Verilog Sudoku Solver

Magson Gao, Shreeyam Kacker

Overview

The aim of this project is to design and implement a Sudoku solver using digital logic written in Verilog HDL. In order to implement this solver it is necessary that the digital logic is written to follow the rules of Sudoku, use several key techniques for solving harder Sudoku and when it is necessary, how to guess values for cells in the Sudoku grid.

The techniques that will be implemented include, but are not limited to, the candidate line, naked group and hidden group methods. Whilst these methods are sufficient for solving most novice to intermediate level Sudoku, the most difficult Sudoku require backtracking. The previous guesses will be stored on a stack based architecture to allow for previous states of the grid to be restored. The purpose of implementing the aforementioned techniques is to significantly reduce the search space required for guessing, allowing the solution to be found in fewer clock cycles and requiring less memory to store previous guesses for the typical Sudoku. The FPGA that will be used is that on the Digilent Nexys 4 development board.

Block Diagrams

The Sudoku solver should be able to take data inputs from the camera and display the solved Sudoku on an external monitor via VGA cable. It is also desirable that a tutorial mode is implemented which will teach users how to solve a given Sudoku. A possible hardware architecture is given in Figure 1.

Modularized Block Diagram

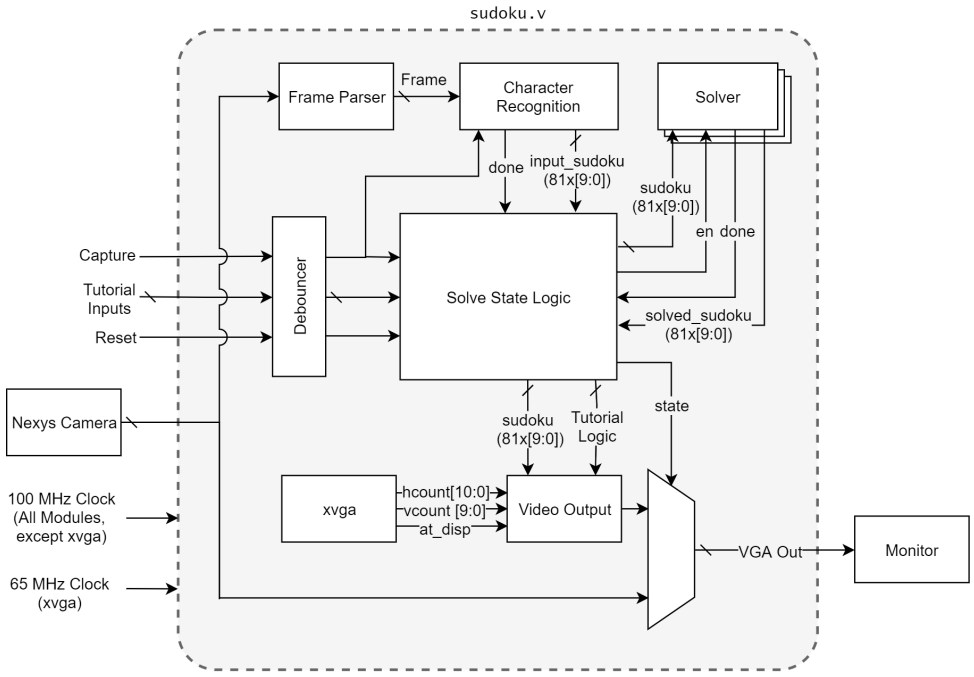

Figure 1: Initial high level functional diagram split up into Verilog modules

The functions performed by the system is given in Figure 2. These states will be managed in the Solve State Logic block shown in Figure 1. The Sudoku solver logic will contain multiple sub-states which are given in Figure 3.

Functional Diagrams

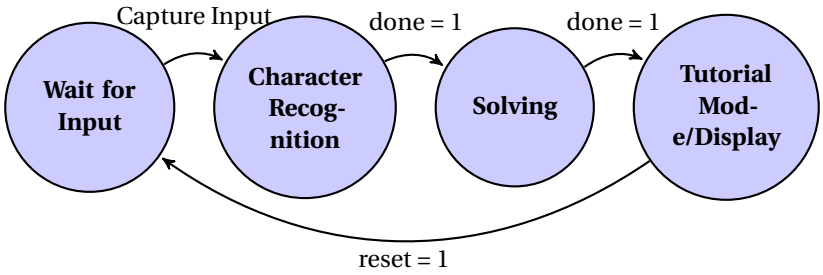

Figure 2: Overall functional diagram showing flow between overall states, with tutorial mode if such a stretch goal is implemented, otherwise displaying the solution on screen,

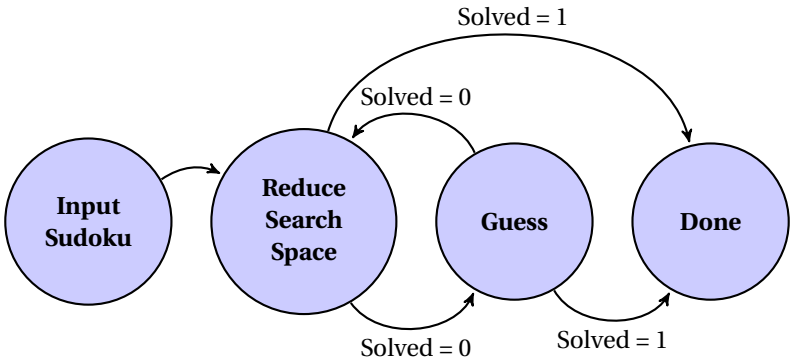

Figure 3: The high level functional diagram for the Sudoku solver, showing high level states.

Nomenclature

In this section, the nomenclature used in the subsequent sections is defined. The squares containing a single digit in Figure 4 is defined as a cell and the 3 by 3 block of cells highlighted by the thickened border is a square. Each 9 cell row and 9 cell column are referred to as a row and column respectively.

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

Unsolved Sudoku

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

Solved Sudoku

Figure 4: Example unsolved and solved Sudoku puzzles.

Goals

Baseline

The baseline requirements for the project are to be able to solve intermediate Sudoku, with some ability to recognize Sudoku puzzles from an image that can be uploaded via a ROM or some other

input method such as an SD card.

In order to be able to solve intermediate Sudoku puzzles, numerous the techniques that shall be implemented are as follows:

Single Position

From information given on the board in terms of filled in rows, columns, and squares, the solver must be able to fill in a value if there is only one possible value for a particular cell can take.

Candidate Lines

The candidate line technique narrows the state space by calculating where a number cannot go. This can be done by observing the possibilities for placement of a number in a box. From there, if all the possible placements in that particular box lie on a line, then all other possibilities for that number are not possible in other cells on that line.

Expected

Expected requirements are to be able to solve hard Sudoku puzzles, and to have some way of inputting a board through video.

In order to solve hard Sudoku puzzles, the following techniques shall be implemented:

Naked Pairs/Triples

A naked group in a Sudoku puzzle is when a set of numbers $S = \{A_1, A_2, ..., A_n\}$ can exist in n places along a candidate line. From here, it can be reasoned that along that line, none of the other squares can contain any of the elements in S.

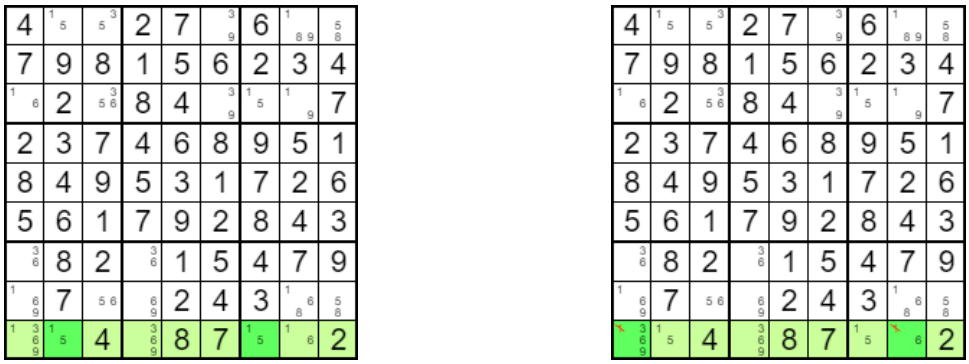

Figure 5: The naked pair in this set is $\{1,5\}$, and after removing the other possible values for $s \in \{1,5\}$ from the other parts of the grid, it's possible to fill in a 6 in the rightmost square [2].

Hidden Pairs/Triples

The hidden set (or unique subset) technique is a super-set of the naked subset technique, where the possible n locations for the n numbers contain other numbers which are in fact impossible. This can be proven by contradiction. If any of the n positions were to take a value which was not in the set S, only $n-1$ positions exist to place n numbers, thus the Sudoku cannot be solved.

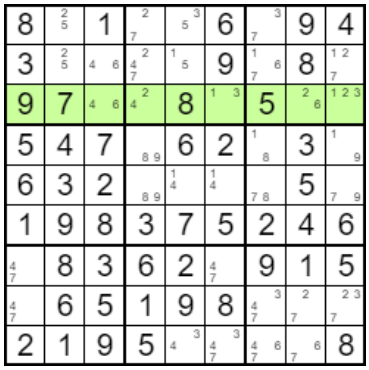

Figure 6: Sudoku grid showing the hidden triple {1,3} that can allow us to remove the 2 in the line [2].

Video Display

The captured state of the board or the current camera input will be displayed on screen and the user will be able to interact with the system by being able to input Sudoku puzzles via camera.

Stretch Goals

For stretch goals, there must be some high level of user interact-ability with the system, and the system must be able to solve every Sudoku puzzle input.

State Space Search

In the case where the usage of all other techniques combined cannot collapse the state space enough to obtain a unique solution, a state space search will have to be implemented from the possibilities of which numbers can exist on each cell in the board.

Tutorial Mode

The user will be able to interact with the puzzle after a solution has been found. Instead of the solution being displayed after solving, a tutorial mode will be implemented where the user can choose to input a number in an empty cell, which will illuminate either red or green depending on if the answer is correct or not. The intended usage of this is as a learning tool for the user to practice Sudoku puzzles in a guided manner.

Implementation

In this section blocks from Figure 1 are explained in detail and approximations of the hardware and time costs are made where appropriate.

Debouncer

Debouncing is necessary on the inputs in order to prevent them being double counted due to their electrically noisy mechanical contacts. This can be especially problematic for the tutorial mode inputs and can be frustrating for the user if they cannot move the cursor to the position they desire. The debouncer Verilog code will be obtained from previous verified work thus does not need further testing.

Frame Parser & Character Recognition

The frame parser will be responsible for collecting data from the camera. Once enough data has been collected from the camera to form a single frame, it will be moved from an internal buffer to the output and a signal will be sent to indicate that a frame has just been parsed.

In order to have a manageable data rate for subsequent modules, the photo of the Sudoku will be cropped and re-scaled.

Character recognition from this input video data will be computed by comparison with a set of templates from which input will first be aligned to and then checked against, with the most probable character being selected. Done signals are output to allow the central main state machine to keep track of internal states without using multiple counters. In order to estimate the required hardware costs for this block it is necessary to make the following assumptions: i) the characters can be stored as 16 pixel by 16 pixel black and white images where each image is 1 bit in size. ii) the threshold of determining whether a pixel is black or white is based off the mean sum of red, green and blue pixel components for each pixel in a 1024 × 1024 pixel image where each pixel is 24 bits iii) the Sudoku grid covers the vast majority of the camera image. These assumptions can be justified by observing that the characters of the Sudoku grid in Figure 7 is still readable where the original Sudoku image has been re-sized and converted to black and white.

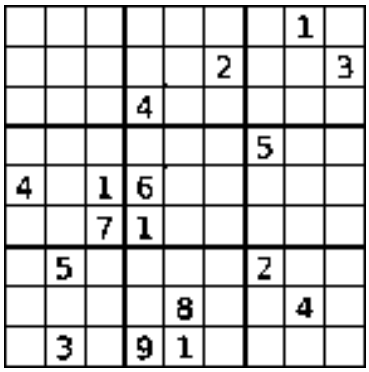

Figure 7: A 1024 by 1024 pixels image converted to only black and white pixel values and scaled to 144 by 144 pixels. The size of each cell is 16 by 16 pixels.

To determine the threshold for a black versus a white pixel a 5 bit adder is used on 1024 × 1024 pixels. To reduce the size of the resulting number, the red, green and blue values have to be normalized such that the red, green and blue pixels consist of the three most significant bits of the original 8 bit values. The shifting operation to reduce the bit sizes does not add to the hardware requirements. The summation of each of the 3 bit values results in a 5 bit number per pixel. The average can be approximated by adding rows of 1024 pixels together and shifting to the right by 10 bits. This results in the average pixel intensities for each of the 1024 rows. Rows with an average intensity less than 8 are assumed to not contain the Sudoku as the Sudoku grid has a white background. To prevent the threshold from being skewed due to these rows the value of the previous row is taken. The average is finally approximated by adding together all the 1024 × 5 bit numbers and shifting to the right by 10 bits. The resulting 5 bit number is used as a threshold to determine whether a pixel is black or white, thus a 5 bit comparator is required. The operation of converting raw image to black and white requires approximately 2 clock cycles per pixel, one for the average and one for the comparison. A majority encoder is used to convert 7 pixel by 7 pixel blocks to single pixel values in the resulting 144 pixel by 144 pixel image used for template matching. The majority encoder takes in 49 inputs which in the worst case is mapped to 12 × 6 input look up tables on the Nexys 4.

In total the hardware and time requirements for the image pre-processing is:

- 2 clock cycles per 1024 by 1024 pixels. Thus the number of clock cycles is 2,097,152.
- Full adders to sum 3 bit values for each of red, green and blue. A 5 bit result register will be needed to store the result.

- A 15 bit register is required to store the summation used to calculate the average intensity per row. 1024×5 bit registers are used to store all the averages.
- A single 6 input look-up table is required to check if the row intensity average is greater than 8.
- A 15 bit register is required to sum the average row intensities.
- In order to compute the averages an adder is required which takes a 15 bit input and a 5 bit input to produce a 15 bit output.
- 12×6 input look-up tables are required for the majority encoder.

The template matching occurs by comparing the 81×256 bits representing each 16 by 16 pixel image of each number in the Sudoku grid with the 9×256 bit number templates. This is achieved using a worst case of 256 AND operations per comparison. The output is added, resulting in 8 bit numbers for each number from 1 to 9. If all the 8 bit numbers are less than 128, the Sudoku square is assumed to be empty. This requires at most 2×6 input look-up tables for all 8 templates as only the most significant bit is used in this computation. If this is not the case, whichever template results in the maximum number results is the inferred number in the cell.

In total the hardware and times requirements for the character recognition is:

- 81 clock cycles are required to template match each of the 81 cells.
- 9×256 bits are required in ROM to store the templates for numbers from 1 to 9.
- Each cell must be compared with each template, thus 9×256 AND operations are required.
- Each template requires 256×1 bit numbers to be added together. This results in the requirement of 128 half adders, 64×2 bit adders, 32×3 bit adders, 16×4 bit adders, 8×5 bit adders, 4×6 bit adders and a single 7 bit adder.
- The comparison operation between the resulting 9×8 bit numbers can be computed using 12×6 input look-up tables.
- 2×6 input look-up tables are required to check if a cell is empty.

Solver

An initial implementation which was considered used a naive binary representation of each number on the Sudoku grid. The hardware requirements to determine which numbers were missing a row, column or square would involve 9 comparators for each cell in the Sudoku grid as each cell can take a value from 1 to 9. For each of the 27 sets of cells of cardinality 9 there would be 81 comparators producing 9 single bit results for each of the 9 cells in the set. In order to determine if a set consists of a solved set, 9×9 input OR operations are required. This requirement can be reduced significantly be realizing that the output of the comparators are in fact the one hot encoding of the values in the cell. Thus the solver can instead store each number of each cell as a one hot code, thus removing the need for comparators entirely.

For the chosen implementation the number of bits required to represent a Sudoku grid is 81×9 bits. In order to determine which numbers are missing in the 27 sets, the solver requires $27 \times 9 \times 8$ OR operations (each containing 2 single bit inputs). This is because 8 OR gates are required to compute the OR operation between 9 single bit numbers representing the presence of 1 of 9 possible numbers. This is implementable using $27 \times 9 \times 2$ 6-input look-up tables in the worst case.

To implement the advanced techniques which apply to the possible values of each cell, each cell will require a mask of size 9 bits, formed from AND-ing the 3 masks generated from the row, column and square sets. This results in 9×3 input AND operations, which is implementable using (in the worst case) $9 \times$ 6-input look-up tables. Each mask is modified further using the advanced techniques.

The candidate line technique can be implemented by storing 6 masks, 3 for each column and 3 for each row associated with each of the 9 squares of a Sudoku grid. Each mask contains the possible values which can exist in the row or column. This is computed by OR operations over each of the 3×9 bit masks. By computing the AND operation of each of the 6 masks which every other mask in the row or column (requiring $6 \times 2 \times 9$ bit AND operations per square), a candidate line can be identified by observing all zeros as a result of the 2 AND operations.

The mask modifications from the advanced techniques require significant computation and thus parts of the computation will be done sequentially. In terms of digital logic, the hidden group problem is in fact a super-set of the naked group problem and can be solved using the following algorithm:

1. Assume each cell has a 9 bit mask containing 1s for values which it can take and 0s for values which it cannot take.
2. Using a 4 bit output adder, compute the number of 1s in each mask.
3. For each set in the 27, AND each bit of the mask with the bits in each of the other masks. For each set store the 8 resulting masks.
4. If the number of stored masks which are 0 are equal to $(9 - number\ of\ ones\ in\ initial\ mask)$ and not all the stored masks are 0 then a hidden or naked group is possible.
5. If the number of resulting masks containing the same number of 1s as the initial mask is 1 less than the number of 1s in the initial mask a naked group or hidden group has been detected.
6. If this is the case, replace the masks with the stored masks.

The required hardware for each of the steps is as follows:

1. 81×9 bit masks.
2. The number of 1s in each mask can range from 1 to 9. A simple implementation will use 4×6 input look-up tables. Thus for all cells in a set, 72 look-up tables are required.
3. For each set in the 27, this step requires $72 \times 9 \times 2$-bit AND operations.
4. This step requires approximately 27×2 look-up tables. This is because there are 27 masks per set each of size 9 bits. Thus if 2×6 input look-up tables are used per mask (this is an over-estimate), 54 look-up tables are required in total. Counting the number of 0 masks requires approximately 5×6 input look-up tables. The comparisons (with 0 and with $9 - number\ of\ ones\ in\ initial\ mask$) requires an additional 2 look-up tables.
5. This requires an additional 72 look-up tables to determine the number of 1s in each of the stored masks and an additional 81×9 bit masks. In addition equality must be checked with 9×8 pairs of values. The outputs are summed up using at most 9×4 look-up tables and an additional equality operation is required to compare the number of matches with the number of 1s in the initial mask.
6. 9 bit AND operations are required to modify each of the 81 masks of the grid.

All state space reduction techniques that will be mentioned in the goals section will be implemented as separate modules or functions which are instantiated or are part of the main solver module.

The state space search will use a stack-based approach to allow the state of the Sudoku grid to be restored if a guess is deemed incorrect. The state space search will discard a state if there are no solutions for any of the squares in the grid. The worlds hardest Sudoku requires the optimal solver to guess values in the grid which are determined either correct or incorrect 10 guesses later. As the solver is not doing a comprehensive state space reduction and guesses will not be made optimally, it is

assumed that 50 guesses may be required. As the state of the grid is saved on each guess, $50 \times 81 \times 9$ bits are required for the stack.

Video Output

Video output will be done using the xvga modules previously used in labs. Depending on the state, the video sent to the monitor may be equivalent to that coming from the camera, or it may be switched to the Sudoku board that has been captured from before either solved and in tutorial mode or displaying it on the screen, or in the state of being solved.

Testing the state machine logic will be done in a test bench, while testing the video output will be done visually, as it is expected that this will be faster than creating a large test bench.

Solve State Logic

The solver logic needs to keep track of approximately 4 states. Transition between the states occurs based on the done signals as shown in Figure 1. The state can thus be encoded in a 2 bit variable. The required logic is negligible compared to the solver and character recognition blocks.

Testing

The solver module will be evaluated by a testbench containing a set of multiple randomly generated Sudoku for each difficulty level. Each Sudoku will be generated using the QQWing Sudoku generator [4] as it provides not only the Sudoku itself but a list of the required algorithms used to solve the Sudoku puzzle and the number of required guesses. The testbench will also include the "World's Hardest Sudoku" [1] which is one of the approximately, 50,000 minimum hint Sudoku each of which contain only 17 of the 81 cells pre-filled [3]. The testbench contains 4 modes: simple, easy, intermediate and hard. The simple mode tests the basic solver functionality to represent the grid as a one hot code and compute row, column and square solves when only single possible value can be inserted. The easy mode tests this with real world Sudoku generated from the QQWing generator where multiple possible values are initially possible for each square but advanced techniques are not required. The intermediate Sudoku require these advanced techniques but do not require backtracking. The hard Sudoku require all the aforementioned techniques as well as backtracking. The solver will be tested in order of increasing difficulty as it will be difficult to determine which technique is used to solve the Sudoku (any Sudoku can be solved given a long enough time and sufficient memory using backtracking).

Character recognition and frame parsing will be tested with preset ROMs taken from the camera first in a testbench, before being tested dynamically with raw camera input directly on the board.

Similarly, the overall solver state machine will also be tested in a testbench before integration testing it with all the other components.

Required Hardware

1. Nexys Camera for state recognition.
2. Nexys 4 DDR Board.
3. Monitor with VGA input.

All hardware mentioned above can be found in the lab and hence nothing needs to be bought for this project.

Timelines

Week of 2018-10-29

- Project organization, division of tasks, hardware collection

Magson:
- Refinement of single position techniques, initial work on candidate line and double pair techniques.

Shreeyam:
- Learn how to interface Nexys camera with board and obtain an image displayed on the screen.

Week of 2018-11-05

Magson:
- Continued work on the candidate line technique, initial work on naked/hidden pairs and triples techniques.

Shreeyam:
- Initial work on character recognition, including being able to locate the board and all cells from an input ROM and recognize each character.

Week of 2018-11-12

Magson:
- Continued work on naked/hidden pairs techniques, initial work on state space search.

Shreeyam:
- Initial tests with camera, ability to display Sudoku board on a monitor, with intelligent switching to toggle between input video and stored Sudoku board.

Week of 2018-11-19

Magson:
- Refinement and optimization of state space search techniques.

Shreeyam:
- User input, interactivity with output video.

Week of 2018-11-27

- Total subsystem integration and refinement.

Week of 2018-12-03

- Polishing, integration testing, final report.

Week of 2018-12-10

- Final report, project submission.

Bibliography

[1] Artika Inkala. In: (2010). URL: http://www.aisudoku.com/index_en.html.

[2] Astraware Limited. In: (). URL: https://www.sudokuoftheday.com/techniques/.

[3] Gary McGuire, Bastian Tugemann, and Gilles Civario. "There is no 16-Clue Sudoku: Solving the Sudoku Minimum Number of Clues Problem". In: *CoRR* abs/1201.0749 (2012). arXiv: 1201.0749. URL: http://arxiv.org/abs/1201.0749.

[4] Stephen Ostermiller. In: (). URL: https://qqwing.com/generate.html.