An FPGA-based Othello Endgame Solver

C.K. Wong, K.K. Lo and P.H.W. Leong
Department of Computer Science and Engineering,
The Chinese University of Hong Kong
Shatin, N.T., Hong Kong
{kitwong, loka}@alumni.cse.cuhk.edu.hk, phwl@cse.cuhk.edu.hk

Abstract

A single chip FPGA-based Othello endgame solver is presented in this paper. The solver includes all the hardware for move checking, disc flipping, move selection, board evaluation and alpha-beta pruning. On a Xilinx Virtex XCV1000E-6 device operating at 50 MHz, the chip can search 3.14 million Othello positions per second. The endgame chip achieves a speedup of 3.5 over an 800 MHz Pentium III machine, showing that performance similar to that of a high end microprocessor can be achieved using modest FPGA resources. By using a larger FPGA, a more sophisticated search algorithm and an improved datapath, we believe that a single FPGA based endgame solver with at least two orders of magnitude better performance can be developed.

1. Introduction

Big Blue was a high profile IBM project which defeated the World Chess Champion, Garry Kasparov by using a high speed VLSI chess chip which could search 100 million chess positions per second [12]. Othello is a game which is simpler than chess and one in which computer programs are already stronger than the human world champion [15]. It is our belief that field programmable gate array (FPGA) devices can be used to accelerate the searching required in an Othello playing program, leading to improved playing strength.

In this paper, we present a design and implementation of an FPGA based endgame solver for Othello. In the endgame, the entire game tree is searched and hence an optimal move can be computed. If a hardware accelerated endgame solver can be developed, and the game tree can be searched to a greater depth than software in the same time, playing strength can obviously be improved.

There are several key components our Othello endgame solver: a legal move generator which finds all legal moves in parallel given a board position; a disc flipper which executes a move to produce a new board position; a board evaluator which assigns a score to a terminal board position; and a search engine which uses the previous blocks to traverse the game tree using the alpha-beta pruning algorithm to find the optimal move. We present novel architectures to implement these blocks and compare the performance of the resulting system with a software based implementation.

To the best of our knowledge, the only previously reported hardware Othello machine design was proposed in 1987 by Hewlett [1]. In this design, 30 Signetics PLS101 field programmable logic array (FPLA) devices were used to develop a hardware disc flipper which takes a board position and move position as inputs and produces an updated board position as output.

This paper is organized as follow: In section 2, the rules of Othello, Othello playing programs and a brief review of the alpha-beta search algorithm are presented. In section 3, the hardware implementation of the whole system and some small main modules are presented. In section 4, results are given. Conclusions are drawn in section 5 and some ideas for further research in this area are described in section 6.

2. Computer Othello

Othello is a two player game, played on an 8×8 board of 64 squares with discs which are white on one side and black on the other. Outflanking involves placing a disc of your own color so that you have your color disc at each end of a line of discs (in the horizontal, vertical or diagonal directions), then all of your opponent's discs which have been outflanked (in all directions) are flipped to your own color. Players consecutively take turns unless no legal move is available to a player in which case the player skips his/her turn. If neither player has a legal move, the game is over. When the game is over, the player with the most discs on the board is the winner. Please refer to [13] for a more detailed description of the rules.

In a two player game such as Othello, the possible outcomes can be represented as a directed graph which is

known as the game tree. A node in the game tree represents a state (or position) of the game, and the children of that node are the states reached by making a particular move. For endgame analysis, an evaluation score is given to all leaf nodes, namely the difference in number of discs between the players, representing the winning margin in a game of Othello.

Given a game tree and the current state, a search can be made to determine the best move. This optimal move is the one which minimizes the maximum value of possible replies to that move (i.e. the move to which the opponent has the least best reply). The minimax algorithm is the naïve implementation of this search and requires a complete transversal of the tree.

Strong Othello programs use sophisticated search techniques and pay particular attention to move ordering which serves to increase the number of nodes which can be pruned from the search, hence improving the search speed. In the early 80's, Iago [4] used iterative deepening and killer heuristics. Bill [3] used iterative deepening, hash and killer tables, a zero-window search and a two-phase endgame search. Logistello used the Multi-ProbCut algorithm which improves search speed by a factor of ten compared with alpha-beta pruning [5, 16].

Although we intend to explore more sophisticated search algorithms such as Multi-ProbCut in the future, a simple algorithm, namely alpha-beta pruning, was chosen so as to facilitate a simple hardware implementation. More sophisticated searches can be implemented by modifying the search finite state machine without requiring modifications to other parts of the system.

The alpha-beta pruning algorithm is able to prune the search space by avoiding unnecessary subtrees and can be described in pseudocode as follows:

```
abeta (node, alpha, beta)
{
    if node is a leaf
       return the value of node
    if node is a minimizing node
       for each child of node
          beta = min (beta,
                abeta (child, alpha, beta))
          if beta <= alpha
             return alpha
       return beta
    if node is a maximizing node
       for each child of node
          alpha = max (alpha,
                abeta (child, alpha, beta))
          if beta <= alpha
             return beta
       return alpha
}
```

Figure 1. Alpha-beta pruning algorithm pseudocode.

3. Hardware Implementation

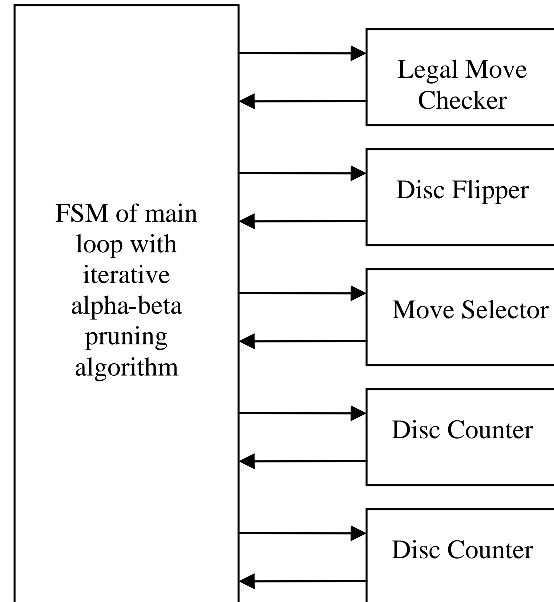

Figure 2. System block diagram

A block diagram of the endgame solver is given in Figure 2. We had originally intended to implement the search algorithm on a microprocessor and the rest of the machine as a coprocessor on an external FPGA board, but found that such an approach imposed a performance bottleneck due to the large amount of I/O required. Thus the alpha-beta pruning algorithm was implemented on-chip using a finite state machine (FSM). An iterative alpha-beta pruning algorithm was derived from the recursive code of Figure 1, stacks being used to store states for implementing the recursion.

In each iteration of the search's main loop, all legal moves are determined in parallel by the Legal Move Checker and a bit-array returned. The Move Selector is used to select a move and the board position is updated using the Disc Flipper. The two Disc Counters compute the number of white and black discs on the board. The above process is repeated in accordance with the alpha-beta pruning algorithm, and when a leaf node is reached, the value of the node is simply the difference in number of white and black discs.

Computation in the FSM proceeds according to the node type which can be: a leaf node, a tree node, a pass node and a search ended tree node. In each iteration, 1 of the 4 states below is executed depending on the characteristic of the current node. If the current node is a game ended node, the leaf node state is executed and the score calculated. If the current node is not game ended, there are still some legal moves to consider and the tree

node state is executed and alpha and beta determined. If beta <= alpha, searching will cut off. Otherwise, one of the legal moves would be taken and a deeper search initiated. If the player of the current node has no legal moves and has to pass, the pass node state is executed. If all the legal moves of the current node have been tested, it will execute the search ended tree state. This is equivalent to the return function in the recursive algorithm and is used to return the score of the child node to the parent node.

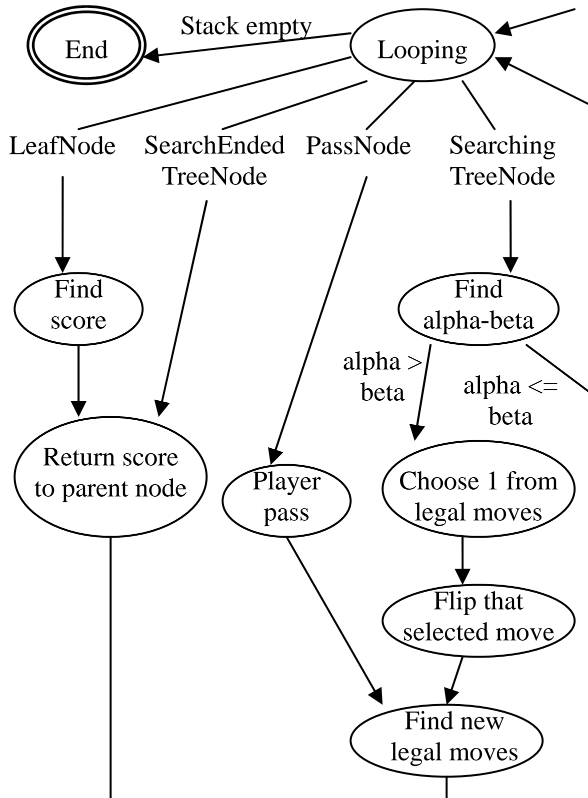

Figure 3. State diagram of FSM

3.1. Legal Move Checker

The Legal Move Checker is used to generate an array representing all legal moves for the current board. It is implemented purely in combinatorial logic as a sum of products.

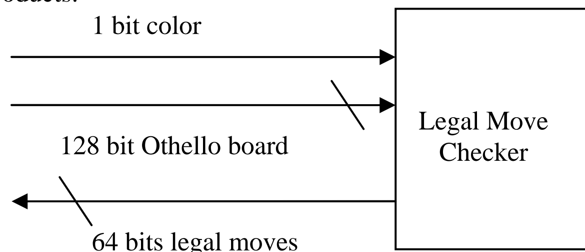

Figure 4. Interface of legal move checker

The Legal Move Checker takes a color bit (indicating whether it is white or black's turn) and a 128 Othello board (in which two bits are used to indicate whether a square is empty, white or black) as inputs, and generates a 64-bit output which indicates which positions are legal moves.

In order to determine whether the square is a legal move, we first consider the subproblem of determining whether a move in a particular direction is legal. Since an 8×8 square board is used, in any direction, at most 8 squares need be tested. This leads to at most 6 cases in which a legal move can occur and a Boolean expression of these situations in which a white disc is legal is:

$$\overline{W0} \bullet \overline{B0} \bullet \left\{ \begin{array}{l} (B1 \bullet W2) \cup \\ (B1 \bullet B2 \bullet W3) \cup \\ (B1 \bullet B2 \bullet B3 \bullet W4) \cup \\ (B1 \bullet B2 \bullet B3 \bullet B4 \bullet W5) \cup \\ (B1 \bullet B2 \bullet B3 \bullet B4 \bullet B5 \bullet W6) \cup \\ (B2 \bullet B2 \bullet B3 \bullet B4 \bullet B5 \bullet B6 \bullet W7) \\ \end{array} \right\}$$

Relative positions are assumed and the above equation, which we call a length 8 legal move checker, says that it is legal to use a white piece at position W0 if there is no piece already present AND it outflanks at least one black piece (where, for example, B1 indicates that there is a black disc in the adjacent square and W2 indicates there is a white disc adjacent to the black disc). Some additional circuitry is used to allow both black and white cases to be handled, this being controlled by the color bit. Using the above technique, a unidirectional legal move checker can be constructed.

When considering all 8 possible directions from a square, 8 unidirectional legal move checkers of different lengths can be combined. In order to generalize the design, the Othello board can be divided into 10 different types of squares.

	A	B	C	D	E	F	G	H
1	A	B	C	D	D	C	B	A
2	B	E	F	G	G	F	E	B
3	C	F	H	I	I	H	F	C
4	D	G	I	J	J	I	G	D
5	D	G	I	J	J	I	G	D
6	C	F	H	I	I	H	F	C
7	B	E	F	G	G	F	E	B
8	A	B	C	D	D	C	B	A

Figure 5. 10 kinds of legal move checkers

Table 2 indicates the number of AND terms in the legal move checker equation for each of the 10 different shaded square types in Figure 5.

Type	Direction							
	R	BR	B	BL	L	TL	T	TR
A	6	6	6					
B	5	5	6					
C	4	4	6	1	1			
D	3	3	6	2	2			
E	5	5	5					
F	4	4	5	1	1			
G	3	3	5	2	2			
H	4	4	4	1	1	1	1	1
I	3	3	4	2	2	1	1	1
J	3	3	3	2	2	2	2	2

R=right, BR=bottom-right, B=bottom, BL=bottom-left
L=left, TL=top-left, T=top, TR=top-right

Table 2. Table of number of product terms in legal move expression for different groups of squares.

From the above table, it can be seen that the legal move checker equation for some square types such as A need only consider 3 directions whereas others need to consider all 8 directions. Also, since type J positions are already occupied at the start of the game, they do not need to be considered. To implement legal move checking for the non-shaded squares of Figure 5, symmetry is used.

3.2. Disc Flipper

The Disc Flipper is used to flip the outflanked pieces when a move is made. The disc flipper is a combinatorial circuit which propagates a signal from the position of the newly placed disc, in all directions in parallel, until it can be determined whether outflanking occurs.

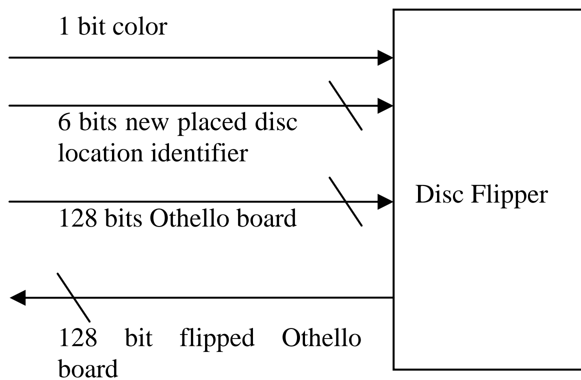

Figure 6. Interface of disc flipper

128 bits are used to represent the original Othello board, 6 bits to represent the row and column of the newly placed disc, and 1 bit is used to identify the color of the newly placed disc. The disc flipper returns a 128 bit flipped Othello board.

In the disc flipper design, a ripple chain is used. A chain can be either a row, a column or a diagonal in a particular direction with a starting and ending square so 8 different directions (right, left, top, top-left, top-right, bottom, bottom-left and bottom-right) need to be considered.

Each chain is implemented in 3 phases: forward, backward and flip phases.

Forward Phase:

a) The newly placed disc triggers a forward signal to its neighbor square in a given direction (e.g. Left chain => left neighbor)
b) The neighbor square propagates the forward signal to the neighbor in the same direction if it contains an opposite color disc.
c) The propagation continues until the forward signal reaches a square containing a disc with the same color as the newly placed disc. In this case an endchain signal and a backward signal are generated. The phase is changed to the backward phase.

Backward Phase:

a) The generated backward signal is propagated to its neighbor in the reverse direction (e.g. Left chain => right neighbor)
b) The neighbor square propagates the backward signal in the opposite direction of the chain's name if it contains an opposite color disc.
c) The propagation continues until the backward signal reaches the square containing the newly placed disc.

Flip phase:

a) All squares having both forward and backward signals are changed to the color of the newly placed disc, hence flipping the outflanked discs.

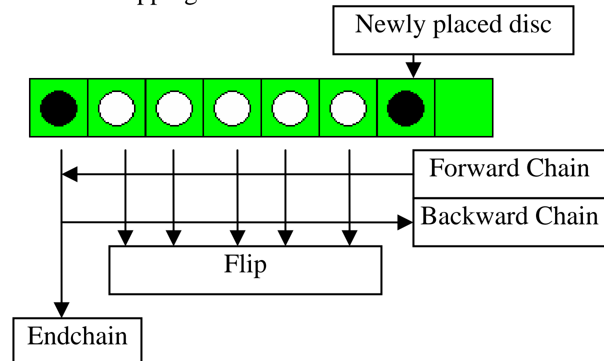

Figure 7. Example of chain concept in disc flipper

A newly placed disc generates 8 forward signals in the different directions. If all the 8 directions can be flipped, there would be a forward signal propagation line and a backward signal propagation line in each chain. If outflanking does not occur in a direction, the propagation of the forward signals in those directions are stopped by empty squares or board edges and no backward signals are returned and they are not flipped.

After finding the squares that require flipping, the outflanked squares are flipped and the newly placed disc is added to the flipped board.

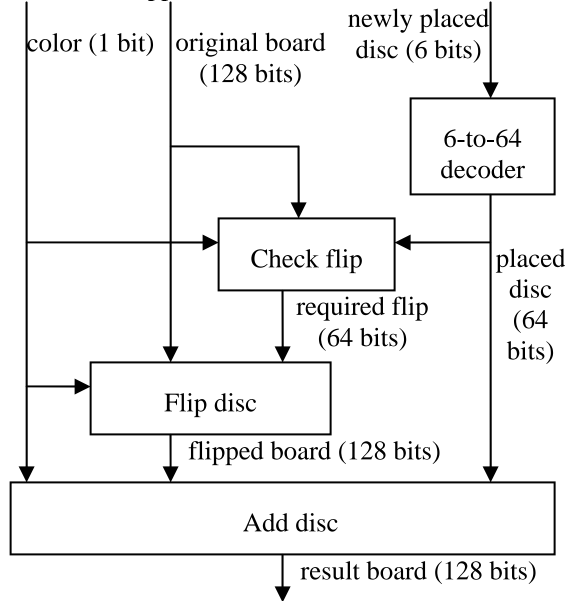

Figure 8. Block diagram of disc flipper

3.3. Move Selector

For every branch of the game tree, the player must select one of the legal moves from the output of the legal move checker. In order to increase the amount of pruning, it is best to consider the best moves first. Moves are thus selected by the Move Selector based on position.

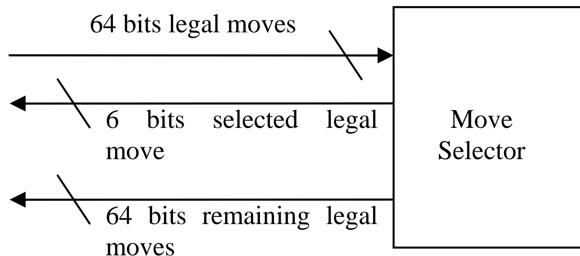

Figure 9. Interface of move selector

Squares are assigned different priorities based on their position. For example, higher priority is given to the corner squares because they cannot be flipped once placed and hence serve as anchors for outflanking the opponent's pieces.

Using letters to represent columns and numbers to represent rows (as illustrated in Figure 5), the corner squares A1, A8, H1 and H8 are given the highest priority. Consequently, squares B2, G2, B7 and G7 are given lowest priority since placing a disc on that square makes it possible for the opponent to gain a corner square. Priorities are assigned as shown in Figure 10.

	A	**B**	**C**	**D**	**E**	**F**	**G**	**H**
1	B	I	C	E	E	C	I	B
2	I	J	H	G	G	H	J	I
3	C	H	D	F	F	D	H	C
4	E	G	F	A	A	F	G	E
5	E	G	F	A	A	F	G	E
6	C	H	D	F	F	D	H	C
7	I	J	H	G	G	H	J	I
8	B	I	C	E	E	C	I	B

Figure 10. 10 classes of squares with class A for highest preference for taking moves and class J for lowest

The board is classified into 10 regions with A having the highest priority and J the lowest.

However, as squares in class A are the center squares, which are always occupied, only classes B to J need be considered.

The input to the move selector is a 64 bit signal where each bit in the signal indicates the respective location on the board where a legal move can be made. These inputs are generated by the legal move checker.

In our design, the move selector is a priority encoder implemented in a hierarchical fashion. The basic components used in the Move Selector are 4-Input priority selector and 4-to-1 priority MUXs as shown in Tables 3 and 4 together with Figures 11 and 12.

Figure 11. 4-input priority Selector

In[3:0]	Enable	Outcode[1:0]
0000	0	XX
1XXX	1	00
01XX	1	01
001X	1	10
0001	1	11

Table 3. The logic table of 4-input priority selector

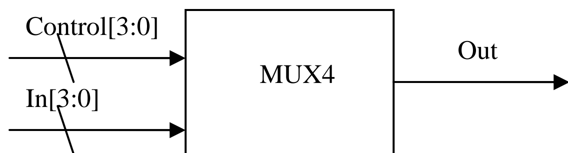

Figure 12. 4-to-1 priority MUX

Control[3:0]	Out
0000	X
1XXX	In[3]
01XX	In[2]
001X	In[1]
0001	In[0]

Table 4. The logic table of 4-to-1 priority MUX

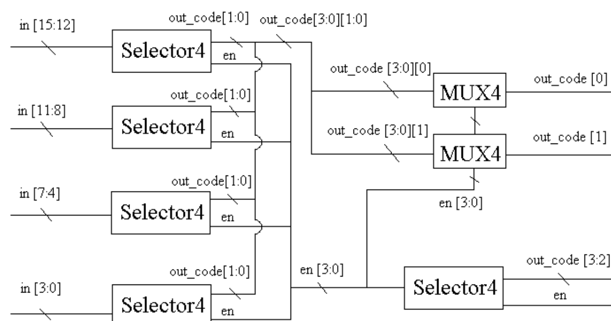

Figure 13. Schematic of 16-input priority selector

By using five 4-input priority selectors and two 4-to-1 priority MUXs, a 16-input priority selector can be constructed as shown in Figure 13. Similarly, by using four 16-input priority selectors, one 4-input priority selector and four 4-to-1 priority MUXs, a 64-input priority selector can be implemented.

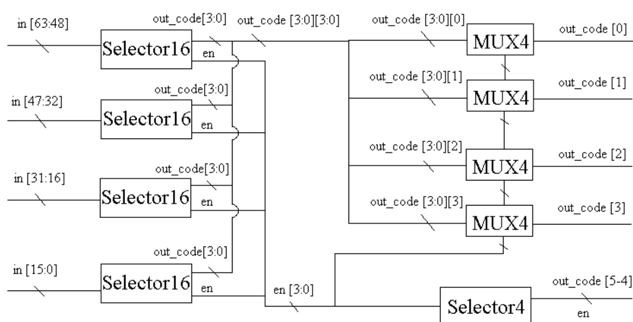

Figure 14. Schematic of 64-input priority selector

The inputs of the 64-input priority selector are arranged according to the priority scheme of Figure 10. As a result, an Input Converter is required to map the input signals to the priority scheme and an Output Converter is required to convert the selected move back to its actual square number on the Othello board as shown in Table 5.

Priority Mapping Scheme	
Class	Squares involved
A	27,28,35,36
B	0,7,56,63
C	2,5,16,23,40,47,58,61
D	18,21,42,45
E	3,4,24,31,32,39,59,60
F	19,20,26,29,34,37,43,44
G	11,22,25,30,33,38,51,52
H	10,13,17,22,41,46,50,53
I	1,6,8,15,48,55,57,62
J	9,14,49,54

Table 5. Preference Scheme

3.4. Disc Counter

A Disc Counter is used to count the number '1' signals within the 64 bit input and a simple tree based adder is used.

3.5. Host Interface

A Pilchard reconfigurable computing card which uses a memory slot interface [14] was used to implement the design.

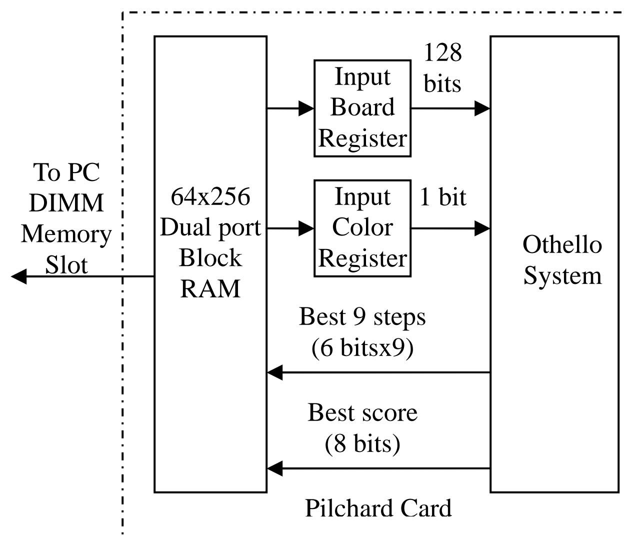

Figure 15. System Interface

In order to control the endgame chip, an ANSI C program is used to read and write from a 64x256 dual port block RAM on the FPGA. The Othello system accesses the dual port block RAM to do its I/O.

4. Results

The Othello endgame solver was implemented in VHDL with automatic Place-and-route (PAR). Simulation was done using Modelsim PE 5.5f and the result was verified to be correct by comparison with a C program running on a Pentium processor. Synthesis, PAR, bit-stream implementation and timing measurement were done using Xilinx ISE 6.1 package. The target FPGA was a VirtexE XVC1000E-HQ240 device with speed grade -6. The design occupied 36% (4428/12288) of slices and 25% (24/96) of block RAMs resources.

The move generator had a maximum delay of 19.902 ns and can operate at a maximum frequency of 50.246 MHz.

For the iterative alpha-beta pruning algorithm, the nodes being processed can be can be classified into 5 different types: Leaf Node, Search Ended Tree Node, Pass Node, Searching Tree Node and Cut-off Node.

Each type of node requires a different number of cycles as described in Table 6.

Node Type	Cycles
Leaf Node	4
Search Ended Tree Node	4
Pass Node	4
Searching Tree Node	16
Cut-off Node	3

Table 6. Cycles required for different node types

For each iteration, 3 cycles are used for score comparison. 6 cycles for initialization and 2 cycles would be used in the end stage. Thus the total number of cycles can be computed with the formula:

Total Cycle = 6 + 3 x ({LeafNode} + {TreeEndNode} + {PassNode} + {CutOffNode} + {TreeNode}) + 4 x {LeafNode} + 4 x {TreeEndNode} + 4 x {PassNode} + 3 x {CutOffNode} + 16 x {TreeNode} + 2

As an example, in a case with 15 empty squares, there are a total of 11,207,919 nodes with 42% (4,474,594) being Searching Tree Nodes, 12% (1,316,494) Leaf Nodes, 34% (3,861,848) Search Ended Tree Nodes, 4% (435,747) Pass Nodes and 8% (851,236) Cut-off Nodes.

The most time consuming node is the Searching Tree Node which is used to select 1 legal move and take that move to search 1 step deeper.

Percentage = cut-off ÷ [cut-off + searching]

Cut-off percentage of the above example is 15.2%. The cut-off percentages for other board examples are close to 15%. As a result, the iterative alpha-beta pruning algorithm used has a cut-off percentage of about 15%

4.1. Comparison with software

Table 7 shows the average execution time (in seconds) of the FPGA-based and the software-based end-game move generator with alpha-beta pruning algorithm for different numbers of empty squares. The FPGA-based move generator was found to be faster than the software-based implementation by a factor of 3.67.

No. of Empties	Execution Time (second)		Speed up
	Software-based	FPGA-based	
16	6	1	6
17	26	7	3.71
18	44	12	3.67
19	451	120	3.76
20	1378	375	3.67

Table 7. Performance comparison between FPGA-based and software-based move generators with alpha-beta pruning.

5. Conclusion

In this paper we presented a novel architecture and implementation for an Othello end game analyzer. The move generator used 4428 (36%) Slices and 24 (25%) Block RAMs of the XCV1000E-HQ240-6 device. Its highest operation frequency is 50 MHz. Using 16 cycles per search position, it can search 3.14 millions Othello positions per second. By comparing the performance with the software-based recursive alpha-beta pruning of different number of empties, a speed up factor 3.67 has been found which is independent of the number of empties on the board.

This work demonstrates that it is possible to create a complete search engine on a relatively small FPGA which has performance similar to that of a high end microprocessor. By using a larger device, higher clock rate and a more sophisticated search algorithm, we expect a performance improvement of two or more orders of magnitude can be achieved using the same architecture.

6. Future work

A larger device such as the XCV6000E has six times more slices than the XCV1000E used in this design. Furthermore, our design currently only occupies 36% of the XCV1000E and hence we expect to be able to use more area to improve parallelism and achieve a twelve-fold speedup over the present design. The finite state machine used in the design is far from optimal, the

bottleneck being the disc flipping circuit. A synchronous design methodology forced us to implement a disc flipping operation in several cycles. A faster design which uses a fully combinatorial disc flipper and waiting multiple cycles for its completion would speedup the design by a factor of two and allow a higher clock rate.

For a 20 empties case, a sophisticated software endgame solver [17] performs its analysis in 17 seconds, which is 22 times faster than the alpha-beta scheme on the same Pentium machine. This program uses a fastest-first heuristic which sorts the moves in increasing order on the number of available opponent responses and results in a much higher cut-off percentage than alpha-beta pruning. Furthermore, a hash table is used to avoid reevaluating already visited nodes. With an added cost of greater hardware complexity, the same algorithm can be implemented in hardware which would greatly reduce the search time. When implementing more sophisticated search algorithms, it may be better to use an on-chip microprocessor such as the Microblaze or PowerPC to replace the FSM of the current design.

We also wish to implement a full evaluation function which can assign scores to non-leaf node positions, considering issues such as position, disc stability and parity [2,3,4,5]. With this feature the machine can also be used for the midgame and the benefits of FPGA based hardware acceleration can be utilized in all computationally expensive stages of play.

References

[1] Clarence Hewlett, “Hardware Help in an Othello Endgame Analyzer”, in Heuristic Programming in Artificial Intelligence: the first computer Olympiad, pp. 219-224, 1989

[2] Anders Kierulf, “New Concepts in Computer Othello: Corner Value, Edge Avoidance, Access, and Parity”, in Heuristic Programming in Artificial Intelligence: the first computer Olympiad, pp. 225-240, 1989

[3] Lee, Kai-Fu, and Mahajan, Sanjoy, "The Development of a World Class Othello Program,”Artificial Intelligence, 1990, Vol. 43, pp. 21-36.

[4] Rosenbloom, P. S., "A World-Championship-level Othello program," Artificial Intelligence, 1982, Vol. 19, pp. 279-320.

[5] Michael Buro, “LOGISTELLO --- A Strong Learning Othello Program”, NEC Research Institute, Princeton, NJ. http://www.cs.ualberta.ca/~mburo/ps/log-overview.pdf

[6] Mark G. Brockington, Jonathan Schaeffer, “APHID: Asynchronous Parallel Game-Tree Search”, Department of Computing Science, University of Alberta, Edmonton, Alberta T6G 2H1, Canada, February 1999

[7] Richard A. Delorme, a program to solve Othello endgame script http://perso.club-internet.fr/abulmo/radoth/

[8] Minimax algorithm, From Wikipedia, the free encyclopedia. http://en2.wikipedia.org/wiki/Minimax+algorithm

[9] Introduction to AI Programming, CSC 243, Computer Based Learning Unit, University of Leeds, 1995-1996 http://www.comp.lancs.ac.uk/computing/research/aai-aied/people/paulb/old243prolog/subsection3_6_4.html#SECTION00064000000000000000

[10] Principal Variation Search --- An enhancement to alpha-beta, Computer Chess, Programming Topics, Bruce Moreland, 2001 http://www.seanet.com/~brucemo/topics/pvs.htm

[11] Xilinx Virtex-II Data Sheet, http://direct.xilinx.com/bvdocs/publications/ds031.pdf

[12] IBM Deep Blue website, http://www.research.ibm.com/deepblue/meet/html/d.3.1.html)

[13] http://home.nc.rr.com/Othello/rules/

[14] P.H.W. Leong, M.P. Leong, O.Y.H. Cheung, T. Tung, C.M. Kwok, M.Y. Wong and K.H. Lee, ``Pilchard - A Reconfigurable Computing Platform with Memory Slot Interface," Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM), California USA, 2001

[15] M. Buro, The Othello match of the year: Takeshi Murakami vs Logistello, ICCA Journal, 20(3): pp. 189-193, 1997.

[16] M. Buro, Experiments with Multi-ProbCut and a New High-Quality Evaluation Function for Othello, Games in AI Research, H.J. van den Herik, H. Iida (ed.), ISBN: 90-621-6416-1, 2000

[17] G. Andersson’s endgame solver, http://www.nada.kth.se/~gunnar/endgame.c