Andrew Latham

Gregory Fowler

Caley Shem-Crumrine

Pat Melvin

**Sudoku Generator and Solver**

The goal of our project was to developed a program in MIPS Assembly language that would be able to generate valid Sudoku puzzles and to solve puzzles generated either by the program or by the user. We chose this project because we wanted to contribute something new to the world instead of creating a program that had been done before. Looking on the internet, we discovered that plenty of Sudoku generators and solvers had been written in high-level languages, but none existed in Assembly. We felt like it would be interesting and challenging to tackle such a complex problem in such a low-level language.

Our team encountered a number of challenges during the creation and design of this program. For example, we were not able to take advantage of an existing backtracking algorithm, as there were no general-purpose algorithms in Assembly that we found that we were able to take advantage of. Also, because we were required to use QtSpim rather than MARS or any other program, we were unable to use the system 42 syscall to generate a random number (Spim does not recognize any syscall above 12); therefore, we had to write our own. Additionally, the MIPS architecture took some getting used to. Output was a challenge, because nothing can be deleted once it is written to the screen; furthermore, basic things that would be handled automatically in higher-level languages, such as returning a certain value, are not as automatic in Assembly.

A number of issues cropped up during the debugging process as well. One such issue was that of addressing errors, where the program would produce an invalid address error. Another was that our difficulty selector did not work at first, printing out the same puzzle every time regardless of what difficulty was entered. Throughout our process, we had a lot of difficulties due to the limited number of registers available.

For this project, we wrote our own backtracking algorithm. For the sake of simplicity, some basic terminology will be defined in order to aid the reader in understanding the functionality of this algorithm. A "value conflict" refers to a given square not being able to hold a particular value, likely because that value is elsewhere in the row, column, or square. A "square conflict" refers to a given square not being able to hold any value at all. An unsolvable puzzle is referred to as a "puzzle conflict”. The algorithm starts at the first square, then recursively traverses every row of the first column. It then continues in the next row, and so on and so forth, until it either reaches the end of the puzzle (and there is no conflict), or until it finds a square conflict. It then leaves that square blank and goes to the previous square, trying the next value that works. The program checks each square for value conflicts; if all numbers 1-9 result in value conflicts, that is how the program knows that a square conflict has arisen. If a square conflict occurs in the first square of the puzzle, then that must mean that the puzzle is unsolvable, resulting in a puzzle conflict.

For the issue of the random number generator, we looked to the second volume of Donald Knuth’s The Art of Computer Programming series, which was on seminumerical algorithms. This book had a full chapter on random number generators, and we found it to be a useful resource while we were developing our program. Based on advice taken from this book, we decided to use the Linear Congruential Method with an *m* value of 90000. This ensures a sufficiently large sequence such that even if the choice of a was horrible, the period would still be far longer than we would ever possibly use (measures to choose an appropriate a value, however were still taken). Furthermore, this ensures that we can get a number between 0 and 8 by simply dividing the resulting value by 10000. One issue we encountered, however, was that of getting different initial input values – the algorithm does not generate truly “random” numbers, only a sequence of seemingly random numbers based on input. We ultimately decided to request two values from the user that we would then add together to form the initial value. In order to ensure that there was not a skew towards small numbers, the user is asked to input a number between bounds that are set to ensure a reasonably large number.

In order to generate a puzzle, we decided to use the most simple and efficient method available – starting with a solved puzzle and then using the random number generator a large number of times in order to thoroughly mix it up. We first switch rows and columns around within boxes a random but large number of times, since those operations can be performed while maintaining the correctness of the puzzle. Then, based on the difficulty level, a random number of values is removed from the puzzle in order to make it an actual Sudoku puzzle. The higher the difficulty level, the more values are removed from the puzzle. This generation method produces a rich variety of solvable Sudoku puzzles in a reasonably quick amount of time.

In order to keep users from becoming impatient or suspecting that they were encountering an error in either the code or their computer when the puzzle is being generated or solved, we instructed the program to output a dot every now and then. This was done in the linest that would be encountered rarely in the code, so that performance was maintained and the output did not get too cluttered.

As stated, things that would normally be handled by the compiler in a higher-level language are not handled in Assembly. Therefore, we had to address these issues ourselves. This issue appeared in the most difficult context in the solver - the backtracking method would return 1 on success and 0 on failure. Normally, the GCC compiler would return a 0 automatically, but in Assembly, that does not work. A zero has to be manually returned in the event of an unsolvable puzzle, lest the program run forever. Indeed, this issue was first discovered when the program did indeed run forever because a value of 0 was not returned.

We had a memory address issue that took a while to solve and turned out to be an issue of division: in order to get the memory address of cells, numbers were added, then multiplied by 4 to get the address. The issue was, these numbers were not then divided by 4 before being used again – they were simply being incremented, which led to memory exceptions that were located in strange parts of the code that had little to do with where the problem was originally generated. Once this fact was realized and addressed, the problem was solved.

An issue arose in determining how to represent a blank space in the puzzle. As it stood, we had the program write a zero wherever a blank was supposed to go; however, we did not want to just leave a zero, as at a glance zeroes can look like an already filled-in number (namely, 8). Therefore, a method was added to insert a dash wherever a zero appeared in the output.

Other issues arose when working with the MIPS architecture. Among other things, users cannot erase what they have already written. As such, we have decided to run the program through the terminal in order to work around this limitation.

Furthermore, the console cannot be cleared or modified, save for adding things to it. It was originally hoped that we would be able to modify the puzzle in real time. Much time and effort was spent looking for an algorithm or some code that would make this possible. However, everything we found either failed to work effectively or slowed the code down in a major way. Therefore, attempts to remedy this problem were abandoned.

One particularly irritating issue was that QtSpim will not display monospaced font. We were confused as to how this could be possible as QtSpim is supposed to represent a low-level language; however, it insists on using modern fonts that are not monospaced. To get around this issue, we decided that it would be best for the program to be run in Terminal, which does use monospaced type.

An issue that we ran into repeatedly was that methods, in various places, referred to the wrong register. When the Random Numbers method was implemented, for example, the registers were changed to not use any s registers, in order to conform to the MIPS convention specified in the document provided on blackboard. The later methods that used this method, however, did not always use that, and continued to reference the old s registers. It took some debugging to find every instance of the wrong registers being referred to. Indeed, this was what caused our difficulty to not work properly − the method referred to register t3 rather than the correct register of s7, causing the same puzzle to be printed every time. Another issue that we had trouble with the convention for $a0. MIPS convention states that it should be used as a parameter passed to functions; however, it is also the parameter used for syscalls, which meant that we had to work around this fact every time we wanted to do a syscall. Finally, an issue that took a little while to solve was that jal overwrote $ra and then bounced to a completely different part of the code. When the return address was used again in the code after the jal statement, it would have an incorrect value if the value had not been saved in the stack before the jal, which led to a lot of problems that were difficult to solve.

A technique that we found particularly useful in the development of our algorithm was writing the code in a higher-level language (namely, C and Java) first and then translating it to Assembly by hand. This way, if there were any issues with the algorithms that we had written, we could discover them in the higher-level language first, where the errors were a lot easier to detect, identify, and solve. In Assembly, it was often difficult to tell what was causing an error, or that there was an error at all. We also made extensive use of pencil-and-paper outlines and sketches in order to plan how the MIPS code would work, because writing it was very tedious and we wanted to ensure that we knew exactly what we were doing as it was easy to get lost in the code once it was written. In order to ensure ease of reuse and debugging, our code was documented quite thoroughly, with the purposes of most registers specified in the headers to methods and any function whose purpose was difficult to understand explained in detail.

We enjoyed the opportunity to write in a low-level language like Assembly, because it helped us to gain an appreciation for how our code looks after it is compiled. The lessons learned from this project will help us all to become better high-level programmers as we will know how to think in terms of memory management and avoiding expensive operations.