VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

HARDWARE ACCELERATION OF THE SUDOKU GAME

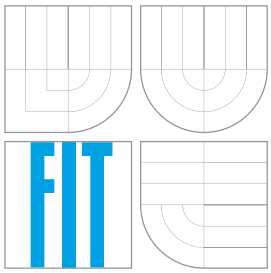
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

RÓBERT JURINEK

BRNO 2010

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

HARDWAROVÁ AKCELERACE HRY SUDOKU

HARDWARE ACCELERATION OF THE SUDOKU GAME

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — RÓBERT JURINEK
AUTHOR

VEDOUCÍ PRÁCE — Ing. JAN KAŠTIL
SUPERVISOR

BRNO 2010

Abstrakt

Tato práce pojednáva o implementaci hardwarové jednotky řešící SUDOKU. V práci jsem zadefinoval pojmy týkající se hlavolamu SUDOKU a popsal některé jeho vlastnosti, zejména z hlediska řešení na počítačovém systému. Práce dále popisuje některé techniky používané při řešení SUDOKU a možnosti jejich hardwarové implementace. V hlavní části je popsána konkrétní realizace jednotky řešící SUDOKU a také je zhodnocena výkonnost navržené jednotky. Jednotku jsem ověřil i na reálném hardwaru. V závěru práce jsem zhodnotil možnosti dalšího rozšíření navržené jednotky.

Abstract

This work deals with the implementation of a hardware-based SUDOKU solver. SUDOKU terminology is described as well as SUDOKU puzzle metrics related to computer puzzle solvers. Solving techniques are introduced and possibilities of a hardware-based implementation are discussed. The implementation of the SUDOKU solver is described and the performance of the implemented unit is assessed. The designed solver was also verified on a real hardware platform. In conclusions possible unit extensions are proposed.

Klíčová slova

SUDOKU, FPGA, řešení hlavolamů, hardwarová akcelerace

Keywords

SUDOKU, FPGA, puzzle solving, hardware-based acceleration

Citace

Róbert Jurinek: Hardware acceleration of the SUDOKU game, bakalářská práce, Brno, FIT VUT v Brně, 2010

Hardware acceleration of the SUDOKU game

Prohlášení

Hereby I declare, that this thesis is my authorial work. I have worked it out by my own. All sources, references and literature used during elaboration of this work are properly cited and listed in complete reference to the due source.

.....
Róbert Jurinek
May 18, 2010

Poděkování

I would like to thank Ing. Jan Kaštil for his suggestions, comments and time he spent helping me with this work.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

SUDOKU was first published in 1979 by Dell Magazines as Number Place. *Suuji (or suji) wa dokushin ni kagiru*, which was later abbreviated to SUDOKU, can be translated as *the digits must be single* or *the digits are limited to one occurrence*. This name comes from Japan, where it was first introduced in the paper Monthly Nikolist in 1984.

The SUDOKU game became very popular in the last twenty years. It can be found in almost every newspaper all over the world. Most people noticed widened SUDOKU software. It is available through numbers of websites as well as for many computer platforms, mobile devices and video game consoles. It even appeared in the Teletext service. An annual international SUDOKU competition, The World SUDOKU Championship, is visited by teams from various countries. These are services, activities and commodities intended for a popular amusement.

There is also software designed for SUDOKU solving. However, much less of hardware is available. The goal of the design competition at FPT'09 (the 2009 International Conference on Field-Programmable Technology) was to develop a general purpose SUDOKU solver on field-programmable gate arrays. One of leading designs will serve as a performance reference to my solver.

In the first chapter, the SUDOKU game is described. SUDOKU terminology is introduced and puzzle basic characteristics which influence the solver's design are discussed.

The second chapter introduces obvious solving techniques applicable in hardware-based solvers. The performance of these techniques is predicted and estimation of needed resources is provided.

A description of a target hardware platform is the topic of the third chapter. Generic structure of programmable chips is foreshadowed. The next part of this chapter introduces features of the target hardware platform. The chapter is finalized with the description of a hardware programming process.

The design of the solver is the subject of matter of the fourth chapter. Particular components and application of used solving methods are described. The performance of the solver is also discussed. Results of verification on real hardware platform are mentioned in the last part of this chapter.

Historical and general facts concerning SUDOKU used in this chapter were adopted from [7] and [5].

Chapter 2

The SUDOKU game

A popular form of SUDOKU is presented on a square grid of 9×9 cells. In oncoming text, terminology and characteristics of SUDOKU game will be introduced and generalized for grids of any size.

2.1 SUDOKU rules

2.1.1 Latin square

Latin square was introduced by Leonhard Euler. It is rectangular grid containing $n \times n$ cells. Every cell is filled-in by a symbol. Every symbol can occur within every row and column exactly once. Word Latin refers to Latin alphabet symbols which Euler used to fill-in cells with. Any set of symbols can be used for this purpose. The example Latin square filled with natural numbers is pictured in Figure 2.1.

2	1	3
1	3	2
3	2	1

Figure 2.1: Example Latin square

2.1.2 Generic SUDOKU

SUDOKU game is derived from Latin square. The puzzle of *order* n is represented by a rectangular *grid* of $n^2 \times n^2$ *cells*. The task is to fill partially filled grid with *symbols*, so that every *row*, *column* and *block* contains every symbol exactly once [3]. Symbols are represented by natural numbers from 1 up to n^2. Since rows, columns and blocks have almost all the attributes in common [5], these will be further referred as *virtual lines*. Every cell is a member of three virtual lines (row, column, and block). In the puzzle, there are $3n^2$ such virtual lines.

An example of the SUDOKU puzzle of order 3 is shown in Figure 2.2. Blocks are emphasized by a thicker line. The puzzle contains 30 *given symbols*. *Candidates* are symbols that can be filled into the certain cell. Candidates are in-scripted into each cell by small signs. The grey cell's candidates are 1, 2 and 4.

5	3	124	26	7	2468	1489	1249	248
6	247	247	1	9	5	3478	234	2478
12	9	8	23	34	24	13457	6	247
8	125	1259	579	6	147	4679	2459	3
4	25	2569	8	5	3	579	259	1
7	15	1359	59	2	14	4589	459	6
139	6	134579	357	35	7	2	8	4
23	278	237	4	1	9	36	3	5
123	1245	12345	2356	8	26	1346	7	9

Figure 2.2: Example SUDOKU puzzle [7] (modified)

2.2 Sets of candidates

Candidates in-scripting into each cell is a favourite practice of humans when solving SUDOKU. After placing a symbol into some cell, this symbol is eliminated from the sets of candidates of all the cells that share the same virtual line. Computer-based solvers can emulate this manner. The construction of such a solver demands storing the sets of candidates related to every single cell. Allocation of one flag bit per candidate (candidate present, or candidate absent) requires n^2 bits for each cell. Since there is n^4 cells, this technique requires storing n^6 bits.

Considering the example cell from Figure 2.2, the sets of candidates of the related virtual lines are as follows (R refers to row, C refers to column, and B refers to block):

- $R_1 = \{1, 2, 4, 6, 8, 9\}$
- $C_3 = \{1, 2, 3, 4, 5, 6, 7, 9\}$
- $B_1 = \{1, 2, 4, 7\}$

The cell can be only filled by symbols that are elements of all these sets. Consequently, the set of candidates related to the cell equals the intersection of the sets of candidates related to all the three virtual lines [6]. The example situation is depicted in Figure 2.3.

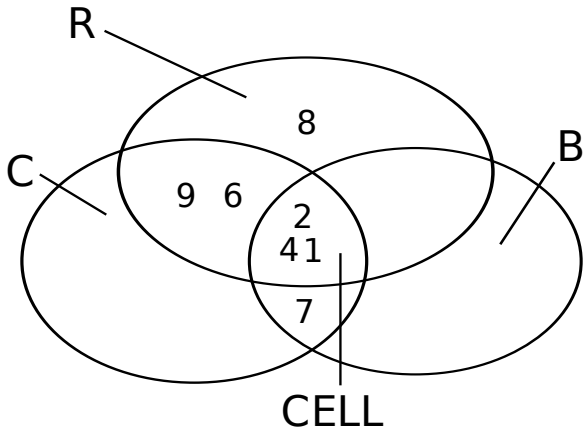


Figure 2.3: Intersection of the sets of candidates

Elaboration of storage capacity required is shown in Table 2.1. As it can be seen in the table, this approach reduces data volume by factor of $\frac{1}{3}n^2$ SUDOKU solvers of high-ordered puzzles can be designed by utilizing this method. My work focuses on puzzles of orders 3 up to 10.

n	$\|\|S\|\| = n^2$	$G_s = n^4$	$V_c = 3n^2$	$G_s \cdot \|\|S\|\| = n^6$	$V_c \cdot \|\|S\|\| = 3n^4$
3	9	81	27	729	**243**
4	16	256	48	4 096	**768**
5	25	625	75	15 625	**1 875**
6	36	1 296	108	46 656	**3 888**
7	49	2 401	147	117 649	**7 203**
8	64	4 096	192	262 144	**12 288**
9	81	6 561	243	531 441	**19 683**
10	100	10 000	300	1 000 000	**30 000**

Table 2.1: Calculations of data volume

- S - the set of symbols
- G_s - the grid size
- V_c - the virtual lines count

In my work, I used a method of storing the sets of candidates similar to that described in [6]. The sets of candidates are represented by bitmaps. These bitmaps are separately stored in registers for each virtual line. These registers are grouped into clusters. There are three such clusters of registers related to rows, columns and blocks. Initially, all the bits are set high (`1`). When puzzle is sequentially read into the solver, respective bitmaps are eliminated. The related position within a bitmap is set low (`0`). This is performed by the bitwise `XOR` logic operation. The same operation is performed during solving process when solver fills-in any of cells. If the solver needs to return to previous state for some reason, removed candidate is restituted by the bitwise `OR` logic operation.

The set of candidates for a particular cell is obviously needed by the solver. If this is required, the intersection of three sets of candidates related to the cell is performed. The bitwise `AND` logic operation is actioned over three mentioned bitmaps. This operation is demonstrated in Figure 2.4 using example situation depicted in Figure 2.3.

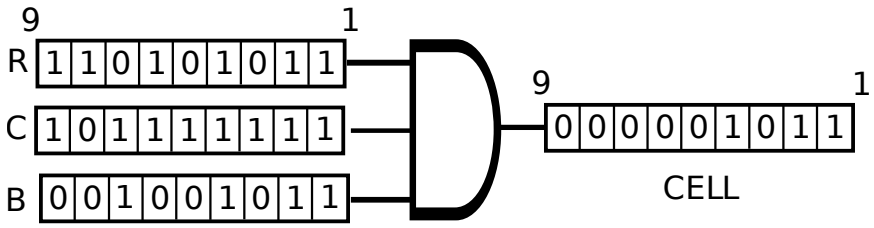


Figure 2.4: Intersection of the sets

The operation described above is performed very fast due to parallel logic used. Parallel nature of performing tasks is substantive principle of hardware-based acceleration. The solver is now able to quickly recognize candidates of any cell. The utilization of this ability will be introduced in the forthcoming chapter.

Chapter 3

Solving techniques

This chapter describes obvious SUDOKU puzzle solving techniques applicable in hardware-based solvers. Two approaches to solve SUDOKU are mentioned. The first approach employs very fast techniques using logical constraints. This approach is described in the first part of this chapter. The second one utilizes a brute-force search. This is a very general method but also it has exhaustive time demands. I took an advantage of both of them in my solution.

3.1 Logical constraints

Almost all of the SUDOKU problems of order 3 can be solved by applying only logical constraints [3]. This fact can be applied to SUDOKU problems of any order [5]. Applying logical constraints performs elimination of sets of candidates. Advanced solving techniques that use logical constraints have exaggerated resources demands. This is because these methods require information about various groups of cells. I found two of such logical constraints that can be plainly applied in hardware-based solver. Both of them only require knowledge of the sets of candidates of virtual lines. Special attribute of these two techniques is that they directly allow solver to fill in a cell.

3.1.1 Naked single

This is the most straightforward method of SUDOKU solving. It is assumed that there is only one remaining symbol in the set of candidates of a cell. Such a candidate is marked as a *naked single* candidate. In this case, the symbol can be written into the cell [5]. An example of this situation is depicted in Figure 3.1. The main liability of this technique is that it is unable to solve all the puzzles. Therefore algorithm that uses only this technique would be incomplete.

The solver is able to work on the set of candidates related to any cell as it was described in the previous chapter. Application of the naked single method requires recognition of sole candidate within a set. I used a *one-hot detector* for this purpose since the set is represented by bitmap. The one-hot detector is a combinational logic unit that detects a presence of the single bit that is set high. If the one-hot situation is detected, the output bit of the detector is set high. Otherwise, the output bit is set low.

The naked single method was deprecated by designers of the solver that my implementation is compared with. They considered it too weak and hence they only employed exhaustive search with guessing. This technique worked well for puzzles of order 3. Puzzles

with a lot of empty cells of order more than 3 were not solved in reasonable time. Puzzles of order more than 8 were not solved in 24 hours.

I decided to probe naked single method. This method allows to solve a puzzle in constant time, if there is naked single candidate present in the puzzle on every step. If there is none, a guess should be executed to continue solving, or some another logical constraint applied. Puzzles with a lot of guesses seems to be solved in unpredictable time. On the other hand, I predicted that this method should reduce search tree of brute-force searching algorithm employed in pure guessing method. Presentation of results is in Chapter 6.

5	3	124	26	7	2468	1489	1249	248
6	247	247	1	9	5	3478	234	2478
12	9	8	23	34	24	13457	6	247
8	125	1259	579	6	147	4679	2459	3
4	25	2569	8	5	3	579	259	1
7	15	1359	59	2	14	4589	459	6
139	6	134579	357	35	7	2	8	4
23	278	237	4	1	9	36	3	5
123	1245	12345	2356	8	26	1346	7	9

Figure 3.1: Naked single candidates [7] (modified)

3.1.2 Hidden single

Another straightforward method of SUDOKU solving is looking for a *hidden single* candidate. The method requires investigation of virtual lines. If there is unique cell in a virtual line that can contain particular symbol, this symbol must be assigned to this cell [5]. This method requires gathering unit that stores information about cells within a virtual line. After the virtual line is traversed, found hidden single candidates are filled-in.

I also considered that this information could be stored separately for each virtual line. For every symbol there would be pointer to all possible locations within a virtual line. This approach requires $3n^6$ pointers to be stored. Therefore, this approach is of no use for high-ordered hardware-based SUDOKU solvers.

If the solver proposed in this work should be extended to support this method, implementation of a gathering unit would be only required.

5	3	124	26	7	2468	1489	1249	248
6	247	247	1	9	5	3478	234	2478
12	9	8	23	34	24	13457	6	247
8	125	1259	579	6	147	4679	2459	3
4	25	2569	8	5	3	579	259	1
7	15	1359	59	2	14	4589	459	6
139	6	134579	357	35	7	2	8	4
23	278	237	4	1	9	36	3	5
123	1245	12345	2356	8	26	1346	7	9

Figure 3.2: Hidden single candidates [7] (modified)

3.1.3 Other techniques

In addition to the techniques mentioned above, there is another group of elimination-based SUDOKU solving techniques. These techniques do not directly allow to fill-in a cell. Instead, they allow to eliminate candidates from particular cell.

Naked single and hidden single candidate techniques can be extended to more cells. Instead of one candidate present in one cell, there are the same two candidates in two cells, the same three candidates in three cells, etc. These techniques are known as *naked sets*, or *hidden sets*.

Locked candidates are found within certain intersection of block with another virtual lines. If there is a block where the only possible positions for a candidate are in intersecting part with another virtual line, the symbol must appear in that intersection within the block. From another parts of intersecting virtual lines, this candidate can be eliminated. Another similar situation occurs when a symbol missing from a virtual line (row, or column) can be placed only within one of the blocks that intersect that virtual line. Therefore the symbol must be placed on the intersection of these virtual lines. This candidate can be eliminated from other cells within a block.

X-wing is elimination technique applied on two rows, or columns. There is the same candidate exactly twice in two rows and on the same position of these rows. These candidates connected together with `X` creates two pairs. The candidate that created X-wing, must be filled-in on one of these positions. Hence the candidate can be eliminated from other positions within related virtual lines (in this technique, rows and columns can be swapped). There is also *swordfish* technique similar to X-wing. Swordfish is extended to three symbols, rows, and columns [5].

3.2 Guessing

The most frequently used method by ordinary computer solvers is brute-force search. Cells are filled speculatively. The symbol is *guessed* from the set of candidates. Puzzle is traversed in row-major order until a conflict with SUDOKU rules is discovered. On conflict the solver backtracks to a cell that has untried candidates. Otherwise it continues to the next empty cell. This method always finds the solution [5].

3.2.1 Next empty cell search

At first, the method requires to find the first empty cell. Empty cell search performed over thousands of cells is enormously time-consuming. This is accelerated by a unit that can determine address of the first empty cell without search. Such unit operates over bitmap of size n^2 that represents SUDOKU grid. Status of each cell is represented by a value of corresponding bit. If the cell is empty, related bit is set high. Otherwise, it is set low. Address of the first empty cell is then determined by a priority encoder. The priority encoder is a combinational logic unit that outputs position of the first high set bit. Hence address of the first empty cell within SUDOKU grid is determined in one clock cycle. The first empty cell unit was described in [6].

The described solution works well if the solver uses only guessing. The unit, as described above, does not allow to traverse over empty cells. My solver performs search for naked single candidate over empty cells after each guess. Therefore the unit has to be designed considering this requirement. I designed the unit that derives address of the next empty cell from given address. In this unit, bitmaps are arranged into rows and columns according to classic SUDOKU grid. Particular cell is addressed by combination of corresponding row number and column number. This unit will be closely described in Section 5.1.5.

3.2.2 Backtracking

If guessing is employed to solve a puzzle, a conflict with SUDOKU rules can occur. On conflict the solver must be able to return to the first previous cell that has untried candidates. If solver saved status of all the cells on every step, it would allow to return to any previous status in very little time. This approach can not be realised because of excessive data record needed.

Pushing every performed step onto a stack seems to be the best solution. Stack overflow can occur when stack depth is underrated, therefore necessary stack depth should be defined. Considering the worst-case puzzle all the empty cells are filled by guessing. In this case it is necessary to know the minimal number of given symbols in the puzzle that implies the number of empty cells.

The minimal number of given symbols for SUDOKU of order 3 is 17. There is no known valid puzzle of 16 given symbols yet [2]. For SUDOKU of order above 3, the minimal number of given symbols has not been proven neither computed. I decided to use a simple formula that approximately calculates the number of minimal given symbols and therefore the stack depth.

SUDOKU puzzle of order 3 contains $3^4 = 81$ cells. In the minimal puzzle with 17 prefilled cells, $81 - 17 = 64$ cells are left blank. I will assume that number of blank cells in the minimal puzzle of order n (B_{mn}) is computed as follows:

For order $n = 3$ applies

$$B_{m3} = 64 = 8^2 = (9-1)^2 = (3^2-1)^2 = (n^2-1)^2 \tag{3.1}$$

Assuming for order n

$$B_{mn} = (n^2-1)^2 \tag{3.2}$$

Hence depth of the stack will be further designed consulting Equation 3.2. Implementation of the stack will be described in more detail Section 5.1.3.

The stack stores a backtrack path only. The path is composed of addresses on which writing was performed. If solver backtracks, symbols are restituted into the sets of candidates. Therefore the solver needs to know the symbol written on a particular address. Although symbols could be pushed onto the stack, it is not a good practice because it is wasting storage resources. Since solved puzzle should be readable from the solver, I decided to create one common storage unit of symbols. The unit is realised as a standard RAM unit. Every symbol present in the grid is written into this storage. Each symbol is addressed by the row number and column number.

3.2.3 The guess process

If solver guessed symbols randomly, it would be able to recognize which symbols were guessed. For this purpose, enormous data storage would be allocated. My solver performs guess with the highest symbol available. The symbol is represented by the leftmost bit that is set high. I used a priority encoder to determine the value of this symbol. Previously guessed symbols are masked if solver backtracks. The solver reads the value of the last guessed symbol from the symbols storage unit. From the value of this symbol, the mask is generated. The mask is then applied to the bitmap of set of candidates. The masked bitmap only contains previously unguessed symbols. Demonstration of this operation is pictured in Figure 3.3.

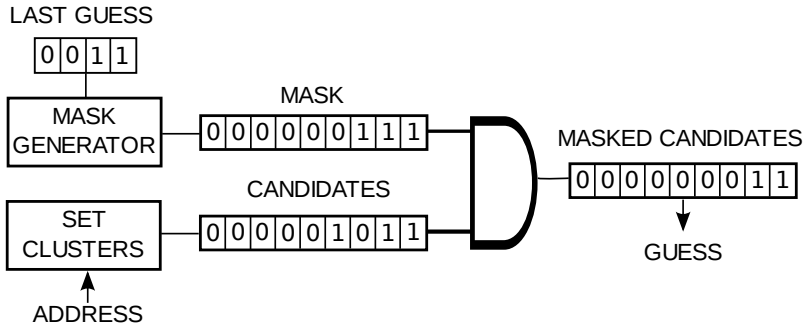


Figure 3.3: The guess process

Chapter 4

Target architecture

There is a variety of programmable hardware platforms on the market. I focused on one family of PLD (programmable logic devices). This type of devices will be described and features of the particular chip used in my work will be presented in this chapter. The last part of this chapter is dedicated to a programming process of such devices.

4.1 FPGA

FPGA (field-programmable gate array) is a chip with predesigned and fabricated cells branded as CLB (configurable logic block). In Figure 4.1, there is a structure of a generic FPGA depicted. The chip is customized by creating (or destroying) connections within the CLBs and between the routing wires and CLBs. Each CLB contains a few slices. Such a slice embodies LUTs (look-up tables), D flip-flops, MUXes (multiplexers) and other random logic [4]. In addition to this basic logic cells, FPGA chips involve ASICs (application-specific integrated circuits) comprehensive DSP (digital signal processing) slices, Block RAMs, etc. Modern FPGA chips provide thousands of slices while possible operating frequency is up to 1.5GHz.

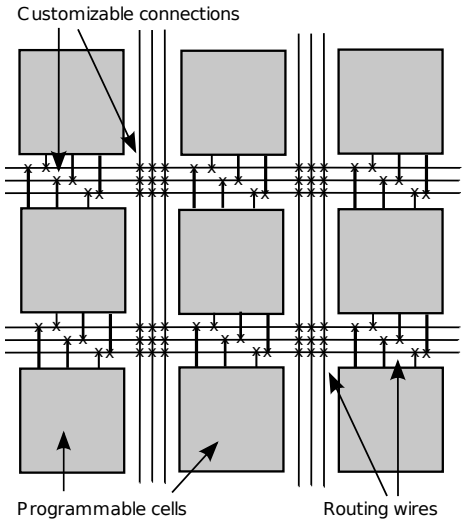


Figure 4.1: generic FPGA

4.2 Employed platform

ML605 Evaluation Kit by Xilinx was picked to verify design of the solver. This board includes the Virtex-6 XC6VLX240T FPGA chip. The chip has been chosen since it offers enough logic resources for the design of the solver. Table 4.1 depicts the chip's available resources.

Slices	SLICELs	SLICEMs	6-input LUTs	Distr. RAM [kb]	Flip-Flops
37 680	23 080	14 600	150 720	3 770	301 440

Table 4.1: Logic resources available in Virtex-6 XC6VLX240T FPGA [9]

Each CLB on the Virtex-6 chip contains two slices. In one slice, there are four function generators (LUTs). Each of the four LUTs in a slice disposes of six independent inputs and two independent outputs. LUTs, in combination with slice multiplexers, provide any function of up to eight inputs in a slice. Multiple LUTs in a SLICEM can be combined to store larger amount of data as a synchronous RAM resource called a distributed RAM element. RAM elements are configurable in various ways to implement different types of RAM modules. RAM module can provide up to 256 bits within one CLB. RAM module configuration used within my design is described in more detail in Section 5.1.2. There are eight (four original and four additional) storage elements in a slice. Common configuration of these elements is as edge-triggered D-type flip-flops. The D input is obviously driven by a LUT output via one of the slice MUXes [9]. Simplified scheme of one pair of flip-flop and LUT within a slice is depicted in Figure 4.2.

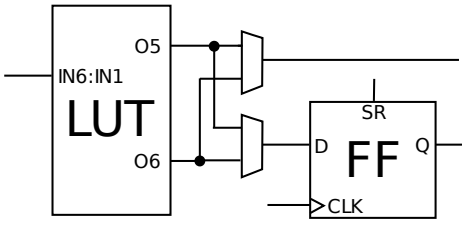


Figure 4.2: LUT-FF pair

The kit provides various communication and configuration interfaces and protocols. In this case, the kit was connected via USB cable. Onboard JTAG interface was employed to configure FPGA chip. The kit also facilitates connections via various modern buses and protocols such as PCI express, Ethernet...

4.3 Logic design flow

The solver is written in VHDL language. VHDL language is a HDL (hardware description language) used for the specification, modeling, synthesis, and simulation of digital logic circuits [4]. XILINX provides software tools including text editing interface, design simulator, and tools for programming PLDs. VHDL code is processed by a *synthesizer*. The synthesizer generates a file that contains optimized logical design data and constraints. This file is processed by design implementation tools. These tools provides mapping design to FPGA

resources, placement of mapped design and routing within placed design. Routed design is then transformed into the bitstream file. This file is typically downloaded to a target device.

Debugging and verification of logic design is typically performed within design process. Functional simulation of design is accomplished before and after synthesis process. There are two simulation environments obviously used. Model Sim provided by Mentor Graphics and Isim provided within ISE by Xilinx. I prefer Isim because it is included in ISE and therefore there is no need to transfer source codes from one program to another.

Static timing analysis is provided during the implementation process. This analysis checks whether specified timing constraints were met. The last method I used is in-circuit verification.

In-circuit verification is provided by ChipScope software tool that communicates via USB within scopes added to the design. Scopes are predesigned specialized cores. CoreGen is used for customizing and generating those cores. Often used cores are integrated logic analyzer (ILA) and virtual input and output (VIO). Both of them are driven by an Integrated Controller (ICON) via 32-bit control signal. ILA unit can be connected to any of the signals within the design being verified. This unit collects data samples from the design being verified. Proportions of captured data samples are adjustable in various ways. I used this unit to verify my design on Virtex-6 chip. VIO unit provides virtual input and output ports for the design being verified. Input ports are fed in real-time via ChipScope software interface [8]. Adding scopes into the design can be provided in two ways. *CoreGen* flow requires generation of cores by CoreGen. Generated cores are then instantiated into the design VHDL code. Units are synthesized as black boxes. VIO core can be only added via CoreGen flow. This approach did not work in school lab for some reasons. In *core inserter* flow, cores are added to the design after synthesis. The advantage of this approach is that cores are included into the design automatically by Core. Specification of connected signals via graphical user interface is only required. The process of design flow is schematized in Figure 4.3.

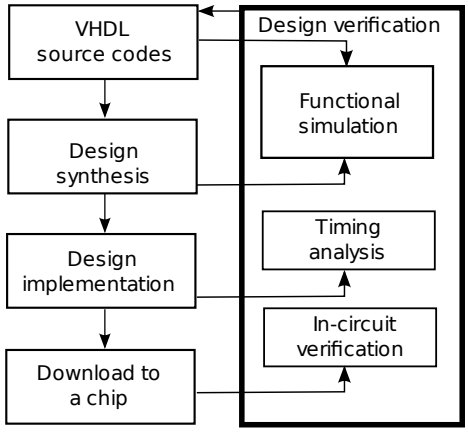


Figure 4.3: Iterative process of design flow

Chapter 5

The SUDOKU solver

In this chapter, I will describe implementation of my SUDOKU solver. The architecture of the solver will be presented and implementation of solving algorithm will be introduced.

5.1 Solver components

The solver's design contains four independent units. Symbols are stored into a *symbols RAM unit*. Backtracking path is being pushed onto a *backtrack stack unit*. Candidate elimination and recognition is performed by an *elimination unit*. *Next empty cell unit* enables to traverse the SUDOKU grid. All the components are driven by a *top module*. The SUDOKU order is defined before solver synthesis (or simulation). All the solver components are fully generic. All the necessary equations related to the design are automatically computed. Used equations will be introduced and results for particular orders will be presented in the later text.

5.1.1 Design analysis

In a popular form of SUDOKU, symbols are represented by natural numbers. My solver utilizes a binary encoded representation of symbols. The symbols are represented by binary numbers from 0 to $n^2 - 1$. Hence the required number of binary digits (bits) to encode the symbols should be defined.

There is n^2 symbols, the number of bits (b) that are needed to encode the symbols was computed as follows:

$$b = \lceil \log_2(n^2) \rceil \tag{5.1}$$

The number of rows and the number of columns is the same as the number of symbols in a particular SUDOKU puzzle. In my design, addressing is provided by the row number and the column number (i.e., the row address and the column address). Hence address' bus width will be twice as wide as symbols' bus width. I performed complete elaboration of buses width requirements. Results of this calculation are depicted in Table 5.1.

n	n^2	$B_S = \lceil \log_2(n^2) \rceil$	$B_A = 2B_S$
3	9	4	8
4	16	4	8
5	25	5	10
6	36	6	12
7	49	6	12
8	64	6	12
9	81	7	14
10	100	7	14

Table 5.1: Buses width calculation

- B_S - the symbols bus
- B_A - the address bus

5.1.2 Symbols RAM unit

The symbols filled into cells are stored in the symbols RAM unit that is composed of distributed RAMs of target FPGA. I chose 256×1S configuration. This configuration provides 256×1-bit RAM. In this configuration, distributed RAM is in a single-port mode. For a single-port configuration, distributed RAM has a common 8-bit address port for writing and reading. This configuration occupies all the four LUTs in a slice. Each distributed RAM module stores one bit of total data width required for symbol's storage (see Table 5.1). The RAM unit that is composed this way provides space for up to 256 symbols. Therefore the unit provides enough storage for puzzles of order 3 and 4. In addition, 8-bit address port can be connected to a compound address bus. In this configuration, the row address occupies the leftmost bits of the compound address bus and the column address occupies the rightmost bits of the compound address. This is not possible when solving higher-ordered puzzles. Therefore I decided to divide the symbols RAM unit into segments. Eight rightmost bits of compound address determine *offset*. The offset is address within one segment. Remaining bits determine address of the segment. The calculation of RAM segments is depicted in Table 5.2.

Data read from RAM is synchronized via output flip-flop. Hence in the case of reading a symbol, the solver must wait one clock cycle until data is ready.

n	$W_a = B_A$	$W_{sa} = W_a - 8$	$S = 2^{W_{sa}}$
3	8	0	1
4	8	0	1
5	10	2	4
6	12	4	16
7	12	4	16
8	12	4	16
9	14	6	64
10	14	6	64

Table 5.2: RAM segments

- W_a - the address width
- W_{sa} - the segment address width
- S - the number of segments

5.1.3 Backtrack stack unit

A stack is a last in, first out (LIFO) type of a computer memory. The designed stack consists of a stack pointer and a stack memory. The stack pointer is up-down counter which stores the address of the next empty memory cell. The memory is composed of distributed RAMs in the same manner as the symbols RAM unit. Stack depth was computed according to Equation 3.2. The computation of stack memory proportions is depicted in Table 5.3. Negative values were rounded up to 0. The width of the stack memory is the same as the address bus width computed since the stack serves as a storage for cell addresses.

n	S_D	$W_a = \lceil \log_2(S_D) \rceil$	$W_{sa} = W_a - 8$	$S = 2^{W_{sa}}$
3	64	6	0	1
4	225	8	0	1
5	576	10	2	4
6	1225	11	3	8
7	2304	12	4	16
8	3969	12	4	16
9	6400	13	5	32
10	9801	14	6	64

Table 5.3: Stack memory segments

- S_D - the stack depth
- W_a - the address width
- W_{sa} - the segment address width

- S - the number of segments

Due to usage of Equation 3.2 some memory resources were saved in stack design. The stack provides standard `POP` and `PUSH` operations. `PUSH` is performed in two clock cycles. The stack pointer points to the first empty position of the memory, therefore in the first cycle, writing to the memory is performed. In the second clock cycle, the value of the stack pointer is increased by one. `POP` operation is performed in three clock cycles. In the first cycle, stack pointer is decreased by one. In the second cycle, actual stack pointer value is presented on the address port of the memory. In the third cycle, synchronizing flip-flops of RAM are fed and data is available.

5.1.4 Elimination unit

The elimination unit is the backbone of the solver's design. The unit contains the set of candidates' bitmaps. The bitmaps are clustered according to the type of the virtual line. There are row, column and block bitmaps clusters. The top module of the solver drives operating mode of elimination unit via `MODE` input signal. The unit provides three operating modes encoded to 2-bit representation as follows:

- `0X - READ`
- `10 - ELIMINATE SYMBOL`
- `11 - RESTITUTE SYMBOL`

The leftmost bit of `MODE` input signal is used as a write enable signal for flip-flops. The usage of flip-flops allows to perform logical operations over the bitmaps in one clock cycle. This is because flip-flop output is connected to the one of LUT inputs. LUT performs selected logical functions and feeds flip-flop input with results. Each result is read into flip-flop on the next clock rising edge.

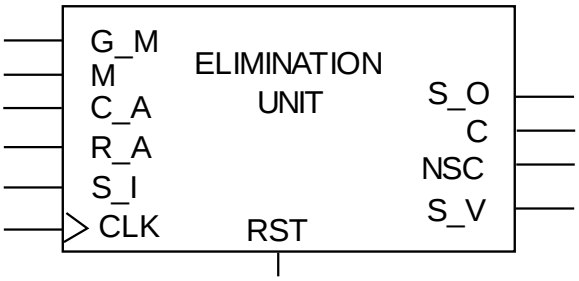


Figure 5.1: Elimination unit

In `READ` mode, the unit provides various operations over the bitmap that represents the intersection of sets of virtual lines . The first of them is naked single candidate detection provided by one-hot detector. The unit outputs result on the next clock rising edge. The result is presented via a dedicated output port. Another function of this unit is decoding the highest symbol available for guess. In this case, guess masking should be enabled. A guess mask generator is included in this unit. Guessing is performed by a priority encoder with two outputs. An encoded symbol value is presented via the first of them. The second 1-bit output reflects whether there is at least one input bit set high. This output is used as conflict detector and this signal is connected to one of unit's outputs. The unit also

detects whether input symbol is valid for the cell specified by input address. This is useful if the solver waits until actual data is ready. Waiting for data is necessary for example after popping an address from the stack. The unit is pictured in Figure 5.1.

5.1.5 NEC unit

The next Empty Cell (NEC) unit derives next empty cell address from given basic address. Basic principles of this unit were described in the above text. The unit is pictured in Figure 5.2.

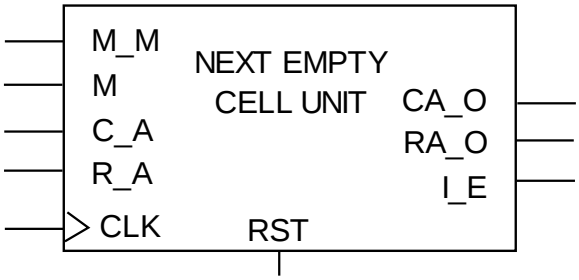


Figure 5.2: Next empty cell unit

This unit operates in four different read modes and two different write modes. Mode encoding is the same as the one for the elimination unit. This allows to connect mode ports of both units to the same driving signal. Determination of the next empty cell address is performed in `READ` mode. The next empty cell address can be determined from given based position in four different ways by feeding `MASK_MODE` input.

Two of these modes are reserved for the purpose of future solver extensions. Remaining modes are used in actual design and these modes should be described. The first mode can be marked as a self-blind mode. In this mode empty cell recognition is provided over cells on higher addresses than the actual cell's address. This is useful when the solver traverses over empty cells. This situation is depicted in Figure 5.3.

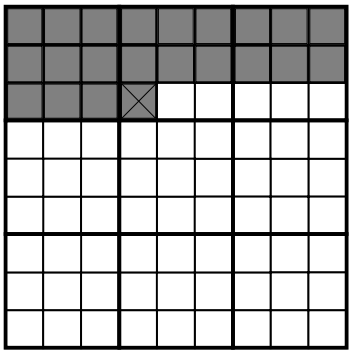


Figure 5.3: Self-blind mode

Actual given position is marked by $\times$ symbol. Grey positions are hidden. If the solver needs to recognize an empty cell within the whole grid, this method is of no use. The solver performs lookup from the begin anchor of the grid. In this case the first cell is blinded. Hence the solver would not be able to fill-in this position. For this purpose another mode is designed. I called it self-sight and this method is depicted in Figure 5.4.

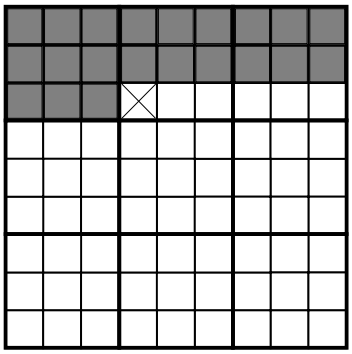


Figure 5.4: Self-sight mode

Actual given position is marked by × symbol. Grey positions are hidden.

5.2 Top module

Top module consists of all the components mentioned above and a finite-state machine (FSM) that provides solving algorithm. Top module mode is driven by dedicated input port. This section describes complete used solving algorithm, and elaborates occupied resources according to puzzle order. The solver interface is also described.

5.2.1 Solving algorithm

The solving algorithm combines two solving techniques, naked single and guessing. This algorithm is performed by a FSM (finite state machine).

The first step is controlling grid on presence of an empty cell. The actual address is set to the begin anchor. NEC logic is set to the mode in which it can detect an empty cell even on the given address (`11 - RESTITUTE SYMBOL`). If there is no empty cell found, the puzzle is solved. The FSM then waits for next commands. Otherwise, the address of the found empty cell is read from the NEC unit. The set of candidates of this cell is checked on presence of naked single candidate. If there is one, respective symbol is written on this position, clusters of the bitmaps are updated. The bitmap inside the NEC unit is updated too. If there was a guess before, the address of the written cell is pushed onto the stack. The performed step of naked single technique is marked into a *naked single flag register*. This register reflects whether any step using naked single technique was performed or not. If the naked single candidate is not present within the set of candidates, the search continues to the next empty cell. After the grid was traversed, the naked single flag register is checked. If at least one step of the naked single technique was performed, the algorithm returns to the first step. Otherwise, a guess is performed.

The guess is executed over the first empty cell within the puzzle grid. The guessed symbol is written into the symbols RAM unit and respective bitmaps are updated. The address of position is also pushed onto the stack. All these tasks are performed in parallel. After guess, search for a naked single candidate is performed. Searching for naked singles after each guess is able to reduce search tree and therefore speed-up the solving process.

If there was a guess before, a conflict can be detected. Conflict detection is performed in parallel with the naked single candidate search. On conflict, the solver backtracks to the first cell with untried candidates within the backtrack path. Backtracking is performed

step by step over written cells. On every step, cells and their respective candidates are restituted. Simplified diagram of described finite state machine is depicted in Figure 5.5. Results of algorithm performance benchmark are presented in Section 6.

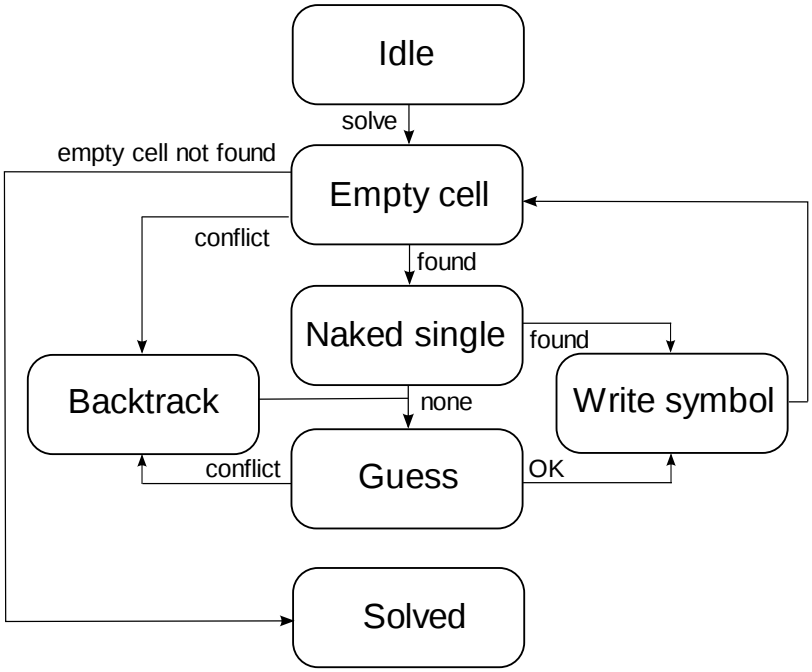


Figure 5.5: Finite state machine of the top module

5.2.2 Occupied resources

Table 5.4 depicts occupied resources of the target chip according to the order of puzzle. With growing place occupied by the design, maximum achievable operating frequency is derogated. This is due to signal delay caused by long interconnecting wires between logic. Maximum frequency was reached while setting global optimization goal of the synthesizer to speed and allowing register duplication.

n	Occupied slices	Utilization	f_{max} [MHz]
3	296	1%	166
4	801	2%	125
5	2040	5%	91
6	3717	9%	71
7	6395	16%	62.5

Table 5.4: Resources occupied by design and maximum achievable frequency

- f_{max} - maximum achievable frequency

Occupied resources related to orders over 7 are missing in the table. This is because synthesizer was unable to efficiently synthesize generic priority encoder of 64 and more inputs. Synthesizable priority encoder could be re-designed as one of design improvements.

5.2.3 Solver interface

In this section, I will provide reference to the solver interface for the case of future testing or utilization of the solver. There are five relevant input ports and two output ports in the design (see Figure 5.6). The unit is synchronized via clock network of the target chip. Port tags are described in Table 5.5.

Port	Description	Signal width [bits]
RST	Reset signal input	1
M	Mode signal input	2
R_A	Row address input	$\lceil \log_2(n^2) \rceil$
C_A	Column address input	$\lceil \log_2(n^2) \rceil$
S_I	Symbol input	$\lceil \log_2(n^2) \rceil$
S_O	Symbol output	$\lceil \log_2(n^2) \rceil$
SOLVED	Status output	1

Table 5.5: Unit interface description

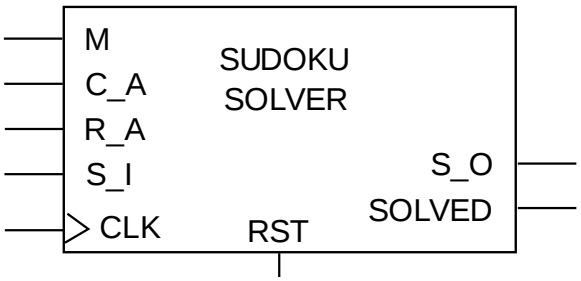


Figure 5.6: The solver unit

At first, the unit should be reset via synchronous RST signal. Reseting the unit causes initializing all the bitmaps and clearing data from RAM modules. The solver provides four specialized operating modes via 2-bit M input port. Those modes allows to read a puzzle into the unit, to read a puzzle from the unit, to instruct unit to solve the previously read puzzle or to stay idle. Encoding of these modes is as follows:

- 00 - IDLE
- 01 - READ SYMBOL FROM GIVEN ADDRESS
- 10 - WRITE SYMBOL ON GIVEN ADDRESS
- 11 - SOLVE THE PUZZLE

Reading a puzzle into the unit is provided in the WRITE SYMBOL ON GIVEN ADDRESS mode. The symbol directed to S_I port is written on the address specified by values of the R_A and C_A ports. Write operation is executed on rising clock edge. Writing symbol on particular position should be performed exactly once. Multiple writing of a symbol

causes inconsistence of sets bitmaps. There is no control mechanism inside the unit. This mechanism is one of proposed extensions of the unit.

After the last symbol has been read into the unit, the unit should be set to the `SOLVE THE PUZZLE` mode. In this mode, the unit is solving the previously read puzzle utilizing algorithm introduced in Section 5.2.1. Status of solving process is reported by a value of the `SOLVED` port. If the puzzle is solved, the value is set high.

The mode `READ SYMBOL FROM GIVEN ADDRESS` allows to read solved puzzle out of the unit. The symbol present on the address specified by specified by values of `R_A` and `C_A` ports is passed to the `S_O` port.

If some serial transfer protocol is used to transfer data into, or from the unit, the `IDLE` mode is used to wait until buffer is filled. The unit performs no operation in this mode. In the oncoming chapter, the results of solver's testing will be presented.

Chapter 6

Solver testing

In the case of solving SUDOKU instances of various orders, the solver's performance is compared to the performance of *TU Delft Sudoku Solver on FPGA*, denotated as the reference solver on this text. This solver won 2^{nd} price of FPT'09 Design Competition Award. I failed to find any reference to the winning design. In the reference solver, guessing based on exhaustive brute-force search was only used. This solver was unable to solve any of benchmarks of order above 8 in less then 24 hours [6]. The goal of my design was to achieve better results than the reference solver. For this purpose, I supported exhaustive search with naked single candidate elimination technique as described above. Using this technique leads to distinctive better results in some cases. Those results will be presented and discussed in the forthcoming text.

6.1 Solver benchmark

Used benchmarks were the same as those used in testing process of the reference solver. These benchmarks were provided by [1] for testing solvers participating in the mentioned design competition. The reference solver's results were presented in [6]. Two types of benchmarks were provided, namely *a* and *b*. The *a* group benchmarks are predicted to be easier solved by hardware solver. These from the *b* group are considered to be very hard to solve by hardware solvers in general. High-ordered puzzles from this group are unsolvable using a brute-force search in less than 24 hours. My solver was unable to solve these puzzles, unfortunately, it failed to solve some of the puzzles from the *a* group too.

I tested my solver using this benchmarks in simulations. I used integrated simulator of ISE, Isim. Benchmark puzzles of order 3 up to 10 were successfully solved. Solving process of particular benchmark puzzles with comments is depicted in Figures 6.1 - 6.3. Benchmark results will be presented in the next section.

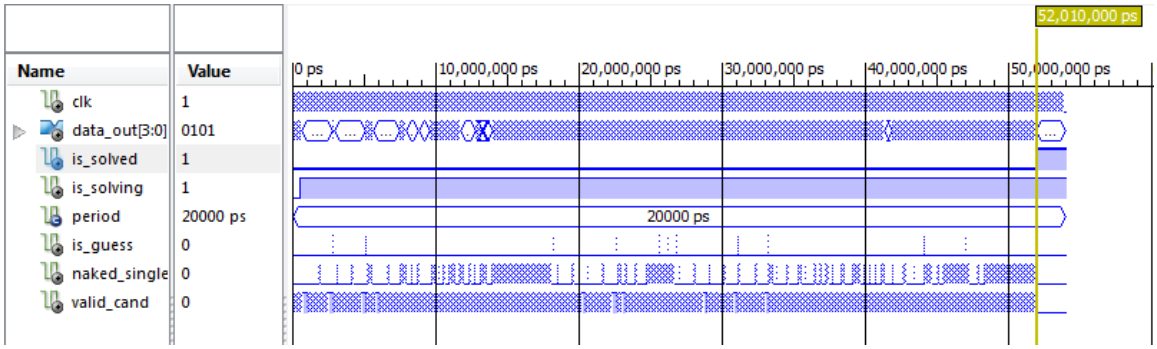


Figure 6.1: Benchmark 03a solving process

This benchmark puzzle was solved in a few μs. Guess was performed 11 times. This puzzle is well-posed for naked single candidate technique.

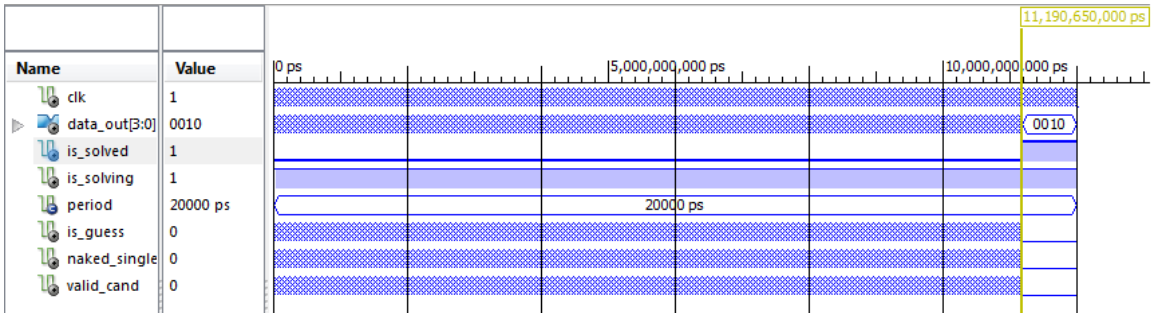


Figure 6.2: Benchmark 03b solving process

Solver needs evidently more effort to solve this puzzle. This is caused by many guesses performed during solving process. Naked single candidate technique is not effective in this case.

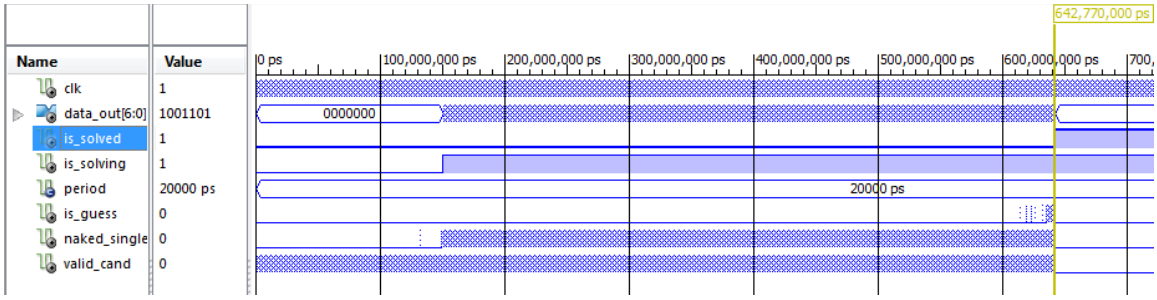


Figure 6.3: Benchmark 10a solving process

In this case, almost all of the cells were filled-in using the naked single method. Some guesses were performed in the last part of the solving process. The puzzle, that is unsolvable by a brute-force, was solved in a less than 0.5 ms.

6.1.1 Benchmark results

In this section, I will present results of solver benchmarks. Maximum achievable frequency (t_{fm}) of my design reflects design performance on the particular chip. This is not good subject of comparison with the reference solver, since the reference solver operating frequency was set to 50 MHz. Operating frequency was set to 50 MHz in simulations. Results are presented in Table 6.1.

The naked single candidate method was not applicable on all the benchmarks. In benchmarks with many guesses executed, solver performance drastically decreased. Therefore, this technique is not suitable to reduce search-tree of exhaustive search. On the other hand, naked single candidate technique was very effective in solving puzzles from the *a* group.

B_i[1]	t_{ms}[s]	t_{tud} [s]	$A_a[s]$	A_r
03a	0.000052	0.020821	0.020769	400.40×
03b	0.011900	0.012460	0.000560	1.057×
04a	0.033764	0.221379	0.187615	6.56×
06a	0.000050	0.115347	0.115297	3251.43×
06b	N/A	N/A	N/A	N/A
07a	0.000065	0.211343	0.211278	2306.94×
08a	0.000130	0.096424	0.096294	741.72×
09a	0.000253	N/A	N/A	N/A
10a	0.000493	N/A	N/A	N/A

Table 6.1: Benchmark time and raw acceleration

- B_i - benchmark identifier
- t_{ms} - solving time of designed solver
- t_{tud} - solving time of reference solver
- $A_a = t_{ms} - t_{tud}$ - absolute acceleration
- $A_r = \frac{t_{tud}}{t_{ms}}$ - relative acceleration

Average relative acceleration (A_{ra}) of my design is computed as follows:

$$A_{ra} = \frac{\sum A_r}{6} = 1118.02\times \tag{6.1}$$

In comparison with the reference solver, my design seems to be much better in solving the same benchmark puzzles. This is caused by effectivity of the naked single candidate search method in some cases. These cases are investigated in the forthcoming text.

I decided to elaborate how long takes to correctly fill-in one cell in a particular puzzle. The prediction is that in larger puzzles, it takes more time to find a naked single candidate. Hence time needed to correctly fill-in one cell should increase with increasing puzzle proportions. Results of this elaboration are depicted in Table 6.2

B_i[1]	C_t	C_e	C_{er}	t_{ms1}[ms]	t_{tud1} [ms]
03a	81	55	67.90%	0.000945	0.378563
03b	81	58	71.61%	0.205172	0.214828
04a	256	146	57.03%	0.231260	1.516295
04b	256	168	65.63%	N/A	N/A
05a	625	300	48.00%	N/A	N/A
05b	625	313	50.08%	N/A	N/A
06a	1296	401	30.94%	0.000125	0.287648
06b	1296	650	48.23%	N/A	N/A
07a	2401	601	25.01 %	0.000108	0.351652
08a	4 096	1 025	25.03%	0.000127	0.094072
09a	6 561	1 641	25.01%	0.000154	N/A
10a	10 000	2 502	25.02%	0.000197	N/A

Table 6.2: Cell fill acceleration

- C_t - cells total
- C_e - cells empty
- C_{er} - cells empty ratio
- $t_{ms1} = \frac{t_{ms}}{C_e}$ - designed solver one cell fill time
- $t_{tud1} = \frac{t_{tud}}{C_e}$ - reference solver one cell fill time

Time required by the solver to correctly fill in a cell depends on an empty cells ratio. Puzzles with less than $\frac{1}{3}$ of empty cells contain many naked single candidates. Therefore cells within these puzzles are correctly filled in a constant time. Consequently, the solver is able to solve high-ordered puzzles in reasonable time. It seems that the solver should be able to solve any high-ordered puzzle by applying more logical constraints techniques.

6.2 Design verification using ChipScope

The solver does not dispose of a communication interface. For the purpose of solver verification on real hardware, I programmed a benchmark unit. The solver is included into this unit. This unit provides reading puzzle into the unit and drives operations performed by the solver. Status of solving process is reported via solver output signal. At first, I created ICON core with one control output. Then I created ILA core with two different inputs. The first input was reserved to symbols output of solver and the second one to status signals. The number of collected data samples was set to 2048. I performed a few verification tests of designed solver. ML605 kit was connected to the PC via USB. I successfully downloaded design bitstream to Virtex-6 chip. Sampled data was displayed by ChipScope software in a waveform. The process of solving benchmark 05a is depicted in Figure 6.4.

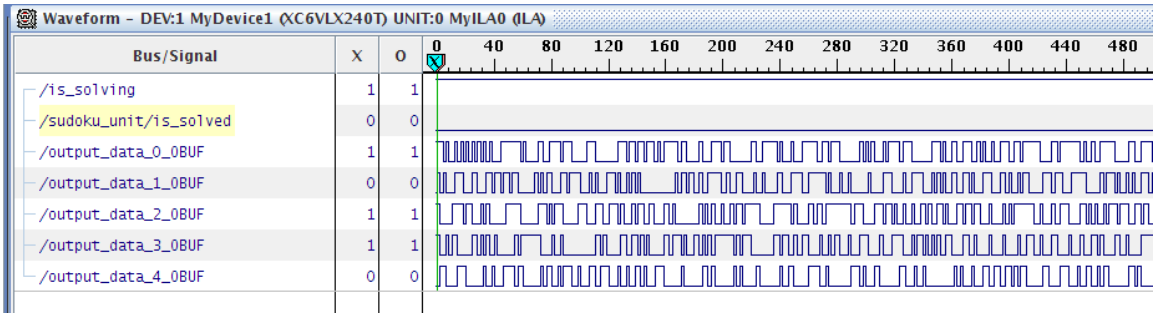


Figure 6.4: Solving process of the benchmark 05a

The solver was unable to solve this puzzle within a few minutes. The solving process was terminated after approximately five minutes. This figure proves that the solving process was performed successfully on the target chip. In Figure 6.5, there is shown waveform that represents data samples after benchmark 03a was solved.

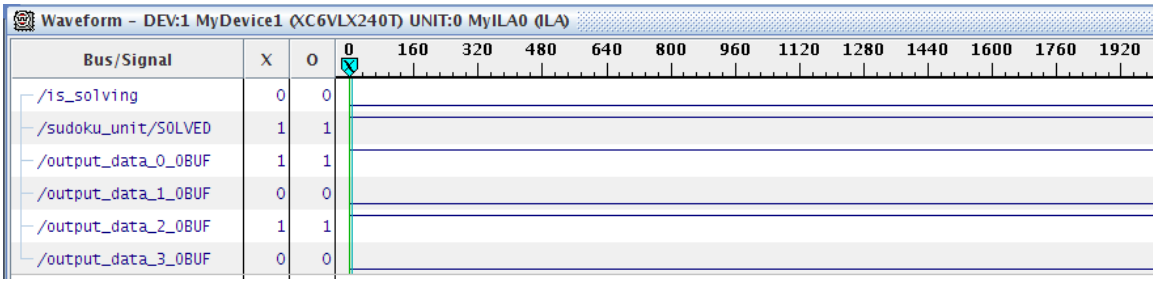


Figure 6.5: Benchmark 03a solved

The solver was able to solve benchmarks on the real hardware platform, hence results measured in simulations were proven.

Chapter 7

Conclusion

In this work, I described principles of the SUDOKU game. I provided investigation on possible puzzle solving methods and hardware-based implementation of them. Two of these methods were chosen and successfully implemented. A solver was designed using a hardware description language, namely the VHDL. I measured solver's performance using specialized benchmarks dedicated to hardware SUDOKU solvers. My solver was able to solve most of these benchmarks in a very short time. Obtained results were compared with the results of another hardware-based solver which was awarded in an international chip design competition. This work provided me with the opportunity to work with one of the newest evaluation kits (ML605 by XILINX). The designed solver was successfully verified on this kit with ChipScope software.

One of the solver's liabilities is that it is unable to solve all the puzzles. Solver's performance could be increased by using more solving techniques based on logical constraints. The hidden single candidate technique, described in this work, could be one of them. Its implementation is partially incorporated in my chip. However, an extension of a gathering unit is necessary to be added.

Another reasonable extension of the solver is an input and output communication unit. The solver interface was described to provide reference for future extensions. The communication unit can transfer data over any serial communication protocol due to solver's ability to wait until input data is present.

I found puzzle solver design very interesting. I have learned new facts about hardware design and programmable logic devices as well as about SUDOKU principles. I hope I will continue in this research. In my future work I would like to implement the extensions proposed in the text above. I assume I might be able to solve all the SUDOKU puzzles of order up to 10 in a reasonable time.

Bibliography

[1] FPT'09. Design competition benchmarks [online]. `http://fpt09.cse.unsw.edu.au/comp/benchmarks.html`, [cit. May, 2010].

[2] Gordon Royle. Minimum sudoku [online]. `http://units.maths.uwa.edu.au/~gordon/sudokumin.php`, [cit. May, 2010].

[3] Helmut Simonis. Sudoku as a constraint problem. In *4th Int. Works. Modelling and Reformulating Constraint Satisfaction Problems*, pages 13–27, 2005.

[4] Sunggu Lee. *Advanced Digital Logic Design: Using VHDL, State Machines, and Synthesis for FPGAs*. Thomson, 2006. ISBN 0-534-46602-8.

[5] Tom Davis. The matematics of sudoku [online]. `http://www.geometer.org/mathcircles`, [cit. May, 2010].

[6] C. van der Bok, M. Taouil, P. Afratis, and I. Sourdis. The tu delft sudoku solver on fpga. In *Int. Conf. on Field-Programmable Technology (FPT)*, pages 526–529, 2009.

[7] Wikipedia. Sudoku [online]. `http://en.wikipedia.org/SUDOKU`, [cit. May, 2010].

[8] Xilinx. Video training: Chipscope pro software overview [online]. `http://www.xilinx.com/training/fpga/chipscope-pro-training-video.htm`, [cit. May, 2010].

[9] Xilinx. Virtex-6 fpga clb user guide [online]. `http://www.xilinx.com/support/documentation/user_guides/ug364.pdf`, [cit. May, 2010].

Appendix A

CD-R contents

1. source codes
2. benchmark puzzles