The TU Delft Sudoku Solver on FPGA

Kees van der Bok #1, Mottaqiallah Taouil #2, Panagiotis Afratis #3 , Ioannis Sourdis #4

Computer Engineering
Delft University of Technology
The Netherlands
[1] C.vanderBok@student.tudelft.nl,
[2] M.Taouil@tudelft.nl,
[3] P.Afratis@student.tudelft.nl,
[4] I.Sourdis@tudelft.nl

***Abstract*—Solving Sudoku puzzles is a mind-bending activity that many people enjoy during their spare time. As such, for those being acquainted with computers, it becomes an irresistible challenge to build a computing engine for sudoku solving. Many sudoku solvers have been developed recently, using advanced techniques and algorithms to speed-up the computation. In this paper, we describe a hardware design for an FPGA implementation of a sudoku solver. Furthermore, we show the performance of the above design for solving puzzles of order N 3 to 15.**

I. INTRODUCTION

Using an FPGA to solve a sudoku puzzle is an interesting challenge and valuable test case for general purpose algorithm execution on FPGA. Recently, executing general purpose applications on FPGA has grown in popularity and has proven to be more efficient in many cases. Although FPGAs are much slower then GPPs, regarding the operating frequency, better performance can be achieved exploiting high degree of parallelism and customization, as well as the ability to keep data local. Before designing our sudoku solver we considered the available algorithms and how they could be mapped to an FPGA. Most algorithms turned out to have excessive resource requirements, either in memory size or in logic. Our design choice is a brute-force algorithm, the only design possibility that fitted in the target FPGA device (Virtex2P-30). We expected the brute-force algorithm to be faster than a software version, because of the more efficient way in which the valid symbols for a cell can be determined. Although we have improved the basic step of algorithm, the symbol selection, we have not solved the exhaustive nature of the brute-force solver (i.e. the solver may have to go through all the valid symbol assignments). The previously described issue causes the solver to become intractable for hard or large sudoku problems. Therefore, we conclude that the brute-force solver needs to be enriched with techniques that prune the search space. We explored the benefits of filling empty cells in a particular order. Although the former reduces the solving time, it would not make the hard and large problems tractable.

The following sections describe our brute-force sudoku solver implemented on an FPGA. The algorithm and design are explained and the performance is analyzed.

II. SUDOKU SOLVING

Automatic solving of sudoku puzzles can be done in various ways. There are a few well-known problems, for which algorithms exist, showing similarity to the sudoku problem. Solving a sudoku is, foremost, a constraint satisfaction problem, but could also be regarded as an exact-cover problem, graph-coloring problem or binary-satisfaction problem.

Besides converting the sudoku problem to a known problem, sudoku solvers that mimic the human solving scheme have been developed. This solving method is usually referred to as the elimination or solving-by-logic method. In essence the method is very closely related to the graph-coloring problem. The elimination method uses logic reasoning based on the constraints of the sudoku puzzle to exclude the symbols that can not be placed in a certain cell. The crux of the method is that if one candidate remains, when all others have been proven infeasible, that symbol can be filled in. Usually these solvers implement rules derived from the pen-and-paper methods. The algorithm requires the possible candidates to be kept in memory. This can be done by assigning to each cell of the puzzle a bitmap in which each bit represents a certain symbol (e.g. bit 0 represents symbol 1 etc.). A set bit identifies that the symbol represented by that bit is a candidate for the cell the bitmap relates to. Elimination rules must be applied until one candidate remains (i.e. one and only one bit remains set in the bitmap). The cell can then be filled with the symbol represented by this bit. An advantage of this algorithm is that it is suitable for parallel execution. Unfortunately the elimination algorithm requires more memory than available on the FPGA. Storing the bitmaps requires $N^4 * 225$ bits, for each cell a 225-bit bitmap. For $N = 15$ we need $15^4 * 225 = 11390625 \approx 12Mb$ This exceeds the available memory of the V2P30 FPGA, which offers only $1.4Mb$ of Block RAM.

There are other, less memory demanding, methods of storing the sudoku information. However, these methods decrease the amount of information available and therefore weaken the strength of the elimination method considerably. Another issue we discovered when analyzing this method is that the algorithm is not complete. Therefore, the algorithm is not

capable of solving hard sudoku instances.

Another algorithm we considered is the dancing-links algorithm, which is an elegant algorithm solving the exact cover problem. Despite the elegance of the algorithm, it turned out to be unfeasible to implement this approach on the FPGA. The latter because the algorithm's memory requirements exceed the memory available on the FPGA.

A method for converting the sudoku problem to a binary-satisfiability problem has been proposed in [1]. Using such a description the sudoku could be solved using an FPGA based binary-satisfiability solver as for example described in [2]. An FPGA based binary-satisfiability solver requires an excessive amount of logic and is therefore only practical for small problems. The solver required to solve a sudoku puzzle following the above approach requires more than the available FPGA logic resources.

The most straightforward method of solving a sudoku is by brute force. Brute-force sudoku solvers fill-in cells speculatively taking into account the constraints that apply to a sudoku. Cells are filled in until a conflict is discovered. On conflict the solver clears the filled-in cells until it has returned to a cell that has untried candidates. The brute-force method performs an exhaustive search which will always find the solution, if one exists. However, finding a solution might require unacceptable time for large puzzles. Heuristics can be used to minimize the search space. The brute-force method could be efficient when combined with an elimination method.

III. The Algorithm

The algorithm used in our design is similar to the brute-force method described in the previous section. In this section, we elaborate in more detail the algorithm and its FPGA implementations.

We use bitmaps to check valid symbols for a certain cell. These bitmaps represent symbols present in a unit. A unit is a row, column or block. Bitmaps are maintained per row, column and block. Table II shows the row bitmaps of the sudoku depicted in Table I. Using the bitmaps we determine a valid symbol in constant time. The algorithm chooses the symbol with the lowest numerical value among the candidates. The puzzle is traversed in row-major order until a cell is encountered that has now valid candidate symbols. In case, during the solving process a cell is encountered that can not be assigned a symbol (i.e. every possible symbol for that cell conflicts with the sudoku constraints), the solver needs to backtrack to a cell that has at least one possible alternative assignment. The initial bitmaps are constructed when the sudoku puzzle is received from the RS-232 unit and stored in memory. While solving the puzzle the bitmaps are continuously updated. This is achieved by using bitwise operations, which set the appropriate bits while writing, while they clear the bits while backtracking.

IV. Design

The design is composed of four main parts (Figure 1): control, communication, storage and processing. Each of these modules are described in the following subsections.

TABLE I
AN EXAMPLE SUDOKU

6	0	0	0	0	0	0	0	3
8	0	0	4	5	6	1	0	0
0	5	0	0	0	0	0	0	0
0	1	5	9	0	0	3	0	0
0	0	0	0	1	0	0	0	0
0	6	0	0	8	0	5	0	7
0	0	2	0	0	0	0	0	0
9	0	0	0	0	1	7	4	0
4	7	0	0	9	0	0	0	6

TABLE II
ROW BITMAPS

row	Symbol								
	1	2	3	4	5	6	7	8	9
0	0	0	1	0	0	1	0	0	0
1	1	0	0	1	1	1	0	1	0
2	0	0	0	0	1	0	0	0	0
3	1	0	1	0	1	0	0	0	1
4	1	0	0	0	0	0	0	0	0
5	0	0	0	0	1	1	1	1	0
6	0	1	0	0	0	0	0	0	0
7	1	0	0	1	0	0	1	0	1
8	0	0	0	1	0	1	1	0	1

A. Main Controller

The overall orchestration of the sudoku solver is done by the main controller which is a simple state machine. This state machine resembles the overall state of the sudoku solver. Based on this state access to the storage module is either granted to the communication interface module or the processing module.

B. Communication

Communication between the board and the host PC uses the well known RS-232 protocol. RS-232 sends the data bit by bit. The nature of RS-232 is asynchronous. Each, serially transmitted, byte will start with a start bit and end with a stop bit. The receiver uses the start and stop bits to synchronize. Two simple state machines perform the communication, one for transmitting and one for receiving. A third state machine is used to orchestrate the higher-level procedure of receiving the puzzle and transmitting the solution after the puzzle has been solved.

C. Storage

All the information regarding the sudoku is maintained within the storage module. Besides storing the sudoku and the bitmaps, the storage module calculates the checksum and updates the bitmaps. Writing or reading from or to the storage module is kept simple. All necessary processing and calculations are hidden from other modules. Two control signals are used to select one of the four operating modes in which the storage module can operate. The operation modes are described in Table III. Neutral reads are used by the processing module to read the symbol and bitmaps that are related to the addressed cell. The clear mode is used for backtracking. In this mode the symbol in the addressed cell will be cleared

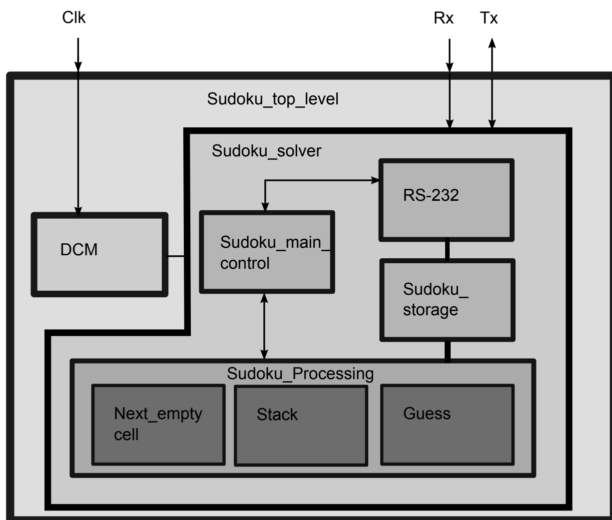

Fig. 1. Top-level View of the Design

as well as the related bits in the bitmaps. The write mode is used to write symbols to the storage module and is used when the initial puzzle is stored or when cells are filled-in. The destructive read is applied when the solved puzzle is read out. In addition, the destructive read clears the bitmaps related to each cell. That is because the bitmaps need to be cleared before the next puzzle is read in. Performing the bitmap clearing while reading out the solution saves valuable time. Whenever the global reset signal is asserted the communication interface will get to a state in which the bitmaps will be cleared as well. In this state every memory location is read once in destructive mode.

TABLE III
STORAGE OF THE ORIGINAL SUDOKU

Mode	Description
Neutral Read	Reads symbol and bitmaps related to the address cell
Destructive Read	Same as neutral read but clears the bitmaps
Write	Write symbol and updates bitmaps
Clear	Clears symbol and updates bitmaps

The storage module contains five memories, one for the sudoku and four for the bitmaps (rows, columns, blocks, and occupied cells). The bitmaps are stored in four separate memories allowing them to be read in parallel. Updating a bitmap requires three steps, namely reading the bitmap, modifying it and writing it back.

Two others modules worth mentioning are the checksum calculator and the block calculator. The former module computes the checksum of the puzzle while the latter determines, based on the row, column and order of the puzzle, which block is addressed. The block calculator is used to address the proper block bitmap.

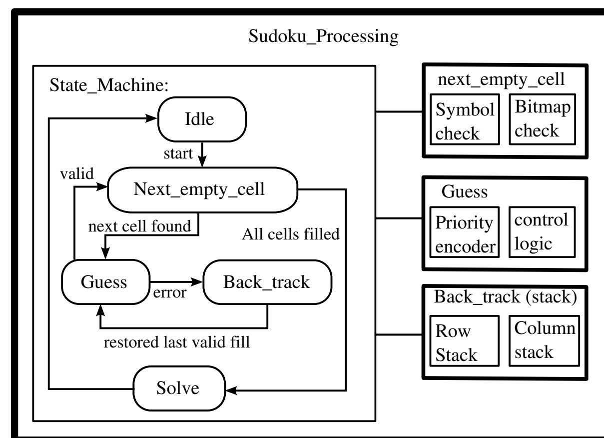

Fig. 2. The Sudoku Processing Unit

D. Processing

Figure 2 is a simplified representation of the processing unit. Although the low-level details are kept out in this figure, it clearly shows which steps are involved in the solving process as well as the flow of it. After being enabled, the processing unit will go to the *next-empty-cell* state. In this state, the next empty cell is determined; this is performed by checking the bitmaps representing the occupied cells. Each bit of the bitmap represents a cell, only the bits representing occupied cells are set. Finding the first not-set bit in a bitmap is done using a priority encoder. After having selected the cell, the state machine proceeds to the *guess* state in which a valid symbol for the selected cell is determined. Based on the puzzle depicted in Table I a valid symbol is determined as follows. The first empty cell in this sudoku is (1,2) (i.e. first row, second column). Table IV shows the bitmaps of the row, column and block of the corresponding cell. Performing a bitwise *OR* of these three bitmaps will give the candidate symbols that could go in the cell (i.e. these symbols are represented by the '0' in the result vector). From the result vector we conclude that 2, 4 and 9 are valid symbols for cell (1,2). We select the first option which is 2. Choosing the first candidate, instead of randomly selecting one, saves logic and memory since we do not have to keep track of which symbols have been tried. Furthermore for choosing the first option we only need a priority encoder. When filling in a *2* in cell (1,2) we need to update the bitmaps; this, however, is done by the storage module and is of no concern to the processing module. Whenever a cell is filled the address of the cell is pushed on stack to memorize the backtracking path. The process of finding an empty cell and filling it repeats until we solve the entire puzzle or until we reach an empty cell that can not be filled due to a conflict. In the latter case, the partially filled sudoku is not valid forcing a backtrack operation. In the *backtrack* state the last visited cell is popped from the stack, the symbol in that cell is read and cleared simultaneously. That read symbol is stored, in doing so the guess process will re-fill the cell only with symbols greater than the one causing the conflict. From the *backtrack* state the processing unit returns to the *guess* state from which

it will backtrack one more cell (in case there is still a conflict) or start filling in empty cells again. Eventually the algorithm fills-in the last empty cell after which the puzzle is solved.

TABLE IV
CANDIDATE SELECTION

	Symbol								
	1	2	3	4	5	6	7	8	9
Row	0	0	1	0	0	1	0	0	0
Column	1	0	0	0	1	1	1	0	0
Block	0	0	0	0	1	1	0	1	0
Result	1	0	1	0	1	1	1	1	0

V. RESULTS

We synthesized and prototyped the design on a Xilinx Virtex2P-30 FPGA. The design occupied 110 BlockRAMs (80% of the available ones), 2,436 Slices (17%), while the operating frequency was 50 MHz limited by long wires required to interconnect our logic with the distributed BlockRAMs. We used the benchmarks provided in [3] to evaluate the efficiency of our design. Our sudoku solver seems to work well for order $N = 3$ sudokus. However, the solver requires significantly more effort solving hard puzzles of order $N = 4$ and higher. The solver is able to solve order $N = 3$ puzzles which are classified as hard. For higher-order puzzles the solver can only solve easy instances. Harder instances take an excessive amount of time to be solved (not completed at least within an hour). Therefore we have not been able to measure the execution time for most of the benchmark puzzles. We tried to solve benchmark puzzle *10a* which the solver was not able to solve within 24 hours. A ten-run benchmark of our solver is depicted in Table V. This table shows the results for the puzzles our solver can solve within reasonable time only.

TABLE V
BENCHMARK RESULTS

	Benchmark Puzzles (puzzle dimension - type of benchmark)					
run	3-a	3-b	4-a	6-a	7-a	8-a
0	0.021153 s	0.012237 s	0.221498 s	0.114990 s	0.211481 s	0.096214 s
1	0.020691 s	0.012235 s	0.220870 s	0.115048 s	0.211223 s	0.096670 s
2	0.020642 s	0.012228 s	0.221676 s	0.115143 s	0.211264 s	0.096429 s
3	0.020732 s	0.012932 s	0.221710 s	0.115157 s	0.211662 s	0.096686 s
4	0.020885 s	0.012348 s	0.221437 s	0.115206 s	0.211501 s	0.096408 s
5	0.020728 s	0.012930 s	0.221729 s	0.115650 s	0.210956 s	0.096666 s
6	0.020788 s	0.012243 s	0.221397 s	0.115777 s	0.210929 s	0.096475 s
7	0.020715 s	0.012249 s	0.221544 s	0.115817 s	0.211370 s	0.096178 s
8	0.020952 s	0.012255 s	0.220798 s	0.115617 s	0.211575 s	0.096357 s
9	0.020926 s	0.012943 s	0.221133 s	0.115062 s	0.211468 s	0.096156 s
Avarage	0.020821s	0.012460 s	0.221379 s	0.115347 s	0.211343 s	0.096424 s
std. dev.	0.000156	0.000330	0.000337	0.000328	0.000249	0.000203

VI. PROPOSED IMPROVEMENTS

We have thought of various optimizations to accelerate the exhaustive search that our solver is performing. Initially, we planned to use a hybrid algorithm composed of an elimination algorithm and the brute-force algorithm. Starting with the elimination algorithm some blank cells might be found. Whenever the elimination algorithm gets stuck, the brute-force algorithm could be used to advance. The elimination algorithm can be used after every guess by the brute-force algorithm, reducing the search space considerable. However, strong elimination algorithms require a significant amount of information to be kept available. The memory usage of the elimination algorithms exceeds the memory offered by the target FPGA by far. We have been experimenting with elimination techniques that only required the bitmaps we have available. We concluded that such algorithm can only deduce the value of a cell in very trivial situations and would therefore be of no use. Another improvement we considered is having multiple brute-force processing units to operate on the puzzle in parallel. This would, however, have a negative impact on the performance because the processing units will interfere with each other which will in most cases not lead to a solution since branches, that might contain the solution, in the search tree could remain unexplored. An optimization we have actually implemented is to traverse the the rows based on the number of filled cells they contain. By visiting the rows in this order the probability of choosing the right path increases. Although, the technique showed promising results (i.e. speed-ups up to 30 times) for order $N = 3$ puzzles it did not help us in solving the hard instances within the time limit. We have ran this technique in simulation only, we failed to have it working on the FPGA before the Sudoku design competition deadline.

VII. CONCLUSION

The Brute-force technique seems to be a feasible method for solving sudoku puzzles. However, the technique is not applicable to hard instances or high-order sudokus. In order to improve the brute-force algorithm the search needs to be directed. A hybrid solver using both the brute-force and an elimination algorithm could lead to a significant decrease in the possibilities that need to be explored. However, we could not find an elimination method fitting the available resources. The brute-force algorithm we have implemented can find the next empty cell and determine a valid symbol in constant time. The former, is the prime improvement over a software version of this algorithm. However, it does not solve the exhaustive nature of the algorithm.

ACKNOWLEDGMENT

We would like to acknowledge the hosts of the design competition since we enjoyed this challenging exercise. Furthermore, we would like to thank all those who inspired us and gave us advice.

REFERENCES

[1] I. Lynce and J. Ouaknine, "Sudoku as a sat problem."
[2] I. Skliarova and A. de Brito Ferrari, "Reconfigurable hardware sat solvers: A survey of systems," *IEEE Trans. Comput.*, vol. 53, no. 11, pp. 1449–1461, 2004.
[3] Sudoku Benchmarks, "http://fpt09.cse.unsw.edu.au/comp/benchmarks.html."