An FPGA Move Generator for the Game of Chess

Marc Boulé

Department of Electrical and Computer Engineering
McGill University, Montreal

A Thesis submitted to the faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of Master of Engineering

August, 2002

Your file Votre référence
ISBN: 0-612-85881-2
Our file Notre référence
ISBN: 0-612-85881-2

Humans have taught computers to execute complex calculations, to control manufacturing plants and even to send rockets into space. But will a computer ever be able to marvel at the beauty of a rose, appreciate the pleasant sound of a symphony or even understand the most basic of human emotions?

—The Outer Limits,
Mind Over Matter

Acknowledgements

I would like to thank my supervisor Zeljko Zilic. His guidance and patience has been greatly appreciated during my two years at McGill. He is always willing to review papers and to send his students to numerous conferences. He is particularly adept at suggesting relevant and interesting research topics as well as accepting students' ideas. For these and many more reasons, I do not hesitate to spend the next three years as a student of his.

Monty Newborn has graciously accepted to co–supervise this work. His knowledge in computer chess is well documented and I thank him for participating in my project. M. Newborn will also be an indispensable advisor for my up–coming doctorate in hardware–accelerated automated theorem proving.

An unexpected contribution to my project was made by a recent acquaintance from www.talkchess.com. Keith Evans has spent numerous hours reading my work and sharing his thoughts through many email discussions.

I would also like to thank the Natural Sciences and Engineering Research Council of Canada for the financial support. This has allowed me to select a research topic regardless of financial or commercial implications.

J'aimerais remercier André St–Amand pour son aide lors de la recherche bibliographique. C'est grâce à lui et les nombreuses discussions sur la programmation des échecs que j'ai choisi ce sujet de maîtrise. Ses nombreuses analyses de parties démontrant les faiblesses de mon programme ont été une source de motivation indispensable.

J'aimerais également remercier Normand Leclerc d'avoir créé les images des piecès et de l'échiquier qui ont servi à l'interface graphique de MBChess. Merci à Patrick Cardinal pour son aide avec LaTeX et à André Boulé pour m'avoir suggéré d'utiliser les "named pipe".

Abstract

This thesis details the use of a programmable logic device to increase the playing strength of a chess program. The time–consuming task of generating chess moves is relegated to hardware in order to increase the processing speed of the search algorithm. A simpler inter–square connection protocol reduces the number of wires between chess squares, when compared to the DEEP BLUE design. With this interconnection scheme, special chess moves are easily resolved. Furthermore, dynamically programmable arbiters are introduced for optimal move ordering. Arbiter centrality is also shown to improve move ordering, thereby creating smaller search trees. The move generator is designed to allow the integration of crucial move ordering heuristics. With its new hardware move generator, the chess program's playing ability is noticeably improved.

Résumé

L'objectif de ce mémoire est d'accroître la force d'un programme d'échecs en concevant un générateur de coups matériel. Pour atteindre cet objectif, des circuits numériques (hardware) sont utilisés pour augmenter la vitesse de calcul de l'algorithme de recherche. Le patron d'interconnexions des cases requiert moins de signaux comparativement à DEEP BLUE, tout en permettant une gestion simple des coups spéciaux (prise en–passant, roque, etc.). L'ordonnancement des coups est amélioré grâce à l'introduction d'arbitreurs programmables. La centralisation des arbitreurs contribue aussi à améliorer l'ordonnancement des coups, ce qui réduit la taille des arbres de recherche. Le générateur de coups permet l'intégration de plusieurs heuristiques cruciales. Après avoir remplacé la version software du générateur de coups par une version hardware, la force du programme d'échecs est nettement améliorée.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

For centuries, humanity has been fascinated by the game of chess. The mere fact that chess has been around for so long is a testament to its mystique. A simple set of pieces being maneuvered on an 8×8 array of squares can, for some, become a lifetime vocation. Throughout its history, different cultures have had variations on the rules and pieces of chess. Today's commonly accepted rules were first proposed in 1851 by Howard Staunton. These rules, along with many other aspects of chess, are governed by the *Fédération Internationale des Échecs* which was created in the 1920's.

With the technological revolution taking place during the middle of the twentieth century, it was inevitable that chess and computer technology would become intertwined. However, the idea of a chess–playing machine was put into practice during the early 1800s. *The Great Chess Automaton* was a large box with gears and mechanical parts inside. The audience was made to believe that this machine could play a game of chess. The only catch was that a human operator was hidden inside the box working the machinery. Real chess–playing machines started appearing in the 1960s with the introduction of the microprocessor. However, the key tree–searching algorithm to be used was introduced in 1950 by Claude Shannon. In *Programming a Computer for Playing Chess* [63], Shannon explained his strategies for tree searching.

For search–based algorithms, computing the best move for a given position involves trying different combinations of moves for both sides up to a certain search depth. In a search tree representation, a node is a chess position and a branch is a move. For simplicity in this thesis, "move" will be used in place of the proper chess term: "half–move". The term *ply* is often used to denote a level in the search tree. When a node is not expanded, it is referred to as a terminal node. To attempt to predict the expected value of the large sub–tree continuing beyond that point, an evaluation function assigns a score to each terminal node. When these scores are backed–up to the root node, we know which move leads to the greatest gains. The complete tree cannot be explored because of the large branching factor and the exponential nature of the tree.

Thus, a computer program that traverses a search tree of possible moves requires a function that can output the chess moves for any node reached during the search. In this thesis, custom digital circuits will be shown to generate moves faster than a software move generator. A faster move generator yields a faster progression through the search tree. Given a fixed amount of time, a deeper search tree improves the quality of the proposed best move.

In this thesis, the hardware move generator was designed, tested and integrated to a fully–functional chess program. The hardware move generator is faster than the software move generator and thus increases the program's playing strength by 150 to 210 chess rating points, depending on the metric used (*absolute* versus *relative* ratings respectively). The program fairs well against players rated below the *expert* skill level. Subsequent improvements necessary for master or even grandmaster performance are stated in Sections 5.5 and 6.2.

1.1 Move Generator History

In 1977, the BELLE chess system was entered in the World Computer Chess Championship in Toronto. Unbeknownst at the time, this marked the beginning of a 20–year period that culminated in the defeat of the reigning world chess champion. The BELLE chess program was the first to use custom digital circuits to increase its playing strength [12, 22]. The most time–consuming operations performed in a chess program are move generation [47] and positional evaluation. The first version of the hardware–augmented BELLE utilized a hardware move generator. Subsequently, other time consuming procedures such as alpha–beta search control, positional evaluation and a transposition table controller were implemented in silicon. With its new hardware, the 1980 version of BELLE was able to increase its search speed from 200 positions per second to 160 000. BELLE won the 1980 World Championship in Austria and was the dominating force in computer chess for many years.

The original BELLE hardware move generator was to serve as a starting point for a new, more powerful chess project. A doctoral student at Carnegie Mellon University, Feng–Hsiung Hsu, started CHIPTEST in 1985. A single–chip, 3–micron VLSI move generator was built and interfaced to a rudimentary chess program. After micro–coding the search engine in 1987, CHIPTEST–M was able to search 500 000 positions per second. At the 12th International Joint Conference on Artificial Intelligence in Australia in 1991, the project had increased in size to twenty–four custom processors and an IBM RS–6000 CPU. This system, named DEEP THOUGHT II, was able to search 6 to 8 million positions per second. Parallel processing of the alpha–beta search algorithm was now becoming commonplace. The computer that defeated Gary Kasparov in 1997, DEEP BLUE, had 30 IBM RS–6000 SP processors coupled to 480 chess chips [47]. With a computational speed of 200 million moves per second, full search depths of 12 plies were reached, with some extensions reaching 40 plies.

Another hardware chess project also originated from Carnegie Mellon University. In the early 1980s, a 64–chip move generator was designed and later became part of the HITECH chess machine. Each of the 64 chips was responsible for computing moves that land on a given square of the chessboard. The performance of the move generator was equivalent to BELLE's, but with better move ordering [23]. The HITECH machine went on to tie for first place in the 1986 World Computer Championship and the 1989 ACM North American Computer Championship. Aside from its ability to evaluate 175 000 positions per second, HITECH also introduced hardware pattern recognizers in its positional evaluation. HITECH attained the Senior Master title and was one of the strongest artificial chess players in the world from 1985 to 1988.

As its title indicates, the Chess–Oriented Processing System (CHEOPS) was also developed as a custom chess–playing machine. The core of CHEOPS is a custom built, 16–bit CPU designed specifically to execute chess instructions. The Chess Array Module, along with controlling logic, is responsible for generating chess moves. The CHEOPS system was integrated to the TECH II and MACHACK chess programs but no performance results are given [44]. CHEOPS was developed at the Massachusetts Institute of Technology during the late 1970s.

This project will focus on the design of a hardware move generator. One aspect common to the four previously mentioned chess machines is that the three principal components of a chess engine are implemented in hardware: positional evaluation, search control and the move generator[1]. Functional details of the move generators used in these designs will be given in Chapter 3.

Perhaps the most intriguing move generator of all is the human brain. When looking at a given chess position, experienced players are not even conscious of the move generation part of a chess "calculation". Pieces are shuffled around in one's imagination in an attempt to find a clever offensive or defensive plan. When Shannon introduced his chess algorithms, it was clear that a computer would be used to

[1]Positional evaluation is not done in hardware in CHEOPS.

generate moves and thus traverse a tree of possible outcomes. Ever since the first computer chess programs appeared, one constant in all of them has been the presence of the move generation function.

Even though humans and machines do not play chess in the same manner, the computer program must also have a function that generates the chess moves in any given position. In a software implementation, this is a fully sequential process. The board is scanned and when a piece belonging to the side to move is found, another loop is executed. In the case of a queen located in the middle of an empty board, a loop in each of the eight directions is necessary. This serial computation is not very efficient. Bit–boards have been used to try to profit from the implicit parallel computation of Boolean operations. Piece locations for all types of pieces are encoded as 64–bit numbers. In concert with pre–calculated attack tables, bit–wise operations are used to speed–up move generation. However, many instructions still need to be applied sequentially in order to obtain usable moves. The next step is therefore to use custom hardware that is specifically designed to generate chess moves.

When comparing a computer to a human, the human searches far fewer combinations of moves but benefits from a huge positional evaluator: his neural network. This is why novice players usually get severely beaten by chess programs. Their neural network (brain) is not trained to detect and evaluate chess positions and with their slow search processing, they are easily defeated. This is like trying to beat a processor in a multiplication contest. However, as the human neural network is trained, it starts to compete with the computer. As shown in the Kasparov versus DEEP BLUE matches, humans are capable of defeating massive searched based machines. However, the hardware used in chess machines continues to advance rapidly, whereas the hardware used in the human brain has remained essentially the same over generations.

1.2 Thesis Overview

The projects mentioned in the previous section were built using discrete logic chips and/or Application Specific Integrated Circuits (ASIC). In this project, the hardware move generator is built using a programmable logic device. An FPGA, or Field Programmable Gate Array, is an integrated circuit composed of an array of logic cells [70]. The routing lines have interconnect matrices to direct signals anywhere throughout the device. Hence, a design instance corresponds to a particular configuration of routing matrices and logic cells. A digital design is uploaded into the FPGA, at which point the desired circuit becomes functional. This process can be repeated at will and does not involve any subsequent material costs. Therefore, a circuit can be tested as the design steps progress, something not possible with ASICs. In this project, the FPGA move generator was used in real–world situations during the final design stages. This flexibility led to important modifications that facilitated integration to the chess program.

With increasing performance and higher gate–counts, FPGAs are slowly replacing ASICs or even custom ICs. In this project, the FPGA will be used to improve computer chess playing skills by increasing the processing speed of the alpha–beta search–tree algorithm used in computer chess. Specific FPGA architectural features will be used to improve the design of the chess move generator. It will be shown that an FPGA chess accelerator can successfully be used in an area traditionally reserved for ASICs, without the lengthy turnaround time, and at a fraction of the cost. Ease of re–programmability and on–chip RAM make FPGAs an ideal target for this application. The acceleration of combinational, search–based algorithms is not restricted to chess and can be applied in many different situations. One example of this is hardware–accelerated Boolean satisfiability.

The proposed FPGA move generator is integrated to MBChess, the author's chess program [15]. This program was developed prior to the design of the FPGA move generator and is based on commonly accepted computer chess concepts. Presentation of how computer chess programs work and MBChess is given in Chapter 2. The reader wanting to know more about computer chess programming may consult Chapter 9 of *How Computers Play Chess* [49]. In Chapter 3, previous move generator designs are surveyed. The proposed FPGA move generator is presented in Chapter 4. Throughout this thesis, the name CodeBlue refers to the FPGA hardware move generator. MBChess by itself refers to the original software–only chess playing program and MBChess–CodeBlue refers to a modified version of MBChess that uses the CodeBlue hardware move generator. The key metric used to evaluate the performance of the design involves playing MBChess–CodeBlue against MBChess in order to determine the improvement in chess rating. Both programs will also play independently on the Internet in order to be able to compare their absolute ratings. These and other results are presented in Chapter 5.

Chapter 2

Background

Before describing hardware move generators, a presentation of the commonly used chess algorithms and heuristics is in order. It is crucial to understand in which context the hardware move generator is to be used before proceeding with the design. Furthermore, the hardware move generator presented in Chapter 4 is used in MBChess, the author's fully functional chess program. In Section 2.1, the core tree–searching algorithm is be presented. An overview of the positional evaluation function is given, as well as details concerning two important speed–increasing heuristics: the transposition table and the killer heuristic. Quiescence search and iterative deepening are also introduced in this section. The effect of heuristics on move ordering and move generators is explained in Section 2.2. Particularities pertaining to the MBChess program are detailed in Section 2.3. A presentation of the FPGA technology used to implement the CodeBlue hardware move generator is given in Section 2.4. To complete the chapter, a formula used to quantify chess rating differences is derived in Section 2.5.

2.1 Chess Algorithms and Heuristics

The three critical components of chess engines were mentioned in the Introduction. These are: move generation, positional evaluation and search control. A conventional[1] chess program or a hardware chess machine must implement these three components with either program code or digital circuits. Search control corresponds to the mechanism used to traverse the search tree of possible moves (alpha–beta algorithm of Section 2.1.1). Once a leaf node is reached, a score is assigned using a positional

[1]As opposed to a neural network chess engine.

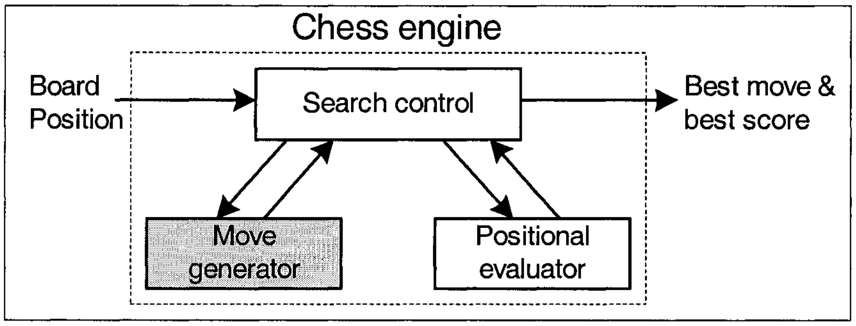


Figure 2.1: Chess engine block diagram. The move generator is implemented in hardware.

evaluation function such as the one overviewed in Section 2.1.2. At each node, the move generator is responsible for returning the next unexplored move. MBCHESS is a fully software chess program. In MBCHESS–CODEBLUE, the move generator is relegated to hardware for improved performance. The hardware move generator corresponds to the grey block in Figure 2.1.

In this section, the key components of a chess–playing program are explained. Other than the ones presented here, many other heuristics are used in modern chess programs such as Schaeffer's history heuristic [60] and null–move depth reduction [8, 31]. However, only the ones used in MBCHESS will be explained. An overview of the different components used in computer chess is introduced in Figure 2.2. Not shown in the figure is the positional evaluation performed at each quiescent node (Section 2.1.5). Other than iterative deepening, each of the topics presented in this section is visible in the figure.

2.1.1 Nega–Max Alpha–Beta Search Algorithm

The basic search mechanism used in chess programs is the alpha–beta depth–first search for the best move [40]. Variants and improvements to the basic alpha–beta algorithm are abundant, some of which are detailed in [3, 55, 56, 57, 59]. The parallel alpha–beta algorithm, as was used by the multiprocessor DEEP BLUE machine, has also been explored [46, 52]. Furthermore, a tree–splitting method based on neural networks has been developed in [41]. The proper explanation of the alpha–beta algorithm begins with the min–max search algorithm shown in Figure 2.3. In the example tree shown at the right of the figure, circle nodes have the player to move and square nodes have the opponent to move. Leaf nodes are always evaluated as seen from the root node player's point of view. The opponent picks moves leading to the minimum

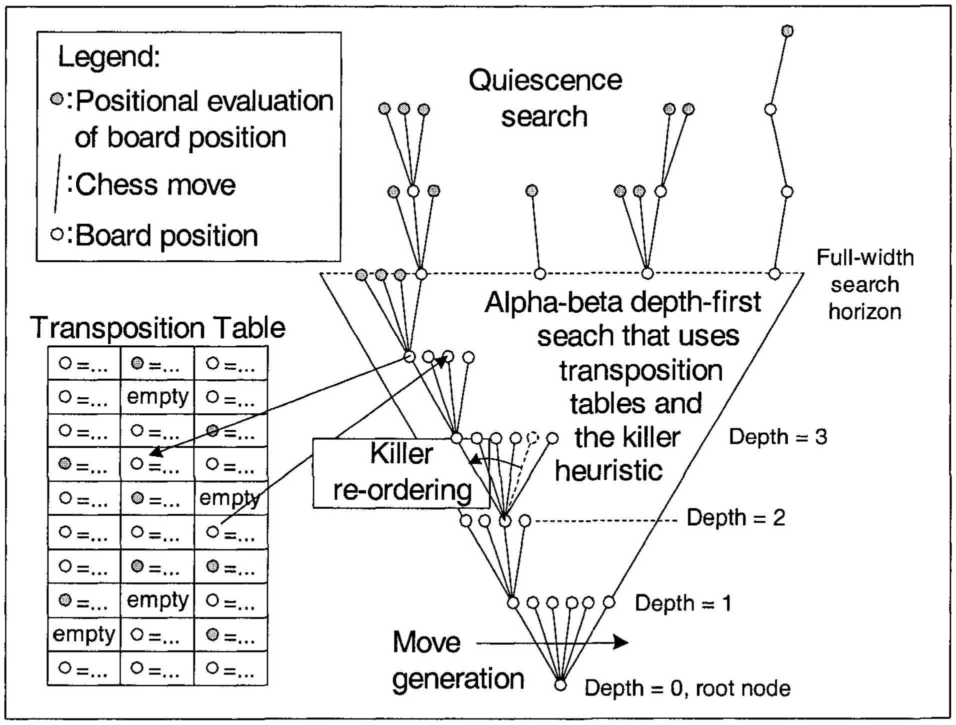


Figure 2.2: Computer chess components and search trees.

scores and the player selects moves leading to maximum scores (best position). The tree is searched in a depth–first manner, from top to bottom as indicated by the vertical arrow. After searching the min–max tree with this algorithm, the program knows which move is the best, as well as the corresponding best score.

The min–max algorithm can be re–written in a simpler form, where the distinction between the type of node (min or max) is not necessary. The algorithm negates the score returned by each subtree and always maximizes the score, hence the *nega-max* designation. In this algorithm, the position is evaluated as seen from the side to move at the terminal node. This formulation is much easier to work with when integrating move ordering heuristics. In this chapter, *player* refers to the side to move in the root position, and *opponent* refers to the adversary. In the search trees shown in this chapter, branches emerging from the root node correspond to moves by the chess program. Moves emerging from nodes at the first ply are moves from the opponent, and so forth. It should be clear that the search procedure plays out moves for both sides and only the resulting best–move is actually played on the chessboard by the program.

A pivotal improvement to the min–max algorithm was introduced in 1958 by Newell, Shaw and Simon [50]. Upon closer inspection of the nodes processed in a

```
FUNCTION: MinMax($depth$)
if $depth = max\_depth$ then
    return EvaluatePos($position$)
if max node then
    $best \leftarrow -\infty$
else
    $best \leftarrow \infty$
for $i \leftarrow 1$ to # moves in $position$ do
    make move $i$
    $score \leftarrow$MinMax($depth + 1$)
    unmake move $i$
    if max node then
        if $score > best$ then
            $best \leftarrow score$
    else
        if $score < best$ then
            $best \leftarrow score$
return $best$
```

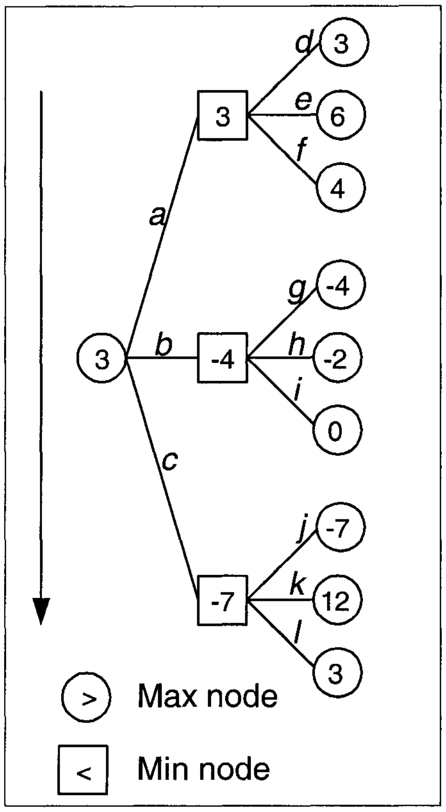


Figure 2.3: Min–max algorithm and example tree. The min–max algorithm reveals a best score of 3 and that the best move for the side to play is move a. The EvaluatePos function is from the root–player's point of view.

```
FUNCTION: NegaMax($depth$)
if $depth = max\_depth$ then
    return Evaluate($position$) {Evaluate from side–to–play's point of view}
$best \leftarrow -\infty$
for $i \leftarrow 1$ to # moves in $position$ do
    make move $i$
    $score \leftarrow -$ NegaMax($depth + 1$)
    unmake move $i$
    if $score > best$ then
        $best \leftarrow score$
return $best$
```

Figure 2.4: Nega–max algorithm, a simpler version of the min–max algorithm.

min–max search, certain branches (moves) can be disregarded without affecting the outcome of the search procedure. In Firure 2.3, the reader should verify that omitting to expand moves h, i, k and l is completely safe. This can be done not because we know that the scores shown are not returned but rather because whatever their value, the backed–up value at the root node is not affected. Another explanation of these cutoffs is that the player will never choose a continuation in which a move tried by the opponent leads to a position inferior to the worst move played by the opponent in a previously searched continuation. Thus, in the non–promising continuation, once a refutation is found, the remaining moves do not need to be expanded. These cutoffs can occur at many levels in the search tree. The closer these occur to the root node, the larger the savings. This pruning algorithm can be categorized as a branch–and–bound optimization technique. When a branch cannot affect or improve the solution to the problem, it is not expanded and the search space is reduced. In branch–and–bound algorithms, a bound is used to help decide whether a branch should be explored or not.

The search tree from Figure 2.3 was arranged to create the most cutoffs possible. Had the moves been searched in a different order, fewer or no nodes would have been removed. With one move ordering being better than another, what constitutes the best move ordering? Once more, using the example tree from Figure 2.3, it can be shown that if each node orders its moves from best to worst, the minimal alpha–beta tree will result. Because the search tree is traversed in a depth–first order, terminal scores are not known in advance when selecting a move at a node. Move ordering represents a much worked–on topic in computer chess and the quest for the minimal tree is ever ongoing. A commonly used heuristic involves searching capturing moves first.

The alpha–beta algorithm is shown in Figure 2.5 (adapted from [59]). A mathematical analysis of the alpha–beta algorithm is shown by Knuth in [40]. When compared to a min–max search tree with the same number of nodes, the minimal alpha–beta tree is twice as deep because of the cutoffs performed. A factor of two in search depth is very significant, given the exponential nature of the search tree. This sizeable reduction in average branching factor enables programs to search deeper thereby playing stronger chess. Consequently, the much needed move ordering places an additional requirement on the design of the hardware move generator. Another requirement is also deduced: the move generator should return moves individually so that no generated moves are wasted when cutoffs occur.

The condition "***if*** *score* $>$ *best*" in the nega–max and nega–max alpha–beta

```
1: FUNCTION: NEGAMAXALPHABETA($depth$,$\alpha$,$\beta$)
2: if $depth = max\_depth$ then
3:   return EVALUATE($position$)
4: $best \leftarrow -\infty$
5: for $i \leftarrow 1$ to # moves in $position$ do
6:   make move $i$
7:   $score \leftarrow -$ NEGAMAXALPHABETA($depth + 1$,$-\beta$,$-\max\{\alpha, best\}$)
8:   unmake move $i$
9:   if $score > best$ then
10:     $best \leftarrow score$
11:   if $best > \beta$ then
12:     return $best$
13: return $best$
```

Figure 2.5: Nega–max alpha–beta algorithm. The condition "***if*** $best > \beta$" is the mechanism used to perform cutoffs.

Table 2.1: Piece values used in MBCHESS.

	Pawn	**Knight**	**Bishop**	**Rook**	**Queen**
Value:	100	300	310	500	900

algorithms implies that among the best equal valued subtrees, the *first* subtree's value will be returned and also that the move leading to this subtree will be part of the best line of play. Conversely, if a $\geq$ was used, the last of the best equal subtrees would be kept as the best line of play. In the nega–max alpha–beta algorithm, a cutoff is also referred to as a beta cutoff because it is this bound that causes the search to prematurely backtrack. The algorithm used in MBCHESS is the NegaScout version of alpha–beta developed by Reinefeld [59]. The NegaScout alpha–beta algorithm will be presented in Section 2.1.3, simultaneously with the integration of transposition tables. The alpha–beta algorithm will be labeled ALPHA–BETA throughout the thesis.

2.1.2 Positional Evaluation Function

As mentioned in the Introduction, evaluation of terminal nodes is done in order to predict the value of the subtree continuing beyond the terminal node. This function (labeled POSITIONAL) is dominated by the material on the chessboard. To allow sufficient resolution without resorting to floating point calculations, the value of a pawn is set at 100. The piece values used in MBCHESS are outlined in Table 2.1. Since integer values are used, the smallest positional effect is one percent of a pawn.

Table 2.2: Some of the positional terms used in MBChess.

Situation	Value	Situation	Value
Doubled pawns	−8	Pawn on home row of column e	−16
Pawn on penultimate row	50	Pawn on row 4 or 5 of column e	6
Queen moved during opening	−8	Piece attacking e4, e5, d4 or d5	3
Knight on row 1 or 8	−4	King–side bishop on home square	−5
Square radiated by queen	1	Queen–side bishop on home square	−3
Square radiated by bishop	1	Vertical square radiated by rook	1
Forfeit king castle	−30	Castle king side with 2 of 3 pawns	12
Forfeit queen castle	−15	Castle queen side with 2 pawns	7

The positional evaluation used in MBChess is quite basic. Examples of positional terms used in the program are given in Table 2.2. The positional factors are summed and added to the material value. This final score is signed according to the side to move, as required by the nega–max form of the search algorithm. Furthermore, a random number between −2 and 2 is added to the returned score to ensure a certain randomness of play by the program. This is labeled the RANDOM property. With this feature, the chess program virtually never plays the same game twice. If the side to move has no legal moves and its king is not in check, a score of 0 is returned to indicate stalemate. If the side to move has no legal moves and its king is in check, a score of −30 000 is returned to indicate checkmate.

Positional evaluation functions are complex and time consuming to develop. The positional function used in MBChess is relatively simple. Many improvements could be made to increase the strength of play, most notable of which is the addition of king safety. More details concerning the creation of suitable positional evaluation functions can be found in [27, 42, 68], as well as in the workings of CHESS 4.5 [26] and Deep Blue [21].

2.1.3 Transposition Tables

Other than the stack space required for N plies, the alpha–beta algorithm requires no memory to operate. One important use of memory concerns transpositions: different sequences of play that result in the same position. For example, from the initial position, the sequence of moves e4–e5–d4 and d4–e5–e4 have the same resulting board position. The first time the position is reached in the search tree, the calculated score is stored in the transposition table (T–TABLE). The second time the same position is reached, no work needs to be done and the score is taken from the table.

Transpositions also occur at different depths in the search tree. As with beta cutoffs, the closer a successful table lookup occurs to the root node, the larger the savings.

The transposition table would not be very efficient if entire chess positions needed to be stored in the table. Furthermore, finding random positions in the table would be time consuming. To solve these two problems, the Zobrist algorithm is used [71]. The Zobrist algorithm involves converting the chess position into an x–bit hash–key. In MBCHESS, $x = 64$. Because the transposition table has less than 2^{64} entries, the n lower bits of the hash–key are used as the hash–index. Therefore, a transposition table has 2^n entries. To compute the hash–key, a 64–bit random number table is constructed. The table has a unique random number for each piece–square combination (12×64). Supplemental values also exist for castling rights, en passant captures and the side to move. The hash–key is obtained by an XOR of all piece–square and board state random values. The hash–key can be easily updated when a move is made. This simply involves an XOR with the source piece–square value and another XOR with the destination piece–square value. The Zobrist algorithm achieves almost perfect distribution throughout the hash table and can be used in many other fields where hashing is performed.

Because there are much more than 2^{64} possible chess positions, many chess positions can map to a single hash–key. Because of the relatively slow processing speed of chess programs, the probability of this causing a problem is *very* small. This is referred to as a hashing error. A hashing collision occurs (much more frequent) when two chess positions are mapped to the same hash–table entry. For example, with a hash–table consisting of 2^{20} entries, a program searching 100 000 positions per second would fill up the table in 10 seconds. When collisions occur, a replacement scheme determines which of the two positions contains the most important information. More information concerning replacement schemes can be found in [20]. Transposition tables are also explored in [18, 19, 45].

In MBCHESS, a hash entry consists of:

1. The 64–bit hash-key;
2. The score of the position;
3. The depth of the corresponding subtree;
4. The best move found if this is not a terminal node (TT–SUGG-MOVE);
5. A flag to indicate if the score is exact, upper–bound or lower–bound.

The best–move is used to influence move ordering when the stored information is not sufficient for a direct look–up. For example, if the position has been found in the table but the required depth is larger than what was searched when the position was stored in the table, the best move from the hash–entry will be searched first. For maximum alpha–beta efficiency, move generators must allow for this rearrangement in move ordering.

As indicated in Section 2.1.1, the NegaScout alpha–beta algorithm will now be presented (Figure 2.6). This algorithm is adapted[2] from [59] and includes the necessary instructions to use a transposition table. The call to the Evaluate function has been replaced by the quiescence search function, which will be explained in Section 2.1.5. The UpdateTT function is also shown in Figure 2.7.

2.1.4 The Killer Heuristic

As with the transposition table's suggested move, the killer heuristic is another important move ordering improvement (labeled KILLER). With this heuristic, moves that have caused beta cutoffs elsewhere in the search tree, at the same level, are searched first. The goal of the killer heuristic is to improve move ordering in order to reduce the number of positions searched in the alpha–beta algorithm. An example of a position where the killer heuristic is effective is shown in Figure 2.8 [27]. After most moves from black, white's fork is still valid (Nc7 in the diagram). It is therefore advantageous to remember the move Nc7 and retry it at other positions after each of black's move at the root node. For example, after black's a6 move, white eventually finds that Nc7 is a killer move. After each of black's pawn moves, white can successfully play the killer move. If this move is played first, the beta cutoff occurs right away and the resulting subtree is smaller. For the example position from Figure 2.8, MBChess–CodeBlue searches 30 849 nodes when executing a four–ply search (POSITIONAL and ALPHA–BETA activated). When the killer heuristic is added, this number is reduced to 18 042 nodes, a 41.5% reduction. The heuristic is not as effective in all chess positions. On average, its use can reduce search tree sizes by approximately 10% to 20%.

Killer moves can be implemented by two different methods. The first method, as shown in [48, 67], involves maintaining a killer slot for each ply. This array is external to the search procedure. In MBChess, killer moves are implemented differently. Each node maintains the killer move for its children nodes. Once a child node finds

[2]The "fail–soft refinement" from the referenced algorithm is omitted.

```
FUNCTION: NegaScoutAlphaBetaTT($depth$,$\alpha$,$\beta$)
if $position$ found in hash_table and depth of hash entry good enough then
    if hash_entry.flag = exact or
    (hash_entry.flag = lowerbound and hash_entry.score $> \beta$) or
    (hash_entry.flag = upperbound and hash_entry.score $< \alpha$) then
        return hash_entry.score
if $depth = max\_depth$ then
    $best \leftarrow$ QuiescenceSearch($depth$,$\alpha$,$\beta$,$NULL$)
    UpdateTT($best$,$\alpha$,$\beta$,NULL)
    return $best$
$best \leftarrow -\infty$
$n \leftarrow \beta$
for $i \leftarrow 1$ to # moves in $position$ do
    make move $i$
    $score \leftarrow -$ NegaScoutAlphaBetaTT($depth + 1$,$-n$,$-\max\{\alpha, best\}$)
    if $score > best$ then
        if $n = \beta$ then {if first move being tried}
            $best \leftarrow score$
        else
            $best \leftarrow -$ NegaScoutAlphaBetaTT($depth + 1$,$-\beta$,$-score$)
        $best\_move \leftarrow$move $i$
    unmake move $i$
    if $best > \beta$ then
        exit for loop {beta cutoff!}
    $n \leftarrow \max\{\alpha, best\} + 1$
UpdateTT($best$,$\alpha$,$\beta$,$best\_move$)
return $best$
```

Figure 2.6: NegaScout alpha–beta with transposition tables algorithm.

```
FUNCTION: UpdateTT($best$,$\alpha$,$\beta$,$best\_move$)
if $best > \beta$ then
    $flag \leftarrow lower\_bound$
else if $best < \alpha$ then
    $flag \leftarrow upper\_bound$
else
    $flag \leftarrow exact$
if hash_table[$position$] is empty or replacement is advantageous then
    WriteHashTable($best$,$flag$,$best\_move$)
```

Figure 2.7: UpdateTT function used in Figure 2.6.

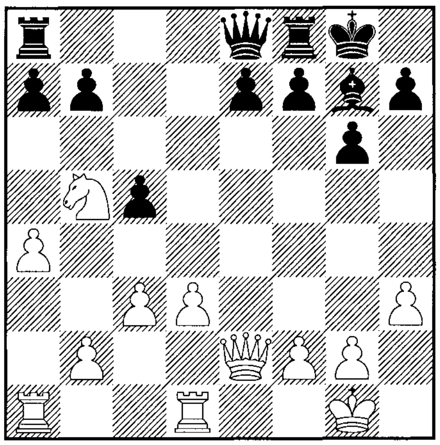


Figure 2.8: Killer heuristic example diagram, black to play. White's forking move (Nc7) is the killer move.

a move that causes a beta cutoff, this move is stored in the parent's killer slot. The child's siblings have access to the stored killer and can use it to improve their move ordering. The idea for constructing killers in this manner was obtained from *Green Light Chess* [25]. With this technique, multiple killer moves can be stored in different regions of the tree. However, rediscovering a recurring killer move can be costly. The killer technique used in MBChess was found to be slightly better than the first method mentioned, however, more experimentation is needed to confirm this. The killer suggested–move also represents a rearrangement in move ordering. Besides allowing a killer move to be executed first, the hardware move generator must also support verifying that a killer move is legal in a given position.

2.1.5 Quiescence Search

Quiescence search is a second level of tree searching that is executed when the full–width search horizon is reached. This second level search aims to continue to expand certain branches until a quiet position is reached (one where virtually no capturing moves exist). A simple but computationally expensive quiescence algorithm would involve searching all captures at a node and calling the quiescence search function recursively. At each node, the side to move is given the choice of making any capturing move or accepting the positional evaluation as is. Once the position is deemed stable, the positional evaluation function is called to assign a score to the leaf node. Quiescence search is an integral part of any successful chess program. A general quiescence search procedure is given in [9].

```
1: FUNCTION: QuiescenceSearch($depth$,$\alpha$,$\beta$,$square$)
2: $best \leftarrow$ Evaluate() {Capturing moves also generated}
3: if $best > \beta$ or no capturing moves found then
4:     return $best$
5: if $square \neq$ NULL then
6:     remove captures that do not land on $square$
7: for each capturing move $i$ do
8:     an alpha–beta search that calls
       QuiescenceSearch($depth$,$-\beta$,$-\max\{\alpha, best\}$,destination square of $i$)
9: return $best$
```

Figure 2.9: Quiescence search function used in Figure 2.6.

In MBChess, the quiescence search operates as follows. When calling the quiescence function from the main search function, all destination squares where the player to move can make a capturing move are retained. After making any capturing move to one of these squares (during the first ply of quiescence), the quiescence function is called again. This time, the destination square for the move that was just executed is used to restrict further iterations of quiescence to moves that capture on this square only. This quiescence function is particular to MBChess and has not been found in the chess literature. This quiescence system was developed to limit the large amount of time spent in quiescence search.

The capturing moves necessary for quiescence search are easily generated within the positional evaluation function. The positional evaluation function presented in Section 2.1.2 scans each piece in each possible direction in order to quantify the squares or pieces attacked. Capturing moves are easily identified with relatively low overhead. The quiescence function used in MBChess is shown in Figure 2.9. Transposition tables are not used in quiescence because the searches are restricted to capture searches and thus are not full–width alpha–beta searches.

A quiescence function that searches too many moves will reduce full–width search depth at the expense of deeper extensions. Conversely, a quiescence function that does not search capturing exchanges deep enough will prevent the evaluation function from accurately assigning a score to a leaf node. In the current version of MBChess, this quiescence scheme represents a good balance between search effort and search extension quality. In quiescence search, the move generator must be capable of generating capturing moves first. Another important property of a hardware move generator has therefore been stated.

Another selective searching procedure involves singular extensions [5, 6]. When

```
1: FUNCTION: ITERATIVEDEEPENING()
2: best ← NEGASCOUTALPHABETATT(1,$-\infty$,$\infty$)
3: best_move ← first move from best line of previous search.
4: $n \leftarrow 2$
5: loop
6:    best ← NEGASCOUTALPHABETATT($n$,$-\infty$,$\infty$)
7:    if search interrupted because no more time left then
8:       exit loop
9:    best_move ← first move from best line of previous search.
10:   $n \leftarrow n + 1$
11: return best_move
```

Figure 2.10: Iterative deepening algorithm. The NegaScout alpha–beta search function is shown in Figure 2.6.

one or more moves are found to be significantly better than the other moves at a node, they are searched one ply deeper. Singular extensions can occur at different levels in the tree, thereby producing selective searching in important regions of the search space. These types of extensions were used in the DEEP BLUE project and were found to play a key role. Selective extensions could be added to a future version of MBCHESS.

2.1.6 Iterative Deepening

The advantage of depth–first searching is that a small amount of memory is required. However, in the context of a timed chess game, selecting the proper search depth for a single search becomes very difficult. The size of the tree is not known and the average branching factor varies according to the stage of the game and the board position. To remedy these problems, iteratively deepening alpha–beta searches are called (labeled IT–DEEP). The apparent inefficiencies of repeating portions of the search tree are not as detrimental as they might seem. First, when the search of depth N is started, many positions from the "$N - 1$"–deep search are present in the transposition table and are used to improve move ordering as well as provide lookups, when possible. Secondly, because of the exponential nature of the tree, the search to depth N requires reasonably more time than the previous searches from 1 to $N - 1$.

When the allotted time for a move is elapsed, the search to depth N (in progress) is interrupted. The best–move found so far cannot be used because the search tree was not completely finished. Therefore, the best–move from the previous search to depth $N - 1$ is used.

Table 2.3: Move ordering in MBChess–CodeBlue (best–first).

Order	Type of move
1	Transposition table's suggested move
2	Killer heuristic's suggested move
3	Direct Checking moves
4	Discovered Checks
5	Capturing moves in MVV/MVA order (includes capturing promotions)
6	Non–capturing promotions
7	Non–capturing moves

From the iterative deepening algorithm shown in Figure 2.10, a simple optimization concerning the alpha and beta bounds becomes possible. Aspiration search, or windowing, involves tightening the alpha and beta values of the current iteration around the previous iteration's best score. A narrower alpha–beta window improves the efficiency of the alpha–beta algorithm. In MBChess, this was not found to improve search speed and was not implemented, however, more experimentation would be needed. When taking the windowing concept to the extreme, the Memory–Enhanced Test Driver (MTD) algorithm is obtained. The algorithm is based on zero–width searches that are repeated until the returned best score stabilizes. Aspiration search and the MTD algorithm are described in more detail in [67] and [55, 56, 57] respectively. With both of these algorithms, when the score returned from a search falls outside of the alpha and beta bounds, the position must be re–searched with different bounds.

2.2 Move Generators and Move Ordering

Move generators, whether software or hardware, must generate chess moves in a best–first order. The goal is to achieve minimal–sized alpha–beta search trees and thus search the same depth in faster time. In Section 2.1.1, the importance of good move ordering was emphasized. In this section, the ordering of moves and the effects of the T–TABLE and KILLER heuristics on move ordering are outlined.

An example of move ordering is given in Table 2.3. This ordering corresponds to the moves used by MBChess–CodeBlue during full–width searching (as opposed to quiescence search). The capturing moves are generated in order of most–valuable–victim / most–valuable–aggressor (MVV/MVA), contrary to the typical ordering of other hardware move generators. A large portion of the move generators shown in

Chapter 3 follow the most–valuable–victim / least–valuable–aggressor (MVV/LVA) ordering. The reason for this change in move–ordering is given in Section 5.2.

In quiescence search, the move ordering follows priorities 5 and 6 from Table 2.3. Priority 5 is modified to obtain the MVV/LVA ordering (the explanation for this is also given in Section 5.2). Selective extensions that follow checking moves could be added in the future.

In MBChess, move ordering is simpler than what is shown in Table 2.3. No distinction is made between direct or discovered checks. Furthermore, no distinction is made between types of aggressors, hence the MVV ordering for capturing moves. A large disadvantage of the software move generator is that *all* moves have to be generated for priorities 3 to 7. When a beta cutoff occurs, all unused moves have been uselessly generated. One of the advantages of the hardware move generator is that all moves are generated individually; no unused moves are wasted when a cutoff occurs.

In both programs, when the current position is found in the transposition table and the stored information does not cause an immediate score lookup, the best move stored in the table entry is searched first. Because the positions match (verified with the hash key), this move does not need to be verified and requires no move generation. If the returned score causes a beta cutoff, no moves have been generated.

When the killer heuristic indicates a suggested move to try, it must be validated in the current board position. In MBChess, only the moves for the killer piece are generated for this validation. If a cutoff occurs for the subtree corresponding to the killer move, no other moves have been generated. In MBChess–CodeBlue, only the moves that land on the killer move's destination square are first generated.

An important distinction must be made regarding the legality of chess moves. All the move generators described in this thesis generate *pseudo–legal* chess moves. The moving player's king could potentially be in check after making any one of the pseudo–legal moves. This simplifies the move generator design considerably. In order to expand *legal* moves during the search procedure, once a move has been made, it is verified for legality. If the moving player's king is in check after the move is made, it is not searched and the next move is examined. In this way, the moves not used because of a beta cutoff have not wasted computing power in order to be verified for legality.

Move ordering and heuristics represent a research topic on their own. More experimentation would be necessary to fully validate the move ordering used in MBChess–CodeBlue (Table 2.3). Furthermore, the design of the hardware move generator imposes certain limits on how move ordering could be re–arranged. For example, one could argue that discovered checks are better than direct checks. This would involve interchanging priorities 3 and 4 from the table, something not feasible given the actual design. For further reading, a thesis on move ordering can be found in [67].

2.3 MBChess Extras

In this section, miscellaneous details concerning the MBChess program are given. For this section, the term MBChess also refers to MBChess–CodeBlue because both programs are essentially identical, except for move generator issues. Here are a few supplemental details pertaining to MBChess:

1. The *internal–node counter* includes terminal nodes. Also included are nodes that lie on the full–width search horizon, before they are extended with the quiescence function. The *quiescent–node counter* includes quiescent nodes deeper or equal to the first level of quiescence. Node counts are used in Chapter 5 for benchmarking;

2. There is no opening book in MBChess. Use of an opening book is empirically beneficial, however, no time has been spent to implement this feature. We believe that a program should first be able to play well by itself before being injected with two hundred years of human expertise in openings;

3. MBChess can be instructed to spend x amount of seconds for each move. The *smart time* option prevents the next level of iterative deepening search to be undertaken when less than half of the allotted move–time remains. More complex time controls allowing the total game–time to be properly distributed would increase the program's performance;

4. Repetition draws are detected only in the first two plies of search. This prevents the program from directly playing a drawing move or prevents the opponent from directly playing a drawing move when this is advantageous. A much better repetition draw detector would involve hashing and would be performed at each node in the entire search tree;

5. Check extensions (CHECK–EXT) are performed to a maximum of two plies beyond the intended full–width search depth. The quiescence search is never called when the side–to–move's king is in check. When a check extension occurs,

the subtree rooted at the given node has its maximum–depth parameter incremented. Because of the full–width depth increase on the subtrees, a selective check extension would be more efficient;

6. Deep thinking was initially part of the MBChess program but was not completed for MBChess–CodeBlue. This option allows the program to search each of the opponent's moves while he/she is pondering their next move. A preliminary best–line of play, as well as the positions stored in the transposition table, can be used to improve move ordering for the computer's turn.

2.4 FPGA Architecture and Requirements

In this section, an overview of the device used to implement the CodeBlue move generator is presented. The Field Programmable Gate Array (FPGA) is a user–reconfigurable logic device. It can be used to implement custom–designed digital circuits, limited only by the size and speed of the device being used. The FPGA is an ideal prototyping platform costing much less than mask–programmed gate arrays. The part can be programmed on–site; no fabrication delays or large non–recurrent engineering costs are incurred. Some families of FPGAs are also targeted for mass production. These are usually lower–capacity parts that are only available in large quantities. The logic capacity of FPGAs is measured by the number of logic cells on the die. A logic cell is typically composed of a small lookup table followed by a flip–flop. FPGAs from different companies or even from different families from the same company have differences in the architecture of a logic cell. This makes the comparison between FPGA capacities more difficult.

Using computer aided design tools, the user's source code and/or schematics are converted into a bit–file that is uploaded into the device. Once the programming of the device is complete, the desired circuit becomes functional and the part can be used for its intended purpose. In this project, VHDL is the input method used to code digital circuits. The "V" in VHDL signifies Very High–Speed Integrated Circuit and HDL signifies Hardware Description Language.

Instead of focusing on specific internal details of the FPGA used in this project, general requirements for a suitable FPGA will be indicated. Someone wanting to implement this design on another family of FPGA need only ensure that the following criteria are met. The first requirement concerns the clock speed of the FPGA. The FPGA must operate at a sufficiently high clock speed to allow the bus interface logic

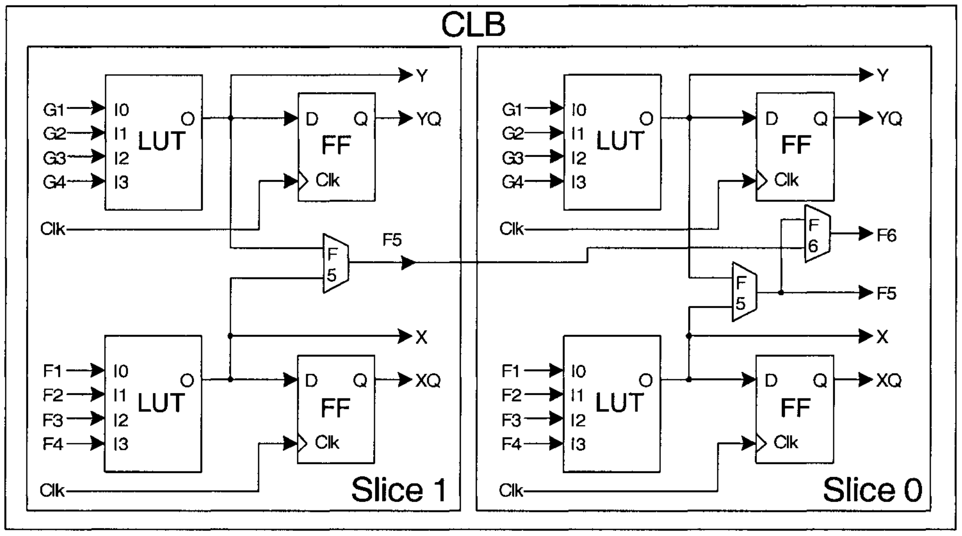


Figure 2.11: Simplified CLB diagram in Virtex devices. Carry logic and routing were removed for simplicity.

to operate correctly. In this case, the bus frequency is 33 MHz. Internal design logic can operate at a slower or faster rate, or even at the same clock frequency with multi–cycle stalls in state machine's states.

The second requirement concerns IO standards. The FPGA must support the signaling standard of the bus it is connected to. In this case, the FPGA pins can be explicitly configured to support the 5V, 33 MHz signaling protocol of the main bus used on the host computer's motherboard. It goes without saying that the FPGA package must have sufficient user–available IO pins to implement the necessary connections to the bus.

The third requirement involves logic capacity. The FPGA must have sufficient logic resources to allow the design to fit into the FPGA. Ideally, the FPGA should have at least 10% to 20% of its logic unused. This allows for extra flexibility in placement and routing and can result in a faster design (as opposed to a fully–loaded FPGA).

The final requirement involves the ability to use a logic–cell's lookup table as random access memory (RAM). This form of distributed RAM is essential when implementing memory–based move masking, a topic which will be covered in Section 4.5.

A simplified view of a Configurable Logic Block (CLB) is shown in Figure 2.11 so that the reader may have a better idea of the building blocks used to construct the move generator. Although this schematic corresponds to a Virtex device manufactured by Xilinx [70], most FPGAs are based on this type of architecture. The Virtex

CLB is composed of two Slices, which are in turn each composed of two logic cells. The device used in this project has an 84×56 array of CLBs. Not shown in the figure is another multiplexer allowing direct access to the flip–flop's D input. Carry logic and routing were also removed for simplicity. The basic logic gates are created by programming the appropriate values in the lookup table (LUT). Thus, any four–input logic function is directly constructed with 1 LUT. The F5 and F6 multiplexers are used to create five–input and six–input combinatorial logic functions. The flip–flop can be bypassed when creating asynchronous circuits. Internal CLB routing configuration is accomplished by programming various multiplexers. Inter–CLB routing is accomplished by programming interconnect matrices that control horizontal and vertical routing lines. All this configuration information is stored in static–RAM bits distributed throughout the FPGA. This implies that when the FPGA is powered–up, it must fetch its configuration bit–stream from an external source. This is typically stored in a programmable read–only memory chip located near the FPGA on the development board.

In Virtex devices [70], the BlockRAM is a 4096–bit synchronous memory. It can be configured for single–port or dual–port usage with variable widths of 1, 2, 4, 8 or 16 bits (with associated depths of 4096 to 256). In the device used in this project, 28 blocks are available. Dedicated routing helps to route signals to these blocks, which are constrained to each side of the chip. In contrast, DistributedRAM allows a LUT to be used as RAM. One LUT can implement a 16×1 (1–bit wide) synchronous or asynchronous memory. Two LUTs can combine to create 32×1, 16×2 or 16×1–dual–port memories.

In this thesis, the emphasis is aimed at *what the FPGA can be used for* rather than a technical analysis of digital design with FPGAs. For the same reason C language has abstracted machine–level instructions away from the programmer, VHDL has abstracted circuit–level details of digital design. Nevertheless, for the same reason that the knowledge of machine–level instructions yields more optimized C programs, knowledge of the underlying FPGA architecture is important when creating high–performance digital circuits.

2.5 Chess Ratings and Formulas

In this section, a presentation of the principal formula used to calculate a rating difference between two chess participants is derived. A confidence interval around the estimated rating difference is also calculated. These formulas are used in Section 5.4

Table 2.4: USCF chess rating categories

Rating	**Category**
2500+	Grandmaster
2200–2499	Master
2000–2199	Expert
1800–1999	Class A
1600–1799	Class B
1400–1599	Class C
1200–1399	Class D

to determine the increase in chess rating resulting from the addition of a hardware move generator to MBCHESS.

Chess ratings are calculated in order to numerically quantify the skill of a chess player. The most popular ranking is the ELO system, developed by professor Arpad E. Elo. The best players in the world have a ranking of approximately 2800 whereas beginners have a ranking of approximately 1200 points. Table 2.4 shows the different categories in chess rankings and their associated ratings [47]. In Section 5.5, MBCHESS–CODEBLUE will be evaluated in absolute terms and will be ranked according to the categories mentioned here.

In order to determine the chess rating difference between two players, a formula expressing the rating difference as a function of win–ratio must be derived. As a starting point for the demonstration to show how to obtain such a formula, an assumption on the strength distribution of chess players is made. This assumption states that the player's strength distribution follows the *normal distribution*[3]. In order for this assumption to be more clearly understood, a quote from *A Comprehensive Guide to Chess Ratings* [28] follows.

> ...suppose that every player brings a box containing many numbered slips of paper when sitting down to a chess game. Each number represents the player's potential strength during the game. This collection of values will be called a player's "strength distribution". Instead of actually playing a chess game, each player reaches into the box and pulls out a single piece of paper at random, and the one containing the higher number wins. In effect, this model for chess performance says that each player has the ability to play at a range of different strengths, but displays only one of these levels of ability during a game. Naturally, this procedure favours

[3]This is Elo's assumption which is reported in [28].

the person who carries a box that contains generally higher numbers, but of course this does not necessarily imply an automatic victory. This is analogous to chess where a better player usually wins, but not always.

Therefore, assuming a normal strength distribution, the area under the normal curve represents the probability distribution of a chess player's performance. Given this model, a chess player's strength distribution X_p is given by $X_p \sim N(\mu_p, \sigma_p^2)$. Here, μ_p represents the player's rating, σ_p^2 represents the variance of the rating parameter and N represents the normal distribution function. The opponent's performance distribution is $X_{opp} \sim N(\mu_{opp}, \sigma_{opp}^2)$. The performance distribution of the player against his opponent is shown in Equation 2.1.

$$D = X_p - X_{opp} \tag{2.1}$$

Using the subtraction property of normal distributions, D also follows a normal distribution.

$$D \sim N(\mu_p - \mu_{opp}, \sigma_p^2 + \sigma_{opp}^2) \tag{2.2}$$

Therefore the probability of the player winning the game against his opponent is the area under the positive portion of the D normal curve. An example where $\mu_p = 2000$ and $\mu_{opp} = 1800$ is shown in Figure 2.12. In the left side of the figure, the strength distributions of the player and his opponent are superimposed. In the right of the figure, the D curve resulting from the application of Equation 2.2 is shown. In this example, $\sigma_p = \sigma_{opp} = 200$. From the right side of Figure 2.12, the area under the curve where $x > 0$ is larger than the area under the curve where $x < 0$. This indicates that the probability of the player winning against his opponent is higher than 1/2. This is in agreement with the ratings given: $\mu_p > \mu_{opp}$, $2000 > 1800$.

Obtaining the exact probability of the player winning the game involves integrating Equation 2.2 from 0 to ∞. This is shown in Equation 2.3. For simplicity, $\sigma_p^2 + \sigma_{opp}^2$ is replaced by σ^2 and $\Delta\mu$ is equivalent to $\mu_p - \mu_{opp}$. Therefore, when using the $\Delta\mu$ symbol, the rating difference is seen from the point of view of the player and not the opponent.

$$P(D > 0) = \int_0^\infty N(\Delta\mu, \sigma^2)dx \tag{2.3}$$

The probability of the player beating his opponent, "$P(D > 0)$", will hereafter be labeled W_e, the winning expectancy for the player. Using the definition of the normal

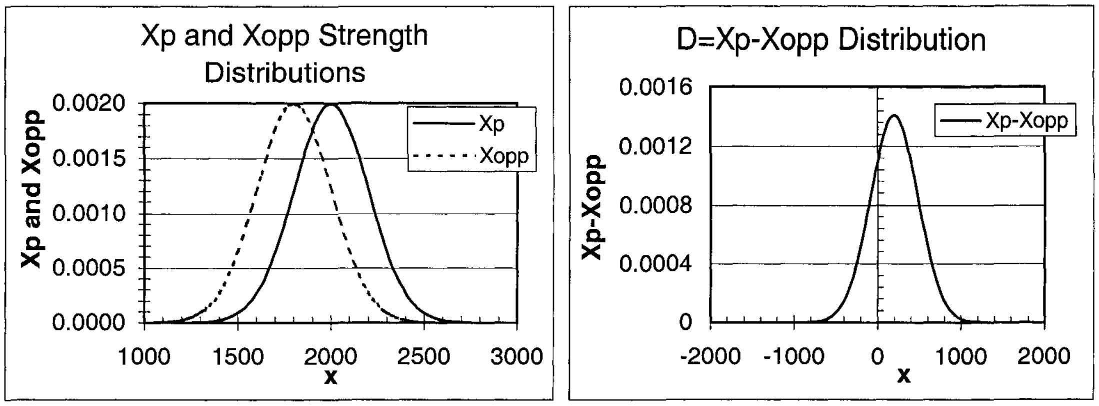


Figure 2.12: Example showing two strength distributions: $\mu_p = 2000$ and $\mu_{opp} = 1800$. The resulting performance distribution D is shown at right.

distribution function, this probability is given by Equation 2.4.

$$W_e = \int_0^{\infty} \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x-\Delta\mu}{\sigma}\right)^2} dx \tag{2.4}$$

However, the goal of the calculation is to obtain a formula that gives the rating difference as a function of winning expectancy W_e. This should involve solving for $\Delta\mu$ in Equation 2.4. However, because no analytical solution exists, another function will be used in place of the normal distribution. Is it to say that all the previous work shown here has been rendered useless? The answer is no, a similar procedure can be re–done when considering that each player's strength distribution follows the *extreme value distribution.* The proposed calculation is beyond the scope of the thesis; the procedure will continue with the logistic distribution curve for the following reason: for the combination of two players' extreme value distributions, the *logistic distribution* is shown to be virtually equivalent to the normal distribution [28]. The logistic distribution function is given in Equation 2.5. This function also has the property of unit area below the curve.

$$f(x) = \frac{e^{\frac{x-m}{\beta}}}{\beta\left(1 + e^{\frac{x-m}{\beta}}\right)^2} \tag{2.5}$$

Where:

- m is related to the normal distribution mean by: $m = \mu$;
- β is related to the normal distribution variance by: $\beta^2 = \frac{3}{\pi^2}\sigma^2$.

Figure 2.13 shows the similarity between the normal distribution and the logistic

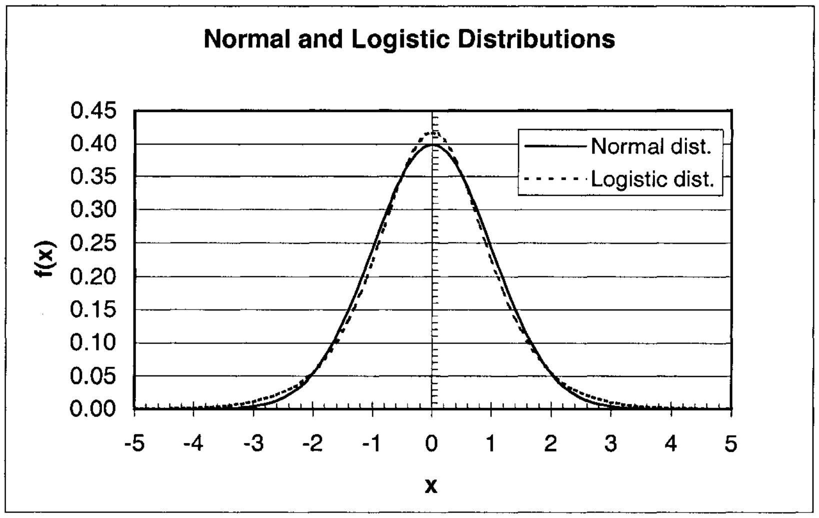


Figure 2.13: Normal and logistic distributions ($\sigma = 1$ and $\beta = 0.6$).

distribution. Both curves are practically identical, hence the use of the simpler logistic distribution formula.

As stated in the previous paragraph, when considering the difference distribution D, the logistic distribution function can be used in the same manner as the normal distribution function. In this case, Equation 2.4 becomes Equation 2.6.

$$W_e = \int_0^\infty \frac{e^{\frac{x-\Delta\mu}{\beta}}}{\beta\left(1+e^{\frac{x-\Delta\mu}{\beta}}\right)^2}dx \tag{2.6}$$

After calculating the definite integral in Equation 2.6, Equation 2.7 is obtained.

$$W_e = \frac{1}{1+e^{-\frac{\Delta\mu}{\beta}}} \tag{2.7}$$

Equation 2.8 is obtained when changing base e for base 10.

$$W_e = \frac{1}{1+10^{-\frac{\Delta\mu}{\ln(10)\beta}}} \tag{2.8}$$

Upon closer inspection of Equation 2.8, the standard deviation appears hidden in the term $\ln(10)\beta$. The *United States Chess Federation* uses a value of $\ln(10)\beta = 400$ in order to realistically model rating differences [29]. Hence Equation 2.9.

$$W_e = \frac{1}{1+10^{-\frac{\Delta\mu}{400}}} \tag{2.9}$$

Where:

- $\Delta\mu$ is the rating difference seen from the players point of view;
- W_e is the probability of the *player* winning the game (winning expectancy).

The logistic distribution has successfully been used in place of the normal distribution. Solving for $\Delta\mu$ in Equation 2.9 is now possible; equation 2.10 is obtained.

$$\Delta\mu = -400 \log\left(\frac{1}{W_e} - 1\right) \tag{2.10}$$

For example, $\Delta\mu > 0$ indicates that the player is stronger than his opponent. However, in order to calculate the rating difference, the wining expectancy must be known. The *exact* value of W_e is obtained when an infinite number of games are played. For the competition proposed in Section 5.4, a finite number of games will obviously be played therefore, W_e will be approximated. Let W_r denote the approximated value of W_e. Thus the win–ratio W_r can be expressed by Equation 2.11.

$$W_r = \frac{x}{n} \tag{2.11}$$

Where:

- x is the number of games won by the player;
- n is the total number of games played.

The number of games won, x, follows a binomial distribution. In order for the approximation of W_e to be valid, W_r must comply with the *Bernoulli Trials* criteria [7]. The following is a summary of these criteria, applied to a chess competition between two players.

- The result of a game must be either a win or a loss;
- n games are played and the number of victories is counted by x;
- The probability of a win is the same for each game of the experiment;
- The games are independent and non–exhaustive.

Therefore, by substituting W_r for W_e the formula expressing the estimated rating difference between the player and his opponent as a function of win–ratio is obtained.

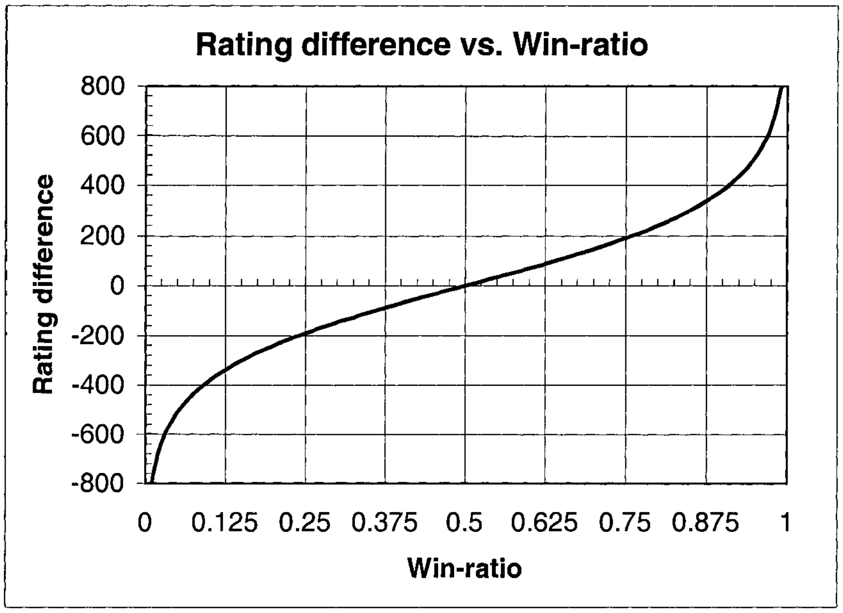


Figure 2.14: Rating difference vs. win–ratio.

This is equation 2.12.

$$\boxed{\widehat{\Delta\mu} = -400\log\left(\frac{1}{W_r} - 1\right)} \tag{2.12}$$

For example, if the player wins three out of four games, he is rated 191 points stronger than his opponent. Figure 2.14 shows the behavior of Equation 2.12. In addition, if a player wins half the games, he is equal to his opponent and has a rating difference of 0 points.

Because the winning expectancy (W_e) is approximated with the win–ratio (W_r), an error margin will be calculated to show the accuracy of the rating difference, given the finite number of games played. The variance of W_r is given in Equation 2.13.

$$\mathrm{VAR}\left(W_r\right) = \mathrm{VAR}\left(\frac{x}{n}\right) = \frac{1}{n^2}\mathrm{VAR}\left(x\right) \tag{2.13}$$

The variance of x is detailed in Equation 2.14.

$$\mathrm{VAR}\left(x\right) = nW_r\left(1 - W_r\right) \tag{2.14}$$

From Equations 2.13 and 2.14, the standard deviation of W_r is derived in Equation 2.15.

$$\sigma_{W_r} = \sqrt{\mathrm{VAR}\left(W_r\right)} = \sqrt{\frac{W_r\left(1 - W_r\right)}{n}} \tag{2.15}$$

Next, with a confidence interval of α=95%, W_e can be expected to be contained in the interval shown in Equation 2.16. In this equation, the value of $z_{\frac{\alpha}{2}}$ is obtained from the table of the reduced and centered normal distribution [7].

$$W_r \pm z_{\frac{\alpha}{2}} \sigma_{W_r} \tag{2.16}$$

The z value that corresponds to $\frac{\alpha}{2} = 0.475$ is shown to be 1.96. Thus the final error margin formula showing the interval in which W_e is expected to be is shown in Equation 2.17.

$$\boxed{W_r \pm 1.96\sqrt{\frac{W_r\left(1 - W_r\right)}{n}}} \tag{2.17}$$

An interesting property of Equation 2.15 is that when $n \to \infty$, $\sigma \to 0$. Combined with Equation 2.17, the error margin around W_r is reduced to 0. Asymptotically, W_r tends to W_e, confirming what was stated previously. However, it is important to consider the limitations of the confidence interval from Equation 2.17. If σ_{W_r} is large and W_r is close to 0 or 1, it is possible that the confidence interval will overflow the 0 to 1 interval allowed for the win–ratio. Thus, as the bounds get closer to 0 or 1, they begin to diverge once converted into chess rating bounds because of the vertical asymptotes at $W_r = 0$ and $W_r = 1$ (see Figure 2.14). Equations 2.17 and 2.12 are used in Section 5.4 to measure the results of the MBChess–CodeBlue vs. MBChess competition.

Chapter 3

Previous Hardware Move Generators

In this chapter, previously designed hardware move generators are reviewed. Important themes are exposed, some of which are exploited in the proposed design. Section 3.1 details three move generators that were successfully designed but were never used in competition–level chess machines. Two fundamental communication techniques are the foundation of move generators shown in this chapter. These are: *brute force interconnect* and *propagation through squares.* The HITECH move generator, based on the first of the two themes, is presented in Section 3.2. In Sections 3.3 through 3.5, the evolution of the propagation method is shown. The propagation method is the basis on which the proposed FPGA move generator is built; design details are presented in Chapter 4.

The alpha-beta algorithm was presented in Section 2.1.1. Two important characteristics must be present in hardware move generators. First, when a beta cutoff occurs, the remaining branches at a given node do not need to be explored. Thus, in order to be efficient, the move generator must be able to generate moves one at a time independently. In this way, no unused moves are wasted when a cutoff occurs. This implies that a node must "remember" what moves it has generated so that when the search returns to it, the next unexamined move can be calculated [22, 33, 35]. Second, as seen in Section 2.1.1, the move generator should return moves in a predetermined order, in order for the alpha-beta algorithm to be effective. In the following move generators, both these guiding principles are implicit goals. These principles are labeled *move masking* and *move ordering.* All the move generators presented in this chapter rely on the implicit parallel structure of digital circuits to increase their performance.

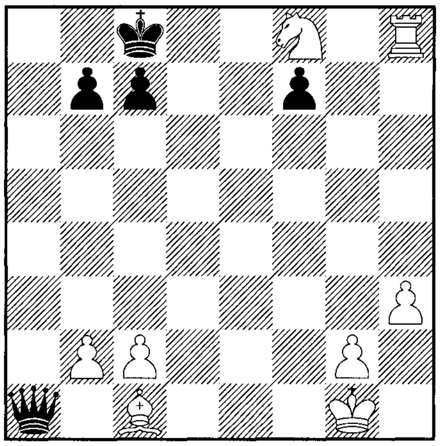


Figure 3.1: Pins and X–ray attacks. The white bishop is pinned to the white king because of the black queen. The white rook has an X–ray attack through the white knight.

3.1 Cheops and Others

In this section, move generators that were not used in complete chess machines are shown. Even though these designs did not participate in mainstream computer chess competitions and that no game–playing performance results are given in the key papers, it is nonetheless important to explore the ideas presented.

3.1.1 Berkeley Chess Microprocessor

The Berkeley Chess Microprocessor (BCM), was developed by J. Testa and A. Despain at the University of California, Berkeley [66]. The BCM is a 200 000 transistor, 1.2 micron CMOS integrated circuit. The chip incorporates a move generator, a basic positional evaluator and search control. No mention of any heuristics such as those given in Section 2 is given. The performance is rated at 3 million moves per second. In the design, each square has a six-bit adder that can be used for many purposes. The adder can sum the values of attackers to influence move ordering. It is speculated that this move ordering is better than the MVV/LVA move ordering presented in Section 2.2. The adders can also be used to calculate the mobility and square control of pieces for the evaluation function. Other features include the detection of pins and X–ray attacks. Pins and X–ray attacks are illustrated in Figure 3.1. These types of situations are detected with the help of a pin–enable control line that allows attack signals to pass through pieces.

Even though few details are given concerning the move generator, the underlying architecture seems to be based on the BELLE move generator. The terminology used is consistent with the BELLE move generator presented in Section 3.3. The BCM's move ordering was shown to be better than MVV/LVA because of the use of adders. The inverse of MVV/LVA priority is also possible however no application for this is mentioned. The move generator presented in Chapter 4 uses programmable–priority arbiters for optimal move ordering during full–width and quiescence search. As for move masking, a 25–level tag stack is used to enable or disable a piece from being an attacker or a victim. This keeps track of which moves have been generated at each ply. It is important to note that when a piece is deactivated, it continues to block attack signals from the rest of the board.

In this move generator architecture, two cycles are needed to obtain a move. In the first cycle, signals are generated in the directions in which the piece can attack. The goal is to find a victim. In the second cycle, the victim generates signals in all directions looking for possible attackers. Section 3.3 will describe in further detail the propagation of attack signals through neighbour squares. An interesting feature is also mentioned in which both attackers and victims send out signals simultaneously. Squares from which an aggressor could attack a victim can therefore be located. No further details or applications are given, however the key procedure for generating checking moves in Section 3.5 and Chapter 4 has discreetly been stated. The "priority logic" most likely corresponds to an arbitration tree used to select a square from the board when move selection is made. The adders used on each square are a good foundation on which to build a hardware positional evaluator. Again, many themes seen in this section were only briefly introduced; further details are given in Section 3.3.

3.1.2 VM* (Schaeffer *et al.*)

In the early 80s, *A VLSI Legal Move Generator for the Game of Chess* was designed at the University of Waterloo [62]. The chip was in fabrication at the time of writing of the referenced paper. Expected performance was evaluated at 350 000 moves per second with a clock frequency of 3 MHz. To begin the analysis of the problem to be solved, important properties of chess moves are indicated:

- *Row, column and diagonal independence* refers to the fact that given a certain direction, activity on all other parallel directions does not affect the moves on the given row, column or diagonal;

- *Move uniqueness* refers to the fact that given a certain direction, only one piece can land on a given square.

The principal communication method used in VM* is the propagation method. A message is allowed to propagate from one square to the next until it is obstructed by another piece. The basic building block of the VM* machine is the square machine. The square machine is a circuit with five ports: A) input port, B) output port, C) legal move output port, D) occupant input port, E) global information input port. Each square machine can propagate signals in one direction. Therefore, each square must have 8 square machines for the 8 directions and another 8 for knight moves. The output port is in all likelihood not used for knight moves because knights are not sliding pieces. Each square machine is connected according to its appropriate direction. The global control port and occupant port are wired in common for a chess square.

To compute a chess move, the board position is distributed throughout the chess machine using each square's occupant port. Each square machine then updates its output and legal move ports. Propagation delays determine the time needed to obtain a chess move; the worst-case propagation delay is seven square machines. At this point, it is important to realize that the amount of circuitry needed to implement the proposed design is rather large. For such reasons, the VM* design was modified to reduce the number of square machines needed and to obtain a manageable circuit. The first modification made to reduce the amount of resources necessary is to make the input/output ports bi-directional. With a global direction signal, half the number of square machines are needed. Two cycles are required given these bi-directional lines. The simplification can be taken a step further when considering a square machine that communicates with four other square machines. In this case four cycles are necessary. Another step yields an eight way square machine requiring eight cycles. Hence, two square machines are used for each square: one for knights and the other for non–knight pieces.

With these modifications, the circuit is smaller but still represented a fair amount of resources given the 8×8 chessboard. Another simplification of hardware entails the reduction of square machines to one for each row. Each square machine has its own state machine that sequentially applies the value in each square of its corresponding row. The chessboard thus becomes an array of square cells where each location is seen as a four–bit occupant register. Generating moves in the left–to–right direction involves reading the square cells from left to right; the opposite is done for right–to–left moves. The total amount of square machines is reduced to eight. It is important to note that with the modification proposed here, the design is no longer based on the propagation–through–squares technique.

The proposed modification does not support vertical moves. To avoid using another eight square machines for the eight columns, flexible horizontal and vertical routing allows the square cells' values to be sent directly to the row square machines. For diagonal moves, the procedure is more complex. Instead of creating diagonal routing, vertical shifting of the output latches of the square machines is introduced. Reading the rows and columns in the proper order will create the effect of reading a diagonal. Knight moves are also handled within this context: the order in which rows and columns are read is modified according to knight squares.

Because of the row, column and diagonal independence property stated previously, it would seem as though the moves that are generated by the VM* are pseudo–legal moves, rather than fully "legal" moves. Pseudo–legal moves are moves that potentially leave the moving side's king in check. When considering legal moves, the row, column and diagonal independence property is not true. For example, a piece may be pinned by an enemy piece on another parallel row, column or diagonal, thus preventing it from moving. Also, no explicit mention of why the generated moves are completely legal is made in [62]. Nevertheless, it is important to mention that 350 000 moves per second is an encouraging result, given the technology used in the early 80s and the partial non-parallelism of the design. In the VM* design, parallel processing of eight rows is used to gain a performance advantage. Very Large Scale Integration (VLSI) or outrageous amounts of discrete logic chips (see the BELLE design) are used to obtain 64 square parallel processing. These move generators are covered throughout this chapter.

3.1.3 CHEOPS

The third move generator covered in this section concerns the CHEOPS project developed by J. Moussouris *et al.* [44] at the Massachusetts Institute of Technology. The Chess–Oriented Processing System (CHEOPS) is designed as a general chess–program accelerator that implements a hardware move generator and alpha–beta search control. The chess program is then free to deal with the application of chess knowledge. The search control portion of the design is accomplished with a CPU based on a 16–bit ALU with 16 accumulators. This processor is specifically designed to execute chess algorithms. When running the alpha–beta algorithm, no mention of move ordering heuristics is made.

The chess array module is an array of random logic that implements non–numerical chess operations, most important of which is move generation. Similarly to the CODE-BLUE move generator that will be presented in Chapter 4, low–level chessboard operations are externally accessible through micro-instructions. The 8×8 chess array module can also be used to see if a king is in check or to see if any capturing moves are possible. Each square is built using approximately 12 TTL DIP (dual inline package) chips. The move generator has two main operations. In the first mode, a square is designated as the destination square. Each square has its own signal to indicate if it contains a piece that can attack the destination square. Conversely, a square can be designated as the source piece. In this case, each square's signal indicates if it can be reached by the source piece or not. These two modes could be useful in the following cases (not mentioned in the referenced paper):

- When the MVV/LVA move ordering is desired, destination square based move generation is used. The destination square is cycled from queen to empty squares, accomplishing the MVV portion (Most–Valuable–Victim);
- When generating moves for a discovered check, source based move generation is used. Assuming that discovered checks cannot be directly generated and that only their locations can be identified, the discovering piece must be scanned to find its possible moves. The discovering piece is therefore tagged as the source.

Additional state bits can instruct the move generator to find only capturing moves or to differentiate pawn captures from piece captures[1]. Another control bit is used to set the entire board as the destination squares in order to quickly determine if any captures exist. The propagation communication technique is used in this design to handle sliding pieces, however, no architectural details are given.

The square lines must be properly analyzed to generate the next move. This is accomplished with hardware DO loops that scan the square lines. With the last square's coordinates, the next active square line can be determined. In the CHEOPS design, the squares are scanned in raster order. Could move ordering be improved if raster order were replaced by a center–prioritized pattern? This question will be answered in Section 4.4. An interesting data representation is created whereby a memory keeps track of where each piece is located and another memory keeps track of the piece contained on each square. In such cases, the advantage of quickly accessing data more than compensates for the extra work involved in managing both memories.

[1]In chess terminology, a pawn is not considered a piece.

A chess move is encoded as two 16–bit words, one for the source square and moving piece and another for the destination square and captured piece. Only the current move at each ply is stored on the PDL (the PDL is a 1024–word stack). When the search returns to a node, the previously searched move that is currently in the stack is used to help generate the next move. Also present on the stack is an alpha or beta value used in the search, along with castling and en–passant state bits. One bit in the opcode determines if the instruction is a CPU instruction or a chess module micro–instruction. As a result, a chess program can use special chess–hardware instructions directly in its source code, a practical combination.

Computer–Aided Design (CAD) played a key role in the construction of the CHEOPS hardware. A template drawing of a chess square circuit was first created. Macros were then used to replicate the circuit for the entire chessboard, indexing signal names according to square locations. Edge effects were handled with other macros. In the CODEBLUE VHDL design, the synthesis software automatically simplifies the edge squares. For example, a corner square has only three neighbours instead of eight. The CAD tools were responsible for generating net–lists, which were used to control automatic wire–wrapping machines. The CHEOPS chess machine was completed in a surprisingly–short one and a half man–years of work. At the time of writing of the referenced paper, CHEOPS was being integrated to the MACHACK and TECH II chess programs.

3.2 Hitech

The HITECH chess–machine [12] is based on a hardware move generator developed by Carl Ebeling and Andrew Palay at Carnegie Mellon University [23]. This move generator is the only one based on the *brute force interconnect* communication technique mentioned at the beginning of the chapter. The move generator is built using 64 identical chips, one for each square of the chessboard. Each chip is packaged in a 40 pin DIP and is fabricated with a 2–micron NMOS process. The move generator is capable of generating 500 000 moves per second and when used in conjunction with the HITECH machine, 175 000 positions per second can be searched. HITECH was the first artificial chess player to obtain the U.S. Senior Master rating (>2400).

The underlying principal on which the HITECH move generator is built is that of computing the subset of the ever–possible moves that are valid for a given chess position. The set of ever–possible moves is approximately 4 000 for each side and indicates the upper bound on the number of different moves that can ever be produced on a chessboard. Each ever–possible move detector can be seen as a function that maps part of the chessboard (approximately 260 bits) to a true or false signal. Move generation then involves selecting from the moves that are true, in a proper order. In a typical middle–game position, 40 to 50 of the ever–possible 4 000 moves are valid. Even though a given function uses only a few bits from the chessboard to compute its move, 8 000 such circuits are needed. Tens of thousands of gates connected by tens of thousands of wires to the chessboard bits is unfeasible using the technology of the early 80s. The problem is not the large amount of logic resources but rather the large amount of communication resources. The chessboard bits also suffer from excessive fanout. The solution to this problem involves replicating the board state for each square. Therefore, each square has its own copy of the chessboard and can calculate moves on its own. A multi–level priority circuit is used to obtain the next best move from the entire set of ever–possible moves that are asserted (true). This circuit also performs the proper ordering of moves required by an efficient alpha–beta algorithm.

Because of the replicated chessboard, an event bus must be used to communicate changes in board state to each square. The set of ever–possible moves is divided into 64 blocks, some of which have the maximum 77 ever–possible moves to decode (ignoring the side to move). For example, the e4 square has 77 possible sources: 8 king, 27 queen, 14 rook, 13 bishop and 7 pawn moves. For an ever–possible move to be asserted, it must comply to the following three criteria:

- The *origin condition* stipulates that the proper piece must be present on the source square;
- The *destination condition* requires that destination square be empty or that it contains an enemy piece. This distinction is important for pawn moves versus pawn captures;
- The *sliding condition* requires empty squares between source and destination squares for queen, rook, bishop and two–square pawn advances.

These three conditions imply that the generated moves are pseudo–legal, as are all the moves returned by the move generators shown in this chapter.

The interconnect pattern of a chess chip is detailed in Figure 3.2. In this example, the chess chip is assigned to the b3 square and is therefore responsible for generating moves that land on b3. The chess square connections that do not coincide with board squares are disabled. Because each chip is identical, logic was not optimized. Each

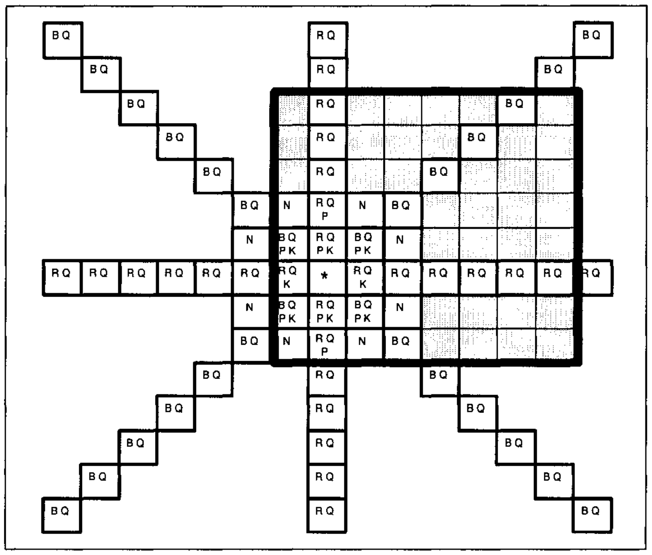


Figure 3.2: HITECH chess chip interconnect pattern. The chip shown here is assigned to the b3 square (marked with a $*$). The type of piece capable of attacking the destination square is shown in each source square.

chip is assigned to a different square of the chessboard by programming a destination address register.

Each square's move signals are sent to a priority encoder in order to find the highest valued move. The value of a move is calculated using three factors: the value of the moving piece, the value of the captured piece (if any) and the safety of the destination square. The safety parameter takes into account the number and color of pawns controlling the destination square. If no pawns have control of the square, the next lowest valued piece is used. The priority encoder of a square follows the least–valued–aggressor (LVA) ordering scheme. Therefore, capturing with lowest valued pieces is preferable to capturing with higher valued pieces. The value of a move is stored as a six–bit priority number. A square computes its best move given its location (destination square) and submits it to the voting network. Each square submits its six–bit move value to the voting network, where the best move is selected. When a voting tie occurs, each square's unique six–bit id is used to break the tie. This six–bit id is preset into each chip during the initialization phase. Adjusting the six–bit id to prioritize moves to the center of the board would be a clever way of implementing arbiter centrality (Section 4.4), however this is not mentioned in the referenced paper.

Because of the depth–first nature of the search algorithm, each chess chip must be able to save and reload the context of move generation at each ply in the search. The index of a move is a number between 1 and 80 and represents the number of the ever–possible move corresponding to a given move. The priority encoder can be instructed to ignore the first N encoder inputs (moves). Because the move index is stored along with the move itself, the previously generated move at each ply is sufficient to mask all previous moves, thus generating the next, un–searched move. The design also offers the possibility to test the validity of killer moves and transposition table moves. Each chip obviously has the necessary logic to make and unmake moves on its local board.

3.3 Belle

The first hardware chess–playing machine to compete in a computer chess tournament was the BELLE machine [22]. BELLE was developed at Bell Laboratories by Joe Condon and Ken Thompson. The first hardware version had 25 chips and competed in the 1977 World Computer Chess Championships. A larger BELLE machine composed of 325 chips placed first and second in the 1978 and 1979 ACM Championships. This machine incorporated hardware positional evaluation, hardware transposition tables and a hardware move generator. The next version of BELLE integrated alpha–beta search control and utilized an astounding 1 700 discrete logic chips. This machine was completed in 1980 and was able to search 160 000 positions per second, much better than the 200 positions per second searched by the purely software BELLE.

In BELLE, the chessboard is an 8×8 array of combinational logic blocks. The main hardware structure in BELLE deals with the communication between chess squares. Each square has a transmitter, a receiver and a four–bit piece register, denoting the current occupant. A given square is only connected to its eight neighbours, except for knight lines that must pass over neighbouring squares. The empty squares are responsible for propagating sliding–piece lines along the different directions. Aggressor pieces activate their appropriate transmit lines, given their piece types, whereas victim squares receive incoming attacks and apply for arbitration. The two major communication blocks are the transmitter and the receiver. Each square has one of each. This design is the foundation of the *progapation through squares* communication technique mentioned in the chapter introduction. The move generators presented in the remainder of this chapter, along with the CODEBLUE move generator designed in Chapter 4, are based on BELLE.

A chess move is a transfer from a source square to a destination square. To construct a move, two cycles are executed. First, a *find–victim* cycle locates the

Table 3.1: BELLE arbitration priority values.

Priority	**Level**	**Find-victim (FV)**	**Find-aggressor (FA)**
(highest)	1	Queen	Pawn
	2	Rook	Knight
	3	Bishop	Bishop
	4	Knight	Rook
	5	Pawn	Queen
(lowest)	6	Empty	King

destination square and then a *find–aggressor* cycle locates the source piece. During the find–victim phase, all pieces belonging to the player–to–move will activate their transmitters. All opposing pieces and empty squares send the output of their receivers to the arbitration network. Once a most–valued victim (MVV) is selected, the find–aggressor cycle executes. Here, the victim found in the first cycle transmits as the union of all piece types and the moving pieces' receivers arbitrate to select the least–valued aggressor (LVA), ordered from pawns to kings. This produces the MVV/LVA move ordering. The find-victim (FV) and find–aggressor (FA) cycles each require 250 ns to complete.

The identification of the square with the highest priority is done with a two–level priority tree. Priority values for find–victim and find–aggressor cycles are outlined in Table 3.1. From this table, the most–valuable–victim / least–valuable–aggressor move ordering scheme should be apparent. Each priority level corresponds to a single asserted line of the six lines connected to the square's arbiter. Not shown in the table is the priority level associated with "nothing to arbitrate". In this case, none of the lines are activated and the square has nothing to contribute. The first level of arbitration is done on a block of 4×4 squares; the second level selects one of these subgroups.

As for the move–masking capabilities of BELLE, 64 bits (one for each square) are dedicated to mask aggressors exhausted for the given victim, or mask fully searched victims (a square is either a victim or an aggressor). This is accomplished by sending the output of the mask memory to the receiver where it is used to disable the six signals sent to the arbiter (priority network). Because of the depth–first search, a stack of 64 levels is used to memorize these mask bits. This move masking method is used in the proposed design (Chapter 4). It is therefore pertinent to clarify the move generation procedure with an example. In Figure 3.3, all the moves from the chess position are returned sequentially. Depth first searching has been removed for

Move generator steps:
1) FV locates the white queen.
2) FA locates the black pawn.
3) "axb2" is returned, black pawn is disabled.
4) FA locates the black bishop.
5) "Bxb2" is returned, black bishop is disabled.
6) FA fails to find aggressor for white queen.
7) White queen disabled, all aggressors re–enabled
8) FV now locates the white pawn.
9) FA locates the black bishop.
10) "Bxe3" is returned, black bishop is disabled.
11) Moves to empty squares follow...

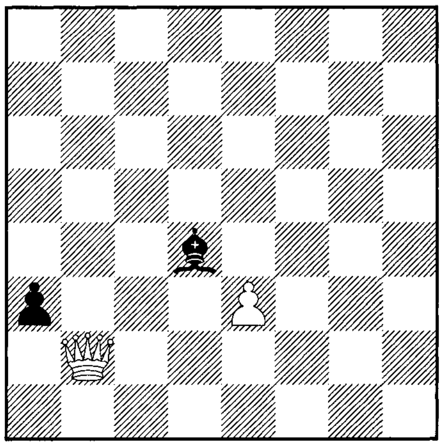


Figure 3.3: BELLE move generation example, black to move. Details of each cycle of move generation and the effects on the move masks are given.

simplicity. In the chess diagram of the example, it is black's turn to play: black pieces are aggressors and white pieces are victims.

In BELLE, special chess moves such as castling, en–passant pawn captures and pawn promotions are handled by additional random logic distributed throughout the design. Pawn promotions are detected with a special PRO–ONLY flag. When this flag is activated, the receivers and transmitters are limited to pawn moves to the last row. Because promotions seldom occur, a single find–victim + PRO–ONLY cycle is executed to determine that no pawn promotions exist. The use of this special instruction ensures that pawn promotions are well placed in move ordering. The CODEBLUE move generator does not use a special instruction for pawn promotions (Sections 4.4 and 4.6). The BELLE design has introduced the *propagation through squares* communication scheme as well as memory–based move masking. These, along with the find–victim and find–aggressor instructions, are utilized in the CODEBLUE design.

3.4 Deep Thought

Aside from HITECH, another hardware chess project originated from Carnegie Mellon University. The initial phases of the DEEP BLUE chess machine (Section 3.5) were CHIPTEST and DEEP THOUGHT [21]. CHIPTEST first played in 1986 and was based on a hardware move generator. Its search speed was rated at 50 000 nodes per second. In 1987, CHIPTEST–M incorporated micro–coded hardware search control

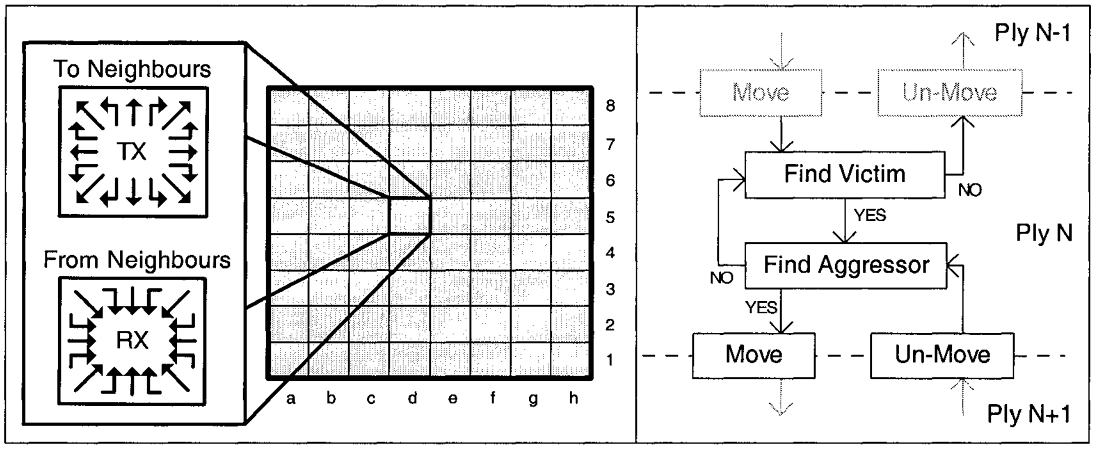


Figure 3.4: DEEP THOUGHT square array and move generation sequence. The transmitter (TX) and receiver (RX) interconnections of a chess square are visible at left. The principal cycles of recursive searching are shown at right.

and was able to search 400 000 nodes per second. With DEEP THOUGHT came multi–processor alpha–beta search. In this section, the hardware move generator used in DEEP THOUGHT is reviewed.

Designed by Feng–Hsiung Hsu in 1985 and 1986, the DEEP THOUGHT move generator is a single–chip, 3–micron CMOS integrated circuit capable of generating two million moves per second [33]. When reviewing previous designs prior to the DEEP THOUGHT design, the BELLE approach was believed to offer better potential for mapping to VLSI than the HITECH design. With this assumption, the receiver/transmitter structure introduced in the BELLE design was used as a starting point for the DEEP THOUGHT move generator. The same two requirements for a hardware move generator have not changed: generate moves one at a time, given a pre–determined move order. A third requirement is introduced, given the importance of quiescence search. The move generator should be capable of generating capturing moves first. Because of the MVV/LVA move ordering exhibited by the BELLE design, generating capturing moves first is implicitly accomplished. The LVA ordering is also stated as being ideal in quiescence search.

Once again, an 8×8 array of chess square circuits is used. However, because this design fits on a single chip, edge–effect optimizations are performed. The same find–victim and find–aggressor cycles shown in the BELLE section are used to generate moves. The chip is also capable of making and unmaking moves to allow it to follow the recursive depth–first search algorithm. A simplified flow chart of this process is illustrated in the right portion of Figure 3.4. For simplicity, move masking was

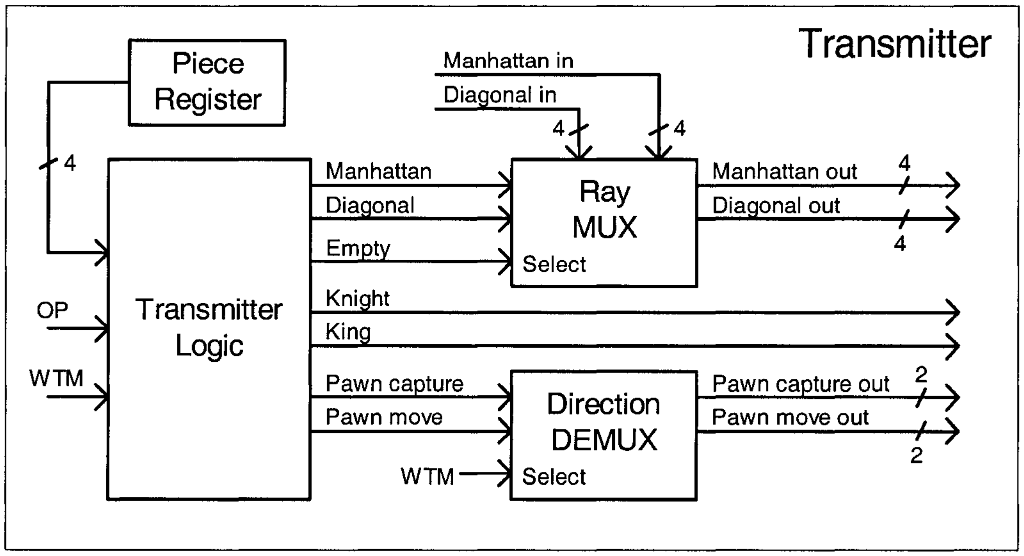


Figure 3.5: DEEP THOUGHT transmitter circuit. Attack signals are generated and/or propagated depending on the state of the inputs at left.

not shown. Combined with the example given in Figure 3.3, the move generation procedure should be getting clearer. Making and unmaking moves is accomplished by writing the appropriate values in each square's piece register. In the left portion of Figure 3.4, transmitter (TX) and receiver (RX) connections for one of the 64 chess squares are shown.

When entering a node for the first time, all aggressors and victims are enabled. From Figure 3.4, the first cycle executed is the find–victim cycle. If this is successful, a find–aggressor cycle is executed. If the find victim cycle cannot locate a victim, all victims have been searched and move generation is finished. In such a case, the move leading to the current position is unmade and the search backtracks to the previous ply. Given a successful find–victim cycle, the find–aggressor cycle locates the attacking piece and the move is now ready to be made. Aggressors are masked as they are found until all aggressors for the given victim have been returned. When the find–aggressor cycle fails to locate an aggressor, the victim is disabled and all aggressors are re–enabled. The next un–searched victim is located with a new find–victim cycle.

The circuit responsible for generating the attack signals used in the find–victim and find–aggressor cycles is the transmitter. The block diagram of the transmitter circuit is shown in Figure 3.5. The *manhattan* keyword is used to describe the horizontal and vertical directions. WTM is a signal indicating if it is white–to–move and OP is the signal indicating either the find–victim or find–aggressor cycles. The transmit logic can be seen as a six–bit ROM with seven outputs. In the find–victim cycle, the seven outputs of the transmit logic are asserted according to the resident piece type (or empty). If a pawn occupies the square, both pawn moves and pawn captures are signaled. However, in the find–aggressor cycle, the seven outputs are asserted as if a super–piece was located on the square. The super–piece represents the union of all piece–types and is necessary to reach out to any possible aggressors. The pawn capture is not asserted if the square is empty. Conversely, the pawn move is not asserted if the square is occupied. The Ray multiplexer is responsible for propagating incoming sliding piece signals if the square is empty or generating the source piece's signals if the square is a manhattan and/or diagonal capable piece. Pawn signals are sent in the proper direction (up/down) with a direction demultiplexer[2]. Furthermore, in the DEEP THOUGHT paper [33], the pawn move and pawn capture outputs are mistakenly drawn as single lines. The demultiplexer is responsible for sending pawn information either up or down, hence two outputs for both types of pawn attacks. This correction is made in Figure 3.5.

Transmitter signals are received and decoded on each square via the receiver circuit shown in Figure 3.6. The receiver is responsible for sending the appropriate priority signals to the arbitrations network. The receiver does not need to know from which direction it was hit. Therefore, each direction is grouped according to each signal type as shown at the left of the figure. A priority signal is asserted if the proper conditions are met. In the find–victim cycle, a hit from any direction and type will cause a priority level dependant on the square's piece value (empty is also permitted). Pawn moves are more constrained: a pawn move hit will only be signaled if the square is empty and a pawn capture hit will only be signaled if the square is occupied. In the find–aggressor cycle, the type of hit must match the capabilities of the resident piece. The victim piece is generating the union–piece and receivers must indicate if they have a piece that can reach the victim. For example, if a rook gets hit in the diagonal direction by the super–piece victim, it should not be signaled as a valid aggressor. How did this victim previously get tagged if the rook never transmitted in the diagonal direction? In the example given here, one or more *other* attacker pieces must have hit the victim during the previous find–victim cycle. As was the case in BELLE, the MVV/LVA move ordering is obtained by inverting priority levels; this is visible in the receiver figure. Illegal board positions are signaled with a 64–input OR gate used to detect when a king is attacked.

[2]In [33] this is erroneously referred to as a multiplexer.

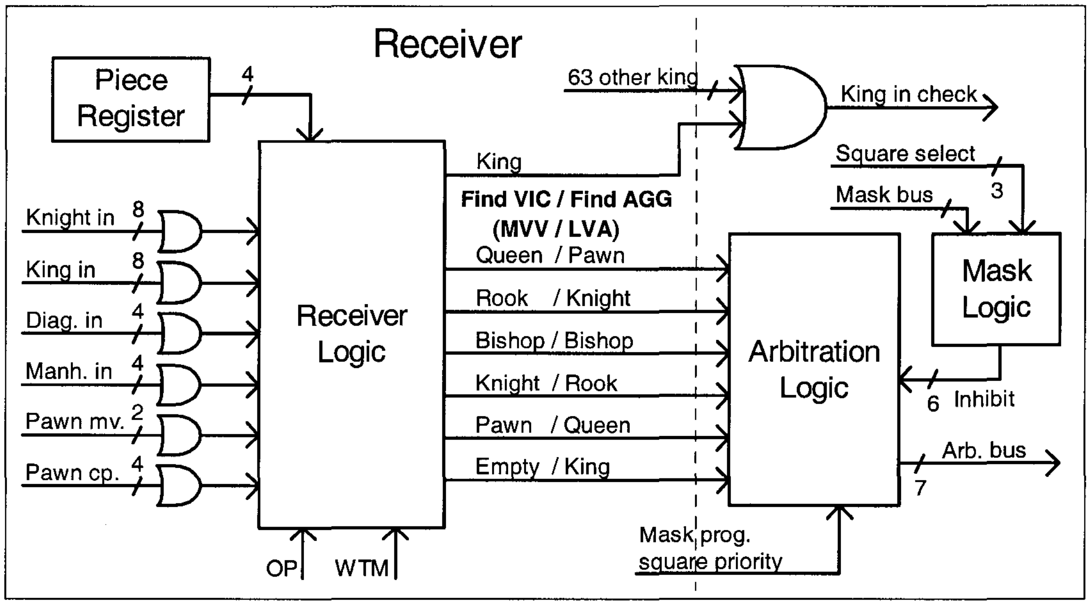


Figure 3.6: DEEP THOUGHT receiver circuit. Attack signals are decoded and priority signals are sent to the arbitration logic. First level arbiters and masking logic are also visible.

In DEEP THOUGHT, the first level arbitration is done on rows of squares and the second level selects between rows. A first–level arbiter is visible in Figure 3.6. Once a square is disabled by bit masking, it no longer participates and other squares with the same priority continue the voting process. This process continues until all possible moves are generated.

The transmitter and receiver circuits shown here are essentially the same as in BELLE. However, the process of masking victims and aggressors is completely different in the DEEP THOUGHT architecture [33]. Instead of using a 64–bit stack of 64 levels, as was the case in the BELLE design, mask bits are calculated from the previously generated move at a node. This masking method eliminates the need for memory, at the expense of logic and decoders. In the VLSI design of DEEP THOUGHT, this approach was advantageous [35]. In the CODEBLUE design, arbitration and square masking will be customized for FPGAs and will be presented in Sections 4.4 and 4.5.

Each square's piece register is dual–ported for maximum speed. The arbitration and masking buses are also used to update the piece registers when moves are made and unmade. Extra logic is added to the third and sixth rows to handle double square pawn advances. The first and eighth rows are also modified to support pawn promotions. Castling and en–passant moves are accomplished by loading shadow pieces and testing the state of the move generator (king in check and find–victim priority). As in BELLE, a promotion–only flag is used to detect pawn promotions. This cycle is not executed when there are no pawns on the penultimate row. With a faster, single–chip design, how can the hardware move generator be improved? Section 3.5 follows with the DEEP BLUE move generator.

3.5 Deep Blue

In 1996, a descendant of DEEP THOUGHT, named DEEP BLUE, played a six–game match against the world chess champion and lost four of the six games [21]. Nevertheless, it was an encouraging result and showed that chess machines were now close to beating the best player in the world at his own game. DEEP BLUE utilized 216 chess chips and was able to search 50 to 100 million positions per second. One year later, the machine used in the rematch would prove to be too strong for Gary Kasparov. With 480 chess chips and a search speed of 100 to 200 million positions per second, DEEP BLUE II won the match by a score of 3.5 to 2.5. Each chess chip is identical and incorporates the three following aspects of computer chess hardware: search control, positional evaluation and the move generator. Because the move generator is essentially the same in both DEEP BLUE machines, no distinction between DEEP BLUE and DEEP BLUE II will be made. The DEEP BLUE II move generator added the possibility to generate moves that attack the opponent's pieces.

The first major improvement introduced by the DEEP BLUE move generator solves the problem of generating checking moves separately. This can be used to search forcing lines more efficiently during quiescence search. In MBCHESS–CODEBLUE, this feature is used to prioritize checking moves in the full–width–search move ordering. To generate checking moves explicitly, two transmitters are used, as well as a receiver with twice the number of inputs [35]. During the find–check phase, the pieces for the side to move activate their find–victim transmitters and the opposing king activates its find–aggressor transmitter. The find–victim transmitter is hardwired to transmit according to the resident piece's capabilities whereas the find–aggressor transmitter is hardwired to transmit the union of all piece types. Squares that register appropriate hits from opposing sides will indicate a square from which a piece can check the enemy king. In this thesis, these squares are referred to as *pivot squares*; the cycle is called the *find–pivot* cycle. The second cycle necessary to generate the checking move is most likely a find–aggressor cycle where the pivot square radiates as the super–piece; however this was not documented in the referenced papers.

An example of a find–pivot cycle is given in Figure 3.7. The white bishop on

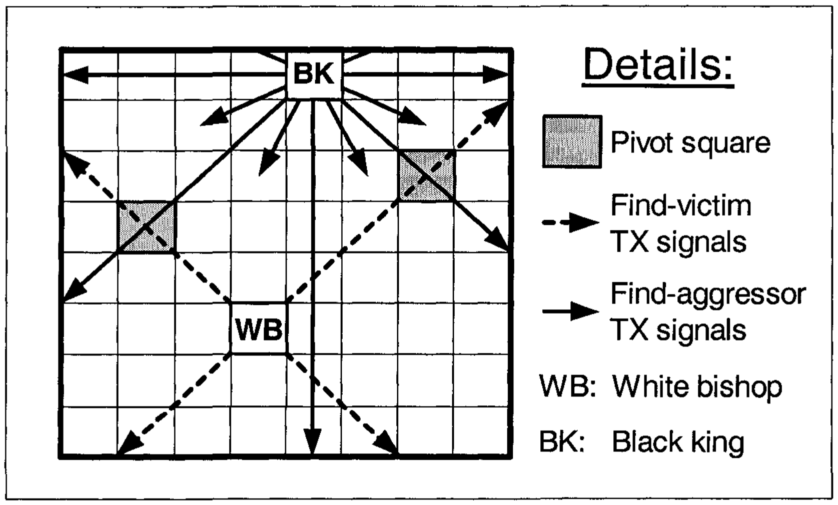


Figure 3.7: DEEP BLUE find–pivot cycle example. The intersection of diagonal lines on the two pivot squares (grey squares) indicate a square from which the white bishop could check the opposing king.

d3 can check the black king on e8 with two different moves. The king activates its find–aggressor transmitter and the bishop activates its find victim transmitter. For simplicity, only knight and sliding piece signals are drawn for the union–piece emanating from the black king. When signals from opposite players align correctly on a square, a pivot square is detected and the destination portion of the move is obtained. The source portion of the chess move is found using a find–aggressor cycle from a pivot square. Extra constraints are needed for this to function correctly; details are given in Section 4.3. Receivers have two sets of inputs in order to accept signals from both types of transmitters simultaneously. During the ordinary non–checking move–generation cycles, each transmitter is used in its respective cycle (the find–aggressor transmitter is used during the find–aggressor cycle and the find–victim transmitter is used during the find–victim cycle).

The published papers contain few details concerning DEEP BLUE's checking move generation [21, 35]. In the DEEP BLUE find–victim transmitters, two sliding–piece signals are transmitted in each of the eight directions [36]. However, in [35], it appears as though only one signal is transmitted in each direction. The two signals allow pivot squares to differentiate queens from bishops when being hit in the diagonal directions, as well as to differentiate queens from rooks when hit in the manhattan directions. Figure 3.8 shows an example position of this situation. The transmitter interconnect pattern used in the CODEBLUE design and implications for checking moves are detailed in Section 4.3.

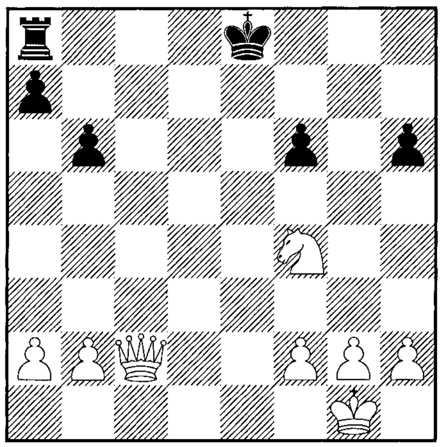


Figure 3.8: Checking moves: the e4 pivot square can differentiate a queen from a bishop using two signals. The queen hits the e4 square from the southwest diagonal lines. A queen reaching the pivot square causes a check whereas a bishop does not. The e4 square is the white square immediately to the left of the white knight.

The chess square interconnect pattern used in the CODEBLUE design utilizes fewer inter–square wires than the DEEP BLUE design. In DEEP BLUE, the number of wires needed to connect a square to its neighbours is shown in Table 3.2. For piece propagations throughout the chessboard, a fully–connected square has 68 inputs and 68 outputs. These values will be used in Section 4.3 for comparison with the proposed design. The find–victim transmitter used in DEEP BLUE resembles the transmitter shown in DEEP THOUGHT (Figure 3.5). The only exception is the doubled manhattan and diagonal outputs mentioned previously (this explains the "16"s in the table). In the DEEP THOUGHT transmitter, even though the king output is not labeled as an eight–bit bus, the king signal is sent to its eight neighbours, thus representing eight wires. The number of bits of the output function, in this case one, should not be confused with the various destinations for the signal, in this case eight. The same analysis applies to the pawn capture signals: a two–bit bus is shown, one wire for north captures and one for south captures. Both directions are needed in order to support both color pawns. In this case, the north capture signal is sent to the north–west and north–east squares. This explains why four pawn capture wires are shown in the table.

The arbitration network and move masking are very similar to what was described in DEEP THOUGHT. However, an interesting quote from a recent DEEP BLUE paper [21] reveals an interesting move ordering issue.

Table 3.2: Number of chess–square connections in DEEP BLUE.

Chess–square interconnects	OUT	IN
Find–Victim signals	(Transmitter)	(Receiver)
Pawn move, 1 north (N), 1 south (S)	2	2
Pawn capt., N–East, N–West, S–East, S–West	4	4
King, all directions	8	8
Knight, all directions	8	8
Queen, rook, bishop, all directions	16	16
Find–Aggressor signals	(Transmitter)	(Receiver)
Pawn move, 1 north (N), 1 south (S)	2	2
Pawn capt., N–East, N–West, S–East, S–West	4	4
King, all directions	8	8
Knight, all directions	8	8
Queen, rook, bishop, all directions	8	8
Total	68	68

The chess chip uses an ordering that has worked well in practice, first generating captures (ordered from low–valued pieces capturing high–valued pieces to high–valued capturing low–valued), followed by non–capturing moves (ordered by centrality).

Centrality was mentioned in Section 3.2 as a potential improvement to the HITECH move generator. No further implementation details are given as to how this centrality is accomplished in DEEP BLUE. The other centrality theme mentioned in the paper deals with positional tables and should not be confused with hardware move ordering. Details concerning CODEBLUE centrality will be given in Section 4.4.

3.6 Summary of Characteristics

To complete the present chapter, a summary of the move generators presented in Sections 3.2 to 3.5 is presented in Table 3.3. Each of the fields in the table also applies to the CODEBLUE move generator and should become obvious once Chapter 4 is read.

Other themes recurrent in most or all of the previously mentioned designs are:

1. *Computer–aided design* was used in many different forms to aid the creators in the fabrication and the design of their complex digital circuits;

Table 3.3: Summary of move generators.

	HITECH	BELLE	DEEP THOUGHT	DEEP BLUE
Year:	1983–1990	1973–1980	1989–1995	1996–1997
Communication technique:	brute force interconnect	propag. through squares	propag. through squares	propag. through squares
Move ordering based on:	dynamic square values	MVV/LVA	MVV/LVA	MVV/LVA
Move masking:	last–move decode	stack (memory)	last–move decode	last–move decode
Can generate checking moves separately?	no	no	no	yes
Can generate captures only?	yes	yes	yes	yes
Check evasion mode:	yes	no	no	yes

2. Each design was influenced by the fundamental property of chess moves which involves moving a piece from a *source* square to a *destination* square;

3. All the move generators presented here were designed to return *pseudo–legal* chess moves. Some authors refer to these as legal moves. Pseudo–legal moves were explained in Section 2.2;

4. Special chess moves are handled by a combination of software routines and/or extra random logic distributed throughout the appropriate squares of the chessboard.

Chapter 4

Hardware Move Generator Design

This section presents the design of the CodeBlue move generator, including FPGA issues critical to the move generator architecture. The fundamental principle of *propagation through squares* is maintained. However, a more efficient method of propagating piece information is introduced. In Section 4.1, a summary of design goals and characteristics is made. In Sections 4.2 and 4.3, chessboard and chess square operations are detailed. Arbiter design and move masking follow in Section 4.4 and 4.5. A brief explanation of how special chess moves are implemented is given in Section 4.6. The state machine that controls the chessboard is overviewed in Section 4.7. The bus interface controller used to connect the hardware move generator to the computer's main bus is considered in Section 4.8. Synthesis and implementation of the move generator circuits is the topic of Section 4.9. As a final section to this chapter (Section 4.10), the integration of the CodeBlue move generator to the chess program is discussed.

4.1 Design Goals and Characteristics

The general goal of the project was stated in the Introduction. In Chapter 3, previous move generators were explained. Some important themes shown therein are exploited in this design. Performance evaluation of the CodeBlue move generator is presented in Chapter 5. A crucial first step in the design process is to outline important goals and characteristics that must be supported by the hardware.

- The design must return chess moves in the best possible order (best first). This was mentioned in Section 2.2. A deviation from the MVV/LVA ordering is used in CodeBlue. A small improvement is obtained when using the most valuable victim / *most* valuable aggressor (MVV/MVA) move order, during full–width search. Results that support this are presented in Section 5.2. As seen in Section 2.1.1, proper move ordering reduces the size of the search tree. This has the same effect as making the hardware faster;

- The move generator must support the use of software move ordering heuristics as well. In this case, the killer heuristic and transposition table require specific operations from the move generator. This is explained in Section 4.7;

- The hardware design should intrinsically be the fastest possible. This includes reducing propagation delays and reducing the number of logic levels required to implement a given logic function. These optimizations are done once the architecture and circuit design are done;

- Prior to optimizing the coded circuits, the choice of design architecture must also be made to utilize the fewest logic gates as possible. The same concerns are also targeted at limiting the amount of routing resources used by the design.

4.2 Chessboard Representation

The chessboard is an 8×8 array of chess–square circuits. The chess square circuit is detailed in Section 4.3. In this section, chessboard–level considerations are given. Contrary to the *brute force interconnect* communication method, the *propagation* method implies that a chess square is only connected to its eight (or less) neighbours. Knight lines are the only exception to this rule. Neighbour square communication and knight lines can be seen in Figure 4.1. It is important to note that the arrows shown in the figure do not imply two sets of buses between chess squares. A bus is a transmitted signal *and* and received signal, depending on the point of view. For example, each of the arrows emanating from the c3 square is seen as received signal for each of the squares at the end–points of the arrows. The inverse observation can be made for the g7 square: each of the received signals is a transmitted signal from its origin square. Each square is connected according to the pattern shown in the figure. Edge squares obviously do not have as many interconnects as center squares. The details of the five–bit neighbour bus and two–bit knight bus are given in Section 4.3. These busses are used to indicate that a given chess piece radiates an attack in a given direction. The capabilities (type) of piece determine which lines are asserted.

Operations that target a particular square can select the intended square using

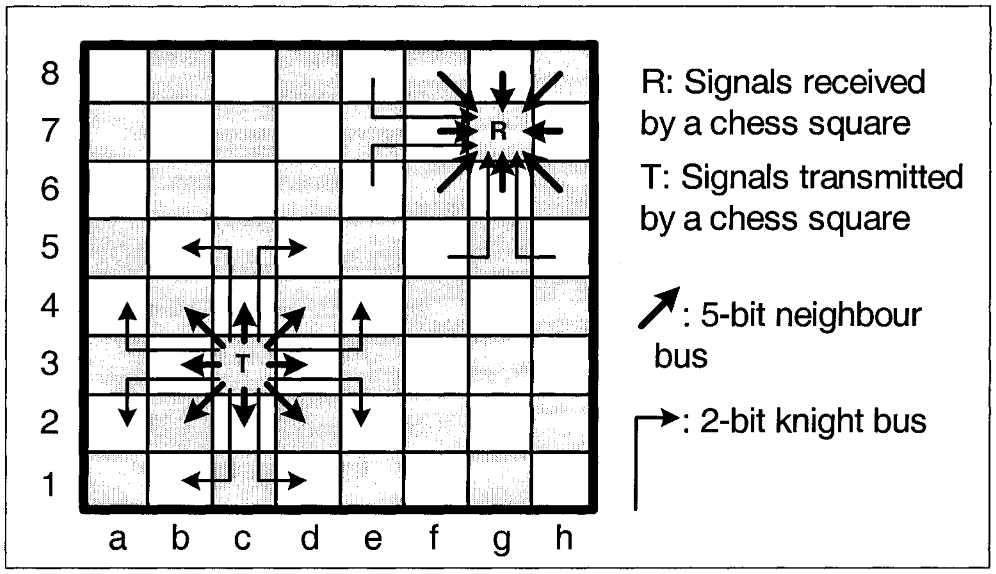

8
7
6
5
4
3
2
1
a b c d e f g h
R
T
R: Signals received by a chess square
T: Signals transmitted by a chess square
: 5-bit neighbour bus
: 2-bit knight bus

Figure 4.1: Chessboard signals. Each square is connected to its immediate and knight–reachable neighbours.

Table 4.1: Common signals to all squares.

Bus or Signal	**# Bits**	**Purpose**
state–mode	3	Current instruction to perform (see Table 4.2).
mask–mode	2	How to affect the mask bits (may depend on the square–select pairs, see Table 4.2).
white–to–move	1	White to play (1) or black to play (0).
write–bus	4	Four–bit piece value to be written into a square (if SS1 is asserted). This bus is also used to update the five–bit depth register (the 5th bit is sent through white–to–move).

one of two square–select pairs. The first pair, SS1, has two 3–to–8 decoders, one to select a row and another to select a column. A square is selected when both row and column signals are asserted. The second square–select pair, SS2, is similar to SS1 with the addition of an enable control signal. In this case, SS2 can be used to signal optional information to a square. For example, when an en–passant pawn capture is possible, the victim pawn is informed of this via SS2. Global information such as state–mode, mask–mode, white–to–move and the write–bus must be routed to all squares. These signals are explained in Table 4.1. The fanout required by these signals is unacceptably large, due to the 64 chess squares. To reduce signal loading and delay in an FPGA implementation, buffers are added to drive groups of eight squares.

Table 4.2: State–mode and mask–mode instructions.

Instruction	Value	Description
state–mode		
SM_FV	000	find–victim
SM_FP	001	find–pivot
SM_FA	010	find–aggressor
SM_IDLE	100	do nothing
SM_DAAA	101	disable–almost–all–aggressors (the square selected by SS2 is not disabled)
SM_W	110	write piece register
SM_WD	111	write depth register
mask–mode		
MM_EAV_EAA	00	enable all victims / enable all aggressors
MM_DV_EAA	01	disable victim (selected by SS1) / enable all aggressors
MM_DA	10	disable aggressor (selected by SS1)
MM_NO_CHANGE	11	mask bits unchanged

Proper constraints must be added to prevent the buffers from being removed during synthesis. The "Keep" command is used in the constraints file to preserve input and output signals connected to the buffers whereas the "don't touch" attribute is used to preserve the buffer components themselves. To make better use of the mode busses, the fifth mask mode (MM_DAAA) was instead coded in the state–mode bus and was labeled SM_DAAA. This is used to generate moves for discovered checks and is explained in Section 4.7. The find–victim and find–aggressor instructions were explained in Chapter 3. The find–pivot instruction was also introduced in Section 3.5.

In Deep Blue, masking and arbitration buses are also used as piece register read/write buses [35]. In this design, the write bus is used to write piece values to the different squares. The piece to be written is sent to all squares, however, only the square selected by the first square–select pair (SS1) writes the piece into its piece register. Another select pair is also used to clear a square's piece register. Thus, making and unmaking an ordinary non–capturing move requires one cycle. Castling moves, en–passant pawn captures or unmaking capturing moves require an extra cycle. The depth register is used to control the move–masking memory depth. Each square's depth register is simultaneously updated with the write–depth–register instruction. This occurs during initialization and also when the search depth is incremented or decremented as a result of a move being made or unmade.

An important feature of the move generator is that piece registers do not need to be read when making and unmaking a move. This decreases the amount of routing

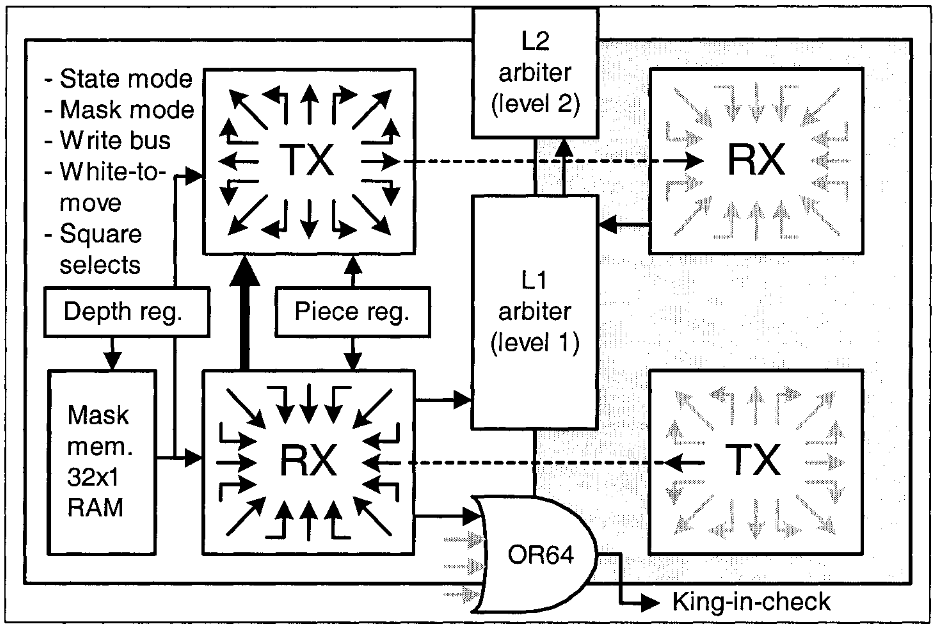


Figure 4.2: Block diagram of a chess square.

resources needed and increases the speed of the design. However, an open–loop design of this nature is more difficult to debug than a closed–loop design. In an open–loop design, reading the state of the chip after an operation was performed cannot be used to find errors. In this case, obtaining an open–loop design is only possible when the moving piece and captured piece (if any) are stored as part of the move word. The bit–fields of a move word are detailed in Section 4.7.

4.3 Chess Square – Minimizing Interconnects

In this section, the chess square circuit and related interconnections are analyzed. Any output signal encountered is assumed as being implemented by a logic function dependant on a given number of inputs. A block diagram of a chess square is shown in Figure 4.2. L–shaped arrows in the transmitter (TX) and the receiver (RX) are two–bit knight buses; straight arrows are five–bit neighbour buses. The thick black arrow from the receiver to the transmitter is used to symbolize that when the square is empty, the transmitted signals for sliding pieces are the propagation of the received signals. A few exceptions involving the propagation of king and pawn bits are used to solve special chess moves; this is shown in Section 4.6. When the square is not empty, incoming attack signals are blocked and the generated signals for the resident piece are instead transmitted. Each square also receives the signals indicated at the top–left of

Table 4.3: Internal extended piece word.

	Bit 5	**Bit 4**	**Bit 3**	**Bit 2**	**Bit 1**	**Bit 0**
	Color	Row/column	Diagonal	King	Pawn	Knight
Neighbour bus:	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	
Knight bus:	$\surd$					$\surd$

the figure. The dashed lines show two of the many connections between transmitters and receivers. One of the 32 first–level arbiters and one of the 16 second–level arbiters are also visible.

During normal move generation, the transmitter outputs of a square obey the find–victim and find–aggressor behaviors discussed in Chapter 3. The transmit lines must have the property of additivity: the union of all pieces must also be capable of being broadcast during the find–aggressor cycle. To achieve this goal, an extended piece word of six bits is decoded from the four–bit piece register. The bit fields are presented in Table 4.3. For example, 111000 represents a white queen and 011111 represents a victim square radiating the union piece. Bits 1 through 5 are sent to the square's eight neighbours, hence the *neighbour bus* designation. The knight bit and the color bit (bits 0 and 5) are sent to the knight–reachable squares, hence the *knight bus* designation. These buses are visible in Figures 4.1 and 4.2.

As a result of using these busses, the chess square circuit does not need two transmitters and double input receivers, such as DEEP BLUE. With the proposed interconnect protocol, a chess square has fewer connection than in the DEEP BLUE design. In Table 3.2 from Section 3.5, it was shown that a DEEP BLUE chess square has 68 input wires and 68 output wires. In the CODEBLUE design, this is reduced to 56 inputs and 56 outputs. Eight neighbour busses and eight knight busses imply a total of $8 \times (5 + 2) = 56$ connections. The reason for the apparent inefficiency in the DEEP BLUE connections can be explained by the fact that the design allows for certain input combinations that are not possible. For example, during a find–pivot cycle, a square cannot simultaneously receive a piece propagation from the enemy king *and* from an aggressor piece in the same input direction. Doubling the inputs and outputs can be avoided if the color bit is used to differentiate between aggressors and victims on the input bus of a given direction. Remembering the *move uniqueness* property expressed in the VM* (Schaefer *et al.*) move generator could not be more pertinent at this point.

During normal find–victim and find–aggressor cycles, input piece color is unnecessary. However, during the find–pivot cycle, opposing colors that align properly on the pivot square will indicate a checking move. These inter–square busses also create a more uniform interconnect pattern and maximize information distribution. The find–check operation is therefore a find–pivot cycle followed by a find–aggressor cycle.

To prevent this find–aggressor cycle from generating moves that are not checking moves, a four–bit find–aggressor–enable memory in each square inhibits the piece type outputs according to the previously executed find–pivot cycle. For example, if the previous find–pivot cycle found a checking move along a diagonal line, the rook output will not be asserted during the find–aggressor cycle from this pivot square. Even if a rook can land on the pivot square, it is not a checking move because of the diagonal–check constraint found during the find–pivot cycle. This four–bit memory has a depth of 32 in order to function correctly given the depth–first search–tree algorithm. Because the king cannot be an aggressor for a checking move, the king output does not need a disable bit. This allows the memory to be reduced from five bits to four bits in width (see Table 4.3). For simplicity, the find–aggressor–enable memory is not shown in Figure 4.2.

When an aggressor piece occupies the pivot square, the find–aggressor cycle is not executed. This situation corresponds to a potential discovered check. In such a case, the move returned indicates a square from which a friendly piece could un-block the checking lane. For circuit optimization reasons, discovered check pivot squares are generated after direct checking moves. In order to obtain a full move for a discovered check, destinations for the pivot piece must be generated. In these cases, a special mode allows all other friendly pieces to be disabled, thereby generating only moves for the selected piece (SM_DAAA, disable almost all aggressors). A few exceptions exist where the discovering piece does not open the checking lane: a pawn move with a vertical checking lane or a pawn capture when the checking lane is diagonal (in the direction of the pawn capture). These rare exceptions affect only move ordering and are not resolved.

It is important to note that checking moves do not follow the MVV/MVA move ordering that is implicitly exhibited by the move generator. Therefore, when entering a node for the first time, a loop to find checking moves will first be executed. The mask bits will mask off generated moves in a non–regular manner (when a pivot square is exhausted, it will be marked as finished). When entering the normal move generation loop, if the mask bits are not reset, non-checking moves that land on former pivot squares will not be generated. The solution to this is to reset the mask bits between the two phases and ignore checking moves when they are generated during the normal phase.

Because the generated moves are pseudo–legal, a verification of the king–in–check status is performed after a move is made. This is accomplished with a 64–input OR gate that combines the king–in–check status of each square. This logic gate is visible in Figure 4.2. During the first find–victim cycle of a new illegal node, the output of the OR gate informs the state–machine that an aggressor can capture the victim king, hence the illegal position. The software then backtracks and generates another move. In reality, the 64–input OR gate is constructed with $16 + 4 + 1 = 21$ LUTs (a LUT has four inputs). The use of the 64–input OR gate could be avoided if the king was added as a possible victim during find–victim arbitration. The king would have the highest priority and once a new node is reached, the first find–victim cycle would identify an attacked king. This was not done in CodeBlue because the three–bit arbitration bus has its eight possible values assigned. This can be seen in Table 4.4 of Section 4.4. The idea for this optimization was proposed by Evans [24].

4.4 Arbiter Design – MVV/XVA Move Ordering

In order to locate the best aggressor, the best victim or the best pivot square during their corresponding instructions, an arbitration network is used. Each square may or may not have a value to arbitrate. An arbiter circuit is responsible for sending the best of two squares' values to the next level of arbiters. The values in question are the square's priority level and its coordinates, three bits and six bits respectively. Because the chessboard has 64 squares, a six–level binary tree of arbiters is required. For example, during a find–victim instruction, arbitration priorities of attacked squares are presented to the arbitration tree. A square that was not hit by any attack signals has nothing to arbitrate. The location and value of the most–valued–victim is therefore obtained at the output of the final arbiter at the sixth level. The priority values associated to the different instructions are presented in Table 4.4. There are 32, 16, 8, 4, 2 and 1 arbiters on the 1st, 2nd, 3rd, 4th, 5th and 6th levels of the network, respectively. A binary tree arbitration structure also has shorter propagation delays than a row/column–based topology, such as the Deep Though and Deep Blue arbitration networks.

In Table 4.4, "nothing" signifies that a square was not hit by any attack signals. This corresponds to the lowest possible priority. When priority level 0 propagates to the output of the sixth level arbiter, the given instruction has not found a square or piece. During find–victim, this implies that all victims have been searched and that

Table 4.4: Arbitration priority levels.

Priority	**F–Vic.**	**F–Agg. MVA**	**F–Agg. LVA**	**F–Pivot**
7	Queen	Queen	King	Queen
6	Rook	Rook		Rook
5	Bishop	Bishop	Pawn	Bishop
4	Knight	Knight	Knight	Knight
3	Pawn	Pawn	Bishop	Pawn
2	Empty promotion		Rook	Empty
1	Empty	King	Queen	Disc. check pivot
0	Nothing	Nothing	Nothing	Nothing

the current node is finished. During find–aggressor, this implies that all aggressors for the current victim have been returned and that a new victim must be located. During the find–pivot cycle, a priority level 0 reaching the final arbiter output implies that no more pivot squares exist. The priorities for the find–pivot cycle are very similar to those of the find–victim cycle. In the find–pivot cycle, direct checks that capture high–valued victims have the highest priority. Direct checks involving the move of an aggressor to an empty square follow. Because discovered checks are located on pivot squares containing aggressor pieces, discovered–check pivot squares must be returned after all of the direct checking moves. If this is not the case, a pivot square containing a piece that can give a discovered check *and* a direct check could cause a problem: if the discovered check is found first, the square will be disabled and the direct check will not be generated. The following direct checking moves are not generated by the find–pivot instruction: a promotion, castle or en–passant pawn capture that checks the opposing king. These move are instead generated during normal move generation. Their omission affects only the move ordering and not the completeness of the move generator.

A block diagram of the arbiter circuit is shown in Figure 4.3. The typical schematic view of such diagrams was replaced by a text–based diagram to better illustrate the input method used to code the move generator. Depending on the level in the binary tree, an arbiter circuit uses between 8 and 13 LUTs each. Lower level arbiters are optimized by the synthesis software; this is explained in Section 4.9. The arbiter can be instructed to invert the priority levels according to the desired move ordering scheme. For example, to create the most–valuable–aggressor ordering (MVA) during the find–aggressor cycle, invert–priority is not asserted (logic 0). A second ordering for the find–aggressor cycle is obtained when generating the least–valuable–aggressor

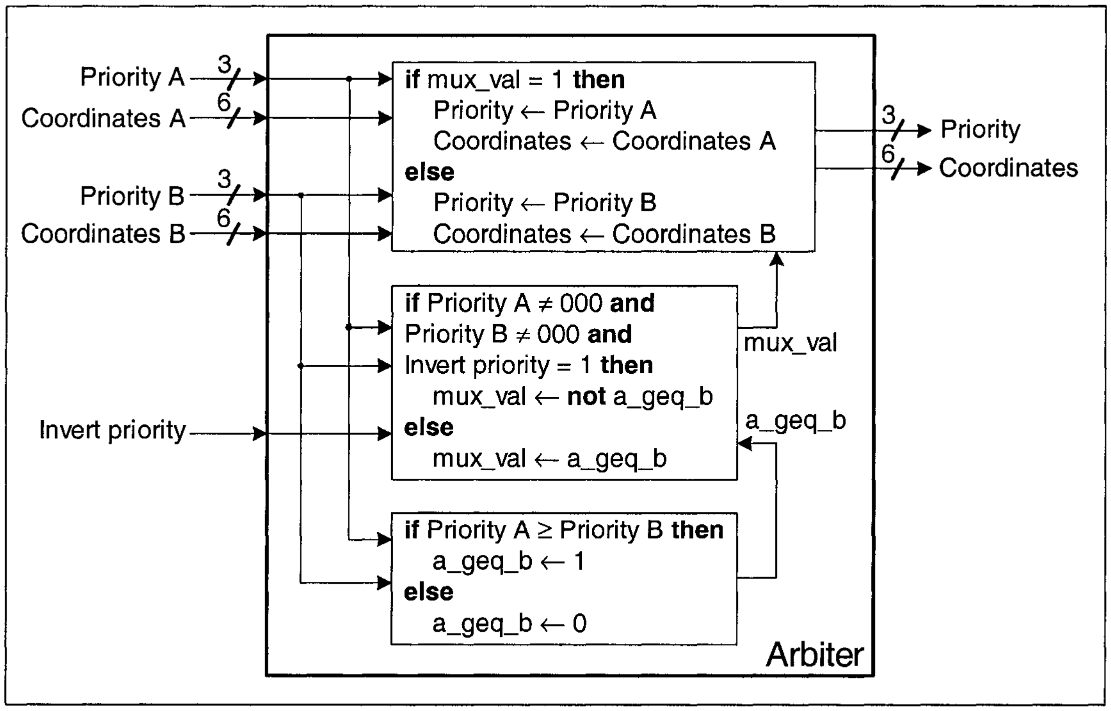


Figure 4.3: Block diagram of an arbiter.

first (LVA). In this case, invert–priority is asserted (logic 1). The invert–priority signal is not asserted during the find–victim cycle, hence the most–valuable–victim (MVV) ordering.

In Section 2.2, it was stated that MVV–MVA ordering is used in full–width search and that MVV–LVA ordering is used in quiescence search. The invert–priority signal is dynamically controlled by the chess program each time a move is generated. The programmable arbiters are thus said to perform MVV–XVA move ordering. The X is used to denote either "Most" or "Least".

In an earlier version of the design, arbiters were distributed in a regular manner throughout the chessboard. This pattern gave highest priority to squares at the top right corner of the board, and lowest priority for squares at the bottom left corner. The consequence of this is that arbitration between squares with the same priority values have their tie broken by location. Therefore, after captures have been exhausted, non–capturing moves are returned in order of their destination square from top–right to bottom–left. This move ordering is not optimal given the positional evaluation function used. In the evaluation function, pieces are given higher scores for occupying the center of the board. It was believed that re–arranging arbiters so that priority was ordered from the center toward the edges and not from top–right to bottom–

left would produce better move ordering. After more tests of this nature (presented in Section 5.2), it was clear that re–arranging arbiter locations would be beneficial. This improvement is labeled *arbiter centrality.* The only hardware penalty incurred when re–arranging the arbiters concerns propagation delays. No additional logic is required.

4.5 Move Masking

The memory resources of the FPGA used to implement the CodeBlue move generator were briefly described in Section 2.4. DistributedRAM is obtained when LUTs are used as random access memory. This type of memory is useful when small quantities of local data need to be stored. BlockRAM is used to store larger amounts of data that do not need to travel throughout the entire chip. The memory capability of LUTs influences the design of move masking logic. In an FPGA, should move masking be performed with last–move–decode circuits such as in the Deep Thought design or with memory–based move masking such as in Belle?

The original Belle move masking method is shown to be the ideal move masking scheme for FPGA move generators. A one–bit, 32–deep synchronous memory in each square is used to memorize mask bits. This memory is shown in the block diagram of a chess square in Figure 4.2. The memory is instantiated using the RAM32x1S primitive and uses two LUTs. The buffered signals used to write piece values are also used to update the depth register in each square. The depth register controls the five–bit address of the move masking memory. The memory has one bit for each ply of the search and is responsible for disabling aggressors or victims as moves are generated. The move masking procedure is explained in the Belle section of Chapter 3. Because the design uses few flip–flops, each square has its own five–bit depth register in order to decrease the amount of routing. It is very unlikely that last–move–decode move–masking requiring fewer that two LUTs per square could be designed in an FPGA.

With a memory–based mask–bit stack, all the logic implied by the Deep Blue move masking decoders is unnecessary. Even with its dedicated routing, BlockRAM is not the best solution to store mask bits in this case. Delays ranging from four to eight ns were observed on signals going–to and coming–from the block memories. With local memory, as described above, these delays are virtually eliminated. This highlights the importance of choosing the appropriate type of RAM resource in a design. However, the BlockRAM could be used to implement a small on–chip transposition table. This is analogous to the level–one cache found in conventional microprocessors.

The move making memory is controlled via the mask–mode bus described in Table 4.2 of Section 4.2. Different instructions, such as disable–aggressor, disable–victim & enable–all–aggressors and enable–all are used to mask moves as move generation progresses in a given chess position. An example of move masking operations is given in Figure 3.3. The depth of the move masking memory could be increased to support deeper searches. In the current implementation, search depth is limited to 32 plies. Given the limited search extensions used in MBChess, very few searches are confronted to this limitation. However, in certain endgame positions, a maximum search depth of 64 would be better. For simplicity, the depth of these memories was limited to 32 in the CodeBlue design.

4.6 Special Chess Moves

In this section, the implementation of chess exceptions is explained. The four special moves in chess concern kings and pawns. These are castling, en–passant pawn captures, promotions and two–square pawn advances. The five–bit omnidirectional outputs allow pawn and king propagations to travel two squares in distance, when necessary. Third and sixth row squares propagate the pawn bit so that the fourth and fifth rows can see double square pawn advances. This is done in both vertical directions in order for both the find–victim and find–aggressor cycles to detect the pawn move. The white–to–move signal is used to ensure that the two–square pawn advance is only valid for pawns on their home row.

Squares f1, f8, d1 and d8 propagate the king bit so that castling destination squares can signal castling moves. These square each receive the corresponding castling status bit. If castling rights are no longer valid for a given castling move, the intermediate square does not transmit the king bit in its transmitter. This is also done in both the left–to–right and right–to–left directions. A special trick is used to ensure that a king does not castle through check or that a king does not castle out of check. Once the castling move has been made, extra kings are written in the source square and in the intermediate king–travel square. An instruction that verifies if the victim king(s) is in check is then used to ensure the legality of the castling move. The extra kings are obviously removed after the castling legality test. These extra procedures are controlled by the chess program.

Horizontal pawn outputs are used for en–passant pawn captures and ensure that this pawn exception is well placed in the move ordering. No extra cycles or software adjustments are necessary for en–passant capture generation. In reality, the chess hardware "believes" that under certain circumstances, a pawn may capture horizontally. The destination row for the move is then adjusted to point to the third or sixth row, depending on the aggressor's color. A pawn is informed that it is an en–passant victim with the square–select–2 signals.

Pawn promotions must also be considered for proper move ordering. Pawn promotions that capture an opponent are treated as an ordinary capture in the move ordering. Therefore, promotions that capture a piece are automatically generated with the other capturing moves. Pawn promotions to empty squares have a priority level greater than a move to an empty square, but lower priority than the captures mentioned previously. In this way, move ordering is almost ideal and does not require special cycles for pawn promotions, as is the case in BELLE.

4.7 Chessboard State Machine

A finite state machine controls the operation of the digital chessboard. The state machine applies the proper combinations of instructions in order to generate chess moves. The state machine is also responsible for maintaining the FPGA's board representation as moves are made and unmade. The state machine is clocked at the bus interface's clock frequency so that only one clock domain is needed for the entire chip. Therefore, depending on the instruction to perform, an appropriate number of stalls are introduced so that propagation delays can be respected. Table 4.5 shows the different commands that can be used by the chess program to control the move generator. These high–level commands are managed by the state machine. For example, for the generate–next–move command, the state machine may use the sequence: find–victim, find–aggressor to locate the next un–searched move. The operation of the chess state machine is detailed using state transition diagrams in Appendix B.

The move array mentioned in the table can be seen as a 32–level stack containing the current move at a each depth. This is analogous to the 32–level stack used for the move masking bit of a chess square. Once a move is generated at a given node, it is returned to the chess program through the main bus. Data transfers are 32–bits in width. Given the bit–fields of a chess move from Table 4.6, only one such transfer is needed.

The "write move" command from Table 4.5 is used to expand a transposition–table's suggested move. This move would normally be returned in its normal order during the move generation of the node. Because of the re–ordering created by this

Table 4.5: State machine commands.

Command	Comment
Write piece register	Four–bit piece value, one write per square
∗Write board state	White–to–move, en-passant and castle bits
∗Reset node	Reset mask bits and move at current depth
∗Reset depth	Depth registers in all squares are set to 0
Disable almost all aggressors	Except for the aggressor denoted by the source coordinate of the current move
Write stall values	The state machine will stall for x extra clocks in the find–victim, find–pivot and find–aggressor states. ($x \in [0, 3]$)
King–in–check?	Verify if victim king is in check (uses one find–victim cycle)
∗∗Unmake move	Unmake the move stored in the move array at previous depth
∗∗Generate or write move	Generate the next un–searched move (normal or checking move, MVV–LVA or MVV–MVA) or write a given move in the move array at current depth
∗∗Make move	Execute the move stored in the move array at current depth

∗: can be combined with other ∗ commands.

∗∗: can be combined with other ∗∗ commands.

Table 4.6: Bit fields of a chess move.

Bits	Used for
31–30	Special: when flag = 0, 00 = king in check, 10 = no moves left. When flag = 1, 00 = normal move, 01 = castling move, 10 = en–passant capture, 11 = promotion.
29–24	Source coordinate of move: three bits for x, three for y.
23–22	Promotion type (if applicable): 11 = queen, 10 = rook, 01 = bishop, 00 = knight.
21–16	Destination coordinate of move: three bits for x, three for y.
15	Flag: 0 = no move stored here, 1 = this is a pseudo–legal move.
14–12	Moving piece type: type of the moving piece (three bits), color is deduced from the board state.
11–9	Captured piece type: type of the captured piece (three bits), color is deduced from the board state.
8	En–passant square valid? 1 = yes, 0 = no.
7–5	En–passant location, x coordinate (if valid).
4	En–passant location, y coordinate (if valid): 1 = on white side of board, 0 = on black side of board.
3–0	Castling permissions (both sides, king and queen side): 1 = yes, 0 = no.

heuristic, it must be written into the move stack at the current depth. This move does not need to be validated because of the matching hash keys (see Section 2.2). The combination of commands also allows the move to be automatically executed after it has been written into the chip.

The killer heuristic's suggested move, however, must be validated. The move may not be valid in the current line of play. Therefore, the move generator is "tricked" into generating moves that land on the killer move's destination square, so that it can be confirmed. To accomplish this, a partial move containing the destination square of the killer move is written into the chip. This move has a flag = 1 to force the move generator to start in the find–aggressor cycle. The aggressors are then returned in sequence until the killer move is matched or until no more aggressors are found for the given destination square. Usually, very few cycles are necessary to determine whether the killer move is valid or not. Because this is the first move generation to be executed when reaching a new node, the mask bits are reset after this procedure. The killer move is ignored when it shows up in its natural order during normal move generation.

When the find–pivot instruction locates a pivot square containing an aggressor piece, a possible discovered check in indicated. The move generator does not automatically generate the moves for the discovered check because of the disorder that would be created in the mask bits. However, once all of the discovered check pivot squares have been returned, each discovering piece can be analyzed. Generating the moves for such a piece involves disabling all of the other aggressors on the board. This is accomplished with the SM_DAAA command on the state–mode bus. Once this is done, normal move generation is stared and only the moves for the intended piece are generated. Before moving on to another piece that releases a discovered check, the mask bits are reset. As was mentioned previously, the checking moves, direct and discovered, are ignored by software when they appear during normal move generation. Had the state machine been more complex, a programmable state machine (micro–code engine) would have been developed.

4.8 PCI Interface

The Peripheral Component Interconnect is the main bus architecture used to connect the FPGA to the host computer. The FPGA is mounted on a card equipped with a PCI edge connector (see Appendix C); the only hardware resource required by the host computer is an empty slot. The technical specification is governed by the PCI

Table 4.7: FPGA move generator performance, 33MHz clock.

Instruction	**#Cycles**	**Instruction**	**#Cycles**
dec. depth, undo move	1, 1 or 2*	find–victim	3
do move, inc. depth	1 or 2*, 1	find–aggressor	3
all writes	1	find–pivot	3

*: Normal, non-capturing moves require one cycle. Castling moves and unmaking captures require 2 cycles.

Special Interest Group and is described in [53]. The FPGA move generator must therefore include a PCI interface to connect it to the computer running MBChess. The interface logic is responsible for decoding read and write commands from the bus. The interface must also support the dynamic memory mapping procedure initiated by the PCI BIOS (Basic Input Output System) when the computer is powered–up.

As seen in Section 4.7, many different commands allow the communication overhead to be diminished. For example, in a single read from the card, the move generator can be instructed to undo the currently stored move, generate and return the next move and execute that move on its hardware chessboard. This simultaneous write–and–read is possible when part of the address is used to send a command rather than address memory locations. Table 4.7 presents the performance obtained for a 33 MHz clock frequency. It should be noted that the master clock for the entire design is that of the PCI bus (33 MHz). The find–victim, find–pivot and find–aggressor states are prolonged according to a stall register to allow for sufficient time to account for propagation delays. Because these stalls can only extend the duration of an instruction for an integer number of clock periods, the device is not used at its maximum speed. Before implementing programmable arbiters and arbiter centrality, the find–victim and find–aggressor had a duration of two cycles. The one–cycle penalty associated to these improvements is more that compensated by the better move ordering and the smaller search trees produced.

The PCI interface was hand coded to support only the most basic operations and uses only 135 LUTs and 85 flip–flops. Parity is generated but not tested. A move consists of one 32–bit double–word therefore no burst transactions are needed. Furthermore, the worst–case latency does not exceed the 16–cycle limitation described in the PCI protocol. This occurs when the unmake–and–generate–move command is executed and the move to unmake is a two–cycle operation. The design does not need to issue retry and always terminates with "disconnect with data". An advantage in not using a core for this application is that custom asynchronous handshaking signals can be created to decrease latency. Write data does not get latched in the PCI interface (the address does) and goes directly to the chess state machine. The chess state machine has access to the PCI address/data bus (used for write commands). The PCI interface does, however, latch the read data from the chess state machine before sending it out on the PCI bus. It should be noted that when the FPGA is re–programmed, the base–address registers are re–initialized and the move generator can no longer be accessed. Each time the FPGA is reprogrammed, the computer must be rebooted in order for the device to be re–memory–mapped.

4.9 Synthesis and Implementation

The FPGA design was done in VHDL [54] and the chess program was coded in C. The chip used is an XCV800–4 and the implementation tools are by Xilinx. A device driver interfaces the FPGA mounted on the PCI card to the chess software. A C program was created to generate the VHDL file responsible for interconnecting 64 instances of chess squares and 63 instances of arbiters. Location constraints were also generated with this program and are used to inform the place and route tool that the chessboard is an 8×8 array. This reduces implementation time and produces a design with better performance. In this case, a 17% speed increase was obtained. It is also advantageous to place the PCI interface near the side of the chip, close to the IO pins. Area constraints were once again used to prohibit the placer from mixing the chess state machine with the chess squares. Place and Route effort levels were set to "highest" (with an extra effort level of 1); three delay–based router cleanup passes and five cost–based router cleanup passes were also performed to increase circuit performance. The entire design uses approximately 10 100 LUTs, 350 32×1 RAMs, 800 flip–flops and has approximately 40 000 connections that must be routed. The multi–pass place–and–route indicates that of the first ten cost tables, cost table 5 yields the fastest design. The mapper was also instructed to map logic to 5–input functions (use F5 MUX). Because of the large amount of combinatorial delays involved in propagating signals from one side of the board (chip) to the other, the find–victim, find–pivot and find–aggressor instructions have a duration of three cycles in the chess state machine. The bit–file representing the entire design is 575 KBytes in size and requires 30 seconds to upload into the FPGA via a parallel–port upload cable. The equivalent gate–count for the entire design is 158 221 gates.

The synthesis software automatically performs logic optimizations due to edge effects. Because the chess square module is the same for all 64 chess squares, the in-

Table 4.8: Maximum fanouts in the design.

Wire (net)	**Fanout**
Clock	903
Reset	464
Eight state–mode buffer outputs (bit 0)	133 or 134 each
Eight white-to–move buffer outputs	between 90 and 92 each
Eight state–mode buffer outputs (bit 2)	69 or 70 each

stantiation program assigns a logic–0 to unused inputs. Unused outputs are connected to unused signals. The synthesis tool removes all logic used to produce an unused output. The software also propagates the logic simplifications brought on by a constant logic level on an input. Because of these logic optimizations, a corner square uses as few as 77 LUTs for logic whereas a center square uses approximately 160 LUTs. The arbiters are also simplified in this manner. The coordinates of a square are composed of signals hardwired to logic–0 or logic–1. These create optimizations in the first few levels of the arbitration tree. This is the reason why most of the first–level arbiters require 8 LUTs and why fifth and sixth–level arbiters require 13 LUTs.

The synthesis tool can also create different implementations of a state machine. In an FPGA, the most efficient coding technique is labeled *one–hot*. In this scheme, each state is represented by one flip–flop bit. Because the state machine can only be in one state at any given time, only one bit is asserted (hot). State transitions involve changing which flip–flop's bit is active. A one–bit–per–state structure facilitates the decoding of a state and allows better use of the abundant flip–flops.

The worst–case fanouts encountered in the design are shown in Table 4.8. Because the chessboard logic cannot be pipelined, very few synchronous elements are used. This explains the relatively small fanout of the clock and reset signals. In general, fanouts exceeding 100 are not encouraged. A high fanout increases the loading on a net and contributes to slower overall performance. One of the state–mode bits would certainly benefit from additional buffers to help drive the many loads it is connected to. It should be mentioned that the clock signal is driven with a specialized clock buffer and that a fanout of 900 is by no means excessive.

In Figure 4.4, a view of the placed design is visible. The graphical primitives correspond to slices and LUTs. The 8×8 array of chess square circuits is clearly visible. At the left of the array, the chess state machine is visible in light grey. The PCI interface is also visible in dark grey; it is separated in two regions above and to the bottom–left of the state machine's logic. The IO pins are also visible in the

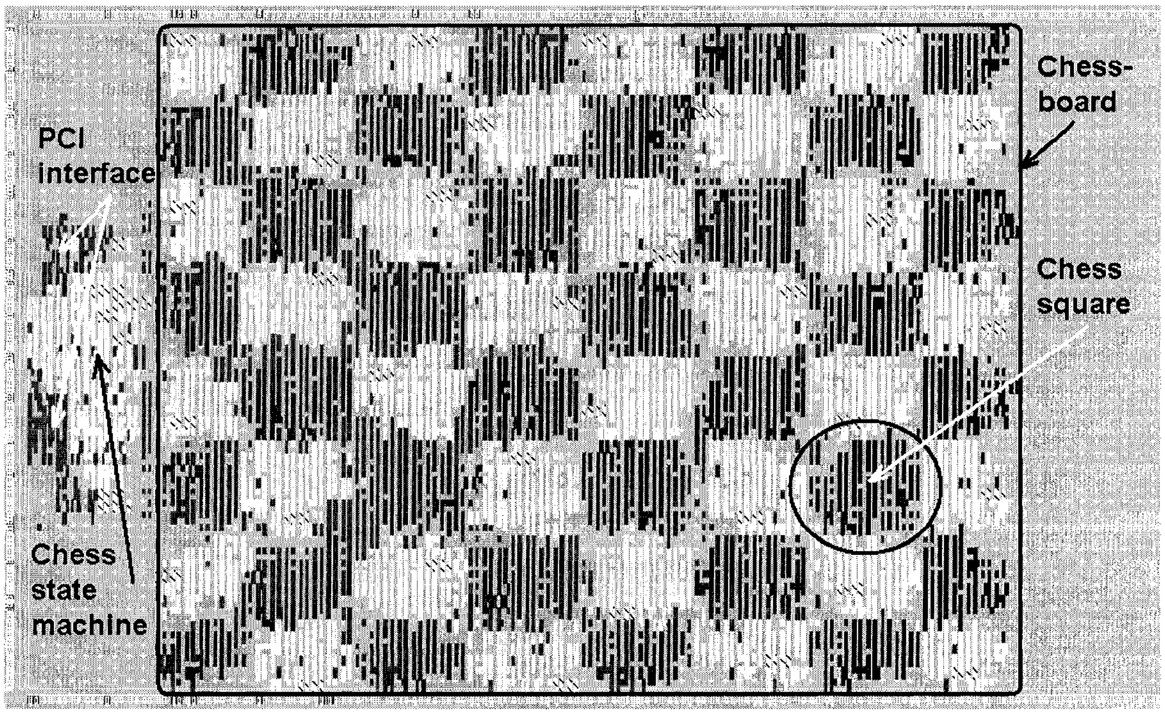


Figure 4.4: Mapper view of the FPGA move generator.

periphery of the device. Horizontal and vertical routing is used throughout the chip. Because of this, signals that must travel diagonally suffer from additional delays. In addition to faster logic and routing, the diagonal routing found in Virtex–II devices would be another advantage contributing to a faster move generator.

4.10 Integration to MBChess

In this section, the integration of the hardware move generator to the MBChess program is explained. As a starting point for the new chess program, named MBChess–CodeBlue, a copy of MBChess is made. The software move generation function is deleted and replaced with appropriate calls to the hardware. This is a simplistic view and does not account for the many modifications implied. The move generator is closely linked to the search tree algorithm and to the move ordering heuristics. Before expanding on chess program details, a brief word on the device driver is given.

The move generator is connected to the computer's main bus using a PCI interface. On the software side, a device driver is responsible for translating software commands to low–level hardware events. Reads and writes are accomplished without any protocol overhead because the PCI card is memory mapped into main address

Table 4.9: Full–width search move ordering in MBCHESS–CODEBLUE.

Order	Type of move
1	Transposition table's suggested move
2	Killer heuristic's suggested move
3	Direct Checking moves
4	Discovered Checks
5	Capturing moves in MVV/MVA order (includes capturing promotions)
6	Non–capturing promotions
7	Non–capturing moves

space. The device driver locates the card based on the traditional device and vendor id, which were assigned arbitrary values. Once the driver determines that a memory–mapped region exists (indicated by base address 0), all subsequent reads and writes are simply accomplished with the equivalent of the assembler–level "mov" instruction. Thus, two functions that can read and write a double–word to the chess card are used to communicate with the move generator. The creation of device drivers is explained in [51].

The main difficulty encountered when integrating the move generator to the chess program deals with the ordering of moves. The full–width move ordering used in MBCHESS–CODEBLUE is presented in Table 4.9. The majority of the chess moves encountered during the search are in priorities 7 and 5. Moves expanded for priorities 1 to 4 are kept in an array so that they can be ignored when they appear in the normal move ordering (priorities 5 to 7). Generating the move for priority 1 does not involve the hardware move generator. This move, when applicable, is read directly from the transposition table. In Section 4.7, generating the killer move was explained. The other moves generated during the killer move's validation are simply ignored. Priorities 1 and 2 can be seen as exceptions and are not implicitly part of the move generator's sequence. Priorities 3 and 4, however, are part of the move generation sequence. The mask bits must be cleared after the killer move validation has been performed. The first move generation sequence begins with a flag indicating that checking moves are requested. The direct checking moves from priority 3 are returned individually. After the direct checking moves are completed, all discovered–check pivot squares are returned before any discovering check can be generated. Each pivot square corresponds to the source coordinate for a potential discovered check. For a given pivot square, all other aggressors are deactivated and the move generator is used to generate the moves for the pivot piece. When a pivot piece has no more moves,

Table 4.10: Quiescence search move ordering in MBCHESS–CODEBLUE.

Order	Type of move
1	Capturing moves in MVV/LVA order (includes capturing promotions)
2	Non–capturing promotions

the mask bits are cleared and the next pivot square is analyzed. This new move generation mode shows how a destination–based move generator can be modified to generate moves for a given source piece instead.

Once priority 4 is finished, the mask bits are once again cleared and the normal move generation sequence is ready to begin. Priorities 5 to 7 are all part of the same sequence and require no additional software control other than setting the arbiter mode. In full–width search, the arbiters are in MVV/MVA mode. It should be noted that even though the bit–fields for a chess move support the promotion of a pawn to the four possible piece types, the hardware only returns promotions to queen. Once the software receives a promotion move, it is responsible for generating the three other promotions. However, the hardware has the ability to make and unmake all types of promotions.

During quiescence search, move generation is much simpler. Only capturing moves and promotions are expanded. Move generation priorities in quiescence search are shown in Table 4.10. Priority 1 is equivalent to priority 5 from the previous table, with the exception of the capturing order. In quiescence search, MVV/LVA ordering is used. Because the move generator can return capturing moves explicitly, the positional evaluation function is used strictly for evaluation. This is not the case in the software–only version. It was shown in Section 2.1.5 that the generation of capturing moves can be integrated to the positional evaluation function. When the value of the capturing piece becomes 0 and promotions are finished, the quiescence search move generation is finished.

The new move generation function is thus responsible for managing the state of the hardware move generator. With the exception of pivot squares, each move is returned individually. When a beta–cutoff occurs, no unused moves have been generated.

Functional verification of the hardware move generator was performed using the node counter. A search performed by MBCHESS–CODEBLUE is compared with the same search executed by MBCHESS. In any given position, searches performed by both programs are expected to have the same number of nodes. In order to compare two search trees performed by two different move generators, the following factors must be considered:

1. The search must be deterministic, i.e. not random;

2. The search trees produced must be identical in size when both move generators are operating properly. Alpha–beta must therefore be removed in favour of a min–max search. This is necessary because of the differences in move ordering;

3. Because move ordering is different, the transposition tables cannot be used;

4. The killer heuristic and quiescence search are not activated;

5. Errors are easier to identify when iterative deepening is not activated.

Many different positions are searched in order to ensure that the design is error–free. Verifying a specific portion of the move generator involves selecting a starting position from which the desired types of moves will be encountered. For example, when a three–ply search is started from TP6 (Appendix A), many different en–passant pawn captures occur in the search tree. If a design error has occurred in CodeBlue, the resulting node counts will differ. In order to reduce the likelihood of one type of error canceling out another error, the search is performed to a depth of four or five. This procedure was repeated in positions where castling moves, promotions and checking moves are to appear in the search tree. Writing an algorithm to perform a 100% functional verification would involve testing the approximately 4 000 ever–possible moves that can be performed on a chessboard. In this case, the laws of probability have helped to drastically reduce the testing procedure.

It should be noted that because the FPGA was physically used to perform this functional verification, timing verification was also implicitly performed. When errors are found, design changes are performed and testing is repeated with the corrected design. This is not possible with an ASIC design. When a discrepancy in node counts occurs between hardware and software move generators, the stall counters in the state machine are temporarily augmented. This allows extra time for signal propagations. If the error is no longer present, a timing error has been detected. If the error persists, a functional error has been found.

Chapter 5

Results and Performance

In this chapter, the performance improvement attributable to a hardware move generator is investigated. A large portion of the tests compare MBChess–CodeBlue with MBChess (hardware–accelerated vs. software–only). All measurements were done using an AMD K6–2 processor operating at 450 MHz with 256 MB of PC100 RAM. The operating system is Windows 2000. As mentioned previously, the chess hardware consists of an XCV800–4 FPGA from Xilinx. It is mounted on a PCI card; the PCI operating frequency is 33 MHz. The FPGA and computer system are of the same generation and represent a suitable combination on which to perform the following experiments. In Section 5.1, test positions are used to evaluate the difference in processing speed between MBChess and MBChess–CodeBlue, given many combinations of heuristics. The principal metric used is the number of nodes processed per second. In Section 5.2, the effect of key improvements on move ordering are measured using the total–nodes metric. A brief motivation to explain the high priority of checking moves in MBChess is given in Section 5.3. In Section 5.4, both programs play complete games against each other and a rating difference is calculated based on the win–ratio. In Section 5.5, both programs play independently on an Internet Chess Server in order to obtain absolute ratings.

5.1 Processing Speed Comparisons

The goal of this section is to benchmark the increase in processing speed obtained when a hardware move generator is used. Different positions are searched to a fixed depth, without any game playing switches such as iterative deepening or draw detection. The performance difference varies depending on which heuristics are activated in the programs. Therefore, results for different combinations of heuristics will be

Table 5.1: Heuristic abbreviations used in Table 5.2.

Code	Name	Description
CF	CHECKS–FIRST	Checks are placed first in move ordering
AB	ALPHA–BETA	Alpha–Beta Nega Scout algorithm
Q	QUIESCENCE	Quiescence search (capture search)
P	POSITIONAL	Positional evaluation of leaf nodes
TT	T–TABLE	Transposition tables used (16 MB)
K	KILLER	Killer heuristic (one killer)
CE	CHECK–EXT	Check extensions (at most 2 plies)

shown. The test positions shown in Appendix A will be used for comparison purposes. These positions are a taken from two lines of play starting from the initial position. Test positions 4, 5, 9 and 10 can be considered as middle game situations whereas the others are categorized as opening game positions. The reason for not having selected test positions closer to the endgame is that the programs do not possess any particular endgame knowledge or algorithms. Furthermore, a program must successfully pass the opening stage before hoping to win a game. Therefore, the opening to middle–game is a critical phase; this is where the benchmarks will be focused. For completeness, the endgame was found to be even more favourable to the hardware accelerated version, for all combinations of heuristics.

All test positions were searched with a depth of six plies with the exception of TP2. With a depth of 6 plies, TP2 consistently yields smaller search trees. It is therefore searched one additional ply in an attempt to balance the experiment. Table 5.2 shows performance comparisons using different combinations of heuristics. MBC refers to MBChess and MBC–CB refers to MBChess–CodeBlue. The codes enclosed in curly braces are abbreviations for the different heuristics. These abbreviations are detailed in Table 5.1.

In summary, depending on which heuristics are activated, performance is increased by a factor of 1.5 to 6 times. Before analyzing the effect of different heuristics on the speed difference between both versions, a comment on the effect of a heuristic in *absolute* terms is important. Each heuristic mentioned in Table 5.1 improves the playing strength of the MBChess program. This is an absolute gain and all heuristics must be activated to make the program play at its best. This must not be confused with the *relative* effect of the heuristic on the comparison being done here. In Section 5.5, all heuristics are activated for best results.

The first observation that can be extracted from Table 5.2 is that the FPGA

Table 5.2: Speed comparison ($\sum$ of TP1 to TP10).

{CF,AB}	**Total nodes**	**Total time**	**#Nodes/sec.**	**Speed incr.**
MBC	3.765 M	76.98 s	48.91 kN/s	
MBC–CB	3.597 M	12.75 s	282.16 kN/s	5.77×

{CF,AB,Q}	**Total nodes**	**Total time**	**#Nodes/sec.**	**Speed incr.**
MBC	4.114 M	120.10 s	34.26 kN/s	
MBC–CB	5.848 M	27.11 s	215.71 kN/s	6.30×

{CF,AB,TT}	**Total nodes**	**Total time**	**#Nodes/sec.**	**Speed incr.**
MBC	3.903 M	67.67 s	57.67 kN/s	
MBC–CB	2.663 M	10.94 s	243.40 kN/s	4.22×

{CF,AB,K}	**Total nodes**	**Total time**	**#Nodes/sec.**	**Speed incr.**
MBC	3.794 M	41.00 s	92.53 kN/s	
MBC–CB	3.025 M	10.59 s	285.63 kN/s	3.09×

{CF,AB,P}	**Total nodes**	**Total time**	**#Nodes/sec.**	**Speed incr.**
MBC	6.702 M	205.74 s	32.58 kN/s	
MBC–CB	6.706 M	84.65 s	79.22 kN/s	2.43×

{CF,AB,P,TT}	**Total nodes**	**Total time**	**#Nodes/sec.**	**Speed incr.**
MBC	4.335 M	141.44 s	30.65 kN/s	
MBC–CB	4.581 M	62.77 s	72.98 kN/s	2.38×

{CF,AB,P,Q}	**Total nodes**	**Total time**	**#Nodes/sec.**	**Speed incr.**
MBC	1.016 M	303.79 s	33.43 kN/s	
MBC–CB	1.197 M	195.07 s	61.37 kN/s	1.84×

{CF,AB,P,TT,K}	**Total nodes**	**Total time**	**#Nodes/sec.**	**Speed incr.**
MBC	3.288 M	71.39 s	46.06 kN/s	
MBC–CB	3.618 M	53.37 s	67.80 kN/s	1.47×

{CF,AB,P,TT,Q}	**Total nodes**	**Total time**	**#Nodes/sec.**	**Speed incr.**
MBC	6.050 M	176.4 s	34.30 kN/s	
MBC–CB	7.519 M	103.95 s	72.33 kN/s	2.11×

{CF,AB,P,TT,Q,K}	**Total nodes**	**Total time**	**#Nodes/sec.**	**Speed incr.**
MBC	4.980 M	110.71 s	44.99 kN/s	
MBC–CB	6.080 M	80.34 s	75.68 kN/s	1.68×

move generator increases the alpha–beta processing speed by a factor of approximately 6×. The differences between {CF,AB} and {CF,AB,Q} can be explained by the fundamentally different way that both move generators operate. The software move generator must generate all moves at a time whereas the hardware move generator returns moves one at a time: no generated moves are wasted when a beta cutoff occurs. This effect is even stronger in quiescence search when only captures are needed. In this type of search, the software move generator must scan each piece to find its capturing moves. In contrast, the hardware move generator implicitly returns captures first (when checking moves are not requested). This explains the difference between 5.77× and 6.30×.

When comparing {CF,AB} with {CF,AB,TT}, the performance difference drops by about 1/3. This can be attributed to the manner in which the transposition table's suggested move is used. When board positions are found in the transposition table and their depth value is not deep enough to be used, the suggested move (TT–SUGG–MOVE) is tried first during move generation. Moves are only generated if the sub–tree returning from the TT–SUGG–MOVE does not cause a beta cutoff. As seen in Section 2.2, no moves are generated when the suggested move is first tried *and* a beta cutoff results. When such cases occur, the advantage mentioned in the previous paragraph is not present. Thus the speed difference is 4.22×.

The same kind of reasoning can be applied to {CF,AB} versus {CF,AB,K}. In this case, the killer heuristic is responsible for a significant reduction in the speed difference. Once again, as seen in Section 2.2, if a beta cutoff occurs as a result of the killer move, no other moves are generated. Therefore, in MBChess, only the moves for the killer piece were generated (using the PieceShowPossibleMoves function). In MBChess–CodeBlue, only the moves that land on the killer's destination square were generated. This reduces the penalty associated with having to generate all moves at a time and is responsible for the somewhat lower 3.09× factor.

Perhaps the easiest effect to explain is that of {CF,AB} versus {CF,AB,P}. In this case, positional evaluation is added to each terminal node in both programs. This has the effect of adding a constant overhead in both programs and thus diminishes the effect of faster move generation. The fact that the speed difference drops from 5.77× to 2.43× indicates that the positional evaluation function is more computationally expensive than the move generator. Would *An FPGA Positional Evaluator for the Game of Chess* have instead resulted in an even grater performance increase?

Another interesting comparison is {CF,AB,P} versus {CF,AB,P,Q}. As seen in Section 2.1.5, when positional evaluation is activated in MBChess, capturing moves are easily calculated because of the similar scanning done in both tasks. In MBCHESS–CODEBLUE, capturing moves used in quiescence search come from the move generator. An interesting test would be to use the quiescence function of MBCHESS in MBCHESS–CODEBLUE. However, because a hardware positional evaluator would not be designed with a sequential scanning approach, this has not been tried. A move generator capable of generating only capturing moves first is therefore needed. Because capturing moves are returned by MBCHESS's positional evaluation function at a low cost, the performance difference between both programs drops from 2.43× to 1.84×.

The total–nodes column from Table 5.2 also reveals a peculiar behavior. When using the {CF,AB}, {CF,AB,TT} and {CF,AB,K} heuristics, MBCHESS–CODEBLUE searches fewer nodes than MBCHESS (9.285 M compared to 11.462 M). However, with heuristics {CF,AB,Q} and in the second half of the table, the software–only version searches fewer nodes (23.783 M compared to 28.843 M). This can be explained by the sensitive nature of the alpha–beta algorithm to move ordering for a given board position. Since both programs do not have exactly the same move ordering, differences are inevitable. For this reason, the nodes–per–second metric is more appropriate. However, the total–nodes count must be verified to ensure that both move ordering methods are comparable.

5.2 Move Ordering Improvements

In this section, the effects of previously mentioned improvements that affect the ordering of moves will be examined. First, results showing the benefits of the most–valuable–victim/*most*–valuable–aggressor (MVV/MVA) move ordering method presented in Section 4.1 will be benchmarked. Second, the effects of the arbiter centrality improvement presented in Section 4.4 will be tested. It will also be shown that most–valuable–victim/*least*–valuable–aggressor (MVV/LVA) move ordering is beneficial during quiescence search. Thus, the use of programmable arbiters that can switch between both methods is motivated here.

Table 5.3 shows node counts for both MVV/LVA and MVV/MVA schemes during full–width search. The arbiter location–priority (centrality) improvement was also tested with the better of the two previous methods. This is the third set of results in the table. It should be noted that these results were compiled with the ALPHA–BETA, CHECKS–FIRST and POSITIONAL heuristics enabled. The test positions used are presented in Appendix A. All ten test positions were tested with a search

Table 5.3: Centrality and MVV/MVA improvements.

#Nodes	**TP1**	**TP2**	**TP3**	**TP4**	**TP5**
MVV/LVA	636503	3040625	271740	673401	1809676
MVV/MVA	572645	1666763	261241	720472	1539412
MVV/MVA+centrality	465356	994388	164072	707723	1205886
#Nodes	**TP6**	**TP7**	**TP8**	**TP9**	**TP10**
MVV/LVA	711198	1117574	1359077	2153598	487688
MVV/MVA	651647	958710	1242337	2520480	492737
MVV/MVA+centrality	353335	647381	856020	1040153	271683

Results	**#Nodes**	**#Nodes/sec.**
MVV/LVA	12.261 M	67.8 kN/s
MVV/MVA	10.626 M	68.8 kN/s
MVV/MVA+centrality	6.706 M	69.2 kN/s

depth of six plies with exception of TP2, which was searched with a depth of seven plies. The first column of the results section of the table is the sum of all ten node counts for each test.

Table 5.3 shows that each improvement decreases the total node count, thus producing a smaller search tree for the same depth. This can be explained by beta cutoffs that occur earlier in node expansion, which are the result of better move ordering. For the ten test positions tested here, during full-width search, MVV/MVA produces a 13% smaller search tree than MVV/LVA. If this is combined with the arbiter centrality improvement, the resulting search tree is 45% smaller, a noticeable improvement. It is also interesting to notice a subtle improvement in nodes/second processed as the tree size decreases. It should be noted that because of these two improvements, the cycle times for the find–victim, find–pivot and find–aggressor instructions were increased from two to three. This decrease in design speed is more that rewarded by the reduction in search tree size.

However, MVV/MVA move ordering is not consistent with the anticipated move ordering needed in quiescence search: MVV/LVA. For these capture extensions, it is easy to see that a capture exchange sequence is usually best performed when capturing with the least valued piece first. Table 5.4 reveals which move ordering method is best during quiescence search. For these results, full–width move ordering is set to MVV/MVA, as was found previously. The heuristics activated are the same as before with the addition of QUIESCENCE and T–TABLE. In quiescence search, a 9% reduction in the amount of quiescent nodes is gained when using MVV/LVA move ordering (1.63 million nodes vs. 1.80 million nodes). Not shown in the table is a slight

Table 5.4: MVV/LVA improvement in quiescence search.

Quiesc. move ord.:	**#Nodes**	**TP1**	**TP2**	**TP3**	**TP4**	**TP5**
	int. nodes	191544	781235	121626	187840	901421
MVV/MVA	qui. nodes	68984	470872	71880	84118	340181
	int. nodes	191605	779200	121632	187819	901589
MVV/LVA	qui. nodes	65902	446943	66411	74420	319056
Quiesc. move ord.:	**#Nodes**	**TP6**	**TP7**	**TP8**	**TP9**	**TP10**
	int. nodes	175717	148676	581122	344639	135889
MVV/MVA	qui. nodes	71527	89118	388586	82711	131428
	int. nodes	175714	148674	580609	340246	134501
MVV/LVA	qui. nodes	67195	62222	337546	73658	119155

Results	**#int. nodes**	**#qui. nodes**	**%quiesc.**
MVV/MVA	3.57 M	1.80 M	50.407 %
MVV/LVA	3.56 M	1.63 M	45.837 %

increase in nodes–per–second processed when using MVV/LVA in quiescence search: 104.5 kN/s to 106.4 kN/s. It is also interesting to notice that the number of internal nodes is essentially unaffected by the choice of quiescence move ordering. In both cases, the total number of internal nodes is roughly 3.5 million nodes.

In conclusion, it was shown that most–valuable–victim/*most*–valuable–aggressor is better than most–valuable–victim/*least*–valuable–aggressor move ordering during full–width search. However, in quiescence search the opposite is true. Therefore, programmable arbiters that can be instructed to do either scheme are used to obtain optimal performance (this concept is labeled MVV/XVA). When combining the centrality–of–arbiters improvement with programmable arbiters, move ordering is much improved and contributes to smaller search trees. Smaller search trees take less time to search and with a fixed amount of time, deeper searches are possible. Deeper searches improve the quality of play, something that will be measured in the MBChess–CodeBlue vs. MBChess section.

5.3 Checking Moves and Move Ordering

Move ordering was shown to play a key role in the efficiency of the alpha–beta algorithm (Section 2.1.1). In this section, a brief experiment shows that the choice that was made concerning the ordering of checking moves is adequate. In Table 2.3 from Section 2.2, it was stated that checking moves are searched before capturing moves, after the transposition table and killer's suggested move. The results from Table 5.5

Table 5.5: Checking moves before capturing moves, results.

Move ordering:	**#Nodes**	**TP1**	**TP2**	**TP3**	**TP4**	**TP5**
Checks before captures	int. nodes	238.6 k	610.5 k	106.0 k	133.5 k	242.9 k
	qui. nodes	89.6 k	250.5 k	60.0 k	64.8 k	144.2 k
Checks after captures	int. nodes	240.6 k	758.1 k	104.6 k	133.3 k	242.6 k
	qui. nodes	87.8 k	261.0 k	62.9 k	65.7 k	146.0 k
Move ordering:	**#Nodes**	**TP6**	**TP7**	**TP8**	**TP9**	**TP10**
Checks before captures	int. nodes	117.3 k	337.2 k	389.3 k	284.2 k	260.3 k
	qui. nodes	40.1 k	104.8 k	219.7 k	99.9 k	168.1 k
Checks after captures	int. nodes	116.6 k	334.5 k	385.8 k	282.2 k	256.3 k
	qui. nodes	40.3 k	100.2 k	217.5 k	101.8 k	188.1 k

Results	**#int. nodes**	**#qui. nodes**	**total nodes.**
Checks before captures	2.720 M	1.242 M	3.962 M
Checks after captures	2.855 M	1.271 M	4.126 M

indicate that when checking moves are searched before capturing moves, search trees have approximately 4% fewer nodes. However, this is not the most important observation. In more general terms, the search trees for nine of the ten test positions are virtually the same size. When considering TP2, searching checking moves before capturing moves reduces the number of nodes searched by approximately 20%. Other than the test positions from Appendix A, a few other test positions were also verified for consistency. As in the test positions used here, in most situations, ordering checking moves before or after captures produces similar node counts. In a few cases, searching checking moves before captures reduces the number of nodes searched. The tests were performed with MBChess with all heuristics activated. A more exhaustive test suite would be necessary to confirm these results.

5.4 MBChess–CodeBlue vs. MBChess

In this section, the increase in performance of MBChess–CodeBlue will be evaluated by playing complete games against the original software–only version. This testing procedure represents a better way to establish the effects of the faster hardware move generator. Different metrics such as nodes–per–second and total–nodes are important to consider, however, the actual game–playing effects are what is most important. The advantage of having working programs that incorporate most of the popular chess heuristics is that results cannot be disputed for lack of realism.

For all games played in this section, both programs are set to five seconds–per–

move. Therefore, iterative deepening is activated for reasons seen in Section 2.1.6. Furthermore, the draw detection algorithm presented in Section 2.3 will be activated so that repetition draws do not needlessly occur. Both programs run on the same machine under the same conditions. The option to *think* on the opponent's time is not activated thus both programs never execute searches simultaneously. Each program has access to 100% CPU usage for its five–second time slice. The move executed is transmitted to the opponent program using inter–process communication features from the Win32 API. Two *named pipes* are used, one for each direction. When a program is waiting for the other to transmit its move, no CPU time is wasted.

As seen in Section 5.1, the difference in processing speed is dependent on the heuristics that are activated. Therefore, for the competition proposed in this section, the following settings will be tested: {CF,AB,Q}, {CF,AB,TT,POS,Q} and {CF,AB,TT,POS,Q,K,CE}. The {CF,AB,Q} setting was retained because of its large speed difference (6.30×). The second setting represents more realistic playing conditions with the addition of transposition tables and positional evaluation. With these two additional heuristics, the speed difference between both programs was shown to be 2.11×. The third setting corresponds to full game playing mode; the killer heuristic and check extensions are added. This is the setting that is used in Section 5.5 for Internet play. In this section, 100 games are played with each setting in order to determine:

- the rating difference obtained with each setting;
- whether the rating difference varies with the speed difference shown in Section 5.1.

With Equations 2.12 and 2.17 from Section 2.5, results can now be shown and a stable rating improvement can be calculated. These two equations are repeated in Equations 5.1 and 5.2 respectively.

$$\widehat{\Delta\mu} = -400 \log \left(\frac{1}{W_r} - 1 \right) \tag{5.1}$$

$$W_r \pm 1.96 \sqrt{\frac{W_r \left(1 - W_r\right)}{n}} \tag{5.2}$$

Results of the MBCHESS–CODEBLUE vs. MBCHESS competition are shown in Table 5.6. The calculation of $\widehat{\Delta\mu}$ is done using Equation 5.1. For this series of games, each game was started with the first half–move belonging to the set {e3, e4, d3, d4,

Table 5.6: Competition results, MBC–CB vs. MBC.

Label	**Heuristics**	**#Wins**	**#Games**	W_r	**Rating Diff.** $\widehat{\Delta\mu}$
H1	{CF,AB,Q}	88	100	0.88	346.12
H2	{CF,AB,TT,POS,Q}	70	100	0.70	147.19
H3	{CF,AB,TT,POS,Q,K,CE}	77	100	0.77	209.91

c3, c4}. These different starting positions are introduced so that a variety of openings can be tested. For each starting position, each program plays the same number of games as white and black (eight each). Another four games were played from the initial position, for a total of 100 games. Because of the RANDOM property of the evaluation function (Section 2.1.2), the same game is never played twice. Drawn games where both programs have equal material were not considered for the win–ratio calculation. For example, if six drawn games are played this is considered as six half wins, resulting in a win ratio of 3/6. If drawn games are not kept, this equality of play should statistically yield three wins and three losses. This again corresponds to a win–ratio of 3/6. Therefore the omission of drawn games does not affect results. Repetition draws where one side has a material advantage were considered as wins for the program that is up in material. The fact that this rule, albeit non–conventional, is the same for both programs does not affect the validity of the results. This rule was helpful in accelerating the testing procedure. The programs do not have dedicated endgame algorithms, nor can they detect drawn games because of insufficient mating material. Both programs were set at five seconds per move, the default time control for MBChess.

However, Table 5.6 does not reveal any details concerning the stability of the final rating difference. It is important to consider the graph showing the evolution of the rating difference as the number of games played increases. Figure 5.1 shows this graph for the {CF,AB,Q}, {CF,AB,TT,POS,Q} and {CF,AB,TT,POS,Q,K,CE} settings. It should be noted that the first few data points on all three curves are not valid. A win–ratio of 0 causes a division by zero in Equation 5.1, which results in a rating difference of $-\infty$. Conversely, a win–ratio of 1 causes a log(0), which results in a rating difference of $+\infty$. In the corresponding graph, these invalid points have a value of 0. From this graph, the rating differences stabilize somewhat after 70 games. Furthermore, it becomes apparent that the choice of heuristics affects the performance increase attributable to a hardware move generator.

Another method for evaluating the stability of the ratings shown in Table 5.6 is

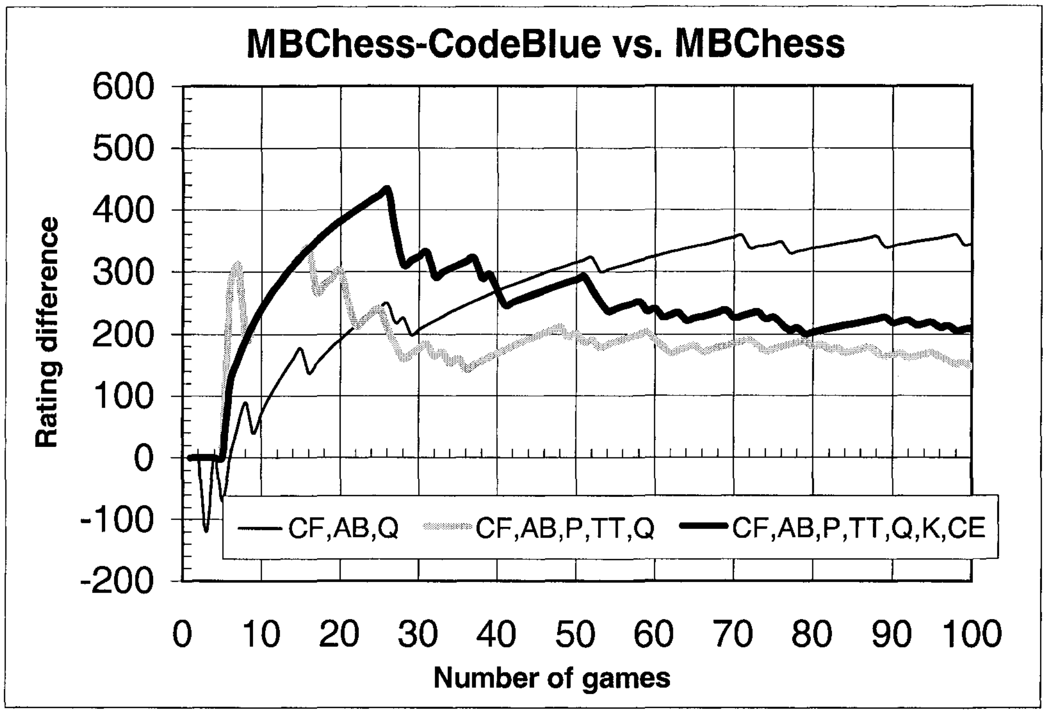


Figure 5.1: Rating difference vs. Games played.

to use Equation 5.2 to determine the error margin on W_r, for each set of heuristics. The results are shown in Figure 5.2.

When considering exclusively the move generator portion of the search procedure, the CodeBlue move generator increases MBChess's playing strength by 346 points. This corresponds to the first entry in Table 5.6. This is the net effect of changing the move generator from software to hardware. It is also crucial to mention that the arbiter centrality improvement mentioned in Section 4.4 is in part responsible for the large increase in strength. Because this improvement favours moves towards the center of the board, a slight positional bias is introduced because of the effect on move ordering. Since the granularity of the evaluation function is purely material count (the POSITIONAL heuristic is not activated), the choice between many equal moves will depend on the choice between "$>$" and "$\geq$" in the alpha–beta algorithm. With the "$>$" used (see Section 2.1.1), the first of all equal–outcome moves will be retained for the best line of play. Since moves are ordered from the middle of the board to the periphery, moves that occupy the middle of the chessboard are favoured, hence the implicit positional effect. Because this gain requires no additional hardware resources, the arbiter centrality was not removed for the comparison.

When considering the second heuristic set–up, positional evaluation is seen as a

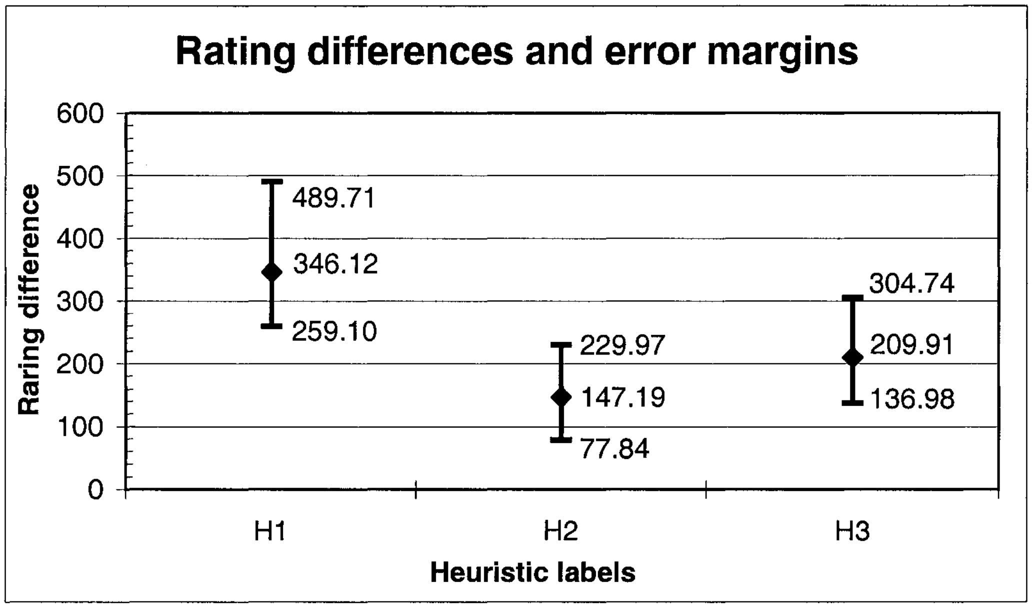


Figure 5.2: Error Margins on $\widehat{\Delta\mu}$. Heuristic labels are described in Table 5.6.

constant overhead to both programs and therefore attenuates the effect of a faster move generator, hence the 147 points in rating difference. The final heuristic set–up shows the addition of the killer and check extension heuristics. From the last two entries in Table 5.2, the addition of the killer heuristic was shown to reduce processing speed from a factor of 2.11× to 1.68×. However, Table 5.6 indicates a rating difference of 210 points, more than 60 points better than without the killer heuristic. This contradiction could be due to the fact the check extensions were also added in this experiment. Another explanation for this is the fact that the test positions used in obtaining Table 5.2 do not cover the endgame. As was stated previously, the speed advantage obtained with the hardware move generator is even greater during this phase of the game. It is therefore probable that many games were won during the endgame phase, where MBChess–CodeBlue's performance increase is even stronger.

To summarize, with all heuristics activated, MBChess–CodeBlue is shown to be 210 chess rating points better than MBChess. As seen from Table 2.4, this represents a little over one full rating category. But where does MBChess–CodeBlue fit in this ranking table? Also, is the relative rating difference obtained here consistent with the difference in absolute ratings obtained independently for both programs? Section 5.5 will attempt answer these questions.

Table 5.7: Rating results, MBC–CB and MBC on FICS.

Program	**Final rating**	**Rating category**
MBChess–CodeBlue	1844	Class A
MBChess	1692	Class B

5.5 Absolute Ratings

In Section 5.4, the performance improvement obtained with the hardware move generator was shown by a competition with the software–only version. In this section, the performance difference is measured in absolute terms; both programs play independently and obtain their own rating. For this experiment, MBChess–CodeBlue and MBChess play on the *Free Internet Chess Server* (FICS) at www.freechess.org. Both programs have all heuristics activated ({CF,AB,TT,POS,Q,K,CE}). The draw detection is activated and the RANDOM property of positional evaluation is turned on. The time controls are mainly 12–0, which allows 12 minutes per game and 0 seconds added after each move (this is known as *Fisher time*). A small portion of games was played with the 5–5 and 2–10 time settings. Since the average duration of the game is roughly equal in all three cases, and that the time is the same for both players (human opponent and program), results are not adversely affected.

Figure 5.3 shows the evolution of the rating of both programs as the number of games played increases. As in Section 5.4, greater stability would be achieved with a higher number of games played. However, from the graph showing rating vs. games–played, it is apparent that MBChess–CodeBlue has a rating of at least 1850. The overall curve seems to be increasing at slow rate. Furthermore, MBChess's rating curve stabilizes somewhat at around 1700. The effect of the CodeBlue move generator is therefore an increase in chess playing ability of 150 rating points.

Table 5.7 shows the final ratings obtained for both programs after 65 games each. However, the corresponding chess rating categories must be considered carefully. The assumption made here is that FICS ratings are comparable to the USCF ratings shown in Table 2.4. Given this assumption, MBChess–CodeBlue is a Class A chess program and MBChess is a Class B chess program. Once more, the improvement brought on by the hardware move generator is noticeable. After 65 games each, the exact rating improvement is 152 points. The R_D, or rating deviation, of both programs reached below the threshold level of 80. A rating deviation below 80 indicates that a reasonably stable rating has been attained.

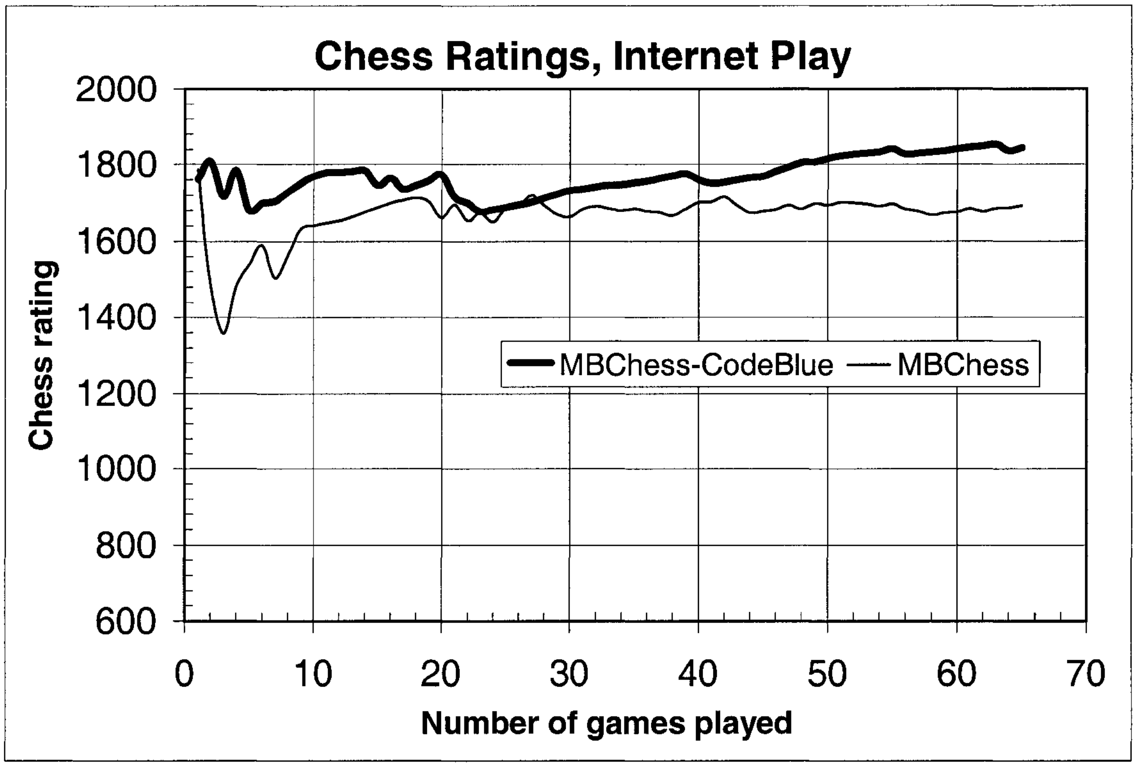


Figure 5.3: Absolute rating vs. Games played.

For comparison purposes, it was reported that a doubling in processing speed increases the playing strength by approximately 100 rating points [33, 38]. From this rough estimation, a relation between the speed increase and the rating improvement can be extrapolated. This is done in Equation 5.3.

$$R_{inc} \approx 100 \log_2 (S_{inc}) \tag{5.3}$$

Where:

- R_{inc} corresponds to the increase in rating;
- S_{inc} corresponds to the ratio of increase in processing speed.

Solving for S_{inc}, Equation 5.4 is obtained.

$$S_{inc} \approx 2^{\frac{R_{inc}}{100}} \tag{5.4}$$

This equation should not be taken literally, nonetheless, it is interesting to evaluate Equation 5.4 with $R_{inc} = 152$ to get an idea of the speed increase provided by the hardware move generator, from a different point of view. This results in a speed increase factor of $S_{inc} \approx 2.87\times$. This is greater than what was reported in Section 5.1. However, the rating difference of 152 points obtained in this section is smaller than the 210 rating difference that was reported in Section 5.4. This could be explained by the fact that the hardware accelerated version's FICS rating could be higher with more games played (slight positive slope in Figure 5.3). In total, the effect of the CodeBlue move generator can be summarized as an increase of approximately 150 chess rating points, almost a full ranking category.

The absolute ratings obtained by both programs in this section could have been much better had the following changes been made to the chess programs:

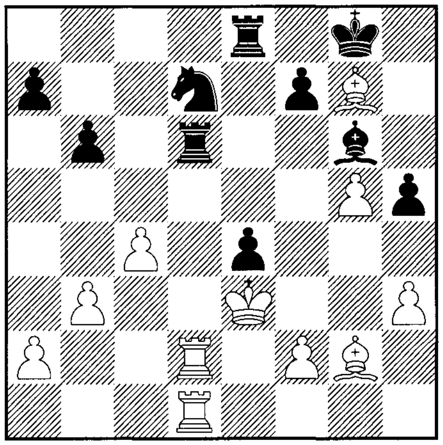
1. Add an opening book;
2. Use a direct Internet interface as opposed to manually translating moves from one chessboard to the other;
3. Better time control. For example, spend more time when the score varies with each iterative deepening search (unstable position);
4. Deep thinking to profit from opponent's time;
5. Add king safety component to positional evaluation function;
6. Add null–move depth reduction.

With these improvements, MBChess–CodeBlue could reach the expert skill level and potentially, the master level.

In this chapter, games were played with a generally short time control. In this section, four to twelve seconds per move were used to produce the time controls mentioned previously. For the MBChess–CodeBlue vs. MBChess competition, five seconds were given for each move. This is far quicker than the three–minutes–per–move time control normally used in tournaments. Suffice to say that it would not have been very practical if games were to last four to five hours each, given that over 400 games were played for the results shown in this chapter. However, at three minutes–per–move, the rating difference would probably be smaller. For example, if program A is faster than program B by a factor which permits A to search one ply deeper than B, the difference between a 5 ply search and a 6 ply search is greater than the difference between a 15 ply search and a 16 ply search. As the search depth increases, differences of one ply become less important. In such cases, positional evaluation becomes the

Players:

∘ yuri (1739)
• MBChess–CodeBlue (1832)
ICS Rated blitz match
freechess.org
2002.04.29



1.d4 d6 2.g3 ♘f6 3.♗g2 e6 4.c4 ♗e7 5.♘f3 ♘c6 6.♘c3 O–O 7.O–O e5 8.d5 ♘a5 9.b3 ♗f5 10.♘e1 ♗g4 11.♕c2 c6 12.♗a3 cxd5 13.♘xd5 ♘c6 14.♘xe7+ ♕xe7 15.e3 ♕e6 16.♘d3 ♗f5 17.♕e2 e4 18.♘f4 ♕e5 19.♕d2 ♖fd8 20.h3 d5 21.♗b2 d4 22.exd4 ♘xd4 23.♖ad1 ♕xf4 24.♗xd4 ♕xd2 25.♖xd2 ♖d6 26.g4 ♗g6 27.♖fd1 ♖ad8 28.♔f1 b6 29.♔e2 ♖e8 30.♔e3 h5 31.g5 ♘d7 32.♗xg7 ♖d3+! White resigns

Figure 5.4: Game #55, move 32, black to play. MBChess–CodeBlue (black) played Rd3+, white resigned. The loss of the white bishop on g7 is inevitable.

crucial factor. This behavior is also found in man vs. machine chess games. In blitz matches, computers have been able to beat the best players in the world since the early 80s[1]. However, it was not until 1997 that a computer was crowned champion in standard time controls (two hours for 40 moves).

To end the present chapter, a game played by MBChess–CodeBlue on the Free Internet Chess Server (FICS) is presented in Figure 5.4. This game was selected because of the strength of the opponent and the final move of the game; a move that causes the opponent to resign. Even though all is not lost for white (opponent), in such a close match–up this represents a sizeable disadvantage. The opponent should have continued the game; the outcome was certainly not predetermined. White's resignation was perhaps due to the psychological effect of losing a bishop: an effect not present in computer chess! It is up to the reader to decide if this is an advantage or not…

[1]More precisely, in 1977, a computer defeated a grandmaster in a five–minute–per–side blitz game.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

An FPGA move generator was successfully developed to increase the plying strength of chess programs. Although some features were specific to the MBCHESS program, the proposed contributions can extend to many other chess programs. Here is a short summary of what was developed:

1. A simpler inter–square communication protocol was shown to require fewer wires than the DEEP BLUE design. Whether the design is implemented in an ASIC or an FPGA, a more efficient use of resources is always beneficial. The interconnect pattern was also shown to handle special chess moves easily.

2. This design shows how traditional BELLE–style move generators can be modified to incorporate popular chess heuristics. Computer chess heuristics are an integral part of chess programs and must also be considered when designing chess hardware. Special instructions were designed to allow the use of the killer heuristic and the transposition table's best move.

3. The arbiters were arranged to prioritize the center of the board when a voting tie occurs. Arbiter centrality improves move ordering because of the center bias found in typical positional evaluation functions. Hence, the size of the search trees is reduced and no supplemental material resources are required.

4. Programmable arbiters which can generate least–valued–aggressors or most–valued–aggressors first are introduced. Labeled MVV/XVA, programmable arbiters are used to perform the appropriate move ordering during full–width and quiescence search. In programs that behave differently than MBCHESS, the

ability to control the ordering of capturing moves is also an advantage, even if the settings are different than those used in this project.

5. The design can generate checking moves separately as well as indicate the location of discovered checks. In addition to the inherent destination–square based move ordering exhibited by BELLE–style move generators, a novel addition to the design also permits moves to be generated from a source square's point–of–view. This source–based move generation is not limited to discovered checks.

6. In MBCHESS, the ability to generate checking moves separately was used in move ordering. The circuitry used to generate checking moves could also be used to perform check extensions during quiescence search. The detection of mating sequences could also benefit from this hardware.

The design, testing and integration–to–MBCHESS phases were accomplished during an 8–month period. The chip was used in real game–playing situations during the final design steps, something not possible with an Application Specific IC. The ease of re–programmability of FPGAs coupled with the high level of abstraction provided by the design and implementation tools have made this project possible given the short time frame. The re–configurability of FPGAs allows a design to be modified in the same way a chess program is modified when an opponent discovers a weakness or when a new heuristic is added.

In more general terms, this thesis has shown how a combinatorial search procedure can be accelerated with the use of digital circuits. As will be stated in the following section, other portions of the algorithm can also be implemented in hardware for even greater gains. The re–configurability of FPGAs becomes essential when considering tasks such as hardware accelerated Boolean satisfiability and hardware accelerated automated theorem proving. In both cases, *each* problem instance is dynamically converted to a digital circuit. Once the device has been re–programmed, the algorithm has a faster, parallel platform on which to solve the problem.

At the time of this writing, the next mainstream Human–Computer chess match up is scheduled for the fourth quarter of 2002. It will pit Vladimir Kramnik, current world chess champion, against DEEP FRITZ. The chess hardware will consist of a multiprocessor supercomputer with no special purpose hardware. The complete system can calculate over 6 million moves–per–second [17]. What DEEP FRITZ lacks in processing speed (when compared to DEEP BLUE), it makes up in chess knowledge and optimized programming. We believe that the proposed FPGA move generator, along with hardware evaluation and search, could further improve the chess-playing strength of DEEP FRITZ.

6.2 Future Work

Obtaining a higher level of performance would involve integrating the evaluation function and search control into the FPGA. With on board transposition tables and suitable databases, a single–chip FPGA grandmaster might be possible. As hard–processors become the norm in upcoming FPGAs, integration of program code to the IC will make the FPGA a complete and even more powerful solution.

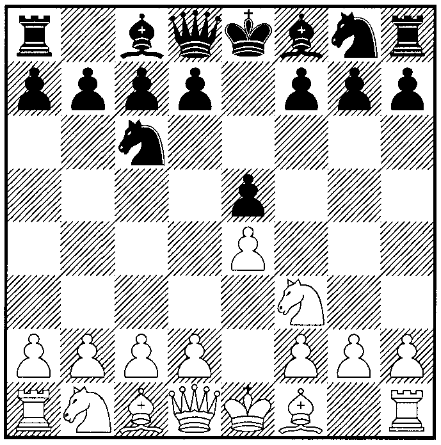
The Virtex–II Pro series of FPGAs could be used to implement a powerful, single–chip parallel chess engine. For example, at the time of this writing, the largest Virtex–II Pro has four PowerPC processors and 125 136 logic cells. This would be sufficient for a four–way chess chip with four hardware move generators and four hardware positional evaluators. Faster logic, diagonal routing and a direct interface between logic and processors would each contribute to increasing the speed of digital chess–playing circuits.

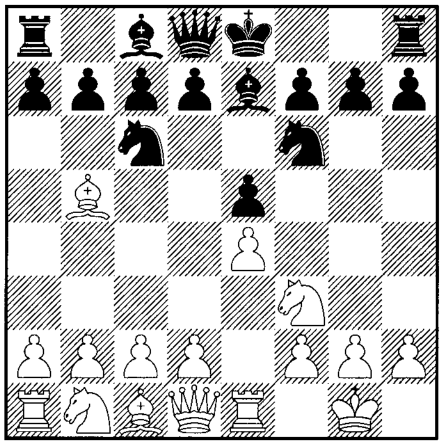
On the software side, many aspects of the MBCHESS program need to be improved. However, the hardware move generator developed here is a solid foundation on which to build a world–class chess–playing machine.

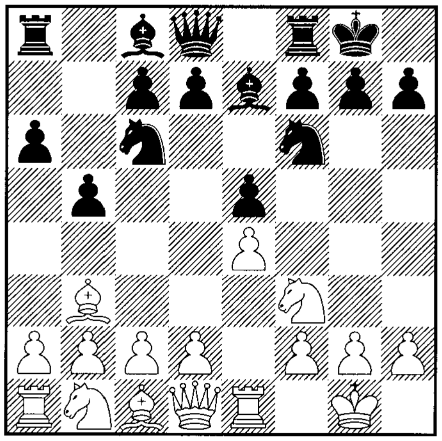
Appendix A

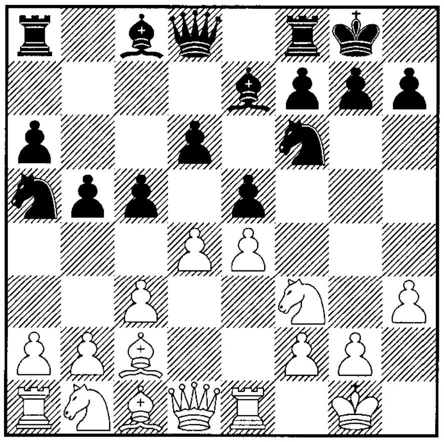
Test Positions

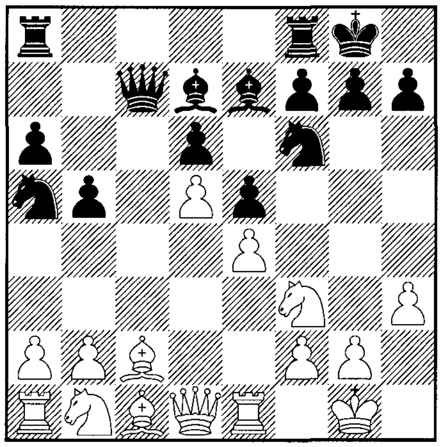
The following test positions are used in Section 5.1 and Section 5.2.

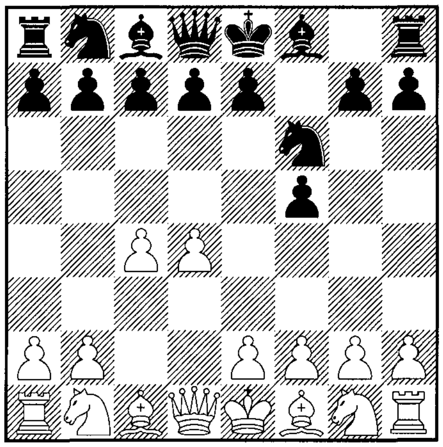

TP1, white, move 3

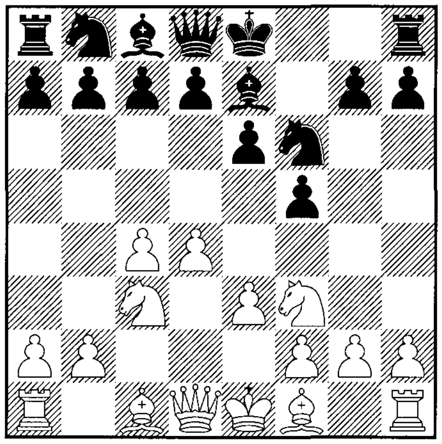

TP2, black, move 5

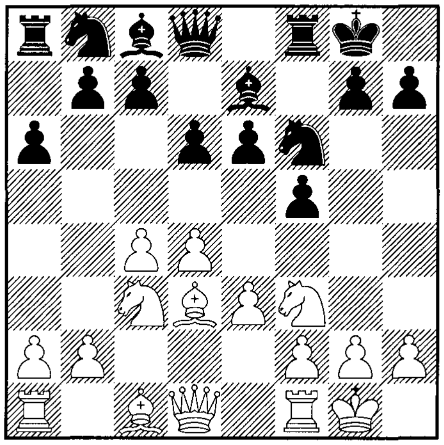

TP3, white, move 8

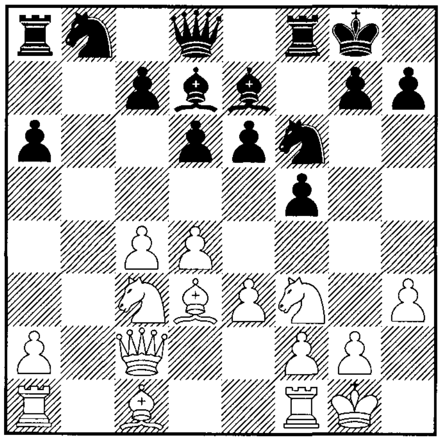

TP4, black, move 11

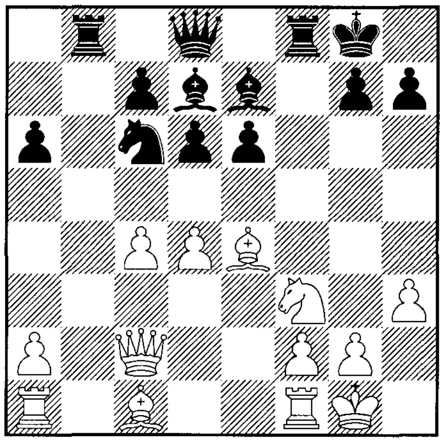

TP5, white, move 14


TP6, white, move 3


TP7, black, move 5


TP8, white, move 8

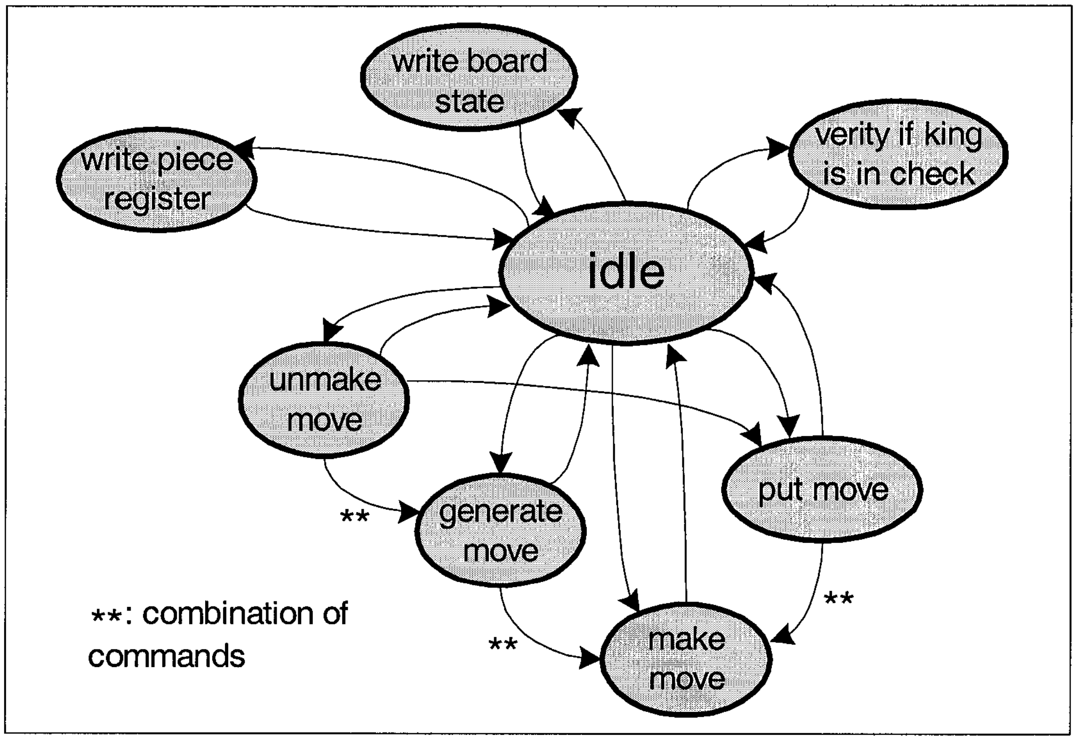

TP9, black, move 11


TP10, white, move 15

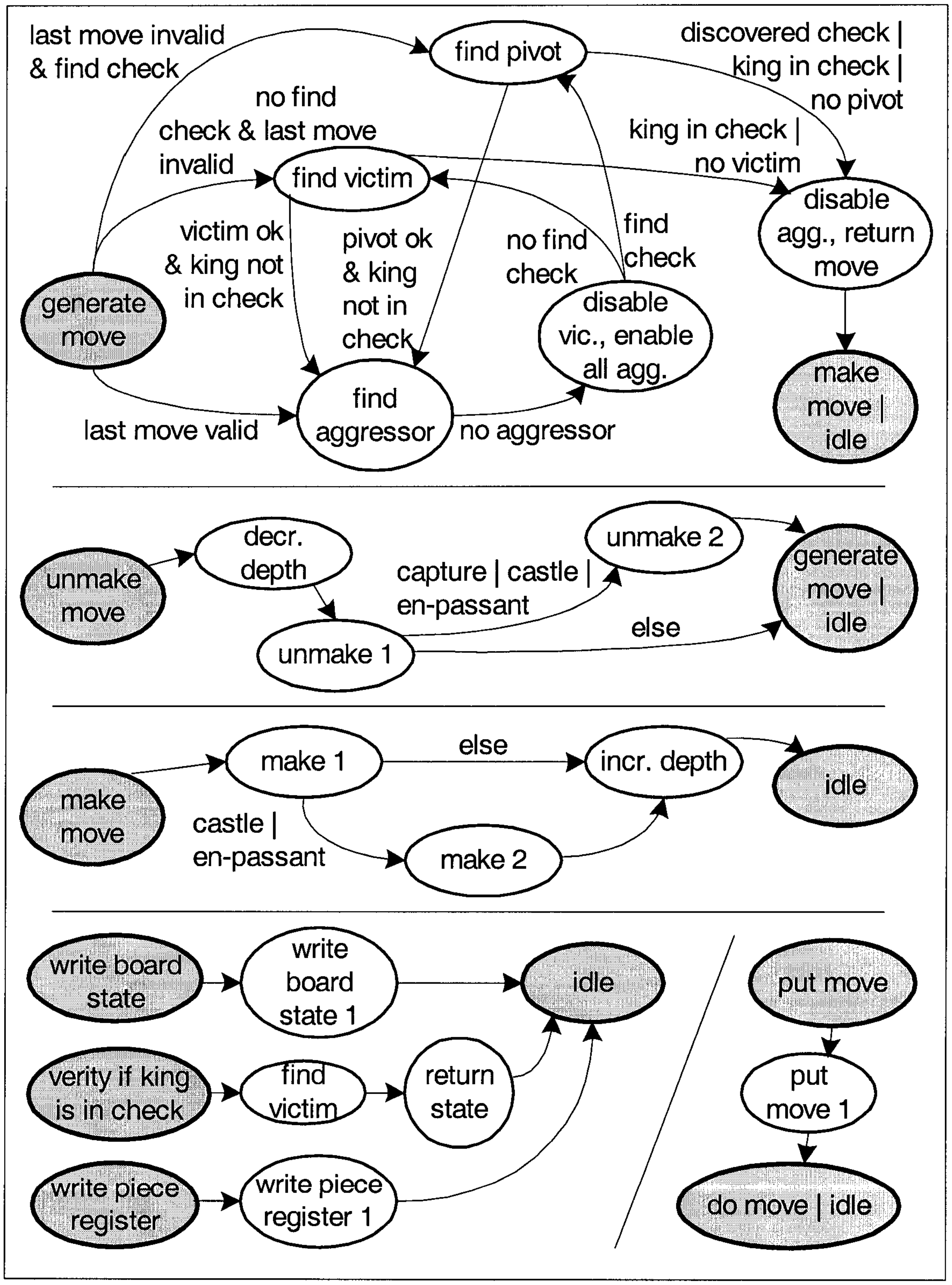
Appendix B

Chess State Machine Diagrams

In this appendix, more details concerning the chess state machine described in Section 4.7 are given. A high–level diagram indicates the relations between the different commands that can be invoked by the chess program. Depending on the bit–field of the command word, some states may or may not terminate at the *idle* state. In the cases where combinations of commands are requested, state transitions may follow the ** state transitions. A second figure showing the different instructions used for each command follows.



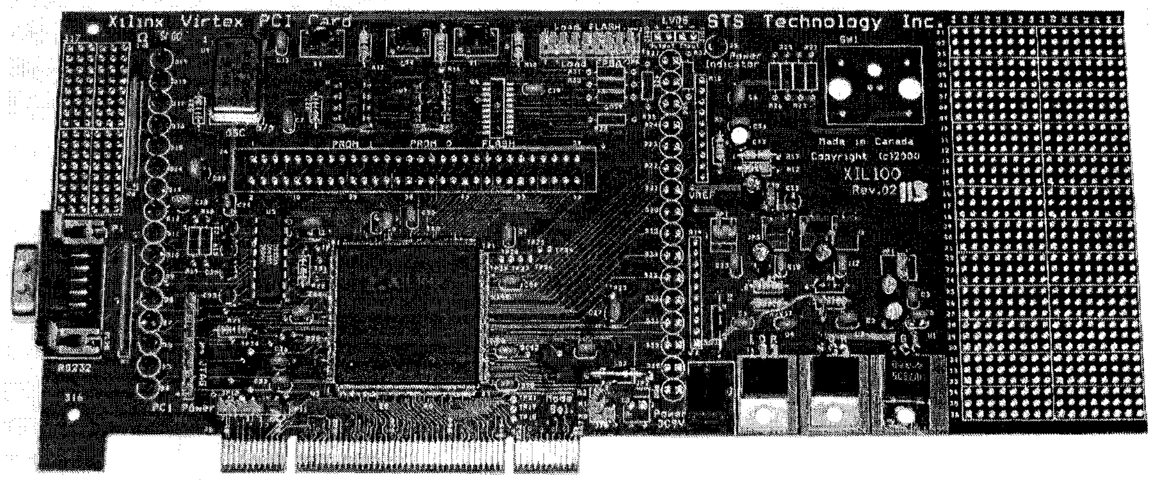
Chess finite state machine high–level diagram.



State machine and chess instructions. The “|” symbol represents “or”.

Appendix C

FPGA Development Board



FPGA Development Board.

References

[1] G.M. Adelson-Velsky, V.L. Arlazarov, A.R. Bitman, A.A. Zhivotovsky, and A.V. Uskov. Programming a computer to play chess. *Russian Math. Surveys*, 25:361–371, March/April 1970.

[2] G.M. Adelson-Velsky, V.L. Arlazarov, and M.V. Donskoy. Some methods of controlling the tree search in chess programs. *Artificial Intelligence*, 6(4):361–371, 1975.

[3] I. Althöfer. An incremental NegaMax algorithm. *Artificial Intelligence*, 43(1):57–65, 1990.

[4] T.S. Anantharaman. Confidently selecting a search heuristic. *ICCA Journal*, 14(1):3–16, 1991.

[5] T.S. Anantharaman. Extension heuristics. *ICCA Journal*, 14(2):47–65, 1991.

[6] T.S. Anantharaman, M.S. Campbell, and F.-H. Hsu. Singular extensions: Adding selectivity to brute–force searching. *Artificial Intelligence*, 43(1):99–109, 1990.

[7] G. Baillargeon. *Probabilité, Statistique et Techniques de Régression.* Les Éditions SMG, Trois–Rivières, QC, 1989.

[8] D.F. Beal. Experiments with the Null Move. *Advances in Computer Chess 5*, pages 65–79, 1989.

[9] D.F. Beal. A generalised quiescence search algorithm. *Artificial Intelligence*, 43(1):85–98, 1990.

[10] H.J. Berliner. Some innovations introduced by HITECH. *ICCA Journal*, 10(3):111–117, 1987.

[11] H.J. Berliner. HITECH chess: From master to senior master with no hardware change. *IEEE International Workshop on Industrial Applications of Machine Intelligence and Vision*, pages 29–35, 1989.

[12] H.J Berliner and C. Ebeling. HITECH. *Computers, Chess and Cognition*, pages 79–109, 1990.

[13] H.J. Berliner, G. Goetsch, M.S. Campbell, and C. Ebeling. Measuring the performance potential of chess programs. *Artificial Intelligence*, 43(1):7–20, 1990.

[14] H.J. Berliner, D. Kopec, and E. Northam. A taxonomy of concepts for evaluating chess strength. *IEEE Conference on Supercomputing*, pages 336–343, 1990.

[15] M. Boulé. MBCHESS v9.01. *http://www.macs.ece.mcgill.ca/˜mboul.*

[16] M. Boulé and Z. Zilic. An FPGA based move generator for the game of chess. *IEEE Custom Integrated Circuits Conference 2002 Proceedings*, pages 71–74, 2002.

[17] Brains in Bahrain.com. Kramnik vs. FRITZ, (about). *http://www.brainsinbahrain.com/about.*

[18] D. Breuker. *Memory versus Search in Games.* PhD thesis, Maastricht University, Department of Computer Science, Netherlands, 1998.

[19] D. Breuker, J. Uiterwijk, and H. Herik. Information in transposition tables. *http://citeseer.nj.nec.com/130130.html.*

[20] D. Breuker, J. Uiterwijk, and H. Herik. Replacement schemes for transposition tables. *ICCA Journal*, 17(4):183–193, 1994.

[21] M. Campbell, A. J. Hoane, and F.-H. Hsu. DEEP BLUE. *Artificial Intelligence*, 134:57–83, 2002.

[22] J.H. Condon and K. Thompson. BELLE chess hardware. *Advances in Computer Chess 3*, pages 45–54, 1982.

[23] C. Ebeling and A. Palay. The design and implementation of a VLSI move generator. *IEEE 11th Annual Internationnal Symposium on Computer Architecture*, pages 74–80, 1984.

[24] K. Evans. Personal communications. April 2002.

[25] T. Foden. GREEN LIGHT CHESS. *http://www2.prestel.co.uk/diamond/chess.htm (no longer available)*, downloaded on 01/04/1999.

[26] P.W. Frey. *Chess Skill in Man and Machine.* Springer–Verlag, New York, 1977.

[27] P.W. Frey. An empirical technique for developing evaluation functions. Some thoughts with a review of mitchell's thesis. *ICCA Journal*, 8(1):17–22, 1985.

[28] M. E. Glickman. A comprehensive guide to chess ratings. *http://math.bu.edu/people/mg/research.html*, 1995.

[29] M. E. Glickman. Parameter estimation in large dynamic paired comparison experiments. *http://math.bu.edu/people/mg/research.html*, 1999.

[30] S. Hammilton and L. Garber. DEEP BLUE's hardware–software synergy. *IEEE Computer*, 30(10):29–35, 1997.

[31] E.A. Heinz. Adaptive null–move pruning. *ICCA Journal*, 22(3):123–132, 1999.

[32] H. Horacek. Reasoning with uncertainty in computer chess. *Artificial Intelligence*, 43(1):37–56, 1990.

[33] F.-H. Hsu. A two–million moves/s CMOS single–chip chess move generator. *IEEE Journal of Solid-State Circuits*, 22(5):841–846, 1987.

[34] F.-H. Hsu. Computer chess, then and now: The DEEP BLUE saga. *Symposium on VLSI Technology, Systems and Applications*, pages 153–156, 1997.

[35] F.-H. Hsu. IBM's DEEP BLUE chess grandmaster chips. *IEEE Micro*, 19(2):70–81, 1999.

[36] F.-H. Hsu. Personal communications. June 2002.

[37] R.M. Hyatt. Chess and supercomputers: details about optimizing CRAY BLITZ. *IEEE Conference on Supercomputing*, pages 354–363, 1990.

[38] R.M. Hyatt and M. Newborn. CRAFTY goes deep. *ICCA Journal*, 20(2):79–86, 1997.

[39] A. Junghanns. Are there practical alternatives to alpha–beta? *ICCA Journal*, 21(1):14–32, 1998.

[40] D. Knuth and R. Moore. An analysis of alpha–beta pruning. *Artificial Intelligence 6*, pages 293–326, 1975.

[41] V. Manohararajah. Parallel alpha–beta search on shared memory multiprocessors. Master's thesis, University of Toronto, Department of Electrical and Computer Engineering, Toronto, Canada, 2001.

[42] T.A. Marsland. Evaluation–function factors. *ICCA Journal*, 8(2):47–57, 1985.

[43] T.A. Marsland and R.M. Hyatt. CRAY BLITZ: A computer chess playing program. *ICCA Journal*, 8(1):23–24, 1985.

[44] J. Moussouris, J. Holloway, and R. Greenblatt. CHEOPS: A Chess–Oriented Processing System. *Machine Intelligence 9*, pages 351–360, 1979.

[45] H.L. Nelson. Hash tables in CRAY BLITZ. *ICCA Journal*, 8(1):3–13, 1985.

[46] M. Newborn. Unsynchronized iteratively deepening parallel alpha–beta search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 10(5):687–694, 1988.

[47] M. Newborn. *Kasparov versus* DEEP BLUE. Springer–Verlag, New York, 1997.

[48] M. Newborn. Outsearching Kasparov. *Proceedings of Symposia in Applied Mathematics*, 55:175–205, 1998.

[49] M. Newborn and D. Levy. *How Computers Play Chess.* W.H. Freeman & Co., New York, 1991.

[50] A. Newell, J.C. Shaw, and H.A. Simon. Chess playing programs and the problem of complexity. *IBM Journal of Research and Development*, 4(2):320–335, 1958.

[51] W. Oney. *Programming the Microsoft Windows Driver Model.* Microsoft Press, Redmond, Washington, 1999.

[52] S.W. Otto and E.W. Felten. Chess on a hypercube. *IEEE International Specialist Seminar on the Design and Application of Parallel Digital Processors*, pages 30–42, 1988.

[53] PCI Special Interest Group, Portland, Oregon. *PCI Local Bus Specification – Revision 2.1*, June 1995.

[54] D. Perry. *VHDL – Third Edition.* McGraw–Hill, New York, 1998.

[55] A. Plaat. *Research Re: search & Re-search.* PhD thesis, Erasmus University, Rotterdam, Netherlands, 1996.

[56] A. Plaat and J. Schaeffer. New advances in alpha–beta searching. *24th ACM Computer Science Conference*, pages 124–130, February 1996.

[57] A. Plaat, J. Schaeffer, W. Pijls, and A. De Bruin. An algorithm faster than negascout and SSS* in practice. *Proceedings of Computing Science in the Netherlands*, pages 182–193, November 1995.

[58] A. Plaat, J. Schaeffer, W. Pijls, and A. De Bruin. Exploiting graph properties of game trees. *13th National Conference on Artificial Intelligence*, pages 234–239, August 1996.

[59] A. Reinefeld. An improvement to the scout tree–search algorithm. *ICCA Journal*, 6(4):4–14, 1983.

[60] J. Schaeffer. The history heuristic and alpha–beta search enhancements in practice. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 11(1):1203–1212, 1989.

[61] J. Schaeffer. Conspiracy numbers. *Artificial Intelligence*, 43(1):67–84, 1990.

[62] J. Schaeffer, P. Powell, and Jim Jonkman. A VLSI legal move generator for the game of chess. *VLSI Design*, pages 64–71, May/June 1983.

[63] C.E. Shannon. Programming a computer for playing chess. *Philosophical Magazine*, 41:256–275, 1950.

[64] J. Si and R. Tang. Trained neural networks play chess endgames. *International Joint Conference on Neural Networks*, 6(1):3730–3733, 1999.

[65] R. Siedel. Chess, how to understand the exceptions ! *ICCA Journal*, 8(1):14–16, 1985.

[66] J. Testa and A.M. Despain. A CMOS VLSI chess microprocessor. *IEEE Custom Integrated Circuit Conference*, pages 15.3.1–15.3.4, 1990.

[67] É. Thé. An analysis of move ordering on the efficiency of alpha–beta search. Master's thesis, McGill University, School of Computer Science, Montréal, Canada, 1992.

[68] W. Tunstall-Pedoe. Genetic algorithms optimizing evaluation functions. *ICCA Journal*, 14(3):119–128, 1991.

[69] D. Wilkins. Using patterns and plans in chess. *Artificial Intelligence*, 14:165–203, 1980.

[70] Xilinx. *The Programmable Logic Data Book 2000.* San Jose, California, 2000.

[71] A.L. Zobrist. A new hashing method with application for game playing. *ICCA Journal*, 13(2):69–73, 1990.