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.

Index Terms—Chess, FPGA, hardware move generator, Faile

I. INTRODUCTION

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.

II. HW/SW PLATFORM

The target System on Chip (SoC) FPGA used in this work is a Xilinx Zynq®-7020 All Programmable SoC, embedded in the Zedboard™ development kit board.

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.

III. CHESS GAME TREE SEARCH

Minimax is one of the most commonly used algorithms in game tree search. It 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.

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.

Alpha-Beta Pruning is an improvement to 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.

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.

IV. STATE OF THE ART

Belle was the first computer built exclusively to play Chess, designed at Bell Laboratories in the 1970s with hardware by Joe Condon and software by Ken Thompson. 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.

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 Chess engine chosen for this work was the Faile chess engine. It 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".

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.

V. HARDWARE DESIGN

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.

As shown in Fig. 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 Chess moves from the game data supplied by Game State; 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, allowing the updates to be done to the game state and the generated moves to be transferred to the PS for use by the Faile chess engine.

A. 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.

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.

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.

B. 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 Block Random Access Memory (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 Fig. 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.

Each piece module contains a Finite-State Machine (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.

As evidenced in Fig. 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. The first module called First Bit Extractor, outputs a bitboard with only one 1 bit, corresponding to the first 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 square on the board, and provide direct information about the file (bits 0 to 2) and rank (bits 3 to 5) of that square.

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 Read-Only Memory (ROM). A 64-line ROM contains pre-generated bitboards with the piece-specific possible destination positions, for each of the 64 squares on the board.

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. 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.

C. 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, functions were created to update the hardware game state and also to retrieve the hardware generated moves. These functions act as wrapper functions for the AXI4 read/write functions.

VI. RESULTS

A. 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 combination of the 6 pieces is 208.3 million moves per second.

B. Tests

Two types of tests were made to compare the performance of MPAJF against that of the move generator in Faile.

In the first one, the times both move generators took to generate moves for 4 distinct game states were measured. On average the MPAJF was over 2.5 times slower than Faile. As in the case of the MPAJF, what was measured is essentially the PL-to-PS transfer times, the move generation-only times, derived from simulation, were also calculated. These times, which do not include the move data transfer, show a speedup greater than 30 when compared with the Faile move generator.

In the second test, times were measured on the complete search procedure, again on 4 distinct game states, to provide a full picture of the performance of the move generators and their integration into the search functions. The times obtained show that the search with the MPAJF move generator takes about twice as long than with the original move generator.

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.

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.

VII. CONCLUSIONS

A. 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 obtained 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.

In light of the conclusions presented above, the results of the second test 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.

B. 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.

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: 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