An FPGA-Based Processor for Shogi Mating Problems

Youhei Hori, Masashi Sonoyama and Tsutomu Maruyama

Institute of Engineering Mechanics and Systems, University of Tsukuba
1-1-1, Ten-nou-dai, Tsukuba, Ibaraki, 305-8573 Japan
E-mail: {hori, sonoyama, maruyama} @ darwin.esys.tsukuba.ac.jp

Abstract

After the success of DEEP BLUE *in computer chess,* shogi, *or Japanese chess is a next challenging target in artificial intelligence for game playing. The complexity and huge search space of shogi have been motivating researchers to make shogi programs, but none of them is competent enough to play against human experts. To improve competence of shogi programs, it is a promising approach to develop dedicated hardware systems. However, inflexible architecture and lack of hardware resource have been the significant problems in hardware development. The flexibility and recent progress in gate size of FPGAs are expected to give solutions to the problems. As a first step to shogi hardware, we implemented modules to generate check and defense moves in* tsume-shogi, *or mating problems in shogi. With the latest FPGA, we successfully implemented all modules on a single chip and eliminated the bottleneck of memory bandwidth. In this paper, we describe a procedure for parallel move generation in tsume-shogi hardware and architecture of the modules implemented on an FPGA. Discussion about performance of the hardware is also made in the paper. The hardware is roughly estimated to work 10 – 50 times faster than software.*

1 Introduction

In chess, DEEP BLUE demonstrated that a large system with dedicated hardware could beat a human world champion [8]. After the success in computer chess, *shogi*, or Japanese chess is a next challenging target in artificial intelligence for game playing. Shogi is recognized as much more complicated game than chess from a computational point of view. So far, several studies have been made on the complexity of chess and shogi [1, 10, 13]. According to the latest report, the number of game positions reachable from the initial position in chess is estimated as 1.07×10^{54}, and that in shogi 10^{71} [10]. Another paper reported that the number of nodes in the game tree of chess and shogi are 10^{123} and 10^{226} respectively [6]. The complexity of shogi is mainly attributed to *reuse* of captured pieces. In shogi, pieces captured from the opponent can be put back onto the board (this type of move is called a *drop*), while they can not be reused in chess. Reuse of pieces causes considerably large number of moves and the maximum number of legal moves per position in shogi is known as 593 [9].

To date, lots of algorithms to deal with the huge search space have been proposed by researchers in several countries as well as in Japan [2, 3, 5, 11], but no shogi program is competent enough to beat human professional players. To improve play ability of shogi programs, constructing dedicated hardware is obviously an essential approach. To the best of authors' knowledge, however, there has been only one attempt at shogi hardware reported so far [4].

There are two significant problems with developing shogi hardware. One problem is that there is no consensus about algorithms for move generation, position evaluation, tree search and so on. Shogi algorithms are still under discussion and revised often. Therefore developing hardware with ASICs would be resultless because it would be soon outdated. Another problem is that shogi is so large an application that it requires a lot of hardware resource and wide memory bandwidth. Shogi is quite a complex game with huge search space and a program consequently has various kinds of modules. These modules should work in parallel to achieve high-speed computation, but in such highly parallelized processing, lack of hardware resource and narrow memory bandwidth are always bottlenecks to decline the performance of hardware.

With the flexibility and the recent progress in size of FPGAs, these problems are expected to be resolved. The flexibility of an FPGA enables us to keep up with improvement of algorithms. The latest FPGA has large hardware resource and wide-width memory, therefore a shogi program is expected to be implemented on a single chip.

As a first step to a shogi processor, we are currently implementing a circuit to solve *tsume-shogi* (mating problems in shogi). A tsume-shogi solver tries to find checkmate by exploring the game tree with an algorithm specialized for end-game. A tsume-shogi solver does not use database of mating procedure for typical end-game positions, while it was a great success in computer chess. In shogi, similar positions hardly appear in end-game because the number of pieces remaining on the board does not decrease due to drop moves, thus building database of mating procedure is impossible.

In this paper, we present solutions for problems in development of tsume-shogi hardware. Throughout the implementation of tsume-shogi processor, we show the feasibility of an FPGA for the large and complicated application. This paper is organized as follows. Section 2 describes features and rules of shogi and tsume-shogi. Some important features that cause shogi's complexity and huge search space are explained. Section 3 describes data structures and procedure for generating moves. It is given how the move generation in tsume-shogi is performed in parallel and in pipeline. Section 4 describes architecture of a module to generate piece cover data. An explanation is given in detail to show an example of parallel and pipeline architecture. Section 5 describes the performance of implemented modules. Frequency, usage of hardware resource and computation time of the tsume-shogi circuit is given in this section. Finally, section 6 summarizes the current status and gives a further direction of this study.

2 Shogi and Tsume-Shogi

2.1 Feature of Shogi

Shogi is a Japanese chess-like game. The object of shogi is to checkmate opponent's king, as is the same for chess. Figure 1 is the initial position of shogi described with *kanji* fonts (Chinese characters). To promote international discussion, the position in Figure 1 is sometimes described with chess-like fonts (Figure 2 [1]).

As already pointed out, the most important rule in shogi related to the large number of possible moves is that the captured pieces from the opponent can be put back onto the board again. In addition to the reuse of pieces, shogi has some more features that causes the huge search space. Here we summarize rules and features of shogi that are directly connected with the complexity of programming.

- **Reuse of Captured Pieces**
  The captured pieces are called *pieces in hand*. When it is a player's turn, he/she can freely choose to move a piece on the board or to drop a piece in hand on a vacant square. It is, however, prohibited to drop a pawn onto the file where another pawn already exists (prohibition of *double pawn*).

[1]Figure 1 and Figure 2 are described with a special package called OhTEX. For further explanation for Figure 2, see [7].

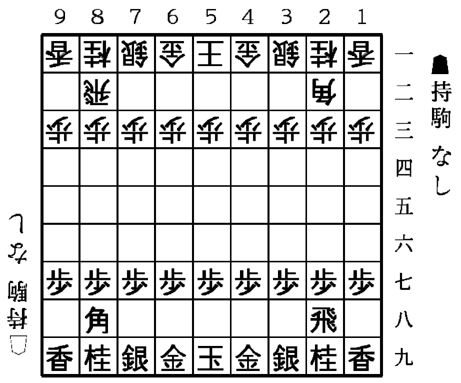
9 8 7 6 5 4 3 2 1
一 二 三 四 五 六 七 八 九
☗持駒なし
☖持駒なし
香 桂 銀 金 王 金 銀 桂 香
飛 角
歩 歩 歩 歩 歩 歩 歩 歩 歩
歩 歩 歩 歩 歩 歩 歩 歩 歩
角 飛
香 桂 銀 金 玉 金 銀 桂 香

Figure 1: The initial position of shogi in Japanese style.

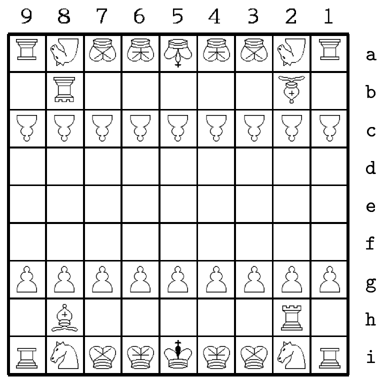

Figure 2: The initial position of shogi in western style.

- **Board Size**
  The size of shogi board is slightly larger than that of chess board. The size of shogi board is 9×9, while that of chess is 8×8.
- **Number and Kind of Pieces**
  The total number of pieces in shogi is 40, while that in chess is 32. Additionally, there are 8 kinds of pieces and 6 kinds out of them can promote in shogi, while there are 6 kinds of pieces and only pawn can promote in chess.
- **Promotion Zone**
  In shogi, promotion zone is large and the rule of promotion is complicated. Promotion zone for black is rank `a` to `c` in Fig. 2, and that for white is rank `g` to `i`. A player can freely choose to promote a piece or not when (1) the piece moves into the promotion zone, (2) the piece moves inside the promotion zone, (3) the piece moves from the promotion zone to the non-promotion zone.

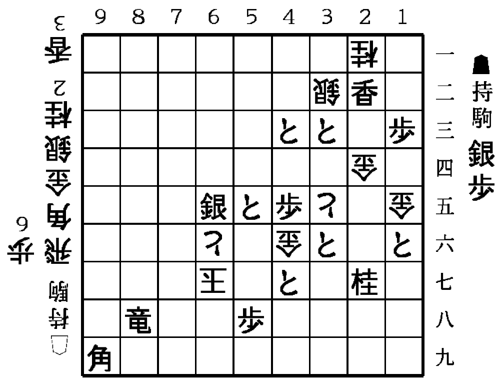
Figure 3: An example position extracted from a tsume-shogi problem with 117-ply solution.

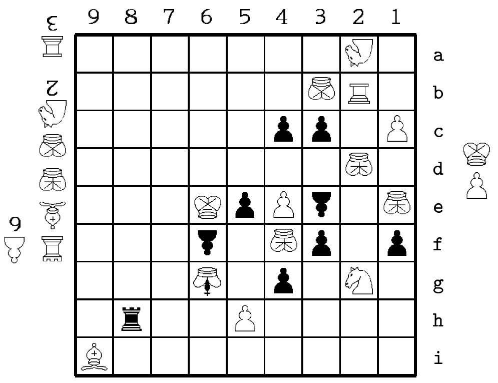
Figure 4: The same position as Fig. 3 in western style.

2.2 Feature of Tsume-Shogi

Tsume-shogi are checkmating problems in shogi. The object of the attacker is to checkmate opponent's king and of the defender is to prolong the mate as long as possible. Fundamental rules of tsume shogi such as mobility of pieces and re-use of pieces are the same as that in normal shogi. The most important difference between tsume-shogi and normal shogi is that in tsume-shogi each move by the attacker must be a check, and consequently a move by the defender must be one to get out of the mating threat. Although valid moves in tsume-shogi are limited by the rule, algorithms for move generation and tree search are still complicated because (1) the number of possible positions can be large due to drop moves and (2) tree search can reach a quite deep part of the game tree with a solution of more than a hundred ply. Figure 3 is an example position of tsume-shogi, which has a solution of 117-ply (the same position is described in western style in Figure 4). The number of possible moves in Figure 3 is 153 and the number of check moves is 13. Figure 3 indicates that high-speed move elimination as well as move generation is vital to reduce computation time and to perform further deepening.

2.3 Categories of Check and Defense Moves

To gain high parallelism, we categorized check moves into 3 groups according to the way of attack. Explanation of categories of check moves are given as follows.

- **Direct Check**
  When an attacker's piece on the board moves and the piece itself attacks the opponent's king, the move is categorized as *direct check*.
- **Indirect Check**
  When the cover of an attacker's long-range-piece (*e.g.* rook) reaches opponent's king after an attacker's obstacle piece moved, the move is categorized as *indirect check*.
- **Drop Check**
  When an attacker's piece in hand is dropped and it attacks the opponent's king, the drop is categorized as *drop check*.

Defender's moves are also categorized into 3 groups. Explanations for the categories are given below.

- **King Escape**
  When a defender's king moves to escape from the attacker's check and successfully avoid to be checkmated, the move is categorized as *king escape*.
- **Defense by Capture**
  When a defender's piece on the board captures the piece attacking defender's king, the move is categorized as *defense by capture*.
- **Defense by Drop**
  When a defender's piece in hand is dropped and it blocks the cover of a long-range-piece attacking defender's king, the drop is categorized as *defense by drop*.

3 Data Structure and Algorithm

In this section, we first describe the structure of data used in tsume-shogi hardware. Then, we describe the procedure for generating check and defense moves in the circuit.

3.1 Data structure

Here we describe important data structures that are used for move generation.

- **Piece Data**
  Piece data is 6 bit data assigned for each piece to distinguish a type of piece.

- **Board Data**
  Board data is a set of piece data and is information about which piece is on which square. As piece data is 6 bit and the board size is 9×9, total size of board data is 486 bit. In our hardware, a rank (9 squares) of board data is loaded at a time, thus the data is stored in 9 (depth) × 54 (width) size memory.
- **Black/White Direct Piece Cover**
  If a piece **P** can move to square **S**, we say that **S** is *directly covered* by **P**. Black (white) direct piece cover is information about which black (white) piece is covering which square. Since direct cover data assigned for one square is 66 bit, total size of the data is 5346 bit. To read/write a rank of data at a time, cover data is stored in 9 (depth) × 594 (width) size memory.
- **Indirect Piece Cover**
  Suppose a position where the cover of the long-range-piece **P** is blocked by the obstacle piece **Q**. If **P** can move to the square **S** only when **Q** is removed, we say that **S** is *indirectly covered* by **P**. The data assigned for one square is 8 bit, thus the whole data is stored in 9 (depth) × 72 (width) size memory.
- **Direct Check Mask**
  Direct Check Mask is used to eliminate invalid moves so that only direct check moves remain. The data assigned for one square is 5 bit, thus the whole data is stored in 9 (depth) × 45 (width) size memory.
- **Indirect Check Mask**
  Indirect Check Mask is used to eliminate invalid moves so that only indirect check moves remain. The data assigned for one square is 4 bit, thus the whole data is stored in 9 (depth) × 36 (width) size memory.
- **Move Data**
  Move data is 24 bit data and is information about which piece moves to which square. There are at most 10 pieces (except pieces in hand) that can move to the square **S**. If all of the pieces can promote when they move to **S**, at most 20 moves can be generated (except drops). Our hardware deals with 9 squares at a time, thus a memory to store the move data needs to have wide enough bandwidth to read/write 180 moves simultaneously.

3.2 Procedure for Move Generation

Since all attacker's moves must be check in tsume shogi, an elimination of invalid moves from all legal ones is to be performed. For an efficient move elimination, we use *mask* that distinguishes check moves from the others. The mask is generated in parallel to computation of other various data. After the mask and those data are generated, check moves are generated in parallel. As already explained in Section 2.3, check moves are categorized into 3 groups and these 3 types of check are generated in discrete circuits to achieve the high parallelism. The procedure for generating check moves is summarized as follows.

1. Update of Position
   (a) Board data
   (b) Piece in hand
   (c) Pawn data
2. Update of data used for move generation
   (a) Direct piece cover
   (b) Indirect piece cover
   (c) Direct check mask
   (d) Indirect check mask
3. Check move generation
   (a) Direct check
   (b) Indirect check
   (c) Drop check
4. Store check moves to memory

In this procedure, operation from (1) to (4) are processed in course grained pipeline and (a) to (d) are processed in parallel.

Defender's moves are necessarily ones to avoid attacker's check. Computation of defense move generation is also performed in parallel and in course grained pipeline. The procedure for generating defense move is summarize as follows.

1. Update of position
   (a) Board data
   (b) Piece in hand data
   (c) Pawn data
2. Update of white direct piece cover
3. Defense move generation
   (a) King Escape
   (b) Defense by Capture
   (c) Defense by Drop
4. Store defense moves to memory

3.3 Block Diagram

Figure 5 shows a block diagram of the move generator implemented on an FPGA. The number described on each module corresponds to the pipeline stage of computation of move generation. All modules described in Figure 5 work in parallel, and additionally, parallel data processing of 9 squares and fine-grained pipeline computation are performed in all modules. Therefore all groups of moves are generated in quite short time.

As space is limited, architecture of all modules can not be explained. We describe architecture of *Black Direct Cover Circuit* in the next section as an example of parallel and pipeline processing.

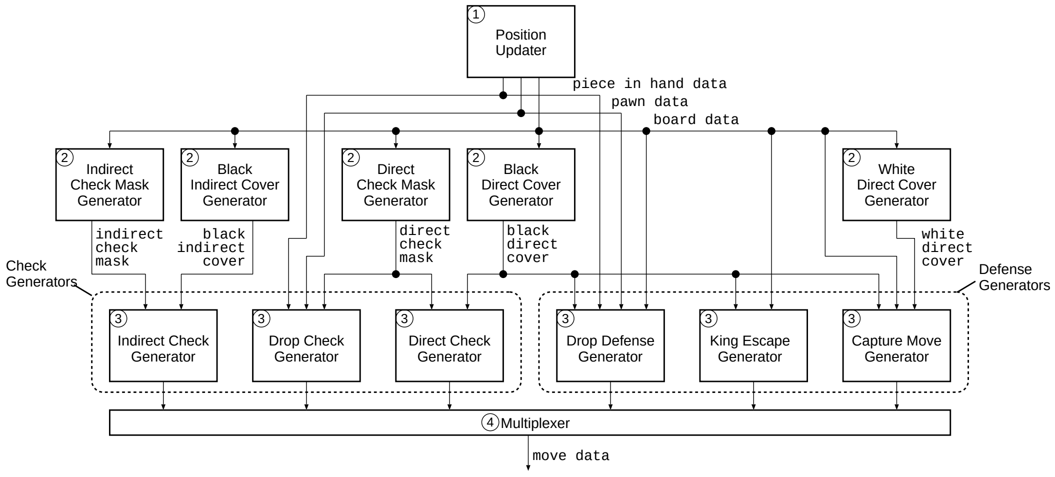

Figure 5: A block diagram of tsume shogi hardware

4 Black Direct Cover Generator

Black Direct Cover Generator is a module to calculate attacker's piece cover data. Piece cover data is information about which piece can move to which square. In this section, the structure and performance of the piece cover generator is given as an example of implemented parallel and pipeline architecture.

4.1 Procedure for Piece Cover Computation

Ideally, piece cover computation with hardware should be performed to all 81 (9x9) squares simultaneously. In that case, however, complicated wiring causes a long delay and a decline of hardware performance. In our module, 9 squares (1 rank) are processed at a time to generate piece cover data. Although it takes 9 clocks to read all board data with this method (we call this data loading a *scan*), hardware works at high frequency because of simple wiring. Additionally, since whole board data (9 rank) is processed in pipeline, total computation time is not long.

The cover of long-range-pieces (*e.g.* rook) is transmitted to adjacent squares every clock. When we scan the board from rank 1 to rank 9, the cover of a long-range-piece is only transmitted from top to bottom. Therefore the board scan must be performed from top to bottom (*top-down scan*), from bottom to top (*bottom-up scan*), from left to right (*left-right scan*) and from right to left (*right-left scan*). These four scans are performed in parallel, thus total computation time is as short as the one-way scan. Piece cover data is obtained by calculating OR of outputs of the four scans.

4.2 The Structure of the Cover Generator

Figure 6 shows the block diagram of Black Direct Cover Generator. *Top-Down Scanner* and *Bottom-Up Scanner* namely perform a board scan from top and bottom respectively, and *Rook Cover Scanner* performs a left-right/right-left scan. These scanners all work in parallel, and inside of each scanner is also highly parallelized for further speed-up.

As space is limited, we explain the structure of Top-Down Scanner as an example of parallel and pipeline processing performed inside a module. The structure of Top-Down Scanner appears in Figure 7. $Sq\,X$ ($X = 1, 2, \cdots, 9$) is a module to calculate the cover of the piece on file X. As Figure 7 shows, piece cover computation for 9 squares is performed in parallel. Other scanners in Figure 7 and data generators in Figure 5 also have similar architecture to perform parallel processing.

4.3 Performance of the Cover Generator

In the Black Direct Cover Generator, following 3 processes are performed in pipeline: loading board data, cover computation and storing piece cover. Figure 8 shows the timing of the computation. We see from Figure 8 that 9 rank computation is finished in 11 clock with fine-grained pipeline processing.

5 Implementation and Performance

In this section, we first give an explanation for the device used for the tsume-shogi solver. Next, we show the result of the implementation. Lastly we discuss the performance of implemented modules.

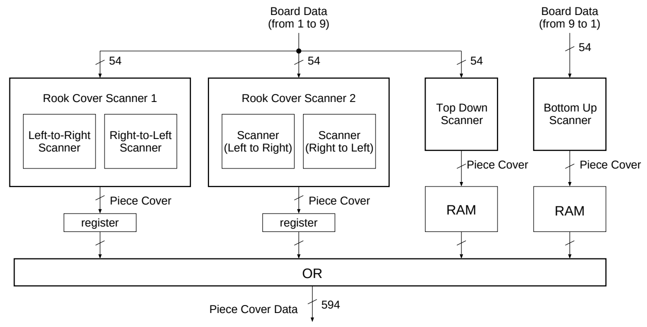

Figure 6: The structure of *Black Direct Cover Generator*.

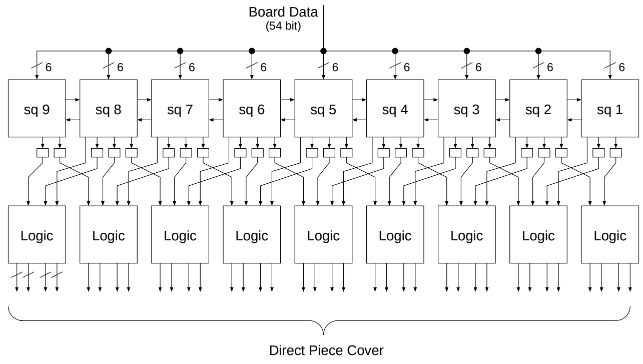

Figure 7: The structure of *Top Down Scanner*.

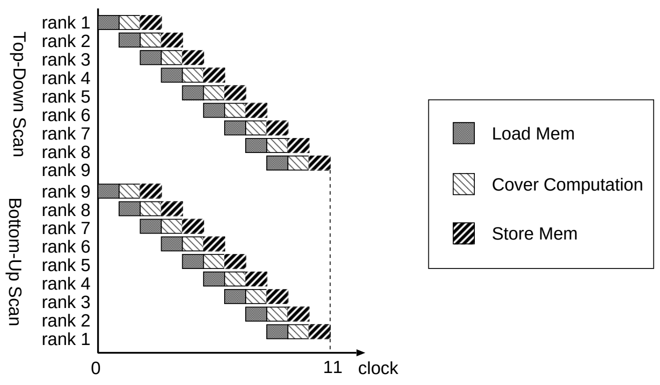

Figure 8: Pipeline processing in *Black Direct Cover Generator*.

5.1 Virtex II

Modules in the tsume-shogi circuit are implemented on Virtex II XC2V6000FF1152-5 (Xilinx, Inc.). The chip has 33792 slices and 144 of 36864 bit block RAMs [14]. The width of a block RAM can be changed from 1 to 36 bit (single port) or from 2 to 72 (dual port), so the maximum memory width that the chip can provide is 10368bit. Virtex II also provides *Distributed RAMs* using LUTs as memory instead of logic gates. When distributed RAMs are implemented, the maximum memory width exceeds 10368 bit.

5.2 Implementation Results

Here we show the result of the implementation of modules in Table 1. *Frequency* shows the frequency of each module implemented on an FPGA. The whole circuit works at 37.84 MHz as it depends on the slowest module. *Resource* is the number of slices used in each module. The Table 1 shows that implemented modules totally use 81% of hardware resource. We see from *Output width* that quite wide data is output from each module at a time. The problem of this exceedingly wide outputs was resolved with a large number of block RAMs. As Table 1 shows, data that must be written to memory at a time is about 8000 bit-width, and 100 block RAMs are used to store all data simultaneously.

Figure 9 describes the timing chart of the tsume-shogi solver. Figure 9 shows that required clock cycles for move generation is $62 + N$. N is the number of valid moves by the attacker or the defender in a position. In all modules, required clock cycles except N is always constant and does not depend on the number of moves to be generated.

5.3 Performance

As the frequency of the circuit is $37.84\,\mathrm{MHz}$ and required clocks is $62 + N$, computation time for generating N moves in a position (T_{N}) is given by the following equation:

$$T_{\mathrm{N}} = (62 + N) \times \frac{1}{37.84}\ [\mu\mathrm{sec}]. \qquad (1)$$

As found out from the equation above, computation time for the move generation in hardware does not significantly

Table 1: The implementation result of each module.

Module	**Frequency**	**Resource**	**Output width**	**Block RAM**
Position Updater	78.95 MHz	244 Slices	114 bit	2
Black Direct Cover Generator	54.56	2418	594	17
Black Indirect Cover Generator	101.08	474	32	3
Direct Check Mask Generator	79.97	676	45	3
Indirect Check Mask Generator	85.35	577	36	3
Direct Check Generator	70.23	856	1080	9
Indirect Check Generator	65.19	2115	1440	12
Drop Check Generator	101.55	159	63	1
White Direct Cover Generator	54.87	1932	594	8
King Escape Generator	50.71	596	214	1
Capture Move Generator	40.93	3074	2160	20
Drop Defense Generator	46.67	3081	1512	21
Multiplexer	37.84	10881	24	-
TOTAL		27083 (81.7%)		100 (69%)

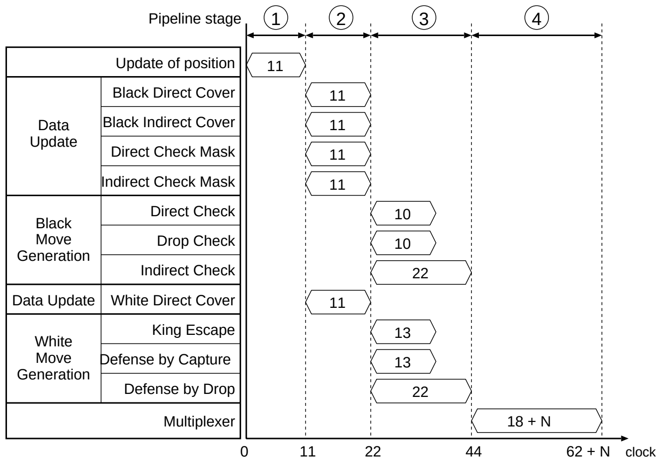

Figure 9: Timing chart of move generation in tsume shogi hardware.

depend on the number of generated moves because of highly parallelized architecture of the circuit. For example, we give T_5 and T_{10} below.

$$T_5 = (62+5)\ \times\frac{1}{37.84} = 1.77\ [\mu\text{sec}] \qquad (2)$$

$$T_{10} = (62+10)\times\frac{1}{37.84} = 1.90\ [\mu\text{sec}] \qquad (3)$$

T_{10} is only 1.07 times longer than T_5 in hardware, while that in software would be about twice longer. Therefore the circuit can work efficiently for a tsume shogi problem where lots of possible moves are to be generated.

Then, let us discuss how many moves could be generated per second with a fully implemented tsume-shogi solver. The total performance of the tsume shogi solver depends on the implemented tree search algorithm. As there are lots of search algorithms, it is difficult to estimate all varieties of tsume shogi solvers. To estimate the maximum performance of the tsume shogi solver, we focus on the most simple search algorithm: full-width search. In full-width search, computation of tree search can be overlapped with move generation. In this case, performance of the circuit reaches the highest peak and it is calculated by the following expression:

$$\frac{1}{T_\mathrm{N}} \times N. \qquad (4)$$

Since the average of N is known as 5 [12], the average number of generated moves per second is given by the following equation:

$$\frac{1}{T_5} \times 5 = 2.824 \times 10^6. \quad (5)$$

It is difficult to compare the performance of hardware with that of software, because the performance of hardware does not significantly depend on N but software does. Furthermore, there are lots of tsume-shogi algorithms and there is no consensus about which algorithm is best. When compared with our software, the performance of the circuit is roughly estimated as 10 – 50 times faster.

6 Current Status and Future Works

In this paper, we presented an FPGA-based processor to solve *tsume-shogi* (mating problems in shogi). Shogi is a challenging target in artificial intelligence for game playing. In shogi, however, frequently revised algorithms and quite large application size have prevented researchers from developing dedicated hardware. The flexibility and recent progress in size of FPGAs are expected to be solutions to the problems in hardware development: outdated architecture and lack of hardware resource.

We implemented all modules required for move generation in tsume-shogi and tested the feasibility of FPGAs for the large and complicated application. The latest FPGA enabled us to implement all tsume-shogi modules on a single chip and to eliminate the bottleneck of memory bandwidth. With 100 block RAMs on an FPGA, we realized highly parallelized move generation in spite of the exceedingly wide data output. When the number of branching factor of a tsume-shogi tree is 5, the computation speed of move generation in hardware is roughly estimated as 10 – 50 times faster than that in software. The performance of hardware against software depends on the game tree complexity. Hardware shows better performance in more complex tsume-shogi problems.

What is remaining to be done in our study is to implement a tree search controller. Since the completed modules are general for most tsume-shogi programs, we can implement various tree search algorithms by changing the search controller. Some algorithms can very quickly solve short-ply tsume shogi problems but can't solve long-ply ones, while some can solve long-ply problems but are inefficient for shot-ply ones. Therefore it could be effective to reconfigure the search controller dynamically according to search depth. We continue the implementation of tsume-shogi hardware and attempt to test the feasibility of reconfiguration for tsume-shogi.

References

[1] Allis, L. V.: Searching for Solutions in Games and Artificial Intelligence, Ph.D thesis, University of Limburg, Maastricht (1994).

[2] R. Grimbergen: A Plausible Move Generator for Shogi Using Static Evaluation, *Game Programming Workshop*, pp.9-15 (1999).

[3] R. Grimbergen: Plausible Move Generation Using Move Merit Analysis with Cut-Off Thresholds in Shogi, *Computers and Games*, pp.315-332 (2000).

[4] Y. Hori et al.: A Shogi Processor with a Field Programmable Gate Array, *Computers and Games*, pp.297–314 (2000).

[5] Iida, H. and Uiterwijk, J. W. H. M.: Thoughts on Grandmaster-like Strategies, *European Shogi Workshop*, Shogi Deutschland and EMBL, Vol.2, pp.1-9 (1993).

[6] Matsubara, H.: Algorithms of move generation in a Shogi program, *Proc. of The Game Programming Workshop*, Computer Shogi Association, Hakone, pp.134-138 (1994). (in Japanese)

[7] H. Matsubara and R. Grimbergen, Differences between Shogi and western Chess from a computational point of view, *Board Games in Academia*, an interdisciplinary, Leiden, The Netherlands (1997).

[8] Monty Newborn: "Kasparov versus Deep Blue: computer chess comes of age," Springer (1997).

[9] Nozaki, A.: Logical Shogi Introduction, Chikuma-shobo, Tokyo (1990). (in Japanese)

[10] Ohtsuki, A.: Logical Shogi Introduction, Chikuma-shobo, Tokyo (1995). (in Japanese)

[11] Jeff Rollason: SUPER SOMA – Solving Tactical Exchanges in Shogi without Tree Searching, *Computers and Games*, pp.277-296 (2000).

[12] M. Seo: The C* Algorithm for AND/OR Tree Search and its Application to a Tsume-Shogi Program, M. Sc. thesis, Department of Information Science, University of Tokyo, Japan (1995). (in Japanese)

[13] Schaeffer, J., et al.: Heuristic Programming in Artificial Intelligence 2, the second computer olympaid (eds. D. N. L. Levy and D .F . Beal), pp.119-136, Ellis Horwood Ltd., Chichester, England (1991).

[14] Xilinx, Inc.: Virtex-II 1.5V Field-Programmable Gate Arrays – Advance Product Specification (2001).