Nonogram Solver Final Report

Nina Gerszberg
Dept. of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
ninager@mit.edu

Veronica Grant
Dept. of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
vgrant@mit.edu

Dana Rubin
Dept. of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
Danaru@mit.edu

***Abstract*—This project aims to solve a nonogram the fastest way possible through parallelizing the solving process. Nonograms are picture logic puzzles which were invented in the 1980s. They are represented as a grid in which cells in a line must be colored or left blank according to a set of numbers that each line has. Each number in the line's set requires coloring of a consecutive group of cells, followed by at least one blank cell before the next coloring. Solving nonograms is an NP-complete problem which has different solution techniques.**

I. Introduction

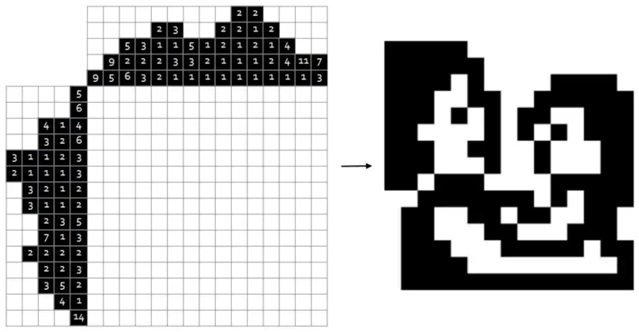
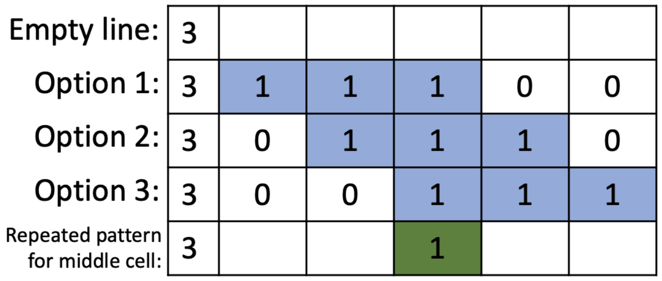


Fig. 1. Empty nonogram board (left) and solved board (right)

Nonograms are grid logic puzzles which have a set of numbers for each row and each column; we'll refer to this set as "constraints". The numbers in the constraint set represent how many unbroken consecutive cells should be filled in, or colored, we'll refer to a group of n number of consecutive colored cells as "block". Between each two neighboring blocks there has to be at least one blank, uncolored, cell. We will refer to a colored cell as having value of "1" and blank as having value of "0".

To solve a Nonogram board one essentially needs to determine and assign which cells will be blocks and which will stay blank. There are several methodologies to extract a cell's value. Here, we will aim to incorporate the following methods in our design:

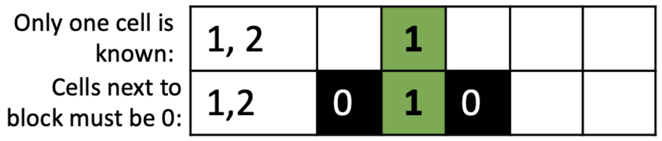
- simple blocks: for a given line, one can determine cells' assignments as "1" if a number in the constraint set is larger than $n/2$. It can be done recursively by breaking the board to smaller pieces.



Empty line:	3					
Option 1:	3	1	1	1	0	0
Option 2:	3	0	1	1	1	0
Option 3:	3	0	0	1	1	1
Repeated pattern for middle cell:	3			1		

Fig. 2. simple blocks - blue cells are valid assignments to "1", then middle cell can be deducted-green.

- simple blanks: Similarly to the simple blocks idea, one can determine cells' assignments as "0" if we have an unknown cell next to a known block.



Only one cell is known:	1, 2		1			
Cells next to block must be 0:	1,2	0	1	0		

Fig. 3. simple blanks -The green cell is known to be "1", for the first assignment, so we can deduct that the cells next to it are known to be-"0".

- Forcing: When there is a gap of n unassigned cells between two cells that are assigned to be "0" and $n <$smallest number on board, one can assign the cells in the gap to "0".

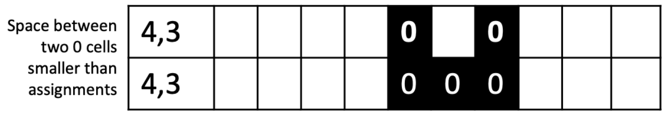


Fig. 4. Forcing -First row is given, we can deduct the middle cell is also "0" since the gap is too small to fulfill any block.

In mathematical perspective the Nonogram problem can be viewed as a constraint satisfaction problem (CSP). There have been several Nonogram solver implementations in the past, from solving Nonograms by combining relaxations iteratively on the Nonogram's lines, [3] to using evolutionary algorithms or a combination of heuristic solvers on the Nonogram. [2]

Here, we aim to apply continuous line simplifications using the methods described before in a parallel way. We chose to represent a board and its constraints in a computational form using the disjunctive normal form (DNF) formula. DNF can be described as applying OR logic to several groups of AND logics, or as a sum of products in boolean logic. [4]

II. Board representation

To translate a given board into a DNF we programmed a python implementation which generates all valid options for a row or column given the constraints set forth by the board specification. Over all these valid options, each of which is represented as a combination of ANDs logic, we apply OR logic as we eventually aim to get *one* of these options right. After combining all the valid options for each line in the above manner, we send this board representation as long sequence bits, which is further described in Communication section.

III. Storage

The main method of storage for this program is a FIFO queue. Using the Xilinx fifo_generator, the current FIFO we are using has enough space to store a 11x11 board. The FIFO has width 16 and depth 2048 (22x84). The depth is that amount since the worst number of possible options for a puzzle line of size 16, and there are 22 total rows and cols. The number of possible options for a row is a combinational problem. As stated in the introduction, each line has a number of blocks of colored spaces (b) and a number of blank spaces (s). An option is a unique ordering of the colored blocks and blank spaces, so we can use n choose k, or binomial coefficients, to find the total number of options for a given b and s which $\binom{(b+s)}{b}$. The combination that maximizes the number of options for a 11x11 board is $b = 9$ and $s = 3$ as 9 choose $3 = 84$.

Since the FIFO is the size of the worst possible outcome for an 11x11 board, it is important to store how many options each line has stored in the FIFO. This is stored in a 2D array such that the puzzle line index will map directly to the number of options a particular line has which is equal to the number of entries in the FIFO. These are the most important data structures for this design, and there is still space left for other transient information.

The FIFO format is as follows:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
line index															
option #1															
...															
line index															
...															

A line index is an arbitrarily assigned number to uniquely identify the row or column that the constraints belong to. Line indexes are from 0 to (m+n)-1 for an mxn board. The line index goes first and everything following it until the next line index are the constraints for that line. An option is one possible sequence of 1s and 0s that obey the constraints for a given line. We only consider the first m or n bits, so the current FIFO configuration could support a max size of 16. Using the FIFO and the options per line, we are able to store all the information necessary for the solver to work correctly.

IV. Communications (Veronica)

The game board is stored on a PC, so it must be transmitted to the FPGA to be solved in a readable format and then sent back to the PC for display. The protocol we used to do this communication was UART with a baud rate of 9600 that contains 1 start bit, 8 data bits, no parity, and 1 stop bit.

A. PC to FPGA

On the PC side, PySerial is used to send the bits representation of the board to the FPGA using the protocol outlined below. The UART_RX module is always listening, so it correctly outputs 8 bits of data at a time to the parser module. The parser buffers 1 byte and waits until it has received a full message (2 bytes) before processing the message. *n* and *m* are extracted from the first two messages. Each line is fetched one at a time, and each option is consolidated into a buffer that is written into the central FIFO. The number of options for each line is stored in a 2D-packed array that is indexed into by the corresponding line index. Once the board has been completely processed, the parser signals the solver to start.

B. Protocol

Each message is 16 bits, so it takes 2 UART transmissions to completely send over one message.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
flag			index												val

There are six message types, and the **control** bits (bits 13 - 15) are a flag for which type of message is being sent/received.

- 111 = start new board
- 000 = end board
- 110 = start new line
- 001 = end line
- 101 = option assignment
- 010 = next option

If the **control** is 111, or start new board, the **index** bits (bits 1 - 12) are set to either *m* or *n* to communicate the size of the board (*m*x*n*) to the FPGA. If the **control** is 101, or option assignment, the **index** and **value** (bit 0) bits are set to the correct assignment for the current option. **value** is 1 if the current line's option has a 1 at **index** and vice versa. For all other flags, the **index** and **value** bits are set to 0 as there is no additional information that needs to be conveyed.

The flow that a board is sent to the FPGA is as follows:

1) start new board with *m* in the index value
2) start new board with *n* in the index value
3) start new line
4) option assignment(s)
5) next option
6) repeat 4 and 5 until line is finished
7) end line
8) repeat 3 - 7 until board is finished
9) end board

C. FPGA to PC

Once the solver module is done, the board's solution is ready to be transmitted to the PC using the assembler and UART_TX modules. The assembler will take the solution array from the parser and form the messages that are sent to the transmitter as fast as possible. The first two messages are start messages with *m* and *n* in the index values. Then *m*x*n* messages are sent with the cell by cell index and correct value. The last message is a stop message. The transmitter will send the board back to the PC utilizing a subset of the message types (start new board, end board, and option assignment). The result is then processed by PySerial and printed to the terminal for display on the PC.

V. Solver (Dana and Nina)

There are a few components to the solver- the FIFO queue, an array which holds the number of options per line, and the solver module itself. The FIFO queue is constructed in a way so that it stores a line index in a slot, and in the following slots it has the valid options for this line, until the next slot with a new line index. The number of options per line array is constructed so that it "maps" a line index to the number of options left for that given line index. This allows the solver to know when it has found the final assignment for a line because there is only one valid option left.

The solver module has 2 main components, two 1D arrays in the size of the maximal board, which it uses to solve the board and reduce the FIFO. Known- the first array, which keeps track of which cells have a known assignment - "0" if a cell is not known, and "1" if a cell is known. Assigned is the second array which keeps track of the cells' assignments- a cell will be assigned to "1" if it is known and its assignment is "1", and "0" if it is either not known or if it is known and its assignment is indeed "0".

The solver then gets line by line from the FIFO, and assigns the board or reduces options for a given line based on known information from within our Known 2D array mentioned above. There are two ways to assign a piece of information. First, is if a given cell has the same value for all possible ways to fill a row or column, for example Line 0 (Figure X) has two options- 011 and 110, therefore when we get to the next line index we know the middle cell in Line 0 can be assigned to "1". We will refer to this method as "Consistent Value". The second method in which we assign is if there is only one valid option to fill a row or column, for example we can consider Line 2 (Figure X).

To reduce the number of options in the FIFO we check for each option if it contradicts our known information. Contradiction check is done using a formula that use XOR logic on a possible option for a line with the assigned information at these spots. Discrepancies appear in the result as a 1. We then compare this to see if a discrepancy occurs at an index of known vs unknown information. This is done using an AND logic. If the final outcome is a 1, then we know that this option is not compatible with our assigned and known information so we do not add it back to the queue.

We also simplify by Consistent Value checking if a given index is always 0 or always 1 over all options for that line index. This is done by using AND logic on all the the valid options that we got from the FIFO for a given line with two arrays full of 1's. One array represent slots that are consistently "1" ($always_1$) and the other represent slots that are consistently "0" ($always_0$). For $always_0$ we AND it with $options$. Any remaining ones at the end mean that these indices of the board must be assigned 1 or 0 respectively.

Over time, the FIFO will get smaller, and we assign more values to our known and assigned board. We know we solved the board when all slots in Known array are equal to '1'. This means that every line is fully known. For every column to be known, every row must also be known. This means that everything is known and we have successfully solved the nonogram.

We've iterated through multiple possible solvers to get to this one. Originally there was just a BRAM but we encountered issues with wanting it to behave as FIFO queue, and access multiple lines at once. In addition, we originally had a queue where all of the options for a given line were stored together. However, the maximum size of a queue line was 1023 bits which limited us to 11 by 11 board and rapidly consumed all of the available memory within the FPGA. We also used to have a different completed check involving every bit within known to being 1. Unfortunately, this doesn't work for boards that are under our max capacity as the remaining bits will be 0 (in our final design Known and Assigned arrays are set to be the max possible size).

To test bench the solver, we used a variety of different boards with different properties. We began with a smaller 3 by 3 board that tested many basic functionalities. It tested what happened if multiple rounds were needed to solve the board or if there was only 1 option for a given row/column, etc. Once we achieved this, we expanded into a pretty complex 4 by 4 board which required more rounds to solve the board. We debugged the solver based on this testbench and then expanded into a full 11 by 11 board. This tested the full capacity of our solver and demonstrated that our solver could solve big nonograms to create images. This made testing a bit easier as we could simply confirm if the image produced was what we were expecting, in this case the image we focused on was a large x. Some challenges we encountered when testbenching were ensuring that our nonogram didn't get too complex too quickly. For instance, at one point we considered writing the number 15 out but considering this led to over 70 options for some lines, therefore, we decided against it and left larger boards up to our overall integration tests where we wouldn't have to generate each option manually.

A. *Parallelizing*

To parallelize the code, we created a new top level, 2 FIFOs, a parallel solver which uses both FIFOs, and explored evaluating 2 options at a time. We placed all rows and their options into one FIFO and all columns and their options into another FIFO and took an option off each FIFO on each clock cycle. This could accelerate the solving process. This proved to be challenging as we needed to make sure we don't ask the FPGA to access Assigned or Known arrays at the same time. We think we fixed that bug and related bugs by separating the parallel solver into 2 parallel cases, while making sure the columns part does not write at the same time the rows part does. Our test benches worked as expected, however, unfortunately the parallel solver did not produce the right board assignments. For our final submission, we have a parallel solver that uses two FIFO's that passes multiple testbenches that we have written for it.

B. *Invalid board*

Initially we built our solver with the intention of only handling solvable board inputs. However, we added a parameter to our solver module that returns if the board is not solvable. There are two ways to know if a board is not solvable. First is if there aren't enough options to actually solve the board. An example would be if there are 0 options that would satisfy another row/column. Second is if you try to try to make an assignment to a cell that has already been assigned to a different value. An example would be if you try to assign the value '1' to a cell that we know from other constraints must be a '0'. If a board is found to be unsolvable, we signal this in our solver module output but do not otherwise change anything.

C. Walk Through Solving

To clarify how exactly the solver works we will walk through the solver step by step for an example of 3x3 board (fig 6). The FIFO is constructed as displayed in Figure 7. When solver gets line index 0 it access "number of line options" array and sees Line 0 has 2 options. Then it get each option and uses logic to construct $always1$ and $always0$ according to the detailed explanation in the main sover section. When solver get line index 1 from FIFO it checks which slots in Line 0 are consistent- and it deduces that the middle cell is '1'. Therefore it assigns it to '1' in known and assigned. For Line 2 Solver access number of options and sees that Line 2 has only 1 option, therefore on the next clock cycle it will assign all of line 2 to known and the assigned cells which correspond to the 2nd line will be assigned accordingly to the only option available. Therefore after the first round, our FIFO will hold Line 2 index but will not have any option- since it was assigned. Similarly for Line 3. etc. The solution will be transmitted from the FPGA to the computer when all of known is equal to '1'.

VI. Evaluation

A. Memory

We used one or two FIFOs for the serial and parallel solver respectively. This was particularly inefficient because the FIFOs had to have a depth large enough to support the largest number of options each m or n could possibly support and then have a width equal to the largest dimension. This is not sustainable for really large puzzles upwards of 11x11. Using n choose k, each line of a 16x16 board has 720 possible options.

The solution being stored in an mxn array was also not very good. Same as with the FIFO, the larger the board, the bigger the data structure. As this entire array is passed between modules, the size of the solution affects the timing as it becomes harder to route the data the bigger it is.

B. Latency & Throughput

Each major part of this design has a significant amount of latency. For the receiver and transmission sections, there is little that can be done about it since we are using the UART protocol which is known to be slow. The one thing that could have been done was to increase the BAUD rate on both parts of communication, but the FPGA has a maximum baud rate of 12Mbaud which is equivalent to 40MB/s.

The solver currently goes through 4 stages in a FSM each time it processes a single option from the FIFO. This probably could be pipelined to increase the throughput of the solver resulting in an overall quicker solver.

C. Goals

We met all our goals for a minimum viable product as we have a working solver that can read puzzles in a DNF form from the computer and send the solution back to the computer to be displayed. It took us quite a bit longer than expected to get the minimum viable product, so none of the stretch goals have fully been integrated into the system. Unfortunately, the stretch goals we originally came up with no longer fit well with the design direction we went in, so we started working on different stretch goals such as supporting bigger boards and parallel solving.

For the bigger boards, the steps we made to making this possible was parameterizing all of the modules. The second part to this issue was thinking of ways to store the puzzle in a smaller amount of memory since that was a concern as stated above. The parallel solver required at least 2 FIFOs, one for the rows and one for the columns. The rows and columns can be solved independently, so two parts of the solver work on each by reading and writing from their respective FIFOs. Both bigger boards and parallelization worked in testbenches.

VII. Implementation Insights

A. Timing

All modules were initially running at 100Mhz, but the addition of the solver made it impossible. Pipelining was tried between the modules and inside the solver, but we could not make the slack nonnegative although we got pretty close at -0.047ns. Currently everything works and slack is not violated when the clock runs at 50Mhz. Maybe if the solver was better pipelined across the FSM, a 100Mhz clock could have been achieved.

B. Metastability in UART transmit

The major roadblock in achieving our minimum viable product was getting the uart_rx working consistently enough to be confident that the board was received by the FPGA worked correctly. The receiver would work 3 or 4 times before receiving the wrong data causing the parser to quite early. We tried different UART_Serial_USB adapters to see if that was the issue, used different computers, etc., but to no avail.

Eventually, we discovered that the issue was a metastability issue between the FPGA's clock domain and the signal being received by the computer. It was causing enough of a clock skew that the receiver was reading in the stop bit way too early and appending 1s to the end of a byte every few messages. This issue arose because of how much data we were reading using the UART protocol at a single time. The solution was to store the data inside of registers and then do the processing on the data in the registers.

VIII. Contributions

We collaborated well as a group and did a large portion of this project together through pair programming techniques and lots of collaborative meetups.

Figures 1, 8, and the byte fields were done by Veronica. Figures 2, 3, 4, 5, 6, 7 were done by Dana.

Veronica came up with the idea and helped with initial research on ways to implement this in hardware. She wrote all of the communication modules with their testbenches (uart_tx, uart_rx, assembler, parser) and the python communication methods in nonogram.py and serial_comm.py. She also wrote top_level, generated the FIFO IP, and refactored the solver to use a FSM and pipelining. Finally, she debugged the entire project and wrote the storage, commmunications, evaluation, and implementation sections.

Dana contributed doing research on how previous nonogram projects were constructed, what are the best known methods to approach this problem, discussing with the group on how to best implement the project, coded DNF board generator, Solver, Parallel Solver, parts of top level and parser, and whole top level for parallel solver, test benches for solver, parallel solver, and assembler. In the final report wrote abstract, introduction, and solver.

Nina began by brainstorming ways to solve nonograms using a SAT solver ultimately decided on our current design instead with Professor Steinmeyer. Nina coded the solver, the parallel solver, lots of testbenches, some of the top levels. The DNF board generator, the manual version of inputting a board, a bit of integration as well as code a lot of maintenence/cleaning. Nina started started the final report by modifying our preliminary report with all the new information that has changed since then. After that, she focused on the solver section discussing the ways we have testbenched the code, our parallelization strategies, invalid boards and general project overviews.

IX. GitHub Code

check out our code base: https://github.com/2nina2/Nonogram-Solver

References

[1] de Harder, Hennie. Solving Nonograms with 120 Lines of Code. Towards Data Science, 1AD, https://towardsdatascience.com/solving-nonograms-with-120-lines-of-code-a7c6e0f627e4.

[2] Daniel Berend, Dolev Pomeranz, Ronen Rabani, Ben Raziel, Nonograms: Combinatorial questions and algorithms, Discrete Applied Mathematics, https://www.sciencedirect.com/science/article/pii/S0166218X14000080

[3] K.J. Batenburg, W.A. Kosters, Solving Nonograms by combining relaxations, Pattern Recognition, https://www.sciencedirect.com/science/article/abs/pii/S0031320308005153

[4] DNF $https://en.wikipedia.org/wiki/Disjunctive_normal_form$

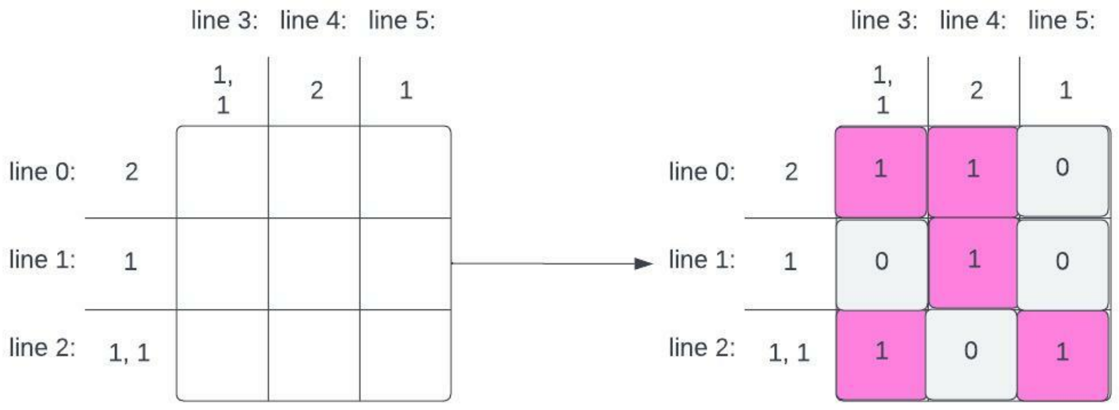


Fig. 5. 3x3 board on the left and its solution on the right

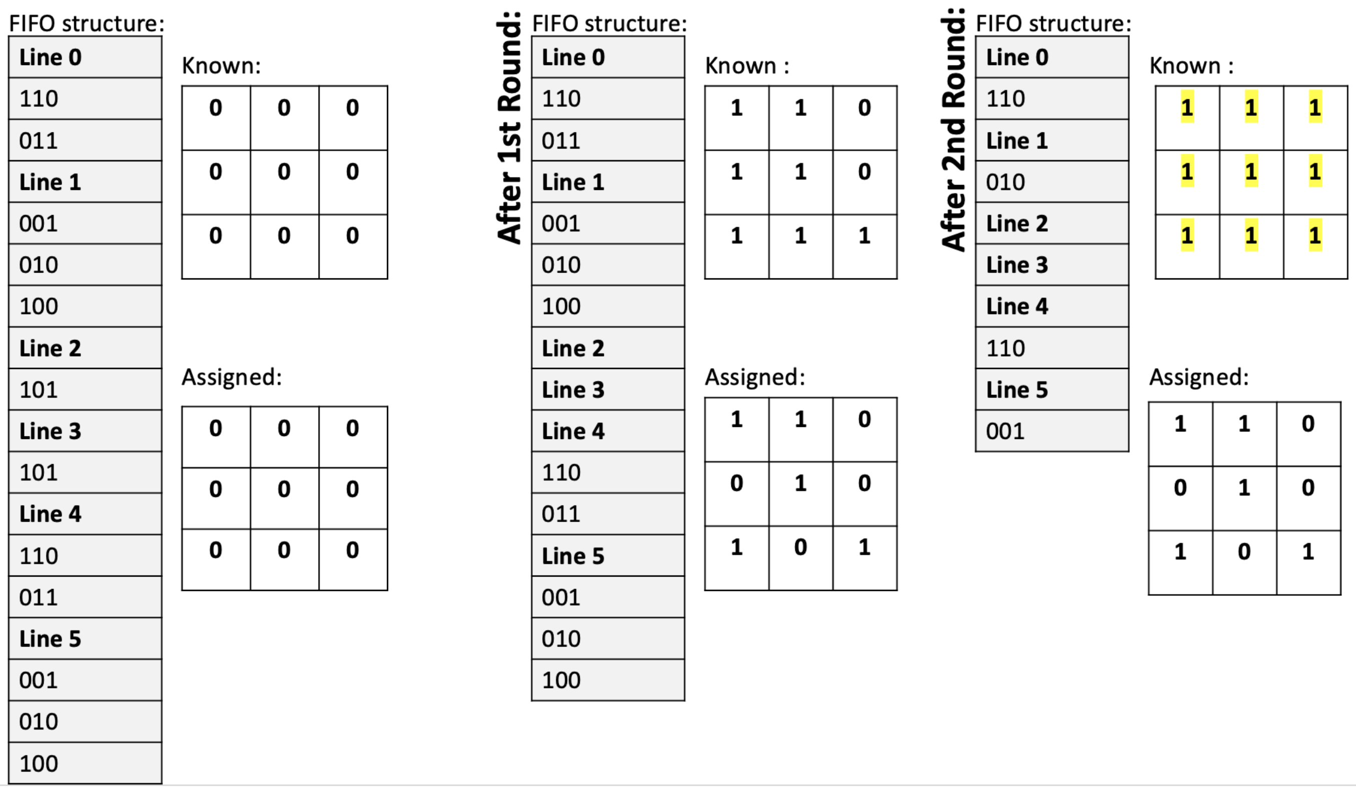


Fig. 6. 3x3 board solver walkthrough example.

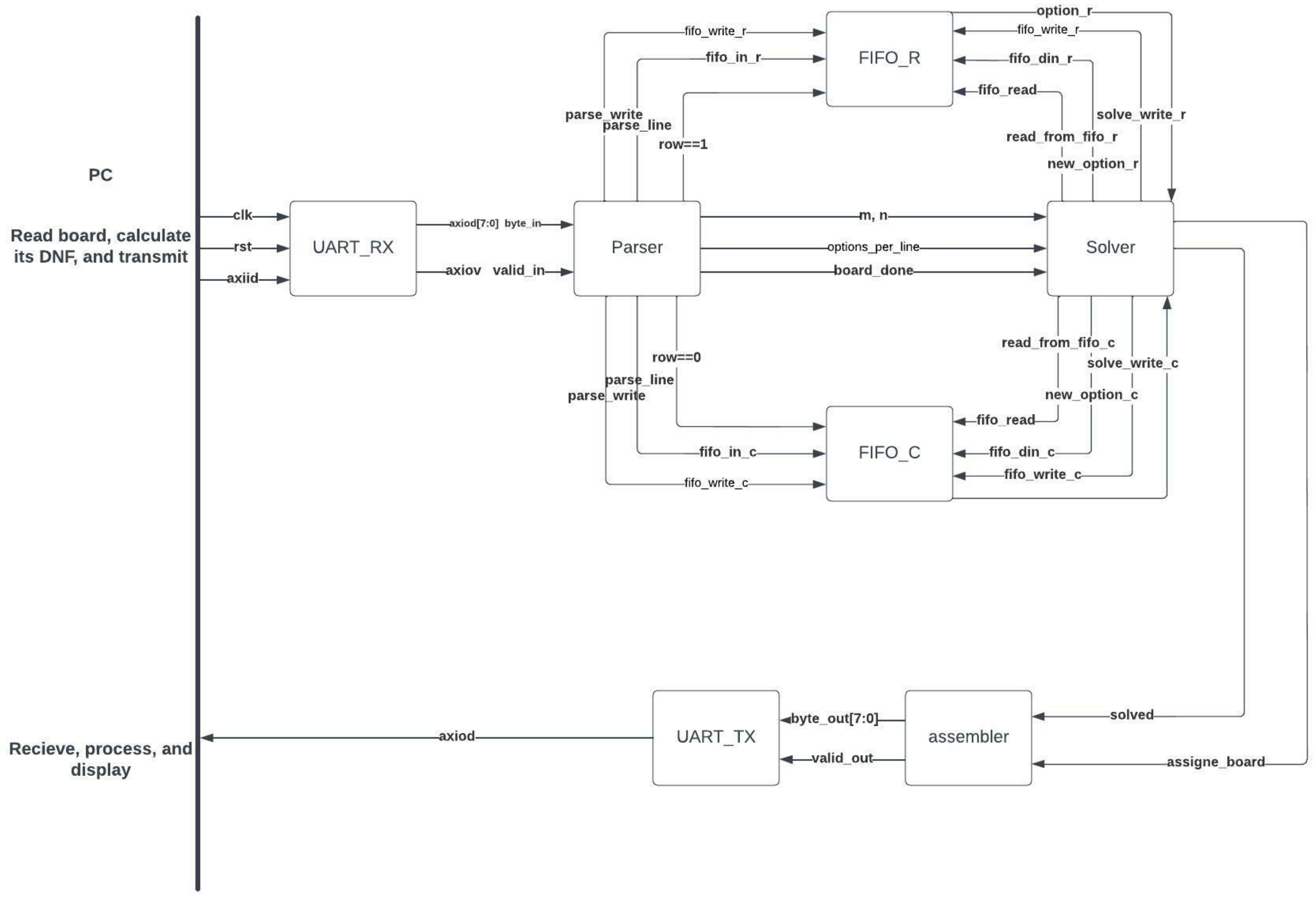


Fig. 7. Block Diagram for Parallel Solver

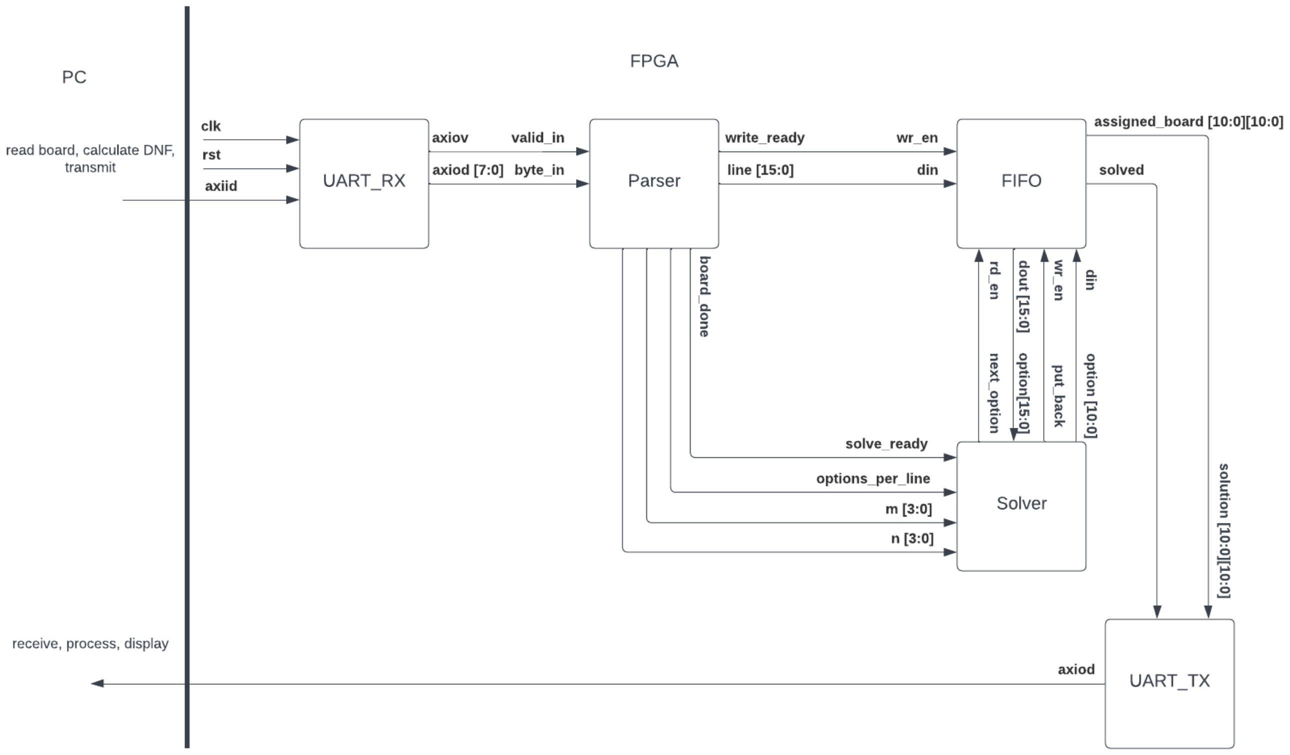


Fig. 8. Block Diagram