7.4 Belle Chess Hardware

Joe Condon and Ken Thompson

First published in *Advances in Computer Chess 3* (M. R. B. Clarke, editor), Pergamon Press, Oxford, 1983, pp. 45-54. Reprinted by permission of Pergamon Press Ltd.

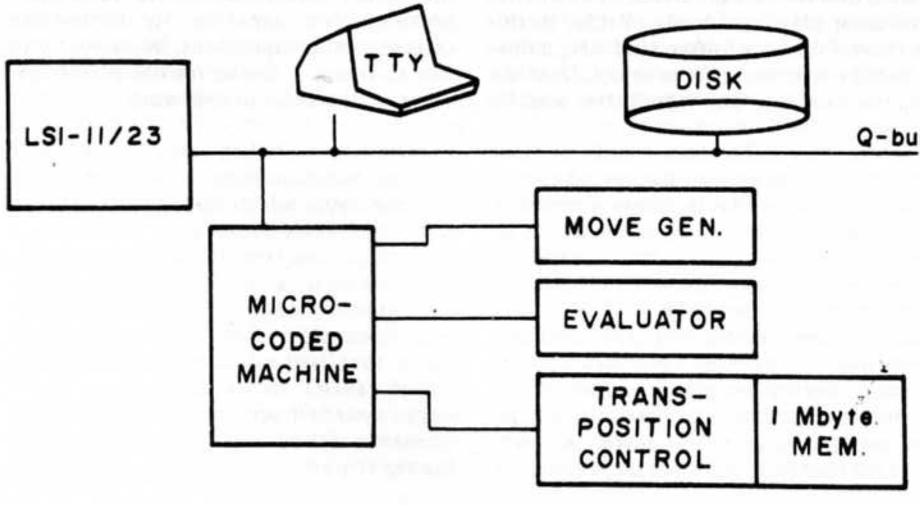
Abstract

The computer chess program BELLE is currently the World Computer Chess Champion and the North American Computer Chess Champion. In human play, BELLE has consistently obtained master performance ratings. This paper describes the special-purpose hardware that gives BELLE its advantage: speed.

Introduction

The first version of BELLE was written in 1973 and first competed in the fourth ACM Championships in Atlanta. A slightly revised version competed the next year in San Diego. This experience made it painfully clear that computer chess belonged to the fastest computer. During this time, others were working on chess hardware. Moussouris, Holloway and Greenblatt (1979) constructed a highly publicized machine but failed to include an evaluation function. Despain designed a chess machine, described by Babaoglu (1977), but never obtained funding for full construction. The authors built a small (25 chip) machine and entered it in the 1977 World Computer Chess Championships in Toronto. This first machine at best broke even on speed, but it whetted our appetites and gave us valuable experience. The next machine was quite a bit larger (325 chips). It had three separate sections - a move generator, a position evaluator and a transposition table. The alpha-beta search (Kunth and Moore, 1975) was still carried out by a minicomputer. This hardware/software combination proved to be comparable to the largest mainframes. It entered the 1978 (Washington DC) and 1979 (Detroit) ACM Championships, placing first and second respectively.

The latest machine was designed and constructed over a period of six months and contains approximately 1700 chips. The overall structure is the same as the previous machine, but the alpha-beta search has been embedded into hardware. The machine is controlled by an LSI-11/23 running a general purpose time-sharing system.



The LSI-11 contains the chess programs, opening book, and operating system software. It interfaces to a micro-coded alpha-beta machine that in turn controls a move generator, an evaluator and a transposition table memory.

The software version of BELLE examined chess positions at about 200 positions per second. The first chess machine did not change the speed much. The second machine increased the speed to about 5000 positions per second. The current machine runs at about 160,000 positions per second. These numbers are hard to compare, but the chess-specific computing power has increased about 800 times while the general purpose computing has decreased by a factor of 5.

Move Generator

The move generator is arranged in an 8 × 8 array of similar (except for edge effects) combinatorial circuits. The move generator performs two transformations. The transform to be performed is selected by a single wire *op-code.* The op-codes are named FIND-VICTIM and FIND-AGGRESSOR.

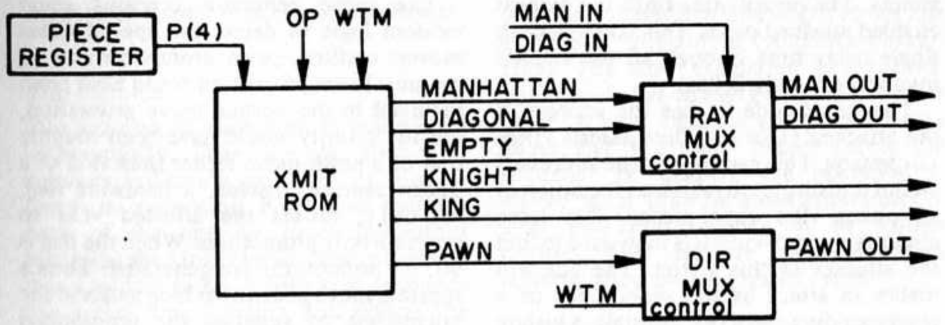
The FIND-VICTIM operation locates the highest valued piece under attack. The FIND-AGGRESSOR op-code finds the lowest valued piece attacking a given square. The given square is, of course, the previously found VICTIM square. The AGGRESSOR and the VICTIM squares are then the *from* and *to* squares for a chess move.

Transmitter

Each of the 64 circuits on the move generator consists of a transmitter, a receiver, a four-bit register holding the piece and a one-bit disable memory. Each transmitter sends signals to its (chess) neighbours. Likewise the receiver accepts signals from its neighbours.

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 is true if White is on the move.

The rays (manhattan and diagonal) are propagated or generated at a square by a multiplexor selected with the square being empty or not. There are eight ray multiplexors on each square; one for each ray direction. The pawn attack outputs are multiplexed north or south selected by WHITE-TO-MOVE.

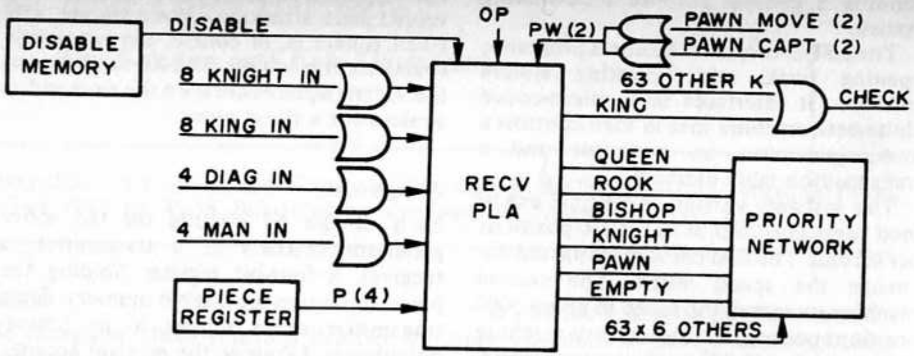


Receiver

The receiver section analyzes the attacks generated by the transmitters. The main component of the receiver is a programmed logic array (PLA).

The eight knight attacks are *OR*ed together. Similarly the eight king attacks, the four diagonal (bishop) ray attacks, the four manhattan (rook) ray attacks, and the two pawn attacks are *OR*ed together. 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.

The king attacked is *OR*ed over all 64 squares and is treated specially as a CHECK condition code for the micro-coded pro-



cessor. The job remaining is to select the highest priority of the remaining 64 × 6 signals and to discover where this signal comes from. This is done in a two level priority encoding tree. The first level is performed on a local group of 4 × 4 squares. The second level priority selection determines which prioity level is highest and which 4 × 4 section generated this level. This information is combined with the location on the 4 × 4 section to complete the task.

Move generation is performed as follows. The micro-code asserts FIND-VICTIM. This causes each transmitter to activate signals corresponding to the piece on each square. The receiver sections then activate priority leads for all enabled attacks. The priority tree finds the highest enabled attacked piece. This takes place in ripple delay time through all the circuits involved (roughly 250ns).

The micro-code latches the address of the attacked piece and then asserts FIND-AGGRESSOR. This causes only the addressed victim transmitter to radiate as the union of all pieces (the super-piece). The same circuitry as FIND-VICTIM is then used to find the attacker of this victim. The PLA will match an attack by the super-piece on a corresponding piece (for example, a bishop attacked by the super-piece, but only down a diagonal). The priorities are inverted by the PLA so that the lowest valued aggressor on the selected victim is found. This operation takes another 250ns.

The micro-code latches the aggressor location and then makes the indicated move. (The aggressor is moved to the victim and the old aggressor is cleared.) After the processing of this move, the original victim and aggressor are restored. The aggressor is disabled and the next lowest priority aggressor is found until there are no more aggressors on this victim. The victim is disabled and all potential aggressors (friendly pieces) are re-enabled. Then the next victim is found and then all aggressors on this victim. This continues until all victims are exhausted.

This process is recursive. The 64 disable bits represent the state of the move generation. This state has to be saved and all disable bits must be cleared as a new move generation is started. The 64 bits are really a 64 word stack of 64 bits each. The state is saved by incrementing the address on the disable stack. The state is restored by decrementing the stack address.

The move generator contains some random logic to detect the special chess moves: castling, pawn promotion and en passant. Pawn promotion could have been included in the normal move generation, but the priority would have been roughly that of a pawn move rather than that of a queen capture. Instead, a hardware flag, PRO-ONLY, causes the affected PLAS to generate only promotions. When the flag is off, no promotions are generated. Thus a separate move generation loop exists in the micro-code to generate the promotions first. Since promotions are fairly rare, the micro-code usually only executes a single FIND-VICTIM + PRO-ONLY to determine that there are no promotions.

Incremental Evaluation Function

The evaluation function is broken into two evaluation functions, the incremental evaluation function and the slow evaluation function. As the name implies, the incremental function consists of a series of accumulators that are incrementally updated as the chess board is altered. A two-phase machine tracks all write operations to the chess board. As a square is written, the incremental evaluator reads the old contents, issues a SUBTRACT cycle, then allows the new contents to be updated and issues an ADD cycle.

The Material Register

The total board material is maintained in a pair of registers – one for White, one for Black. Two bits in each register are allocated to count each of the queens, rooks, bishops and knights. Four bits count the number of pawns. Several material evaluations are accomplished by ROMS addressed by the material registers. The total material is calculated as queen = 900, rook = 500, as given in beginning texts. Certain cooperating pieces are given bonuses such as a bishop pair and a queen plus knight. Minimum mating material is evaluated by discounting a bishop or knight with no supporting pawns. A material balance is evaluated to encourage trading pieces, but not pawns, when ahead. Another ROM attached to the material registers calculates the stage of the game for each side. This is done by estimating the amount of enemy material on the board.

King Position

The location of each king is maintained incrementally. There are also four 9-bit registers indicating existence of friendly pawns on each wing. Four ROMS evaluate each of the possible pawn formations assuming the king is behind the pawns and it is a middle game. A fifth ROM examines the actual king location and selects the pawn structure corresponding to the king's location. This ROM also evaluates the king's centralization assuming it is an end game. The actual evaluation is a weighted sum of the king safety and king centralization selected by the stage of the game.

Piece Position

Every piece has a value for occupying a square. This value is maintained incrementally. Pawns are more valuable for being centralised and for being advanced. Knight mobility is statically evaluated based on the squares that they occupy. Knights and bishops are penalized for being in their original square.

Hash Code

The incremental evaluator maintains a 48-bit hash code of the current position. This is used by the transposition table described below. For every piece on every square (16 × 64 = 1024) a ROM generates a different constant. This constant is XORed into the incremental hash accumulator on every ADD or SUBTRACT.

All of the above incremental evaluations are summed and the total is made available to the micro-code as a "fast evaluation". The fast value is available to the micro-code in less than an instruction time. Thus the micro-code can alter the board in one instruction and use the fast evaluation in the next. A conservative fast evaluation can quite often be used to cause a beta cutoff without the need for a full evaluation.

Slow Evaluation

The slow evaluation consists of small positional considerations added to the fast evaluation to obtain the full position evaluation. The slow evaluation is asynchronously initiated on every write to the chess board. It is an eight cycle sequence taking about 2μs. If another occurs before the sequence is finished, the operation is started again. Like the move generator, the slow evaluator consists of 64 similar circuits. Two separate evaluations are accomplished over the eight cycles.

Pawn Evaluation

Each cycle examines a file for pawn formations. The friendly and enemy pawns are examined on the file in question and on the file to each side. Separate evaluations are obtained for passed pawns, isolated pawns, backward pawns and doubled pawns. These values are summed and accumulated over the eight files.

Ray Evaluation

On every square, there is a finite state machine. Its input is a ray state, the piece occupying the square, a bit from the pawn evaluator indicating that this square is behind enemy lines, a bit from the pawn evaluator indicating that this square is a piece outpost, and a bit from the king location ROM that this square is near the enemy king. The output of the finite state machine is an output ray state and a 4-bit evaluation. All 4-bit evaluations are summed in an adding tree to yield a 10-bit ray evaluation to be accumulated over the eight slow evaluator cycles. The input state to each finite state machine is selected by an 8-to-1 multiplexor connected to the output states of the square's nearest neighbours. The multiplexor is selected by the phase of the eight evaluator cycles. Thus on the first phase, all of the finite state machines are connected north to south – evaluating rook and queen rays in that direction. The next cycle connects the finite state machines north-east to south-west. After the eight cycles, every ray on the chess board has been examined and accumulated.

The accumulated slow evalaution is added to the fast evaluation and made available to the micro-code. A condition code indicating that the slow evaluator is busy is available to the micro-code. Thus the micro-code can wait for the full static evaluation (fast plus slow) to become stable.

Transposition Table

The transposition table is an associative memory addressed by current board position. It is used as a cache of recently evaluated positions. It is also used to detect repeated positions. For every position remembered, two 16-bit data words are available to the micro-code. The use of these words will be described later.

The transposition table is implemented by a small fast (90ns cycle) micro-processor. The micro-processor manipulates the control leads of a standard commercial bus containing a megabyte of memory. The memory is viewed as 128K 8-byte positions. The positions are addressed by 16 of the 48 incremental hash code bits and WHITE-TO-MOVE. Thus the memory is divided into positions with white to move and positions with black to move. Four of the eight content bytes are used to hold the two 16-bit data words used by the micro-code.The remaining four content bytes hold the remaining 32 hash-code bits. The extra hash code is used to resolve collisions in the original hash addressing. There will be a false match approximately every 2^{32} probes. (Note that 2^{32} times the probe interval is about 30 hours.)

The micro-processor accepts and executes four instructions from the main micro-code. *Clear* writes zero in all of the memory. *Read* will compare the 32-bit hash code at the current hash address to the incremental hash code. If it matches, the two 16-bit data words are read and made available to the micro-code. *Write* will overwrite the current hash address position with 32-bits of incremental hash code and two 16-bit words made available from the micro-code. *Test* is used for diagnostics and not during normal operation.

The memory speed is approximately a microsecond per 16-bit word. Thus a *read* operation takes 1 µs for a typical miss and 4 µs for a hit. A *write* operation takes 4 µs. The *clear* operation takes a half-second.

The Micro-Code

The micro-code is implemented on a 1K × 64-bit horizontal micro-processor. Its main function is to perform an alpha-beta search using the move generator, evaluation functions and transposition table. The clock for the micro-code runs in two phases. The first phase is of constant duration and selects the timing of the second phase. Thus each micro-instruction selects its own execution time. The shortest instruction (a simple jump on a condition) takes 100ns. The longest instruction (maximum delay through the move generator) takes 375ns.

The micro-code simultaneously controls the gating on four separate data buses – the 16-bit board address bus, the 5-bit piece type bus and the 10-bit micro-instruction counter bus. Each of the buses has a 256 word read-write stack memory. All of the stacks are addressed by a 6-bit increment/decrement counter that points at the base of a four word stackframe. This stack pointer also addresses the stack of disable bits in the move generator. A stack offset of −8 to +7 comes from each micro-instruction. Thus a micro-instruction can access any word in the last two stack frames, the current stack frame or the next stack frame. (This is just convention since there is nothing magical about the current frame.)

The Value Bus

The value bus is used to perform the minimal calculations necessary to implement the alpha-beta sarch needed in chess. The value bus gates 16-bit values to and from the stack, transposition table, evaluators and the LSI-11. The only operations available on these are add and compare. For diagnostic purposes, there are also connections on the value bus to and from the other buses.

Eight of the value bus lines can be sensed as condition codes in the micro-code. Commands from the LSI-11 are decoded by a binary search on these value bus lines. Flags stored in the transposition table are also tested with these condition codes.

The Board Address and Piece Type Buses

The board address and piece type buses are used to read and write the board memories in the move generator, incremental evaluator and slow evaluator. The stacks on these buses hold the altered contents of the chess board after a move has been made so that the state of the chess board can be restored.

The Micro-instruction Counter Bus

The micro-instruction bus is used to gate micro-instruction jump addresses. The stack is used to hold recursive return addresses.

The Algorithm

The micro-code accepts and executes nearly fifty comands from the LSI-11. Most are diagnostic in nature. Only one command is important from a chess point of view – execute a full width alpha-beta search to a given depth. The approximate algorithm used is given in an Algol-like language below.

```
search($\alpha, \beta$, n)
{ if (no_moves())
  { if (in_check())
      return (MATE);
    else
      return (DRAW);
  }
  if (!in_check())
    n = n−1;
  while (next_move())
  { make move();
    if (n > 0)
      v = −search($-\beta, -\alpha$, n);
    else
      v = −quies($-\beta, -\alpha$);
    restore_move();
    if (v > $\alpha$)
    { if (v $\geqq \beta$)
        return ($\beta$);
      $\alpha$ = v;
    }
  }
  return ($\alpha$);
}

quies ($\alpha, \beta$)
{ if (in_check())
    return (search($\alpha, \beta$, 0);
  if (fast_eval()−MARGIN $\geqq \beta$)
    return ($\beta$);
  v = slow_eval();
  if (v > $\alpha$)
  { if (v $\geqq \beta$)
      return ($\beta$);
    $\alpha$ = v;
  }
  while (next_capture())
  { make_move();
    v = −quies($-\beta, -\alpha$);
    restore_move();
    if (v > $\alpha$)
    { if (v $\geqq \beta$)
        return ($\beta$);
      $\alpha$ = v;
    }
  }
  return ($\alpha$);
}
```

Search will perform an *n*-ply full width alpha-beta search. Getting out of check does not count as a ply. At each leaf of the full width search, a quiescence search is performed. The quiescence search examines captures to an arbitrary depth and also all moves are examined to get out of check.

Not shown in the C code is the transposition table. On every full width node, but not at quiescence nodes, the transposition table is consulted. If it matches the current position, some processing is performed as described below. If the processing does not cause the node to be exited, then the current position is written in the transposition table entry along with an ACTIVE flag. This is a warning to descendant nodes that a subsequent match should be treated as a repeat of the position (DRAW). When processing of this node is complete, one of three results is stored in the transposition table entry: (1) a beta cutoff and the beta value; (2) an alpha cutoff and the alpha value; (3) a true value and that value. Along with the type of table entry and value is stored the depth to which this entry is valid.

Now it is possible to describe the processing done when entering a node after a match in the transposition table. First if the ACTIVE flag is set the node is immediately exited with a value of DRAW. If the depth in the node is not as great as the desired depth of search, then normal processing continues. If this node was an alpha cutoff, that means the value of the position is less than or equal to the stored alpha value. This is implemented by setting the current beta parameter to the stored alpha value plus one. This is done only if it improves (lowers) the current beta. This reduction of beta can cause an alpha cutoff on the current node. This is checked with a possible early exit from the search. Next, if the stored value is a beta cutoff, then the stored beta value (minus one) can replace the current alpha value. Likewise, this is only done if it improves (increases) alpha and when done it can cause a beta cutoff. Lastly, if the stored table entry is an exact value, that is returned.

A quiescent node that has a fast evaluation beta cutoff takes about 3 μs. The average execution time for a node is between 5 μs and 10 μs depending on position.

Construction

The chess hardware is built on ten large wire-wrap boards. The move generator is made out of four identical boards. Each move generator board implements a 4 × 4 quarter of the chess board. Each part is constructed as if if were the south-west quarter of the chess board. Addresses to the other boards are altered to acheive the desired rotations and reflections. The colours of the pieces along with WHITE-TO-MOVE are inverted to the north-east and north-west boards so that pawns will move in the right directions. The evaluator is also made from four identical boards each implementing a 4 × 4 section. The microcode, LSI-11 interface, and the second level of the move generator priority tree all fit on a board. The incremental evaluator, slow evaluation controller, slow evaluation accumulator and transposition table microprocessor fit on the last board.

Most of the components are low power Schottky (LS) logic. This logic family is slightly slower than Schottky (S) logic, but draws much less power, thus reducing power supplies and cooling. The complete machine (LSI-11, disk, transposition memory, chess machine and power supplies) fits in a box 46cm × 48cm by 71cm tall. It weighs about 60kg. It is portable, but one has to be dedicated to take it anywhere.