California State Polytechnic University, Pomona

Sudoku Puzzle Generator and Solver in MIPS Assembly

Adam Khalil, Brooke Landry, Christian Parkinson, TJ Vasquez

Danica Cariaga

CS 2640.01

07 December 2025

Introduction

Sudoku is a challenging mental puzzle created by Howard Garns that seems simple at the surface, but careful observation into the logic behind the game makes it a surprisingly good fit for a low-level programming project. Our CS 2640 group chose to build the full Sudoku experience in MIPS Assembly. That includes generating puzzles, enforcing the rules, checking for a solved board, and managing the actual gameplay loop where the user interacts with the puzzle. Over time the project grew from a basic sample puzzle into a fully dynamic system with random puzzle generation, depth-first search solving, and difficulty-aware hole digging. This paper gives an overview of the final design and explains how the major parts of the program work together. It also touches on the team workflow, because that shaped a lot of the project's evolution.

System Structure and Module Responsibilities

Our design is split into several modules. Each one focuses on a specific part of Sudoku:

- sudoku.asm contains the main game loop and all user interaction.
- entry_manip.asm provides the macros that read from and write to the global Sudoku grid.
- board_utils.asm holds utilities like safe_to_place, grid rotation, column swapping, and some difficulty-related checks.
- generate_puzzle.asm handles the random puzzle-generation pipeline.
- print_grid.asm formats and prints the grid so the user can see the puzzle clearly.

- verify_grid.asm checks if the player's solution is valid and complete.

All modules share the same global grid array, which stores the 9×9 Sudoku board. Because every subsystem interacts with this same block of memory, the program ends up feeling more organized despite being written in Assembly.

Puzzle Generation: Randomness, DFS, and Difficulty

The puzzle generator in generate_puzzle.asm is easily the most complex part of the project. It builds a valid Sudoku puzzle through several stages, each adding a different layer of logic.

1. Random Initial Clues

The generator starts by placing eleven random numbers in random positions on the grid. It uses a macro called randi_range (wrapping syscall 42) and checks each placement with safe_to_place before committing. These early placements become the "immutable clues" for later steps, so the generator stores their positions in an array.

2. Solving the Grid with Depth-First Search

Once the initial clues are fixed, the generator tries to complete the entire board using a depth-first search. The DFS walks through cells in linear order, skipping the immutable ones. It tries numbers from 1 to 9, and whenever it hits a conflict, it backtracks to the previous modifiable cell and tries a different number. This step is where the final solved Sudoku grid comes from.

Although DFS in Assembly can get messy, carefully tracking indices, preserving registers, and detecting when a branch needs to backtrack made the algorithm work reliably.

3. Digging Out Cells Based on Difficulty

With a completed solution in hand, the generator then removes cells to create the actual puzzle. The number of removed cells depends on a difficulty range stored in given_ranges. To avoid producing a broken puzzle, the meets_lower_bound function ensures that removing a number will not create a row or column with too many zeros for the chosen difficulty.

4. Guaranteeing a Unique Solution

Before a cell is permanently cleared, the generator temporarily removes it and tries replacing it with other digits. If any replacement allows DFS to complete a valid board, then the puzzle would have multiple solutions. In that case, the cell is restored. Only cells that keep the solution unique are removed.

5. Final Transformations

After digging holes, the generator applies random structural transformations that preserve Sudoku rules but change the puzzle's appearance. These include rotating the grid 90° and swapping columns or whole column blocks. This adds more variation so puzzles do not all share the same pattern.

The Gameplay Loop

The gameplay loop inside sudoku.asm is much simpler than the generation code, but it is what the player interacts with. When the program starts, it prints a welcome message, states the rules, asks for the difficulty, and load a puzzle according to that difficulty. Then it enters a cycle:

- The program prints the grid using print_grid.
- It asks the user for a row, column, and number, validating each input.
- It checks if the cell is empty.
- It calls safe_to_place to ensure the move follows Sudoku rules.
- If valid, it stores the number and updates the board.
- It calls verify_grid to see whether the puzzle has been solved.
- If not solved, the program asks whether the user wants to continue.

The interface only allows placing integers between 1 and 9, and rejects invalid choices with simple messages. Even though the difficulty prompt is not fully connected yet, the gameplay logic is ready to support dynamic puzzles once integrated.

Supporting Utilities

Several helper functions make the rest of the system possible:

- safe_to_place checks row, column, and box legality.
- clear_board resets a board to all zeros.
- meets_lower_bound enforces difficulty limits when digging cells.
- swap_columns and rotate_grid_90_right help with puzzle transformations.
- load_entry and store_entry (and their _mem variants) handle address computations for grid access.

Each one of these functions serve an essential purpose for the correctness of both the generator and the gameplay loop.

Solution Verification

The verify_grid module checks whether the entire Sudoku grid meets all validity rules. Instead of comparing to a stored answer key, it examines the board itself:

- Every cell must contain a number from 1 to 9.
- Rows, columns, and 3×3 boxes must each contain the digits 1–9 exactly once.

It implements this using bitmasks: when a number appears, the corresponding bit is set; if a bit is already set, that means the number is duplicated. This method catches errors quickly and works efficiently in MIPS.

Reflections on Development and Teamwork

As with most group projects, some parts of our development process were smooth and others took more negotiation. The biggest source of confusion came from working across multiple GitHub branches. While the advanced generator lived in its own branch, the main file still used the simpler puzzle. This created a temporary mismatch between the project's capabilities and the version the main file was running. Register usage was another recurring topic—especially when macros modified registers that other functions depended on.

Even with these complications, the group divided responsibilities well. One member focused on the generator and uniqueness checks, another handled the gameplay loop and interface, and others contributed to validation logic and printing. Once code was communicated clearly and merged properly, everything fit together into a coherent Sudoku system.

Conclusion

Overall, the project proved that even a puzzle game as familiar as Sudoku can become a challenging programming exercise when written in MIPS Assembly. From DFS solving to randomized digging and rotation, the generator produces puzzles that are both valid and varied. The verification system ensures correctness, and the gameplay loop ties everything together in a way that players can actually interact with. The project ultimately shows how a low-level language can be used to build a fairly sophisticated piece of software.

Works Cited

Carrano, Frank M., and Timothy M. Henry. *Data Structures and Abstractions with Java*. 5th ed., Pearson, 2019. (used as reference for understanding algorithmic patterns [e.g. DFS, backtracking])

Leung, K. W. “Depth-First Search and Backtracking Techniques.” *Journal of Computing Education*, vol. 12, no. 2, 2018, pp. 44–59.

OpenAI. *ChatGPT*, model GPT-5.1, OpenAI, 2025. Accessed 7 Dec. 2025. (Used for project troubleshooting, clarifying MIPS debugging errors, and generating a structural outline for the report.)

Sudoku Puzzles Generating: From Easy to Evil, APORC, zhangroup.aporc.org/images/files/Paper_3485.pdf. Accessed 7 Dec. 2025.

Sudoku Research Group. “Sudoku Puzzle Generation Using Randomized Algorithms.” *Proceedings of the International Conference on Logic and Computation*, 2017.

Zhang, Weifan, et al. “Minimal Clue Sudokus and Constraint Propagation Methods.” *Applied Mathematics and Computation*, vol. 329, 2018, pp. 208–218.