A Shogi Processor with a Field Programmable Gate Array

Youhei Hori, Minenobu Seki, Reijer Grimbergen†, Tsutomu Maruyama and Tsutomu Hoshino

Institute of Engineering Mechanics and Systems, University of Tsukuba,
1-1-1, Ten-ou-dai, Tsukuba, Ibaraki, 305-8573 Japan
e-mail: hori@darwin.esys.tsukuba.ac.jp

† Electrotechnical Laboratory,
1-1-4, Umezono, Tsukuba, Ibaraki, 305-8568 Japan
e-mail: grimberg@etl.go.jp

Abstract. In this paper we describe the architecture of a shogi processor based on reconfigurable hardware. For our implementation, we have used Field-Programmable Gate Arrays (FPGAs), which can be reconfigured dynamically by downloading configuration data from host computers. Because of this reconfiguration flexibility, it is possible to implement and evaluate new algorithms quickly and to make small subsystems (of very low cost) that can be used on demand. For shogi these two features are especially important, as there are no stable subsystems that can be ported to special purpose hardware. Also, in shogi different modules are needed for different stages of the game. To test the feasibility of using FPGAs for shogi, we have implemented two modules that are general for all strong shogi programs on one off-the-shelf PCI board with one FPGA. The piece cover module on an FPGA is 62 times faster than the software module, while the module for finding mate on an FPGA is 9 times faster than the software module.
Keywords: Reconfigurable hardware, FPGA, Shogi.

1 Introduction

In chess, Deep Blue demonstrated that a large system with special purpose hardware can beat the human world champion [9]. Deep Blue's performance is often credited to its hardware power, which is indeed impressive. However, the number of positions that needs to be searched for strong chess play is too large to rely on hardware speed alone [5]. Computation power is important, but the key to the success of Deep Blue is the combination of hardware speed and sophisticated algorithms for tree search, position evaluation, move generation and so on. It was only possible to make special hardware for chess because there had been a long history of research into chess algorithms and there was consensus about which algorithms were most effective.

This consensus does not necessarily exist in other games. An example is *shogi*, a Japanese game that is similar to chess. Even though the objective of shogi is the same as for chess (capture of the king of the opponent), there are some important differences between the games. There are more pieces in shogi than in chess (40), there are different pieces (no queen, but golds and silvers) and the board is slightly bigger than for chess (9x9 instead of 8x8). The most important difference between chess and shogi is the possibility of re-using captured pieces in shogi. Pieces that have been captured from the opponent are put next to the board at the side of the player that captured them (these pieces are called *pieces in hand*). If it is a player's turn to move, either a move with a piece on the board can be played, or one of the pieces that was previously captured can be put back on an empty square of the board (this type of move is called a *drop*). As a result of this rule, the average number of legal moves in shogi is much larger than in chess. Therefore, using the same algorithms as in chess does not lead to programs that are strong enough to compete with human experts.

Modifications of chess algorithms and even completely new search algorithms have been proposed for shogi with mixed results. Descriptions of two of the best programs can be found in [6][14]. So far, there has been no attempt at making specialized hardware for shogi. One of the reasons for this is that there is no sponsor like IBM to fund such a costly project. However, the second problem is that there is no consensus about the algorithms that work best for shogi. Hardware based on fixed chips like those used in DEEP BLUE can not be used for shogi, because the chips will already be outdated by the time they have been manufactured. A third problem of the DEEP BLUE approach is that it is very hard for others to improve upon the state-of-the-art. The DEEP BLUE chip is not available to others for further improvements.

We think that shogi will benefit from the use of high speed hardware. In this paper, we propose a solution to the problems of outdated and inaccessible hardware. We describe how a shogi program can be implemented on reconfigurable hardware devices called *Field-Programmable Gate Arrays* (FPGAs). FPGAs are relatively cheap, flexible and have a short turn-around-time as it takes only a couple of hours to reconfigure an FPGA.

This paper is organized as follows. Section 2 introduces the features of systems with reconfigurable hardware. Section 3 describes the shogi program which we are currently implementing on FPGAs. Section 4 gives the architecture of the shogi processor. Section 5 describes the implementation issues of the shogi processor. In Section 6 and Section 7, details and results for two modules of the shogi processor are given. Finally, conclusions are given in Section 8.

2 High Speed Computation with Reconfigurable Hardware

The basic structure of a shogi program is the same as that of a chess program, and most time for finding the next move is spent on search. Therefore, improvements of hardware speed have the same impact on shogi programs as on chess programs and we expect considerable improvements of the playing strength of shogi programs by using special purpose hardware.

However, typical special purpose hardware like ASIC takes several months to be manufactured, after which a month or so is needed for modification. Such a long system development time is undesirable in shogi, as most of the important modules are still being revised often. Therefore, a shogi chip would be out of date at the time is can be used. A possible solution is to produce multiple chips, replacing only the modules that have been revised. A problem with this approach is that the circuit for a shogi program requires wide memory bandwidth and a high data transfer rate. When a shogi program is implemented on several chips, data has to be transferred between these chips. However, the number of I/O pins on the chips is smaller than the required data width and the operation speed of I/O pins is much slower than that of the internal processor. For high speed computation, it is best to have all modules of a shogi program on a single chip.

We think that hardware systems that are based on reconfigurable devices are a more promising approach to speed up shogi programs. In our research, we have focused on *Field Programmable Gate Arrays (FPGAs)*, which are very popular reconfigurable devices. A detailed hardware description of an FPGA is outside the scope of this paper, but more information about FPGAs can be found in the proceedings of several international conferences on FPGA-based systems that have been held [16][17][18]. Here we limit ourselves to the way an FPGA can be used for implementing a shogi program.

An FPGA chip consists of logic cells, internal memory and the connections between these cells and memory [15][1]. The function of the logic cells and the connections between cells and memory can be changed by downloading configuration data to the chip. The configuration data is generated by compiling a program written in a hardware description language. Compiling programs from the hardware description language takes several hours, while the actual reconfiguration of the FPGAs takes less than 100 msec in general.

The reconfigurability of FPGAs makes it possible to modify circuits for new algorithms relatively quickly. Minor modification of the circuits on the FPGAs can be finished and tested on the hardware in several hours. The size of FPGAs as well as the speed has improved drastically in the last couple of years and we expect that it will soon be possible to have a complete shogi program running on a single FPGA chip.

The flexibility of the FPGAs also makes it possible to realize a small but high performance shogi processor by removing unnecessary modules and reconfiguring required modules. This is important for shogi, as the behavior of shogi programs depends on the stage of the game. In the opening, the system has to perform efficient pattern matching between the current position and positions in opening libraries. In the middle game, it is necessary to search the game tree efficiently to find the best move. In the endgame, attack and defense of the king is vital, and special circuits to find mate are required. Also, different parameters and different evaluation functions may be required depending on the strategies that have been employed in the game.

Of course, FPGAs will not be as fast as the specialized hardware used by DEEP BLUE, but FPGAs are relatively cheap and flexible. Once there is consensus about the most effective software and hardware for shogi, it is still possible to produce a special purpose chip for further speed improvements.

3 The SPEAR Shogi Program

As said, the basic structure of a shogi program is similar to a chess program. The next move to play is determined by exploring a mini-max tree [13] with iterative alpha-beta search. Other common techniques taken from chess programs are transposition tables, quiescence search [2], principal variation search [10] and the history heuristic [11]. The main difference between programs for chess and shogi is that in shogi programs considerable effort is spent on dealing with the high number of legal moves, which on average is about 80 [7]. This is more than twice the average number of legal moves for chess, which is about 35[8].

Therefore, to search deeply it is important to discard moves early, often without any search. Most strong shogi programs make decisions about which move to play by the following steps: (1) update the position, (2) generate all legal moves, (3) evaluate the position after each move, (4) discard bad moves, thereby keeping the plausible ones, (5) search plausible moves. This procedure has the disadvantage that moves that are obviously bad still need to be evaluated. Evaluation is expensive, so this extra evaluation is not efficient.

SPEAR is a shogi program that aims at generating plausible moves more efficiently by using the evaluation after the previous move to generate moves [3][4]. During evaluation, information about *game conditions* like piece shape, castle shape, king danger and so on are collected. Plausible moves are then generated based on the current position and the game conditions.

In this section, we describe the data structures that are being used in SPEAR and how SPEAR finds the best next move.

3.1 Data Structures

The data structures in SPEAR are divided into data structures for positional data and data structures for the game conditions. Here are the data structures for positional data:

- `Board data`: information about which piece is on which square.
- `Piece cover table`: information about which piece can move to which square.
- `Piece info table`: information about which side the piece belongs to, where it is located and if the piece is promoted or not.

Here are the data structures for the *game conditions*:

- `Weak piece table`: information about which piece is not active, undefended, threatened by opponent pieces and so on.
- `Strong piece table`: information about which piece is active, able to promote, attacking opponent pieces and so on.

- `Piece square table`: the relative value of the pieces on each square.
- `King danger`: the evaluated value of king danger.
- `Soma value`: the expected profit of capturing a piece on a square.

3.2 Procedure for Deciding Upon the Next Move

In the program, the following steps are repeatedly executed to decide which move to play next: (1) update the position, (2) evaluate the position and update game conditions, (3) generate plausible moves based on the game conditions, (4) search plausible moves.

Figure 1 shows the block diagram of this program. Each module directly corresponds to one of the stages above. We will now give a more detailed explanation for each module.

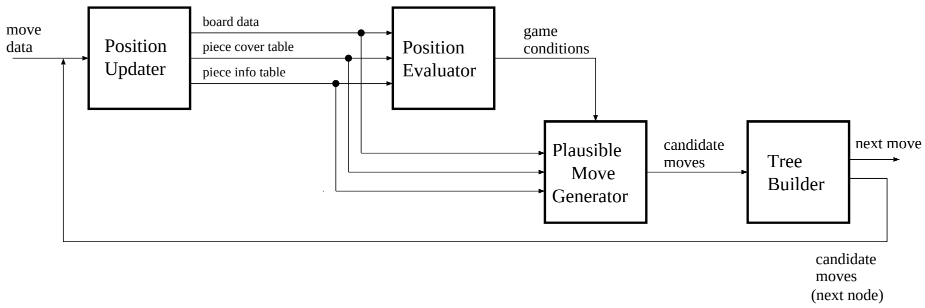

Fig. 1. A block diagram of the SPEAR program

Position Updater Based on the move data, this module updates `board data`, `piece cover table` and `piece info table`. This data is then used for position evaluation and plausible move generation.

Position Evaluator This module evaluates the position and updates the game conditions. The evaluation module has many sub-modules such as *piece shape evaluator*, *castle evaluator*, *king position evaluator* and so on. Each submodule evaluates the position and adds the evaluation to the game conditions. For example, *piece shape evaluator* evaluates the weakness or strength of a piece formation, and this evaluation is added to `weak piece table`, `strong piece table` and `piece square table`.

Plausible Move Generator This module generates plausible moves based on the game conditions. Using the game conditions (as well as board data, piece cover table and piece info table), only promising looking moves are generated. A value is given to each plausible move, and the best N moves are selected and added to the game tree (in the latest tournament version of SPEAR N was set to 20).

Tree Builder This module adds the remaining plausible moves to the game tree. Game tree search in shogi is similar to the search in chess programs.

4 Architecture of the Shogi Processor

As pointed out earlier, for efficient hardware implementation a single chip is best. Since a complete shogi program does not fit on a single FPGA chip yet, we must reduce the size of the circuit by reconfiguring the chip for different stages of the game and use different configurations for different strategies. In this section we discuss the issues in designing a shogi processor. First, we will explain the circuits for each game stage. Second, we will give the architecture for the middle game and endgame circuit for which we have made a partial implementation.

4.1 Circuits for Different Game Stages

An FPGA chip can be reconfigured in about 100ms, so deleting obsolete modules and reconfiguring other modules for each stage of the game is a viable option for reducing the size of the program. The following observations for the circuits in different game stages are not specific to SPEAR, but apply to most shogi programs.

Opening Circuit Moves in the opening are usually generated by an *opening book*, so high speed matching between the current position and the positions in the opening book is required. However, the opening book is typically too large to fit in memory, so data has to be loaded from external memory. The necessary data transfer and I/O operations will slow down the matching with the opening book.

By using an FPGA, we can store the opening library on the processor because this library (which will occupy a large part of the FPGA memory), can be overwritten if necessary. It is expected that this internal processing will speed up the matching with the opening book considerably.

Middle Game Circuit This circuit finds the next move by searching the game tree. Most time during this stage is spent on position evaluation and candidate move generation. With an FPGA, we can expect high speed computation by processing different sub-modules in parallel and in pipeline. For example, the evaluation of a position has many different features like piece activity, piece formation shape, capturing profit and so on. Some of these features are independent and can be processed in parallel. Moreover, we will later explain how different features can be processed in pipeline, leading to further speedups.

Move generation can also be performed in parallel and in pipeline. The plausible moves have many different categories like attack moves, defense moves, captures, promotions and so on. These categories can be generated in parallel and each move can be generated by pipeline processing.

Endgame Circuit In the endgame, the weights of the evaluation function features change dramatically. Mating the king becomes the most important feature and material gain is only a secondary consideration. One of the modules that most strong shogi programs use is a *tsume shogi solver* to find mate (more details will be given in Section 7). Because of the reconfigurability of an FPGA, this module can be prepared on demand.

Library For/Against Strategies Using a library to deal with different strategies is a good way to guide the search to promising parts of the game tree. For example, different castles have different strengths and weaknesses, so an attack should be aimed at the weak point of the castle. There are many different strategies and castle formations in shogi, so such a library would be too large to keep in memory on the chip. With an FPGA, we can load and configure only the required libraries.

4.2 Data Structures

For a hardware implementation, we need the following data structures. Even though these data structures have been taken from SPEAR, similar data structures are used in most shogi programs.

- The `move data` contains information about a move. It shows which player played the move, if the moving piece promoted, which square the piece came from and which square the piece moved to.
- For each piece, there are 10 bits of `Piece data`. The bit assignment, shown in Figure 2, has the following meaning.
  - The 3 bits of `piece_type` are used to store the kind of piece (King, Rook, Bishop and so on). In shogi there are 8 different pieces, so 3 bits are needed.
  - The `promotion` bit is set if the piece is promoted.
  - The `ownership` bit is set if the piece is owned by black.
  - The 5 bits of `index` distinguish pieces of the same type. For example, there are 18 pawns in shogi, which can be uniquely identified by the index. The `piece_type` and `index` are used to find the piece in the piece cover table and the piece info table.

  The piece data in Figure 2 means that the piece is Pawn_{12}, promoted, and owned by white.
- The `board data` is a set of piece data and contains the location of pieces. For the complete description of the board, 810 bits (=81 squares$\times$10 bits of piece data) are needed.
- The `piece cover table` stores which piece can move to which square. One square has 40 bits of piece cover data where each bit corresponds to one of 40 pieces (40 is the total number of pieces in shogi). If a piece can move to a square, the corresponding bit is *1*. If not, the corresponding bit is *0*. In Figure 3, the piece cover data shows that King_1, Rook_1, Bishop_1, Gold_1 and Gold_2 can move to square$(5, 8)$.

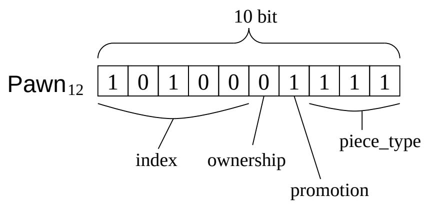

Fig. 2. Bit assignment of the piece data

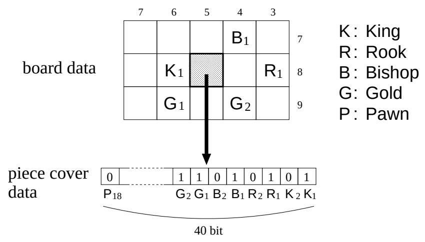

Fig. 3. The piece cover data of the square$(5, 8)$

4.3 Architecture of the Middle Game Circuit

Before implementing a shogi program in hardware it is important to think about the architecture of the processor. Not all applications are speeded up by the use of an FPGA. If an application has no possibilities for parallel and pipeline processing, the performance will be disappointing, sometimes even slower than the corresponding software application.

With this in mind, we designed a middle game circuit for shogi that is highly parallelized and designed to perform pipeline processing. The architecture directly corresponds to the structure of the software program given in Section 3. Therefore, it has four major modules that we will now describe in more detail.

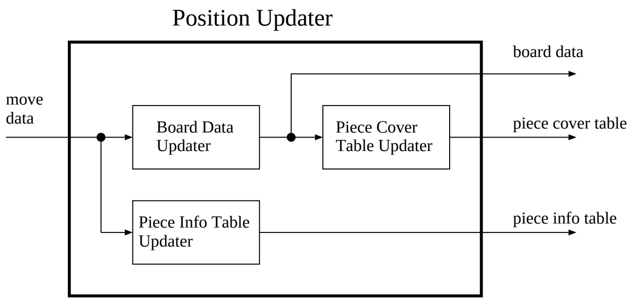

Fig. 4. A block diagram of `Position Updater`

Position Updater In `Position Updater`, board data, piece cover table and piece info table are updated for each move. Figure 4 shows the structure of the circuit for `Position Updater`. The modules `board data updater` and `piece info table updater` can be processed in parallel, and the modules `board data updater` and `piece cover table updater` can be processed in pipeline.

Position Evaluator A block diagram of `Position Evaluator` is shown in the upper half of Figure 5. `Position Evaluator` has many sub-modules. These modules can be divided into the following three groups.

1. Piece square table evaluator.
2. Modules for evaluating pins, discovered attacks, piece mobility and king distance.
3. Modules for evaluating the soma value, piece shape, promotion threats, castle shape and king position.

These three groups can be processed in pipeline, while the modules in each group can be processed in parallel. A position is evaluated by these modules, and the results of this evaluation are stored in the data structures for the game conditions such as *weak piece table*, *strong piece table* and *king in danger*.

Plausible Move Generator Each position is evaluated before generating plausible moves, and only promising moves are generated. The hardware modules shown in the lower half of the Figure 5 are used for this process. All these modules can be processed in parallel.

Game Tree Builder The game tree is stored in the memory of the FPGA. A node is represented by the move data and a relative address from its parent. This module builds nodes by adding this address to the move data, and then stores the node information in memory.

4.4 Architecture of Endgame Circuit

As we pointed out earlier, the weights of the features of the evaluation function change in the endgame. Despite these changes, the endgame circuit still should perform a tree search, so the basic structure is similar to that of the circuit used for the middle game. There is one exception, as there is a need for a circuit performing specialized mating search. This module is a vital part of strong shogi programs. Details of a tsume shogi circuit will be given later in Section 7.

5 Implementation of the Shogi Processor

Unfortunately, reconfigurable hardware is not a magic box where a shogi program in a common programming language like C can be entered, a button can be pushed and the FPGA chip rolls out at the other end. We already pointed at the importance of good hardware design, taking advantage of the parallel and pipeline processing abilities of the FPGA chip. In the next two sections we will see how the special features of the FPGA make it necessary to rewrite some of the basic building blocks of our shogi program.

As a result of these FPGA specific design issues, implementing a full shogi processor is a time consuming task. However, if we wait until the shogi processor is finished, we miss an opportunity to take advantage of another appealing

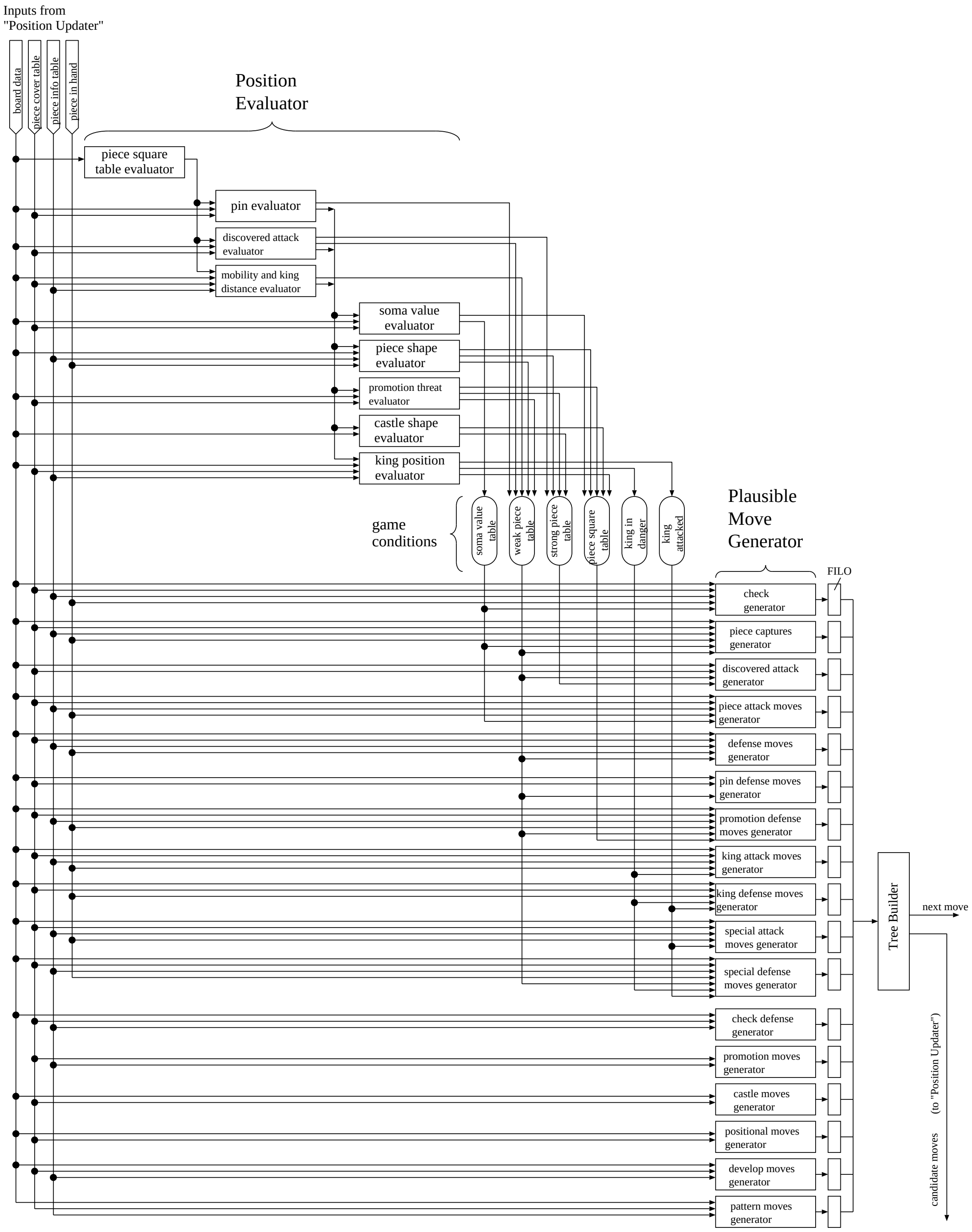

Fig. 5. The structure of the shogi processor.

feature of FPGAs: they are relatively cheap and can be used by many different researchers. By combining the efforts of different people, we can reduce the work on the full shogi processor and we can also take advantage of improvements by others on modules that have already been implemented.

As a first step, we have made two partial implementations to get an indication of how a shogi program can be speeded up by the use of FPGAs. There are of course many candidates for partial implementation, but we chose to implement two modules that are general for all strong shogi programs: a circuit for computing the *piece cover table* and a circuit for mating search, the *tsume shogi solver*. As these modules are stable, there is probably no need for important modifications later.

In Section 6, we describe the *Piece Cover Circuit* in detail. From this description, it will be clear that the method for acquiring piece cover data used in this circuit is general and can be used in most other modules as well.

The *Tsume Shogi Circuit*, described in Section 7 is not as general as the *Piece Cover Circuit*, but the advantage of this module is that it is completely independent from the other modules. Move generation is different (only checks and defenses against checks), position evaluation is different (estimate of distance to mate) and even the tree search is different (proof number search instead of alpha-beta search). Therefore, this module is a good test of the stand-alone abilities of an FPGA.

6 Piece Cover Circuit

As a first step in implementing our shogi processor, we have designed the *Piece Cover Circuit* that will be described in detail below. As pointed out earlier, it is necessary to use parallelism and pipeline computing for good speed-up results. The implementation on an FPGA is therefore very different from the software implementation of a piece cover table.

6.1 Obtaining the Piece Cover Data

Parallelization of the computation of the piece cover table is achieved by loading 9 squares (= one rank) of board data simultaneously at each clock cycle. The whole board can therefore be scanned in 9 clock cycles (as the dimension of a shogi board is 9x9). In our method, we use two scans of the board: a top-down scan and a bottom-up scan. To balance the size of the circuits the top-down scan and the bottom-up scan, each give piece cover data of four of the eight adjacent squares. An example of the top-down scan is given in Figure 6(a), while the bottom-up scan is given in Figure 6(b)

Two separate scans are necessary, because the problem with a single scan is that piece cover data for *long range pieces* (rook, bishop and lance) can not be obtained. These pieces can not only move to an adjacent square, but also to squares that are further away. An example of how the piece cover data for the bishop is obtained by a top-down scan and a bottom-up scan is given in

Figure 7. Let's assume that the fifth rank is currently scanned. The board data of this rank is loaded, but there will be no piece cover data as there is no piece on this rank. We will get the same result after scanning the sixth rank, as there is still no piece found. Only after loading the board data of the seventh rank, the bishop is found and the piece cover data of the squares $(2, 8)$ and $(4, 8)$ is obtained. The information that there is a bishop on (3,7) is stored in a register. The information in this register can be used at the next clock cycle, as the board data for the eight rank is loaded and the piece covering of square $(1, 9)$ and $(5, 9)$ by the bishop is obtained. The top-down scan is now finished, and the bottom-up scan is performed next. The piece covering of the squares in Figure 7(b) can now be obtained in the same way as for the top-down scan.

After both the top-down and the bottom-up scan have finished, the piece cover table is simply the logical OR of the piece data obtained from the top-down scan and the bottom-up scan.

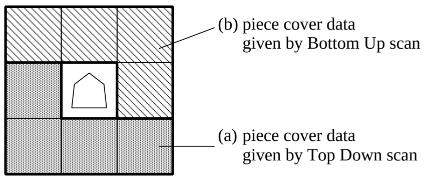

Fig. 6. The piece cover data given by each scan

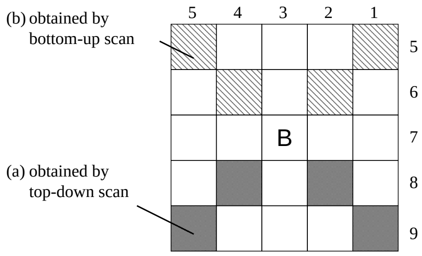

Fig. 7. An example of obtaining the piece covering of longe range pieces.

Getting the piece cover data of the rook is a little more difficult, as the horizontal piece cover data of the rook is opposite to the scan direction. In this case an extra left-to-right scan and a right-to-left scan is necessary on the rank where the rook is positioned. The board data of this rank is stored in a register and the two horizontal scans are performed on this register.

Here is a summary of the computation of the piece cover table:

1. Carry out the top-down scan and the computation of the rook rank covering in parallel:
   - A top-down scan of the board is executed to compute piece cover data shown in Figure 7(b). The top-down piece cover data is stored in memory on the FPGA.
   - Compute the rook rank cover data by a left-to-right and a right-to-left scan. The rook rank cover data is stored to a register.

2. Carry out a bottom-up scan to obtain the piece cover data shown in Figure 7(a). The bottom-up piece cover data is stored in memory on the FPGA.
3. Compute the logical OR of the top-down piece cover data, the bottom-up piece cover data and the rook rank cover data.

6.2 Structure of the Hardware

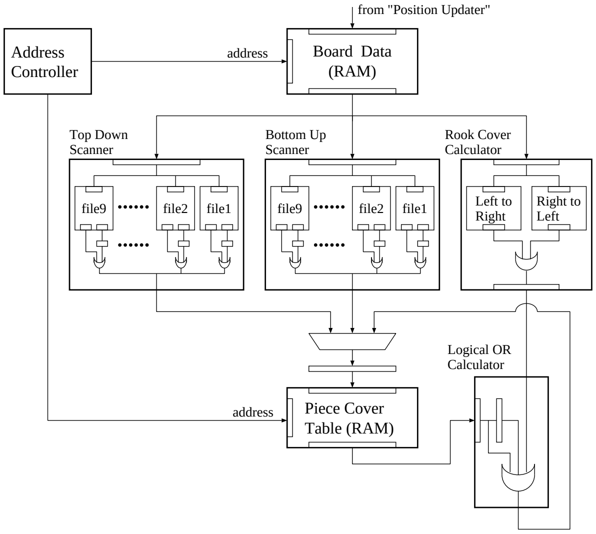

Fig. 8. A block diagram of the circuit for computing the piece cover data.

Figure 8 shows the design of the circuit. The circuit has the following modules:

Board Data One piece is represented by 10 bits and one rank has 9 squares, so the board data of one rank is 90 bits. The board has 9 ranks, therefore the complete description of the board requires 810 bits. This module is a 9×90 bit size RAM in which the board is stored. 90 bits of board data are loaded at one clock cycle.

Top Down Scanner/Bottom Up Scanner The `Top Down Scanner` does a top-down scan. The input of this module is one 9×10 bit data block from the `Board Data`. This data block is split into 10 bit blocks that are processed in parallel by the sub-modules *file1*, *file2*, ..., *file9*. Each of the sub-modules checks the data of one square and gives the piece cover data on that square as output. This data is stored in `Piece Cover Data`. One rank of piece cover data is 360 ($= 9 \times 40$) bits wide.

The Bottom Up Scanner does a bottom-up scan. The construction of the Bottom Up Scanner is similar to that of Top Down Scanner. Both the parallel processing and the pipeline processing is performed in the same way.

It takes eight clock cycles to compute one rank of piece cover data. Obtaining the piece cover data of the whole board takes 25 clock cycles. For this it is necessary to use pipeline processing, where the computation of the piece cover data of one rank does not need to finish before computing the piece cover data of the next rank. This is shown in Figure 9.

Rook Cover Calculator The Rook Cover Calculator performs a left-to-right scan and a right-to-left scan to compute the rook rank piece cover data. The piece cover data obtained by these scans is stored to registers in the module.

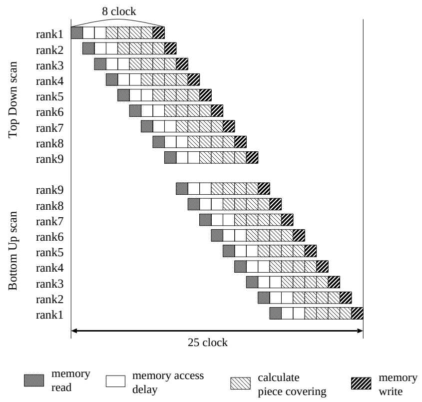

Fig. 9. Pipeline processing of the piece cover table computation

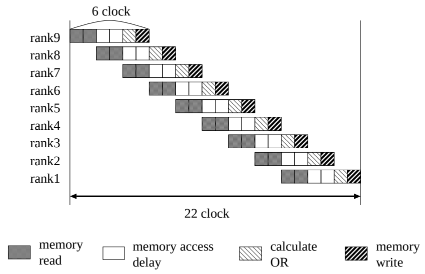

Fig. 10. Pipeline processing of the calculation of the logical OR

Logical OR Calculator This module calculates the logical OR of the top-down piece cover data, the bottom-up piece cover data and the rook rank cover data. It takes 6 clock cycles to calculate a logical OR and write it to RAM. The computation of the final piece cover table finishes in 22 clock cycles by pipeline processing as shown in Figure 10.

Piece Cover Table One rank of piece cover data requires 360 bits because each square has 40 bits of piece cover data. Thus 3240 (= 9 ranks × 360) bits are needed to represent the whole board. The top-down piece cover data, the bottom-up piece cover data and the logical OR each require 9×360 bits, therefore the Piece Cover Table can be stored in 27×360 bits of RAM. 360 bit of the piece cover table is stored to or loaded from RAM at one clock cycle.

6.3 Performance

To test the performance of the piece cover circuit, the computation time was compared to that of SPEAR. The result of this test is shown in Table 1.

Table 1. Piece cover computation time per position

	Frequency [MHz]	Computation time [μsec]	Performance ratio
SPEAR	400	74.89	1
FPGA	38.46	1.20	62

The piece cover table computation of SPEAR was performed on a 400MHz Pentium II. The test consisted of 100 shogi positions. The average computation time of the piece cover table in these positions was 74.89μsec.

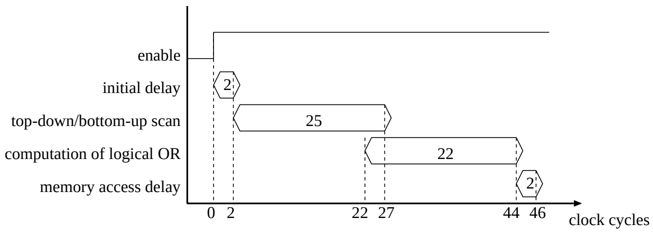

Fig. 11. A timing chart of the circuit for piece cover table

Figure 11 shows that in hardware the number of clock cycles required to compute the piece cover table is always 46. The frequency of the circuit was calculated with the simulator in MAX+PLUS II, which is a circuit design tool. MAX+PLUS II can find the critical path in the circuit because of the delay-predictable connections in an FPGA [1]. The longest delay was 26.0 ns and the maximum frequency of the circuit was 38.46 MHz. Therefore, the computation time of the cover table is

$$46 \times \frac{1}{38.46} = 1.20\mu\text{sec}.$$

Therefore, the computation speed of the circuit is 62 times faster than that of SPEAR.

Note that the scanning method described in this section is not limited to the piece cover circuit. It will be the same for most of the modules of the position evaluator and move generator. Therefore, the performance of the *Piece Cover Circuit* is a good indication of the overall performance that can be expected of our shogi circuit.

7 The Tsume Shogi Circuit

In this section we describe the circuit for finding mate in the endgame. Building a program for solving mating problems in shogi (the *tsume shogi solver*) has been an independent research area in shogi for years with impressive results. The most important difference between mating problems in chess and tsume shogi is that in tsume shogi every move by the attacker has to be a check. As a result, the search space of tsume shogi is much smaller than that of normal shogi play, as the branching factor of the search tree is only about 5 [7][12]. Tsume shogi problems are used for endgame training or just as puzzles and there are tsume shogi problems that have solutions of several hundred ply. The longest tsume shogi problem has a solution of 1525 ply. The quest for solving this problem called *Microcosmos* inspired researchers for years until Masahiro Seo build a program that could solve *Microcosmos* in 1997. Tsume shogi is currently the only area where programs outperform the human experts.

A partial module for tsume shogi was our first test for using FPGAs in shogi. It seems clear that the use of FPGAs will have more impact in a search domain with a high branching factor, so it is interesting to compare the results for tsume shogi with the speed-ups given in the previous section. Even though the tsume shogi circuit was not completely finished, the performance of a complete tsume shogi circuit can be estimated by the intermediate results we have collected so far.

7.1 Structure of the Circuit

The tsume shogi circuit finds mate by processing the following three stage in pipeline:

1. Update position after a move.
2. Generate all moves, generate check mask and generate defense mask.
3. Distinguish check/defense moves from all moves, and store them in memory.

To implement this procedure, we designed the circuits `Position Updater`, `All Move Generator`, `Check Mask Generator`, `Defense Mask Generator` and `Check/Defense Move Generator`. The function of each circuit is as follows.

Position Updater This module updates the position. The board data is updated based on the move data.

All Move Generator The structure of `All Move Generator` is similar to the `Piece Cover Circuit` in Section 6. The piece cover table contains information about which piece can move to which square, so the data collected by this circuit is the same as the data that is needed to generate all legal moves.

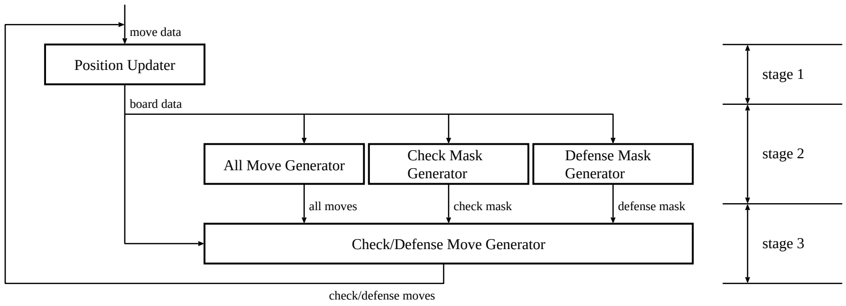

Fig. 12. A block diagram of the circuit for shogi mating problem

Check Mask Generator Generating a *check mask* makes it possible to discard the non-checking moves in an efficient way. This module scans the board twice (top-down and bottom-up) and checks if it is possible for a piece to check the king from a square. Figure 13 shows an example of a board position and the generated check mask. A white square means that there is a piece that can check the king by moving or dropping there. A black square means that no piece can check the king from that square.

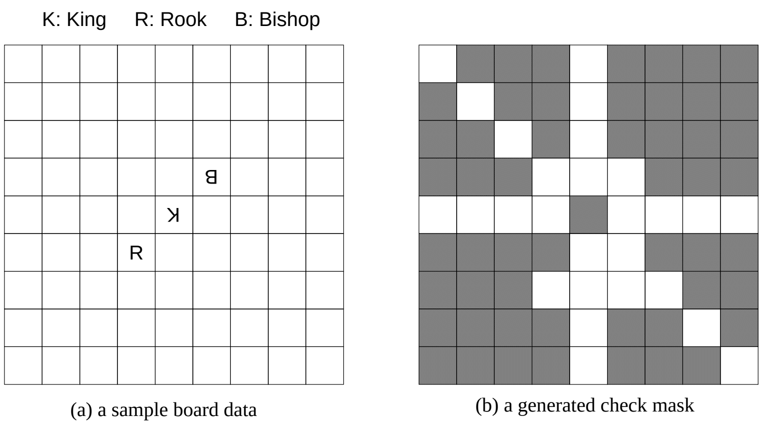

Fig. 13. An example of board data and the check mask.

Defense Mask Generator Generates a defense mask in the same way as a check mask. This module also scans the board twice and tests if there is a piece that can defend against the opponent check from a square.

Check/Defense Move Generator All legal moves are generated in `All Move Generator` and `Check/Defense Move Generator` distinguishes the check and defense moves from the other moves. This module generates check moves using all moves and the check mask, and defense moves using all moves and the defense mask.

7.2 Performance

Although we have not implemented the module `Defense Mask Generator` yet, we can estimate the performance of the full tsume shogi circuit by looking at the results of the partial implementation. Table 2 shows the frequency of the modules in the circuit. The frequency of each module was calculated with MAX+PLUSII.

Table 2. table: The frequency of each modules

module	frequency [MHz]
Position Updater	80.06
All Move Generator	54.87
Check Mask Generator	90.07
Check/Defense Move Generator	22.32

The performance of the whole tsume shogi circuit can now be estimated as follows:

- The `Defense Mask Generator` runs at almost the same frequency of `Check Mask Generator` because the structures of these two modules are similar. Thus, the frequency of the whole circuit does not change if `Defense Mask Generator` is added to the circuit.
- The required clock cycles do not change if `Defense Mask Generator` is added to the circuit because it runs in parallel to `Check Mask Generator`.

Therefore, the frequency of the whole circuit is 22.32 MHz even if the `Defense Mask Generator` is added to the circuit. The required number of clock cycles is 48 (obtained with MAX+PLUSII), thus the computation time is

$$48 \times \frac{1}{22.32} = 2.15\mu\text{sec}$$

.

We implemented a tsume shogi solver in C++ and compared its computation time for 100 tsume shogi problems to that of the circuit. The software ran on a Pentium II-700MHz.

The result in Table 3 shows that the computation time of the hardware is about 9 times faster than that of the software.

Table 3. The computation time of tsume shogi

	Frequency[MHz]	Computation time[μsec]	Performance ratio
Software	700	18.9	1
FPGA	22.32	2.15	8.8

In our current implementation, the operation speed of `Check/Defense Move Generator` is relatively slow. It is possible to improve the results by further pipelining the circuit, improving the operation speed to more than 54.87MHz. This is the frequency of `All Move Generator`, which is then the new bottle-neck of the tsume shogi circuit. If we can speed up `Check/Defense Move Generator` in this way, we can expect a speed-up of 23 instead of a speed-up of 8.8 as given in Table 3.

8 Conclusions

In this paper, we have described the architecture of a shogi processor based on reconfigurable hardware. Reconfigurable hardware promises significant speed-ups with a short development time. Also, reconfigurable hardware is relatively cheap, so a cooperative effort in the hardware design of a shogi processor is possible.

To evaluate the use of *Field Programmable Gate Arrays* (FPGAs) as the reconfigurable hardware for our shogi processor, we have implemented two modules of the shogi processor. The speed-up of the module *Piece Cover Circuit* was 62 times, while the speed-up of the *Tsume Shogi Circuit* was 9 times compared to an equivalent software program. We will continue our work on the processor by finishing the tsume shogi circuit and designing the other modules of the shogi processor that we have described in this paper.

FPGAs are a rather new technology and are still being improved fast. The speed gain is almost proportional to the hardware size and in the near future one PCI board with several FPGAs can be expected to be more than a thousand times faster than the current hardware.

Another area that will be improved is how the FPGA chips are being reconfigured. At the moment it is still necessary to write circuits for each module in a hardware description language. This requires more time than programming in a traditional programming language like C. There have been many proposals for generating hardware circuits automatically from software programs, but so far it has been difficult to generate efficient circuits. However, by preparing hardware libraries for efficient data management in shogi, we expect that it is possible to write most of the modules of a shogi processor in C without negative effects on the performance. Preparing these libraries is also a future work.

References

1. Altera Corporation, San Jose, CA, USA, *Embedded Programmable Logic Data Sheet*, August 1999, ver 2.02.
2. D. Beal. A Generalised Quiescence Search Algorithm. *Artificial Intelligence*, 43:85–98, 1990.
3. R. Grimbergen, "Candidate Relevance Analysis for Selective Search in Shogi", in *Advances in Computer Chess Conference*, Paderborn, Germany, 1999.
4. R. Grimbergen, "A Plausible Move Generator for Shogi Using Static Evaluation", in *Game Programming Workshop*, pp. 9-15, Kanagawa, Japan, 1999.
5. S.Hamilton and L.Garber, "Deep Blue's Hardware-Software Synergy", *IEEE Computer*, Oct., 1997, pp.29-35.
6. G.Kakinoki, "The Search Algorithm of the Shogi Program K3.0", In H.Matsubara, editor, *Computer Shogi Progress*, pp.1-23. Tokyo: Kyoritsu Shuppan Co, 1996. ISBN 4-320-02799-X. (in Japanese).
7. H.Matsubara and K.Handa, "Some properties of shogi as a game", *Proceedings of Artificial Intelligence*, 96(3):21-30, 1994. (in Japanese).
8. H.Matsubara, H.ida and R. Grimbergen, "Natural developments in game research", *ICCA Journal*, 19(2):103-112, June 1996.
9. Monty Newborn, "Kasparov versus Deep Blue: computer chess comes of age", Springer, 1997, ISBN 0-387-94820-1.
10. J. Pearl. *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison Wesley Publishing Company: Reading, Massachusetts, 1984. ISBN 0-201-05594-5.
11. J. Schaeffer. The History Heuristic and Alpha-Beta Search Enhancements in Practice. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 11(11):1203–1212, 1989.
12. M. Seo, "A tsume shogi solver using conspiracy numbers", in H. Matsubara, editor, *Computer Shogi Progress 2*, pp.1-21, Tokyo:Kyoritsu Shuppan Co, 1998, ISBN 4-320-02799-X. (in Japanese).
13. C.E.Shannon, "Programming a computer for playing chess," *Philosophical Magazine* 41(1950): 256-75.
14. H.Yamashita, "YSS: About its Datastructures and Algorithm", In H.Matsubara, editor, *Computer Shogi Progress 2*, pp.112-142. Tokyo: Kyoritsu Shupppan Co, 1998. ISBN 4-320-02799-X. (in Japanese).
15. Xilinx, Inc., San Jose, CA, USA, *The Programmable Logic Data Book*, 1999.
16. URL: http://www.fccm.org
17. URL: http://xputers.informatik.uni-kl.de/FPL/FPL99/index.html
18. URL: http://www.ece.cmu.edu/~fpga2000

This article was processed using the LaTeX macro package with LLNCS style