An FPGA Based Move Generator for the Game of Chess

Marc Boulé and Zeljko Zilic
McGill University, Montréal, Canada
{mboul,zeljko}@macs.ece.mcgill.ca

Abstract

This paper details the architecture of an FPGA chess-move generator. The design is based on *Deep Blue*'s move generator. The inherent differences between ASICs and FPGAs imply many design changes. We present improvements that exploit important FPGA features (lookup-table based logic, routing resources, distributed and block RAM).

1. Introduction

With increasing performance and higher gate-counts, FPGAs are slowly replacing ASICs or even custom ICs. New generations of FPGAs integrate resources such as, on chip RAM, multiple IO standards, microprocessors and advanced clock control circuitry. In this paper, we show how FPGAs are used to improve computer chess playing skills by increasing the processing speed of the alpha-beta search-tree algorithm used in computer chess. We will show how some of the FPGA architectural features can be used to improve the design of the chess move generator.

In 1977, the *Belle* chess system was the first to use custom digital circuits to increase its playing strength (3). The most time-consuming operation performed in a chess program is move generation (7). Along with other chess hardware, *Belle* was able to increase its search speed from 200 positions per second to 160 000. Roughly 1700 ICs were used to construct the chess machine. The move generator used in *Belle* served as a starting point for a more powerful chess machine. The computer that defeated Garry Kasparov in 1997, *Deep Blue*, had 30 IBM RS-6000 SP processors coupled to 480 chess chips (7). This machine was capable of sustained computational speeds of 200 million moves per second, a mere one million times faster than *Belle*.

We describe an FPGA move generator design to improve our chess program *MBChess* (10). The move generator is the core of any search during the game; the critical details of the *Belle* / *Deep Blue* move generators will be presented in Sec. 3. The description of our FPGA move generator follows in Sec. 4. We will show how FPGA architectures influence the design of key hardware arbiters. Additionally, the flexible on-chip RAM resources allow us to develop a novel method of implementing circuitry for masking moves. The FPGA implementation results are presented in Sec. 5. We show that FPGA chess accelerators can successfully be used in areas traditionally reserved for ASICs, without the lengthy turnaround time, and at a fraction of the cost. Ease of re-programmability and on-chip RAM make FPGAs an ideal target for this application.

2. The Move Generator

The basic mechanism used in chess programs is the alpha-beta depth-first search for the best move (7,8). With this search algorithm, a good move ordering is rewarded by the reduction in tree sizes and faster searches. A competitive chess machine must generate moves in a determined order, from best to worst (1,2,8). An example of a good chess move ordering is (best first):

A1: checking moves, direct and discovered;
A2: promotions, if they exist;
A3: capturing moves, ordered by most valuable victim / least valuable aggressor (MVV/LVA);
A4: non-capturing moves.

Intrinsic behaviour exhibited by the alpha-beta algorithm is the beta cut-off (7,8). When a beta cut-off occurs, we simply backtrack from the current node; the remaining branches do not need to be explored. Thus, in order to be efficient, the move generator must also be able to generate moves one at a time independently, given the global fixed move order. This implies that a node must "remember" what moves it has generated so that when the search returns to it, the next unexamined move can be calculated (1,2,3).

3. Previous Designs

3.1. Belle

In *Belle*, the chessboard is an 8x8 array of combinational logic blocks. The main hardware structure in *Belle* deals with the communication between chess squares. Each square has a transmitter, a receiver and a piece register, denoting the current occupant. A given square only transmits to its eight neighbours, except for knight lines that must pass over neighbouring squares. The empty squares are responsible for propagating sliding-piece lines along the different directions. In chess, *aggressor pieces* activate their appropriate transmit lines, given their piece types, whereas *victim squares* receive incoming attacks and apply for arbitration. The two major communication blocks are the transmitter and the receiver. Each square has one of each.

A chess move is a transfer from a source square to a destination square. To construct a move, two cycles are executed. First, a find-victim cycle locates the destination square and then a find-aggressor cycle locates the source piece. During the find-victim phase, all pieces belonging to the player-to-move will activate their transmitters. All opposing pieces and empty squares send the output of their receivers to the arbitration network. A two-level priority tree selects the next victim, ordered from queens to empty-squares (the king cannot be attacked). The first level of arbitration is done on a block of 4x4 squares; the second level selects one of these subgroups. Once a most-valued victim (MVV) is selected, the find-aggressor cycle executes. Here, the victim found in the first cycle transmits as the union of all piece types and the moving pieces' receivers arbitrate to select the least-valued aggressor (LVA), ordered from pawns to kings. This produces the MVV/LVA move ordering. As for the move-masking capabilities of *Belle*, 64 bits (one for each square) are dedicated to mask aggressors exhausted for the given victim, or mask fully searched victims (a square is either a victim or an aggressor). Because of the depth-first search, a stack of 64 levels is used to memorize these mask bits (3). In addition to move masking, *Belle* provides the characteristics A3 and A4 from Section 2.

3.2. Deep Blue

The receiver/transmitter structure of *Belle* was a starting point for the *Deep Blue* move generator. In *Deep Blue*, the first level arbitration is done on rows of squares and the second level selects between rows. Once a square is disabled by bit masking, it no longer participates and other squares with the same priority continue the voting process. This process continues until all possible moves are generated.

The process of masking victims and aggressors is completely different in the *Deep Blue* architecture (1,2). Instead of using a 64-bit stack of 64 levels, mask bits are calculated from the previously generated move at that node. This masking method eliminates the need for memory, at the expense of logic and decoders. In the VLSI design of *Deep Blue*, this approach was advantageous (1). *Deep Blue* arbitration and square masking will be customized for FPGAs in our design and will be presented in Sections 4.3 and 4.4.

The first major improvement introduced by the *Deep Blue* move generator solves the problem of generating checking moves first. The high priority of checking moves in move ordering is very important, as we observed in earlier versions of *MBChess* (10). To implement this feature, two transmitters are used, as well as a receiver with twice the number of inputs (1). During the find-check phase, the pieces for the side to move activate their find-victim transmitters and the opposing king activates its find-aggressor transmitter (union of all piece types). Squares that register appropriate hits from opposing sides will indicate a square from which a piece can check the king. In this paper, we will refer to these as *pivot squares*; the cycle will be called the find-pivot cycle. The second cycle necessary to generate the checking move is an ordinary find-aggressor cycle where the pivot square radiates as the super-piece.

4. The FPGA Move Generator

This section presents our move generator, including FPGA issues critical to the chess move architecture. The fundamental principle of neighbour-square communication is maintained; however, a different method of propagating piece information is introduced.

4.1 The Chessboard Representation

The chessboard is represented as an 8x8 array of chess-square circuits. A block diagram of a chess square is shown in Fig. 1. L-shaped arrows in the transmitter (TX) and the receiver (RX) are 2-bit knight buses; straight arrows are 5-bit neighbour buses. Each square also receives the signals indicated at the top-left of the figure. The dashed lines show two of the many connections between transmitters and receivers. One of the 32 first-level arbiters and one of the 16 second-level arbiters are also visible. The 6-bit depth register is updated using the 5-bit write bus and white-to-move.

Global information, except for the piece propagation output, must be routed to all squares. Fanout required by these nets is unacceptably large, due to the 64 chess squares. To reduce net loading and delay in an FPGA implementation, buffers are added to drive groups of 8 squares. Proper constraints must be added to prevent buffers from being removed during synthesis. The affected signals are white-to-move, a 5-bit write bus and the mask and state information.

In *Deep Blue*, masking and arbitration buses are also used as piece register read/write buses (2). In this design, the write bus is used to write piece values to the different squares. The piece to be written is sent to all squares, but only the square selected by the two square-select lines writes the piece into its piece register. Another select pair is also used to clear a square's piece register. Thus, making and unmaking a castling move or an en-passant pawn capture takes the same number of cycles as for an ordinary move. An important feature of the move generator is that piece registers do not need to be read when making and un-making a move. This decreases the amount of routing resources needed and increases the speed of the design.

4.2. Chess Square - Solving the Checking Move Ambiguity

The transmitter outputs of a square are shown in Fig. 1. During normal move generation, these lines obey the find-victim and find-aggressor behaviours discussed previously.

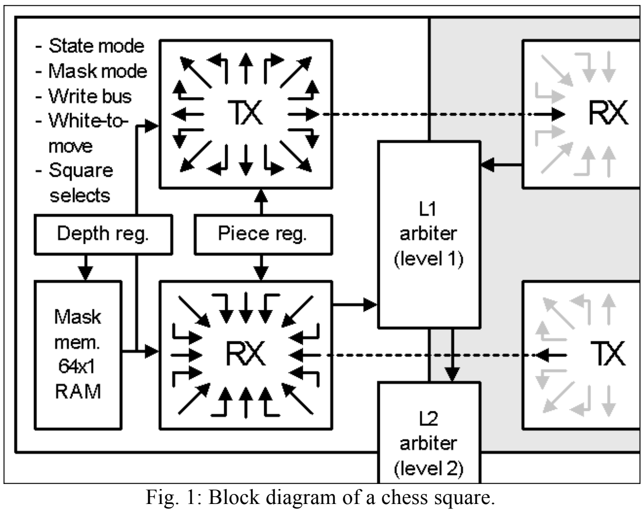

Fig. 1: Block diagram of a chess square.

A potential limitation is observed in *Deep Blue*. Given the transmitters used in the *Deep Blue* design, a pivot square cannot differentiate a queen from a bishop when hit in the diagonal directions. A queen reaching the pivot square may check a horizontally or vertically aligned king whereas a bishop may not. Our interconnection scheme realizes its full potential when considering such checking moves. During the find-pivot cycle, squares must know what kind of sliding pieces are hitting them in order to generate checking moves correctly.

The transmit lines must have the property of additivity: the union of all pieces must also be capable of being broadcast during the find-aggressor cycle. To achieve this goal, an extended piece word of 6 bits is decoded from the 4-bit piece register. The bit fields are presented in Table 1. For example, 111000 represents a white queen and 011111 represents a victim square radiating the union piece. The top 5 bits are sent to the square's eight neighbours, while the knight bit and the colour bit are sent to the knight-reachable squares.

As a result, the chess square circuit does not need two transmitters and double input receivers. All in all, the total amount of interconnects is approximately the same as in *Deep Blue*. During normal find-victim and find-aggressor cycles, input piece colour is unnecessary. However, during the find-pivot cycle, opposing colours that align properly will indicate a checking move. This representation also creates a more uniform interconnect pattern and maximizes information distribution. The find-check operation is therefore a find-pivot cycle followed by a find-aggressor cycle. It is important to note that checking moves do not follow the MVV/LVA move ordering that is implicitly exhibited by the move generator. Therefore, when entering a node for the first time, a loop to find checking moves will be executed. The mask bits will mask off generated moves in a non-regular manner (when a pivot square is exhausted, it will be marked as finished). When entering the normal move generation loop, if the mask bits are not reset, non-checking moves that land on former pivot squares will not be generated. The solution to this is to reset the mask bits between the two phases and ignore checking moves when they are generated during the normal phase.

4.3. The Arbitration Binary Tree

Lookup table (LUT) based FPGAs used in this design (9) impose many physical constraints. The *Deep Blue* receiver outputs represent a large amount of unnecessary routing: 6*64 nets (1,2) for six priority levels. In an ASIC design, a large amount of orthogonal and structured routing can be advantageous if the logic gates needed to decode and operate are fewer. However, in FPGAs, logic functions operate differently. A 5 input XOR gate occupies the same area as any five-input function (2 LUTs). Therefore, receiver outputs are encoded in a 3-bit value, which the arbiters use to compare. An arbiter can select between two squares. The final result is output via a 6-level binary tree structure of arbiters. Each arbiter uses 4 LUTs (any 6 input function) for decision and a maximum of 6 LUTs for multiplexing the winning square's coordinates. This design has shorter propagation times than a row/column-based topology.

Table 1: Internal extended piece register word.

Bit 5	**Bit 4**	**Bit 3**	**Bit 2**	**Bit 1**	**Bit 0**
Colour	Row/column	Diagonal	King	Pawn	Knight

4.4. Move Masking

In Virtex devices (9), the BlockRAM is a 4096-bit synchronous memory. It can be configured for single or dual port usage with variable widths of 1, 2, 4, 8 or 16 bits (with associated depths of 4096 to 256). In our case, 28 blocks are available. Dedicated routing helps route signals to these blocks, which are constrained to each side of the chip. In contrast, DistributedRAM allows a LUT to be used as RAM. One LUT can implement a 16x1 (1-bit wide) synchronous or asynchronous memory. Two LUTs can combine to create 32x1, 16x2 or 16x1 dual port memory. This type of memory is useful when small quantities of local data need to be stored. BlockRAM is used to store larger amounts of data that do not need to travel throughout the chip.

The original *Belle* move masking method (3) is implemented in this architecture. A 1-bit, 64-deep synchronous memory (using two RAM32x1S primitives) in each square is used to memorize mask bits. The buffered signals used to write piece values are also used to update the depth register in each square. Because the design uses few flip-flops, each square has a 6-bit depth register in order to decrease routing. With a memory based mask-bit stack, all the logic implied by the *Deep Blue* move masking decoders is unnecessary. Even with its dedicated routing, BlockRAM is not the best solution to store mask bits in this case. Delays ranging from 4 to 8 ns were observed on nets going-to and coming-from the block memories. With local memory, as described above, these delays are virtually eliminated. However, BlockRAM is the ideal candidate to store the move generated at each level in the search tree. Two memories are needed to store a 32-bit move. The state machine that controls the chessboard is therefore placed near the side of the chip, close to the associated BlockRAMs. This highlights the importance of choosing the appropriate type of RAM resource in a design.

4.5. Non-symmetries

Seldom mentioned in chess hardware literature is how one deals with chess exceptions. The 5-bit omni-directional outputs allow pawn and king propagations to travel two squares in distance. Third and sixth row squares propagate the pawn bit so that the fourth and fifth rows can see double square pawn advances. Furthermore, squares f1, f8, d1 and d8 propagate the king bit so that castling destination squares can signal castling moves. Horizontal pawn outputs are used for en-passant pawn captures and ensure that this pawn

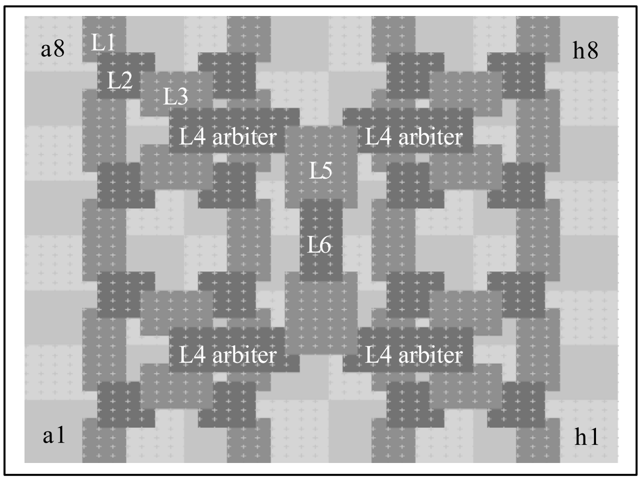

Fig. 2: FPGA floorplan of the silicon chessboard. The 64 squares' area constraints are in light grey and the arbiters' area constraints are in dark grey. Regions overlap in order to get the best device utilization.

exception is well placed in the move ordering. Since pawn promotions have no simple integration to the MVV/LVA move ordering, these moves are treated with a special find-victim-promotion cycle that generates only pawn advances to the last row. To remove redundancy, normal move generation doesn't generate these pawn advances. This must be done because of the disorder promotion moves bring to the mask bits: victims are fully searched before being masked off. This is incompatible with generating promotions first and then continuing with normal move generation.

5. Implementation and Results

The FPGA design was done in VHDL and the chess program was coded in C. A device driver interfaces the FPGA mounted on a PCI card to the chess software. A C program was created to generate the VHDL file responsible for interconnecting 64 instances of chess squares and 63 instances of arbiters. The chip used is an XCV800-4 and the implementation tools are by Xilinx. The Floorplanner was used to inform the place and route tool that the chessboard is an 8x8 array. This reduces implementation time and produces a design with better performance. In this case, a 17% speed increase was obtained. Fig. 2 shows the floorplan used for the chessboard and arbiters. The arbiters are placed between the blocks used as inputs in order to minimize routing delays. The entire design uses approximately 7000 LUTs, 1000 flip-flops, 2 BlockRAMs and runs at 33MHz.

The chess move generator also includes a PCI interface to connect it to the computer running *MBChess*. Many different commands allow the communication overhead to be diminished. For example, in a single read from the card, the move generator can be instructed to undo the currently stored move, generate and return the next move and execute that move on its hardware chessboard. This simultaneous write-and-read is possible when part of the address is used to pass a command rather than address memory locations. Table 2 presents results obtained for 33MHz PCI bus speeds. When expressed in the same metric as in (2), it can generate approximately 8 million moves/s.

Table 2: FPGA move generator performance, 33MHz bus speed.

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

Effects of the hardware move generator on the playing strength of the author's program, *MBChess*, will be assessed through chess tournaments. Anticipated improvements are in the range of hundreds of chess rating points.

6. Conclusions and Future Work

This project presents an application of FPGAs in computer chess. The fundamental difference in resources between ASICs and FPGAs brought forth solutions that are well suited for FPGAs. The embedded RAM found in modern FPGAs was shown to play a key role. The Floorplanner also helped to increase the speed of the design. With the simpler interconnect scheme, chess asymmetries and the problem of differentiating checking pieces were resolved. As hard-processors become the norm in upcoming FPGAs, integration of program code to the IC will make the FPGA a complete and even more powerful solution.

At the time of this writing, the next mainstream Human-Computer chess match up is scheduled for the first quarter of 2002. It will pit Vladimir Kramnik, current world chess champion, against *Deep Fritz*. The chess hardware will consist of a multiprocessor supercomputer with no special purpose hardware. The complete system can calculate over 4 million moves per second (6). What *Deep Fritz* lacks in processing speed, it makes up in chess knowledge and optimized programming. We believe that the proposed FPGA move generator could be used to accomplish similar performance at a tiny fraction of the cost.

References

(1) F.-H. Hsu, "IBM's Deep Blue chess grandmaster chips," *IEEE Micro*, vol. 19, no. 2, 1999, pp. 70-81.

(2) F.-H. Hsu, "A two-million moves/s CMOS single-chip chess move generator," *IEEE Journal of Solid-State Circuits*, vol. sc-22, no. 5, 1987, pp. 841-846.

(3) J.H. Condon and K. Thompson, "BELLE chess hardware," *Advances in Computer Chess 3*, 1982, pp. 45-54.

(4) H.J Berliner and C. Ebeling, "Hitech," *Computers, Chess and Cognition*, 1990, pp. 79-109.

(5) J. Testa and A.M. Despain, "A CMOS VLSI chess microprocessor," *IEEE Custom Integrated Circuit Conference*, 1990, pp. 15.3.1-15.3.4.

(6) "World champion to battle chess supercomputer," http://www.cnn.com/2001/TECH/ptech/08/02/chess.battle.idg/.

(7) M. Newborn, "Kasparov versus Deep Blue," *Springer-Verlag*, New York, 1997.

(8) P.W. Frey, "Chess skill in man and machine," *Springer-Verlag*, New York, 1977.

(9) "The programmable logic data book 2000," *Xilinx*, San Jose, California, 2000.

(10) M. Boulé, *MBChess v9.01*, http://www.macs.ece.mcgill.ca/~mboul.