IBM's Deep Blue Chess Grandmaster Chips

THE IBM DEEP BLUE SUPERCOMPUTER THAT DEFEATED WORLD CHESS CHAMPION GARRY KASPAROV IN 1997 EMPLOYED 480 CUSTOM CHESS CHIPS. THIS ARTICLE DESCRIBES THE DESIGN PHILOSOPHY, GENERAL ARCHITECTURE, AND PERFORMANCE OF THE CHESS CHIPS, WHICH PROVIDED MOST OF DEEP BLUE'S COMPUTATIONAL POWER.

Feng-hsiung Hsu
IBM T.J. Watson
Research Center

●●●●●● Creating the first World Champion-class chess computer belongs among the oldest challenges in computer science. When World Chess Champion Garry Kasparov resigned the last game of a six-game match against IBM's Deep Blue supercomputer on 11 May 1997, his loss marked achievement of this goal.

The quest for a "chess machine" dates back to 1769 when the Turk—with a human player hidden inside—debuted in the Austrian court. The arrival of electronic computers in the late 1940s spurred new research interest in chess programs. Early programs emphasized the emulation of the human chess thought process. The Chess 4.5 program[1] in the late 1970s first demonstrated that an engineering approach emphasizing hardware speed might be more fruitful. Belle,[2] a special-purpose hardwired chess machine from Bell Laboratories, became the first national master program in the early 1980s. Following the same trend, Cray Blitz[3] running on a Cray supercomputer, and Hitech,[4] another special-purpose chess machine, became the top programs in the mid-1980s.

For the next 10 years or so, chess machines based on a move generator of my design[5]—ChipTest (1986-1987), Deep Thought (1988-1991), and Deep Thought II (1992-1995)—claimed spots as the top chess programs in the world. In 1988 the Deep Thought team won the second Fredkin Intermediate Prize for Grandmaster-level performance for Deep Thought's 2650-plus rating on the USCF's scale over 25 consecutive games.

Deep Blue's 1996 debut in the first Kasparov versus Deep Blue match in Philadelphia finally eclipsed Deep Thought II. The 1996 version of Deep Blue used a new chess chip designed at IBM Research over the course of three years. A major revision of this chip participated in the historic 1997 rematch between Kasparov and Deep Blue. This article concentrates mainly on the revised chip.

Task description and design philosophy

Good chess programmers pay a great deal of attention to their programs' speed. Initially, we also emphasized search speed. One of my original goals when we designed Deep Thought was to see what would happen when a Belle-class chess machine was speeded up, say, a thousandfold.

Solving the "computer chess problem"

involved winning a *match* against the human World Chess Champion under *regulation* time control. The games had to play out no faster than three minutes per move. Both conditions—using regulation time control and winning a match—presented problems.

Under faster time controls, it's much easier for a computer to beat chess Grandmasters. A computer first defeated a Grandmaster in 1977 in a blitz game, where players have five minutes each for the entire game. It took 11 years before a computer finally defeated a Grandmaster in a regulation game. More recently, in 1988, number-two-ranked chess player Vishwanathan Anand lost to a PC program in blitz and shortened time games by a score of 1.5 to 4.5. Yet, he defeated the same program by a score of 1.5 to 0.5. Even world-class Grandmasters have trouble avoiding simple mistakes in fast games.

The need to beat the World Champion in a match introduced additional difficulties. Two examples demonstrate the hurdle a computer encounters in a match against a well-prepared human opponent. In 1984 Cray Blitz played a four-game match against International Master David Levy. Despite the comparable playing strength, Levy successfully exploited Cray Blitz's typical computer weaknesses and won by a 4-0 score. In 1996 Deep Blue was even with Kasparov after four games in the first match. Kasparov pinpointed Deep Blue's weaknesses and won the remaining two games easily. Computation speed alone apparently didn't suffice.

What caused the problems for Cray Blitz and Deep Blue in the 1984 and 1996 matches? First, they played adaptable human opponents. Humans learn. Computers also "learn" but not very well. Second, Cray Blitz in 1984 and, to a much lesser degree, Deep Blue in 1996 had serious gaps in their grasp of chess knowledge, which a human opponent would simply exploit.

What countermeasures would help? Our team could put as much chess knowledge as possible onto the chess chip and reduce the number of serious knowledge gaps. We could also make it as easy as possible to change the chess chip's playing behavior. These melded into one mandate: to integrate the maximally possible amount of software-modifiable chess knowledge onto the chess chip. The

integration level had to come first; speed would be second.

Chess computers optimize precisely what their programmers tell them to optimize. Sometimes, this imperative leads the program to exhibit strange behavior not seen in human play—computers have no common sense. To correct for the strange behavior, we further have to include chess knowledge not covered in chess books.

To deal with previously unknown computer weaknesses showing up in a match, we made the weights associated with the positional features individually adjustable from software. This approach had the additional advantage that I didn't need to know how to play chess well, just whether a positional feature is important. An added hardware escape hatch allowed an external circuit, such as an FPGA (field-programmable gate array), to recognize new positional features and modify the chess chip hardware operations. In the 1997 match, we didn't use the hardware escape hatch, but we did change the weights for the positional features between games.

Speed, though secondary, remained important. Theoretically, we could increase the chip speed dramatically using better technology and speculative execution. The risk involved with a speedier implementation seemed too great in the limited time we had before the 1997 match.

The chess chips in 0.6-micron CMOS searched 2 to 2.5 million chess positions per second per chip. Recent design analysis showed that speedup to about 30 million positions/s is possible with a 0.35-micron process and a new design. Such a chip might make it possible to defeat the World Chess Champion with a desktop personal computer or even a laptop. With a 0.18-micron process and with, say, four chess processors per chip, we could build a chess chip with higher sustained computation speed than the 1997 Deep Blue.

Because of the Deep Blue system architecture, we did not have to have the fastest possible chess chip. We based the Deep Blue system on an IBM RS/6000 SP supercomputer, which you could view as a collection of IBM RS/6000 processors or workstations connected through a high-speed switching network. Each processor in the system controlled up to 16 chess chips, distributed over two Micro

Channel cards with eight chess chips on each card. Since the RS/6000 SP supercomputer in theory could have up to thousands of processors, we faced no serious limitation in the ultimate system speed. The 1997 Deep Blue has a 30-way machine with 30 RS/6000 processors (and 480 chess chips). There is at least one RS/6000 SP with over 4,000 processors. If willing to spend the money, you could build a chess machine more than a hundred times faster than the 1997 Deep Blue without resorting to faster chess chips.

The chess chips provided enormous computational power. On a general-purpose computer, the computation done by the chess chip for a single chess position requires up to 40,000 general-purpose instructions. At 2 to 2.5 million chess positions/s, one chess chip operates as the equivalent of a 100-billion-instruction/s supercomputer. Because this speed was adequate, I did not spend much time optimizing speed.

General architecture

The "Inside a chess machine" box describes the basic operating principles of a chess program. The following is Deep Blue specific.

System configuration

The 1997 version of Deep Blue included 480 chess chips. Since each chess chip could search 2 to 2.5 million chess positions/s, the "guaranteed not to exceed" system speed reached about one billion chess positions per second, or 40 tera operations. This assumes that, on average, each chess position needs 40,000 general-purpose instructions to process. The sustained speed reached 200 million chess positions/s, or about 8 tera ops. More software work could speed up the system by a factor of two to four, but we decided to apply software work in the chess knowledge area instead.

The search occurs in parallel on two levels, one distributed over the IBM RS/6000 SP switching network and the other over the Micro Channel bus inside a workstation node. For, say, a 12-ply search, one of the workstation nodes—working as the master for the entire system—would search the first four plies in software. (A ply represents a move by either player.) After four plies from the current game position, the number of positions increases about a thousand times. All 30 workstation nodes, including the master node, then search these new positions in software for four more plies. The number of positions increases by another thousand times. At this point, the chess chips jump in and finish the last four plies of the search, including quiescence search.

Partitioning the search into the (two-level) software search and the hardware search permitted a great deal of design flexibility, yet maintained overall search speed. The software handled less than one percent of the total positions searched, but it controlled about two thirds of the search depth. The software portion of the search can be arbitrarily selective without slowing down the system.

The eight plies of software search performed on the RS/6000 SP included many complicated search extensions, which extended the search deeper along lines the computer considered "forcing." Some experimental evidence suggested that the playing strength would increase significantly if the search extensions went all the way down to quiescence search. Implementing the full software search extensions on the chess chip seemed too risky a proposition, given the design time constraint. During the 1997 match, the software search extended the search to about 40 plies along the forcing lines, even though the nonextended search reached only about 12 plies.

Chip overview

Each chess chip operates as a full-fledged chess machine. Writing a chess program in silicon offered design possibilities not available in a software-based design. In particular, it required reexamining the competing algorithms' time complexity. An algorithm unacceptable in a software design might work perfectly well in hardware. Either the algorithm could be trivially parallelizable in hardware without significant area penalty, or the time-scaling factor might drop from the instruction cycle time to a simple gate delay. I'll explain some examples of these hardware-specific algorithms as they come up.

The chess chip divides into four parts: the move generator, the smart-move stack, the evaluation function, and the search control. The smart-move stack further divides into a regular move stack and a repetition detector. Figure 1 shows the chess chip's block diagram,

Inside a chess machine

Today's chess programs and chess machines alike follow the basic design Claude Shannon[6] outlined in 1949. Slate and Atkin's article on Chess 4.5[1] gave a classic example of the refinements developed up to the late 1970s, and their article remains relevant today—all modern chess automata are clones of Chess 4.5 in one way or another.

Shannon's basic design uses a minimax search to decide which move to play. The design assumes the existence of an evaluation function that assigns a numerical value to a chess position. Further, assume it assigns this numerical value from the computer player's viewpoint. The simplest evaluation function might return +1 if the computer wins, 0 if the position is a draw, and −1 if the computer loses. If we have enough computational power, the simple {+1, 0, −1} evaluation function, in combination with the minimax search, suffices for computing the best move. We simply search all possible moves for both sides until we reach positions with known outcomes.

For a position with the computer player to play and with all the outcomes of the children positions known, the position's value would simply be max{values of children positions}. So if one of the children positions is a win (+1) for the computer, the position is also a win (+1) for the computer. For a position with the computer's opponent to play, the value of the position is min{values of children positions}. If one of the children positions is a loss (−1) for the computer, the position is also a loss (−1) for the computer, as the opponent would pick the best outcome. By backing up the value according to this minimax rule, eventually the values for all the positions after the computer's possible first moves become known. The computer would simply play the move that leads to the maximum backup value, effectively "solving" the game.

Shannon observed that, in reality, chess was too complicated to solve this way. He proposed limiting how many moves from the present position the computer may search, instead of searching all the way to positions with known outcomes. This limit could depend on the time and the computational power available. With such an artificial limit, the simple {+1, 0, −1} evaluation function no longer worked, as the positions reached probably don't have known outcomes. Shannon proposed using a heuristic evaluation function: back up the heuristic values with the minimax rules, then choose the move based on the minimaxed heuristic values. The heuristic evaluation function for a position could rely on an estimate of the probability of winning, drawing, or losing the game from that position. Chess programs usually base the evaluation function roughly on how far ahead or behind the computer has gotten in units of pawns or hundredths of a pawn.

Shannon's basic scheme divides naturally into three components. They are the move generator that generates the chess moves and allows the search to go forward in time, the evaluation function that computes the values of future positions, and the search control that goes backward in time and backs up the future values to the present position. The Deep Blue chess chip added a smart move stack to detect when the chess position repeats.

By Chess 4.5, several important new developments had occurred. Two proved particularly important in designing a hardware chess move generator: quiescence search and the alpha-beta pruning algorithm.

Also called capture search in its simplest form, quiescence search extends search beyond the original limit set by Shannon's scheme. For a capture search, the side to move gets the option of making capturing moves to gain material as well as simply accepting the position as is. If the moving side selects the capturing option, the opponent gets the option of making his own captures as well as accepting the new position as is. The capture search (Figure A) can go on for many moves until one side runs out of captures or until one side decides to take the position as is.
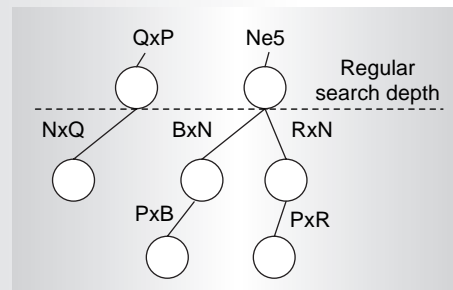

Figure A. The capture search takes place beyond the regular search depth, until one side runs out of captures or one side decides to accept the position.

Besides capturing moves, more general quiescence search might include other types of forcing moves, say, checking moves.

Quiescence search has a major impact on a chess program's performance. An early measurement[7] showed that a program with quiescence search matched a program without quiescence search but searching four plies (two moves each for white and black) deeper. Quiescence search usually increases the number of positions searched to the same depth by only two to four times. Four more plies of search, on the other hand, usually increase the number of positions by up to a thousand times. It's no wonder all modern chess programs use some sort of quiescence search.

A program with quiescence search usually spends at least half of its computation time there. Therefore, computation speed for quiescence search proves critical for high-performance chess computers. All successful hardware chess move generators can generate at least the capturing moves quickly. The Deep Blue chess chips can generate other forcing moves quickly as well.

The alpha-beta pruning algorithm appeared several years after Shannon's proposal. In reality, human players have used the alpha-beta algorithm or its variants implicitly for ages.

continued from p. 73

The alpha-beta algorithm relies on the observation that we don't really need to look at all of an opponent's responses to our bad moves—we just need one refutation. If the bad move endangers (hangs) the queen, we just need to know that our opponent can capture the queen on the next move. However, to determine a valid refutation, we must examine all of our responses to it. In the hung-queen example, we would need to examine all of our responses to the queen's capture to make sure that our opponent indeed wins the queen without adequate penalty. The same refutation principle applies to the opponent's bad moves, which also just need one refutation each. This idea of refutations leads to an optimal search tree (see Figure B).

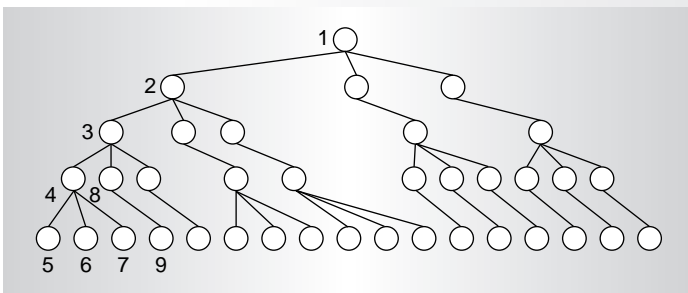The search tree grows top down and left to right. An optimal

Figure B. Optimal search tree.

search tree orders the moves best-first, so the first move searched for a given position is also the best move or at least a refutation move for that position. In the figure, the leftmost branch of the search tree also represents the principal variation—the hypothetical line where both sides play the best moves. Thus, we must examine all responses to them—no refutation to a best move exists. On the other hand, all the sibling moves to a principal variation move are inferior and have at least one refutation we must examine.

A refutation line exhibits the repeated pattern of a tree level with one refutation followed by a tree level of all responses. This is the characteristic growth pattern of an alpha-beta search tree. The algorithm lets a chess program search to roughly twice the depth achievable with a minimax search when the move ordering is close to best-first ordered.

For a program searching close to 40 billion positions for each move—as the 1997 Deep Blue did—the alpha-beta algorithm increases the search speed by up to 40 billion times. This speedup, however, depends strongly on the move ordering's quality. In worst-first move ordering, the alpha-beta algorithm searches a tree the same size as the minimax search tree. Obviously, the hardware move generator would need to provide a decent approximation of the best-first ordering in any high-performance chess-playing system.

Another development not usually cited but, to the best of my

knowledge, first described by Slate and Atkin,[1] was lazy evaluation. Chess 4.5 avoided calculating the entire evaluation function when the material balance veered too far from the expected value. For example, if either side has lost a queen while we're expecting the position to be even, clearly there's no reason for a complete evaluation. The normal evaluation function, at least in the case of Chess 4.5, could not possibly bring the evaluation function anywhere close to even.

The Deep Blue chess chips use a more elaborate lazy evaluation scheme, which disables lazy evaluation when an unusual position occurs. In particular, if the number of pieces on the board drops too low, sometimes a position could be drawn even with one side up a full piece. A deep, brute-force search has a high percentage of off-balance leaf positions as a direct consequence of searching all the moves, which naturally include some outrageously bad moves from either side. Typically, for a 12-ply brute-force search, 60% to 90% of positions are off by more than a pawn. Lazy evaluation can thus become quite effective.

At the left in Figure C is a tree for a three-ply search from the opening position. After three plies, the search reaches a leaf position and enters the quiescence search region. Some chess programs restrict quiescence search to capturing moves only. Deep Blue includes checking moves and check evasion moves in its quiescence search under certain conditions. This meant adding special circuits to generate such moves quickly.

The flow chart on the right side of the Figure C gives a simplified view of the processing done for each chess position searched. Entering a chess position after making a move, the chess machine processes two parallel paths: move generation and decision evaluation.

The left path—the move generation path—first checks for the legality of the opponent's most recent move by checking whether we can capture the opponent's king. If so, the last move the opponent made is illegal, and we should exit the position and return to the parent position. If the last move is legal, we start the move generation process. If we cannot find a move (either no legal moves exist or, in the case of quiescence search, no suitable forcing moves exist), we return to the parent position, possibly with whatever score the evaluation function provides. If we do have a move, and the evaluation function says we cannot exit yet, we continue the search to the next level.

The right path handles evaluation decisions. On entering the position, we first check whether it's a leaf position (usually by checking whether we have reached sufficient search depth). If not, we don't need to do the evaluation function, and we merge with the move generation path. If we have reached a leaf position, we do a fast evaluation, which gives a quick and dirty estimate of the positional score. If we like this estimate, we take it and exit the position (with an early termination of the move generation path). If this estimate looks very bad for us, we merge with the move generation path, hoping to find some forcing move that wins something for us. If this estimate seems neither good
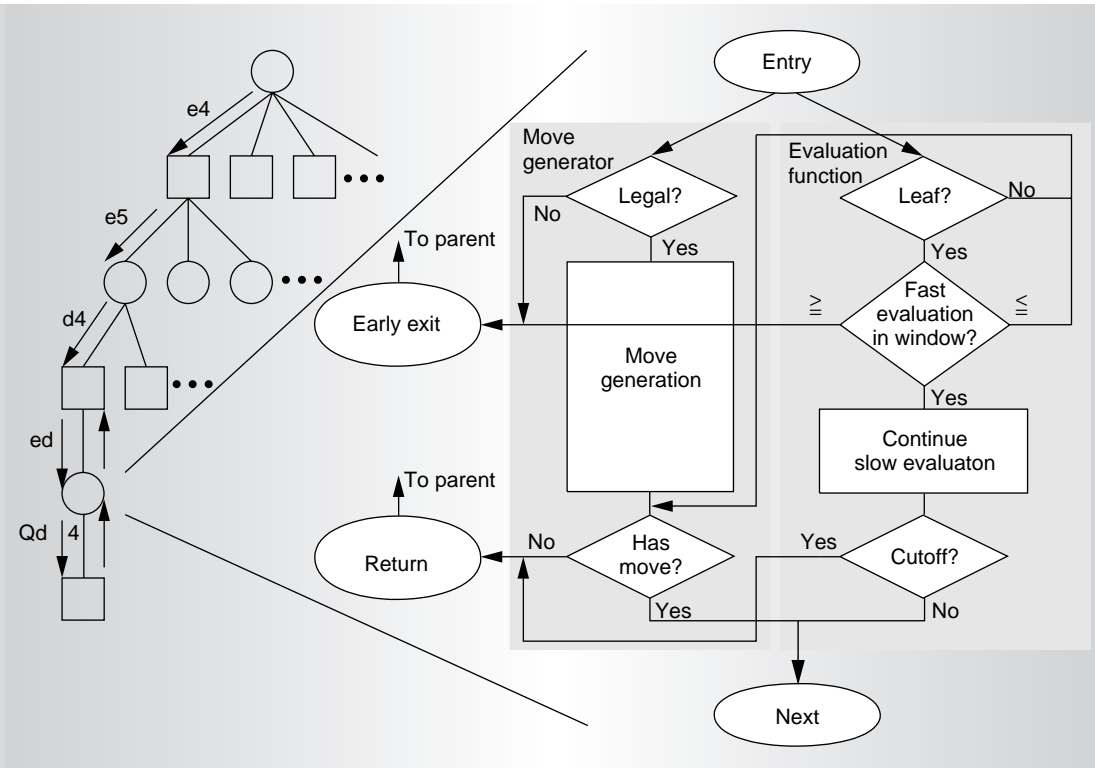
Figure C. A chess chip's basic search algorithm: search tree (left), flow chart (right).

nor bad, we compute the slow evaluation to find the full evaluation for the position. If the full evaluation satisfies us, we can cut off the search and return to the parent position. Otherwise, if the move generator says a suitable forcing move exists that might win something for us, we search forward.

along with the connections between the blocks. The chess chip's die photo appears in Figure 2.

The chip's cycle time lies between 40 and 50 nanoseconds in a 0.6-micron, three-metal layer, 5-V CMOS process. The average number of cycles per position searched is about 10; power dissipation is about one watt. This power consumption level falls within the
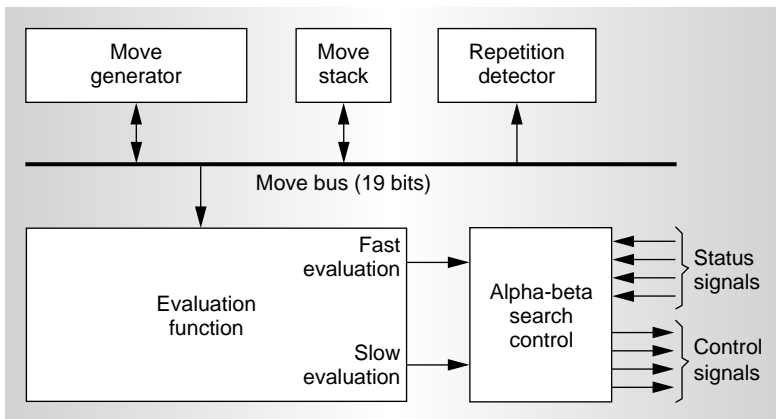

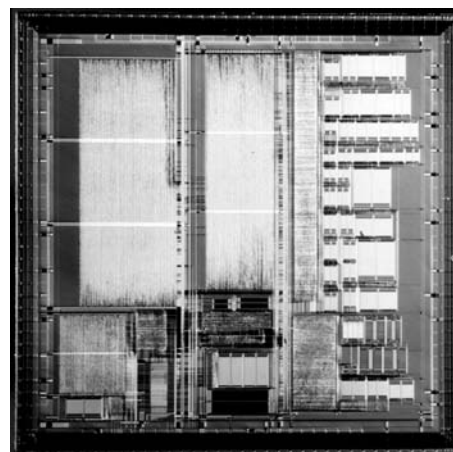Figure 1. Block diagram of the chess chip.

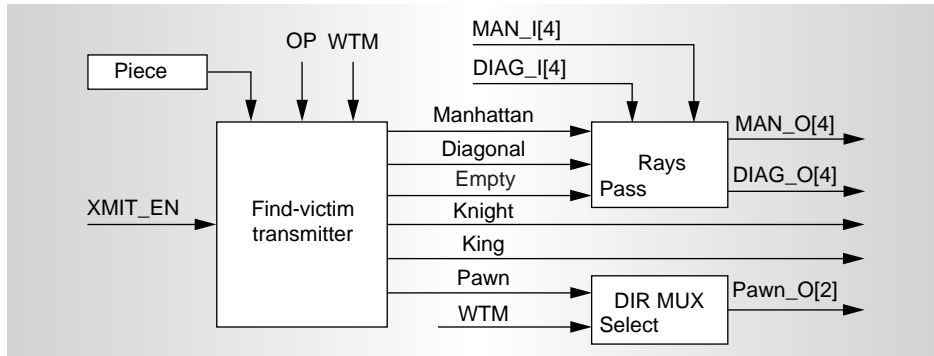Figure 2. Die photo of the chess chip.

Figure 3. The find-victim transmitter. WTM: white to move, OP: operation, DIR mux: direction multiplexer.

power-handling limit of the Micro Channel bus for an eight-chip board. The chip includes about 1.5 million transistors, and the die measures $1.4 \text{ cm} \times 1.4 \text{ cm}$.

Move generator

The move generator, an 8×8 array of combinational logic, appears in the die photo as the block at the upper right. A hardwired finite-state machine controls move generation. The move generator is more complicated than the move generator used in Deep Thought,[5,8,9] which cannot generate the checking or check evasion moves directly. The new move generator can generate capturing, checking, and check evasion moves directly. The move generator used in the chess chips for the 1997 Deep Blue could also generate attacking moves. This 1997 addition supports hardware pruning of irrelevant chess moves at the last few plies of positions immediately before quiescence search. The basic move generation algorithm is the same as in the Belle move generator.[10]

The combinational logic array is effectively a silicon chessboard. Each cell in the array has four main components: a find-victim transmitter, a find-attacker transmitter, a receiver, and a distributed arbiter. Each cell contains a four-bit piece register that keeps track of the type and color of the piece on the corresponding square of the chessboard.

When enabled, the find-victim transmitter (see Figure 3) radiates appropriate attacking signals for the resident piece. If the square is vacant, incoming attack signals from a ray piece (a bishop, a rook, or a queen) pass through the cell. Third-rank squares have additional circuits to handle the two-square

pawn moves. At the start of a typical move generation sequence, a find-victim cycle executes, and all of the moving-side's pieces radiate attacking signals.

The radiated attacking signals then reach the receiver in Figure 4, and a vote takes place to find the highest valued victim. Although the receiver has several different operations, for now, assume that it determines whether some piece of opposing color attacks the resident piece. If yes, the receiver asserts a priority signal based on the piece type. Since we want to find victims, the priority rises for higher value pieces, with the queen highest, then rook, bishop, knight, pawn, and empty square in descending order. The king cannot become a victim, as then the position is illegal.

The priority signals from all the squares then go to the distributed arbiter, or the arbitration network in Figure 4, to find the highest value victim to capture. For victims with the same piece priority, additional square priority breaks the tie.

With the victim chosen, the find-attacker cycle executes. The find-attacker transmitter (see Figure 5) on the victim cell transmits reverse attacking signals as if it were a super piece—a piece that can move in reverse directions of any of the possible attacking pieces.

The receivers on all the squares then detect whether an incoming reverse attacking signal matches the resident piece's type. If the resident piece is an appropriate attacker, the system asserts a priority signal. Since we want to use the lowest valued attacker first, the priority takes reverse order, with the pawn having the highest priority. The priority signals then go through the arbitration network, and, with both the victim and the attacker computed, we have the move.

You may have noticed that the move generator, as described, computes all the moves implicitly even though it generates only one move. In software, we would consider this unacceptable. A hardware implementation computes all the moves in parallel and incurs no time penalty.
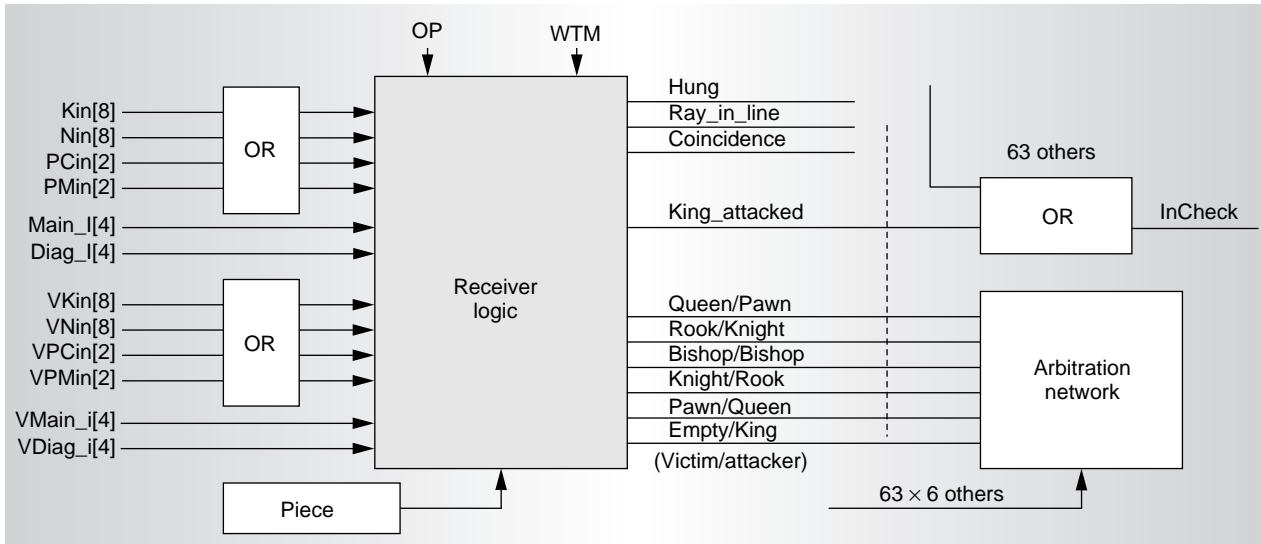
Figure 4. The receiver.

This discussion holds for the first move generated from a new position. Generating the other moves from a partially searched position requires masking off the moves already searched.

In the earlier Belle design, a 2-bit disable-stack masks off squares already searched. One disable-stack bit masks off attacker squares already searched for a given victim. The other bit masks off fully searched victims. This part of the Belle design relies on a direct translation of a software implementation, where bit masking proves efficient. A VLSI-based design, however, offers a new alternative. In the Deep Thought and Deep Blue chess move generators, the last move searched from the position is used to compute the two-bit disable-mask. The last searched move tells us the priority levels of the last victim piece and the last attacking piece. We can use a modified decoder to mask off squares with the same type of resident piece but lower square priority. In software, the cost of recomputing the mask exceeds the cost of retrieving it. But in hardware, the reverse holds true. The disable-stack needs decoders to operate anyway. The modified decoder operates at a speed comparable to the disable-stack decoders and
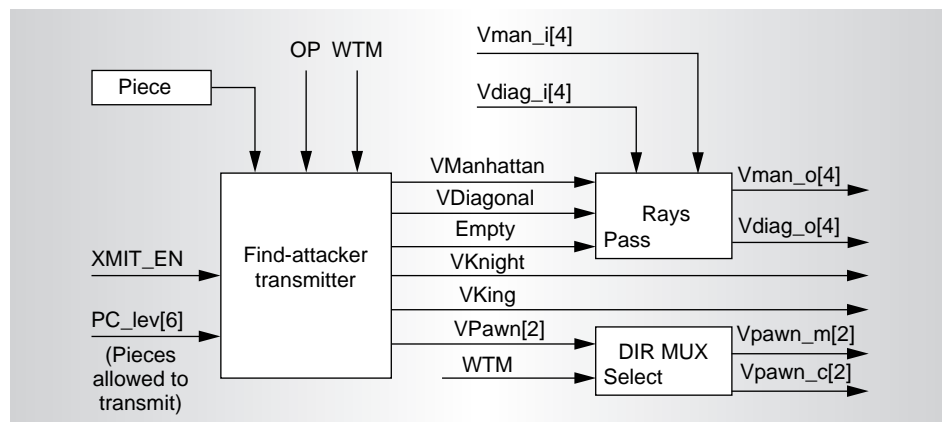

Figure 5. The find-attacker transmitter.

takes less space. This method avoids using the disable-stack, which probably needs to be at least 64 levels deep.

The Deep Blue move generator supports other move generation modes. When generating checking moves, it activates all the find-victim transmitters as well as the opposing king's find-attacker transmitter. When both sets of signals coincide on the same square, we have a square from which a piece can check the king. When ray signals align properly on a square with a piece belonging to the moving side, the piece can give discovered checks when it moves. A special move-generation mode generates check-evasion moves. Another mode generates moves used in hardware move pruning. This last mode would cost too
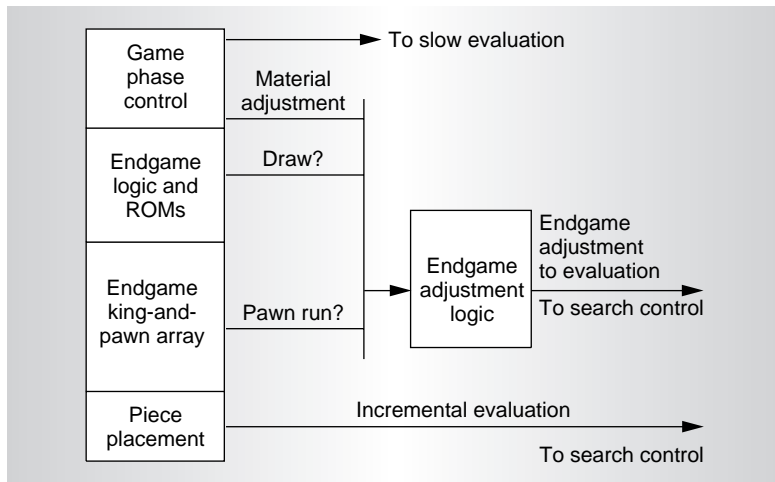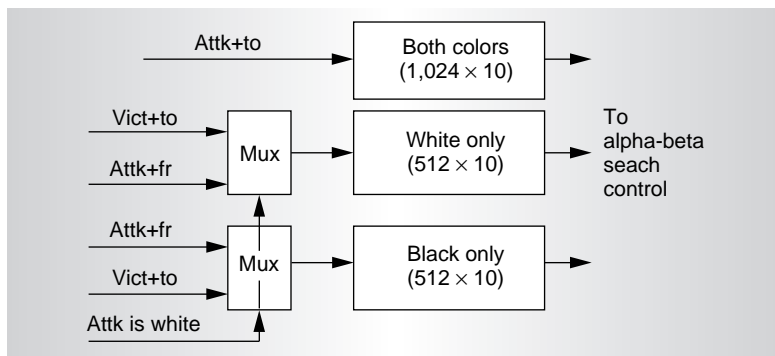
Figure 6. Fast evaluation.

Figure 7. Piece placement table. Attk = attacker

much in a pure software implementation.

The move generator also detects hung pieces and measures the forcefulness of checks. We can enable simple hardware search extensions at the first level of quiescence search, and we used these in the 1997 version of Deep Blue. The move generator has about 52,400 gates.

Evaluation function

The entire evaluation function contains about 66,000 gates, not including the RAMs and ROMs. In Figure 2's die photo, all the sub-blocks to the right of the move generator belong to the evaluation function. The lower sub-blocks provide fast evaluation. The upper sub-blocks, the systolic evaluation array, the pipelined evaluation RAMs, and the pipelined postevaluation logic compute slow evaluation. Fast evaluation, which computes in a single cycle, contains all the easily computed major evaluation terms with high values. Slow eval-

uation scans the chess board one column (file) at a time systolically. Slow evaluation has a latency of three cycles per column and takes 11 cycles to compute, given the chessboard's eight columns. Partitioning the evaluation function into fast and slow evaluation proved necessary to fit the design into a single chip. Fortunately, we seldom need slow evaluation.

Fast evaluation contains four parts, as depicted in Figure 6: the piece placement table, the king-and-pawn array, the endgame logic and ROMs, and the game phase control. The piece placement table computes a position's incremental evaluation based on the average values of the pieces on their resident squares. The piece placement table actually uses three RAMs, as shown in Figure 7. The game phase control block contains a piece-count register that counts the number of pieces of each type left on the board for either player. For every type of piece, the system also maintains an XORed piece-location register. The piece-location register contains the precise location of a piece of a particular type if only one piece of the right type remains. The piece-count register addresses game phase control RAMs.

The game phase control RAMs produce several control values. One of the control values is simply a bonus or penalty to be added to the evaluation based on the material left on the board. Good piece combinations get a slight increase in evaluation. The other control values serve are game phase dependent multipliers for other evaluation weights. The most important game phase control value, the king-safety relevance, tells the chess chip the importance of king safety. Another game phase control value tells the chess chip how to adjust the penalty for bad pawn structure and the bonus for passed pawns.

The king-and-pawn array mainly detects the "pawn can run" condition, where the king can no longer catch up with an opponent's passed pawn. This means that the passed pawn effectively becomes a queen if no other piece remains, assuming no other pawn can win the pawn race. The king-and-pawn array recognizes other endgame features, including some special rook-versus-passed-pawns conditions and the presence of widely separated passed pawns that can outrun the king.

The endgame logic and ROMs block mainly recognizes unusual endgame conditions that

lead to draw endings. This block contains random logic to detect certain simple endings that are very likely drawn. This block also recognizes some simple fortress draws. Figure 8 shows the endgame ROM interface to the four endgame ROMs. The biggest ROM is the king-and-pawn versus king ROM, which tells whether the position represents a win for the pawn side.

Slow evaluation constitutes the single most complicated element on the chip, occupying close to half of the chip core. At very shallow depths, about half the positions searched require slow evaluation, but at realistic search depths, the percentage drops to around 15 percent. As shown in Figure 9, slow evaluation has a three-stage pipeline, starting with an 8×1 systolic array that runs for eight cycles, one cycle per file. Next are the 40-plus synchronous RAMs, followed by an adder tree that accumulates the results. The controller can stop the slow evaluation sequence on the fly to reduce power consumption.

Slow evaluation computes values for chess concepts such as square control, pins, x-rays, king safety, pawn structure, passed pawns, ray control, outposts, pawn majority, rook on the 7th, blockade, restraint, color complex, trapped pieces, development, and so on. This chess evaluation function probably is more complicated than anything ever described in the computer chess literature. As an example, look at the king safety evaluation. Before the king castles, the system computes three king safety evaluations, one for king-side castling, one for queen-side castling, and the base value for staying in the
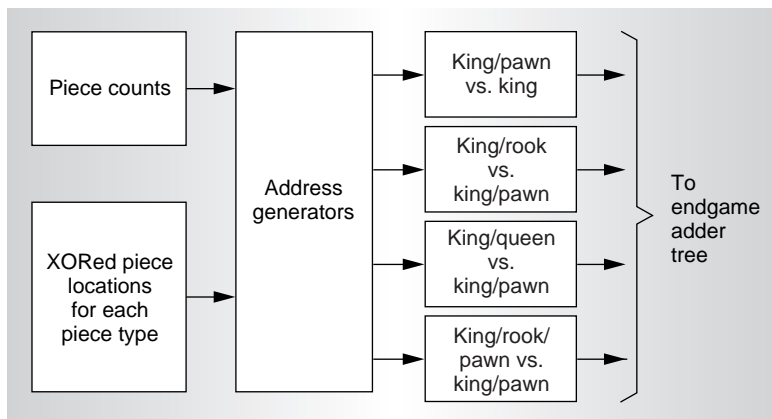

Figure 8. Endgame ROM interface.

center. Each of these king safety evaluations takes into account the types of pieces attacking, the soundness of the king's shelter, presence of attacking pawns, color complex around the king, square and ray control around the king, and so on. The final king safety evaluation is a weighted linear combination of the three king safety evaluations. The weights used in the linear combination are based on how easy it is to castle as well as the relative safety ranking of the three destinations for the king.

Smart move stack

The smart move stack does not exist in the chess chip's older release. The old chip contains a regular move stack, but not the repetition detector. The repetition detector contains a 32-entry circular buffer of the last 32 plies of moves. Using a hardware content-address-
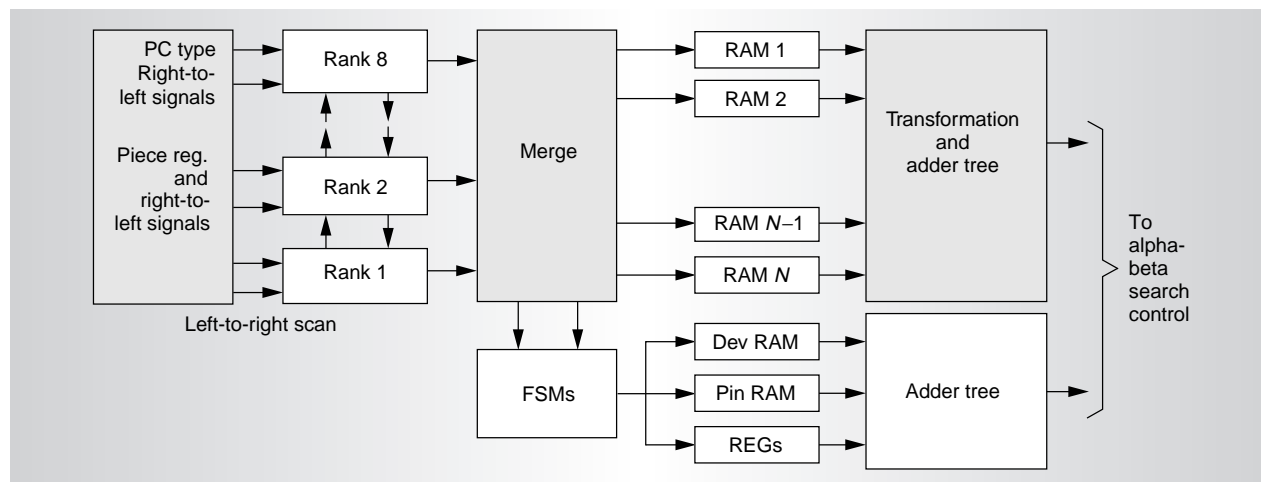

Figure 9. The main slow evaluation flow.

able memory algorithm,[8] the repetition detector maintains the numbers of pieces displaced in each of the last 32 positions with respect to the current board position. When the number of pieces displaced equals zero, we have a repeated position.

The repetition detector actually detects more than just simple repetition. If there is only one piece displaced, it also detects whether there is a move that might lead to repetition or near repetition. If the move is legal, the side to move can at least claim a draw.

The repetition detector as implemented has a time complexity of $O(n)$, when n is the depth of the repetition circular buffer. Software usually implements repetition detection by probing a hash table, which gives a time complexity of $O(1)$. The big difference here is that the time constant for the hardware repetition detector equals the gate delay instead of the instruction cycle time. Also, normal software repetition detectors cannot tell us that a position is about to repeat.

The repetition detector uses about 20,000 gates.

Search control

The search control does not really implement the regular alpha-beta search algorithm.[11] Rather, it implements a minimum-window alpha-beta search algorithm.[12] This eliminates the need for a value stack. A regular alpha-beta search maintains two temporary variables, α and β, on a value stack. The relatively new minimum-window search can only tell us whether the position searched is better or worse than a single test value. In the regular alpha-beta search, for moves not the best, we just need to know whether they're better or worse (refuted, then) than the current best move—precisely the same function provided by the minimum-window search. Of course, when the new move is better than the current best move, we may need to research the new move. We can either use a regular alpha-beta search or repeat the minimum-window search multiple times, raising the test value slightly each time. Efficiency-wise, minimum-window-based search seems about the same as the regular alpha-beta search.

The search control contains a 16-bit data path and three state machines controlling sections of the data path. Two of the state machines also control the move generator indirectly. The data path uses multiple cascaded adder/subtractors to compute the conditional flags that the search algorithm needs in as few cycles as possible.

One unusual search control feature, a low-pass digital filter, estimates the slow evaluation function. The last observed slow evaluation serves as the input to the low-pass filter. This filter gives us the low-frequency component of the slow evaluation function as the search progresses. Since the slow evaluation function does not change drastically for closely related positions, its low-frequency component offers a decent estimate. Without this estimate, we would have to widen the window used to determine whether to compute the slow evaluation function. Consequently, the search speed would suffer.

At the system level, the chip appears as a 32-bit device with a 17-bit address space. Writing to some of the addresses initiates a search from the current position on the chip, usually for four or five plies beyond the software search depth. This frees up the host processor to perform housekeeping chores or initiate a search on another chip.

Performance

We have used the chess chips in various system configurations, from a single chip to multiple chips running on a single workstation (the Deep Blue Jr.) and finally to the full Deep Blue. The earliest games, in early 1997, used a single chip running at 70% clock speed and at one-tenth to one-fifth efficiency as the result of a hardware bug. This reduced the chip to 7% to 14% of its regular speed, or about the same search speed as the fastest commercial chess program on a Pentium Pro 180 MHz PC. Two of the top commercial programs, running on the Pentium Pro PC, served as opponents in the early chip debugging sessions. Of the 10 games played, the single-chip program won all 10. This gives about a 95% confidence level that a single chip, even at reduced speed, was at least 200 points stronger than the commercial chess programs in machine-versus-machine play.

We played another 30 games with either the single-chip version or Deep Blue Jr. against the commercial chess programs. Of the 40 games total, the chess chip(s) lost two points and scored 95 percent against the PC pro-

grams. This places the performance 300 to 500 rating points higher than the PC programs, depending on the rating formula used. This rating has no bearing on the real playing strength, as cursory examination showed serious positional weaknesses in the commercial programs that the chess-chip systems exploited repeatedly.

The more interesting games pitted Deep Blue Jr. against the Grandmasters working on the project. The Grandmasters' average ratings were in the high 2500s on the international scale. Deep Blue Jr. scored better than a three-to-one ratio against them, which placed it at 2700 plus, or among the top 10 players in the world.

The 1997 version of Deep Blue only played six games, all against Kasparov. Deep Blue won the match by the score of 3.5 to 2.5. Kasparov is rated around 2815, which placed Deep Blue's performance at about 2875. However, we couldn't take this rating too seriously because of the small sample size.

I am forming an independent start-up to create a new chess chip for consumers. This new chip could make it possible for a desktop machine to defeat the World Champion in a formal match as early as the year 2000. Further down the road, the skills used to create Deep Blue could be used to conquer other games. One of the prime candidates is the Japanese game of *shogi*, which has higher computational complexity but also similar characteristics. The difficult game of Go, or *Wei-chi*, however, might still be well beyond a computer's reach in the near future. MICRO

Acknowledgments

References

1. D.J. Slate and L.R. Atkin, "Chess 4.5–the Northwestern University Chess Program," *Chess Skill in Man and Machine*, P.W. Frey, ed., Springer-Verlag, 1977, pp. 82–118.
2. J.H. Gondon and Ken Thompson Belle, *Chess Skill in Man and Machine*, 2nd. ed., P.W. Frey, ed., Springer-Verlag, 1983, pp. 201–210.
3. R.M. Hyatt, "Parallel Chess on the Cray x-mp/48," *Int'l Computer Chess Assoc. J.*, 1985, pp. 90–99.
4. C. Ebeling, *All the Right Moves: A VLSI Architecture for Chess,* PhD thesis, Carnegie Mellon Univ., Computer Sci. Dept., Pittsburgh, Apr. 1986.
5. F.-h. Hsu, "A Two Million Moves/sec CMOS Single-Chip Chess Move Generator," *1987 ISSCC Digest Technical Papers*, Feb. 1987, p. 278.
6. C.E. Shannon, "Programming a Computer for Playing Chess," *Philosophical Magazine*, Vol. 41, 1950, pp. 256–275.
7. J.J. Gillogly, *Performance Analysis of the Technology Chess Program,* PhD thesis, Carnegie Mellon Univ., 1978.
8. F.-h. Hsu, *Large-Scale Parallelization of Alpha-Beta Search: An Algorithmic and Architectural Study,* PhD thesis, Carnegie Mellon Univ., Computer Science Dept., Feb. 1990.
9. F. Hsu et al., "Deep Thought," *Computers, Chess, and Cognition*, T.A. Marsland and J. Schaeffer, eds., Springer-Verlag, 1990, pp. 55–78.
10. J.H. Condon and K. Thompson, "Belle Chess Hardware," *Advances in Computer Chess 3*, Pergamon Press, Oxford, England, 1982, pp. 45–54.
11. D.E. Knuth and R.W. Moore, "An Analysis of Alpha-Beta Pruning," *Artificial Intelligence*, Vol. 6, 1975, pp. 293–326.
12. J. Pearl, *Heuristics: Intelligent Search Strategies for Computer Problem Solving,* Addison-Wesley, Reading, Mass., 1984.

Feng-hsiung Hsu is a research scientist at IBM T.J. Watson Research Center. His current technical interests include parallel systems, special-purpose architectures, VLSI circuits, reprogrammable systems, hardware algorithms, computer shogi, and "things simpler than a chess machine." Hsu received his PhD in computer science from Carnegie Mellon University in 1989 and his bachelor's in electrical engineering from National Taiwan University in 1980.

Readers may contact Hsu at IBM T.J. Watson Research Center, MS 27:233, PO Box 218, Yorktown Heights, NY 10598; e-mail fhh@watson.ibm.com or fhh@cs.cmu.edu.