinction after rigorous matches, earn their living teaching Go or evaluating Go matches.

The subject of a good game of Go though, is one of another Thesis, which must follow this one and will deal with the question "*what is a good move to play given a board configuration?*". The most important question about Go for this Thesis is this: "*How can stones be efficiently placed on the board and how can we know fast which stones are captured?*".

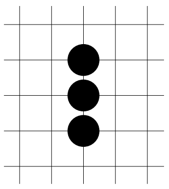
In order to be able to follow chapter 3, three more important Go concepts are needed to you, reader: *liberties*, *groups* of stones, *cuts* and *suicide* moves. As Kaoru Iwamoto says: "*A single stone sitting in the middle of the board, as in* **Dia 10**, *may be compared to a man standing at an intersection in a big city.*". A single stone placed in the middle of the Go board has initially 4 liberties as shown in **Dia 11** . If all four liberties are occupied by opponent stones, then this stone is considered captured and is removed from the board and counted, as explained before, as one more point in the opponent's score. Correspondingly, **Dia 12** presents the liberties of stones sitting at corners or on the edges of the board, which are 2 and 3 liberties respectively. Realizing this, one can now understand Kaoru Iwamoto's perhaps urgent advice in his introduction quoted above: "*notice that the players here do not play right on the edges of the board*"; that is, edge stones are captured easier.

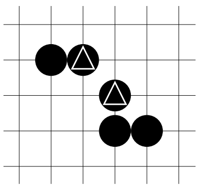

Dia. 10


Dia. 11

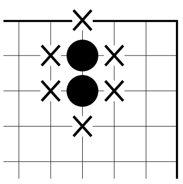

Dia. 12

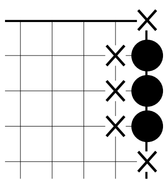
Groups of stones of the same color are considered as *solidly connected* when they are joined together by direct connections along the black lines of the board. Such a group is shown in **Dia 13**. The unexperienced reader might mistake the black stones of **Dia 14** as a solid black group. It is in fact though only, two, disparate groups which are separated between the ▲ marked stones. This very point of separation is called a *cut*.


Dia. 13


Dia. 14

A solidly connected group of stones functions as one. Thus, in order to be captured, all its perimetrical liberties have to be occupied by opponent stones. Counting these liberties is what is essential in order to decide whether a group is captured or not. **Dia 15** and **Dia 16** show two groups with 6 and 5 liberties, respectively.


Dia. 15


Dia. 16

2.2.3 *Computational complexity*

It has been quoted numerous times, and probably in all Go-related publications that Go is indeed a complex computer problem to solve, or at least most of the traditional techniques used in solving similar problems are quiet inapplicable in the Go field. This creates a misunderstanding.

What we have to realize about Go is that we'd really like to use traditional AI searching techniques like MINMAX with AB-pruning, but we can't, mainly because there is no admissible evaluation function. Most reports may make people believe that there should be somewhere some new, radical, totally different method of solving Go. This is not true. We just search for a decent computer Go player until someone comes to unite things under some good evaluation function. Computer chess research had begun with a similar state of mind around the 70s. Research was back then all about mimicking the human way of thinking and translating it to a computer program. Human knowledge is incomplete though, and knowledge on how man thinks is twice that incomplete. Only after man realized that computers should better do what they best can do, that is calculations, could computer chess take off from a level C player where it was stuck with the previous state of mind.

There are some important facts regarding Go's complexity, computation-wise, which every researcher observes. These initial confrontations occur when one tries to use MINMAX search along with some pruning, to answer about what is the best move for a board position.

The first obstacle is the traditional Go *board's size*. A 19x19 board, leading to 361 possible moves in the beginning and 200 possible moves on average during the game for about 200 rounds, results in a search space, larger than any technology nowadays could cope with, as far as memory consumption and computation speed are concerned. That means that either custom machines have to be built up to tailor the Go problem, or the pruning algorithms have to become more sophisticated. Distributing the computations needed to search deeper in the moves tree among hundreds of custom FPGAs is a solution that fits best the first scenario and studying the way professional players think while pruning less promising parts of the board fits the second one. My own opinion on this is that the first strategy is more attractive, simply because this depends solely on resources , that is money, while the latter on easily fallible methods. The following table shows a comparison of the combinatorial complexity among the most known games.

Game	Combinatorial complexity
Go	10^{400}
Chess	10^{123}
Othello	10^{58}
Checkers	10^{32}

The above estimation is done with the B^L formula where B is the branching factor and L equals the average game length.

Furthermore, a move played during the beginning of the game, the *opening* stage as it is called, might prove useful only after 100 moves. That means that evaluating a move with a static analysis of the board, like counting the number of settled areas with strong groups, or evaluating the quality of the patterns created, can be totally misleading for the global quality of a player's stones. Pattern recognition is a field where the human mind excels, and in a highly visual game as Go, where stones remain on board for a long time unless captured, humans can be very efficient. The inability to map this knowledge to a concrete *evaluation function* is a major obstacle in using local searching techniques in Go.

2.2.4 *History of Computer Go*

For a problem, which contains subproblems, for which no definite optimal solution has been proposed, being able to quickly prototype new ideas is a major consideration. Therefore, the short history of computer Go comprises of ideas borrowed from other computer fields and mainly the AI field, and programs implementing those ideas into complete Go playing software.

Bruno Bouzy in his paper *"Computer Go : an AI oriented Survey"* explains the history of computer Go:

"It seems that the first Go program was written by D. Lefkovitz [Lefkovitz 60]. The first scientific paper about Computer Go was published in 1963 [Remus 63], and it considered the possibility of applying machine learning to the game of Go. The first Go program to beat a human player (an absolute beginner at that time) was the program created by Zobrist [Zobrist 69,70b] . It was mainly based on the computation of a potential function that approximated the influence of stones. Zobrist made another major contribution to computer games by devising a general, and efficient, method for hashing a position. It consists of associating a random hash code with each possible move in a game, the hash of a position being the XOR of all the moves

made to reach the position [Zobrist 70a]. The second thesis on Computer Go is Ryder's [Ryder 71].

The first Go programs were exclusively based on an influence function: a stone radiates influence on the surrounding intersections (the black stones radiate by using the opposite values of the white stones), and the radiation decreases with the distance. These functions are still used in most Go programs. For example, in Go Intellect [Chen 89, 90, 92], the influence is proportional to 1/2distance, whereas it is proportional to 1/2 distance, in Many Faces of Go [Fotland 86, 93].

Since the early studies in this field, people have worked on sub-problems of the game of Go, either small boards [Thorpe and Walden 64,72], or localized problems like the life and death of groups [Benson 76].

The first Go program to play better than an absolute beginner was a program designed by Bruce Wilcox. It illustrates the subsequent generation of Go programs that used abstract representations of the board, and reasoned about groups. He developed the theory of sector lines, dividing the board into zones, so as to reason about these zones [Wilcox 78,79,84, Reitman and Wilcox 79]. The use of abstractions was also studied by Friedenbach [Friedenbach 80].

The next breakthrough was the intensive use of patterns to recognize typical situations and to suggest moves. Goliath exemplifies this approach [Boon 90]. State-of-the-art programs use all these techniques, and rely on many rapid tactical searches, as well as on slower searches on groups, and eventually on global searches. They use both patterns and abstract data structures.

Current studies focus on combinatorial game theory [Mueller 95], [Kao 97], learning[Cazenave 96c], [Enzenberger 96], abstraction, and planification [Hu 95], [Ricaud 95, 97], and cognitive modeling [Bouzy 95a].

The eighties, saw Computer Go become a field of research, with international competitions between programs. They also saw the first issue of a journal devoted to Computer Go, as well as the release of the first versions of commercial programs. In the nineties, many programs were developed, and competitions between programs flourished, being regularly attended by up to 40 participants of all nationalities [Fotland and Yoshikawa 97]. An analysis of the current state of the Computer Go community has been published by Martin Mueller [Mueller 98]."

State of the Art in Computer Go

3.1 Board Representation

Almost all research in the Go field has been and is done naturally in software and a high level one. How much could some idiomatic high level code or algorithm or data structure running on a general-purpose processor help in the design of custom hardware? Practically, very little. No meaningful, complete and elegant solution has managed till today to unite software and hardware under a common language that expresses high level concepts and is at the same time translated automatically into an efficient hardware design. On the other hand, the more a software solution strives for performance, the closer the programmer has to come to the underlying hardware.

Computers playing Go have been taking part in organized matches against other computers and against humans for the last 20 years. The time constraints of a real match instead of an "in vitro" benchmark of a program has lead to the optimization of certain, key parts that almost every Go implementation has to consider. We shall examine together these parts under the prism of GNUGo and MoGo, two different computer Go programs whose designs, at least initially, represented the two most popular but distinct approaches in tackling the game of Go.

3.1.1 *The board itself*

The question of how to represent the board itself comes as a natural first. A straightforward approach for any software programmer would indicate to implement this as a two-dimensional array of values which would represent an empty position, a white, a black stone or a border. These 4 distinct states would require 2 bits of information. Moreover, as the indexing of a two-dimensional array poses the extra overhead of a multiplication and addition to retrieve the exact memory position of the wanted board position in a computer's one dimensional memory, one could argue that an one dimensional array for the board would suffice, and as a gift we spare some cpu cycles *(the complexity would be in either case constant as an array structure guarantees O(1) time complexity, so we only decrease the hidden constant factor).*

The representation of the board is tightly dependent on the context of its use; that is the move generation algorithm employed to guess a good move. If a board has to be continuously copied and altered as in a MINMAX searching strategy, then economy of space is the designer's top priority. If on the other hand, a capable static evaluation of the board is used to guess the next move, then a board, capable of retaining as much information on the groups and state of stones on it in the form of various caches, will be able to feed the algorithm with the needed information with great efficiency.

Without going into much detail, it suffices for now to say that GNUGo employs a combination of local searching techniques and complex board analysis for the move evaluation step. Because of this, the board has to be updated and copied continuously during the evaluation stage. At the same time, the same board state can occur fre-quently in the different sub-branches of the search tree, leading to a duplication of information and a waste of computation cycles to reevaluate this state's position. Therefore, a hashing scheme has been used by GNUGo's designers for the repre-sentation of the board state, based on a popular technique from the world of chess programming called Zobrist hashing.

The idea behind Zobrist hashing is that if we could represent a board state by a single value, different than any other value representing some other board state, and thus unique, then we could use this value as the key of a hash table, which would store all the results of the board evaluation for that board state. That means, whenever this board state is met again in a search tree, we can fetch all its evaluation information in constant time from the hash table.

The characteristic of Zobrist keys, which makes them attractive for board games like Go and chess is that due to the way they are implemented, you can manage the keys incrementally. That means, whenever a new move is played, then the new hash key which represents this board state, can be directly computed from the previous key, without examining all the stones on the board again. For implementation details on Zobrist hashing see [REF: ZOBRIST].

On the other hand, MoGo follows the path of Monte-Carlo simulations for move evaluation (*see below for a deeper analysis*). It suffices to say, that for such a design, a Go board is needed, which is able to perform its basic operations like, putting a stone on a position, or capturing groups of stones or emitting an error for an occupied position, at the maximum possible speed. In this approach the moves are generated in a random fashion, thus there is no delay from the move evaluation stage and the board is constantly fed with new moves. The speed with which complete random matches are played, or in other words, the number of completed random matches played in a defined slice of time, are directly correlated to the efficiency of this approach. Note, that this does not mean by any case that the GNUGo approach could not take advantage of a fast board; it means only that the cost of the basic board operations is negligible compared to the time spent on the complex board evaluation functions which GNUGo employs.

An array provides still the fastest way to change the state of board positions. And in the pure Monte-Carlo approach, no other information has to be held in the board other than the board itself. Thus, the only factor which could determine the performance of a board is the stone capturing algorithm, which is coincidentally this thesis's main contribution.

3.1.2 *The stone*

One could argue that the board without the stones needn't possibly hold any other information other than its size and perhaps the score or whose turn it is. Usually though, all the information on strings of stones, safe or weak groups etc. is held in the board structure. From a software standpoint this is convenient as the board plays the role of a centralized object holding most of the game's state. Moreover, since in Go there is no possibility of having two boards per game (*at least as far as I know there is no such variation*) this information is also not replicated in more than one instance.

The stone itself thus is represented by the board position itself, whether it is an array or a list and is identified by an array or a list index. There is no need to hold any other, inner, information in each stone so in software one doesn't find a distinct class or any other mechanism to store state for the stone. The value of a board position (*white, black, empty, border*) is the same as the stone or the lack of it.

3.1.3 *The move & the capturing*

The very playing of a move, and not the generation of it, is implemented according to the data structure chosen to implement the board positions. Thus, for an array it is a simple memory assignment, or for a list a traversal to find the wanted position which predates the assignment, and the assignment itself.

The move is what affects a Go board. Not dice, not luck, not random players. Everything starts with a move and all the changes of the board are emitted from where the last stone was played. Those changes are reduction of liberties, merges of groups, ko state, and captures of stones.

A Go board program must support these fundamental changes and modify its inner structures to reflect the new board state. Captured groups may have to be removed after a move has been played on the board. From the programmer's standpoint, deciding whether a group is alive or dead is the first major design obstacle. A group is captured if all its perimeter is occupied by stones of the opponent, and examining the perimeter of one stone is an operation that is inherently recursive as after examining our perimeter and finding no empty positions, we have to examine our perimeter's perimeter. Recursion, that is the case where a function calls itself, is a mostly useful idea which was introduced with the LISP language and till recently was a privilege of the software world. Considerable work in this field has been done by [REF]. Still, I think that for most of the readers, the following iterative solution to the capturing problem will be more easy to grasp, while at the same time, equally efficient with the recursive one.

Starting from the played move we check its four neighbours. If they are the same colour as we are then we insert them in a queue. Then, for every stone in the list we examine its own neighbours and insert them accordingly if 1> they are not already present in the list and 2> are the same color as we are. If we find an empty neighbour position then the list is cleared, we stop searching and know that there is at least one liberty and thus our group is safe. If on the other hand no empty neighbours are found and the list is emptied, it means that we are surrounded by borders and/or opponent stones and therefore captured. In this case all the stones in the list are removed from the board or pushed in another temporary list as captured stones.

Still, if we cache for speed each group's liberties, after removing the captured stones we have to recount the liberties of the group which caused the capture, as they will have increased. The previous algorithm does not demand such a cached knowledge and counts each group's liberties from the beginning each time a new move is played. In reality though, most programmers retain this information and for this reason I think I should include this information here on how to redistribute these liberties for the sake of completeness. Most programs remove the stones of the group or groups, which caused the capture, from the board and replay them, counting their new liberties incrementally as each stone is placed and merged with the rest of the group. Such a simple solution compared to another which would imply much more bookkeeping, is probably the result of profiling the two approaches and finding the best complexity versus efficiency ratio. I have not tested these two approaches in software, so this is something that you have to research yourself.

3.2 Move Generation

Even if this thesis is not about move generation and considers the move generation & evaluation module already present, one has to go a little deeper in move generation approaches in the field of Go to understand the constraints each approach poses on the board implementation strategy.

3.2.1 *Classical approach: Divide & Conquer*

Programs based on classical AI techniques to solve Go, such as the GNUGo engine, conceptually follow the same game plan. They use a break-down of the whole game and each game round apply local searches and global tactics. Their playing style is based on goal-oriented sub-games, that means, the best move in this situation is probably the best move globally. Clearly this is not always the case in Go but this is adequate for a competent computer Go player.

In more detail, playing goal oriented sub-games involves the evaluation of stone strings, their connections, their dividers, their potential, eyes and of course life and death analysis. If we wanted to further divide the programs which use the divide and conquer approach (*no pun intended*) we would find two different strategies. The first group of programs bases its analysis on a zero depth global move evaluation. This evaluation value is the weighted sum of various local tree searches on parts of the board and of domain dependent knowledge. In this group we find programs like GNUGo, Explorer and Handtalk. The second group applies moreover a shallow global tree search using some conceptual evaluation function in addition to the local searches. In this group we find Many Faces of Go, Go Intellect, Indigo2002.

The advantages of using a divide & conquer approach is today mostly a matter of computer resources. This approach is feasible computationally on current computers, while at the same time being locally precise. Being locally precise means that a beginner or medium level player can indeed improve by playing against such a computer player, learning to shape his stones well and to deepen his reading ability in life and death problems.

On the other hand, the break-down approach as you can guess is not valid in the game of Go. All professional Go players more often than not offer Go players including myself, moves to think about for weeks, proving that the bad move can be in a future context the best position. After playing many games of Go, you will start seeing that the sub-games themselves are not independent; which is taken as true in a divide & conquer approach. Moreover, the encoding of domain dependent knowledge is a big undertaking on its own; adding one new move pattern in a database of moves played by professional players, can adversely affect the functionality of the complete database. Even if after many years the quality of such databases has increased, maintaining and extending them is a tedious and error-prone procedure. From a hardware implementation point of view, the complex data structures used in this approach to preserve and update the complex relationships between groups of stones are a major headache.

3.2.2 *New approach: Monte Carlo Tree Search*

On 1990 Abramson proposed a model of terminal node evaluation based on simulations. He applied his idea to 6x6 Othello. On 1993, Brugmann used simulated annealing with a "all-moves-as-first" heuristic for the evaluation of Go moves. These new ideas were further advanced by the work of Bouzy & Helmstetter on 2003. Bouzy later experimented with a combination of Min-max and monte carlo on 2004 and on 2005 extended his design his Go knowledge bases. Another breakthrough came on 2006 with Kocsis & Szepesvari with their conceptual connection of the monte carlo strategy in the evaluation of Go moves and the multi-armed bandit classic AI problem. UCT, a very efficient algorithm designed to tackle this problem, increased considerably the strength of Go programs of this new era to the level of a lower dan professional Go player, a case expected by researchers only at least twenty years later.

The basic Monte Carlo strategy as applied to move evaluation in computer games, follows this plan; N random games are launched. When all games come to an end the score is computed for each game, possibly signifying a win or a loss with a positive or negative score. At this point the score is easy to compute as what exists on the board is only eyes and stones. These steps are present in all Monte Carlo programs. What affects though the strategy's true efficiency is the algorithm used to choose the most promising moves among those with positive scores. Suppose we have launched 100 games for each one of 10 possible next moves. A depth-one greedy approach chooses each time the move with the best mean of scores. Billings and Sheppard came in 2002 with an improved technique which employed progressive pruning of bad moves. The first move choice was done randomly and afterwards each move inferior to the best move was pruned. This improved MC's efficiency.

Realizing that the choice of the most promising move during MC simulations is in fact an exploration versus exploitation problem, researchers turned to algorithms tackling this problem. The tradeoff is between the amount of exploration, which corresponds to the number of simulations per examined move, and exploitation, which corresponds to the pruning of “bad” moves and the reward of “good” moves which have to be further explored. UCT, an algorithm which favours exploration is still today the basic building block of the most powerful Go programs after Bouzy presented his most important paper in 2006. New heuristics are presented continuously which improve UCT’s efficiency but the main algorithm remains the same which proves how fit the correlation was between MC in Go move evaluation and the multi armed bandit problem.

The next step was the application of MC to a tree search, that is to combine the new move evaluation strategy with the classical tree search approach used throughout the AI field. Launching N random games and choosing in the end the best move to be played is like doing a tree search with depth 0. During a tree search the same amount of evaluation is done both for ones moves and for the opponents moves replying to ours. As long as the MC strategy leads to consistent results then it can indeed be used as an evaluation function to prune away branches of the search tree which contain unpromising moves and all references to MC efficiency above, refer to this very thing; how consistently can an MC simulation find the good move among bad ones.

A Go Board in Hardware

4.1 Performance through concurrency

The following chapter begins with the idea of a parallel Go board. In the end of this chapter we will end up with a simulator of a hardware Go board capable of supporting an extremely fast execution of the basic board operations, which is a fundamental necessity for every Go evaluation scheme based on Monte Carlo simulations. As far as my choice of MC in favor of complex move evaluation functions as in GNUGo is concerned, such a choice does not arise from the new breed of powerful Go programs based on MC. According to my view, the computer player doesn't necessarily have to have too much intuition to play strong. Just good enough so that he can prune the search tree safely. This is exactly what happened with Deep Blue; it's not that it was such a great chess player, as an adequate chess player with extraordinary reading ability.

4.1.1 *A word on concurrency*

Hardware is inherently concurrent. Simply put, many things can happen in parallel. And apart from a reality, concurrency is also one means of improving performance. If a problem can be split into smaller independent subproblems, then solving them in parallel improves the performance by a factor equal to the number of the subproblems. In software, performance gains came in the past only from tuning an algorithm to cooperate well with the underlying hardware (*caches, branches, macroinstructions*), in hardware from an efficient coordination of independent entities. From a high-performance viewpoint, the crucial factor in software design is to know exactly how the hardware will execute your program, and in hardware design is to design for peak usage of the hardware fabric, even if under normal conditions most of the chip's area stays idle doing nothing.

The need for processes running in parallel was obvious from the beginnings of software. With the advent of Unix a standard way came into being for software programmers to simulate concurrency in common processors by means of creating different processes or later threads. Today, in the era of multiple cores in commercial CPUs, there is true concurrency. The problem is that most of the concurrent programming models have been inherited from the past, where concurrency was only an illusion achieved through complex synchronization constructs. Whenever there is a shared resource, then concurrency becomes synchronization, and the only way to avoid this elegantly, is to correct the problem from its root; eliminate all shared resources.

Functional programming languages have long ago taught their proponents ways to circumvent the use of global variables, which is a synonym for shared resources. No state is held anywhere globally and the only means of communication is through arguments and results of function calls. Designing in hardware is not different than writing functions in a functional language. A hardware entity seen from above, works like a function, constantly producing output dependent only on its input like a black box. One of today's functional languages, Erlang was used to design a prototype of the proposed datapath, and at the same time a simulator of its instruction set. The inclusion of an introduction to the notions of a thought of as academic and perhaps obscure to most readers language, in the context of a hardware design Thesis, might be a surprise to you. Inspiration though for seemingly irrelevant things lives in these most obscure corners. Poincare, one of the biggest contribu-tors to the current understanding of mathematics, found the answers to his greatest problems while walking on the beach or getting on the bus. This strange correlation between totally irrelevant events made him realize a function of the mind which he named over-consciousness. I prompt you to investigate this.

4.1.2 *Erlang*

Erlang has been built from the ground up as a modern software language to tackle the problems of *concurrency*, *distribution* and *fault tolerance*. These three features are an absolute necessity for some specific kinds of applications like telecommunications, where the uptime of the system is the most important metric of its quality.

Erlang is a *functional* programming language. The main building block of its programs is the function. A program written in a functional programming language is a chain of functions which call other functions, each receiving input from the output of other functions or from the user.

Erlang is a *concurrent* programming language. It provides a way to easily create new, independent paths of execution, which automatically run truly in parallel if the underlying hardware supports this (*multiple cores*), or give the illusion of parallelism by means of a timesharing scheduler. A new path of execution is called a thread, and Erlang allows the programmer to create millions of threads even on mainstream hardware. This flexibility makes the thread a notion of the same importance in an Erlang program as the function and one usually sees that most of the functions of an Erlang program run on different threads. Moreover, similar to how functions communicate by means of argument passing and result values, Erlang defines a protocol and the needed semantics for the communication among different threads by means of exchanged messages. This model of communication is called the Actor model [REF].

Erlang is a *distributed* programming language. There is no distinction whether some thread is running locally or in another machine in the network. Erlang can exchange messages with local and remote processes under a unified syntax. Thus, even in the cases of single core CPUs, there can be true concurrency within a grid array of connected processing nodes. Moreover, Erlang automatically administers the credentials of the remote machines participating in this extended grid for safety reasons.

Finally, through distribution Erlang manages to ensure *fault-tolerance.* If a processing node fails (*from a hardware failure, not the programmer's design error*) then the rest of the nodes start receiving and processing its messages for it in a transparent way. Statistics of maintenance promise that at least one node will continue serving all the load, which makes fault-tolerance a reality.

Function definitions and message exchange are all supported by pattern matching in Erlang. Function arguments are pattern matched to identify the correct function definition according to the given arguments and message contents are pattern matched to decide what to do with the message. Pattern matching in binary data traveling through network ports from storage devices, enables the programer to define a communication protocol or unpack data with headers and sections in a most easy and elegant way.

Erlang is very strict with variables, which is normal since they are the reason behind most of the bugs found in concurrent programs. Variables do exist, but one cannot change their value, which is one of the things that immediately strikes as alien to most programmers approaching Erlang. When we learned our first programming language, we were shown things like "X = X + 1". Everyone protested, and definitely argued that "you can't do that!" so we had to unlearn what we learned in math class. In Erlang, variables are just like they were back in the math class. They are symbols associated with some value, an assertion, a statement of fact. And that's that.

This model of computation, which guarantees parallelism and imposes a nothing-shared state of mind, guarantees also that a prototype of an idea for a hardware design in Erlang could easily be translated into hardware with the use of some hardware description language like VHDL. As the simulator was developed I observed the following analogies between the two initially alien worlds:

Thus since debugging tools in software development environments are usually better than the hardware ones, Erlang was chosen as the best candidate to experi-

Erlang	**VHDL**
thread	entity
function argument	memory element
function body	process
variable	constant
message contents	data
message type	opcode
pattern matching	decoding

ment with the design of a Go board taking advantage of the concurrency of hardware for high performance. The transition to VHDL would be a matter of defining the correct bit widths for all messages exchanged and translating the function bodies to actual hardware circuits doing the same job.

For a complete reference of the Erlang programming language see [REF] and of VHDL see [REF].

4.2 Design Of A Parallel Board

Throughout this Thesis we will consider that there is an external module either implemented in software or in hardware that sends a move to be played. Moreover, since this board is part of a design tuned for MC simulations as a means of evaluating a board move, we consider that this move is random and thus can be illegal. As in most MC based implementations, we consider the following moves as illegal:

☞ play out of the board
☞ play on an occupied position
☞ play a suicide move
☞ play inside one's own eyes

With these considerations, a Go board supporting the basic operations on it, must meet the following requirements:

‣ store and retrieve board positions
‣ identify and remove captured stones
‣ emit errors for illegal moves

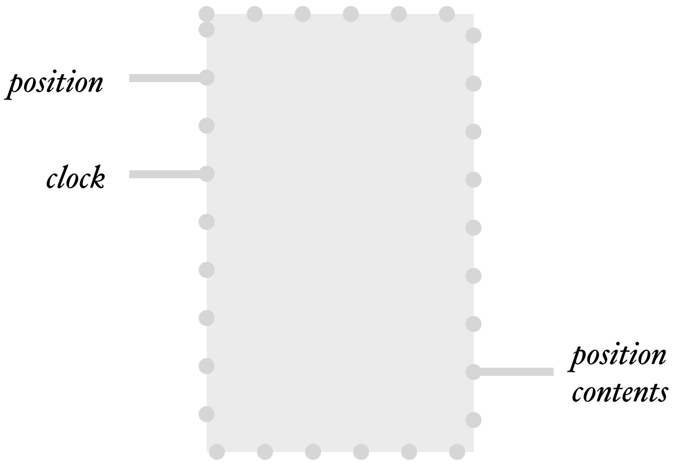
Additionally our board could support the following operations which though not basic, are necessary for a move generating module:

‣ report when the game has ended
‣ report the game's score when the game has ended

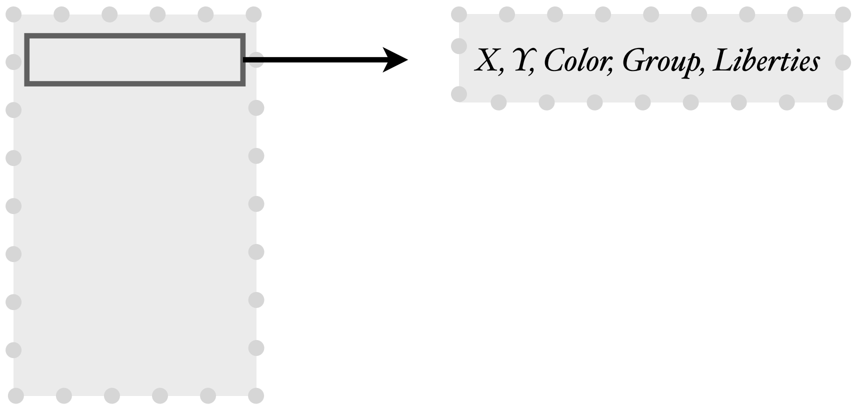
Note that estimating the game's score during a match is a difficult problem on itself both for computers and for humans. Beginner players are puzzled by the fact that they don't know whether they are winning or losing at any point of a match and are even more puzzled at the sight of professional players declaring their resignation even during the first 60 moves. To make this even more complex, there are many different score counting rules like the Chinese or the Japanese rules which can themselves affect the score of a game, to the point of reversing the winning and the losing player. All these mean, that there is no trivial way to count a match's score other than playing it to the end where only filled positions and eyes exist on the board. The player with the most eyes is the winner. This is the reason why in MC simulations games have to be played till the end.

Returning to our own subject, what we would like to do for a high performance board is to be able to store, retrieve, and tell whether some group is captured all in constant time. With the choice of an array in software or any block of memory addressed by the position's coordinates in hardware we could achieve the first two.

The following figure presents such a design:



Each memory slot must hold information on whether this position is occupied or not, and if yes what is the color of the stone lying on it. It could furthermore hold extra information such as being an eye or not or an identification code for the group it belongs to.



To decide whether a stone or a group of stones is captured, we have to examine all its neighbours. Chapter 2 already explained an algorithm used in software for the capturing problem. With this board representation we can store and retrieve positions in constant time but for the capturing decision we have to examine all neighbours in a sequential manner which makes this solution unattractive.

Another approach would be to create a new data structure able to identify groups of stones faster than linearly by holding extra information on stone formations. Even then though, as long as the algorithm has to examine the neighbours in order to decide whether a group is dead or not, we are bound to non-constant time complexity, dependent on the group's size, which itself has as an upper bound the size of the board. This is of great importance to realize.

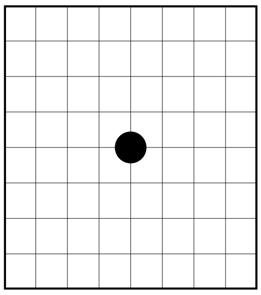
The only scenario in which stones could be captured in constant time, *independent* of the board size is when we will not have to examine them. And this can only happen when the stones know *on their own* when they are captured. If the stones could decide this themselves then they could also remove themselves, which is the same as setting their status to empty, without having to communicate this to anyone. Going this idea a bit further we realize that if such a design is possible, the stones which *independently* decide whether they are captured or not, start resembling the independent entities, which we talked about in the beginning of this chapter regarding concurrency. Since each stone's decision can be independent on the others' then all these decisions could naturally happen in parallel.

Playing the role of a single stone, to decide that I am captured I have to examine my liberties. If all my four liberties are occupied by opponent stones or by borders of the board then I am definitely captured. So for this simplified Go, where we meet only single stones on the board, each stone could be an independent entity which stores its color and examines all the time its neighbours.

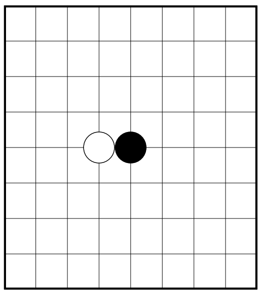
This scenario fails as soon as one neighbour is a stone of our own color. As a solution, I, the stone, could as well retain information on my neighbour's neighbours, too. If he says he is not captured, then I, belonging to the same group as he could tell this, too. Soon one realizes though the recursive nature of this problem. In a huge group every stone has to be connected to all others and having to examine all these stones' neighbours is nothing else than the original scheme of examining linearly all the neighbours of a stone to decide whether it is captured or not; except if someone imagines a huge AND gate for each entity combining all the neighbours' state in one flag of whether we are captured.

The key point to realize here though, is that what we are interested in, when stones form groups, are not the liberties of each stone independently, but the liberties of the whole group. A single stone in a group is captured when the whole group is devoid of liberties. It cannot be captured alone, which is also the essence behind the formation of solidly connected groups. Taking this a bit further I could argue that there is no single stone in fact, but only groups of stones, and a single stone is nothing else than a group which consists of one stone.

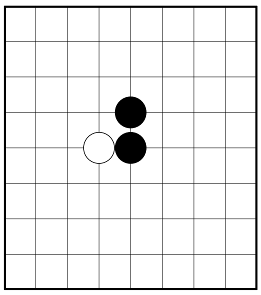
At this point we know that each independent stone entity has to keep track of its color and the liberties of the group it belongs to. The question now becomes "*how can we count a group's liberties in constant time?*" . The answer here is given by the very incremental nature of the Go game. As in the first chapter, playing a game of Go is the best way to see the formation and the capturing of a group without any loss of generality.



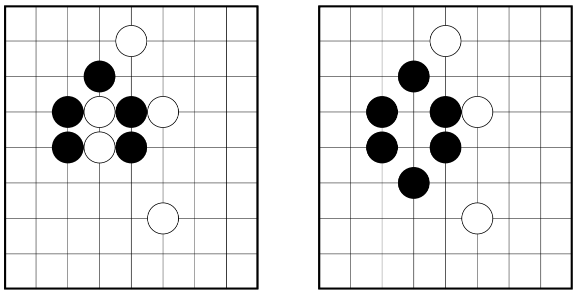
A single stone is placed on the board. This stone checks its neighbours. All are empty positions thus this stone forms a new group which consists only of itself and has four liberties.



A white stone is played adjacent to the black stone afterwards. The stone checks its neighbours again. Three of them are empty positions and one of them is a stone of the opposite color, thus it forms a new group which has three liberties. Moreover, the black stone has its group's liberties reduced by one.



When a new black stone is placed next to the first black stone, a new group is created from the merging of the two black stones. This is technically the same as having the new black stone merge with the first's group. The extended group has now five liberties or the sum of the stones' liberties before the merging minus the two liberties reduced due to the touching point.



The above figures show a possible development of the match which ends in the first figure just before the capturing of the white group while the second figure shows the board after the capturing of the white group.

From the above example we observed that we want to be able to do the following in constant time:

- reduce a group's liberties
- merge two groups keeping track of the total group liberties

This leads us to search for a way to be able to address stones by group as much as by position, since both of the operations that we identified, refer to groups. In the first case all the stones belonging to a given group have to reduce their liberties by some given number and in the second case all stones belonging to a group have to move to another group carrying along their own liberties.

We have agreed till now that all the board positions are implemented as independent entities which communicate with some sort of messages. These messages could contain information on neighbouring positions or give them instructions such as "*decrease your liberties by one*", or "*merge with my group*". These are the messages that these independent entities will understand.

The solution to the above problems comes now when we realize that as much as an entity can accept messages from others, the same way can it also *ignore* those messages that do not refer to it. A stone, which knows that it belongs to group number 1 for example can ignore a message like *"group 3 decrease your liberties by one because an opponent stone was played adjacent to you"* which refers to a group on the opposite corner of the board.

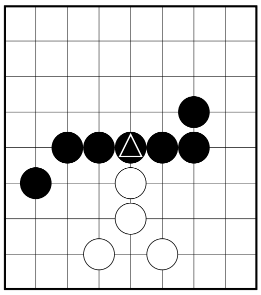
Thus, a grid of entities which store all their attributes like color, group number, number of liberties etc. can be addressed by any of them easily by having them receive all the messages sent to everybody but care only for those referring to themselves, similarly to how network packets flow in an ethernet network, or similarly to how one could have among a group of people, only those wearing a red t-shirt, jump. One doesn't have to examine each one and repeat his plead; he sends the message to everybody and he is sure that his request is going to be granted as long as the t-shirt wearers have been given the intelligence to compare their t-shirt color to the message's t-shirt color.

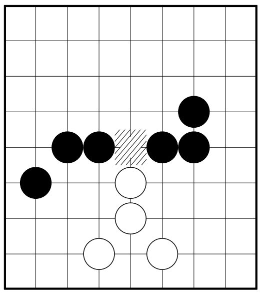
The only problem which seems to go hand in hand with this strategy is the message conflict problem. If one entity broadcasts a message and at the same time another entity broadcasts another, how can we guarantee that both messages reach successfully their destination? Can this ever happen? The answer comes again from the sequentiality of the Go game. Playing one stone at a time, *only the currently played stone causes changes to the state of the board*. This means, that if every cycle one move is played, then the entity which corresponds to this position will be the only one in this cycle that will have to send some message to some other entities. Thus, each entity receives only one message every time, which is the same message that all the other entities receive, having as source the currently played entity. Under the assumption that all operations needed for capturing can be contained in a single message, then every cycle one new move is played on board and in the beginning of the next cycle all entities decide whether the broadcasted message refers to them and act accordingly. This guarantees by definition that no two messages could cause two different, conflicting changes at the board, which is synonym to saying that this design guarantees that there cannot be any communication deadlocks.

Implementing the broadcasting feature is a matter of designing a common bus to which all entities are connected. The entities could either send and receive messages from the same bus, which demands for a bidirectional bus design, or have separate channels to read and write messages which demands two separate buses.

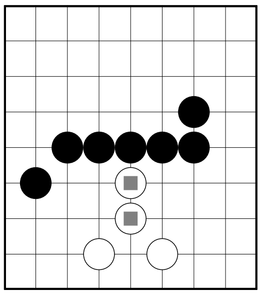
Thus, as a group is formed, the affected groups respond to the broadcasted messages telling them either to decrease their liberties by one because of the new stone placed, or merge with the new stone and add their libs to the new stone's into one new group.

The following figures show such a complex case where two existing groups merge and some opponent group has to decrease its liberties by one. The first figure presents the move marked with a triangle and the following figures explain the series of messages sent to the entities marking along the entities which respond each time to the message:



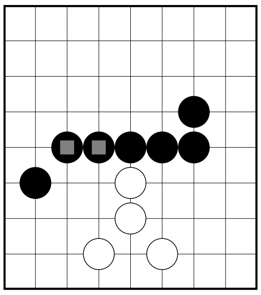


External Module - Set E 5 to black

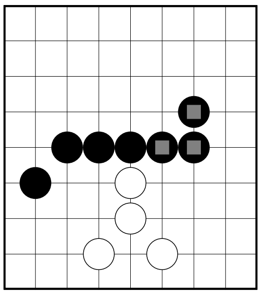


E 5 - White group at South decrease your liberties by one

E 5 - Black group at Weſt merge with me and add my liberties to yours [At the same time E 5 fixes his own group indicator and liberties to conform to the reſt of the group's ſtones]



E 5 - Black group at Eaſt merge with me and add my liberties to yours [The same as above. Now E 5 has the total liberties after merging with the Eaſt group and the Weſt]

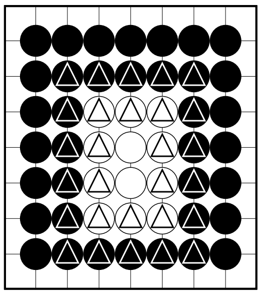


Another key observation is that even if another, unconnected group existed in the upper left corner, the entity at E 5 does not have to care about it as it cannot possibly affect it liberties. That means, that each entity which is enabled after a move is sent on it, has to care only for its four neighbouring entities, which sit on its four liberties. The four neighbours of each entity carry all the information we need to inform them with messages on reducing their liberties or merging. Let me note at this point, that in another scenario, where the board design would have to support a complex evaluation of the stones, considering future group connections, we should indeed care about that separate group lying in the upper left corner. You, reader, must always keep in mind that this design is indeed built to be efficient but its development is constantly driven by the need for fast Monte-Carlo simulations. Re-examine all ideas presented in here and unite them accordingly for your own needs.

What we created till now, is a design capable of changing one position's color, of merging groups and of counting group liberties, all in constant time. These are enough to support all the states through which groups of stones being captured by other groups of stones pass. Each entity checks its liberties, which matches the liberties of all the other entities belonging to the same group, and when they reach zero, the entity and its group is guaranteed by definition to be captured. At this point the entity can reset its state to an empty position.

The last step before the capturing process is complete, is the return of the liberties to the opponent groups that caused the capture. As explained, in the previous chapter, programmers till now have been removing the stones of the groups that caused the capture and replaying them after the capture. In the absence of the captured group, the liberties are counted once more.

Clearly this solution is inadequate for the above design. It takes many steps to complete and totally deviates from the concurrent philosophy of the independent entities. Still this is a major problem on its own. How can a group of stones which is captured, return immediately the shared liberties to the capturing group? Observing the problem on the following board, what we would like to do is have the the outer white stones of the captured group give one liberty to every stone they touch of the black group. The sum of these shared liberties is the amount of liberties that have to be added to the capturing group.



This creates two problems:

‣ Stones of the black group not touching the captured white group have no means of receiving these returned liberties. This creates a mismatch between the number of liberties ſtored in the the inner black ſtones and the outer ones, while our design demands all ſtones of the same group to be in synch.
‣ The sum of the perimeter liberties returned is indeed the number of liberties that have to be added. But having each ſtone give one liberty to the adjacent opponent the group's liberties are eventually only increased by one.

Adding those single liberties with some sort of variable input adder seems complex enough and will probably take a variable number of steps to be completed (*a stack in which those liberties have to be pushed might be employed for such an operation*). Once again, the redistribution of liberties has to be done automatically, independently. And exactly as with capturing and the counting of liberties, this task has to be once again left on the bookkeeping of each of the independent entities. Our question now is "*how can such an entity know how many liberties have to be added to its group after it captures an opponent group?*". This demands two things:

‣ Firſt, the capturing group muſt know when the opponent group is captured.

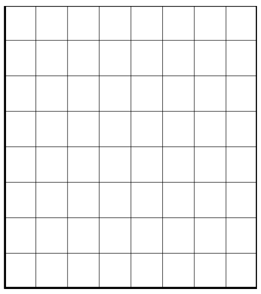
▸ Second, the capturing group must keep track of the liberties shared with that captured group.

The first prerequisite is easily granted if we remember that each entity is directly connected to its four surrounding stones. This connection lets the entity know about the neighbouring group's group identification number, its liberties and its color. When we see that an opponent neighbouring group has one liberty, and one stone is played adjacent to it, then we know beforehand that in the next cycle, that group will realize its death and all its entities will return to the empty state. At this point we can add the appropriate liberties to ourselves by sending a message to our group, having it increase its liberties by the required amount.

The second is the most important part. Observing the example dialog between the entity that responds to the current move and the messages it has to send one can identify the exact point where the shortage of liberties takes place. The solution lies in the message *"E 5 - White group at South decrease your liberties by one"*. From this message, the white group at South knows that the group represented by the E 5 entity asks it to decrease its liberties by one, due to a new touching point. Going this a bit further, we can realize that each group that asks us to decrease our liberties is a neighbour that touches us. By having one counter, for each one of our touching groups, increase when a new touching point is created with a neighbouring group, we can any time answer the question *"how many liberties do you share with that neighbouring group?"*. The above example though can be misleading. A group can touch many opponent groups all at once. Keeping track of the touching points with each one of these occupies a huge amount of space. The maximum number of distinct touching groups can be half the maximum number of liberties of a group, since they have to be connected to our perimeter stones leaving one space in-between them. Thus, we want for each entity an array of size *max_liberties/2 * max_liberties/2*. Only then can we keep track all the time how many liberties we share with each one of our adjacent groups. Exactly as with the problem of counting the group liberties, this new tradeoff is a choice that you, reader have to take according to the absolute speed considerations of this design. Therefore, I present the above solutions here for the sake of completeness.

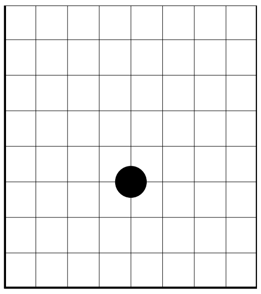
Getting away from the above solutions though, you can implement the return of the captured group's liberties in a fashion similar to the accumulation of the liberties in a merging group. Exactly as the liberties are added up to a group each time a new stone causes the merging of some groups, that is incrementally, the captured group's stones could be removed from the board one after the other causing the opposite effects of adding them. Each time an opponent stone is removed its surroundings will update their liberties. I will call this strategy "undo" strategy, as it is the opposite of the "move" operation. Removing a group from the board is a multiple step operation with this strategy. A fast circuit clock on the other hand guarantees that this solution is preferable to any of the above relying on extremely heavy bookkeeping. The move generating module will generate "undo" opcodes for all of the stones belonging to a captured group as soon as a group becomes captured which can be known in constant time.

The communication protocol is presented below in a real world example of message passing between all the entities of a 9x9 board. From left to right I show first the board configuration, which implies also the positions of the entities, the message currently on the bus and a table of all the inner variables of each entity. Note that effects to the board and to the table happen one cycle after the message has reached the bus, so as to give a better emulation of the hardware circuit's behaviour.



[*new_move , e , 4 , black , 1 , 4*]
the external module sends a new move to the board

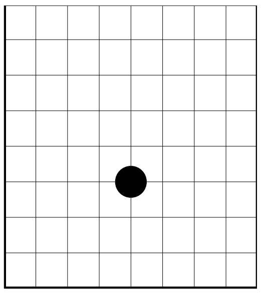
Y	X	Color	Group ID	Liberties
1	a	empty	0	0
2	a	empty	0	0
...				
...				
...				
9	j	empty	0	0



[*modify_liberties, 1, 0] OURSELVES*
[*modify_liberties, 0, 0] NORTH*
[*modify_liberties, 0, 0] EAST*
[*modify_liberties, 0, 0] SOUTH*
[*modify_liberties, 0, 0] WEST*

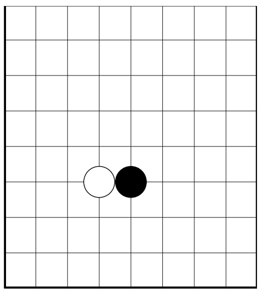
entity e4 sends its attributes to its four neighbours. all are
empty so none responds back. The above messages are sent in a serial manner
but are here presented all together.

Y	X	Color	Group ID	Liberties
1	a	empty	0	0
2	a	empty	0	0
...				
4	e	black	1	4
...				
9	j	empty	0	0



[*new_move* , *d* , *4* , *white* , *2* , *4*]
the external module sends a new move to the board

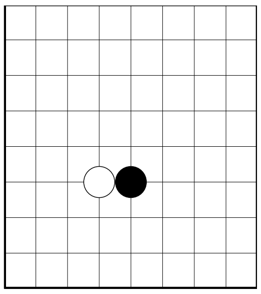
Y	X	Color	Group ID	Liberties
1	a	empty	0	0
2	a	empty	0	0
...				
4	e	black	1	4
...				
9	j	empty	0	0



[*modify_liberties, 2, -1] OURSELVES*
[*modify_liberties, 0, 0] NORTH*
[*modify_liberties, 1, -1] EAST*
[*modify_liberties, 0, 0] SOUTH*
[*modify_liberties, 0, 0] WEST*

entity d4 reduces its own group's liberties and the liberties of the adjacent black group.

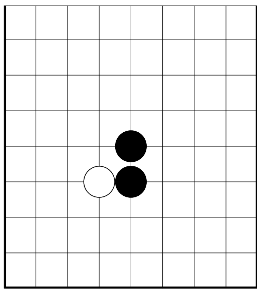
Y	X	Color	Group ID	Liberties
1	a	empty	0	0
2	a	empty	0	0
…				
4	d	white	2	4
4	e	black	1	4
…				
9	j	empty	0	0



[*new_move , e , 5 , black , 3 , 4*]
the external module sends a new move to the board

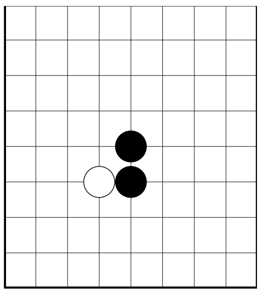
Y	X	Color	Group ID	Liberties
1	a	empty	0	0

Y	X	Color	Group ID	Liberties
2	a	empty	0	0
...				
4	d	white	2	3
4	e	black	1	3
...				
9	j	empty	0	0



[merge, 3, 1 , 5]
[modify_liberties, 0, 0] NORTH
[modify_liberties, 0, 0] EAST
[modify_liberties, 1, 2] SOUTH
[modify_liberties, 0, 0] WEST

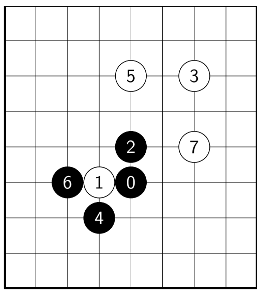
Y	X	Color	Group ID	Liberties
1	a	empty	0	0
2	a	empty	0	0
...				
4	d	white	2	3
4	e	black	1	3
...				
5	e	black	3	4
...				
9	j	empty	0	0



[*new_move , e , 3 , white , 4 , 4*]
the external module sends a new move to the board

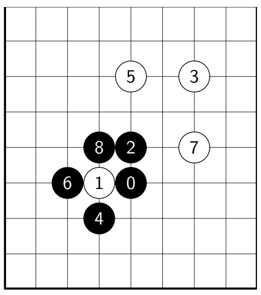
Y	X	Color	Group ID	Liberties
1	a	empty	0	0
2	a	empty	0	0
...				
4	d	white	2	3
4	e	black	1	5
...				
5	e	black	1	5
...				
9	j	empty	0	0

Some moves are omitted at this point. Suppose that the match continues as the following board shows, to present an example communication for a capture of an opponent stone.



[*new_move , d , 5 , black , 4 , 4*]
the external module sends a new move to the board

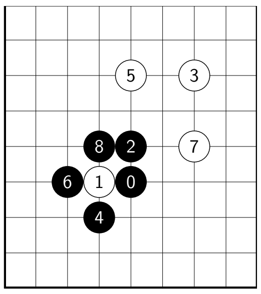
Y	X	Color	Group ID	Liberties
1	a	empty	0	0
2	a	empty	0	0
...				
4	c	black	6	3
...				
3	d	black	4	3
4	d	white	2	1
4	e	black	1	5
...				
7	e	white	5	4
...				
5	e	black	1	5
...				
5	g	white	7	4
7	g	white	3	4
...				
9	j	empty	0	0



[*merge, 8, 1, 6*]
[*modify_liberties , 0 , 0*] NORTH
[*modify_liberties, 1, 1*] EAST
[*modify_liberties, 2, -1*] SOUTH
[*modify_liberties, 0, 0*] WEST

Y	X	Color	Group ID	Liberties
1	a	empty	0	0
2	a	empty	0	0
...				
4	c	black	6	3
...				
3	d	black	4	3
4	d	white	2	1
5	d	black	8	4
...				
4	e	black	1	5
...				
7	e	white	5	4
...				
5	e	black	1	5
...				
5	g	white	7	4
7	g	white	3	4
...				

Y	X	Color	Group ID	Liberties
9	j	empty	0	0

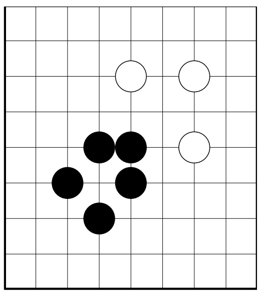


[*undo, d, 4, white*]
[*modify_liberties , 1 , 1*] NORTH
[*modify_liberties, 1, 1*] EAST
[*modify_liberties, 4, 1*] SOUTH
[*modify_liberties, 6, 1*] WEST

the white group 2 is captured and the external module sends an undo opcode

Y	X	Color	Group ID	Liberties
1	a	empty	0	0
2	a	empty	0	0
...				
4	c	black	6	3
...				
3	d	black	4	3
4	d	white	2	0
5	d	black	1	6
...				
4	e	black	1	6
...				
7	e	white	5	4
...				
5	e	black	1	6

Y	X	Color	Group ID	Liberties
...				
5	g	white	7	4
7	g	white	3	4
...				
9	j	empty	0	0



Some new move comes at this point from the external entity. We just show below the board's final state after the capturing has taken place.

Y	X	Color	Group ID	Liberties
1	a	empty	0	0
2	a	empty	0	0
...				
4	c	black	6	4
...				
3	d	black	4	4
4	d	empty	0	0
5	d	black	1	9
...				
4	e	black	1	9
...				
7	e	white	5	4
...				

Y	X	Color	Group ID	Liberties
5	e	black	1	9
...				
5	g	white	7	4
7	g	white	3	4
...				
9	j	empty	0	0

As a final note, observe that the true liberties of the black group with id 1 are not 9 as computed, but 6. Common liberties among stones of the same group such as those in position d 4 are counted once for each one of the group's stones. Thus, when the d 4 stone is captured, not one, but three liberties are added to the black group. At the same time though, if a new white stone was played there (*we suppose that c4 or d3 did not exist, so that d4 wouldn't be considered illegal)* we would steal three liberties from group 1, one for each touching point. This abnormality has to occur for optimization reasons. You can see on your own, that to identify liberties as shared with another stone of the same group and not count them as duplicates, we have to examine not our four neighbours but the four diagonals, too. This is a fundamental architectural change with obvious new space considerations.

4.3 Design Of A Simulator

The simulator is an direct software implementation of the previous ideas in Erlang. In Section ... I explained the possible mappings between an Erlang construct and a hardware circuit element. Following this convention, each independent, board position entity described in the previous section is modeled now as a separate thread. Messages exchanged between entities are data packets sent among threads with Erlang's standard message passing facilities, and the bus is seamlessly provided by Erlang's inner architecture as explained below.

When the simulator starts, it creates a number of such independent entities, equal to the number of positions on the wanted board. So, for a 19x19 Go board, 361 new threads are created. Each entity holds all the information shown in the last figure of the previous section. Its X and Y coordinates are given initially and will never change as they are the basic id of the entity. These two are implemented as variables, while the rest of the needed attributes like color, liberties or group-id are given to the thread as function arguments on which the thread recurses every time a new message is received. Thus, if a message alters our liberties, we call ourselves again with the same arguments but for the liberties which is substituted with the message's value. When a message is received, which does not refer to us, we call ourselves again with all our arguments unmodified. This is by the way the most usual convention in functional languages to hold state.

All these threads wait idle until they receive some message. Once again an external module generating move positions is taken for granted, and we substitute it by broadcasting manually a new move message to all threads. The new move message has the form:

{ new move , 3 , 3 , black }

All entities decode this message and decide that this message's type is a "*newmove*" message. Choosing the correct execution path they will compare their X and Y values to 3 and 3 and only one will pass from this test, the one we would like to receive the message from the beginning. At this time, entity E[3,3] must check its neighbours. Considering every possible case, for merging, reducing its liberties or reducing its neighbours' liberties, it will broadcast the appropriate messages.

Until this series of messages has ended, and the new-move process is completed, no other entities should use the bus to broadcast anything as it would interrupt the messages E[3,3] send, possibly creating adverse effects. As explained before this synchronization problem is managed by a convention. Only the "*newmove*" message creates a response. All other messages are received and processed by entities without having to broadcast anything afterwards. This is important, if you, reader, want to extend the instruction set of the entities. Perhaps for the needs of a move evaluator one would need an new instruction that would return the inner information on group color and liberties of a single entity.

If new commands are needed, which create an answer or a status code, then a new probably more complex bus design must take care of synchronizing the messages by putting them in queues and reorganizing them. In fact, great caution has been taken to avoid such a path. Erlang gives each thread a construct called mailbox. All messages coming to a thread are stored in this mailbox. This means:

‣ No messages are ever lost. Even if a multitude of messages arrive at the same time, they will be queued and processed. This is convenient and a well-reasoned choice from Erlang's point of view, but a major design undertaking on its own for a hardware translation of the simulator.
‣ Messages which are not found to match some of the thread's accepted message types, lie dormant in the mailbox for future use in a second queue. This allows one to redefine the program code in runtime and have the program accumulate the changes immediately and responding as if things always where like the new version of code. This feature is one of the fundamental points making an Erlang program completely dynamic, supporting code-swapping and updates on the fly.

Unfortunately, the messages in the suggested design have to be sent in the correct order, thus a simple queuing scheme cannot guarantee the faultless operation of the design. Suppose that a new white stone is placed next to a white group with only one liberty. If a new "*reduce liberties by one*" message reaches the white group before the "*merge*" message comes, then the group will think it is dead and will thus remove itself from the board. Whereas, normally, the merging would give this group new liberties before or at the same time as reducing its own, and the expanded group would live on the board. This proves that you will have to either redesign the instruction set so that the order of the messages is independent of the final board state, or you will have to implement a queuing and message reorganizing design.

When the board has "settled" after a new move, then a new cycle starts. The board informs the external module that it is ready to receive a new move and the module sends a new "*new-move*" message. Such a communication with the move generating module is necessary as in other cases the external module would generate moves faster or slower resulting in lost moves and errors about moves played on occupied positions respectively.

The following table presents the instruction set created for the independent entities:

INSTRUCTION	DESCRIPTION	EXAMPLE
reset	*resets the entity to an empty position state. comes from external module*	reset
new move	*sends a new move tuple to all entities. comes from external module*	[newmove, 3 , 3 , black]
merge	*the group receiving this message changes its group id to the message's group id*	[merge , 1 , 2 , 6] *(group 1 will become 2 with 6 liberties)*

INSTRUCTION	DESCRIPTION	EXAMPLE
modify	*modifies all attributes of an entity*	[modify , 3 , 3 , 1 , 10] *(modify entity 3, 3 to group 1 and 10 liberties)*
modify liberties	*increase or decrease the liberties of a group by a number*	[modify , 11 , -1] *(reduce group's 11 liberties by one)*

Moreover, the simulator defines the following instruction for debugging purposes which clearly cannot be synthesized:

INSTRUCTION	DESCRIPTION	EXAMPLE
print	*prints the entity's attributes to the standard output*	[print , 3 , 3]

The following chapter moves to the hardware implementation of this design, where under a new light, the light of hardware efficiency, the architecture matures and now inspires itself changes to the simulator, from which it was originally born.

DESIGN & IMPLEMENTATION

Transition to Hardware

Nowadays the word Giga seems to be ubiquitous in the field of computer science; giga in cycles per second of the CPU, in bytes of RAM, in secondary storage devices, in intercommunication speeds. This abundance of computer resources has lead to an emergence of rich development tools with optimizing compilers and effective debugging tools, seen from the developer's standpoint. Usually, in an ironic mood, people quote Bill Gates's prediction "*64k ought to be enough RAM for anybody.*" said back in 1981. Desktop application programming never had any serious space considerations and thus the quote can be easily reinterpreted today as *"16GB ought to be enough RAM for anybody",* again a probably humorous statement for a future reader.

But high performance computing always strived for different goals. Space becomes a secondary consideration only for the sake of speed, not for the sake of some programmer's convenience. Trade-offs in space, time and costs of development are prevalent in every choice a hardware designer has to make and these new points of view couldn't but redefine once again a design. Thus in this chapter, we move from a functional architecture (*no pun intended*) to an implementable architecture.

5.1 Towards a Master Slave Design

I will not delve into the details of the VHDL language used for the description of the wanted hardware design. It is of great importance though to any VHDL programmer to know the inner workings of his tool of choice. The description, synthesis and simulation of all basic hardware building blocks such as registers, simple gates, adders and multiplexers is a most important phase from which one has to go through in order to be able to pre-visualize the synthesis's results from a given description. Such an observation, aided by Xilinx's invaluable hardware design tools has allowed me to program in a high level manner in VHDL, letting the compiler take all the important decisions according to the implementation strategy and the target platform, whether it would be a space reducing strategy or a circuit-speed one.

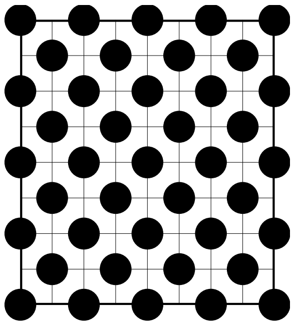
Based on the previous chapter's analysis, our board building block is the independent position entity. From a hardware point of view, this entity which is capable of receiving some data from its environment, processing them, storing and modifying internal data and occasionally produce output, is a simple processor, which has a quiet unusual instruction set compared to the mainstream, general purpose processors. Instead of asking it to *"add two registers and return the result"* our processor knows how to *"reduce your liberties by two"* or *"merge with the north group"*.

Two discrete hardware modules have to be designed for this circuit, the position entity which will be duplicated numerous times according to the board size, and the intraconnection bus. Following our convention the following table shows the different parts of the entity described in the previous chapter and their hardware counterpart.

SIMULATOR PART	TYPE	HARDWARE PART
X coordinate	*constant*	*fixed wiring*
Y coordinate	*constant*	*fixed wiring*
Color	*variable*	*register*
Group ID	*variable*	*register*
Group Liberties	*variable*	*register*
message processing	*function*	*combinatorial logic*
message	*data packet*	*raw data*
message type	*constant*	*opcode*
communication port	*socket*	*in out port (wiring)*

For all of the above types we have to examine their size in bits. How many bits are needed for all the possible values of a coordinate or a group id? It is easy to see that all types' sizes, except for color, depend on the size of the board. The coordinates of a position, for a board of size S can have values from 1 to S. Further observations and research has to be done though for variables like the group id or the liberties of a group. We have to answer these questions, "*What is the maximum number of distinct groups that could form on a board?*" and "*What is the maximum amount of liberties one group could possess on a board?*"

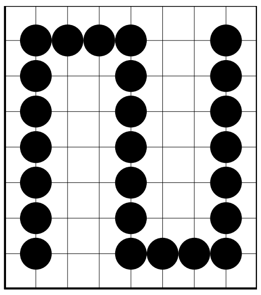
Regarding the first question, the maximum number of distinct groups on a board is met when we fill the the board with the maximum number of unconnected stones, as each stone represents a different group. The following board shows this stone arrangement.



With analogy one can infer that different board sizes exhibit the maximum number of distinct groups from a similar stone arrangement, thus we can correlate the board size with the maximum number of groups with the following formula:

$$\text{Groups}_{max} = \lceil BoardSize/2 \rceil * \lceil BoardSize/2 \rceil + \lfloor BoardSize/2 \rfloor * \lfloor BoardSize/2 \rfloor$$

As far as the second question is concerned, the maximum number of liberties one group can have depends on the group's shape. A group exhibiting a maximum number of liberties would be have to span all the board, similar to the main (*and only*) character in the famous "Snake" arcade game. This non-solid shape creates the maximum perimeter which is synonym to the maximum number of liberties. The following board shows such an arrangement of stones:



Contrary to the above analysis regarding groups, this is a more complex problem to tackle. The above stone formation gives us an upper bound of liberties for a board:

$$\text{Liberties}_{max} = 2*(n-2)+2*\lceil BoardSize/4 \rceil *(n-3)+\lfloor BoardSize/4 \rfloor *4$$

An important note regarding the upper bound is that its nature is quiet theoretical, since we haven't taken into account the intermingling stones of the opposite color which would decrease the above number by a half. This safe reduction is translated in one less bit of needed space. Thus in average:

$$\text{Liberties}_{average} = Liberties_{max}/2$$

The following table sums up the needed space in bits for each entity attribute as a function of the board's size when the two elements are correlated or as a constant number when the the attribute's size is independent of the board's size.

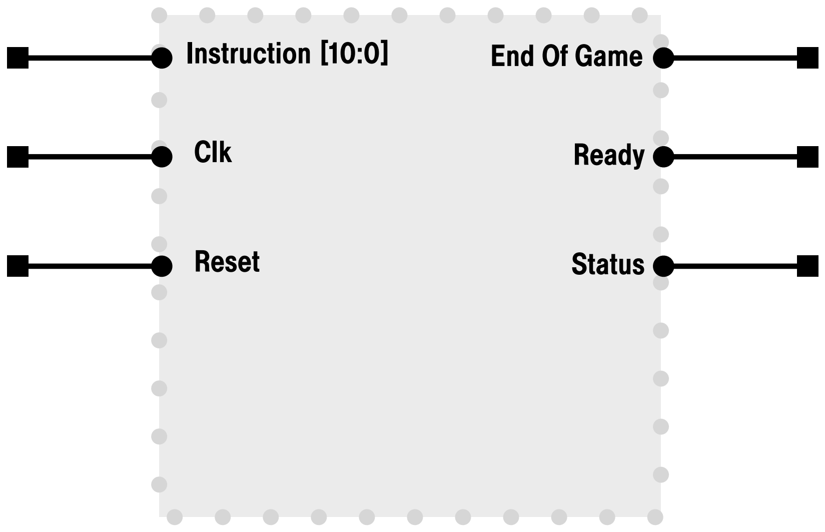
FIELD	**SIZE IN BITS**
coordinate	*ceil[lg(board_size)]*
color	*2*
group id	*ceil[lg(groups_max)]*
liberties	*ceil[lg(liberties_max)]*

FIELD	SIZE IN BITS
opcode	*3*
message	*opcode+2*coordinate+color +group_id+liberties*

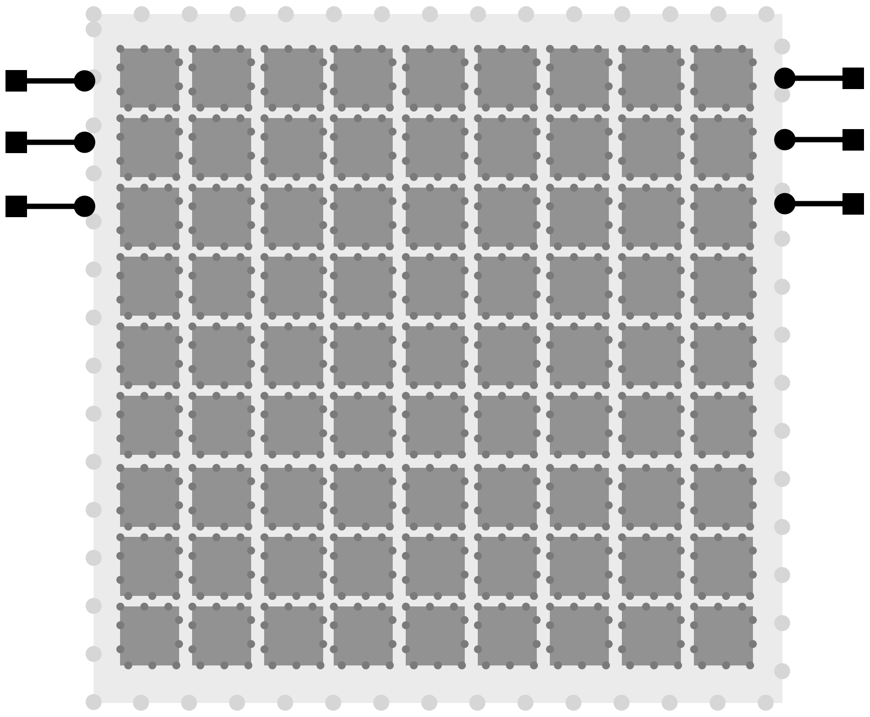
Following a convention and giving the zero group id to all empty positions and borders we can eliminate one bit from the color attribute, effectively making it one bit, as the identification of the nature of each position could be done by combining the information of the color and the group id. This optimization arises from the observation that in our architecture, color and group id information are always present in each message broadcasted. Moreover, it is preferable to eliminate one bit from a wide bus traveling among 381 entities for the 19x19 Go board than striving to use small comparators; FPGA's performance is largely affected by long wiring.

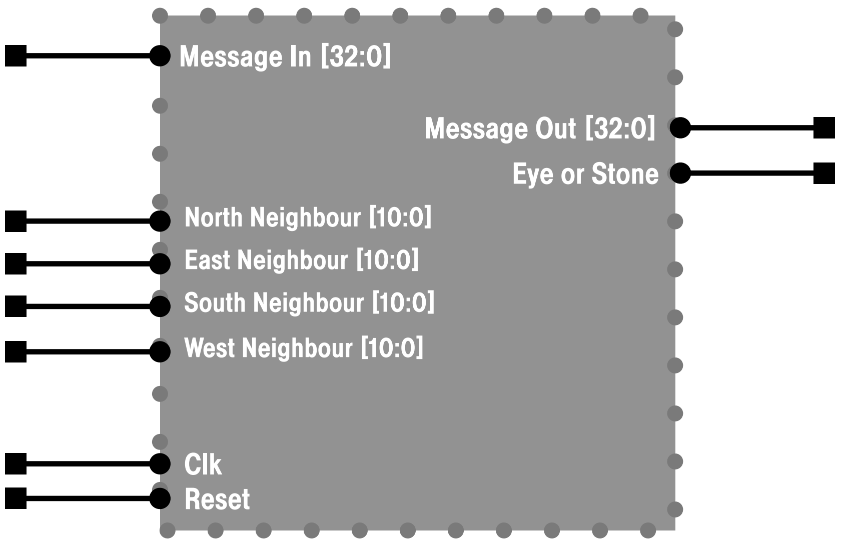
Due to these complex relationships, the GENERIC feature that VHDL provides for compile time configuration of parameters proves inadequate. To avoid writing an unreadable source code, I wrote a centralized script which given a board size produces all the wanted files for the software simulator and the hardware design. This script written in Python constitutes the single source code file required for the whole project and is written according to the premises of Literate Programming; source code and documentation are intermingled creating a readable text reminiscent more of a literature book than a source code aided by comments. For more on Literate Programming see [REF].

The following figures show an assembly of the above constructs into a concrete hardware entity implementing the wanted functionality for a 9x9 Goban:



The top level Circuit. *Instruction* comes from the external module generating moves.


The 81 entities inside the board.



The above entity's design was described in VHDL and synthesized for the Xilinx 5 50 FPGA within the Xilinx IDE. At this point we can examine a final hardware circuit with almost physical characteristics such as space consumption, arrangement of elements and speed. Two immediate observations were made after this first synthesis:

‣ The Xilinx compiler successfully figured out the correct implementation details from a high level behavioral description of the circuit. No more registers were created anywhere apart from those that I had in mind, which would create most of

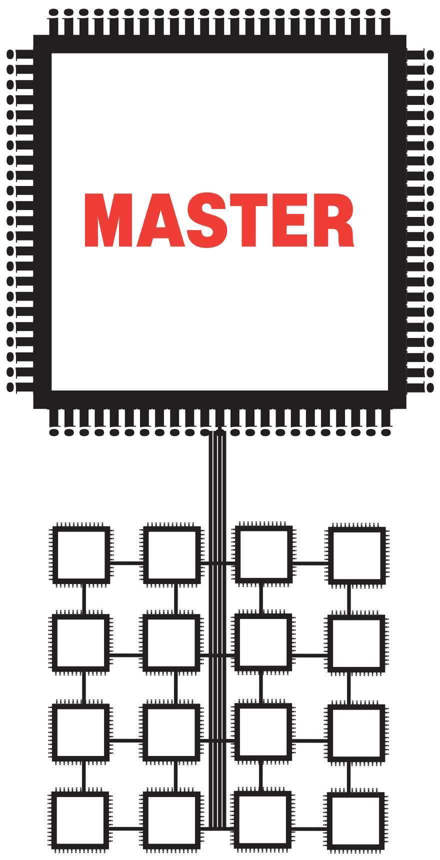
the times perplexities in the operation of the circuit, and the message decoding circuit was efficiently implemented as a series of multiplexers exactly as visualized.
‣ The size of the design is prohibitive for a 9x9 board on this FPGA. Either we would have to use a larger FPGA or modify the design.

A test-bench confirmed the correct operation of the circuit (*see next Section for a complete evaluation*), but the space problem still exists. If we generate with our script all sources for a 9x9 Go board and synthesize it we realize this; the resulting circuit needs 240% of the space provided by the chosen FPGA. Considering that a standard 19x19 board is our goal, we have to reiterate through the design.

The major issue of space consumption is caused by the duplication of a specific circuit; that is the message broadcasting circuit. This circuit receives all information coming from the four neighbours, that is the color, the group and the liberties of each neighbour and after examining them decides what kind of message has to be sent to each one. All the broadcasted messages are emitted from this circuit of each slave. This part itself occupies 2% of the available FPGA space due to the sheer amount of cases it has to consider. The exact number of them is equal to 256, the result of creating all the combinations for four neighbours and four possible states each one can take (*same color, opponent, empty, border*). If this circuit's size is multiplied by 81, which is the number of position entities that have to be generated for a 9x9 Go board, we already realize that this circuit alone is already above the available FPGA space. We either have to simplify this circuit or find a way to move it out of the slaves.

Observing once more the additive nature of the Go moves during a match, we realize that only one position has to take such a decision each time a new move is played; all the other positions have these message generating circuits completely idle. As we explained before, from a high performance point of view this over-consumption of space is normal and expected, on the other hand it constitutes a practical problem, when a circuit for a 9x9 Go board cannot fit into an average sized FPGA. Our new question is: "*Could we possibly have only one such circuit which could be used by all the independent entities, now that we know that only one will need it each time?*"

With the introduction of an external hardware entity, which would perform all the complex calculations for the messages that have to be sent across the board, similarly to a co-processor which once was a popular design choice, a new master-slave architecture starts to emerge. This center of calculations on which the slaves' state depends cannot but function as the core of the complete circuit. Moving all the calculations to this new processor, the board entities would only have to be able to receive a message and modify occasionally their inner registers, such as the color register or the group id register, and given that the entities are the ones which are multiplied as a Go board becomes larger in size, it is prudent enough to put *them* on a diet. Seen from above, this external CPU coordinates the modifications of the board state by sending instructions to be executed by the board entities which now operate as clever pieces of memory, able to decode and respond to incoming messages. The only drawback in this design is that, all the information which was internally readily available at each position entity for its decision circuit, now has to be transfered to this external CPU which will do the job for it. Such a wide bus design, which has to carry information on color, group and liberties for four neighbouring entities is a choice of fundamental design choice. We know that an FPGA's performance is heavily affected by long wiring that leads to great latency numbers. Moreover, as the board size increases we need to employ a wider bus, as the slave attributes increase in size.



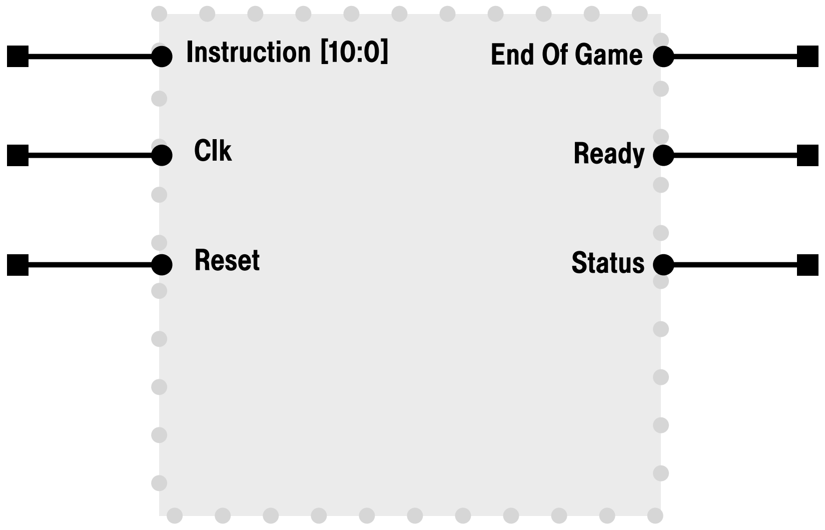
Performance-wise, the original architecture enables each entity to decide in one clock cycle the messages it has to broadcast, whereas in the revised master-slave architecture one cycle is spent to transfer the neighbour information through a bus to the external CPU and one cycle to receive the calculations' results. In other words, the effective circuit speed is theoretically halved. Splitting the messages sent to the master in smaller parts in order to allow a narrower bus would cripple the performance even more. At the same time though, space-wise, the synthesis process produced a circuit for the 9x9 board which occupies only 25% of the available FPGA space, and the 19x19 board oversized this FPGA size by only 15%.

The choice between these two architectures cannot be evaluated out of the application context. Where speed is absolutely the major consideration at every cost, the first design would be used and when a more balanced speed/space ratio is needed the second design is the preferred choice. My own opinion on this subject, reinforced by the performance evaluation analyzed in the next chapter which prophetically is used at this point, is that the second architecture is so fast on its own, that the extremely bad space usage of the first architecture is unjustifiable.

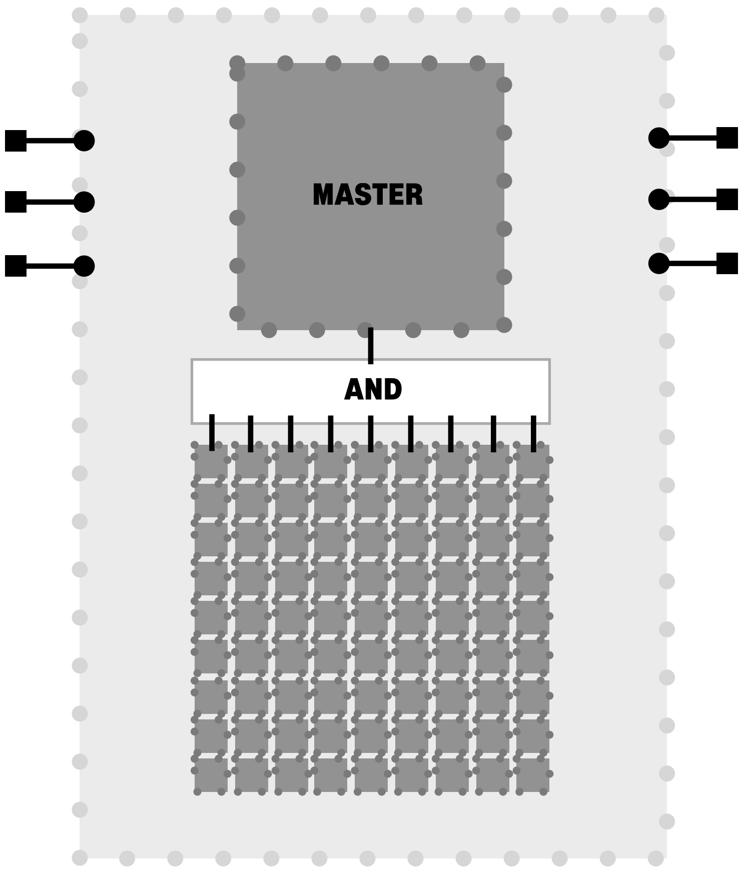
Due to the new communication protocol the messages now traveling through the bus are of two types, those going from the slaves to the master and those from the master back to the slaves. Their respective sizes are shown in the following in the following table:

FIELD	SIZE IN BITS
message to master	*4* [color + group + liberties]*
message to slaves	*X + Y + color + 4*group + liberties*

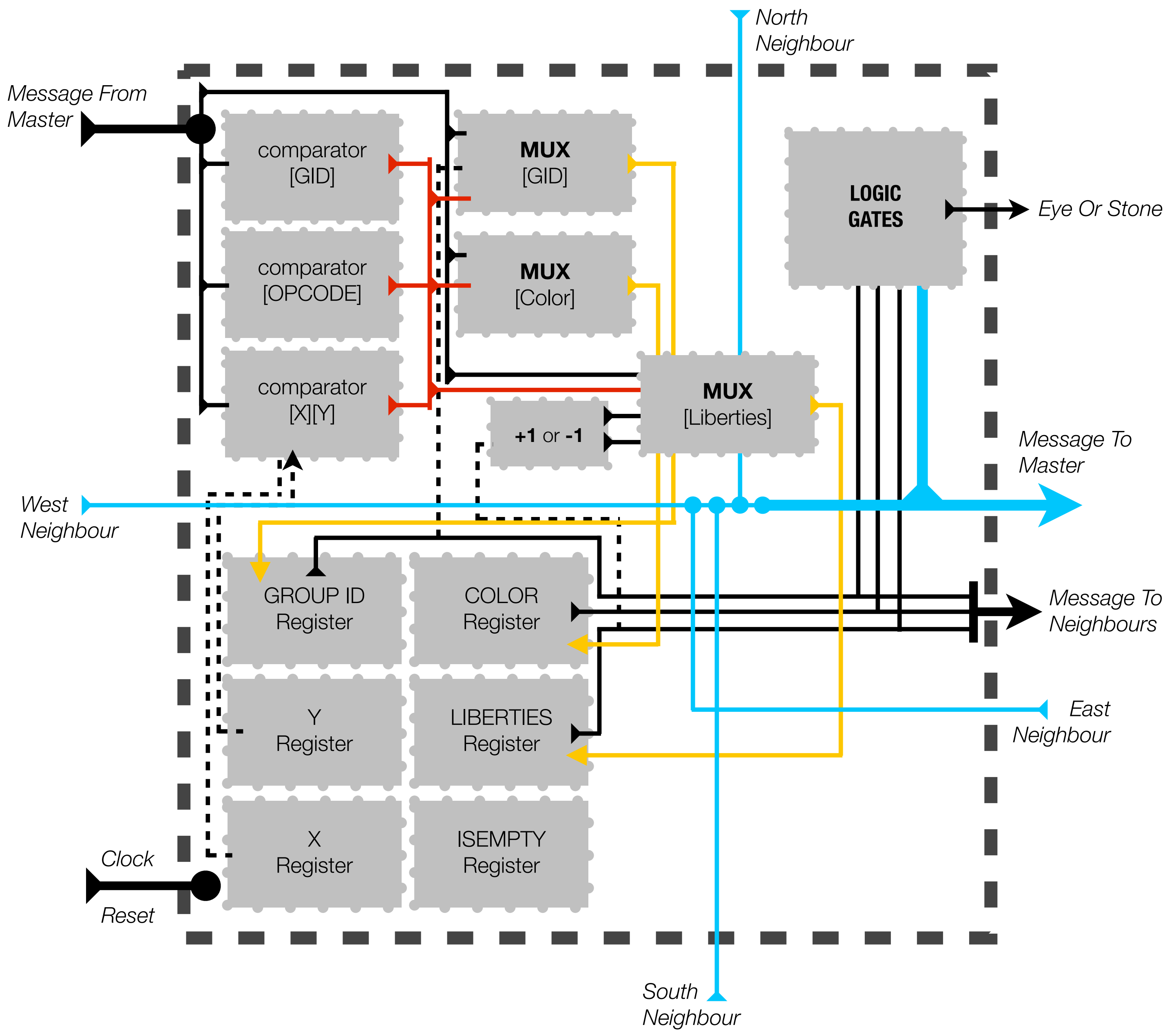
The following figures present the modified modules for this architecture:



The top level circuit remains the same. This is important as the communication with the external module generating moves is preserved intact despite the internal reorganization.



Master Slave architecture.



As this section closes to an end, some further assumptions made throughout this chapter have to be clarified; the exact structure of the messages exchanged between master and slaves, the structure of the messages reaching and going out of the complete master-slave entity for its communication with the outer world, that is the external API of this hardware Go board and the design of the master-slave interconnecting bus.

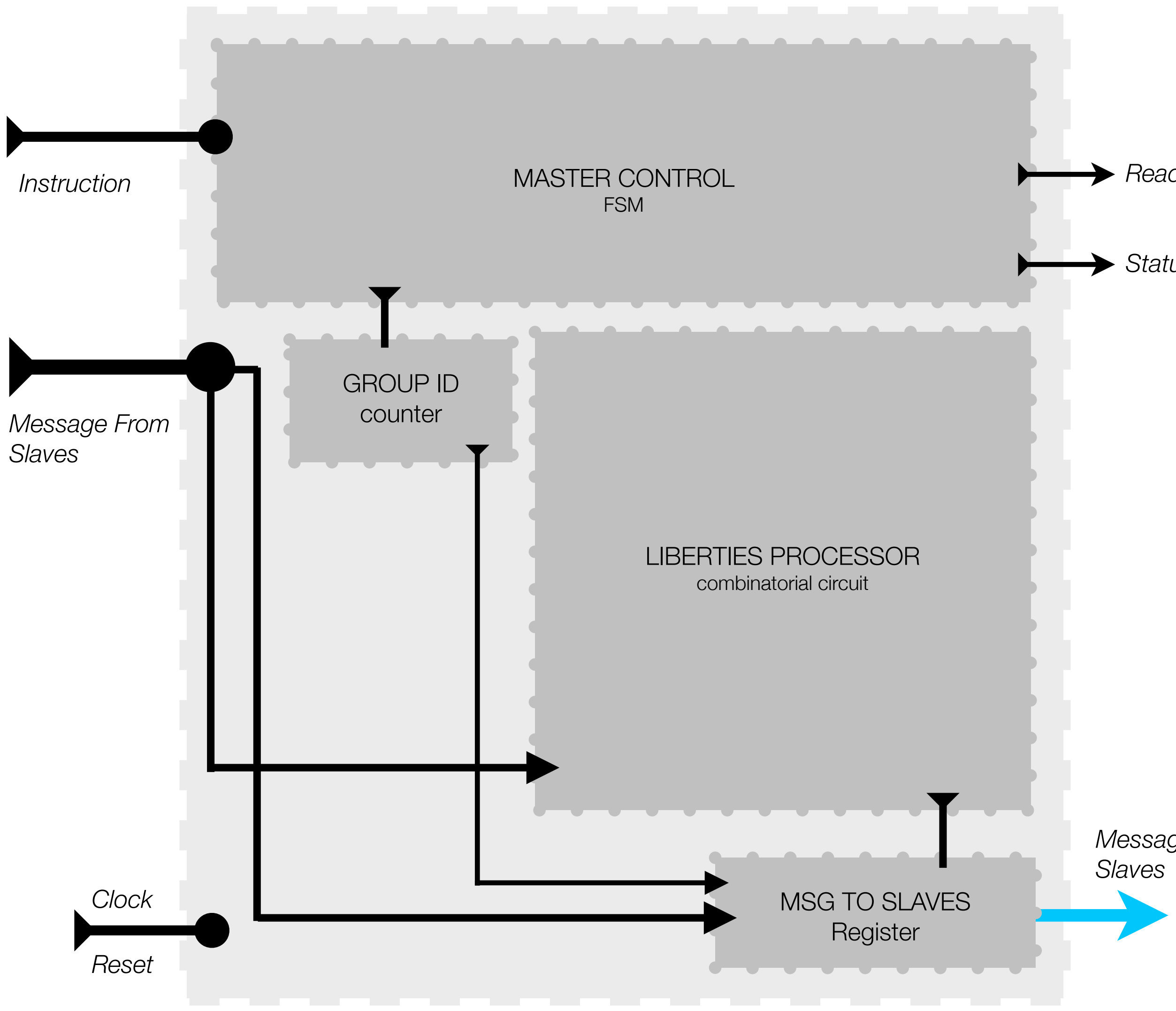
In chapter 3 we saw that the longest operation as far as the number of messages that are demanded for board settling is concerned, is a move causing some merges along with liberty reductions in neighbouring opponent groups. In the worst case where we play a move which would connect 4 separate groups of our color we would have to broadcast 16 messages, 4 for merging, 4 for one liberty reduction due to a new touch, 4 messages to further change the liberties of all groups to the accumulated liberties of the new big group and finally 4 messages for each merge and liberty modification for the new stone itself. This cycle of 16 messages for the completion of a new move operation effectively results in a worst case performance which is sixteen times lower than the one assumed previously.

Computer architecture design has for a long time debated the subject of RISC versus CISC architectures. RISC proponents have been proposing simple efficient designs which base their operation on an orthogonal, simple and compact instruction sets. CISC proponents, on the other hand have been proposing large instructions sets which provide more high level instructions performing operations such as matrix multiplications in one instruction which would in a RISC architecture take multiple instructions to complete. No choice comes without a cost and RISC architectures depend on a fast clock speed to operate efficiently while CISC architectures demand complex logic parts and wide buses with obvious maintainability and manufacturing issues.

The instruction set on which the slaves has been based till now has a clear RISC taste in it, which for our needs and for the time being has been efficient, clear and complete, but now it seems as the worst design choice combined with an FPGA for hardware fabric, a technology which never boasted the fastest clock cycles around. As explained, FPGAs succeed when many things happen in parallel, while RISC architectures are tuned for extremely fast sequential evaluation of small instructions. Could we possibly send more than one message at a time?

The concurrency dogma which we have cultivated, promptly asks us to search for shared resources if we want to parallelize things. If messages sent to a neighbouring group are totally independent of the messages sent to another, then we could pack as many messages as we want in one big message and send this new message to all slaves, effectively cutting the amount of cycles needed to send each message alone. Moreover, compared to the size of the message sent to the master CPU, the size of those received was minimal, and such a waste of bus bandwidth has been puzzling me for some days. The only drawback would possibly be the marginally increased complexity of the message decoding circuit in each slave. Intuitively I don't expect this to pose any serious space problem.

In 1989 Hewlett Packard came up with something similar. HP determined that RISC architectures were approaching a limit at one instruction per cycle. HP researchers investigated a new architecture, later named Explicitly Parallel Instruction Computing or EPIC, that allows the processor to execute multiple instructions in each clock cycle exactly as we would like to do. EPIC implements a form of Very Long Instruction Word or VLIW architecture, in which a single instruction word contains multiple instructions. This architecture would demand a very clever compiler, which would try to take advantage of all possible parallelism in the source code and pack the instructions accordingly, which initially created skepticism on the performance of the architecture. At the same time though, processor design was simplified considerably by eliminating the need for runtime instruction scheduling circuitry. The processor embodying these ideas was codenamed Itanium and was co-developed with Intel. The enormous hype and expectations that these companies cultivated to the market around their new architecture combined with their inability to develop for many years a capable compiler to support it, lead to a design which remains till today one of the biggest fiascos of the computer processor industry. Still, as it seems, those ideas remain totally fresh and applicable.



NOTES:

The Control circuit is a common Moore finite state machine with 4 states and is implemented in a straightforward way by the Xilinx compiler. The Liberties Processor computes the next total liberties for the played move and the adjacent groups which merge with it, if any. This is a combinatorial circuit consisting of 4 stages of comparators adders subtractors and multiplexers, which decide which one of the 256 possible configurations is true for our case (four neighbours and 4 possible colors for each one) and accordingly compute the correct group liberties.

We ask ourselves the question: "*Is there any interdependence between the messages sent to the four neighbours? If yes, can we eliminate it?*". In our complex case of the four merges described previously, there is one interdependence indeed, imposing a strict sequentiality on the broadcasted messages. Each time a new merging takes place, we have to compute the new accumulated liberties and propagate them to the old groups. For example, let *L1* be the liberties of the new played move with group number 1, equal initially to 4. Suppose that we examine the neighbours in a clockwise fashion, and the northern group has *L2* liberties and number 2. After the first merging, group 2 becomes 1 and the accumulated liberties are *(L2 - 1) + (L1 - 1)* removing one liberty from each group's liberties due to the touching point. When we try to merge with our eastern group let it be group number 3 with liberties *L3*, the accumulated liberties now have to take into consideration the result of the previous merging. This dependence makes clear that if all merging sequences reached the four neighbours at the same time then the resulting big group would have indeed the same identification number but each group previous to the merging would maintain its own idea of what the accumulated liberties are. If we could compute the total liberties of the final group after all the merges beforehand, then we would be able to broadcast a message with four group ids, instructing them all to adopt these new group liberties and afterwards change their group id to a new common to all groups id representing the merged group. And this is in fact what happens. Moreover, this complex computation has no other place than the master CPU introduced previously to take on all the processing needed for message generation. To cover all other possible neighbour cases, we can include the color of the new move in this big message, so that neighbouring groups of the opposite color can simply reduce their liberties by one due to the new touching point, ignoring the total liberties in the message. Maybe now, it becomes more obvious to you, reader, why such a master (*processing unit*) slave (*state storage*) proves invaluable after this new introduction of complex computations.

For our current needs the only message reaching our completed hardware board design is a new move or an undo move message, that is a new tuple of X, Y coordinates and, in the case of new move, color, too. The dialog between the entities shown in the previous chapter is now reinterpreted under the prism of the master-slave architecture. Only the messages exchanged are shown as the tables' contents remain the same as before, as the tables present the slaves' registers contents and moving the processing to the master CPU does not alter them.

Finally, as far as the design of the bus is concerned, we have to examine the capabilities of the underlying hardware for its implementation strategy. The most straightforward design for a bi-directional bus would be to implement it as a simple collection of wires connecting all the wanted entities and let the entities decide when someone wants to write something. This might resemble to a software programmer a process scheduling scheme, in which each process does its job and after some fixed quantum of time willingly passes the CPU to the next process. As we have clarified many times till now, the entity which corresponds to the current move is the only active slave, sending any messages through the bus and communicating with the master CPU. The rest of the slaves after realizing that the message just received does not affect them, along with the master CPU, can as well set their output message port connected to the bus to a high impedance state, similarly to a tri-state buffer state. This way only the active slave's output message will be transmitted through the bus as all the others do not affect it. Unfortunately, an FPGA, at least as of this Thesis' writing, cannot support this tri-state behaviour. To emulate such a design I decided to connect all the slaves' output ports toward the master CPU into a huge OR gate, and have the slaves output zeros instead of the high-impedance value. The result is the same as before; the active slave's message is transmitted as it should through the bus to the master CPU. At the same time the bus became directional and a separate bus was employed to transfer messages from the master CPU to the slaves. Other solutions, such as connecting all the outputs of the slaves on a huge multiplexer and let the master CPU decide, which one to select are clearly less effective both space and speed-wise.

The design of the bus, is a subject which has to be further researched by you, reader, as this OR gate with the 361*message_size fan in size is a major obstacle in reducing the size of larger board sizes such as the standard 19x19 Go board. Please consider every possible solution, even the possibility of its elimination, although this implies that the master-slave architecture has to be revised as well, which would lead to a new confrontation with the problems described in the previous analysis we went through. An implementation of this architecture in other hardware design technologies such as ASIC would enable you to use the described tri-state behaviour of the slaves' output ports.

These final remarks conclude the hardware translation of a Go board design, which began with the simulator of chapter 3. The following and last section will present the actual circuit analysis, as it was produced by Xilinx's tools for the different stages of our design and for each different sub-module.

5 . 2 Move Generating module

Having considered throughout this work the move generating module as present, this is the right point to describe its inner workings as they were visualized along with this hardware board and its implementation is the natural continuation of this work.

The MG module regardless of the algorithm used to do the thinking itself is expected to send messages conforming to the following structure:

opcode	Y coordinate	X coordinate	Color

The only opcodes needed for a Monte-Carlo schema are a "*new move*" and an "*undo*" opcode. To extend this instruction set one has to modify the instruction decoding circuit of the master CPU, which further controls the messages sent to the slaves. The source code of this Thesis is the same important as this text, if not more, and an inseparable part of it and in there you can find all the details on the devised architecture of this design including the opcodes and the structure of all exchanged messages.

Since the hardware board expects to receive both new moves and undo operations, the MG module has to know when to produce the ones or the others. Knowing when to produce an undo move is equivalent to knowing when a group becomes captured. One might argue at this point that this sort of information should be kept internal to the board and never get exposed to the outside world; that is to say, it is a matter of the board to know when a group is captured and to remove it.

This choice was made deliberately for two reasons:
‣ Every non-trivial move generator will have to know when a string of stones becomes captured. This is a knowledge that the board can provide in constant time. The most trivial implementation of a move generator would be one that is popping new moves from a random permutation of a stack of all possible positions. Even this most trivial implementation, has to know when a stone is captured so that it can push the freed position back into the stack of possible next moves, or else these captured positions are considered as occupied and won't be played again.
‣ By introducing the move undoing operation, matches can be played back and forth. By holding a board state as a starting point, one could play a match till the end, gather statistics on the winning rate of the starting move and then backtrack to another board state. This is a beautiful reinterpretation of tree searching.

As a final note, this board design implements an EndGame output to signify that only stones and eyes exist on the board and thus the match has ended. This has been presented more as an application of the master slave architecture and as a reader's exercise, than as a necessary part of the design. Since the MG module will hold a list of all possible next moves, when the list has emptied we know that the match has come to an end. In spite of this fact, the EndGame output is left in the hardware description of the board (*the code provides all needed comments to remove it*) as it occupies only 2% percent more space for the 9x9 board configuration.

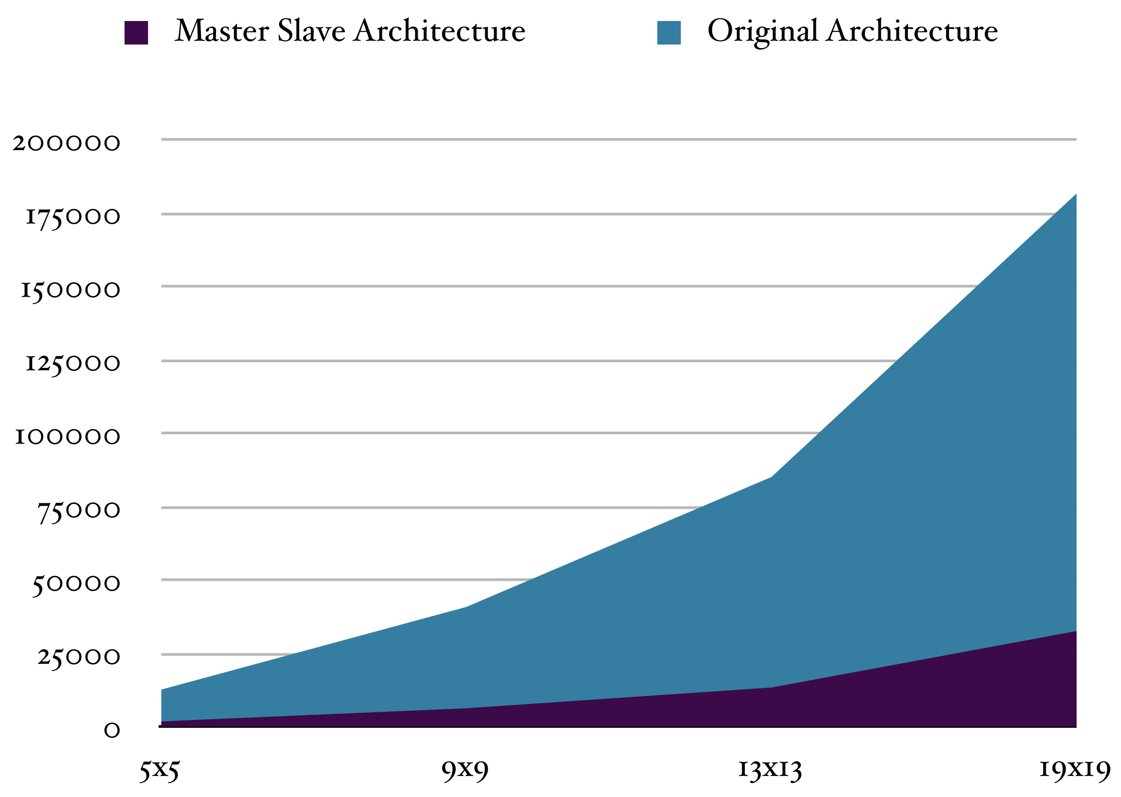
5.3 Circuit Evaluation

The following table presents the space utilization and timing results for the circuit's critical path for the circuit that Xilinx 10.1 produced for the Virtex 5 LX110T FPGA with speed grade -1, which our laboratory owns.

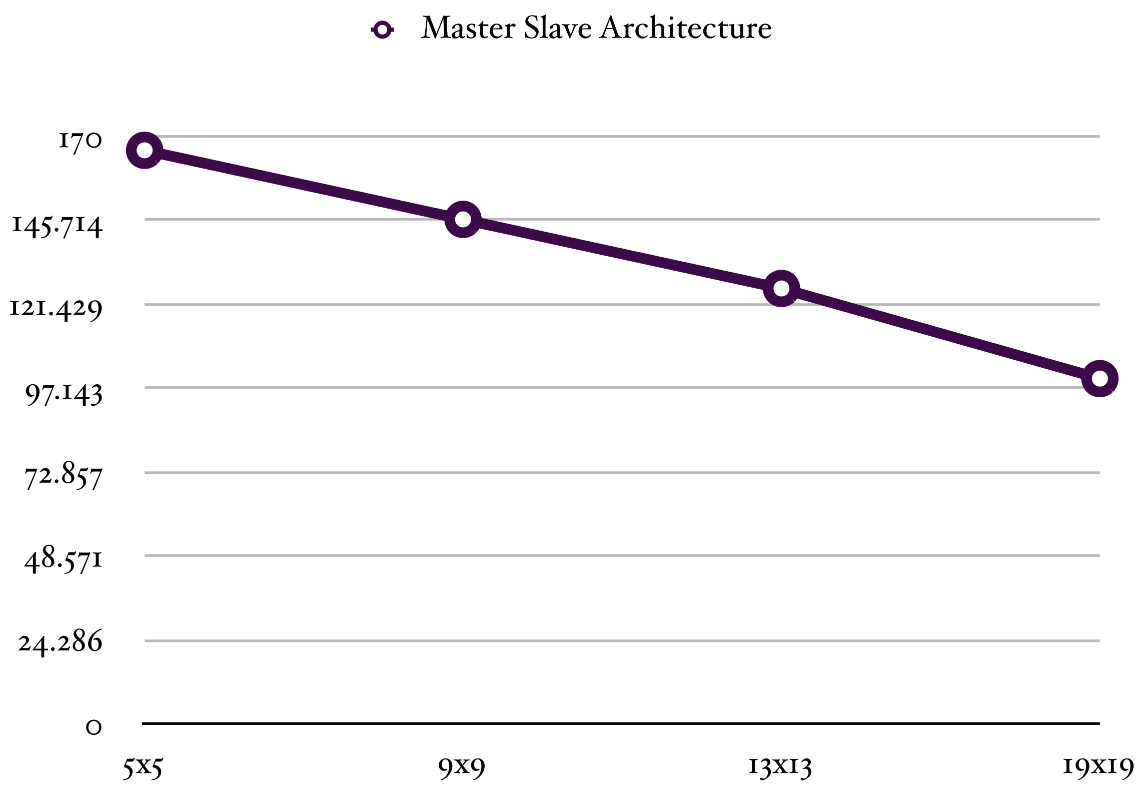
Architecture	**Space**	**Space percentage**	**Speed Mhz *clock***	**Speed Mhz *effective***
Original 9x9	40743/69120	58%	260	260
Original 19x19	181583/69120	262%	260	260
Master slave 9x9	6213/69120	9%	146	73
Master slave 19x19	32523/69120	47%	100	50

To get a better understanding of how the space of the design is distributed, the table below presents statistics for the final Master Slave architecture according to board size. As expected, the space utilization increases exponentially:

Board Size/ element	**5x5**	**9x9**	**13x13**	**19x19**
registers	251	1009	2239	5463
adders	14	14	14	14
addsubs	26	82	170	362
subtractors	13	13	13	13
xor gates	29	85	173	365
comparators	125	405	845	1805
priority encoders	2	0	0	0
counters	1	1	1	1



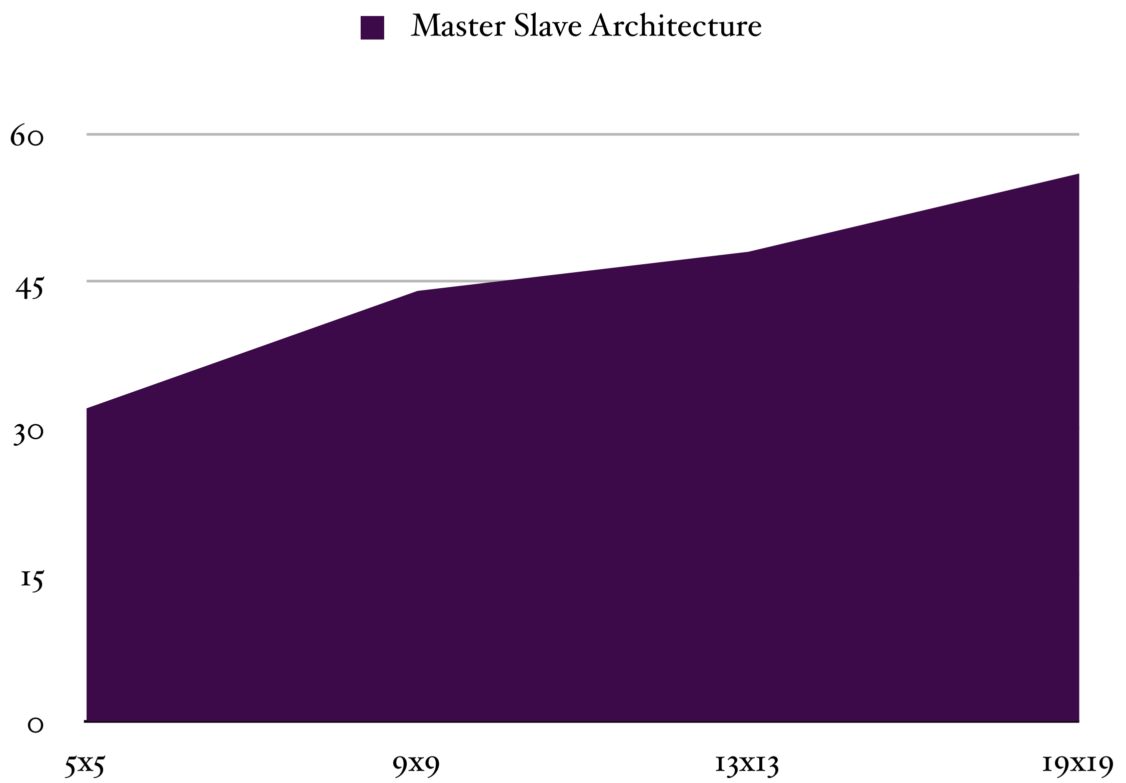
Space Consumption in LUTs



Circuit Speed in MHz

To compare the speed of the circuit with a software implementation I chose the latest version of GNUGo as it is one of the most popular software implementations and most important, open source and thoroughly documented. I searched for that piece of code which plays a new move and caters for all board update steps such as liberty reduction of opponent groups, removal of captured groups and extension of groups through merging. This function is called *play_move(...)* and exists in the *board.c* file of the source code. Two time samples were taken, in the beginning and the end of the function's body.

The timing revealed a minimum delay of 198μsecs and a maximum of 380μsecs. GNUGo's version is 3.8 , is compiled by the 4th version of the GCC compiler and runs on a Intel Core 2 DUO Macbook at 2.1 Ghz with 4Gb of RAM. The average delay was 220μsecs with minimum variation, thus independent of the captured group's size or the merging groups' sizes. This is expected as GNUGo is a mature implementation, relying on efficient bookkeeping of the board's state which makes these basic board operations run in constant time. This means, that the timing of this function, excludes all the time spent in bookkeeping and updating of the board's internal data structures and is therefore a minimum. Even then, with a minimum period of 220μsecs, GNUGo has the ability in my test system to play $1/220*10^{-6}$ moves per second or around 4500 moves per second. On the other side, our hardware architecture can play effectively 50 million moves per second on the lowest speed grade of -1 or 80 million moves per second for a speed grade of -3 for the 19x19 board size. Smaller board sizes offer increased speed. This is a side-effect of the signal transfer delays in larger FPGA designs. The following diagram presents the size of the bus which connects the slaves to the master CPU in relation to the board's size. FPGAs are very sensitive to long wiring and in the case of a 19x19 Go board, there is a 56 bit bus traveling among 361 slave CPUs connecting them into a 361*56 bit to 56 bit AND gate. This dense interconnection is the major cause behind the decline of the board's performance as the size of it increases.



Master Slave interconnection Bus Width in Bits

A first important note is that a captured group in GNUGo is removed in one move, whereas in our implementation we need as many undo moves as the size of the group. If we assume that during the average game on the standard 19x19 board size, no more than 30 stones are ever removed (*in a professional game, a capture of over 10 stones is already considered a loss*), then the true speed of the hardware circuit is not very far from this theoretic.

A second important note is that what is measured here is the time spent to update the Go board after a new move is played. The speed with which the board can be updated by new moves is the ultimate upper bound to the speed with which new moves can be sent to the board and therefore is an absolute metric of the highest possible performance of a computer Go player. I expect the rate with which new moves will be produced by the move generating module for a simple monte carlo player implemented in hardware, to be quite close to this board's basic operations' speed.

As a final observation, despite the fact that the architecture was built from ground up to guarantee constant complexity in all board operations, independent of the board's size, the performance of the circuit declines as the board becomes larger. One has to keep track all the time of the soil on which one builds.

FUTURE WORK

A Natural Continuation

No scientific work is usually left to exist on its own. The words continuity and progress are considered the holy grail which God gave to man, a grail to shine every day so that he can see his self-indulgent face reflected on it. The hive mind, that of every "*field*" including the scientific is at most uninteresting to most people similarly to how in paradise all people play harp even though in this Earth we live, no more than 5% of the people play western classical music and less than 3% of them play the harp. But man still considers all scientific endeavor much more than a simple rectification of every day's thinking, which in fact is. Having lived for quiet some time in this scientific world I could see that most of the people go after the Truth behind science without caring to find or even invent their own truth in it. And part of the problem is the Babel tower of this Continuity in everything scientific. A tower which can most prominently be seen today in Wikipedia.

The problem is in the way the Wikipedia has come to be regarded and used; how it has been elevated to such importance very quickly. And that is part of the larger pattern of the appeal of a new online collectivism that is nothing less than a resurgence of the idea that the collective is all-wise, that it is desirable to have influence concentrated in a bottleneck that can channel the collective with the most verity and force. This is different from representative democracy, or meritocracy. This idea has had dreadful consequences when thrust upon us from the extreme Right or the extreme Left in various historical periods. The fact that it's now being re-introduced today by prominent technologists and futurists, people who in many cases I know and like, doesn't make it any less dangerous.

For instance, most of the technical or scientific information that is in the Wikipedia was already on the Web before the Wikipedia was started. You could always use Google or other search services to find information about items that are now wikified. In some cases I have noticed specific texts get cloned from original sites at universities or labs onto wiki pages. And when that happens, each text loses part of its value. Since search engines are now more likely to point you to the wikified versions, the Web has lost some of its flavor in casual use.

When you see the context in which something was written and you know who the author was beyond just a name, you learn so much more than when you find the same text placed in the anonymous, faux-authoritative, anti-contextual brew of the Wikipedia. The question isn't just one of authentication and accountability, though those are important, but something more subtle. A voice should be sensed as a whole. You have to have a chance to sense personality in order for language to have its full meaning. Personal Web pages do that, as do journals and books. Even Britannica has an editorial voice, which some people have criticized as being vaguely too "*Dead White Men*".

The Wikipedia is far from being the only online fetish site for foolish collectivism. There's a frantic race taking place online to become the most "Meta" site, to be the highest level aggregator, subsuming the identity of all other sites. In the last year or two the trend has been to remove the scent of people, so as to come as close as possible to simulating the appearance of content emerging out of the Web as if it were speaking to us as a supernatural oracle. This is where the use of the Internet crosses the line into delusion.

The beauty of the Internet is that it connects people. The value is in the other people. If we start to believe that the Internet itself is an entity that has something to say, we're devaluing those people and making ourselves into idiots. The collective is good at solving problems which demand results that can be evaluated by uncontroversial performance parameters, but it is bad when taste and judgment matter.

This Thesis began with an introduction mostly of myself and less of this text, was deliberately written in the first singular person and had as a goal to have as few references as possible. I prompt you reader to discover and doubt this text, yourself and others not only through knowledge. Knowledge is limited and static by nature and the only operations one can do with knowledge are comparison, acceptance or dismissal. Find yourself, your interests and your place in this world through clear and free observation, unclouded by previous knowledge and prejudice which is a form of knowledge itself, remembering that freedom of thought is not a goal but a prerequisite.

TABLE OF REFERENCES

Kaoru Iwamoto (March 1977)
"Go for Beginners" - Pantheon Books, New York, ISBN: 0394733312

Bruno Bouzy (April 2007)
"Old fashioned computer Go vs Monte Carlo Go" - Paris Descartes University, France

Bruno Bouzy, Tristan Cazenave (April 2007)
"Computer Go: an AI Oriented Survey" - Université Paris 5, UFR de mathématiques et d'informatique

Marc Boule & Zeljko Zilic (2002)
"An FPGA Based Move Generator for the Game of Chess" - McGill University, Montreal, Canada

Shinichi Sei & Toshiaki Kawashima (1998)
"Memory-Based Approach in Go-Program "KATSUNARI" " - Fujitsu Social Science Laboratory Ltd.

Bernd Bruegmann (1993)
"Monte Carlo Go" - Max-Planck Institute of Physics

Remi Coulom (2006)
"Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search" - LIFL, SequeL, INRIA, Futurs, Universite Charles de Gaulle, Lille France

JOE ARMSTRONG (2007)
"Pragmatic Programming in Erlang" - Pragmatic Bookshelf, ISBN: 978-1-9343560-0-5

SENSEI'S LIBRARY
http://senseis.xmp.net Largest Online Go portal

L. KOSCIS & C. SZEPESVARI (2006)
"Bandit based monte carlo planning" - Computer and Automation Research Institute of the Hungarian Academy of Sciences

PETER J. ASHENDEN (1995)
"The Designer's Guide to VHDL" - Morgan Kaufmann Publishers, ISBN: 1558602704

GNU GO (1989 - 2009)
http://www.gnu.org/software/gnugo/

ROBERT PIRSIG (1974)
"Zen and the Art of Motorcycle Maintainance" - New York: Quill 25th Anniversary Edition ISBN: 0688171664