Anexo I: Artículo publicado en las actas del FPT '09

An Initial Specific Processor for Sudoku Solving

Carlos González [#1], Javier Olivito [*2], Javier Resano [*3]

[#] *Computer Architecture Department, Complutense University of Madrid*
Madrid, Spain

[1] `carlosgonzalez@fdi.ucm.es`

[*] *DIIS-I3A, University of Zaragoza*
Zaragoza, Spain

[2] `creoenoli@gmail.com`

[3] `javier1@dacya.ucm.es`

Abstract—**In this article, we present a design of a Sudoku solver submitted to the FPT Design Competition. Using only the on-chip resources of a XC2VP30 Virtex-II Pro FPGA we have designed a specific processor that can solve Sudokus from order 3 to 11. This processor applies a Branch&Bound approach to explore the solution space. However, this solution space is too big, and frequently the time needed to solve the Sudokus is excessive. To attempt to improve the design we have implemented an equivalent software version, and add several heuristics that reduce the solution space to it. The results have shown that these heuristics can drastically speedup the search. Hence we have included a few of them in the processor: Singles, Hidden Singles, Hidden Pairs, Hidden Triplets and Hidden Quartets that are well-known in the Sudokus literature. This design can still be improved by including other heuristics like Locked Candidates or Naked Candidates. Moreover, it is possible to extend the design for larger Sudokus since all the modules are customizable, but since the on-chip memory resources are very limited, it would be needed to use an external DDR RAM memory.**

I. Introduction

Designing a Sudoku solver is a challenging problem for a digital logic designer. Several important mathematical concepts are hidden behind the scenes: combinatorics [1,2] used in counting valid Sudoku grids, group theory [3] used to describe ideas about when two grids are equivalent, and computational complexity with regards to solving Sudokus. This is nicely explained in [4,5].

The standard version of Sudoku consists of a 9×9 square grid containing 81 cells. The grid is subdivided into nine 3×3 boxes. Some of the 81 cells are filled in with numbers from the set {1,2,3,4,5,6,7,8,9}. The goal is to fill in the remaining cells guaranteeing that the nine digits are used once in each row, each column, and each box.

This standard version can be generalized for other sizes. The 9x9 square is in fact a Sudoku of order 3. A Sudoku of order N is an $N^2 \times N^2$ square grid, subdivided into N^2 boxes, each of size $N \times N$. In this case it must be guaranteed that the digits from 1 to N^2 are used once in each row, column and box.

There are many techniques to solve a Sudoku. Some of these are basics as Singles or Hidden Singles and others are more advanced as Locked Candidates, Naked Candidates, Hidden Candidates, X-Wing, Swordfish, etc. [6,7,8,9] However, these techniques are only heuristics that can reduce the solution space, but cannot guarantee that they find solution. Hence it is interesting to combine them with a trial-and-error approach that explores the remaining solution space once it has been narrowed with the previous techniques.

In this paper, we present the design and implementation of a specific processor that can solve Sudokus from order 3 to 11 applying the following techniques to reduce the solution design-space: Singles, Hidden Singles, Hidden Pairs, Hidden Triplets and Hidden Quartets. After these techniques the processor carries out a final design-space exploration using a Branch&Bound strategy. Our design has been implemented on a XC2VP30 Virtex-II Pro FPGA included in the XUP Development System [10]. This FPGA includes two embedded PowerPCs (not used in this design), 13969 Slices and 2448 Kb Block RAMs. We have developed our design using VHDL language for the specification and the Xilinx ISE 10.1.3 environment [11].

We do not consider that this design is completely finished. On the contrary, we have carried out and extensive analysis using an equivalent software version, and we have identified that the efficiency of this processor will significantly increase if more heuristics are included. As future work we will include at least Locked Candidates and Naked Candidates. And we will extend it to support Sudokus up to order 15. The current version cannot deal with Sudokus of order 12 or greater because we are only using the on-chip memory. However, if we include hardware support to use external memories it should be easy to extend the design.

The rest of the paper is organized as follows: Section II explains the implemented techniques and the outline of our solution strategy. Section III focuses on the proposed architecture. Section IV provides a report on the results of our benchmark evaluation. Section V shows an extensive analysis using a software version. Section VI explains our conclusions and indicates some lines for future work.

II. Implemented Techniques

In this section we will show the outline of our solution strategy. Briefly, the implemented techniques are:

A. *Singles*

When a cell has only one candidate, it can be safely assigned. And that candidate can be removed from all the other cells in the same row, column and box.

B. Hidden Singles

Very frequently, a candidate appears only once in a given row, column or box. This candidate can be safely assigned to that cell.

C. Hidden Pairs, Triplets and Quartets

If two cells in a given row, column or box contain a pair of candidates (frequently hidden among other candidates) that are not found in any other cells in that row, column or box, then other candidates in those two cells can be safely excluded. We have extended this idea for triplets and quartets.

D. Branch&Bound

This solution space exploration technique explores all possible valid options to fill the Sudoku. It starts assigning a feasible value to each empty cell, and whenever is impossible to assign a feasible value it returns to the previous cell and try another feasible value. Hence it generates a design exploration tree and analyses each branch until it detects that it generates a conflict, or until the branch produces the solution to the Sudoku.

These techniques are applied in the order above with the following exceptions: If the application of the Hidden Singles technique fixes a value for a cell, we will apply again Singles technique. If the application of the Hidden Pairs, Triplets and Quartets techniques eliminates any candidate for any cell, we will apply again Hidden Singles technique.

III. PROPOSED ARCHITECTURE

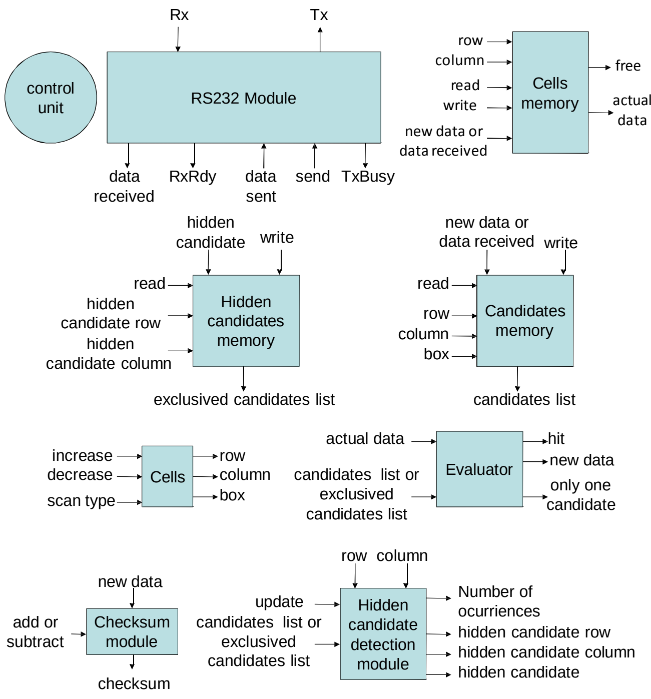


Fig. 1. Top architecture

Figure 1 shows the structure hardware needed to implement the Sudoku solver and the I/O communications. For data I/O, it uses a RS232 module and it stores the Sudoku in different memories. It also has a module for checksum calculation, a counter modified to obtain the row, column and box of the cells depending on the scan type (by rows, columns or boxes), an evaluator and a hidden candidate detection module. We will describe the most important elements in detail.

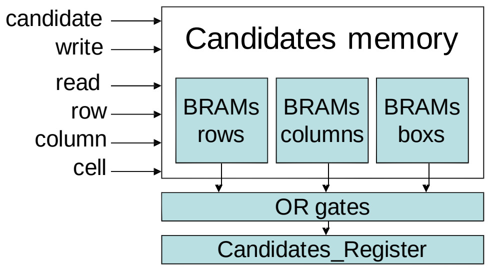


Fig. 2. Candidates memory

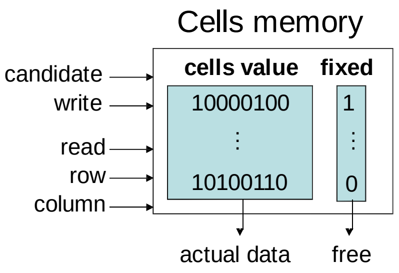


Fig. 3. Cells memory

Sudoku is stored in different memories to easily update his status at any time. The *Candidates memory* (see Figure 2) is composed by three memories where we have the possible candidates from each row, column and box, allowing us to get the cell's candidates in only one read operation. The *Cells memory* (see Figure 3) stores the value of the cell and if it is fixed or not. Finally, we use the *Hidden candidates memory* to store the candidates that are excluded for a particular cell after the execution of the techniques.

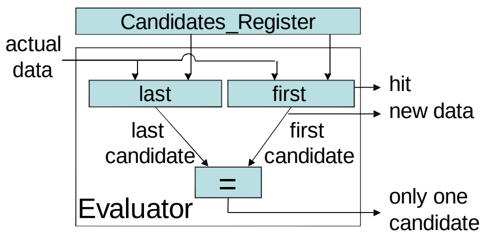


Fig. 4. Evaluator architecture

Evaluator module (see Figure 4) receives the candidates list and the actual data for a cell and it is able to give us the next possible candidate, if it exits, and if the cell has only one possible candidate. To parallelize the search of the first candidate and the last candidate, we divide the candidates list into parts and then we compare the partial results. This module is very useful for the Singles and Branch&Bound techniques.

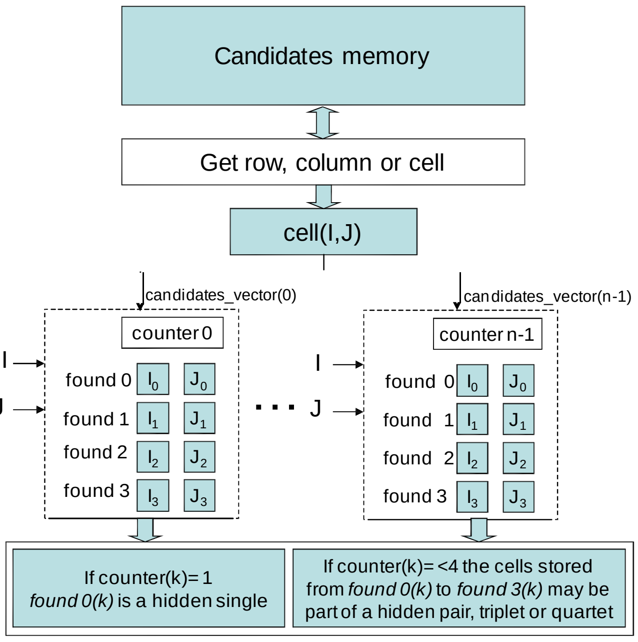

Candidates memory
Get row, column or cell
cell(I,J)
candidates_vector(0)
candidates_vector(n-1)
counter 0
counter n-1
I
J
found 0
found 1
found 2
found 3
If counter(k)= 1 found 0(k) is a hidden single
If counter(k)= =<4 the cells stored from found 0(k) to found 3(k) may be part of a hidden pair, triplet or quartet

Fig. 5. Hardware to look for hidden singles, and possible hidden pairs, triplet and quartets

Figure 5 shows the architecture of the *Hidden candidate detection module*. To fill the structure, we read sequentially the candidates for the cells that form a row, column or box and then, for the possible candidates his counter is increased and the row and column are stored. In this way, we obtain the number of occurrences of a candidate in a row, column or box and the cells where it appears. Therefore, if the counter for one candidate has a value of one, we have found a Hidden Single candidate, and if the counter has a value less than five this candidate may be part of a Hidden Pair, Triplet or Quartet.

TABLE I
RESOURCES USED FOR THE SUDOKU SOLVERS

Techniques	Maximum Order	BRAMs	Slices and Percentage	
Branch&Bound	15	29	1832	13%
Singles	15	29	2183	16%
Hidden Singles	11	134	7675	56%
Hidden Pairs	11	134	9187	67%
Hidden Triplets	11	134	1080	78%
Hidden Quartets	11	134	12265	90%

To conclude this section, Table I shows the maximum order we could solve and the resources used for a hardware implementation of the Sudoku solver according to the techniques have been added. The decrease of the maximum order from 15 to 11 is due to the necessary use of the *Hidden candidates memory* to store the eliminated candidates.

IV. EXPERIMENTAL RESULTS

The FPT Design Competition provides a set of benchmarks to evaluate the design, with two types of Sudokus, (a) and (b), and orders from 3 to 15. The maximum time allowed for solving a Sudoku of order N is defined as: $t_{max}=3•10^{-4}•N^6$ seconds. Table II shows the results, in terms of computation time, of them.

TABLE II
BENCHMARKS COMPUTATION TIME IN SECONDS FOR HARDWARE DESIGN

Order	Type (a)	Type (b)
3	0.0086	0.009762
4	0.024761	Not in time
5	0.060281	Not in time
6	0.117894	Not in time
7	0.224523	Not in time
8	0.058375	Not in time
9	0.300835	Not in time
10	0.322603	Not in time
11	0.381182	Not in time

Results show that our design can solve quickly Sudokus of type (a) up to order 11. However, it is still inefficient when it deals with Sudokus of type (b). Complex Sudokus have an extensive solution space and we need to implement more techniques to reduce it.

V. SOFTWARE IMPLEMENTATION

In order to improve the design we have implemented an equivalent software version, and we have added several techniques that reduce the solution space to it. In order to check which techniques are necessary to solve the different benchmarks, we have implemented a great number of techniques: all the techniques detailed above and an another additional set of the techniques showed in [6,7,8,9]. Next, we are going to expose a brief description of these additional techniques:

A. Naked Pairs

If two cells in a group (row, column or box) contain an identical pair of candidates and only those two candidates, then no other cells in that group could have those values and they can be removed from them.

B. Candidates Lines

Sometimes a candidate within a box is restricted to one row or column. Since one of these cells of the considered box must contain that specific candidate, the candidate can safely be excluded from the remaining cells in that row or column outside of the box.

C. Double Lines

Given two boxes (located in the same “line of boxes”), if a candidate can be placed only in two rows (or columns) for both boxes, and those rows (columns) are the same for both boxes, that candidate can be safely removed from those rows (columns) in the rest of boxes placed in the same “line of boxes”.

D. *Forced Chains*

We take cells with only two candidates and, firstly, we assume that the value of that cell is one of its candidates. Now, we apply the rest of techniques saving which candidates are removed and which values can be now safely placed.

Once no more candidates can be removed and no more numbers can be placed, we assume that the value of the cell is the other candidate, and we apply the same process again.

When it finishes, we compare the removed candidates and the placed values in both assumptions: Any removed candidate in both assumptions can be safely removed from the Sudoku, and any number placed in both assumptions can be safely placed in the Sudoku.

E. *Extension of some techniques*

Some of those techniques explained above are extensible for bigger Sudokus. Hidden Pairs technique can be extended to Hidden Triplets, Hidden Quartets, etc. In the same way, Double Lines technique can be extended to Triple Lines, Quadruple Lines, etc. Forced Chains technique can be applied to cells with three candidates, four candidates, etc.

We have developed this software version using C language and the program was executed on a 1,66GHz processor with 2GB RAM. Table III shows the results, in terms of computation time, for the software implementation and the needed techniques to solve each benchmark.

TABLE III
BENCHMARKS COMPUTATION TIME IN SECONDS FOR SOFTWARE DESIGN

Benchmark	**Computation Time**	**Applied techniques**
3a	0.003	A, B, C, D, E, F, G, H, N
4a	0.01	A, B, C, D, E, F, G, H, I, J, K, N
5a	0.038	A, B, C, D, E, F, G, H, I, J, K, L, M, N
6a	0.024	A, B, C, D, E, F, G, H, I, J, K, L, M, N
7a	0.522	A, B, C, D, E, F, G, H, I, J, K, L, M, N, P
8a	3.210	A, B, C, D, E, F, G, H, I, J, K, L, M, N, P
9a	10.402	A, B, C, D, E, F, G, H, I, J, K, L, M, N, P
10a	48.520	A, B, C, D, E, F, G, H, I, J, K, L, M, N, P
11a	0.652	A, B, C, D, E, F, G, H, I, J, K, L, M, N
3b	0.009	A, B, C, D, E, F, G, H
4b	1.81	A, B, C, D, E, F, G, H, J, N
5b	56.133	A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P

A = Hidden pairs	B = Naked pairs
C = Hidden singles	D = Singles
E = Hidden triplets	F = Naked triplets
G = Candidates Lines	H = Double Lines
I = Triple Lines	J = Hidden Quartets
K = Naked Quartets	L = Hidden Quintets
M = Naked Quintets	P = Branch & Bound
N = Forced Chains (2-candidates cells)	
O = Forced Chains (for 3-candidates cells)	

If we compare Table II and Table III we can say that hardware version is in many cases faster than the software version, although the first one uses fewer techniques. For example, Sudokus 8a, 9a and 10a takes more than 3, 10 and 48 seconds respectively with the software version, while all of them are solved in less than a second with the hardware version.

However, the software version can solve a greater number of Sudokus. This leads us to believe that the implementation of these techniques in hardware will improve the results and increase the number of Sudokus that can be solved in the time required by the FPT Design Competition.

VI. CONCLUSIONS

Our design can solve very fast low and medium-complexity Sudokus up to order 11. However, it is still inefficient when dealing with Sudokus of high complexity, even for small orders. The problem is that it applies a Branch&Bound technique that carries out an extensive exploration of the solution space, and this is not affordable for complex Sudokus. We believe that the best way to improve the results is applying heuristics that frequently can significantly reduce the solution space. We have implemented some of these heuristics (Singles, Hidden Singles, Hidden Pairs, Hidden Triplets and Hidden Quartets), but we have identified that more heuristics are needed to deal with the most complex Sudokus. We expect to include at least Locked Candidates technique or Naked Candidates technique in the near future. Moreover, we would like to use an external memory in order to deal with Sudokus of order 12 and higher.

REFERENCES

[1] Felgenhauer, Bertram, and F. Jarvis. "Mathematics of Sudoku I." 25 January 2006. http://www.afjarvis.staff.shef.ac.uk/sudoku/felgenhauer_jarvis_spec1.pdf

[2] E. Russell and F. Jarvis. "Mathematics of Sudoku II." 25 January 2006. http://www.afjarvis.staff.shef.ac.uk/sudoku/russell_jarvis_spec2.pdf

[3] Wikipedia: The Free Encyclopedia. *Mathematics of Sudoku*. <http://en.wikipedia.org/wiki/Mathematics_of_sudoku>.

[4] Wikipedia: The Free Encyclopedia. *Group (mathematics)*. http://en.wikipedia.org/wiki/Group_(mathematics)

[5] http://www.math.cornell.edu/~mec/Summer2009/Mahmood/Intro.html

[6] http://www.spiritustemporis.com/sudoku.

[7] http://www.scanraid.com/Strategy_Families.

[8] http://www.sudokudragon.com/

[9] http://angusj.com/sudoku/hints.php

[10] Xilinx Web Site. http://www.xilinx.com/univ/xupv2p.html

[11] Xilinx Inc., Available online: http://www.xilinx.com

Anexo II: Poster presentado en el congreso del FPT '09

AN INITIAL SPECIFIC PROCESSOR FOR SUDOKU SOLVING

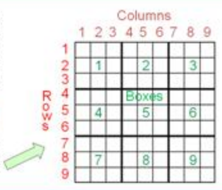
Carlos González*, Javier Olivito†, Javier Resano†
*Computer Architecture Department, Universidad Complutense de Madrid, Spain
†DIIS-I3A, Universidad de Zaragoza, Spain
Emails: carlosgonzalez@fdi.ucm.es, creoenoli@gmail.com, jresano@unizar.es

Introduction

Designing a Sudoku solver is a challenging problem for a digital logic designer. Several important mathematical concepts are hidden behind the scenes: combinatorics [1,2] used in counting valid Sudoku grids, group theory [3] used to describe ideas about when two grids are equivalent, and computational complexity with regards to solving Sudokus. This is nicely explained in [4,5].

Standard version of Sudoku

It consists of a 9×9 square grid containing 81 cells. Some of the 81 cells are filled in with numbers from the set {1,2,3,4,5,6,7,8,9}. The goal is to fill in the remaining cells guaranteeing that the nine digits are used once in each row, each column, and each box.



This standard version can be generalized for other

The 9x9 square is in fact a Sudoku of order 3. A Sudoku of order N is an $N^2 \times N^2$ square grid, subdivided into N^2 boxes, each of size $N \times N$. In this case it must be guaranteed that the digits from 1 to N^2 are used once in each row, column and box.

Techniques

There are many techniques to solve a Sudoku. Some of these are basics as Singles or Hidden Singles and others are more advanced as Locked Candidates, Naked Candidates, Hidden Candidates, X-Wing, Swordfish, etc.

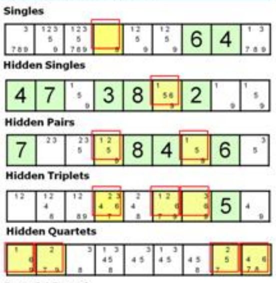
However, these techniques [6,7,8,9] are only heuristics that can reduce the solution space, but cannot guarantee that they find solution. Hence it is interesting to combine them with a trial-and-error approach that explores the remaining solution space once it has been narrowed with the previous techniques.

In this poster

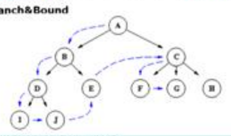
We present the design and implementation of a specific processor that can solve Sudokus from order 3 to 11 applying the following techniques to reduce the solution design-space: Singles, Hidden Singles, Hidden Pairs, Hidden Triplets and Hidden Quartets. After these techniques the processor carries out a final design-space exploration using a Branch&Bound strategy.

HW implementation

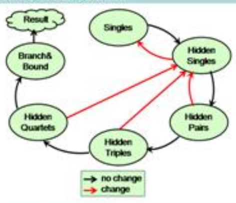
Implemented techniques



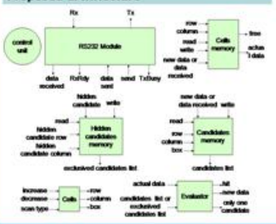
Branch&Bound

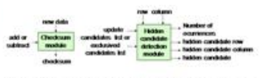


Outline of our solution

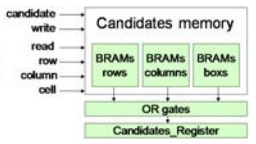


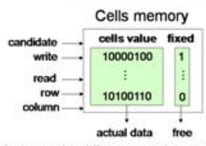
Proposed architecture



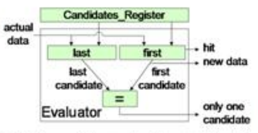


This figure shows the structure hardware needed to implement the Sudoku solver and the I/O communications. For data I/O, it uses a RS232 module and it stores the Sudoku in different memories. It also has a module for checksum calculation, a counter modified to obtain the row, column and box of the cells depending on the scan type (by rows, columns or boxes), an evaluator and a hidden candidate detection module. We will describe the most important elements in detail.

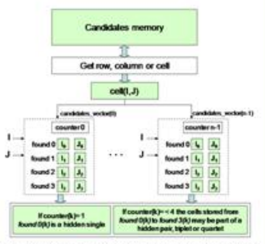




Sudoku is stored in different memories to easily update his status at any time. The ***Candidates memory*** is composed by three memories where we have the possible candidates from each row, column and box, allowing us to get the cell's candidates in only one read operation. The ***Cells memory*** stores the value of the cell and if it is fixed or not. Finally, we use the Hidden candidates memory to store the candidates that are excluded for a particular cell after the execution of the techniques.



Evaluator module receives the candidates list and the actual data for a cell and it is able to give us the next possible candidate, if it exits, and if the cell has only one possible candidate. To parallelize the search of the first candidate and the last candidate, we divide the candidates list into parts and then we compare the partial results. This module is very useful for the Singles and Branch&Bound techniques.


Candidates memory
Get row, column or cell
cell(I,J)
candidates_vector(0)
candidates_vector(n-1)
counter0
counter n-1
found 0
found 1
found 2
found 3
If counter(k)=1 found 0(k) is a hidden single
If counter(k)=<4 the cells stored from found 0(k) to found 3(k) may be part of a hidden pair, triplet or quartet

This figure shows the architecture of the ***Hidden candidate detection module***. To fill the structure, we read sequentially the candidates for the cells that form a row, column or box and then, for the possible candidates his counter is increased and the row and column are stored. In this way, we obtain the number of occurrences of a candidate in a row, column or box and the cells where it appears. Therefore, if the counter for one candidate has a value of one, we have found a Hidden Single candidate, and if the counter has a value less than five this candidate may be part of a Hidden Pair, Triplet or Quartet.

Hardware platform and software used

2 PowerPCs
13969 Slices
2448 Kb Block RAMs

Our design has been implemented on a **XC2VP30 Virtex-II Pro FPGA** included in the XUP Development System [10].
We have developed our design using **VHDL language** for the specification and the **Xilinx ISE 10.1.3** environment [11].

Resources used for the Sudoku solvers

Techniques	Maximum Order	BRAMs	Slices and Percentage	
Branch&Bound	15	29	1832	13%
Singles	15	29	2183	16%
Hidden Singles	15	134	7675	56%
Hidden Pairs	11	134	9187	67%
Hidden Triplets	11	134	1080	78%
Hidden Quartets	11	134	12265	90%

Benchmarks computation time in seconds

Order	Type (a)	Type (b)
3	0.0086	0.009762
4	0.024761	Not in time
5	0.060281	Not in time
6	0.117894	Not in time
7	0.224523	Not in time
8	0.058375	Not in time
9	0.300835	Not in time
10	0.322603	Not in time
11	0.381182	Not in time

SW implementation

In order to improve the design we have implemented an equivalent software version, and we have added several techniques that reduce the solution space to it. In order to check which techniques are necessary to solve the different benchmarks, we have implemented a great number of techniques: Singles, Hidden Singles, Hidden Pairs, Hidden Triplets, Hidden Quartets, Hidden Quintets, Naked Pairs, Naked Triplets, Naked Quartets, Naked Quintets, Candidates Lines, Double Lines, Triple Lines, Forced Chain, Forced Chains (2-candidates cells), Forced Chains (for 3-candidates cells) and Branch & Bound [6,7,8,9].

Platform

We have developed this software version using **C language** and the program was executed on a **1,66GHz processor** with **2GB RAM**.

Benchmarks computation time in seconds

Benchmark	Computation Time	Applied techniques
3a	0.003	A, B, C, D, E, F, G, H, N
4a	0.01	A, B, C, D, E, F, G, H, I, J, K, N
5a	0.038	A, B, C, D, E, F, G, H, I, J, K, L, M, N
6a	0.024	A, B, C, D, E, F, G, H, I, J, K, L, M, N
7a	0.522	A, B, C, D, E, F, G, H, I, J, K, L, M, N, P
8a	3.210	A, B, C, D, E, F, G, H, I, J, K, L, M, N, P
9a	10.402	A, B, C, D, E, F, G, H, I, J, K, L, M, N, P
10a	48.520	A, B, C, D, E, F, G, H, I, J, K, L, M, N, P
11a	0.652	A, B, C, D, E, F, G, H, I, J, K, L, M, N
3b	0.009	A, B, C, D, E, F, G, H
4b	1.81	A, B, C, D, E, F, G, H, J, N
5b	56.133	A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P

A = Hidden pairs	B = Naked pairs
C = Hidden singles	D = Singles
E = Hidden triplets	F = Naked triplets
G = Candidates Lines	H = Double Lines
I = Triple Lines	J = Hidden Quartets
K = Naked Quartets	L = Hidden Quintets
M = Naked Quintets	P = Branch & Bound
N = Forced Chains (2-candidates cells)	
O = Forced Chains (for 3-candidates cells)	

Comparison between HW and SW

If we compare benchmarks computaton time tables we can say that hardware version is in many cases faster than the software version, although the first one uses fewer techniques. For example, Sudokus 8a, 9a and 10a takes more than 3, 10 and 48 seconds respectively with the software version, while all of them are solved in less than a second with the hardware version.

Conclusions and future lines

Our design can solve very fast low and medium-complexity Sudokus up to order 11. However, it is still inefficient when dealing with Sudokus of high complexity, even for small orders. The problem is that it applies a Branch&Bound technique that carries out an extensive exploration of the solution space, and this is not affordable for complex Sudokus. We believe that the best way to improve the results is applying heuristics that frequently can significantly reduce the solution space. We have implemented some of these heuristics (Singles, Hidden Singles, Hidden Pairs, Hidden Triplets and Hidden Quartets), but we have identified that more heuristics are needed to deal with the most complex Sudokus.

We expect to include at least Locked Candidates technique or Naked Candidates technique in the near future. Moreover, we would like to use an external memory in order to deal with Sudokus of order 12 and higher.

References

[1] Felgenhauer, Bertram, and F. Jarvis. "Mathematics of Sudoku I." 25 January 2006. http://www.afjarvis.staff.shef.ac.uk/sudoku/felgenhauer_jarvis_spec1.pdf
[2] E. Russell and F. Jarvis. "Mathematics of Sudoku II." 25 January 2006. http://www.afjarvis.staff.shef.ac.uk/sudoku/russell_jarvis_spec2.pdf
[3] Wikipedia: The Free Encyclopedia. Mathematics of Sudoku. <http://en.wikipedia.org/wiki/Mathematics_of_sudoku>
[4] Wikipedia: The Free Encyclopedia. Group (mathematics). http://en.wikipedia.org/wiki/Group_(mathematics)
[5] http://www.math.cornell.edu/~mec/Summer2009/Mahmood/Intro.html
[6] http://www.spiritustemporis.com/sudoku.
[7] http://www.scanraid.com/Strategy_Families.
[8] http://www.sudokudragon.com/
[9] http://angusj.com/sudoku/hints.php
[10] Xilinx Web Site. http://www.xilinx.com/univ/xupv2p.html
[11] Xilinx Inc., Available online: http://www.xilinx.com