FPGA Multiprocessor for Game Tree Searches

António Pedro Santos Coelho

Thesis to obtain the Master of Science Degree in Electrical and Computer Engineering

Supervisor: Prof. Horácio Cláudio de Campos Neto

Examination Committee

Chairperson: Prof. Nuno Cavaco Gomes Horta

Supervisor: Prof. Horácio Cláudio de Campos Neto

Members of the Committee: Prof. João Paulo Baptista de Carvalho

May 2016

In the past Grandmasters came to our computer tournaments to laugh. Today they come to watch. Soon they will come to learn.

Monty Newborn, computer chess tournament organizer, 1977

Acknowledgements

I want to thank my supervisor Prof. Horácio Neto for his help and guiding me in the correct direction. I also want to thank my colleges and family for the massive support they have given me for so long.

Abstract

This thesis explores the development of a hardware/software system that implements an algorithm for game tree searches, applied to the game of Chess. The system architecture consists of a general purpose processor that executes the software part, and by a dedicated hardware unit that implements a move generator for the game of Chess. Move generation speed is a crucial part of a game tree search algorithm. The faster the move generator, the more tree nodes can be visited per time unit, thus finding more information to help choose the next move. Even though the hardware move generator proved itself to be much faster than the software move generator in the Faile chess engine, the system has a bottleneck in making the moves available to the game engine, so future work is needed to extract the full potential from the designed hardware.

Keywords

Chess, FPGA, hardware move generator, Faile

Resumo

Esta tese explora o desenvolvimento de um sistema hardware/software que implementa um algoritmo de busca de árvore de jogo, para o jogo de Xadrez. A arquitectura do sistema consiste num processador genérico para executar a parte do software, e também numa unidade de hardware dedicada á geração de lances para o jogo de Xadrez. A velocidade da geração de lances é uma parte crucial de um algoritmo de busca numa árvore de jogo. Quanto mais rápido o gerador de lances, mais nós da árvore podem ser visitados por unidade de tempo, encontrando assim mais informação para decidir o próximo lance. Embora o gerador de lances em hardware tenha mostrado que é muito mais rápido do que o gerador de lances em software do Faile chess engine, o sistema tem um estrangulamento em tornar os lances disponíveis para o motor de jogo, de forma que é necessário trabalho futuro para extrair o potencial do hardware projectado.

Palavras Chave

Xadrez, FPGA, gerador de lances em hardware, Faile

Table of Contents

List of Figures

List of Tables

List of Acronyms

AXI	Advanced eXtensible Interface
BFS	Breadth-First Search
BRAM	Block Random Access Memory
CPU	Central Processing Unit
DDR	Double Data Rate
DFS	Depth-First Search
FF	Flip-Flop
FPGA	Field-Programmable Gate Array
FSM	Finite-State Machine
HW	Hardware
IP	Intellectual Property
LSB	Least Significant Bit
LUT	Look-Up Table
MPAJF	Multiprocessador para Pesquisas em Árvores de Jogos em FPGA
LVA	Least Valuable Aggressor
MSB	Most Significant Bit
PC	Personal Computer
PCI	Peripheral Component Interconnect
PLA	Programmable Logic Array
MVA	Most Valuable Aggressor
MVV	Most Valuable Victim
PL	Programmable Logic
PS	Processing System
SoC	System on Chip
RAM	Random Access Memory
ROM	Read-Only Memory
SW	Software
UART	Universal Asynchronous Receiver/Transmitter
USB	Universal Serial Bus
VHDL	VHSIC Hardware Description Language
VLSI	Very-Large-Scale Integration

Chapter 1

Introduction

In this first chapter a description of the problem under study is given, along with the chosen approach taken in this work. The chapter ends with an overview of the target Hardware (HW) / Software (SW) platform.

1.1 Overview

One of the ways for a computer to play a game, while correctly choosing its moves so it can be a worthy opponent to a human, is by exploring the possible scenarios of the game so it can find the best one. The way to do that is by constructing a tree where each node of the tree represents a certain game state, its child nodes are the states resulting from the possible moves available to that node, and the leaf nodes are the end-of-game scenarios. By searching this tree for nodes where it is in advantage, the computer can choose the next move in hopes of "steering" the actual game into that advantageous situation.

As moves must be generated for each visited node of the game tree, so that its child nodes can be identified, move generation is one of the factors that limit the speed of the game tree search. This means that there is a direct relationship between the performance of the search algorithm and the speed at which moves are generated: the faster the move generator is, the better the found move will be for a given search time period, as more nodes of the tree will have been explored.

The aim of this work is to develop a Hardware (HW) / Software (SW) system for the game of Chess. Analysis of the state of the art shows that a hardware move generator can be faster than its software counterparts, thus the choice was made to develop a hardware move generator for an existing software Chess game engine. By using hardware designed specifically for move generation, instead of software algorithms that run on general-purpose Central Processing Unit (CPU) hardware, gains in move generation speed, thus gains in speed search, are hoped to be obtained.

The concept of this work - replacing a software chess move generator with a hardware version - is similar to the one used by Marc Boulé in his thesis work where he developed the CodeBlue hardware move generator. But unlike that work, the hardware move generator designed in this work uses a bitboard chess representation and deviates from the move generator architecture of the Belle computer that has been used on the most prominent hardware chess engines, including CodeBlue. Its implementation in a Field-Programmable Gate Array (FPGA) allows future changes and upgrades to be easily made.

This document is divided into six chapters organized as follows:

- Chapter 1, **Introduction**. This chapter gives an introduction to the subject of the work, as well as a description of the target platform for the designed hardware;
- Chapter 2, **Chess Game Tree Search**. This chapter is dedicated to the main rules of Chess, as well as the main algorithms used for searching a Chess game tree;
- Chapter 3, **State of the Art**. This chapter provides an overview of past hardware move generators, along with an introduction to the Faile chess engine;
- Chapter 4, **Hardware Design**. A description of the designed hardware, and its interface with the Faile chess engine is the subject of this chapter;
- Chapter 5, **Results**. This chapter contains the performance and testing results of the designed hardware;
- Chapter 6, **Conclusions**. This chapter closes the document with the conclusions drawn from this work, and also possible improvements to the designed hardware.

1.2 Target HW/SW Platform

The target System on Chip (SoC) FPGA used in this work is a Xilinx Zynq®-7020 All Programmable SoC.

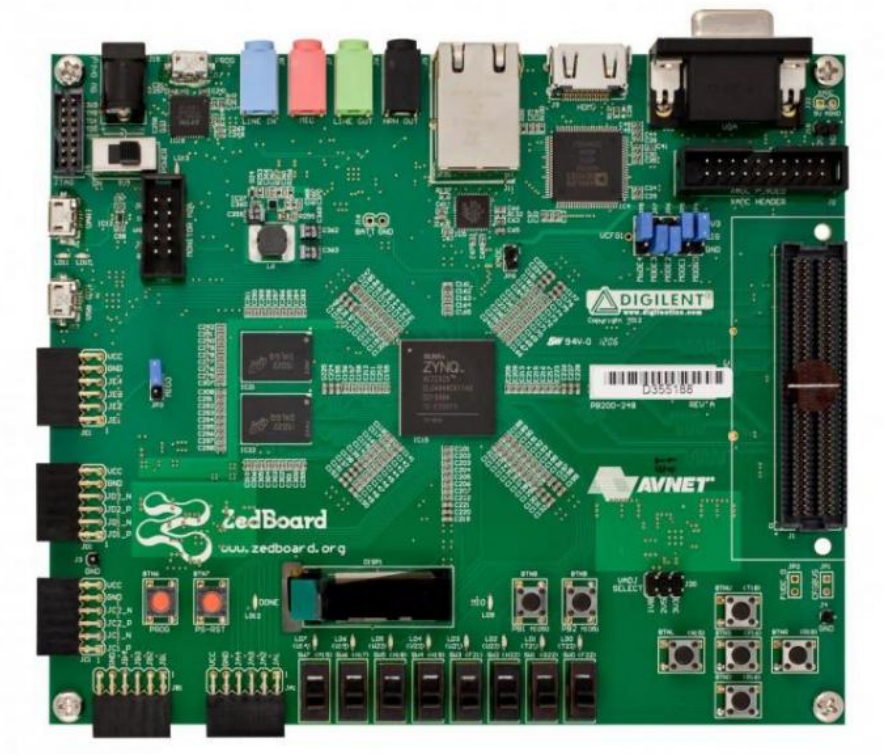

Figure 1.1 Zedboard™ development kit board.

The Zedboard™ development kit board, seen in Figure 1.1 [1], was selected as the target demonstration platform for this work. It includes a Zynq®-7020 SoC FPGA device, whose FPGA part provides resources up to 53200 Look-Up Tables (LUT) and 106400 Flip-Flops (FF) for implementing programmable logic, and up to 560 KB of Block Random Access Memory (BRAMs) for local storage.

The Zynq® SoC architecture consists of two major sections: the Processing System (PS), a processor-based unit that executes the software components of the application; and the Programmable Logic (PL), the part of the device where the hardware components of the application are implemented. A diagram of the PS and the PL is shown in Figure 1.2, extracted from [2].

The PS section is based on a Dual-core ARM® Cortex™-A9 CPU that can execute up to a maximum frequency of 667 MHz. The PL section is based on the programmable logic fabric of the Artix-7 FPGAs [3].

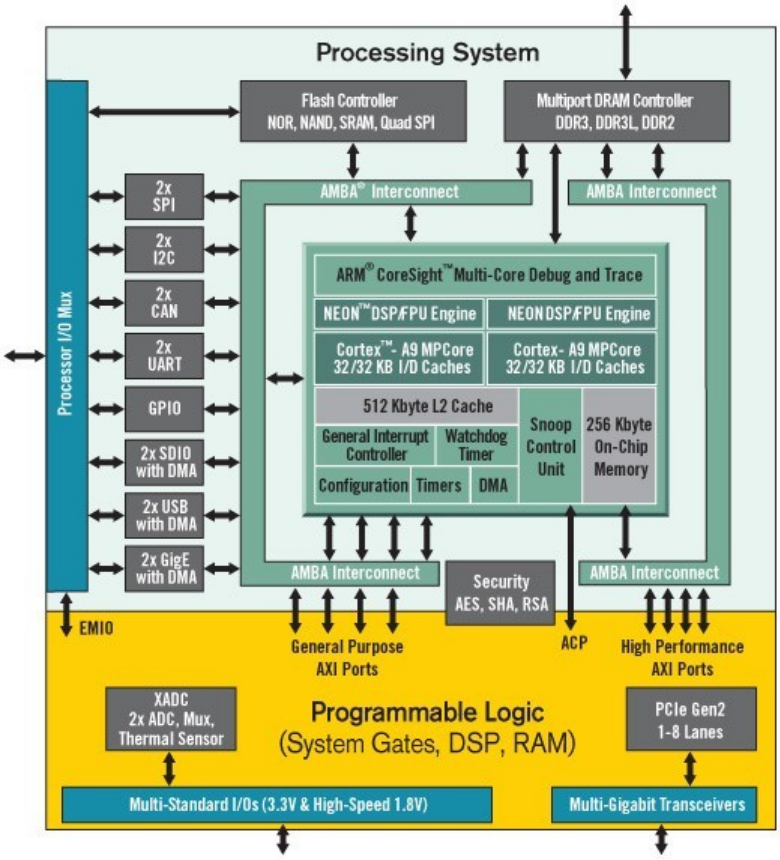

Figure 1.2 Zynq®-7000 diagram.

Of interest to this work, amongst the many features of the Zedboard™ [4], are the 512 MB of Double Data Rate type 3 (DDR3) Random Access Memory (RAM) available for use by the FPGA, Universal Serial Bus (USB) interface to program the FPGA and USB Universal Asynchronous Receiver/Transmitter (UART) connection between the Zedboard™ and a Personal Computer (PC), that allows a user to interface with the Faile chess engine as if it is running in the PC.

The hardware was designed in Vivado 2015.2 and is written in VHSIC Hardware Description Language (VHDL) hardware description language. Xilinx SDK 2015.2 was used to compile the software.

Chapter 2

Chess Game Tree Search

This chapter presents background information about the game of Chess and game tree search. The first half contains an introduction to the game of Chess, where the reader will be exposed to the main rules of Chess, the movement patterns of the pieces and their capabilities. An explanation of the Chess game tree search algorithms encountered in this work is the subject of the second half of the chapter.

2.1 The Game of Chess

Perhaps the most famous game in the world, Chess is a game of strategy that has fascinated people for centuries. A basic overview of the game is given in the following pages, according to the rules in [5].

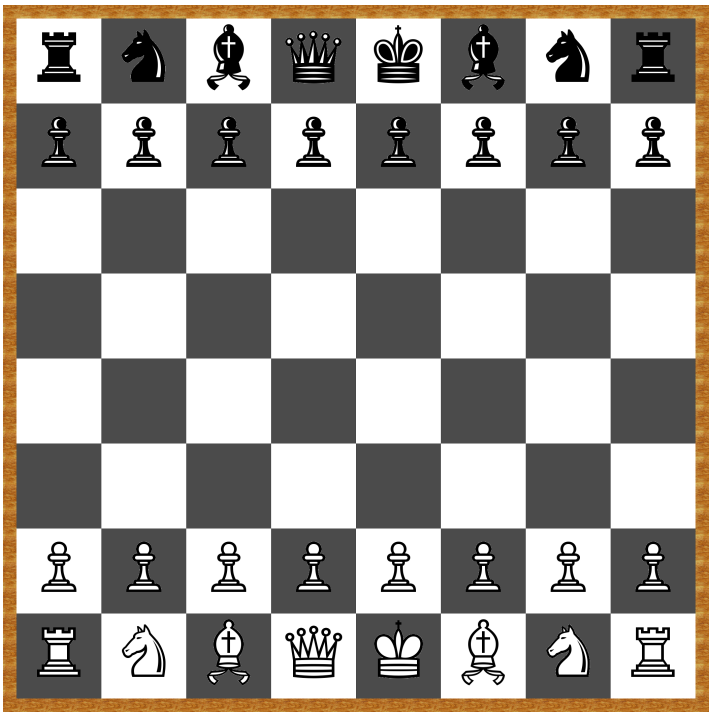

Figure 2.1 Chess board at the start of the game.

Chess is played between 2 opponents, each controlling 16 pieces on a checker board with 64 square positions. The 2 players are identified by the colour of their pieces on the board, Black or White, and they take turns moving their pieces, with the White player starting first. The pieces, which initially are placed on the board in the configuration shown in Figure 2.1, consist of 8 Pawns, 2 Rooks, 2 Knights, 2 Bishops, 1 Queen and 1 King per player. The objective of the game is to capture the opponent's King.

The squares on the board are divided into 8 horizontal lines called ranks, numbered 1 through 8, and also into 8 vertical lines called files, distinguished by a letter from 'a' to 'h'. Two further divisions are the diagonal and the anti-diagonal directions, in which some pieces move. A diagonal is a sequence of squares aligned in a 45º angle, in a left-to-right and bottom-to-top direction. The anti-diagonal direction refers to the similar alignment 90º away.

When a piece is in reach of an opponent piece (the way each piece moves is explained below) it is said to be under attack. Carrying out an attack is called capture, and involves the attacking piece moving to the square of the attacked piece, with the now captured piece being removed from the board and taking no further part in the game.

In Chess a move is said to be illegal, thus it cannot be done, when it leaves the King under attack from an opponent piece. In that situation the King is said to be "in check", and it either has to

move to a square that currently is not under attack, or the attacking piece has to be neutralized either by capturing it or by blocking its attack. Failure to do so results in the King being captured, a situation called “check mate”.

2.1.1 Pawn

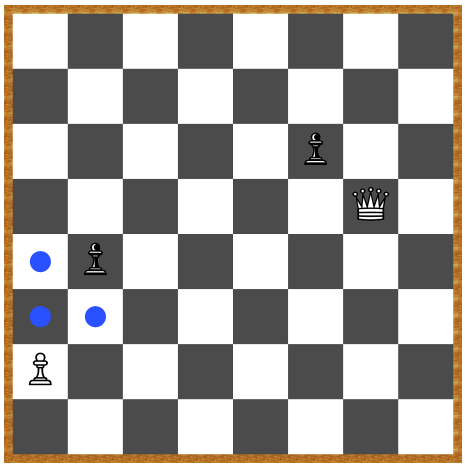

Figure 2.2 Pawn movement pattern.

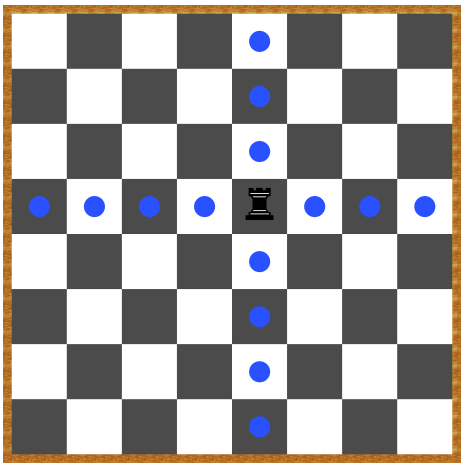

Figure 2.3 Rook movement pattern.

The Pawn, the least valuable piece on the board, always moves 1 square forward (called Push), except in 2 situations: when it is moving from its initial position, when it can move 2 squares forward (a Double Push); and 1 square diagonally forward when capturing an opponent piece. Figure 2.2 exemplifies those moves. The White Pawn shows the Push and Double Push moves, while the right Black Pawn is attacking the White Queen and can capture it. The left Black Pawn shows a Push move, as well as a special capture move available to the Pawns called En Passant. When a Double Push is performed and the Pawn advances 2 squares, as the White Pawn in Figure 2.2 illustrates, and an opponent Pawn is in the position represented by the Black Pawn, for the very next move only the En Passant capture is available. The Black Pawn would perform the diagonal capture move just as if the White Pawn had only advanced 1 square. It is the only capture move that does not end with the capturing piece taking the place of the captured piece.

When the Pawn reaches the rank at the opposite end of the board, it is promoted to a superior piece: a Queen, a Bishop, a Knight or a Rook. The player has the choice of which piece the Pawn is promoted to, but the Queen is the most usual choice as it is the most valuable of the options.

2.1.2 Rook

The Rook is one of the sliding pieces, which can move any number of squares within a file or a rank, until it finds another piece or the end of the board, as shown in Figure 2.3. Under certain conditions, the Rook takes part in the Castling move (see King).

2.1.3 Knight

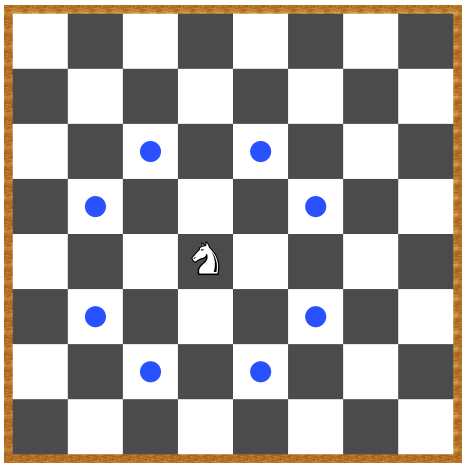

Figure 2.4 Knight movement pattern.

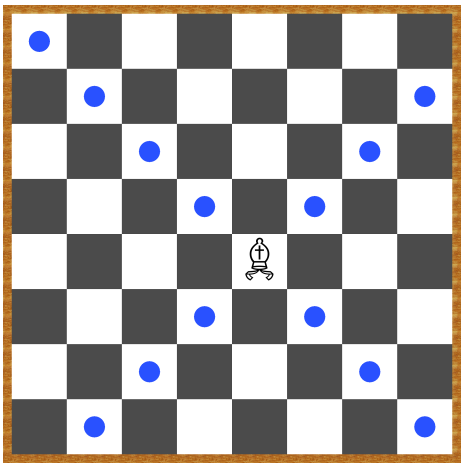

Figure 2.5 Bishop movement pattern.

The particular move pattern of the Knight makes it the only piece that can “jump” over other pieces. It has a 2-squares-forward and 1-square-to-the-side type of movement, as shown in Figure 2.4, regardless of whether the squares it passes over are occupied or not.

2.1.4 Bishop

Like the Rook, the Bishop is a sliding piece, but its movement is made exclusively in the diagonal and anti-diagonal directions, as shown in Figure 2.5. The initial position and movement pattern of the 2 Bishops is such that one of them only has access to white squares, while the other only has access to black squares.

2.1.5 Queen

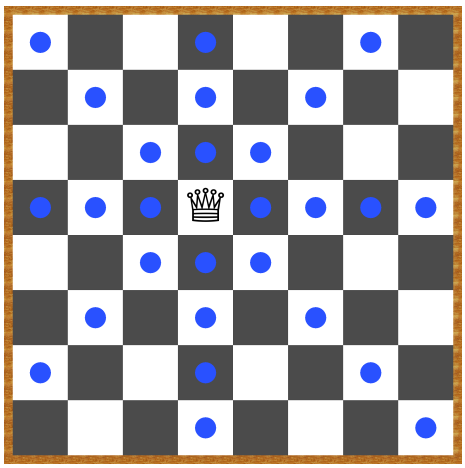

Figure 2.6 Queen movement pattern.

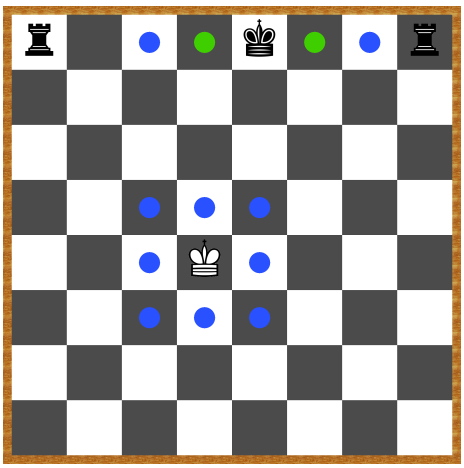

Figure 2.7 King movement pattern.

The Queen can move in all 4 directions (file, rank, diagonal and anti-diagonal), effectively combining the sliding movement capabilities of the Rook and the Bishop, as shown in Figure 2.6. This combination of sliding movements makes the Queen the most mobile piece of them all, with up to 27 possible destinations.

2.1.6 King

The King, the most valuable piece on the board, can only move 1 square in any direction, as demonstrated by the White King in Figure 2.7. There is one exception to that called the Castling move, where the King moves 2 squares sideways. The Castling move switches the order of a King and one of its Rooks, moving the King to a more lateral positon on the board. As illustrated in Figure 2.7 by the Black pieces, the King would move to one of the blue spots, while the Rook on that side would move to the adjacent green spot. This move is only available when: the King has not moved from its initial position; the Rook involved in the move has not moved; and there are no pieces between the King and the involved Rook. It is the only move where 2 pieces move at the same time.

2.2 Game Tree Search

This section summarizes the main aspects of game tree search with information collected from [6].

The game tree is a data structure that represents all the possible game states in hierarchical form. The nodes of the tree represent a game state, with each child node being the state resulting from a possible move available to that node. The leaf nodes are end-of-game situations, while the root node is the start of the game. The edges connecting the nodes represent the moves that relate the nodes.

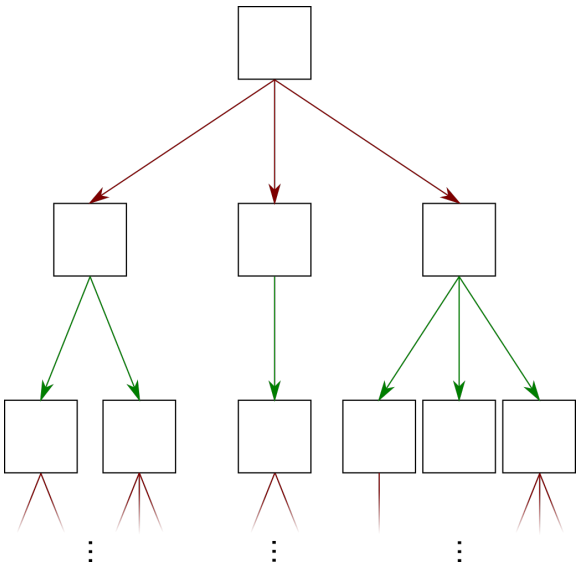

Figure 2.8 Game tree diagram.

To move from a parent node to one of its child nodes, the search algorithm applies a move to the game state of the parent node, thus changing it to the game state of the child node. Moving back up from a child node to the respective parent node is referred to as an “unmove”. The move that was previously applied to arrive at the child node is undone, thus reverting back to the game state of the parent node. For a tree of a 2-player game like Chess, each level of edges consists of the moves of one of the players, alternating with each level, as exemplified in Figure 2.8 by the different coloured edges.

The computer chooses its next move in the game based on the information it gathers by exploring the game tree. This is commonly done using the Minimax game tree search algorithm. It is usually implemented as a Depth-First Search (DFS), as opposed to a Breadth-First Search (BFS), to reduce memory requirements.

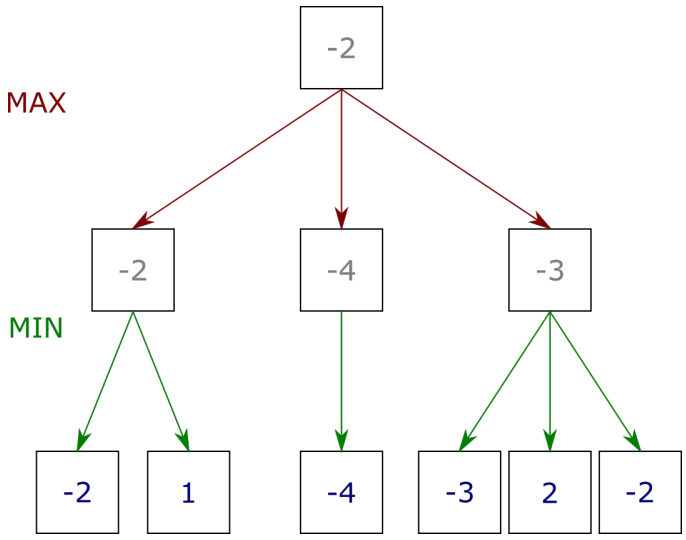

Figure 2.9 Minimax tree search example.

Minimax works by searching the tree to the leaf nodes where, by using an evaluation function, it obtains a score that measures how good that end-of-game situation is for the players. This value is then passed up from child nodes to parent nodes, where one of the values of the child nodes is chosen to become the value of the parent node. This choice is made based on whether the parent node is a maximization node, which are nodes that are reached after moves by the computer, or a minimization node, reached after an opponent move. Maximization nodes, like the name suggests, are nodes where the value of the child nodes is maximized, thus the parent node picks the larger child node value. A minimization node picks the smaller value out of the child nodes. This process maximizes the computer’s score, while at the same time minimizing the opponent's score. This is repeated until the tree is fully explored. After the search is done, the chosen move is one that minimizes the losses, i.e., maximizes the minimum score. Figure 2.9 shows an example of the Minimax search algorithm. The leaf nodes show in blue colour their calculated value, and the values passed up from the child to the parent nodes are shown in grey. In this example, the chosen move

would be the left-most one with the value of -2, thus avoiding the other 2 potentially more costly game scenarios.

In the game of Chess, due to the size of its game tree - Claude Shannon estimated that the typical game has more states than there are atoms in the observable Universe - it is virtually impossible to search it completely, so only a few levels of the tree are searched. This sub-tree which is explored instead, has the current game state for its root node, and a node evaluation is done at the chosen search depth from this root node, instead of evaluating end-of-game results at the leaf nodes. By using a heuristic evaluation function, any game state can be given a score which measures the value of that state to the players. For Chess this is usually calculated based on the type, position and number of pieces on the board.

In a zero-sum game like Chess, where one player's gains are the opponent's losses, the sum of the scores is zero. The score of one player is obtained by negating the score of the other. This variant of the Minimax algorithm is called Negamax and simplifies the algorithm code.

2.2.1 Alpha-Beta Pruning

The Alpha-Beta algorithm is an improvement on the Minimax algorithm, which reduces the amount of nodes that have to be visited to explore the tree, without any loss of information.

By keeping 2 values, a minimum score (alpha) for the maximizing player and a maximum score (beta) for the minimizing player, the Alpha-Beta algorithm can prune away branches of the tree that do not affect the outcome of the search.

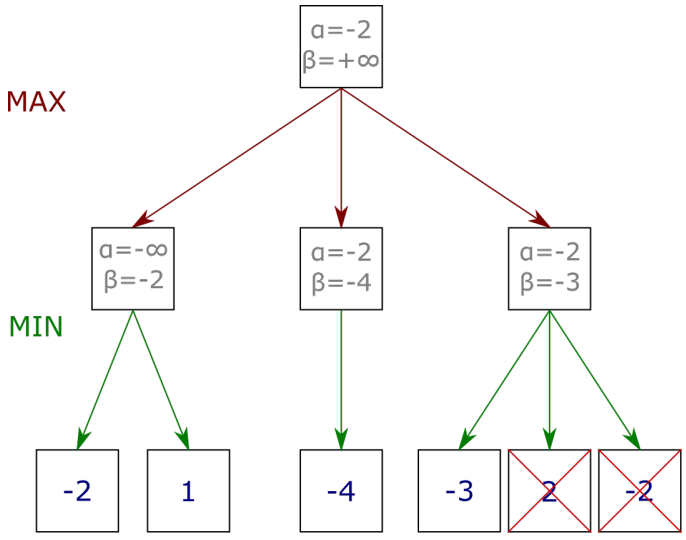

Figure 2.10 Alpha-Beta tree search example.

At the beginning of the search, alpha has the initial value of minus infinity - the absolute minimum score for the maximizing player, while beta is set to plus infinity - the absolute maximum score for the minimizing player. Both the alpha and beta values of the parent node are sent to the child nodes, where they are updated during the search. Alpha is updated with the current score of the node when at a maximization node, and beta is updated with the node score when at a minimization node.

In each node, in addition to the maximization or minimization tasks of Minimax, a test is made after the first child node is explored to check if the remaining child-nodes are worth exploring. For a maximization node, if the current node value is lower than alpha, the information contained in the remaining child nodes is redundant and they are not explored. For minimization nodes, if the current score is greater than beta, the pruning also takes effect. This way, alpha and beta act as cut-off values, which when triggered prune the remaining child nodes and respective sub-trees.

The pruning is done without any loss of information, as the pruned nodes do not affect the final move decision. In fact, the whole sub-tree rooted on the current node does not affect the move decision, as the already explored nodes of it show that this sub-tree has a worse score than what was previously found. Because the Minimax algorithm wants to minimize losses, it becomes redundant to search nodes that *can* produce a worst result than a previously found one.

An example of the application of the Alpha-Beta algorithm is shown in Figure 2.10, where the final values for alpha and beta in each node are displayed. The pruned nodes that were not explored on this example are indicated with a red cross.

To fully exploit the pruning capabilities of the Alpha-Beta algorithm, the order in which the child nodes are visited, i.e., the order in which moves are applied to the parent node, should be such that the highest value child node - for the player making the move - is explored first. If a cut-off is to occur, it will be due to the child nodes with the highest value. Consider again the example in Figure 2.10. If the leaf node with the value -3 was the last one evaluated by its parent node, no nodes would be pruned as it is the only node there with a score below alpha. If it was the second node to be evaluated, only the remaining node would be pruned. It is clear that move ordering is a big factor in the performance of the Alpha-Beta algorithm. In Chess, capture moves are the ones that produce the greatest change in game score, thus are usually the most valuable.

2.2.2 Iterative Deepening Search

In an Iterative Deepening search, the search algorithm is run sequentially with increasing depth: first search to a depth of 1, then search again but to a depth of 2, then 3, etc. Move order information is gathered with each iteration of the algorithm, and is then used to speed up the subsequent iterations. This results in a faster search to a certain depth, than a single “direct” search to that same depth.

2.2.3 Quiescence Search

Stopping a search abruptly upon reaching the desired level can lead to the wrong move being chosen, as the search algorithm has no information about what can happen next. The quiescence search is an extension to the normal search, which is done when a capture move is found at the last level of the search, in order to find the adversary’s response to that capture. This way, a disadvantageous piece exchange is avoided. E.g., if the last move found in the search is a Queen capturing a Pawn it seems to be a good move, but if the next move of the adversary allows that Queen

to be captured, that “good move” turns out to be a costly one. The quiescence search ends when it finds no more captures, a quiescent game state.

2.2.4 Null Move Pruning

The Null Move heuristic is a method to increase the tree branch pruning on the Alpha-Beta algorithm. Under certain conditions - most importantly when the King is not in check, amongst others - the Null Move heuristic effectively passes the next move to the other player[1] and then explores the subsequent sub-tree to a smaller depth than originally planned. If the sub-tree score is such that it causes a cut-off, some nodes are saved due to the shallower search. If not, then a normal search has to take place.

2.2.5 Transposition Table

The game tree contains several game states that can be reached via different move sequences, i.e. a transposition, which causes a search to encounter a certain game state several times. To not waist time re-searching the sub-trees of these repeated game states a list is kept, implemented in the form of a hash table, of the information obtained in a previous search of those sub-trees. Once a repeated game state is found, the information in the transposition table is used, thus reducing the total search time.

[1] A “passing” move is not legal in Chess.

Chapter 3

State of the Art

This chapter presents a review of previous work on chess hardware move generators, followed by an assessment of open source software chess engines suitable for this work.

3.1 Hardware Move Generators

Below, an overview is provided of the main hardware move generators, containing an explanation of their features and capabilities.

3.1.1 Belle

Belle was the first computer built exclusively to play Chess [7] [8] [9]. Designed at Bell Laboratories in the 1970s with hardware by Joe Condon and software by Ken Thompson, Belle used a "brute force" approach to Chess playing. Unlike many computer Chess (software) programs before, which played Chess by recognizing patterns like humans, Belle used algorithms to search the game tree and find the best next move.

The move generation is done by 64 logic circuits, one for each square on the board, connected to neighbour square-generators to allow signals to be propagated, indicating piece attacks using the Most Valuable Victim/Least Valuable Aggressor (MVV/LVA) scheme. The moves generated are pseudo-legal moves[2].

MVV/LVA is a method to generate ordered capture moves by prioritizing the least valuable pieces capturing the most valuable pieces. It works in 2 parts: first a find-victim cycle loops through the opponent pieces currently under attack to find the most valuable one[3]; for each piece found that is under attack, a search is performed on the pieces that are attacking it by the find-aggressor cycle, to find the least valuable attacker.

The square-generator logic of Belle, illustrated in Figure 3.1, contains a 4-bit piece register to keep track of which piece is on that square, a Programmable Logic Array (PLA) receiver (identified as RECV PLA in the diagram) which receives piece attack signals from neighbour squares, signals that were generated by their transmitter Read-Only Memory (ROM) (XMIT ROM in the diagram).

The signals connecting the squares are the Manhattan (Man) signal for piece attacks within a file or a rank, the Diagonal (Diag) for diagonal or anti-diagonal attacks, the King and Knight signals that communicate King and Knight attack patterns, and 2 Pawn signals for Pawn Push and attack. The OP signal controls the hardware to do a find-victim or find-aggressor cycle, and the white-to-move (WTM in the diagram) signal indicates whether the next move belongs to the White player.

[2] A pseudo-legal move is a move that might not fulfill the requirement of not putting, or the requirement of not leaving, the King in check, but otherwise is a valid move in every way.
[3] the Chess pieces ordered by value: King, Queen, Rook, Bishop, Knight and Pawn

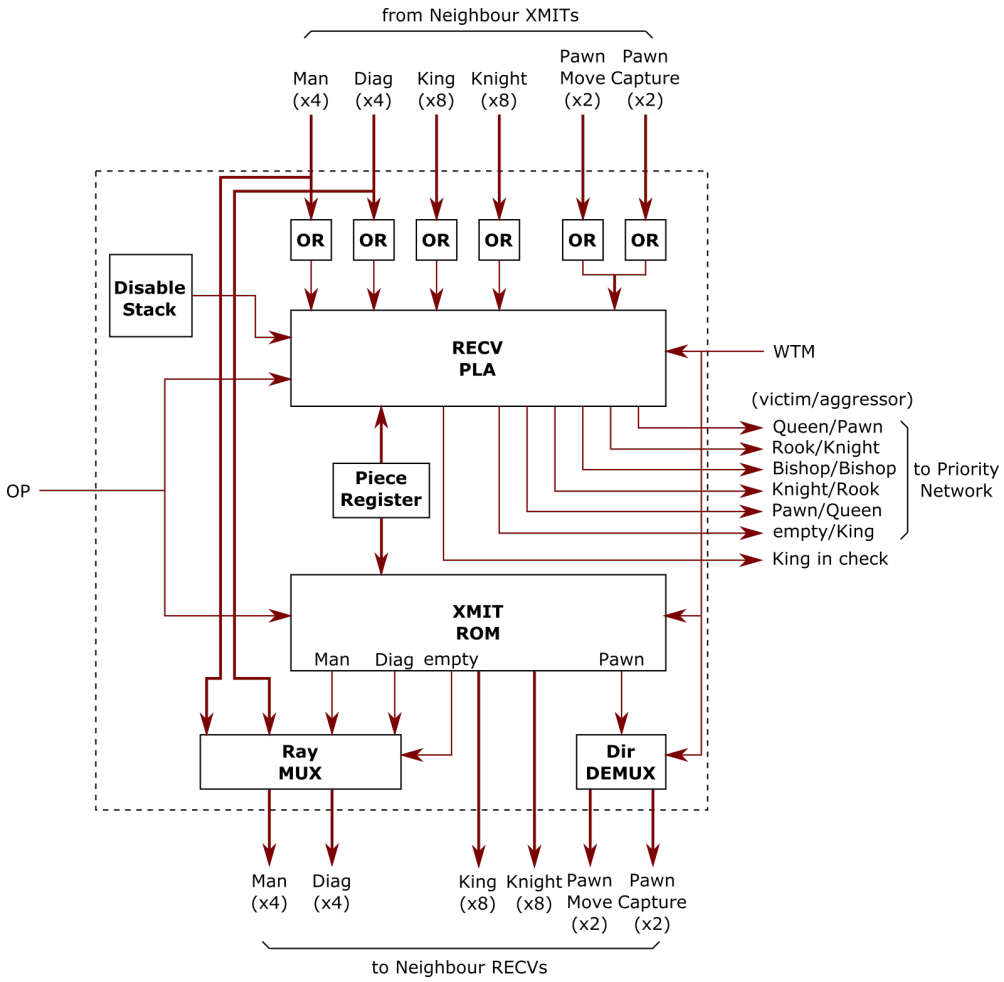

Figure 3.1 Diagram of Belle's square logic.

The victim or aggressor signals generated by the RECV PLA of each of the 64 squares are sent to a Priority Network, which in the find-victim cycle determines the target square of the move, and in the find-aggressor cycle determines the origin square. By using the OR bitwise operation on the 64 King in check signals - one per square - the check situation can be detected.

A disable stack of 1-by-64 bits exists in each square circuit to mask out victim and aggressor pieces as the moves are generated. Once a victim is found and after the move for its first aggressor is generated, that aggressor piece is disabled so that the next find-aggressor cycle finds another piece. Once no more aggressors exist, the victim is disabled and all the previously disabled aggressors are enabled, and the search for the next victim begins.

The move generation starts with the find-victim cycle, which enables the transmitters in each square with pieces of the side-to-move colour, so they transmit to the neighbour squares the appropriate move signals. Which output signals of the transmitter are enabled is based on the type of piece currently in the square, e.g., for a Knight piece only the 8 Knight signals are enabled. Receivers on each square receive those signals and generate outputs according to their own piece type. On empty squares the received Man and Diag signals are passed thru to the next square of that direction until another piece (or the end of the board) is found, so the moves of the sliding pieces can be

generated. The output signals from the receivers are then sent to the Priority Network which chooses the most valuable victim, and saves the square of that piece, i.e., the destination square of the move.

After a victim is identified, the find-aggressor cycle begins in which only the transmitter of that victim piece is enabled. In this mode, the transmitter enables all outputs as a combination of the movement of all pieces. The pieces of the side-to-move colour which receive these signals generate outputs to the Priority Network, thus being recognized as the aggressor pieces to the previously found victim piece. The Priority Network then chooses the least valuable aggressor, thus finding the move origin square and finishing the move generation.

With the hardware move generator coupled with a hardware game state evaluator and a transposition table, all controlled by a microprocessor to perform the Alpha-Beta search algorithm, Belle eventually managed to examine 160000 nodes per second, winning several Association for Computing Machinery North American Computer Chess Championships in the late 1970s and early 1980s.

3.1.2 Deep Blue

Deep Blue is perhaps the most famous Chess engine, as it gained worldwide fame for being the first computer to defeat a Chess World Champion when in 1997 it won a 6-game match against Garry Kasparov. Its origins can be traced back to 1985 when Carnegie Mellon University students Feng-hsiung Hsu and Thomas Anantharaman created ChipTest [9] [10] [11] [12] [13].

ChipTest was a Very-Large-Scale Integration (VLSI) chip controlled by a Sun-3 workstation that used Belle's move generator architecture with a major improvement. The disable stack in each square circuit was removed and its functionality was cleverly replaced with smaller logic circuitry to generate the appropriate mask signals using the data of the last move.

After a move is played, explored and then reverted, a new move needs to be generated by finding another aggressor. In the MVV/LVA method the moves are listed with the aggressor pieces in increasing value, thus by using data from the last move it is possible to determine which of the current aggressors should not be included in the search for the next aggressor piece. Aggressor pieces with lesser value than the one in the last move, pieces for which moves were previously generated, are thus identified and masked out. This scheme also applies to the masking of the victim pieces. This change, along with a redesigned priority network, made it possible to shrink the design of the rack-sized Belle computer of less than a decade before, down to a single chip. After improvements and bug fixes, the latest iteration of ChipTest was able to search 500000 nodes per second.

In 1988, after several improvements and with the development team now augmented by Murray Campbell, Andreas Nowatzyk, Mike Browne and Peter Jansen, ChipTest was renamed Deep Thought.

Deep Thought contained software changes to optimize the evaluation function, the game tree search and more importantly, it was now a multi-processor system. Its last version, Deep Thought II, was a 24 processor system. The project also moved from Carnegie Mellon University to IBM in the early 1990s.

By the mid-1990s the team which now consisted of Feng-hsiung Hsu, Murray Campbell, A. Joseph Hoane, C. J. Tan, Jerry Brody and Joel Benjamin designed Deep Blue. By expanding the architecture of Deep Thought to 480 chess processors controlled by 30 IBM/RS 6000 SP workstation nodes, an average search speed of 100 million tree nodes per second was achieved. Aside from the larger scale, advances in chip technology allowing faster logic, and evaluation function changes increased the performance of the chess processors. The move generator design remained essentially the same as Deep Thought, but it featured some extra capabilities.

The checking moves generation mode allows moves that put the King in check to be generated exclusively. By activating the opponent's King square circuit in the find-aggressor mode and simultaneously activating the find-victim mode on all of our pieces, intersections between the squares attacked by the aggressors and the squares that attack the opponent King can be identified. If such squares exist, a move can be generated that immediately puts the King in check by moving a piece to the identified intersection square. This requires 2 transmitters per square, a find-victim transmitter and a find-aggressor transmitter, and accordingly the receiver now receives twice as many signals from the neighbour squares. This gives the search algorithm the capability of "forcing" the game in a certain direction during the quiescence search. Another special move generation mode is the King check evasion mode, used to defend the King when it is under attack by the opponent.

In Deep Blue, the game tree search was initiated in software by a single workstation node - the master node - which explored the top levels of the tree. The leaf nodes resulting from that initial search were then divided by all 30 workstation nodes, including the master, and those sub-trees were explored further. At the end of this second software search, the chess processors finish the remainder of the search including the quiescence search.

3.1.3 Brutus and Hydra

Started in late 2000 by Chrilly Donninger, Alex Kure and Ulf Lorenz, and with the support of ChessBase, Brutus aimed to be a FPGA version of Deep Blue. By implementing the hardware in a FPGA, the lead times for the manufacturing of custom chips would be avoided, thus making changes to the design easier and the overall system cheaper [14] [15] [16].

The user interfaces with Brutus on a PC running the ChessBase/Fritz GUI, which connects via Internet to a Linux server with 4 nodes, which formed Brutus itself. Each node has 2 CPUs, each associated with 1 FPGA Peripheral Component Interconnect (PCI) card. The 4 nodes are interconnected by a high-speed connection network. The FPGA PCI card contained a Virtex-E family device, in which the move generator, game state evaluator and game state logic are implemented, controlled by a 54-state Finite-State Machine (FSM).

The search work is distributed between the 8 processors of Brutus using the work stealing strategy. Every time a processor has no work, it sends a message to another processor, chosen at random, requesting work. If the chosen processor has a part of its sub-tree that it has not yet explored, a game tree node is sent back to the requester processor, thus dividing the search of that sub-tree between 2 processors. In the case that the chosen processor has no work it responds that it has no

work, and another work request is made to another processor. Once a processor finishes its work, it sends the results obtained to the processor from which it requested the work, and then starts the work request process again. Initially, at the beginning of the search, one processor is given the root node of the game tree and it starts the search, while the remaining processors start the work request process.

The initial levels of the tree are searched by the CPUs in each node, and only the search of the last 3 levels of the game tree, plus quiescence search and evaluation, are executed by the FPGA. This is done to prevent the PCI bus between the CPUs and the FPGAs from becoming the bottleneck of the system.

As with Deep Blue and its predecessors, the move generator of Brutus was also based on the move generator architecture of the Belle. But unlike the strict MVV/LVA move ordering of Deep Blue, Brutus uses additional dynamic ordering. Learning from past searches, moves that proved not to be good are deferred to later testing in favour of other ones. Brutus also features Null Move pruning. It was estimated that Brutus could search 2.5 million nodes per second per FPGA card, which operated at a frequency of 30 MHz.

After a disappointing result at the 2003 World Computer Chess Championship, ChessBase withdrew its support for the project. With support by the Pal Group of Companies the Brutus project continued, being renamed Hydra.

The general architecture of Hydra does not differ much from Brutus. Several upgrades made Hydra one of the best chess computers in the mid-2000s.

3.1.4 CodeBlue

The product of Marc Boulé's thesis work, CodeBlue is a FPGA hardware move generator designed as a replacement for the original software move generator of his own software Chess engine, the MBChess [17].

CodeBlue uses the Belle move generator architecture, but instead of the MVV/LVA move order it uses the Most Valuable Victim/Most Valuable Aggressor (MVV/MVA) order, as a small improvement was obtained with it. The MVV/LVA ordering was retained for the quiescence search.

Like Deep Blue, CodeBlue has the capability to generate checking moves, but unlike Deep Blue it does this without using the 2 transmitters per square and the larger receiver. This is accomplished by using a communication bus between the squares, thus reducing the amount of connections between the squares when compared to Deep Blue.

The victim and aggressor masking is similar to the Belle design, with a small memory storing the mask bits. This was found to be a better design for FPGA implementation than the mask logic version used in ChipTest and its successors.

The FPGA used by CodeBlue is mounted on a PCI card connected to a PC. The PCI connection is used by MBChess to control the operation of CodeBlue by sending commands to a FSM. Commands to update the piece registers, move making and unmaking are available, as well as to request generation of the next move and transfer to MBChess.

This approach of creating of a hardware move generator to replace the software move generator on a software chess engine used by Marc Boulé in CodeBlue was also followed by this work.

3.2 Software Chess Engines

In order to find a suitable open-source software engine for this work, a review of existing Chess engines was made. A Chess engine with a source code containing a clearly defined move generation code was desired, to ease its replacement by the designed hardware move generator.

The source codes of the Chess engines Beowulf [18], Crafty [19], Faile [20], Senpai [21] and Stockfish [22] were inspected, and the Faile chess engine was chosen for this work for its well-structured source code that contained an easily isolatable move generation code.

3.2.1 Faile Chess Engine

The Faile chess engine was developed by Adrien Regimbald as an open source Chess engine that is, in the words of the author, "full featured engine, yet the source is small, clear, neat and well commented". It uses a 12x12 board representation, the search uses the Negamax variant of the Alpha-Beta algorithm, contains a transposition table, features null move pruning and quiescence search. The version used in this work is version 1.4.4, released in 2001.

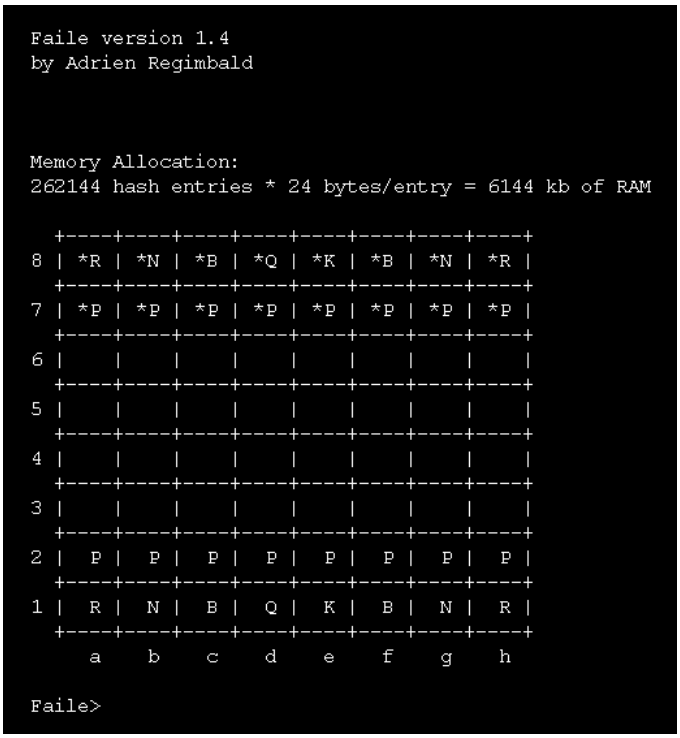

Figure 3.2 Faile command line interface.

The command line interface of Faile is displayed in Figure 3.2 as it looks at the start of a game. The user controls the White pieces (indicated by the letters without the asterisk), and directs a move by specifying the current file-and-rank coordinates of the piece to move, followed by the destination coordinates, e.g., the command *a2a4* would move the left most White Pawn 2 squares forward. Immediately after a user move, Faile begins the tree search process to find a move for the Black pieces, after which the selected move is applied, and the next move is requested from the user.

Changes had to be made to the source code of Faile, so it could be run in the Zedboard™. The original capability to reduce the search depth when the search was taking too much time was removed, as the required time-keeping code was dependent on functions that were not available. Adapting that code so it could run in the Zedboard™ was outside of the scope of this work. A fixed search depth of 6 was established instead.

Due to the organization of the source code of Faile, the replacement of the original move generator was relatively easy. In the original source code, the moves are generated in the *gen* function that outputs unordered moves to an array. Moves are subsequently ordered by another function. The replacement of the move generator was made by simply replacing the call to the *gen* function by a call to another function that accesses the hardware generated moves, and copies them to the move array.

To maintain a hardware version of the game state, that at all times tracks the internal game state of Faile, the changes to the software game state had to be communicated to the hardware. Each time a move, or unmove, is performed by Faile in its internal game state, an extra function is called to send to the hardware the respective move data.

Due to the nature of the search algorithms, extra data transfers were needed to keep the hardware game state updated. Extra calls to relay to the hardware updated En Passant square information - due to the repeated searches of iterative deepening, and side-to-move information - due to the null moves - were added in the tree search code. Communication to the hardware of the change to and from the quiescence search mode was also added, so the appropriate types of moves could be generated.

Faile contains a function to setup the initial game state at the beginning of the program, and to reset the game state after each game. The code in that function was modified so it could define alternative game states. This was useful for the testing of the hardware move generator in different game situations.

3.3 Conclusions about the State of the Art

It was verified that most prominent hardware chess move generators use the Belle architecture as a basis for their move generator designs. This work does not follow this path, so it can explore the advantages of the bitboard chess board representation.

The approach taken by Marc Boulé in his thesis of designing a hardware move generator to replace the software one in an existing Chess game engine was followed in this work. This removes the burden of having to design and test a Chess engine from scratch.

Chapter 4

Hardware Design

The designed hardware move generator and its interface with the Faile chess engine are detailed in this chapter. The logic components that comprise the move generator are identified and the necessary data conversions between MPAJF and the Faile chess engine are explained.

4.1 Architecture Overview

The purpose of the MPAJF (from the Portuguese thesis title “Multiprocessador para Pesquisas em Árvores de Jogos em FPGA”) Intellectual Property (IP) block is to keep track of the game state, and to generate all possible pseudo-legal moves for that state. The filtering of the illegal moves is done by the Faile chess engine.

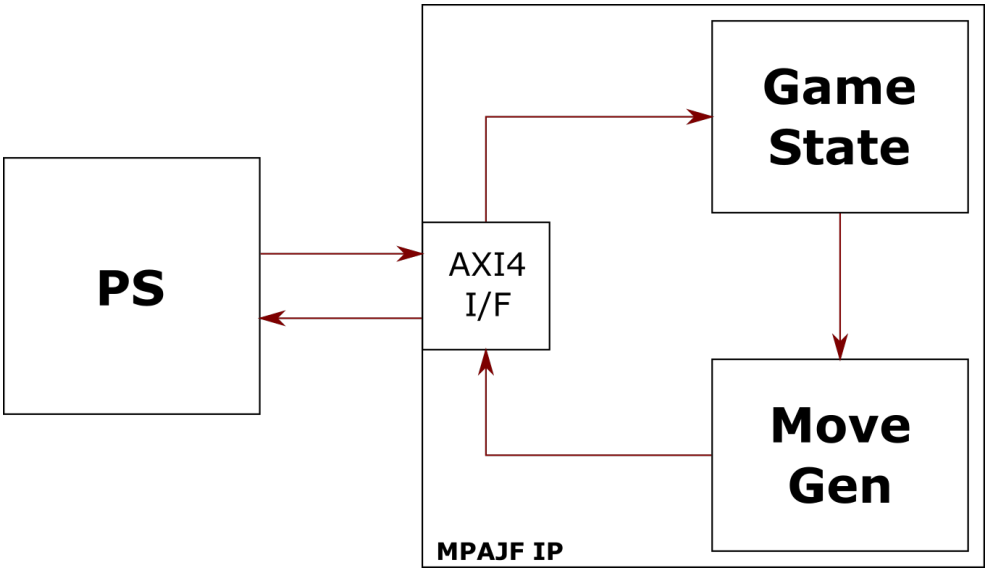

Figure 4.1 MPAJF architecture overview.

As shown in Figure 4.1, the MPAJF IP module is broken into 3 sub-modules: the “Game State”, which contains the hardware to maintain and update the hardware game state; the “Move Gen”, that is responsible for generating the moves; and a Vivado-generated block that provides the necessary logic to interface with the PS by means of the Advanced eXtensible Interface 4 Lite (AXI4) bus.

The arrows in Figure 4.1 show the direction of data flow between the blocks. The Game State block provides the Move Gen block with game state information needed for move generation. The information needed by the Game State block, to maintain its state updated, is received from the PS through the AXI4 interface block, which also provides a data path for the generated moves in the Move Gen block to be transferred to the PS for use by the Faile chess engine.

4.2 Game State

The Game State block maintains the game state, the information needed to identify the pieces and their position on the board, as well as other relevant information necessary for game play and move generation. Maintaining a game state in the hardware is necessary to allow the move generation

logic easy access to it, eliminating the need to copy large amounts of data from the Chess engine each time the game state changes and new moves are required.

In MPAJF, the position of the pieces on the board is saved using bitboards. A bitboard is a piece-centred board representation that consists of an array of 64 bits, one for each of the 64 squares of the board. By maintaining a bitboard for each piece type, one can identify where on the board such pieces exist. Furthermore, a bitboard containing all the pieces of a player can be easily generated by using the OR bitwise operation on all of that player's piece bitboards. The same method can be used on all of the piece bitboards to construct an occupancy bitboard, which contains the positions of all the pieces on the board.

The hardware used to save the game state consists of 12 64-bit registers for the bitboards, one per piece type per colour, an extra register used for the En Passant square bitboard, and for the generation of Castling moves there are also 6 6-bit counters, for maintaining the necessary position information history of the 4 Rooks and 2 Kings.

To ensure that the game state inside the MPAJF IP block is equivalent to that of the Faile chess engine, the game state is initialized at the beginning of the game with data from the PS, and is then updated upon each move, also with data from the PS. To reduce the amount of data transferred between the PS and the PL during the game tree search, where thousands or millions of moves are performed, the PS transfers just the move data instead of the complete state. The initial game state loading operation requires the transfer of 1536 bits of information, whereas the move data only needs the transfer of 32 bits, thus saving time in the transfer but having the drawback of requiring additional hardware logic to effect the move.

Based on the last move, logic determines which side plays next. This information is used by the move generation logic to determine for which side it should generate moves.

4.3 Move Gen

The move generation hardware is input a game state, and generates all possible pseudo-legal moves for that particular state, in an unordered value fashion. It is subdivided into 6 modules, with each one generating moves for a specific piece type and outputting the moves to its own BRAM. This results in moves being generated simultaneously for all 6 piece types.

Although the logic in each of those modules is specific to a piece and its move generation rules, the general layout is common to all. The move generation logic is divided into 2 stages, as shown in Figure 4.2: a "from stage" that generates destination positions for each piece in a bitboard, and a "to stage" that, using the piece origin and destination positions as well as other required game state information, generates the move data and saves it in the BRAM.

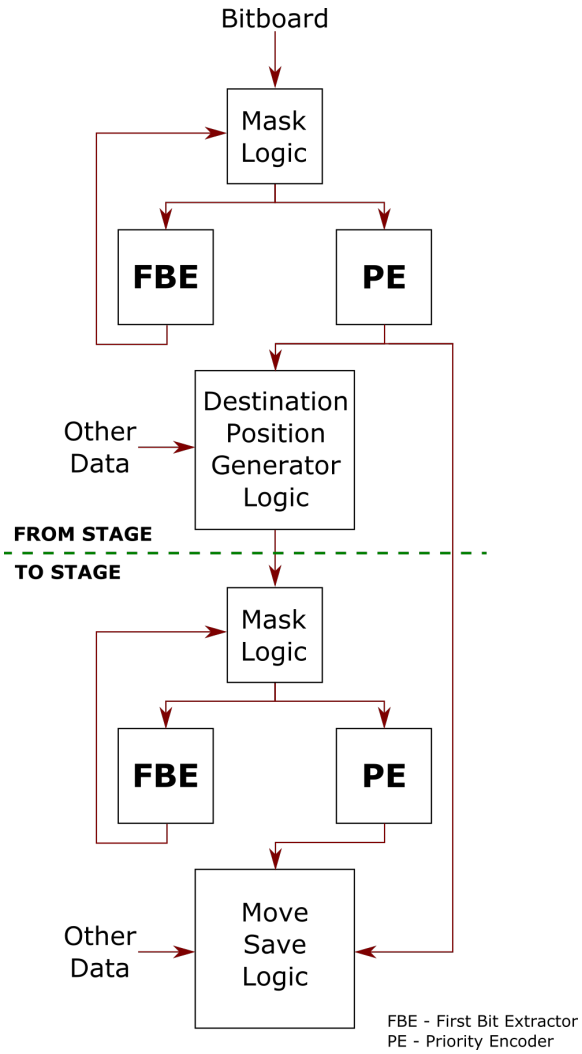

Figure 4.2 Move generator architecture overview.

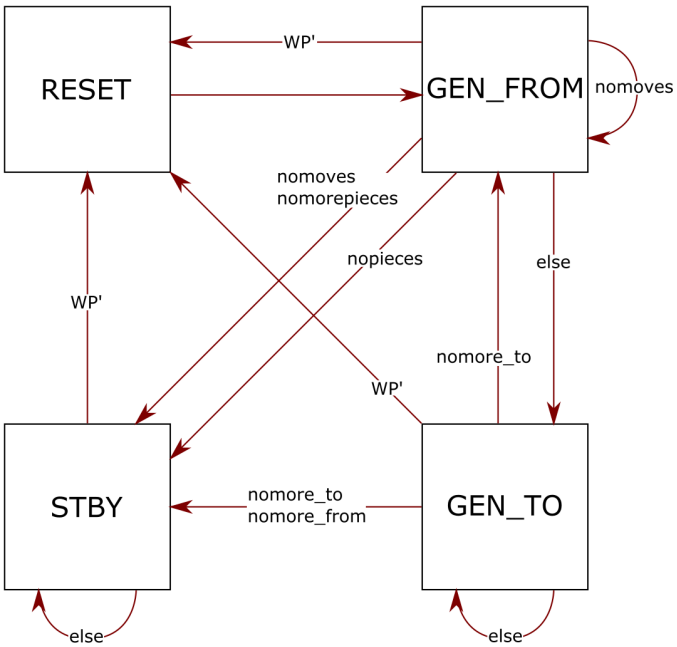

Figure 4.3 Finite-State Machine diagram.

Each piece module contains a FSM to control the operation of the 2 stages. The Moore-type FSM has 4 states: a reset state that setups up the stages and initiates the move generation, one state per stage that runs as needed, and a standby state where the FSM waits after the move generation is completed. The transitions between the 4 states, shown in Figure 4.3, are controlled by the 6 signals detailed in Table 4.1.

Table 4.1 Finite-State Machine control signals.

Signal	Meaning
WP' (reset)	reset signal, triggered when White Plays (WP) changes
nopieces	no pieces were found in the bitboard
nomoves	no moves are available for the current piece
nomorepieces	the last origin position is being processed in the "from stage"
nomore_from	the last origin position has been processed in the "from stage"
nomore_to	the last destination position is being processed in the "to stage"

As the reset state is activated when a change is detected in the side-to-play bit, move generation begins as soon as a new game state is available in the Move Gen block, without the need for additional commands.

As evidenced in Figure 4.2, the stages themselves share the same basic architecture, which consists of logic to sequentially identify, locate and isolate individual *1* bits in a bitboard, as well as logic to mask the found bits for the next cycle. This is accomplished by feeding the bitboard of the piece to which we want to generate moves for, to 2 logic modules.

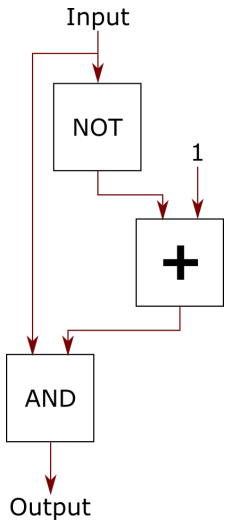

Figure 4.4 First Bit Extractor schematic.

The first module called First Bit Extractor, detailed in Figure 4.4, outputs a bitboard with only one *1* bit, corresponding to the first[4] *1* bit in the input bitboard. The second module, a Priority Encoder, outputs 6 bits corresponding to the position of the first *1* bit in the input bitboard. Those 6 bits identify a

[4] using the order: Least Significant Bit (LSB) to Most Significant Bit (MSB)

square on the board, and provide direct information about the file (bits 0 to 2) and rank (bits 3 to 5) of that square.

The logic to mask the bitboard bits that were already processed consists of a 64 bit register to save a mask, a XOR gate to apply that mask to the bitboard and a OR gate to update the mask with the data from the First Bit Extractor.

At the end of the “from stage”, logic specific to each piece type generates a bitboard containing the destination positions to be used in the “to stage”. The logic to generate the destination positions for each piece is based on the work presented in [23].

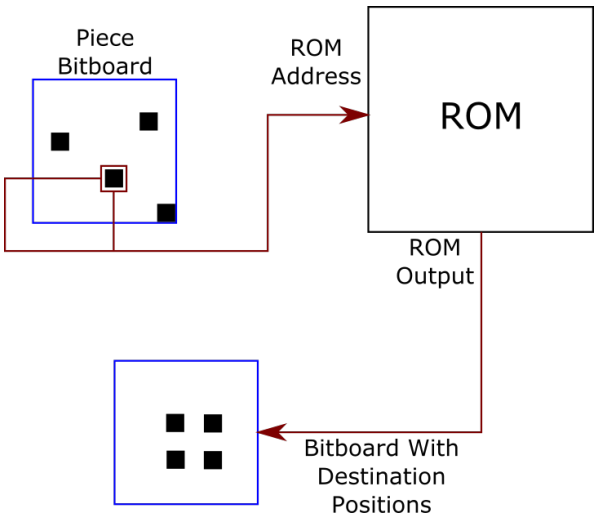

Figure 4.5 Diagram of destination position determination logic for non-sliding pieces.

For the non-sliding pieces, the Pawn, the Knight and the King, the destination positions are primarily obtained by using the position of the piece to index a ROM, as shown in Figure 4.5. A 64-line ROM contains pre-generated bitboards with the piece-specific possible destination positions, for each of the 64 squares on the board.

For the generation of Castling moves, the currently available Castling positions are calculated in the King module - with King and Rook position history data from Game State - and added to the King’s destination positions bitboard. If the King or both Rooks have moved, no Castling is available and no additions are made to the destination positions bitboard.

The destination positions that are output from the ROM in the Pawn module refer only to the possible capture positions available, and as such they must be filtered to only maintain the destination positions which actually contain an opponent piece. For the destination position of the Pawn Push move the current Pawn position is shifted 8 bits, which is equivalent to a one-square movement along a file, and is then checked against the occupancy bitboard to guarantee that the destination position is empty. The logic for the Pawn Double Push move is similar to the “single” Push, but it contains checks to only occur when the Pawn is in its original rank. The logic described above outputs 3 bitboards, that are combined using the OR bitwise operation to obtain the complete Pawn destinations bitboard.

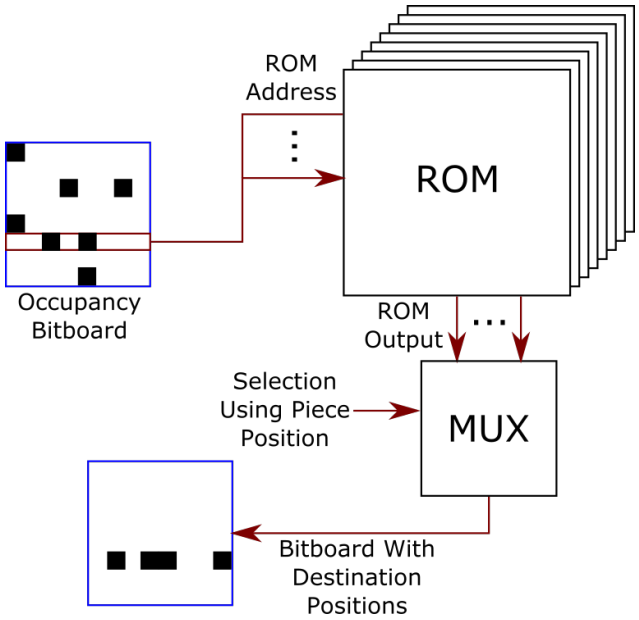

Figure 4.6 Diagram of destination position determination logic for sliding pieces.

The movement of the sliding pieces, the Rook, the Bishop and the Queen, is limited by the occupancy of the rank, file, diagonal or anti-diagonal in which the piece is located. As in any of those directions, there are only 8 squares at most, only 256 different scenarios exist for occupancy in any direction. As illustrated in Figure 4.6, the range of movement for a particular direction can be determined by using the 8 bits of a direction - extracted from the occupancy bitboard - to index a ROM containing pre-generated bitboards indicating to which squares that piece can move. For each direction, 8 such ROMs are need, one for each possible position in the 8 squares. By using this method, both for the file and the rank, the possible movement of the Rook can be obtained. The same applies to the movement of the Bishop, except the diagonal and anti-diagonal information is used instead (for shorter diagonals and anti-diagonals padding is added). The Queen uses this method on all 4 directions, thus needing a total of 32 ROMs.

Through analysis of the information that actually needed to be saved in the ROMs, their size was greatly reduced. Because the piece is always in one of those 8 bits, that bit becomes redundant and the ROM size is halved as the address size is decreased. 2 other bits were removed from the address, further reducing ROM size (see below). As the bitboard being output by the ROMs consists of the bit *0* everywhere, except for the 8 bits of the direction, only those 8 bits need to be saved, leading to an 8-fold reduction in the size of the ROM. Of those 8 bits being output, 1 of them - corresponding to the position of the piece - is always *0*, as the piece cannot move to where it is. The 2 adjacent bits are always *1*, regardless of occupancy the piece can always go there (captures of own pieces is filtered later in logic). This removes another 2 bits both from the address and from the output.

To prevent the generation of destination positions in which pieces of the same colour are located, those positions are filtered at the end of the “from stage” by using the bitboard containing the positions of all pieces of the same colour.

The “to stage” contains logic to address the BRAM and save the generated move information in it, with only the piece-specific move information being saved. Details about the saved move data, and what parts of it each module saves is provided in Table 4.2.

Table 4.2 Move data saved in Move Gen module.

Field	Size	Pawn	Rook	Knight	Bishop	Queen	King
Origin position	6 bits	✓	✓	✓	✓	✓	✓
Destination position	6 bits	✓	✓	✓	✓	✓	✓
En Passant capture	1 bit	✓					
Castling	3 bits						✓
Captured piece	4 bits	✓	✓	✓	✓	✓	✓
Pawn Promotion	4 bits	✓					

The identification of which piece is captured, if any, is obtained by checking which of the opponent piece bitboards contains a piece in the destination position. The Pawn Promotion move - actually 4 different moves - required the addition of a counter to iterate through the 4 possible promotions.

The move generation logic for the sliding pieces required some modifications to the basic architecture: the “to stage” is divided into 2 stages, as the logic needed to generate the destination positions for these pieces posed a penalty for the overall frequency of the circuit. As a result of the extra stage, the FSM has an additional state to control it. To handle the Pawn Promotion move generation, the Pawn module uses a Mealy FSM to control the “to stage” during the generation of those moves.

A quiescence search move generation mode is available, in which only capture or promotion moves are saved.

Table 4.3 Maximum move count estimate.

	Lower estimate	High estimate	Chosen value
Pawn	32	32	32
Rook	125	140	256
Knight	63	80	128
Bishop	83	130	128
Queen	165	243	256
King	8	8	8

In trying to minimize the size of the used logic, an effort was made to bracket the maximum number of generated moves for each piece, so the logic to address the BRAM could by sized accordingly. An exact calculation of these values is extremely difficult due to large number of different positions the pieces can have on the board, the number of possible moves for a piece - up to 27 for each Queen, and the possibility of having up to 10 pieces of a single type on the board simultaneously.

Table 4.3 shows calculated estimates that bracket the maximum number of possible moves for each piece type, and also the actual value used in the hardware. The lower estimate was found by

manually arranging the pieces on the board to find which configurations created the most moves, always considering that both Kings were also somewhere on the board to make it a valid game state. Owing to the large number of possible scenarios, the manual nature of this exercise does not allow conclusions to be drawn about how close this value is to the maximum number of possible moves, but it can be used as the lower end of the range. The high estimate for the maximum number of moves was arrived at by multiplying the maximum number of a piece type on the board by its maximum number of possible moves, providing a number that for most pieces is an impossible number of moves, given that it does not take into account that the pieces would obstruct each other's movements. For the Pawn and the King, the exact maximum number of possible moves was found.

The value chosen to size the BRAM address logic can handle the high estimate on all pieces, except for the Bishop. The small risk of overflow is accepted because of the high spread between both estimates, and the very small chance of the actual maximum value being within 1 move of the high estimate.

4.4 Hardware-Software Interface

The hardware-software communication is done via a data bus using the AXI4-Lite protocol, that allows read and write operations on registers located in the AXI4 hardware block. The data transfers are always initiated from the software side, thus the hardware plays a "passive" role in the communication, while the software plays an "active" role. The AXI4-Lite protocol imposes transfers with a fixed size of 32 bits. For this work, there are 8 32-bit registers in the AXI4 hardware that can be read from, and written to, by AXI4 software functions.

To interface the MPAJF hardware with the Faile chess engine, several functions were created, that act as wrapper functions for the AXI4 read/write functions:

- **MPAJFSetBoardState_Faile** - sends the current game state to the hardware (called at the beginning of the game from the game start function in Faile);
- **MPAJFMakeMove_Faile** - communicates a move to the hardware (called when Faile applies a move);
- **MPAJFUnmakeMove_Faile** - communicates an unmove to the hardware (called when Faile applies an unmove);
- **MPAJFGetMoves_Faile** - copies the hardware generated moves to Faile's move list (the replacement for Faile's own move generator function);
- **MPAJFSetWPEPS_Faile** - used to set the hardware side-to-move information before, and after, Faile does a null move, as well as resetting the hardware En Passant square information after a search;
- **MPAJFSetQSearch_Faile** - used to indicate to the hardware move generator whether a quiescent search is in progress.

In the function *MPAJFSetBoardState_Faile*, the conversion between Faile's 12x12 board representation and MPAJF's bitboard representation is completely made on the software side, as that function is called once, at the beginning of the game, and so it is not time critical. The rest of the conversions, with the exception of the En Passant square information in the *MPAJFSetWPEPS_Faile* function, are done on the hardware side, thus minimizing the conversion overhead during tree search data transfers.

Table 4.4 Breakdown of move/unmove data.

Bits	Data
0-6	Origin position
7-13	Destination position
14	En Passant capture
15-17	Castling
18-21	Captured piece
22-25	Pawn Promotion
26	Move
27	Unmove
28-31	0 (not used)

The move data transferred to the hardware in the *MPAJFMakeMove_Faile* and *MPAJFUnmakeMove_Faile* functions consists of 28 bits, as detailed in Table 4.4.

The size of the move data is not optimal. For example, the Pawn Promotion move has 5 scenarios: promotion to a Queen, to a Bishop, to a Knight, to a Rook and not a Pawn Promotion move, thus requiring only 3 bits instead of the used 4. By using 4 bits, the move data conversion overhead between Faile and MPAJF is minimized, as the piece identification used in Faile can be transferred without conversion. As the minimum data transfer size for the AXI4-Lite protocol is 32 bits, no penalty is incurred as long as the total data does not exceed 32 bits. The position data uses the 7 bits required by the 12x12 board representation used in Faile, and the conversion to a bitboard is done in the Game State module. Bits 26 and 27 indicate if the data is for a move or an unmove.

Before being transferred to the PS by the *MPAJFGetMoves_Faile* function, the hardware generated moves are converted to the data format used by the Faile chess engine. After being read from the BRAMs, the piece coordinates are converted to the 12x12 board representation format and the data is assembled into a 64 bit array, mimicking the move data layout used in Faile. By doing so, the conversion takes place in the hardware at no extra time cost, as opposed to a conversion in software that would require more time. After being transferred to the PS, the *MPAJFGetMoves_Faile* function copies the data to Faile's own variables, thus ending the transfer process. Because the move information is 64 bits in size, 2 32-bit reads are needed to copy a single move from the hardware. The AXI4 logic was modified so it automatically incremented the BRAM address after a move was read, to save an extra data transfer per move to update the BRAM address.

Chapter 5

Results

This chapter provides the results of the implementation of the designed hardware in the Zynq®-7020 FPGA, and of a time performance comparison with the software move generator in the Faile chess engine. The methodology used in the time comparison is explained, as well as the tests performed to ensure that the moves generated by the hardware move generator are correct.

5.1 FPGA Implementation

The implementation of the designed hardware in the Zynq®-7020 FPGA device uses a total of 14241 LUTs and 3962 FFs, or respectively, 26.8% and 3.7% of the total available. The achieved PL frequency is 83.3 MHz, which is equates to a 12 ns cycle duration. The PS is operated at its maximum frequency of 667 MHz.

The minimum hardware move generation throughput, which occurs when each piece only has one possible move, for the 3 sliding pieces combined is 83.3 million moves per second and 125 million moves per second for the 3 non-sliding pieces, giving a combined total of 208.3 million moves per second. This number increases when pieces have more than one move available, as the “from stage” runs less times per each generated move. This gain is more evident in the sliding pieces in which there are 2 “from stages”, e.g., the throughput for a single Queen piece can as high as 77 million moves per second.

5.2 Comparison Setup

To benchmark the speed of the hardware move generator against that of the software move generator in the Faile chess engine, 4 game states where chosen. Game state 1, show in Figure 5.1, was created by Nenad Petrovic [24], and is of interest for being the game state with the greatest number of possible moves known: 218 moves for the White player. Deep Blue versus Kasparov (Game 1), after Deep Blue’s 23rd move [25], is used for game state 2, with the Black player (Kasparov) playing next, Figure 5.2. The third game state was created by the author, see Figure 5.3, and aims to generate a comprehensive list of moves for all types of pieces, as well to serve as an example for an “early game” situation. A total of 54 moves are possible for the White player. Game state 4 is a “late game” scenario, in which few pieces remain on the board, as shown in Figure 5.4.

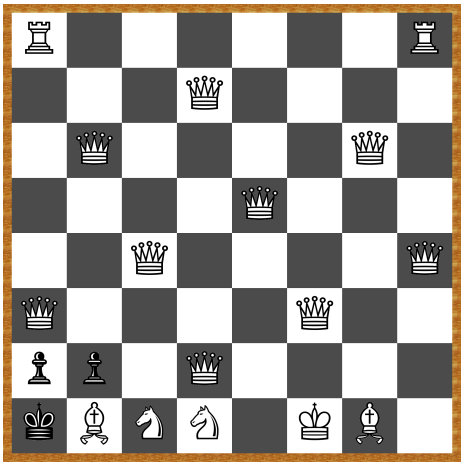

Figure 5.1 Game state 1.

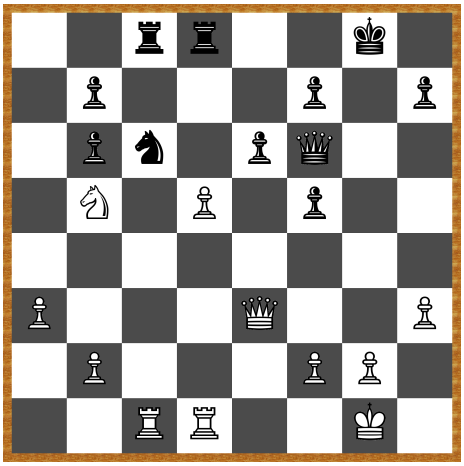

Figure 5.2 Game state 2.

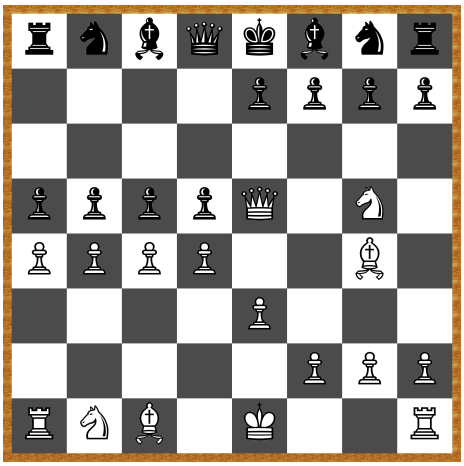

Figure 5.3 Game state 3.

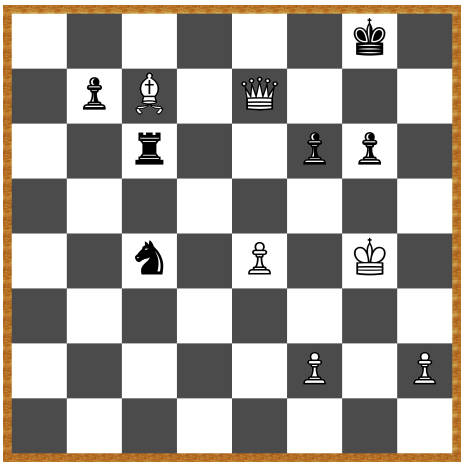

Figure 5.4 Game state 4.

Two separate comparisons were made:

- **Single move** - to compare the time it takes to generate moves for a particular game state. In addition to the measured times, the calculated times for hardware move generation, excluding the PL-to-PS transfer, are also presented;
- **Full search** - to get a performance comparison of the move generators in a tree search environment. This includes the extra time needed to communicate the moves to the hardware to maintain the game state in hardware updated.

In both types of comparisons, the software was compiled with the *-O2* optimization compiler flag. The times were calculated using the XTime functions, available in the Xilinx SDK 2015.2. They allow a pairs-of-cycles count to be obtained, which when divided by half of the CPU frequency gives the time elapsed.

5.3 Single Move Comparison

In single move comparison, times were measured between the start and end of the function call that provides the Faile chess engine with new moves. Essentially, in the case of the Faile chess engine this measures the move generation time, and for the MPAJF it measures the PL-to-PS transfer of hardware generated moves.

To allow a direct comparison of the time each move generator takes, in addition to the times explained above, a calculation of the time needed by the MPAJF hardware to generate moves is also presented. This time is calculated by counting the cycles from the writing of new game state data into the input registers in the Move Gen hardware block, to the writing of the last generated move into a BRAM. The cycle count is then divided by the frequency of the PL, giving the time elapsed.

Table 5.1 6-run average times for single move generation.

Game State	MPAJF (move generation)	MPAJF	Faile
1	2.36 µs	85.7 µs	27.5 µs
2	0.180 µs	15.9 µs	7.96 µs
3	0.264 µs	23.7 µs	11.5 µs
4	0.216 µs	16.2 µs	6.29 µs

The collected times for the 4 game states are shown in Table 5.1. As the average times show, the MPAJF is 2 to 3 times slower when compared with the Faile chess engine. If one compares the MPAJF move generation alone, excluding the data transfer from the PL, we actually see a significant gain in move generation performance.

The strength of the MPAJF move generator is visible in cases where the moves to be generated are spread between the different piece types. In game state 1, the MPAJF move generation time is severely limited by the move generation for the Queen (177 moves versus the second highest move count piece, the Rook with 19). As a result of this, the gains of having moves being generated in parallel for all pieces are overshadowed by the relative high move count of a piece type. This is why game state 1 has the lowest performance against the Faile chess engine, when compared with the other game states, where the move count distribution between the pieces is more even.

5.4 Full Search Comparison

The full search comparison aims to provide a full picture of the performance of the full search of the current setup. The times measured refer to the time duration of the execution of the *think* function in the Faile chess engine, which contains all of the search logic. All the code is the same, except for the MPAJF, for the needed function calls to keep the game state in the hardware updated.

For this comparison, the game tree - of which the test game states are the root node - was iteratively explored to a depth of 6.

Table 5.2 6-run average times for full search.

Game State	MPAJF	Faile
1	1484 ms	681 ms
2	1536 ms	726 ms
3	3595 ms	2375 ms
4	299 ms	158 ms

As is expected after the results obtained in section 5.3, when comparing the times obtained for a full search, shown in Table 5.2, the MPAJF is slower than the Faile chess engine. On average the search with the MPAJF functions takes about twice the time than with the native Faile functions.

5.5 Validation of Hardware Generated Moves

To ensure the correct operation of the MPAJF IP block, the moves generated by it were compared with the ones generated by the Faile chess engine move generator. Tests were performed in several games, played from start to finish, testing the quality of the generated moves under different game states. Two comparisons were made: one to find the MPAJF generated moves in the Faile generated moves, and another to find the Faile moves in the MPAJF moves. This way it was checked that no non-existing moves were being generated and also that no moves were missing.

Also, to guarantee that the game state maintained in the hardware was coherent with that in Faile, comparisons were made with a second hardware game state. After a move, or unmove, was applied to the (main) game state, it was compared with the second game state - derived from Faile's current game state and entirely sourced from the PS - to check if the moves and unmoves were being correctly applied.

Once confidence in the designed logic was established, after many tests in different situations, the test apparatus was removed or disabled.

Chapter 6

Conclusions

This final chapter presents the conclusions drawn from the work done. Possible improvements to the design, to overcome shortcomings and/or to increase performance, are also discussed.

6.1 Results Obtained

A fully functional hardware move generator was developed and successfully tested. By using the Zedboard™ development kit board, the designed hardware move generator was used as a replacement for the software move generator of the Faile chess engine.

As the results in section 5.3 show, there is a time performance bottleneck in the move data transfer from the PL to the PS. Although the move generation in the hardware is very fast, the current way of making that data available to the Faile chess engine does not allow the benefits of that performance to be used. The average speedup for the 4 game states, between the MPAJF move generation alone and the Faile move generation, is greater than 30. A comparison between the MPAJF move generation times with, and without the data transfer, shows that on average the move transfer is over 72 times slower than the move generation itself.

In light of the conclusions presented above, the results in section 5.4 show that a gain can still be achieved in search performance if the move transfer bottleneck is reduced or eliminated. The overhead of the PS-to-PL data transfers - needed keep the game state in the hardware updated - is small enough not to eliminate the large gains made in move generation.

The bitboard board representation is an easy way to maintain a game state, as moving, replacing and removing pieces from the board is done using bitwise operations. For a hardware move generator, bitboards are probably not the best board representation due to its size compared with other board representations. A square-centred board representation could use 256 bits (4 bits per square) instead of the 768 bits used with bitboards for piece positions alone, thus reducing congestion in the move generation hardware.

6.2 Future work

The game tree search speed would increase if the moves are ordered, as the Alpha-Beta algorithm benefits from searching the best cases first. Generating the moves in an ordered fashion, or ordering them after generation, would increase the search speed. A relatively easy way to accomplish a simple move order is to discriminate the moves in capture moves and non-capture ones, and then save them in separate places. The software would then first read the capture moves, which usually produce the most valuable results, followed by the non-capture moves. It is possible that this simple move ordering provides good enough results in the tree search, to allow the removal of Faile's move ordering function, thus gaining time by not ordering the moves in software.

Having the hardware in the PL output the moves to the RAM is a possible way to overcome the identified PL-to-PS transfer bottleneck. Two memory locations to keep the moves would be allocated, instead of the one used by Faile. One of those memory locations would be made available to the hardware, as the location to which it should save the generated moves, and the other would be used by the Faile chess engine. When new moves are requested, the "move generation" function

would just switch have to switch the pointers to the 2 move memory locations. Provisions would have to be in place to ensure mutual exclusion.

To take advantage of the MPAJF's capability to generate moves in parallel for each piece type, the possibility exists to divide the piece bitboards in half, duplicate each of the 6 piece move generator blocks, and input half of the corresponding piece bitboard to each. This would parallelize the move generation even further, although only for cases where 2 or more pieces of the type are located in the separate halves. Obviously, this solution would almost double the FPGA area used, but it would still comfortably fit in the Zynq®-7020 FPGA used in this work.

References

[1] Avnet Inc., "ZedBoard | Zedboard," [Online]. Available: http://zedboard.org/product/zedboard. [Accessed 12 April 2016].

[2] Xilinx Inc., "Zynq-7000 All Programmable SoC," [Online]. Available: http://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html. [Accessed 12 April 2016].

[3] Xilinx Inc., "Zynq-7000 All Programmable SoC Overview (DS190)," [Online]. Available: http://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html#documentation. [Accessed 12 April 2016].

[4] Avnet Inc., "ZedBoard | ZedBoard Technical Specifications," [Online]. Available: http://zedboard.org/content/zedboard-0. [Accessed 12 April 2016].

[5] FIDE - World Chess Federation, "Handbook :: E. Miscellaneous :: Laws of Chess: For competitions starting on or after 1 July 2014," [Online]. Available: http://www.fide.com/fide/handbook.html?id=171&view=article. [Accessed 12 April 2016].

[6] Tangient LLC, "Chess Programming Wiki," [Online]. Available: http://chessprogramming.wikispaces.com. [Accessed 12 April 2016].

[7] Tangient LLC, "Chess Programming Wiki - Belle," [Online]. Available: http://chessprogramming.wikispaces.com/Belle. [Accessed 12 April 2016].

[8] Computer History Museum, "Computer History Museum - Middle Game: Computer Chess Comes of Age - Brute Force vs Knowledge," [Online]. Available: http://www.computerhistory.org/chess/main.php?sec=thm-42eeabf470432&sel=thm-42f15c52333a3. [Accessed 12 April 2016].

[9] F.-h. Hsu, "Two designs of functional units for VLSI based," Carnegie Mellon University, 1986.

[10] Tangient LLC, "Chess Programming Wiki - ChipTest," [Online]. Available: http://chessprogramming.wikispaces.com/ChipTest. [Accessed 12 April 2016].

[11] IBM Research Communications, "IBM Research | Deep Blue | Overview," 23 February 2001. [Online]. Available: http://www.research.ibm.com/deepblue/meet/html/d.3.1.shtml. [Accessed 12 April 2016].

[12] J. Schaeffer and H. van den Herik, Chips Challenging Champions: Games, Computers and Artificial Intelligence, Elsevier, 2002.

[13] F.-h. Hsu, "IBM's Deep Blue Chess Grandmaster Chips," *IEEE Micro,* vol. 19, no. 2, pp. 70-81, 1999.

[14] C. Donninger, A. Kure and U. Lorenz, "Parallel Brutus: The First Distributed, FPGA Accelerated Chess Program," 2004.

[15] C. Donninger and U. Lorenz, "The Chess Monster Hydra," 2004.

[16] Tangient LLC, "Chess Programming Wiki - Brutus," [Online]. Available: http://chessprogramming.wikispaces.com/Brutus. [Accessed 12 April 2016].

[17] M. Boulé, "An FPGA Move Generator for the Game of Chess," 2002.

[18] C. Frayn, "Beowulf Computer Chess Engine," [Online]. Available: http://www.frayn.net/beowulf/. [Accessed 12 April 2016].

[19] T. Riegle, "Crafty Chess," [Online]. Available: http://www.craftychess.com/. [Accessed 12 April 2016].

[20] A. Regimbald, "Homepage of Faile," [Online]. Available: http://faile.sourceforge.net/. [Accessed 12 April 2016].

[21] F. Letouzey, "Chess Programming | Senpai," 12 May 2015. [Online]. Available: http://www.chessprogramming.net/senpai/. [Accessed 12 April 2016].

[22] D. Yang, "Home - Stockfish - Open Source Chess Engine," [Online]. Available: http://stockfishchess.org/. [Accessed 12 April 2016].

[23] D. R. Rasmussen, "Parallel Chess Searching and Bitboards," 2004.

[24] N. Petrović, *Fairy Chess Review,* 1946.

[25] IBM Research Communications, "IBM Research | Deep Blue | Overview," 23 February 2001. [Online]. Available: http://www.research.ibm.com/deepblue/watch/html/c.10.1.html. [Accessed 12 April 2016].