CMU-CS-86-103

Two Designs of Functional Units for VLSI Based Chess Machines

Feng-hsiung Hsu
Department of Computer Science
Carnegie-Mellon University
Pittsburgh, Pennsylvania 15213

Abstract

Brute force chess automata searching 8 plies (4 full moves) or deeper have been dominating the computer chess scene in recent years and have reached master level performance. One interesting question is whether 3 or 4 additional plies coupled with an improved evaluation scheme will bring forth world championship level performance. Assuming an optimistic branching ratio of 5, speedup of at least one hundredfold over the best current chess automaton would be necessary to reach the 11 to 12 plies per move range.

One way to obtain such speedup is to improve the gate utilization and then parallelize the search process. In this paper, two new designs of functional units with higher gate efficiency than previous designs in the literature will be presented. The first design is for move generation only, and is essentially a refinement of the move generator used in the Belle chess automaton, the first certified computer chess master. The second design is a general scheme that can be used for evaluating a class of chess-specific functions, besides generating moves. A move generator based on the second design will be described. Applications of the same general scheme to board evaluation will be briefly discussed.

The research was supported in part by the Defense Advanced Research Projects Agency (DoD), ARPA Order No. 3597, monitored by the Air Force Avionics Laboratory under Contract F33615-81-K-1539.

Table of Contents

List of Figures

1. Introduction

The computer chess scene has been dominated by brute force chess automata—machines that perform full-width α-β search—in recent years. Notable examples include Belle, a special purpose machine, and two programs running on Cray supercomputers, Cray Blitz and NuChess. Both Belle and Cray Blitz search 8 plies (half-moves) or more for typical moves. Belle is the first certified computer chess Master, and Cray Blitz is the reigning world computer chess champion and was until recently, the North American computer chess champion[1]. Cray Blitz was dethroned by Hitech, a new special purpose chess machine from Carnegie-Mellon University, in the 1985 ACM North American Computer Chess Championship tournament. Hitech normally searches 8 to 9 plies in midgame positions and is currently the highest rated chess computer.

The success of these brute force chess programs is mostly based on the increase of playing strength with the search depth. Experimental results [7, 16] showed that an increase of between 200-250 rating points is observed for each additional ply. It is an open question whether the linear increase can still be maintained beyond the current 8-ply search depth of the leading programs. If the linear increase could be maintained, then three additional plies or an increase of 600 rating points would put the computer in the league of world class chess players. The general opinion at the moment seems to be that the increase would start tapering off. On the other hand, it might well be the case that we will see a sudden jump in computer performance against humans when a certain ply depth is reached.

Top human players can now defeat the best chess automata mainly because the human players are capable of long, deep calculations and have better "positional" senses. As machines search deeper, it would become less likely for a human player to find long, thin lines beyond the machines' search horizon. Former world correspondence chess champion Dr. Hans Berliner further observed in his examination of brute force intelligence [2], "It is not at all clear that it is more important to deal with long, thin lines of play [as in human play] better than with bushy lines [as in computer play]; rather the reverse is probably true." The positional aspect of chess is harder though. Deeper searches would certainly improve the machines' positional sense, but an accompanying improvement in the evaluation function might be necessary as well.

To get into the 11 to 12 plies per move range requires speedup of at least a hundredfold over the best current chess automaton. Such speedup can be obtained in two ways: pushing circuit technology or parallelizing the search process. Exotic circuit technologies now still in the laboratories do hold promise of providing speedup of 100 to maybe even 1000, but they are more or less a decade off from "real" applications. A more immediate approach is to improve the gate utilization while staying with existing technologies *and*

[1] Belle searches about 160,000 positions per second, and Cray Blitz searches about 100,000 positions per second in four-processor configuration. The discrepancy in search capability is largely compensated by the better utilization of the transposition table and the more efficient quiescent search used in the case of Cray Blitz.

then parallize the search process. The gate utilization is important because it is a limiting factor for the maximum speedup achievable, given fixed available resources. This paper will be focused on how to obtain better gate utilization than existing designs while getting a small speedup in the process and not on how to parallelize the search process. But we will briefly review recent progress in parallel tree search algorithms here just to see what may be expected from the existing parallel algorithms. The review is necessarily limited in scope and oriented towards theoretical results as the experimental results available are limited to the case of few processors.

Parallel aspiration searching [1], which divides the alpha-beta window among the processors, is known to be typically bounded to speedup of 5 to 6 even for not well ordered game trees. Larger speedup, therefore, cannot be expected from parallel aspiration searching. *Tree decomposition* of the simplest kind [5], which uses a processor tree with one processor for each move orginating from the root and its descendents, can be shown analytically to gain speedup of only $O(N^{0.5})$ with N processors when the game tree is best-first ordered. Speedup of $O(N)$ can be achieved for worst-first ordered game trees, but for games like chess where good ordering can be obtained, the best-first ordered result is closer to reality. The search efficiency can be improved in this simple scheme by limiting the fanout of the processor tree far below the fanout of the game tree, and using the accumulated alpha-beta window to start the search when a slave processor is ready for processing additional moves. Another variation, *mandatory-work-first* searching which utilizes the fact that certain nodes in the game tree have to be searched, has also been theoretically analyzed [6]. For a game tree of uniform width of 38, which is about the average number of moves in chess midgame positions, and a processor tree of fanout of 2, the mandatory-work-first searching is shown to have speedup between $N^{0.78}$ and $N^{0.82}$ when the game tree is best-first ordered. And when the game tree is worst-first ordered, the speedup for mandatory-work-first searching is between $N^{0.93}$ and $N^{0.96}$. A similar algorithm, *key node method* [9], which uses a queued processor pool model and gains further speedup by "eager" communication of the collected bounds, has been reported to show in simulation average speedup of 13 ($20^{0.86}$) for 20 processors. Another recently introduced algorithm, *principal variation splitting* [10, 11], has obtained speedup of similar order experimentally using a real chess program, although only with up to 4 processors, in which case average speedup of up to 3.27 ($4^{0.85}$) has been obtained.

Are we limited to speedup of say $N^{0.8}$? The author has reasons to believe otherwise, but will not detail them here without hard evidence. Assume we would be satisfied with speedup of $N^{0.8}$, then the question of scalability crops up. That is, are we limited to small N as seems to be indicated with the current experimental results? Again the author believes not, at least not for *fast* processors. The number of processors that can be used with high efficiency depends on the speed of the individual processors. With higher processor speed, more processors can be used *efficiently*. The cause of this paradoxical behavior is the fact that the machine is

expected to make its move in *finite* time. Assume the processors are organized as a processor tree. To gain high search efficiency, the processor tree will have to grow much slower than the game tree. Assume the processor tree is of fanout f and depth d, and the game tree is well ordered and has uniform degree of g. For a processor that is capable of searching c plies deep when searching alone, the processor tree would be capable of searching about $2f^d \cdot g^{c/2}$ nodes in the given time, but the game tree will have to be at least of depth d, that is the game tree will have at least $2g^{d/2}$ nodes. Equating the two node figures, we know that the maximum processor tree depth, which is also the upper limit for the game tree depth, is $(\frac{c}{1-2\log_g f})$. For efficient parallelization, f has to be much smaller than $\sqrt{g}$, the effective branching factor, and the limit does exist. Assuming that $f=2$, $g=38$, and $c=4$, which are typical figures for chess programs used in parallelization experiments, we have a maximum search depth of 6.46 plies. However, this search depth is obtained when the processor tree is as deep as the game tree; that is, the terminal processor is evaluating only one single node between message passing. Realistically, to keep the communication overhead under control, the searched game tree normally has to be deeper than the processor tree. Also, we have been assuming perfect processor efficiency, which is also unrealistic. Assume the game tree is k plies deeper than the processor tree and the speedup is N^e, given N processors. Following similar reasoning, we know the maximum game tree depth becomes $(k+\frac{c-k}{1-2e\log_g f})$. For a best-first ordered game tree, a large fraction of the searched nodes one ply above the the terminal nodes will have only one branch that needs to be searched. The parameter k thus probably should be at least 2. Assume $k=2$. Even if we have $e=1$, the maximum game tree depth is still only about 5.23, or maximum *quality* speedup of only about 9.4 (or $38^{(5.23-4)/2}$), and maybe less. Increasing the processor tree fanout will lift this limit, but at the cost of reduced efficiency. The improvement is not too promising though—for $f=3$ and $e=0.8$, which is probably optimistic for a processor tree fanout of 3, the maximum search depth is still only about 5.85. We can thus expect any attempt to try to gain large speedup *efficiently* with *slow* processors unsuccessful, at least under the processor tree model. The experimental results in the literature, which are obtained with "slow" processors[2], seem to bear this simple fact out: the speedup tends to start leveling off at a small number, which can sometimes be as low as 4. To gain higher speedup, the experimenters either increase the processor fanout or increase the processor tree height near the limit, and the speedup degrades. Moreover, the existence of this *slow processor effect* may not be restricted to the processor tree model. The simulation results with the key node method [9], which uses a processor pool model, show that the speedup improves for the larger processor pool when the searched game tree depth is increased. The increase of the game tree depth is equivalent to an increase of available search time, or the use of faster processors. Of course, the existence of the slow processor effect does not mean that no other factors play a

[2]With the exception of Cray Blitz. But Cray processors are hard to come by. *Worst* case speedup of 3.2 was stated for 4 processors, which compares very favorably with worst case speedup sometimes as low as 2 from other experiments. It would be wrong to regard this as an example of the slow processor effect discussed here though. The particular algorithm that was used can be expected to show lower speedup if not for the shared transposition table used. However, a different form of slow processor effect exists in this case. The deeper search tree resulting from the higher processor speed also increases the probability of transposition, and thus the final speedup.

role in the leveling off of the obtained speedup. But the existence does mean that it would be wrong to extrapolate from the experimental curves and jump at the conclusion that no significant speedup can be obtained efficiently. As a final note, even though slow processor effect plagues existing algorithms, it is possible to reduce the effect at the cost of algorithm complexity.

In this paper, we will be examining the possibility of building single chip high speed chess subfunctions. Equipped with such devices, one will be able to explore and experiment with ways of parallelizing the search process with relative ease. The devices are expected to be able to support search speed of 500K - 1M positions per second in single thread configuration, or about 3-10 times the current generation of chess machines. Assuming that further speedup in the range of 100-1000 is obtained by parallelization, a machine that searches 12-14 plies does not seem too far off.

We will first briefly review previous designs for chess machines. A single chip parallel move generator that uses basically the same move ordering as the Belle move generator will then be presented. Circuit refinements that reduce both the device count and the wire count and also increase the layout regularity are applied to the original TTL Belle move generator to derive the single chip move generator. Some minor deficiencies in the original Belle move generator are also fixed. A pipelined version of the move generator will then be shown. The pipelined move generator is smaller than the parallel version but is also slower. Applications of the same pipelined scheme to various chess evaluation functions will then be described.

2. Previous Designs

In the later half of the 70's, two different groups started building specialized machines for chess using off-the-shelf TTL components. The first group [12] at MIT implemented a simple move generator together with a general purpose microprogrammable processor. No hardware evaluation was provided, and the ordering of the moves generated was controlled by the microprogram with some hardware assistance. The other group at Bell Labs went through three iterations which culminated in the now famous Belle automaton [3] in the early 80's. The Belle chess machine has both a hardware move generator and a hardware evaluation function built in. The hardware evaluator is further divided into an incremental evaluator and a 'slow' evaluator. Belle consists of ten large wire wrap boards, with four each for the move generator and the 'slow' evaluation function. The Belle move generator can generate one move per microsecond. The 'slow' evaluation function evaluates the ray control and the pawn structure and takes two microseconds per evaluation. However, Belle only searches about 160,000 positions per second because the controller and the hash table, which were built with late 70's components, slow down the entire system. We will examine the Belle move generator in some more detail in the next section.

VLSI technology has been applied to the design of brute force chess machines with varying degrees of

success. A single-chip move generator [13] was claimed to be able to generate 360,000 moves/sec, but the designers failed to point out that the moves generated are not ordered by the relative merit and that the move generation status is not maintained nor is a masking mechanism provided. In a real system all the legal moves at one position would have to be generated and then sorted by an off-chip processor before the next position could be examined. A more realistic figure would be no more than 60,000 moves/sec, assuming zero overhead for sorting and retrieval of the moves.[3] No fabricated chips were available at the time of publication for this design, and no details were given on how the move sorting and retrieval would be done.

Another move generator design [4] that was done at Carnegie-Mellon University and is now used in the Hitech chess machine took a different approach. Instead of simplifying the circuit and throwing away functionality, a 64-chip, one chip for each square, design is used and a fairly complicated ordering scheme is employed[4]. A maximum speed of 500,000 moves/sec is reached. The maximum speed of this 64-chip move generator may seem to be inferior to the speed of Belle's move generator; however, as a compensation, the move generator has a theoretically better move ordering scheme than Belle and has special setup for generating check evasion moves, both of which should have positive effects on search efficiency. When the evaluation function is slow and dominates the search time, a machine based on the Hitech move generator will be able to search deeper than a machine based on the Belle design. In fact, Hitech is probably the speed champion in mid-game at the moment. It searches about 175,000 positions per second and probably about half a ply deeper than Belle on the average. On the other hand, the 64-chip design is also wasteful in terms of silicon real estate—an enormous amount of redundant circuitry has to be added to reduce the communication cost. And it is not clear that the better move ordering would be adequate compensation for lower speed when evaluation can also be done fast. One might want to argue that the better move ordering would provide exponential savings in deep tree search. In reality, this argument does not completely hold when the transposition table is used to provide additional move ordering information, which is normally the case for the top programs, including Hitech itself. The transposition table in this case provides far better move ordering information than the move generator can reasonably supply, at least for the levels of the game tree near the root. Mechanisms such as killer tables also provide additional ordering information independent of the move generator. And the savings obtained from the better move generator move ordering mainly exist in the last one or two plies and are probably only about 10-20 per cent or less[5]. However, for comparison with Belle, the

---

[3] Assuming an average of 36 moves for typical midgame positions, we get 10,000 positions/sec. But assume at each position 6 moves are actually made on the average, the real figure becomes 60,000 moves/sec.

[4] The actual system uses more than 64 chips. Some additional chips are used for special move generation and a few SSI/MSI chips are used to interface the chips.

[5] Carl Ebeling reviewed these figures and agreed that they were probably correct. Real confirmation of these figures will have to wait until Carl completes his simulation of the various ordering schemes though.

argument partially holds. Based on engineering decisions made back in the late 70s when memory was still an expensive resource, the Belle chess automaton does not store the move ordering information in the transposition table. While it had been speculated that the Belle move generator cannot test move legality, Ken Thompson explained in private communication that indirect move legality testing can be done, and for testing moves from the transposition table, indirect testing would be sufficient. Also since Belle does keep track of the move ordering *explicitly* for the top levels of the search tree, the exponential saving is still only realized for the lower levels of the tree. The unavailability of full move legality check is still a minor problem for the Belle move generator though, as full move legality check would be desirable if killer tables are to be used, and will be resolved as part of the proposed circuit refinement in the VLSI move generators presented in this paper.

3. A VLSI Parallel Chess Move Generator

A description of the Belle move generator [3] and the related move ordering scheme will first be given and then circuit refinements that both reduce the circuit size and give a slight improvement in functionality will be presented.

3.1. The Belle Move Generator

The Belle move generator is composed of an 8×8 array of similar combinatorial circuits and a 6×64-input priority network. Two operations are provided, *FIND-VICTIM* and *FIND-AGGRESSOR*. Each of the 64 combinatorial circuits consists of a transmitter, a receiver, a four-bit register holding the piece and a 1-bit wide 64-bit deep disable stack. Each transmitter sends signals to its (chess) neighbors, and each receiver accepts signals from its neighbors. Figure 3-1 and Figure 3-2 show block diagrams of the transmitter and the receiver circuit based on the drawings and written description in the original article [3].

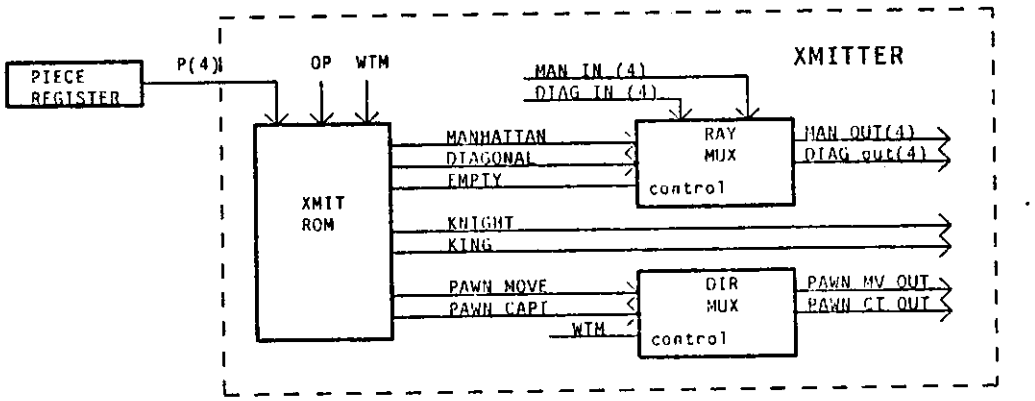

Figure 3-1: The Transmitter Circuit Used in Belle Move Generator

"The transmitter generates attacks radiating from a square. The main component of the transmitter is a read-only memory (ROM). The ROM generates a set of wires active if the square contains the corresponding piece. The global signal WHITE-TO-MOVE [WTM] is true if White is on the move." In the *FIND-AGGRESSOR* cycle, only the addressed victim transmitter is allowed to radiate attack signals and only as a super-piece that is the union of all non-pawn pieces. Also, a pawn move signal is radiated from the victim square if the square is occupied.

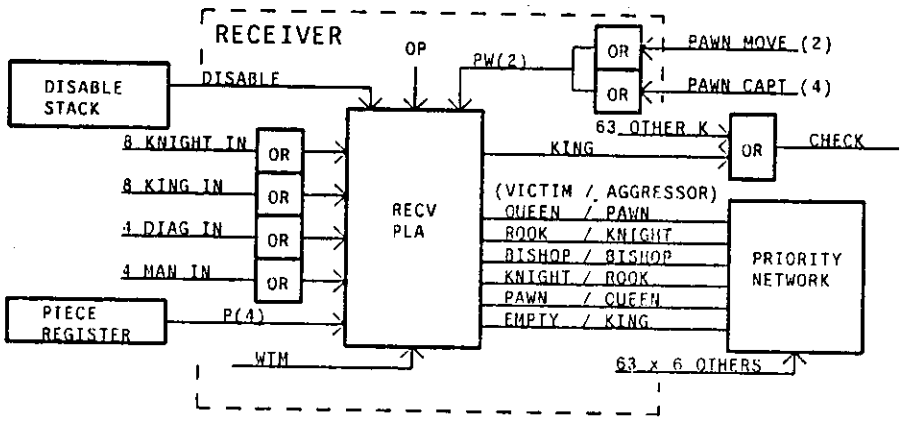

Figure 3-2: The Receiver Circuit Used in Belle Move Generator

"The receiver section analyzes the attacks generated by the transmitters. The main component of the receiver is a programmed logic array (PLA)." "The PLA accepts the concentrated attack signals, the resident piece, WHITE-TO-MOVE, the disable bit and the op-code. It generates seven priority signals corresponding to the value of the attacked piece [in *FIND-VICTIM* cycles]." In *FIND-AGGRESSOR* cycles, the priority signals are in reverse order of the piece value and the priority signals are generated only if there is an attack signal corresponding to the piece on the square.

The moves are generated in the following way. The micro-code asserts *FIND-VICTIM*. The priority network then finds the highest *enabled* attacked piece. The address of the attacked piece is latched and used as the victim square in the following *FIND-AGGRESSOR* cycle in which the lowest valued *enabled* aggressor on the selected victim is found. After the selected move is processed, the last aggressor square is disabled and the next lowest aggressor is found until there are no more aggressors on this victim. The victim square is then disabled and all the aggressor squares enabled. Then the next victim is processed in the same way until all the victims are exhausted. All the victim squares and all the aggressor squares are re-enabled for the next new

position at the same search depth, and the whole process starts all over again. The disable stack is used to keep track of the move generation status at each search depth. The moves generated this way are only *pseudo-legal moves* as the king of the moving side might be checked if the moves are made. Also, the special chess moves have been deliberately left out in our presentation here as they only complicate our discussion without serving useful purpose.

3.2. From TTL to VLSI

Now we will examine the problems associated with directly implementing the Belle move generator in VLSI. Simple calculation shows that the Belle move generator as it is would be difficult to fit into one single chip in today's commercial process. Both the number of wires and the number of devices are too high. The largest bundle of wires is used by the priority network: 384 wires. For the 3 μm MOSIS CMOS process with 7 μm metal pitch, 384 parallel metal wires alone would be almost 3 mm high. And beside these wires, each square has no fewer than 50 wires to its neighbors. In terms of device count, it fares no better. If each stack cell contains 8 transistors, the 64×64 disable stack alone will need 32K transistors. And the transmitter, the receiver and the priority network have not been counted yet. The 64-bit disable stack at each square also further aggravates the wiring problem because most of the wires will have to be routed around the stack[6].

From the above discussion, it is clear that in order to fit a Belle-like move generator into a single die, two modifications will have to be made: reducing the number of wires used by the priority network and reducing the number of devices used by the disable stack. It is also desirable to have a fast, regular priority encoding mechanism.

We will first look at the problem of the disable stack. Is the disable stack really needed? The answer surprisingly is no. With the move ordering scheme used in Belle, the move stack that is normally maintained by the controller actually contains enough information about the move generation status. After an unmove cycle backdating the board position, we know that the victim to be tested is the same as the victim in the move just unmoved and all the aggressors that have lower value than the last aggressor or have the same value as the last aggressor but a lower or equal square priority should be ignored. And if a new victim is needed, the new victim will have to have lower value than the original victim or have the same value and a lower square priority. In other words, the disable stack is redundant. If a proper masking mechanism is provided to generate masks from the move stack, the entire disable stack can be removed. It will be shown how such a masking mechanism can be provided along with a reduction in the size of the priority network. As a sidenote, providing the disable stack instead of a masking mechanism might well have been easier for the Belle TTL

[6] It is possible to move the stack out of the basic cell at the cost of more wires going into the cell, but the proposed complete elimination of the stack in this paper is definitely preferable.

implementation.

The priority network can be viewed as a huge priority encoder with 384 inputs or as an arbiter with 64 competing elements. The Belle move generator takes the priority encoder view and uses up lots of long wires for the priority network alone. However, if we view the priority network as an arbiter, a more wire-efficient scheme can be used if we adopt a distributed arbiter approach. In a distributed arbiter, individual priority lines are replaced with a wired-or arbitration bus. Each competing element monitors the arbitration bus and dynamically removes lower order bits from competing whenever a high order bit loses out. Instead of using N wires, $O(\log_2 N)$ wires are sufficient. Distributed arbiters of this kind have become quite popular in multiprocessor bus design lately. The earliest description of this arbitration scheme was in a 1966 UK patent; Matthew Taub of IBM rediscovered the scheme in 1975 and Leo Paffrath at SLAC rediscovered it yet again for the Fastbus project [8, 15]. A MOS implementation of one competing element of such a distributed arbiter is given in Figure 3-3. The pullup transistors for the wired-or bus lines are not shown in the figure. In this particular circuit, at the end of the arbitration process, the priority of the competing element with the highest priority will appear on the bus in negative logic form. Assume each row of cells use a 7-bit bus, 1 bit indicating the existence of competitors, 3 bits for piece priority and 3 bits for square priority, a total of 56 wires for the eight chess rows would be needed versus the 384 wires for the priority network design.

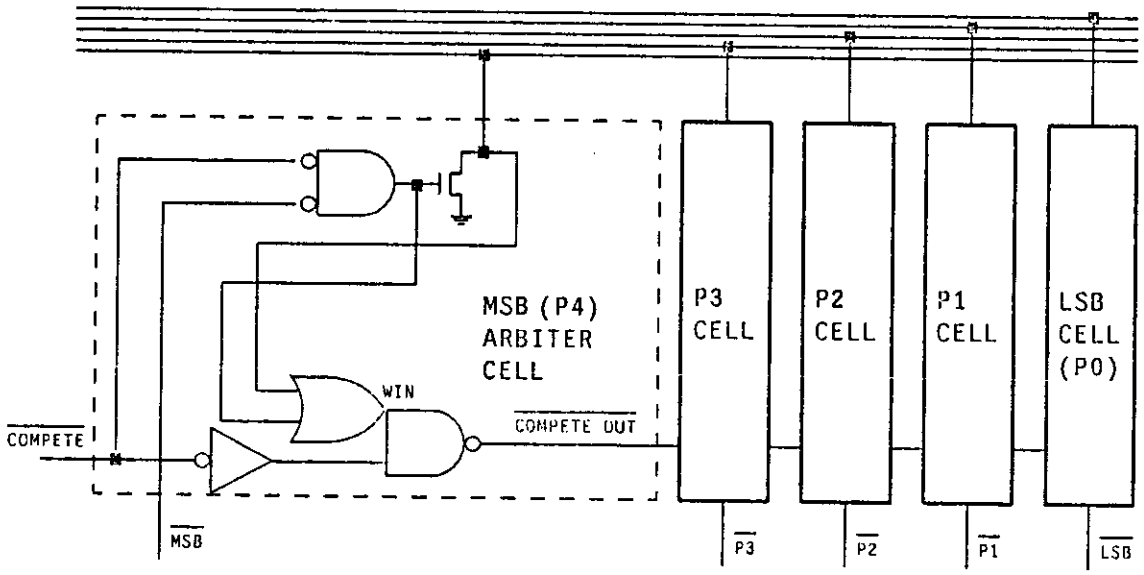

Figure 3-3: A Competing Element of a Distributed Arbiter

Introducing a masking mechanism into the distributed arbiter is relatively straightforward if certain assumptions are made about the relative square priority. Assume a row-major square priority scheme is used.

A mask bus, which can double as the piece register loading bus, broadcasts the type of the highest priority piece now allowed to compete. All squares with higher piece priority inhibit themselves from the competition. And all squares with lower piece priority enter the competition. Additional masking signals as shown in Figure 3-4 are used to decide whether to mask off a square with the broadcasted piece priority. Of these signals, the row-enable signals are set if and only if the row priority is lower than the last square tried; the row-select signals are set only for the row the last square was in; the column-select signals are set for all the columns with lower column priority than the last tried square.

Squares with the piece priority broadcasted by the mask bus are then allowed to compete depending on their row-select, column-select and row-enable signals:

1. If the row-enable signal is set, then the square competes.

2. If both the row-select and column-select signals are set, then the square competes.

3. Otherwise, the square is disallowed to compete.

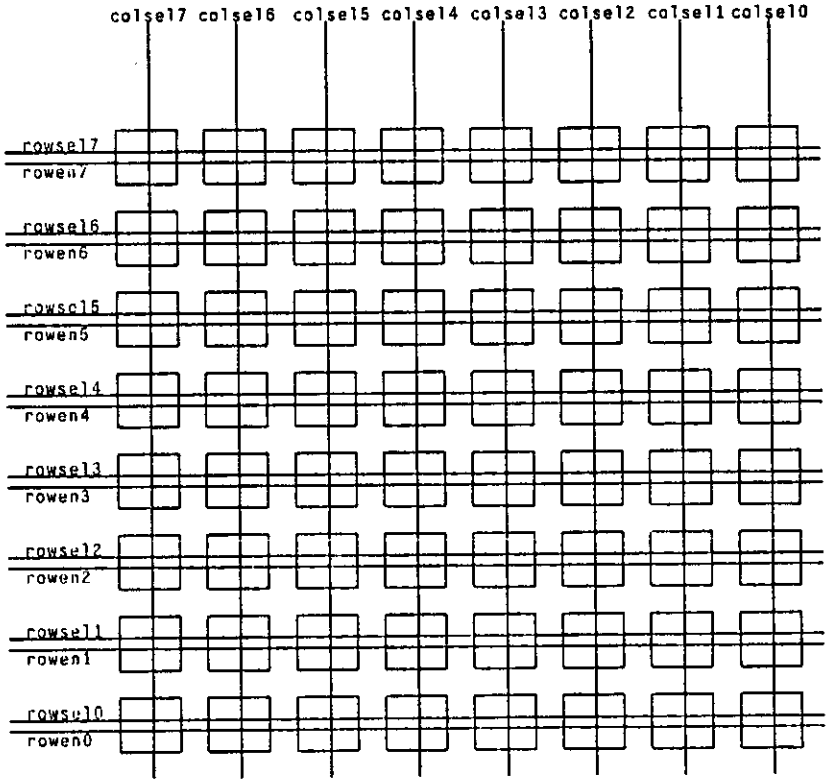

Figure 3-4: The masking signals for squares with the broadcasted piece priority

The elimination of the disable stack and the integration of the masking mechanism into the arbitration

network not only free up the area used by the disable stack, but also simplify the receiver design. The receiver no longer needs to be disabled: the masking mechanism takes care of that. While both the original Belle transmitter and receiver circuits are suboptimal for compact VLSI implementation, we will not go into detail on how to improve them. The built-in masking mechanism also offers a more flexible way of ordering moves than the Belle move generator. For one thing, check evasion moves can be generated with relative ease. And the minor problem of full move legality testing can be solved by using the next higher priority move as the mask and testing whether the generated move is the same as the given move.

All in all, it is estimated that when all the circuit refinements are applied to the Belle move generator, the chip area is probably reduced by more than a factor of four over direct implementation. A CMOS version of such a single chip move generator has been designed and simulated with a switch simulator on a workstation with a simulation accelerator. Spice simulation on circuits extracted from preliminary layout indicates about 3-4 ns worst case ray ripple delay per chess square and about 6-8 ns per arbitration bit. The cycle time is about 100-150 ns, or about 400-600 ns per move[7]. The chip has about 36K transistor sites. The chip fits into a 40-pin dual-in-line package and has a die size of 6912×6812 microns, the MOSIS standard die size.

4. A Pipelined Chess Move Generator

Suppose because of yield considerations and/or power consumption constraint we want to further reduce the chip area over the parallel version of the move generator by serializing part of the operations.

In this section, we will see a pipelined move generator derived from the parallel move generator in the last section. The pipelined version is slower than the parallel version but is also smaller. While there is no plan of actually implementing such a chip, this design seems to be appropriate even for MOSIS 4-micron NMOS process, which was used to implement the Hitech 64-chip move generator, because of the reduced static power consumption. The real reason why this pipelined move generator is presented here is to introduce a general pipelined scheme that can also be used to implement evaluation functions.

4.1. A Semi-Systolic Chess Algorithm

One possibility for serializing the move generation process is to examine one move direction at a time, i. e. multiplexing say the ray signal lines, in order to save the number of wires needed. This approach has been used in the Belle ray evaluator: all the rays in one direction are examined at a time and after 8 evauation cycles, the ray control evaluation is taken from the ray evaluation accumulator [3]. While this approach is fine with ray evaluation as the number of adders is also reduced, it is not clear for a move generator how additional move patterns can be accounted for in this manner without further increasing the number of cycles.

---

[7] The piece registers are dual ported and normal moves can be updated or backdated in one single cycle.

And also the area reduction achievable for the move generator is not sufficient.

Another approach which has been tried in one of the earlier VLSI move generators [13] is to use only 8 *square machines* and to pipe the board representation in various patterns through the square machines to generate moves in one move direction at a time. While the area reduction reached is fine, the design has two serious drawbacks: not enough functionality is designed in and the move generation speed leaves much to be desired. Around 300 cycles are needed to generate moves for one position and the moves generated are not ordered according to relative merits. If the average branching ratio of the α-β search is 6 and a separate chip is available to handle the move sorting and retrieval with no overhead, a 2-chip move generation system based on this design will need on the average 50 cycles to generate one move. That is, on a per-chip basis, this design leads to a 100 cycles per move per chip figure.

Now we examine how we can pipeline the operations more efficiently. In particular, we will be looking at how we can borrow ideas from systolic arrays to obtain a better design[8].

Assume we have a linear array of systolic cells communicating with the nearest neighbors and receiving input data streams from a large register file containing the piece information. Conceptually, we have a hardware configuration similar to the one shown in Figure 4-1. The registers file sends piece information to the systolic array cells and the array cells send and receive signals derived from the piece information to/from its closest neighbors.

First we examine what kinds of input data streams for the systolic array are needed. The simplest possible input streams are columns (or rows, we will use columns throughout the paper) of the board, one column at a time, from one side to the other. Given such a stream sweeping from the right hand side of the board to the left hand side, i. e. KR, KN, KB, K, Q, QB, QN and then QR file, what can we extract from the stream using the linear systolic array? For one thing, all the horizontal rays issuing from right to left and all the diagonal rays that are going from right to left can be tested in 8 cycles. If two streams, one from left to right and the other from right to left, are provided by either dual-porting the register file or by replicating the register file, then all the rays except the vertical ones can be tested in a systolic fashion in 8 cycles. Alternatively, we can time-multiplex a single stream by changing the streaming direction at the end of the first 8 cycles to get a 16-cycle design. For the moment, we will assume a two-stream design. Do we have enough information in these two data input streams to extract the vertical rays and other types of moves? The answer is yes, but non-systolically.

---

[8] Curiously enough, the design just described in the last paragraph was labeled "systolic".

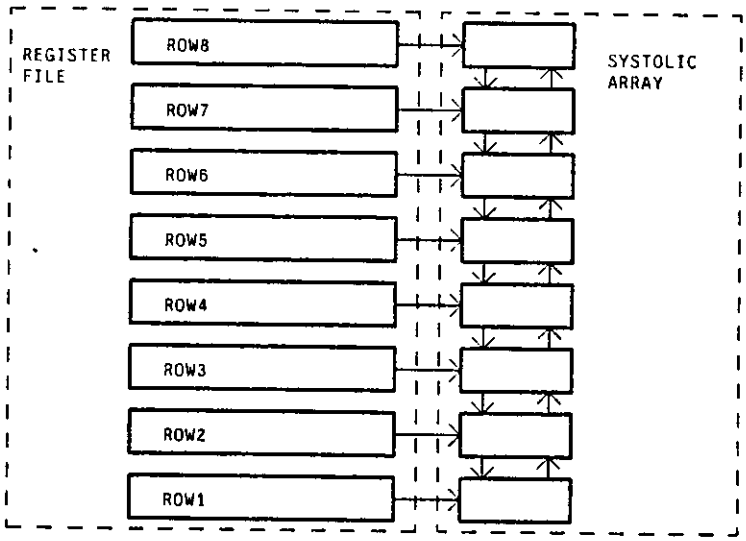

Figure 4-1: A Systolic Chess Machine

Given the piece data of a column, we have sufficient information about the vertical rays in the particular column. The vertical rays in this case can be extracted in a parallel manner as in the ray circuits used in the parallel move generator. The pawn moves and king moves in the vertical directions can also be handled in the same way as in the parallel move generator—simply sending the derived signals to the vertical neighbors. Pawn captures, king moves in non-vertical directions, and knight moves can be handled by delaying the corresponding signal by one or two cycles and then sending out a signal to the square itself, the closest neighbors, and in the case of knights, the second closest neighbors. Figure 4-2 shows the basic design of the pipelined cell that results.

4.2. The Pipelined Version of the Parallel Move Generator

The pipelined cell just described is actually two half-cells. Each half-cell examines one input stream and processes half of the possible moves. Each half-cell implements half of the transmitter semi-systolically and provides a half receiver that is time-multiplexed over one chess row. At every move generation cycle, half of the possible moves from two columns are examined. The pairings of the columns in the 8 systolic cycles are (1,8), (2,7), (3,6), (4,5), (5,4), (6,3), (7,2), and (8,1) respectively. Both the *FIND-VICTIM* and *FIND-AGGRESSOR* operation takes 8 cycles each.

The resolving of the priority is handled by one column-pair at a time. A column-pair priority is provided for the entire linear systolic array for comparison with the current highest priority piece maintained over the 8 systolic cycles. A masking mechanism similar to the one used by the parallel move generator but with a column-pair major ordering can be used. The priority resolving phase can be overlapped with the attack

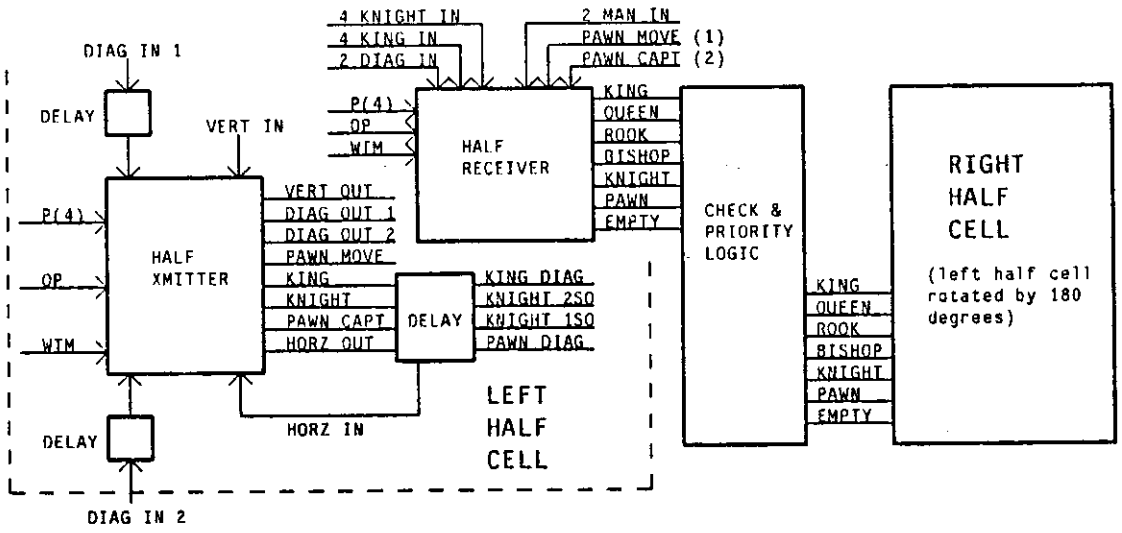

Figure 4-2: The Basic Pipelined Cell

signal generation phase by providing latches before the priority circuit. Design data from the parallel move generator indicate that the two phases take about the same time, which means the cycle time should be around 50-75 ns when the pipeline stage before the priority circuit is introduced. However, we would now need 9 cycles each for the two FIND operations. The pipelined design would thus need about 900 to 1350 ns to generate one move, excluding the updating and backdating of the board. The speed is about 4 to 5 times slower than the parallel version (200 to 300 ns, excluding move updating and backdating), but the area is also reduced by a factor of 4 to 5. Figure 4-3 shows a higher level view of the array section of the pipelined move generator.

An interesting comparison can be made with the 64-chip Hitech move generator. The Hitech move generator partitions the circuit *spatially* and the pipelined scheme partitions the circuit *temporally*. The pipelined design needs to add some additional latches to keep track of the temporal information, and the Hitech design needs to add a set of additional registers for *all* the bearing pieces on the square and the complete move logic for *all* the possible moves to the square in order to keep track of the spatial information. In this particular case, keeping track of temporal information turns out to be much cheaper than keeping track of spatial information. Spatial partitioning scheme should generally be faster than temporal partitioning scheme; this is not true in this case, because the spatial partitioning scheme needs inter-chip communications to resolve the move priority, while the temporal partitioning scheme can be further speeded up by overlapping operations with the introduction of additional pipelined stages[9]. The fundamental cause that the spatial partitioning does not win here is probably the sequential nature of the move generation process which

[9]Yet another reason is that we are comparing a hypothetical design with a real one?!

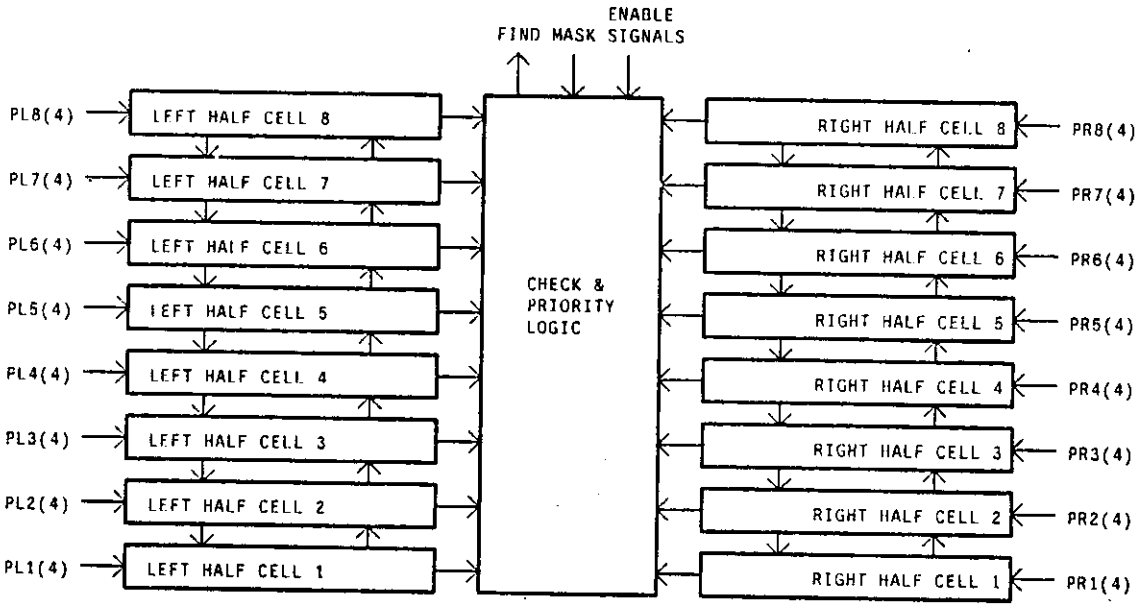

Figure 4-3: The Array Section of the pipelined Move Generator

favors pipelining versus outright parallelization. The Hitech move generator does provide better move ordering, but as we shall see in the next section, if we are willing to double the hardware size, there is no real reason why we cannot obtain similar ordering.

5. Pipelined Evaluation Subfunctions

While there are quite a few features such as material, piece placement, and some pawn structures that are easier to evaluate in an incremental way, certain more subtle features simply require a whole board approach to do the evaluation. The pipelined scheme described in the last section provides one such approach.

Let us review the pipelined scheme and examine it in more detail. The pipelined scheme can be viewed as consisting of two main parts: feature detector, which includes the transmitters and receivers, and feature compressor, which is the check and priority logic circuit in the case of the pipelined move generator.

For the pipelined move generator described in the last section, the feature detector only generates one player's attack signals at a time. The move ordering obtained is roughly equivalent to what one obtains by making a one-ply lookahead to see whether any material can be gained. This need not be the case. More general attack signals could be generated at the cost of more hardware. By providing attack signals from both sides and doubling the hardware size, we can get a move ordering that is roughly equivalent to what one obtains by making a two-ply lookahead to see whether any material can be gained *safely*, or the Hitech move

ordering. We are also not limited to *real* attack signals only either. Pseudo attack signals can be used to detect pins, for instance, which are "deeper" concepts. More description on how pins can be detected will be given shortly. Some pawn structures can also be detected in similar manners, but we will not go into how to implement them.

The most general feature compressor one can get is simply a lookup table, which is of course impractical because of the enormous table size. In the pipelined move generator, we have primarily a FIND-MAXIMUM circuit. Other useful variations include summation circuits, or combinations of summation circuits and small lookup tables, or maybe even some weighted summation circuits with variable weights.

We will describe two evaluation subfunctions that can be implemented with the pipelined scheme: square control and pins.

5.1. Square control

Square control is a very important evaluation subfunction in chess. When the board position is quiet with no immediate tactical gain, the square control term gives the program a sense of strategic direction. Square control is also frequently interrelated with pawn structure and king safety, but we won't go into the details on how to handle the interrelation.

The square control term that is going to be presented here is taken from the program Chess 4.5, which was among the series of the Chess X.X programs that prevailed over other 70s' chess programs and predated the current NuChess program. The square control term used in chess 4.5 [14] is actually a mobility measure instead of real square control, but we will use it as an example here because of its simplicity in concept.

The square control function in Chess 4.5 assigns weights to squares attacked or defended by each ray piece and returns the sum of these weights. Bishops are assumed to exert *no* control over a square that has a friendly pawn and queens are assumed to exert *no* control over a square that is attacked by enemy pawns. Bishops are given a relative bonus of 3 for each square controlled, rooks a bonus of 2, and queens a bonus of 1. The figures given here are relative terms; the actual figures in Chess 4.5 are the above numbers multiplied by a floating point constant.

Let's first look at what attack signals need to be generated. Since the Chess 4.5 square control concerns only the ray pieces and pawns, both the king attack signals and knight attack signals need not be generated. However, the attack signals for ray pieces and pawns from *both* sides need to be generated. Also, the ray attack signals now need to keep track of the types of the ray pieces attacking.

The receiver section now will have to contain logic to discount the bishop control signal when the standing piece is a friendly pawn and also to discount the queen control signal when the square is attacked by enemy pawns. And instead of generating priority signals, a parallel addition over all the square control signals has to be provided and the sum is accumulated and then output.

The priority bus used in the move generator is replaced here with an 8-input parallel adder to sum up the values from the 8 pipelined cells. And an accumulator substitutes the logic used to maintain the highest priority square in the move generator.

The summation-of-eight operation should roughly take about the same time as the priority circuit in the pipelined move generator. Assume the same cycle time of 50-75 ns and 9-cycle evaluation time, with one cycle for the extra pipeline stage, each evaluation should take only about 450-675 ns.

5.2. Pins

A pin involves three chess pieces: the pinner, the pinnee and the pinned object. The three pieces have to be in the same line with the pinnee in between. This relation is shown in Figure 5-1.

Figure 5-1: A Typical Pin

The pinner is a ray piece that is capable of attacking in the line connecting the three pieces. The pinnee and the pinned object are pieces of the opponent side of the pinner. If we make the assumption that the pinned object is always guarded, a pin occurs whenever the value of the pinned object is higher than the pinner. For a pinning queen, the pinned object has to be the king. For a pinning rook, the pinnned object can be either a king or a queen. And for a pinning bishop, the pinned object can be a king, a queen, or a rook. We will assume the value of the pin term is the sum of the signed value of all the valid pinnees for both players.

Pins are only short term tactical chess features under most circumstances. For searches that extend beyond 8 plies, the efficacy of detecting pins has not been fully established. The reason why we examine pin evaluation here is to illustrate how one can evaluate in a pipelined fashion a chess function that is commonly regarded as complicated to compute. Also, we want to have an example that would need to both read from and write to the register file during the computation cycles. As the reader may have noticed, both the aforementioned pipelined move generator and pipelined square control evaluator only read from the register

file during the computation.

Now we examine the hardware.

In the transmitter, 4 bits need to be passed along each ray direction in order to encode the possible piece types: white king, white bishop, white rook, white queen, blocked, black king, black bishop, black rook, and black queen. Thus 16 bits are generated from each transmitter part of the two half cells. Logic for pawn moves, knight moves and king moves is no longer needed.

Before we go into the receiver section, let us examine what happens in the 8 computation cycles. In the first 4 cycles, 4 ray directions are tested for every square. Since we cannot discard this information until we gather the ray status for the other 4 directions, the information has to be stored in the register file during the first 4 cycles. Since 16 bits are used to encode the information for the first 4 rays, the register file is now 20-bit wide instead of 4-bit wide. Also in the first 4 cycles, the newly added 16 bits are write-only; and in the second 4 cycles, the new 16 bits are read-only.

The receiver section performs the store-to-register operation during the first 4 cycles. During the second 4 cycles, the receiver reads the stored ray status and compares the stored rays with the correpsonding opposing rays just computed. And if the pin condition is satisfied for any of the rays, the value of the piece standing on the square is outputed.

The final parallel addition part and accumulation part is the same as in the square control evaluator.

6. Status and Concluding Remarks

In this paper, a new VLSI parallel chess move generator and the pipelined version thereof have been presented. A systolic algorithm used to transform the parallel move generator into the pipelined version is also described. Applications of the algorithm to chess evaluation subfunctions are also shown.

While the pipelined algorithm is presented here as a means to make VLSI implementation feasible, it should also be applicable to board level designs. Such board designs could serve as proofs of concept or, more importantly, as ways to resolve the evaluation function before committing to silicon.

The chip designs described here are for the chess-specific part of the hardware. To realize a complete system, it would be necessary to include a controller fast enough to keep up with the rest of the system. Also necessary would be a repetition detector that keeps track of whether there is any repetition of board positions. The design tasks for the customized controller and the repetition detector are relatively straightforward. Both

tasks are among the well-understood problems in VLSI design. Methodology already exists for building customized controllers, and the repetition detector is nothing more than an enhanced content addressable memory.

A small chess chip set that runs at about 500,000 to 1,000,000 postitions per second is believed to be feasible with the MOSIS 3 micron CMOS process. Notice that this is already about 5-10 times the raw speed achieved by Cray Blitz with a 4-processor Cray-XMP and about 3-5 times faster than Hitech. A single thread machine based on such a chip set should be able to search about 9-10 plies on the average. And if additional speedup of the order of 100 is achieved through parallelization of the search process, machines searching 12 plies are indeed within reach.

Design of the VLSI parallel move generator was started in July 1985. The preliminary logic design was completed by the end of August 1985 after about one mon-month of work. Transistor switch level simulation was done on the entire chip shortly after on a workstation equipped with simulation accelerator. Test vectors were generated by C programs written on a Vax running UNIX. The ability to simulate the entire chip has proven extremely helpful. No fewer than ten iterations of design changes were completed before mid-September 1985 when the design was ready for layout. The layout process turned out to be a nightmarish experience. Unavailability of suitable placement and routing tools has been and remains an acute problem. The first layout attempt was done without proper placement and the array cell was of the wrong aspect ratio. Meanwhile, it was discovered that a 10% reduction in transistor count is possible by changing some of the circuits. The change was made and simulated. The subsequent layout took much longer time than originally expected. The cause of the extended layout time was mainly the desire to get a small die size which ruled out the possibility of using cad tools that do not yet provide sufficient circuit density. The chip has been assembled as of January 1986 and is being verified by netlist comparison.

The design of the evaluation function is expected to a tricky problem. The Hitech evaluation function design shows that a little programmability goes a long way. In the earliest Hitech implementation, the evaluation function was essentially just a programmable piece placement lookup table, plus a simple hardware pawn structure evaluator. Coupled with clever chess programming and higher search speed, such an evaluation function turned out to be a match for the full fledged Belle hardwired evaluation function, at least in endgames. In the two recorded games between the two machines, Belle obtained two-pawn advantages in midgames but managed to lose them in endgames and two draws were registered between the two chess machines. Newly added programmable feature recognizers have now catapulted Hitech's rating to low 2300s. Programmability is certainly desirable, but it is also expensive in terms of chip area. RAMs use about 5-20 times more area than ROMs. It is thus also desirable to replace RAMs with ROMs in a custom chess evaluation function chip. Programmability of the piece placement lookup table can be preserved by using an

off-chip commercial static RAM, but programmability as in the Hitech feature recognizers probably has to be given up. The square control evaluation, which is still missing in Hitech, will probably make up for part of the loss of programmability. Programmability may not be good all the time though. Hitech adjusts the evaluation function based on features detected at the root. For a 12-ply searcher, applying knowledge obtained at the root to the terminal nodes may create erroneous evaluation. Of course, the possibility of having some programmable evaluation at intermediate level of the tree still exists, but probably will not be explored for a while.

Acknowledgement

Dr. Hans Berliner has been instrumental in stirring up my interest in Computer Chess. Carl Ebeling, Andy Palay and Gordon Goetsch have provided many insights related to designing a chess machine. The pioneering work of Carl and Andy in VLSI chess machines has been the main inspiration behind this work. While the move generator designs here are derived from the Belle move generator, they are also strongly influenced by the Hitech move generator design. And the idea of systolic chess machines was developed while discussing designs of hardware evaluation functions for the Hitech chess machine. Besides Hans, Murray Campbell has also been a generous source of knowledge about chess and computer chess in particular. Special thanks are also due to Ken Thompson for answering some of the questions related to Belle.

References

[1] Baudet, G.
The Design and Analysis of Algorithms for Asynchronous Multiprocessors.
PhD thesis, Carnegie-Mellon University, 1978.

[2] Berliner, Hans.
An Examination of Brute Force Intelligence.
In *Proceedings of the Seventh International Joint Conference on Artificial Intelligence*, pages 581-587. 1981.

[3] Condon, J. H. and Thompson, K.
Belle Chess Hardware.
Advances in Computer Chess 3.
Pergamon Press, 1982, pages 45-54.

[4] Ebeling, Carl and Palay, Andrew.
The Design and Implementation of a VLSI Chess Move Generator.
In *The 11th Annual International Symposium on Computer Architecture*, pages 74-80. 1984.

[5] Finkel, Raphael A. and Fishburn, John P.
Parallelism in Alpha-Beta Search.
Artificial Intelligence 19(1):89-106, September, 1982.

[6] Finkel, Raphael A. and Fishburn, John P.
Improved Speedup Bounds for Parallel Alpha-Beta Search.
IEEE Transactions on Pattern Analysis and Machine Intelligence PAMI-5(1):89-92, January, 1983.

[7] Gillogly, J. J.
Performance Analysis of the Technology Chess Program.
PhD thesis, Carnegie-Mellon University, 1978.

[8] Gustavson, David B.
Computer Buses--A Tutorial.
IEEE Micro 4(4):7-22, August, 1984.

[9] Lindstrom, Gary.
The Key Node Method: A Highly-Parallel Alpha-Beta Algorithm.
Technical Report UUCS 83-101, Department of Computer Science, University of Utah, Salt Lake City, March, 1983.

[10] Marsland, T. A. and Campbell, Murray.
Parallel Search of Strongly Ordered Game Trees.
Computing Surveys 14:533-551, 1982.

[11] Marsland, T. A. and Popowich, Fred.
Parallel Game-Tree Search.
IEEE Transactions on Pattern Analysis and Machine Intelligence PAMI-7(4):442-452, July, 1985.

[12] Moussouris, J., Holloway, J., and Greenblatt, R.
CHEOPS: A Chess-oriented Processing System.
Machine Intelligence.
Ellis Horwood Limited, 1979, pages 351-360.

[13] Schaeffer, Jonathan, Powell, P. A. D., and Jonkman, Jim.
A VLSI Chess Legal Move Generator.
In Randal Bryant (editor), *Third Caltech Conference on Very Large Scale Integration*, pages 331-350. 1983.
Also appeared in *VLSI Design*, Vol. IV, No. 3, 1983, pp. 64-71.

[14] Slate, David J. and Atkin, Lawrence R.
Chess 4.5--The Northwestern University Chess Program.
Chess Skill in Man and Machine.
Springer-Verlag, 1977, pages 82-118.

[15] Taub, D. M.
Arbitration and Control Acquisition in the Proposed IEEE 896 Futurebus.
IEEE Micro 4(4):28-41, August, 1984.

[16] Thompson, K.
Computer Chess Strength.
Advances in Computer Chess 3.
Pergamon Press, 1982, pages 55-56.