A Hardware-based Minimax Implementation of a Blokus Duo Agent Reconfigurable Digital Systems - Technical Report

Demertzis Ioannis AM 2013039008
Nikolakaki Sofia-Maria AM 2013039041
September 23, 2014

1 Introduction

Artificial Intelligence (AI) is a term frequently encountered in every day life, thus proving its significant applicability on a wide range of fields. Among these fields, some of the more popular ones include playing games competitively and decision theory problems. The most dominant solutions to such problems are knowledge-based approaches and the Minimax algorithm. The performance of these approaches appears to be interesting in decision making problems that have either one or more of the following properties: a high branching factor, a deep tree or a moderate heuristic function to evaluate non-terminal nodes. An example of such a problem that comprises these properties is the game of Blokus Duo, a perfect information two player zero-sum game with a finite number of moves and no chance elements a.k.a a combinatorial game. The specific game drew our interest because it was the challenge of the ICFPT 2013 design conference, it is relatively new with few published heuristic functions but, most importantly, the first levels of the tree created have an exponentially increasing number of children nodes and the number of candidates is sometimes over 1000, properties that indicate its demanding nature. With the recent development of the MCTS algorithm, it has been commonly accepted that Minimax performs poorly in complex games, like Go. However, although Blokus Duo is a complex problem, it still has a much smaller game size than Go since the maximum length of the game is 42 moves. Thus we cannot predetermine whether a Minimax-based Blokus Duo player is adequately competitive. Taking this into consideration we decided to evaluate the performance of Minimax in a game that is demanding, but not too complex like Go. In addition, to the best of our knowledge the MHL lab of Technical University of Crete has not designed an architecture for a game that uses limited pieces in every round and therefore requires checking the availability of those pieces. However, it has developed many Minimax-based players such as [1] and [3]. This is another challenge, due to the fact that such games require a more complicated control, even if the algorithm we are implementing is a common Minimax algorithm. Taking into account the above, we concluded that the creation of a hardware-based Minimax Blokus Duo player is an interesting problem with an unpredictable outcome in terms of parallelism and performance.

Chapter 2 introduces us to the Minimax algorithm, the rules of the Blokus Duo game, as well as critical strategies followed by professional players. Chapter 3 describes the results of the profiling performed on the respective software. We also analyze hardware opportunities and bottlenecks that this algorithm offers. In Chapter 4 we present the hardware-based implementation of the Blokus Duo player with detailed architecture diagrams and descriptions. Chapter 5 describes performance results, in terms of hardware resource utilization and speed up. Finally, in Chapter 6 a conclusion about the presented work is provided, followed by future work directions that are worth considering.

2 Background

2.1 The Game of Blokus Duo

Blokus Duo (a.k.a Travel Blokus) is an expansion of the game of Blokus. It is a zero-sum, perfect information, 2-player game. Given that Blokus Duo is relatively new there has not been much research conducted on it, which makes it challenging and intriguing. In the following subsections we first present the history of the Blokus Duo game and then we explain the rules. Next, we mention some of the properties that characterize the game and finally we describe existing programs that implement competitive Blokus Duo players.

2.1.1 Rules

In order to explain how the Blokus Duo game is played we adapted the rules published by Mattel.

Components

- 42 game pieces (tiles) in total - Two 21-piece sets (usually one of orange and one of purple color). Each set contains 21 pieces of different shape shown in Figure 1. A tile comprises unit squares i.e. the little squares that compose a tile. There is 1x1-unit tile, 1x2-unit tile, 2x3-unit tiles, 5x4-unit tiles and 12x5-unit tiles.
- A game board with 14x14 grid size, i.e. 196 squares.

Goal

Each player aims to place as many of his 21 pieces as possible on the board.

How to play

1 Each player is assigned a color (orange or purple) and gets the respective tiles. Both players are allowed to flip or rotate any tile before placing it on the board.

2 Whoever decides to play first is deemed as Player 1 and the other player is considered as Player 2. In the beginning of the game, Player 1 places any tile he wants in any way at a specific starting point on the board with coordinates (5,5). Then, Player 2 does the same thing but his starting point has coordinates (10,10) or (a,a) of the example board in Figure 2.

3 The game continues as each player places one piece at a time on the board. The players play in an alternate manner and should follow the following restrictions:

   – The to be placed tile must have at least one corner-to-corner contact with a tile of the same color, as shown in Figure 3
   – The to be placed tile must not have edge-to-edge contact with any tile of the same color, as shown in Figure 4
   – Different colored tiles can touch in any manner
   – The position of a placed tile cannot be changed until the end of the game.

4 In case a player does not have any legal moves to make or cannot find one, that player should pass. From that point to the end of the game the player who passed his turn cannot lay any other piece on the board.

5 The game ends when any of the following conditions are met:

   – One of the two players has placed all his tiles on the board

– Both players have passed their turn

6 Once the game has ended, scores are calculated and the player with the highest score wins.

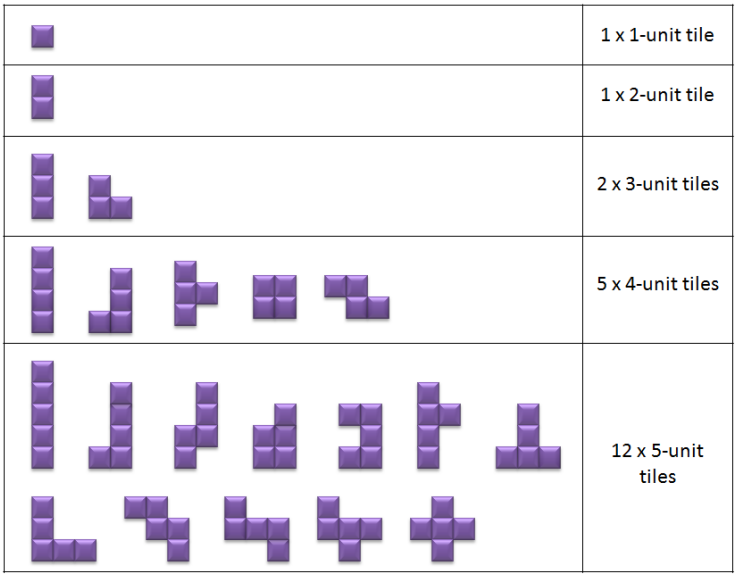

Figure 1: Tiles of the Blokus Duo game. Each tile comprises from one to five units.

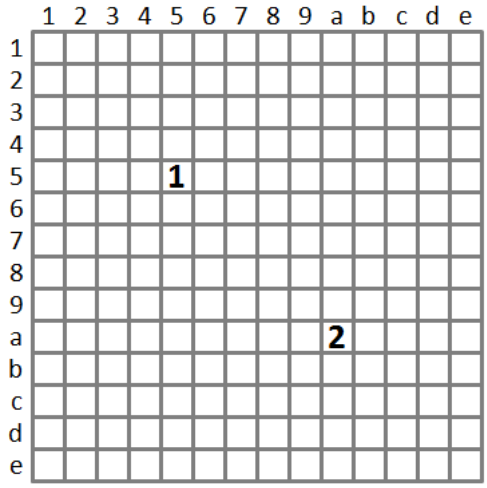

Figure 2: Board of the Blokus Duo game. Square 1 indicates the starting point of player 1 and square 2 indicates the starting point of player 2.

An example of a game is shown in Figure 5

Score

Each player counts the total number of unit squares of their remaining tiles that were not placed on the board. Each unit square counts as '-1' and therefore adding unit squares gives a negative total. The player

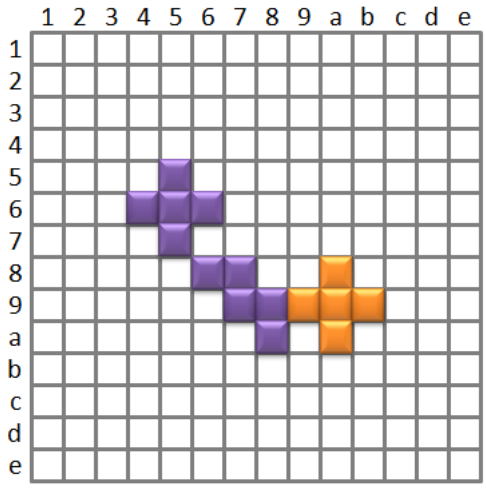

Figure 3: The purple tiles are connected with a corner-to-corner contact.

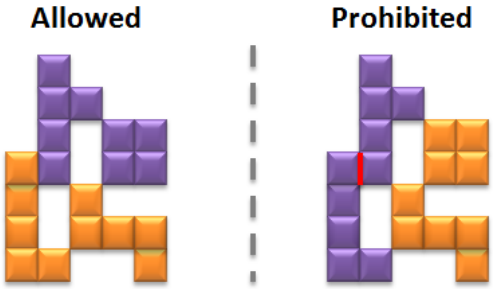

Figure 4: The left move is allowed, since same colored tiles are connected with a corner-to-corner contact but do not have any edge-to-edge contact. The right move is prohibited, since the purple tiles have an edge-to-edge contact.

who gets the highest score wins. Specific bonuses are given in the following situations:

- There is a +15 bonus added to any player that places all of his tiles on the board
- Additionally to the previous +15 bonus there is also a +5 one if the last tile placed on the board is the 1 unit square piece

An example of a completed game and how the score is counted is shown in Figure 6

2.1.2 Strategy Tips

As all games, Blokus Duo also has some fine strategic points that may determine the outcome of the game. We present some of these, to provide a better understanding of the game.

- Given that the score is computed based on the non-placed unit squares, each player should aim to minimize this number. Therefore the bigger the tile, the bigger the urge to place it in the first rounds of the game since later on, it may not fit on the board. One of the common mistakes that new players make is playing a 4-piece when a 5-piece can accomplish exactly the same thing. Note that most of the 4-pieces are contained within quite a few of the 5-pieces and so the latter could be placed when possible, since they are worth more points

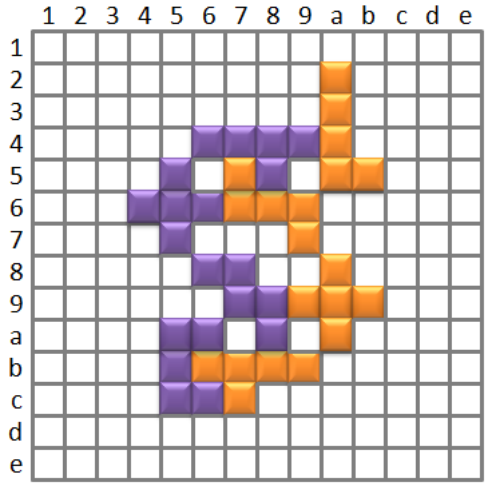

Figure 5: An example snapshot of the game.

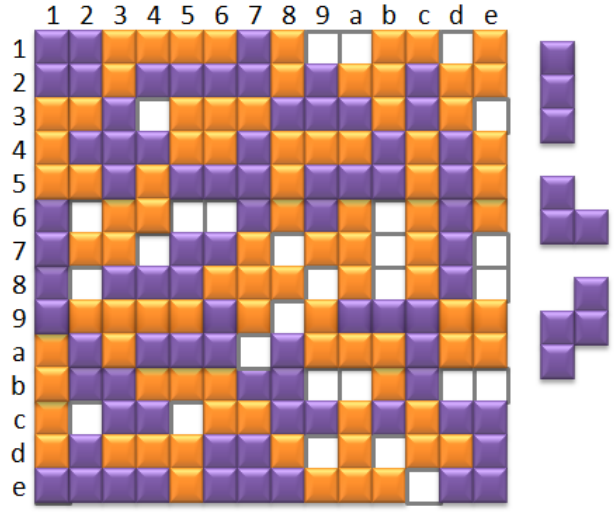

Figure 6: An example of a completed game. The orange player placed all of his tiles, hence he gets a bonus of 15 points. The purple player did not place 2 three-unit tiles and a four-unit tile. Therefore, his score is $2 * (-3) + (-4) = -10$. The orange player has the highest score and is the winner of this game.

- Each player should try and move towards the center of the board from the beginning of the game. The center is of great strategic importance because it is both, a safety net and a control territory. The former, because it provides multidirectional ways out in case a player is blocked and the latter for basically the same reason but now instead of ways out, the center provides paths that lead to areas that can be dominated.
- While playing, a player must think forward before deciding to fight or to flight. More specifically, in a situation where the opponent has claimed a big area somewhere, one might think of making a move to fight back by reducing the number of the opponent's corners or by trying to lessen his space. However this is a risk move because without realizing it the player might let his opponent invade his small space while his attack may evidently have no impact. Another approach would require giving up some

space but making it hard for the other player to access the area you already control. Therefore, before rushing into making any reckless moves, one should try and figure the opponent's possible plans and then act appropriately.

- The basic principle of Blokus Duo is to cover as much space as possible. Both players struggle throughout the game to prevail on the space of the board. Whenever a player controls an area on the board it is more likely that he will dominate it. Hence, it is important to know which areas are the bigger ones and aim to control these first. All areas of the board are shown in Figure 7.
- Detect areas that are open and have usable space, while making moves that will either lead towards the direction of these areas or that will yield good plays in that space later on in the game. For example, one could create available corners in these areas, in order to link other pieces easily and attempt to contol that space. In general, due to the corner-to-corner attachment rule, creating corners implies more available positions to place a tile. However, the same applies for the opponent too and therefore, a player must focus upon creating corners for himself, while blocking the opponent's corners.
- Some tiles must be placed carefully in the early stages of the game. More specifically, some tiles are sneaky ones as they do not only provide corners but they are also flexible to fit in small holes. If played appropriately, they can assist other tiles to be placed towards the end of the game, when each player tries to have as few tiles as possible left.

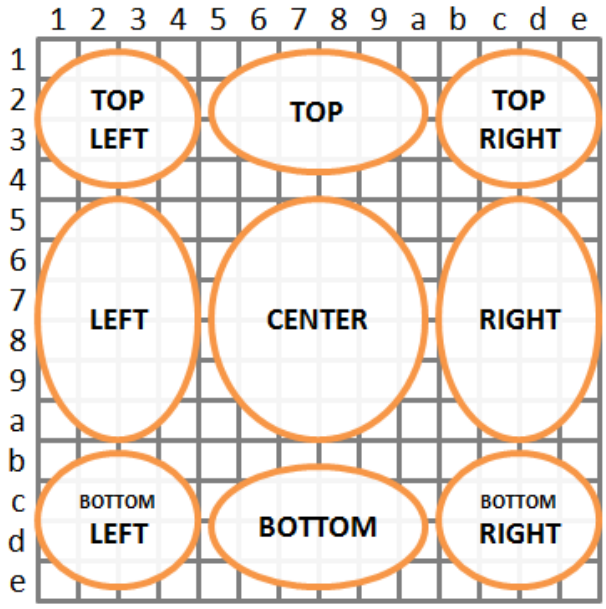

Figure 7: Important areas on the Blokus Duo gameboard

2.1.3 Existing Programs

Due to the fact that Blokus Duo is a quite recent game compared to other popular AI games, such as Go and Chess, there are only few reliable Blokus Duo computer programs. Among these we consider only three reliable, Pentobi, metablok and block'em, all licensed by GNU. Although Pentobi surpasses by far all other attempts, it does not come with a documentation while the others mention that they are Minimax based agents.

2.2 Minimax Algorithm

In actual games it is common to evaluate a player's performance in the terminal states of a game. That's why many algorithms use game trees to depict possible complete scenarios and outputs of the game. Minimax is the most applied algorithm on combinatorial games that involves a game tree.

Minimax is an algorithm that takes into account every possible move that the opponent might make and performs a recursive depth-first search exploration with "back-ups". The goal of Minimax is to estimate the best move in the actual game round and thus the information from the game tree is used appropriately. In order to evaluate game states, Minimax uses the Minimax value which is defined as follows:

$$\begin{gathered}\text{Minimax-value}(n)= \\ \text{UTILITY}(n) \text{ If } n \text{ is a terminal node} \\ max_{sinsuccessors(n)} \text{ Minimax-value}(s) \text{ If } n \text{ is a MAX node} \\ min_{sinsuccessors(n)} \text{ Minimax-value}(s) \text{ If } n \text{ is a MIN node}\end{gathered}$$

MAX selects the move that maximizes the Minimax value, while MIN selects the move that minimizes the Minimax value. The utility value calculated in terminal states represents a player's strategy, since it sums up all the factors that the player considers possible to maximize the likelihood of winning. Furthermore, the utility value is propagated from each leaf towards its predecessors, selecting either the minimum or the maximum value at each level. The first, is propagated whenever the opponent selects a move, since he desires to minimize our score, while the latter is the reverse case, since we desire to maximize our score and play the best move. For further details about the Minimax algorithm we refer the reader to [2, p. 160]

2.2.1 Complexity

- **Time Complexity:** The maximum number of leaf node positions evaluated is $O(b*b*b..b*) = O(b^d)$, where b is the average or constant branching factor and d the search depth of plies.
- **Space Complexity:** $O(b*d)$, where b is the average or constant branching factor and d the search depth of plies.

3 Modeling for Hardware Implementation

In this section we present critical points of the software code that indicate hardware opportunities. In order to do so, we performed profiling of the code using the Linux GNU GCC Profiling Tool (gprof), we computed the size of the structs used in the software code, we detected potential parallelism, as well as potential bottlenecks.

3.1 Code Profiling

The resuts of the code profiling showed that greater time consumption occured in the *check_move()* function. We expected such a result, due to the fact that the complexity of *check_move* is $O(n^2)$, as an 5x5 area of the board is evaluated. Also, knowing whether a move is valid is essential, since most of the rest of the functions depend on it. Therefore *check_move()* is called many times in different parts of the algorithm. The next most time consuming process is executed in the *returnAllValidMoves()* function. This result can be also interpreted, as we know that in order to find all valid moves, we need to consider all remaining tiles, with their respective rotations for each board position. Given that in the worst case the tiles are 21, the rotations are 8 and the board size is 14x14, it is clear that 21x8x14x14 iterations are needed to find all available moves, a number that affects significantly the overall execution time of the algorithm. Furthermore, *GreedyEvaluate()* is the next most time consuming process, as besides knowing the units of the tile to be placed, it requires knowing the available corners of both players before and after placing the tile. However, counting a player's available corners on the board is an expensive process, since a corner might be anywhere on the 14x14 squares of the board.

3.2 Memory Requirements

In order to determine the memory requirements of the Minimax algorithm, we present the size of the two structures that are involved in the specific algorithm. The first is the structure *move* that consists of four integer values and therefore has a size of 16 bytes. The second is the linked list *moves*. Each element in the list contains an object of type *move* and a pointer that shows to the next item of the list, hence each element of the list has a size of 24 bytes. This list contains all valid actions that can be performed for a certain game state and therefore may utmost contain 21x8x14x14 moves, for the reasons mentioned in 3.1. However, this is an unrealistic case, as on average there are 10 available corners for each player (10 squares on the board) and definitely less than 21x8 tiles to be placed, since this number also contains duplicates as mentioned in 4.1.1. Therefore, we can estimate that the size of a list of moves will have approximately a size of (24bytes)x(10corners)x(21tiles)x(4rotations) = 20160 bytes.

3.3 Potential Parallelism

We mentioned that the most time consuming process of the Minimax algorithm is checking whether a tile fits in a specific position of the board or not. We also explained that this procedure requires to check a 5x5 area on the board, hence 25 iterations are needed. However, each square of the 5x5 area on the board can be evaluated independently from the rest, since we do not need a previous value to determine its state. Therefore, we can determine a valid position in a single clock cycle, thus reducing significantly the time complexity of the *check_move function*. Furthermore, computing the number of corners for each player can be also parallelized and determined in a single clock cycle, to reduce the 14x14 sequential checks. However, this would imply a slow clock and instead of a single clock cycle, perhaps 6 or 7 would be recommended.

3.4 Potential Bottlenecks

Finding a move that is valid seems to be a potential bottleneck for a hardware implementation, since it arbitrarily delays the execution of the algorithm. More specifically, checking whether a tile fits in a square of the board requires a single clock cycle. However, we might evaluate several board positions before finding a valid move and therefore this procedure needs an arbitrary number of clock cycles. Note, that in an extreme case where 10 tiles are left and none of them fits in the board, we will need (10tiles)x(8rotations)x14x14 = 15680 clock cycles to realize it. Also, Minimax is a recursive algorithm and this fact inevitably limits the amount of parallelism that can occur.

4 Hardware Implementation

In the hardware implementation Minimax was performed in the Minimax Core Module, shown in Figure 8. The Minimax Core Module comprises of multiple subsystems. The first stage of the algorithm is implemented in the MetaData Module. This module generates new tiles when requested, after the board is updated with an opponent move. In the beginning, a first possible move is produced according to the first available tile that is found in the tile availability vector. Next, the control, shown in Figure 9, activates the Move Module in order to find for a given tile, the first valid move on the board. This move indicates that a new game state should be defined and pushed into the stack for future use. A game state will contain all essential information that describe the state. This information comprises the tile and rotation that are used in the new move, the coordinates of the board where the placement will occur, the player who made the move and the evaluation score. Simultaneously the board is updated according to the new game state. This process continues iteratively until a terminal game state is met or a predetermined depth has been reached. In our implementation this depth is predefined to have value 3 due to the fact that Blokus Duo is a complex game with many possible moves to make in each round. Then the Push Module is responsible for calculating the evaluation score and adding the new move into the Stack. Note that the main factors that are taken under consideration for the calculation of the position score are: i) the incremental number of corners of the player after placing the tile, ii) the reductive number of corners of the opponent after placing the tile and iii) the

new score after placing tile.
Once the predefined depth of the Minimax tree has been met, the Pop Module is activated in order to remove the last added move from the Stack. This move is compared to the currently best move that was found in previous stages of the algorithm and the between the two, the move with the highest or lowest score is stored, depending on whether we are in a MAX or MIN node respectively. In the end of the Minimax algorithm the best move will have been found. Note that we only use one board and two registers that show the availability of the tiles during the Minimax process, so as to avoid storing each time a new board and registers. In fact we implemented a do and an undo function to move on or return to previous game stages. The sequence of the moves is already known from the sequence in the stack (we know which move was inserted first, second or third). In order to calculate all possible outputs, both players need to be considered. Whenever it is the opponent's turn to play, the board is flipped and the respective tile availability vector is taken into consideration.

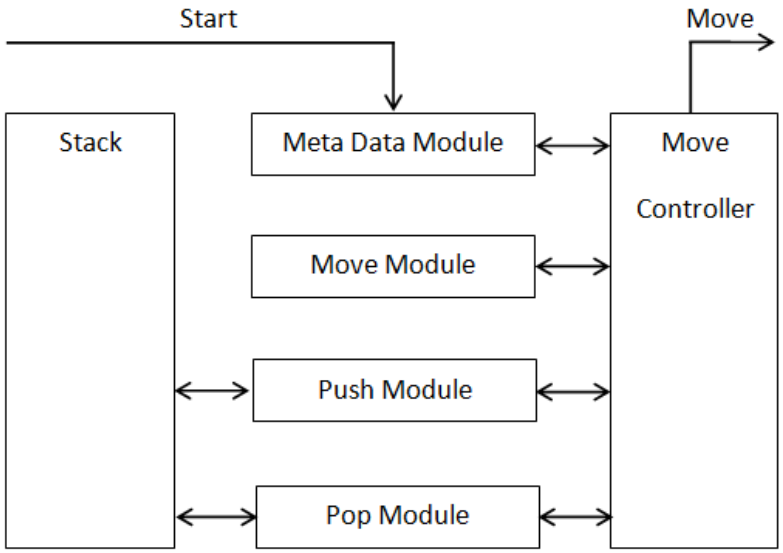

Figure 8: Minimax core module.

4.1 Blokus Duo components

The two components of the Blokus Duo game are each player's tiles and the game board, that were described in 2.1.1. However, during implementation we considered efficient ways for their representation, in terms of execution time and memory allocation.

4.1.1 Tiles

In the beginning of the game, each player has at his disposal 21 different tiles that can be flipped and rotated in any manner. Therefore the player does not consider 21 different tiles, but 168 (21 pieces x 4 rotations x 2 flips). However among these 168 tiles, some are symmetrical with respect to the x or y axis or sometimes both and thus there can be in total eight, four, two or one different representations of a specific tile. By eliminating all duplicates we no longer consider 168 distinct tiles, but 92, thus avoiding examining the same tiles multiple times. A Blokus Duo tile consists of at the most 5 unit squares, thus we needed a 5x5 array to be able to depict each one of them. In our hardware implementation we consider each tile to be represented by a $7x7x2$ array. This representation offers a single cycle determination about whether a tile fits somewhere on the board. More specifically, given a row, a column, a board and a $7x7x2$ tile, we only need to "place"

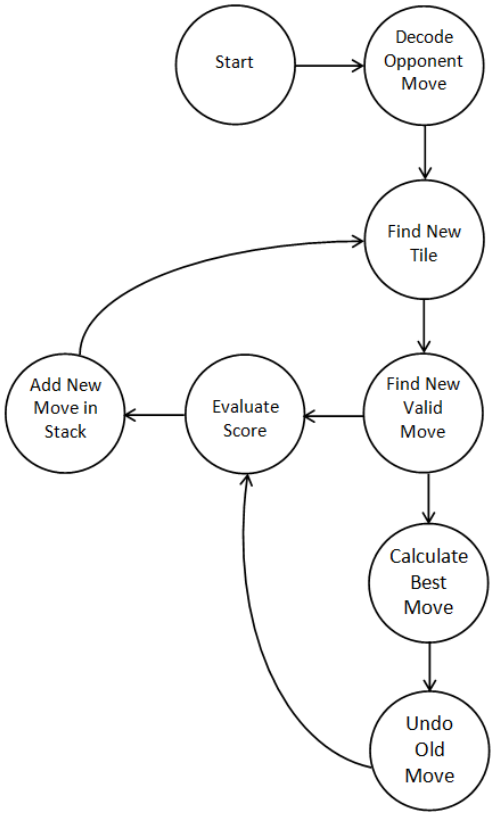

Figure 9: Minimax controller's FSM

the tile on the specific position of the board and examine each of the $7x7x2$ cells of the tile compared to the respective cells of the board. Note that generally the tile only needs a $5x5$ representation to be depicted. However extending it to the $7x7$ dimension allows us to check the game status of the neighboring cells. For example, the tile should not have contact with same colored tiles i.e. should not have neighboring cells of the same color. In addition, we use two bits to describe each cell of the $7x7$ tile so as to represent different states that leads to the single cycle evaluation. In particular, a cell with an "00" indication can be placed anywhere on the board, whereas cells "01" and "10" can be placed on an empty cell and cannot have underneath a same colored cell respectively.

The FPGA's internal Block RAM-BRAM (90x98) was used for the storage of the tiles with their distinct rotations (without duplicates). The distinct rotations of a tile were stored together and in a specific sequential order to ease the implementation. During the Minimax algorithm we need to transform the 98 bit length tile to the $7x7x2$ representation. The reason for storing it in such form is because we need to retrieve a tile from memory in a single clock cycle. Furthermore, a lookup table is used in order to reduce memory accesses and swiftly index specific tiles. This lookup table consists of the first address of each tile, as well as its total number of rotations. Therefore evaluating all rotations of a tile is simple since we know where the first representation of the tile is located and by simply increasing the address by one we move to a next rotation until the total number of rotations is reached.

4.1.2 Game Board

In the Blokus Duo game the board is a 14x14 grid. However, due to the fact that each tile is represented as an array, we considered the board as a 18x18 grid, since in certain cases a valid tile position might be found

in the bounds of the board. We observed that we needed a board of size 18x18 to "place" tiles and to find valid positions on the edges, but during execution we evaluated the 14x14 squares of the actual board i.e. we did not increase the number of possible valid positions. As mentioned in 4.1.1 the specific representation allows us to determine a valid position in a single clock cycle. Further details about this process are provided in **??**.

The board was depicted by a 18x18x2 array. More specifically, the first two dimensions were set in such way, so as to represent each cell of the actual board while the last dimension indicated i) if the board cell is empty (00), ii) if the board cell is occupied by one of the players' tiles (01) or (10), iii) if the cell can be occupied by a tile due to a corner-to-corner contact (11).

4.2 High Level Modules

4.2.1 MetaData Module

In general the MetaData Module is responsible for finding a next move. This move will be later on evaluated to see whether it fits or not on the board. More specifically there are two controllers in the MetaData Module, the Address Generator and the Availability Controller. The Address Generator, either receives a Start Signal or a New Tile signal. The difference between the two is that the first will set the tile address to the beginning i.e. the address of the last tile in the tile memory since we want to traverse tiles from bigger to smaller, whereas the latter will reduce the previous tile address by one in order to move on to a next tile. Requesting a New Tile either means that there are still certain rotations of a tile that need to be evaluated and the respective tile address should be produced or that an entirely new tile should be examined. In order to distinguish each case the Address Generator "remembers" the previous rotation of the tile and knows the total number of rotations for the specific tile. If the previous rotation equals the total number of rotations, then the Address Generator produces a Put Back And Find Next signal to the Availability Controller. Otherwise, it produces the next address by subtracting 1 until all rotations have been examined. The role of the Availability Controller is simple. It receives as input the Availability Register of the current player, the Tile Number that is being evaluated, as well as a signal coming from the Address Generator. If the Address Generator outputs a Put Back And Find Next signal, the Availability Controller sets position Tile Number of the Availability Register with value '1' and finds the immediate next tile that is still available in the game and has not been evaluated and sets its position with value '0'. Note that to evaluate whether there is any tile left in the Availability Register, the Availability Controller requires at most 21 clock cycles (the size of the Availability Register). Once the Availability Controller has found the next tile it outputs its value to the Meta Data module. The Meta Data Module accesses the Blokus Rotation memory. The received tile number is the input address of the Blokus Rotation memory and the output of the memory is a concatenation of the total number of rotations of the tile with its first location address in the Blokus Tile memory. These values are then sent to the Address Generator in order to start producing tile addresses for the new tile until it has been also completely examined. Eventually, the Meta Data module produces the tile address that is received by the Move Module. The Meta Data Module requires at most 21 clock cycles by the Availability Controller, one clock cycle by the Meta Data Module and 3 clock cycles by the Address Generator to produce an output. The respective architecture diagram is presented in 10.

4.2.2 Move Module

The Move Module evaluates if an available tile fits in any position on the board. To be more precise the Move Module receives as input the current board, player, as well as the tile produced by the Meta Data Module. The following two tasks are executed in parallel. The first task concerns configuring the input board in such way so that given a row, a column and a tile it can be evaluated in a single clock cycle if the tile fits that position or not. More specifically, the input board is expanded from size 14x14x2 to 19x19x2. The core 14x14 board remains as it is and the new cells are padded with value "10". The Init Module receives the padded board, finds all valid corners on it and creates a new 19x19x2 board where each valid corner is represented by value "11". The second task uses the tile address produced by the Meta Data Module to

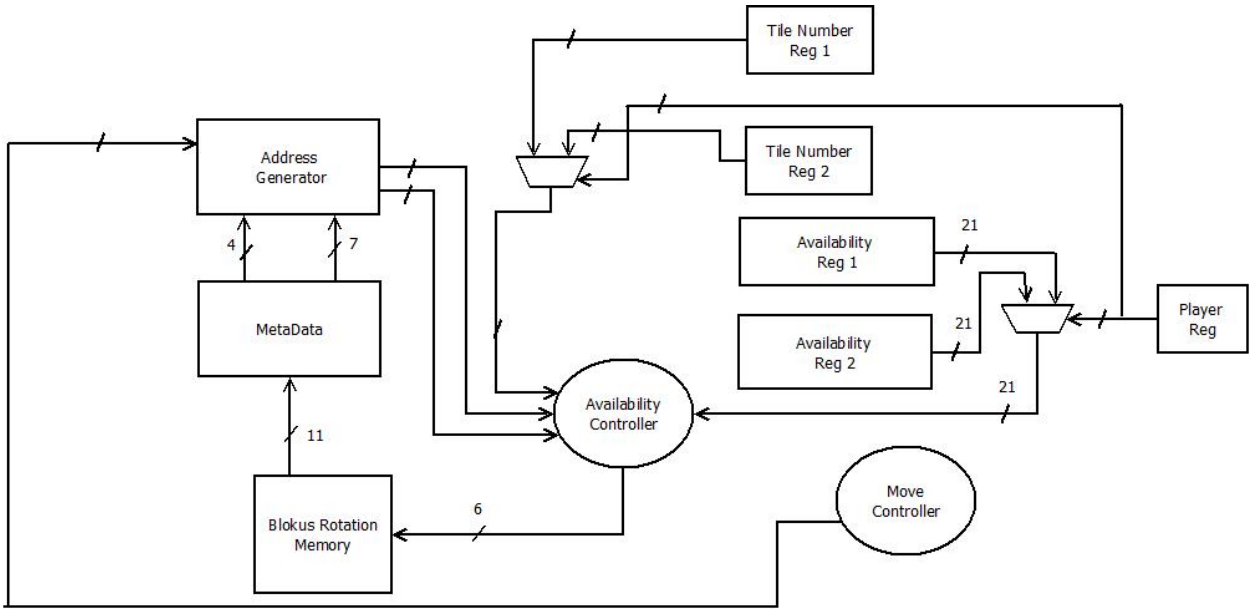

Figure 10: Architecture of Meta Data Module.

access the Blokus Tiles memory. Each address of the Blokus Tiles memory contains an entire tile, so that we only need one clock cycle to retrieve a tile. However, in memory the tile remains in a raw form (length 98 bits). The Mem To Tile module receives the raw form of the tile and produces a 7x7x2 representation of the same tile. This represenation allows comparing and placing tiles on the board, since the board has a similar representation. The above processes are completed in a single clock cycle, due to the fact that they are done in parallel. The output board and tile of the Init and Mem To Tile modules respectively, along with a Start signal coming from the Move Controller are received by the Move Module. The Move module starts from a given row and column and traverses each position of the board. In each clock cycle it outputs a different row-column pair that is received by the Valid Move Module. Given a tile, a board, a row and a column the Valid Move module determines in a single clock cycle if the specific move is valid or not by using the representation of the tile (cells with "10","01" and "00") and the "11" cells on the board. If the move is valid, The Valid Move module outputs a signal to the Move module and the Move Module outputs the move (row, column) along with a Finish signal. Then, the valid move is placed on the board by the Place tile module and the new board is stored in the Board Register. Note that depending on the current player, we need to flip the board that is inputted in the Init module and the tile that is inputted in the Place Tile module. The respective architecture diagram is presented in 11.

4.2.3 Push Module

Once the Move controller has received a Finish signal from the Move module it needs to determine whether Push or Pop should take place according to the Minimax algorithm. Note that the number of moves in the Minimax stack should not exceed the predefined Minimax depth. In our case this depth equals 3 due to the complexity of the Blokus Duo game. The number of elements in the stack is stored in the Stack Counter register. If a new move is to be placed and the number of elements in the Minimax stack is less than 3, Push takes place and the Push Module is activated. Prior to the activation of Push, the updated board and availability registers are inserted in the Evaluation Score module. The Evaluation score module computes the different parameters of the evaluation function (number of corners and new score after the new move has been placed). Then, these scores are summed by the Final Sum module where the weight of each parameter is also taken into account. We only use weights that are a power of 2 to exploit the shift operator and

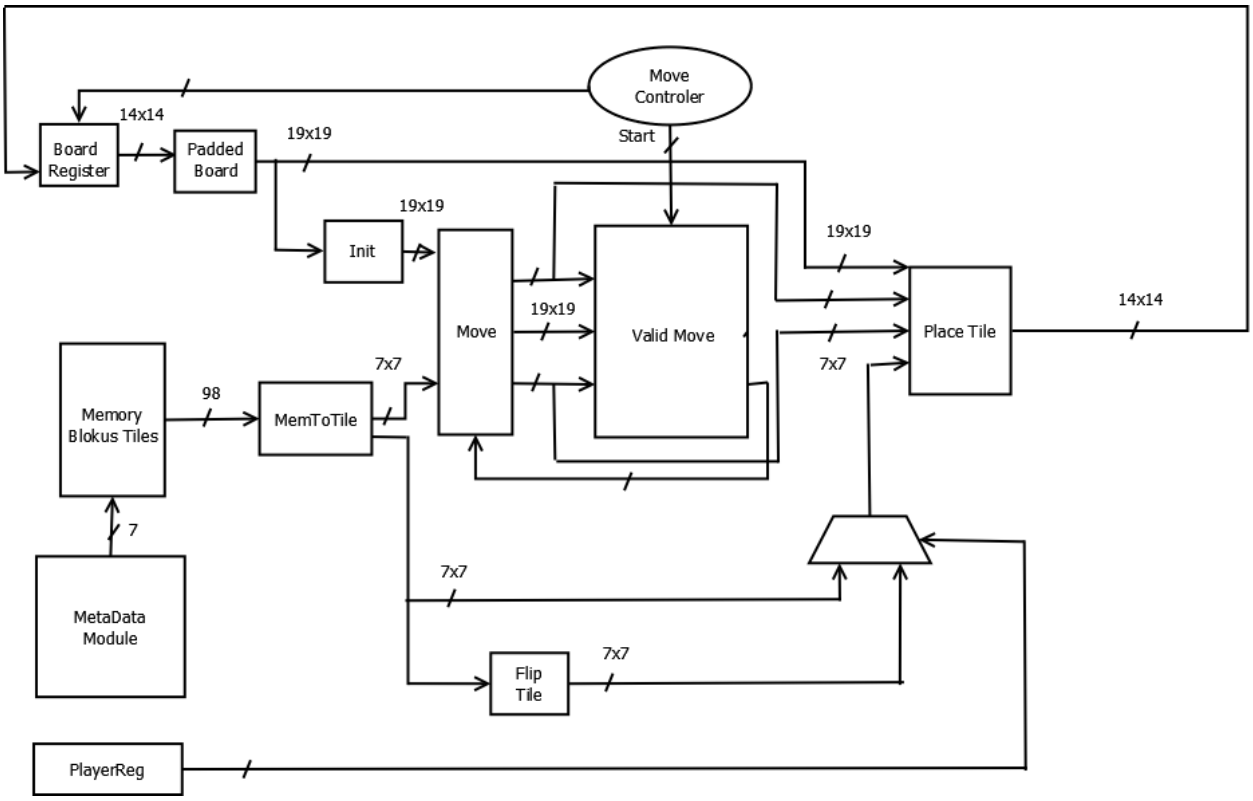

Figure 11: Architecture of Move Module.

avoid multiplication. Once, the Start signal reaches the Push controller the evaluation function has already computed the Minimax score. Therefore the concatenation of the tile number, rotation, row, column, player and score is inserted in the Stack and the respective memory address is equal to the number stored in the Stack Counter register plus 1. We need to keep this information about the newly added move, due to the fact that in a later phase of the Minimax algorithm the same move will be removed from the stack and undone in the game. Once the appropriate data have been stored in the stack the Push controller outputs a Finish signal and the Move controller updates the Stack Counter register with the new number of elements in the stack. At this point the Move controller also requires a new move from the Meta Data module and all the above processes are repeated until all moves have been evaluated. In our implementation the Push module requires 7 clock cycles to complete. The respective architecture diagram is presented in 12.

4.2.4 Pop Module

The Pop Module is only activated when a new move is to be placed into the stack, but the number of elements in the stack exceeds the depth of the Minimax algorithm. In such case the Move controller activates the Pop controller and the following procedure takes place. First, the score of the last item in the stack is compared to the score of the move in the Best Move register by the Max Evaluation module. The Best Move register contains the move that has so far produced the highest evaluation score and therefore is considered the best. The comparison of the two moves leads to determining the new best move (might remain the same) and this move is stored in the Best Move register. Then the evaluated last move in the stack is removed in order for the new one to enter. Furthermore, the deleted move should be also removed from the board and availability register (a complete undoing of the move). These actions are performed by the Undo Move module that receives as input the information of the deleted move, the board from the Board Register and the appropriate

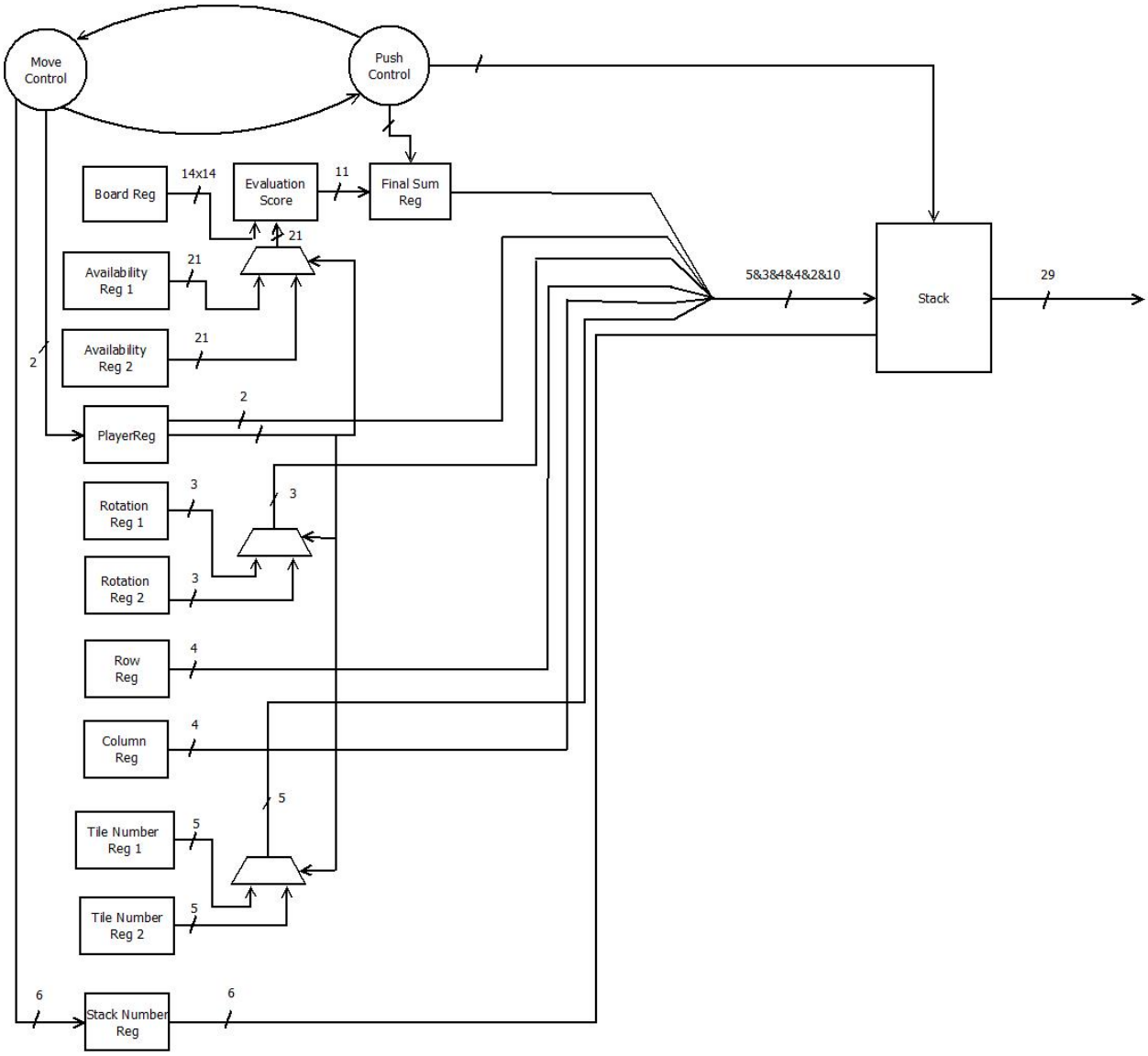

Figure 12: Architecture of Push Module.

Availability Register and restores these components to their previous status. Once the last move is deleted from the stack and has been undone in the game, the new move can take place. Thus, the Move controller waits for the Pop controller to output a Finish signal and then allows the Meta Data module, the Move module and the Push module to perform their actions. In our implementation the Pop module requires 17 clock cycles to complete. The respective architecture diagram is presented in 13.

4.2.5 Evaluation Module

The Evaluation Module does not present any interest in terms of design. However all Minimax-based implementations use an evaluation function (utility function) that determines the quality of the results. Scientific groups study different evaluation functions for months or years before reaching the desired result. Although we desired to create a competitive Blokus Duo player, due to limited time we applied a simple

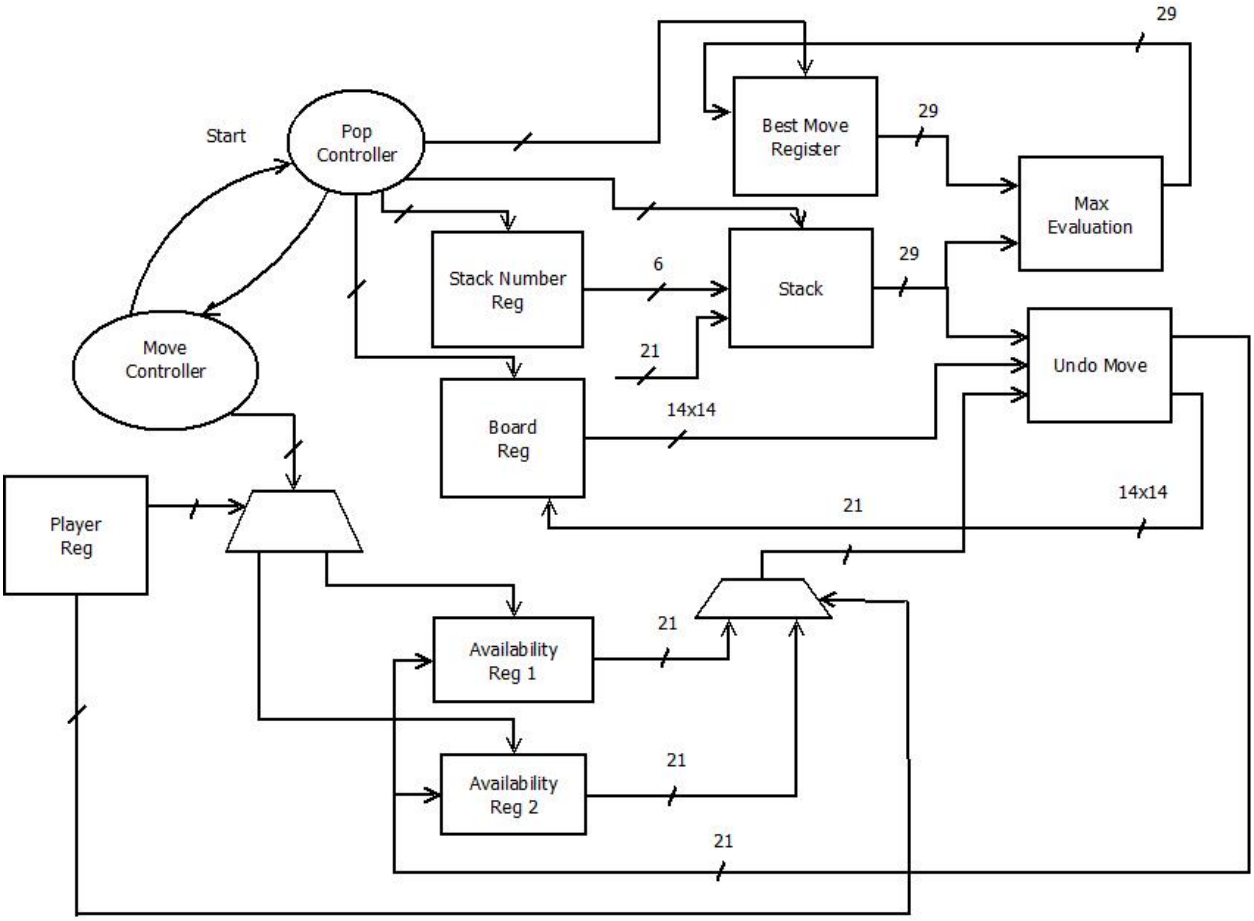

Figure 13: Architecture of Pop Module.

evaluation function without providing further enhances. More specifically, the evaluation function is executed whenever the algorithm reaches a leaf node or a certain predefined depth of the Minimax tree. It requires as inputs all parameters that are in the evaluation function. In our case we only use properties of the piece (tile) that is to be placed, as it is not in the scope of this thesis to create an optimal Minimax-based player. More specifically the evaluation function is the following:

$$value = 0.5 * a - 0.5 * b + 0.7 * c \tag{1}$$

where, a is the incremental number of the Minimax player's corners, b denotes the reductive number of the opponent's corners and c shows the number of unit squares of the piece.

4.3 Exploiting Parallelism and Efficient Resource Utilization

In 3 we analyzed the profiling of the software code and concluded to certain parallelization opportunities. Therefore in this project not only did we implement a hardware-based Blokus Duo player, but we also dedicated most of our time in finding effective solutions to exploit all parallelization opportunities. This section presents in detail the implementation of the "tricks" we used in order to make the hardware implementation faster based on the software profiling results.

4.3.1 Tile Fitting

The software-based approach requires searching a 5x5 area on the board in order to determine whether a tile fits in a specific position or not. Considering that the board comprises 14x14 = 196 positions it is a foregone conclusion that the software evaluates in total 5x5x196 = 4900 positions before outputting a valid position. However in hardware the 5x5 area searching can happen in one clock cycle as mentioned in 3, given that prior to the tile evaluation we have found all available corners in the board that are represented by "11" as shown in Figure 14. Thus in hardware we only examine 196 board positions instead of 4900.

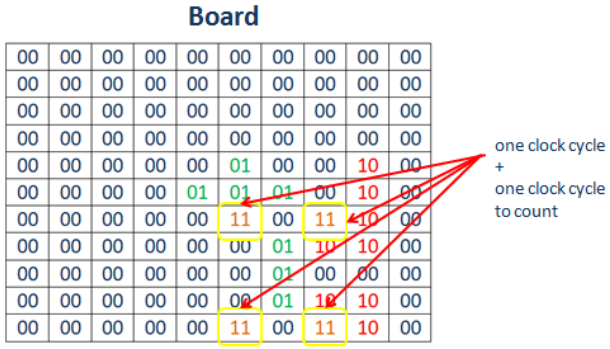

Figure 14: Depiction of corners on the board.

The above tile fitting is executed by the Valid Module and is achieved in the following way. We evaluate in parallel each of the 7x7 cell of the tile. Note that in the hardware implementation we used a 7x7 tile representation as mentioned in 4.1.1. For each cell of the tile we examine the following:

- If the value in the tile cell equals "01" then the respective board cell can neither have value "01" nor "10" as shown in Figure 15 since "01" represents the actual tile.
- If the value in the tile cell equals "01" then at least one board cell that overlaps with "01" should have value "11" as shown in Figure 16. Remember that "11" indicates a corner in the board and that a to be placed tile should have at least one corner-to-corner contact with a same colored tile.
- If the value in the tile cell equals "10" then the respective board cell cannot have value "01" as shown in Figure 17. The purpose of this evaluation is that an edge-to-edge contact with a same colored tile is prohibited.
- If the value in the tile cell equals "00" then the respective board cell can have any value as shown in Figure 18 as these cells do affect the validation result....

Note that the representation of each tile i.e. which cells have value "01","10" or "00", as well as the representation of the board i.e. which cells have value "01", "10", "00" or "11", are crucial in order to parallelize the $check/_{m}ove()$ function described in 3.

4.3.2 Calculating corners

One of the most time consuming functions in the software based player was finding the corners of the board. This is because the evaluation function requires computing all the corners in the board, once for Player 1 and once for Player 2 and this value is calculated for every terminal node (1012 terminal nodes for Minimax depth 3). In addition, finding corners is also necessary for the placement of valid tiles. However, in software this variable requires examining 14x14 board positions. Taking into account that this parameter addresses both players and is computed for nearly 1012 nodes, it is essential that we speed up this process and this is feasible, given that there are no dependencies between the board cells. The above process is executed by the

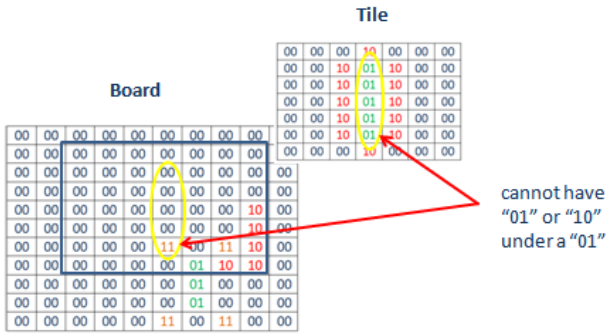

Figure 15: Placing a tile cell with value ”01”.

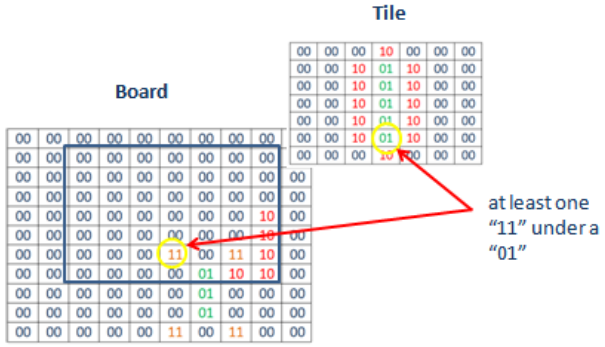

Figure 16: There should be at least one corner when placing a tile.

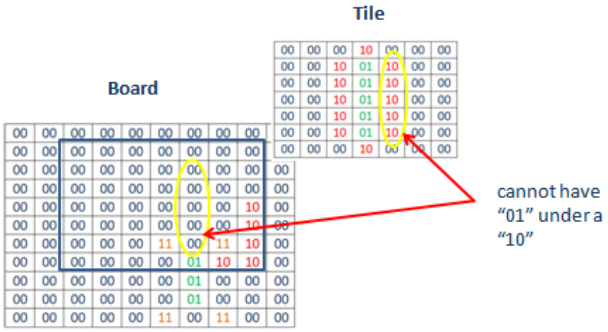

Figure 17: There should be no edge-to-edge contact when placing a tile.

Init Module by considering all posible cell sequences that create a corner. Let us assume that x and y are the row and column coordinates respectively. If we want to find a top left corner and (x,y) is the to be corner cell, then the sequence of cells (x-1,j-1),(x,y-1),(x-1,y),(x,y) can be one of the following: 1) ”01000000” 2) ”01100000” 3) ”01001000” 4) ”01101000”

Note that the total number of possible corners equals 64 due to the fact that we separately need to consider the borders of the board. The final outcome of our implementation is that we radically reduce the time required to find all corners since we only need a single clock cycle in contrast to software which traverses sequentially the board.

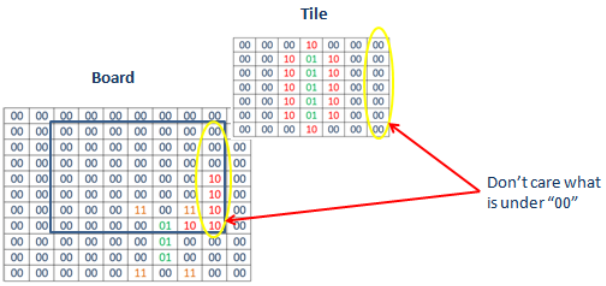

Figure 18: The value of the respective board cell is indifferent.

4.3.3 Evaluation function

We have already mentioned the importance of having an evaluation function. Although due to limited time we did not design an optimal evaluation function, still for the parameters that we took into account we produced a fast implementation. More specifically, we mentioned above how we computed the number of corners (two of the evaluation function factors). Calculating the new score is trivial, since we only have to increase the previous score by the number of cells of the tile that was just placed. Therefore we do not need to traverse the whole availability register of the player (21 clock cycles) to compute the new score. Moreover, if we are provided with the appropriate resources, then in hardware we can compute in parallel all factors of the evaluation function. Even if the evaluation function comprises many and complex parameters, a significant speed up can be achieved due to parallelization.

4.3.4 Do and Undo functions

Last but not least we tried to minimze the resource utilization in our design. More specifically, in similar hardware implementations over and above to the move information (tile, rotation, row, column, score) the previous move is also stored, in order to store the sequence of moves. This implies that the previous board and availability registers should be also saved which adds a significant resource overhead. In our design we implemented an Undo module instead of storing the previous move. The sequence of moves is always known via the stack since we add sequential moves in the stack. In particular, the first move is located in the bottom of the stack and the last is on the top. Once we have added the sequence of moves we want into the stack, we use the reverse order to remove them, which renders the undo function trivial. In contrast to the naive implementation which requires storing approximately 269 bits in each Minimax round, we only store 29 bits (tile,rotation,row,column,player,score).

5 Results

The FPGA Blokus Duo player performance was estimated using a time measurement counter and the number of push and pops of the algorithm. The same algorithm software was developed using C language and executed on an Intel Core i5-3317U 1.7 GHz dual-core processor. The hardware-based Blokus Duo agent was developed and tested under Xilinx ISE 13.4. The resources it occupies on the Xilinx Virtex-5 XC5VLX330T FPGA Board are shown in 19. Results show that the operation of an FPGA Blokus Duo player is always faster than that of personal computer with 20x speedup.

We did not measure the quality of the move selection of our Blokus Duo agent, since it was not in the scope of this project. However, the factors computed in the evaluation function are used by the majority of Minimax Blokus Duo players.

Resources	Used	Available	Percentage
Clock Frequency	70 MHz	-	-
Number of Slice Regs	1624	207360	1%
Number of Slice LUTs	37613	207360	18%
Number of Occupied Slices	12643	51840	24%
Number of BlockRAM/Fifo	5	324	1%

Figure 19: Resource utilization for our hardware-based Minimax Blokus Duo implementation

6 Conclusion - Future Work

In this technical report we described the full implementation of the FPGA acceleration of a Minimax Blokus Duo player. It was not in the scope of this project to create a competitive player and therefore we did not emphasize on optimizing the Minimax evaluation function. Our main goal was to identify potential parallelism and exploit this opportunity with an effective hardware implementation. To begin with, the operation speed of the FPGA Blokus Duo solver which was implemented on a Virtex-5 XC5VLX330T FPGA board becomes 20 times faster than C-based software operation under the same algorithm. Moreover, we succeeded in exploiting all opportunities for parallelization, thus achieving high performance in terms of speed as well as a satisfactory clock frequency of 70 MHz. In order to do so we performed compiling of the respective software source code, studied the results, developed different ideas and implemented the ones that yielded the best performance. Finally, we could not overpass the bottleneck described in 3.4 since an arbitrary number of cycles is requires in order to find where and if a tile fits in some position of the board.

The time for completing a project is always too short for implementing all ideas that arise during the work. At the end, three of them are outlined as outlook for future work. Blokus Duo is a relatively new game and few heuristics have been suggested that improve a player's performance. Therefore, we are interested in finding and implementing an evaluation function that is proven to be effective. Furthermore we intend to find an efficient way to find valid moves that will not require an arbitrary number of clock cycles, in order to benefit the most from parallelism offered by the Minimax algorithm. We could also create players based on Minimax variations such as Negamax, Expectimax or Minimax with alpha-beta pruning. Finally, we have acquired valuable knowledge about critical points and tactics of the game and we would like to create a Blokus Duo player that is competitive against any optimized Blokus Duo agent.

References

[1] P. Malakonakis, M. Smerdis, E. Sotiriades, and A. Dollas. An fpga-based sudoku solver based on simulated annealing methods. In *Field-Programmable Technology, 2009. FPT 2009. International Conference on*, pages 522–525. IEEE, 2009.

[2] S. J. Russell, P. Norvig, J. F. Canny, J. M. Malik, and D. D. Edwards. *Artificial intelligence: a modern approach*, volume 74. Prentice hall Englewood Cliffs, 1995.

[3] M. Smerdis, P. Malakonakis, and A. Dollas. Carlothello: An fpga-based monte carlo othello player. In *Field-Programmable Technology (FPT), 2010 International Conference on*, pages 515–518. IEEE, 2010.