Final Project Report

Group 3: Dennis Zang (704766877), Daniel Park (104809832),

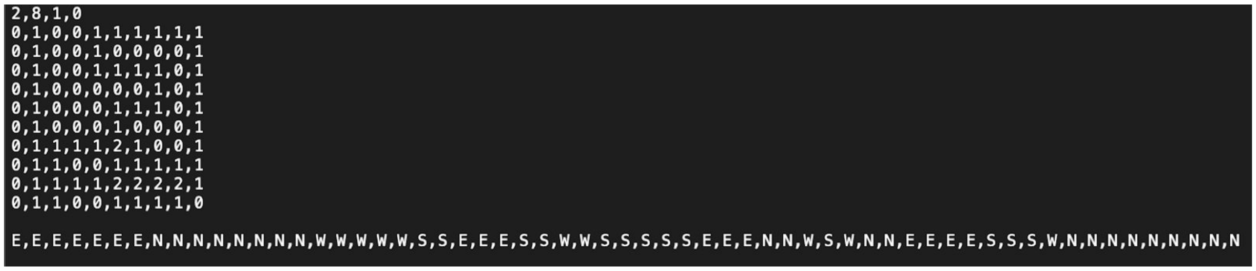
Samuel Pando (904809319)

December 10, 2020

Puzzle Solver Module

[Components and Implementation Logic]

For this final project, our goal was to create a module that, given a solution to a specific puzzle, would be able to test the solution and determine whether or not it is valid. The goal of the puzzle is to reach a grid with only 0s on it. However the player must start and end on pre-set grid locations and can only decrement a grid's tile value by passing over it. To this end, we used one module named *puzzle.v*, one testbench named *puzzle_tb.v*, and two C++ programs *format_in.cpp* and *format_lvl.cpp* to convert the solution and puzzle into an acceptable format for the previously mentioned modules.



```
2,8,1,0
0,1,0,0,1,1,1,1,1,1
0,1,0,0,1,0,0,0,0,1
0,1,0,0,1,1,1,1,0,1
0,1,0,0,0,0,0,1,0,1
0,1,0,0,0,1,1,1,0,1
0,1,0,0,0,1,0,0,0,1
0,1,1,1,1,2,1,0,0,1
0,1,1,0,0,1,1,1,1,1
0,1,1,1,1,2,2,2,2,1
0,1,1,0,0,1,1,1,1,0

E,E,E,E,E,E,E,N,N,N,N,N,N,N,N,W,W,W,W,W,S,S,E,E,E,S,S,W,W,S,S,S,S,S,E,E,E,N,N,W,S,W,N,N,E,E,E,E,S,S,S,W,N,N,N,N,N,N,N,N,N
```

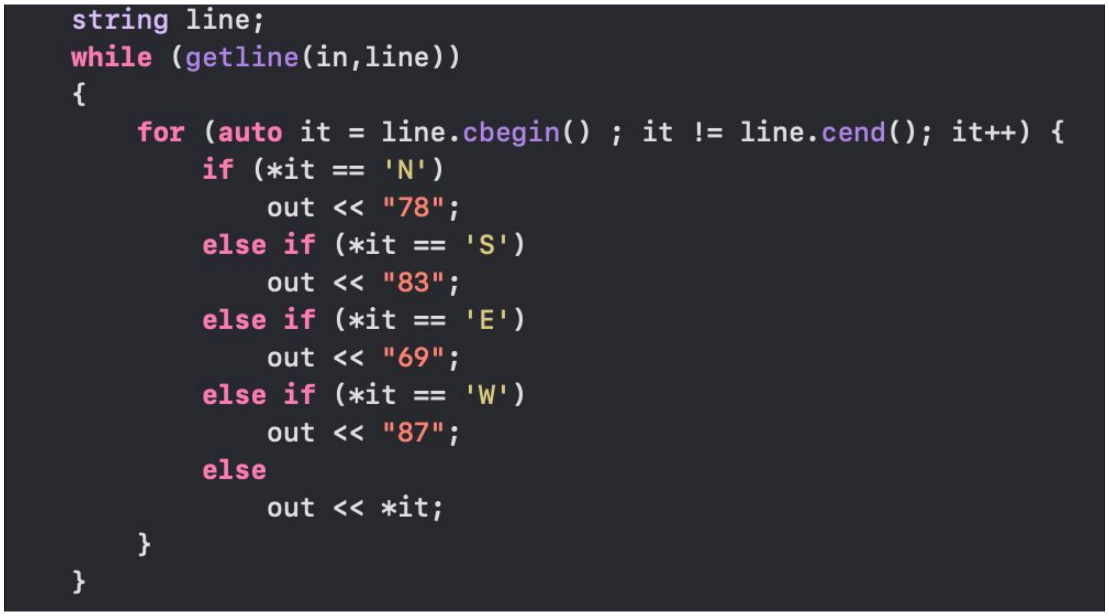
Figure 1.1: An example puzzle (above) and its solution (below). Both are unformatted. The first two pairs of decimals at the top of the puzzle are the players start coordinates and the goal coordinates (respectively).

[puzzle.v]

Puzzle.v operates in two different modes controlled by an "enable" and "enable_process" input. The first mode loads in the puzzle's design while the second tests the player's solution against the loaded puzzle. Using an internal register to save the puzzle's state, the player's solution is read as input to modify the level. If the level is filled with only 0s after the solution has been read, then puzzle.v outputs 1.

[c++ code]

For this project, since the puzzle and input were a bit difficult to deal with as input for the verilog module, we modified them using C++ code. From what we've seen so far, verilog is decent at parsing input with very easily recognizable patterns (this can be handled with $fscanf and a pattern specifier). However, for our solution input, we decided to use the characters ‘N’, ‘S’, ‘E’, and ‘W’ to denote the four cardinal directions in the puzzle, and this caused a few problems. It would be a lot easier to input them as numbers, but then this would be less convenient for our users. Similarly, the puzzle was formatted for readability and not verilog functionality (the constant new lines disrupt the pattern of decimals followed by commas). So instead of finding some workaround to get the module to read this, we just used C++ code to easily convert it before inputting.



```cpp
string line;
while (getline(in,line))
{
    for (auto it = line.cbegin() ; it != line.cend(); it++) {
        if (*it == 'N')
            out << "78";
        else if (*it == 'S')
            out << "83";
        else if (*it == 'E')
            out << "69";
        else if (*it == 'W')
            out << "87";
        else
            out << *it;
    }
}
```

Figure 1.2: Snippet from the *format_in.cpp* file. As we can see, we parse the original input file and convert any instance of ‘N’, ‘S’, ‘E’ and ‘W’ to their easier to use ASCII representation.

[Waveforms/Testbench]

Our testbench consists of puzzle/solution pairs. We load in the puzzle and one of its possible solutions to test for the proper result. Below are two example waveforms. The first was given a puzzle and its solution. The second was given a non-solution that did not solve the puzzle fully.

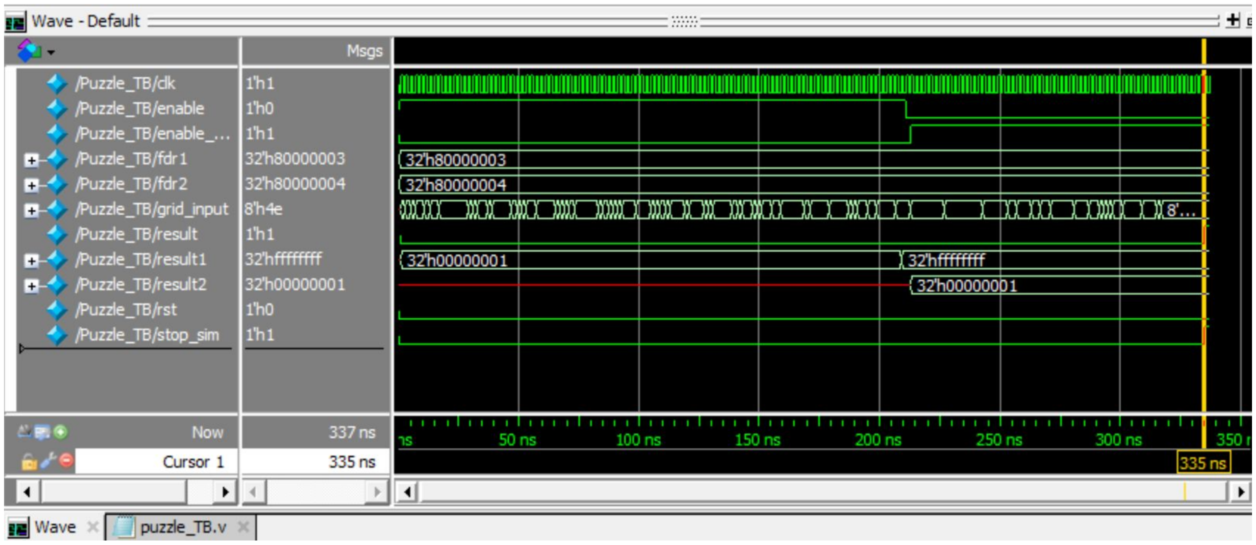


Figure 2.1 The result of a puzzle/solution pair. Notice the value result (not to be confused with result1/result2) is set to 1 at the end of the simulation when the solution has been confirmed.

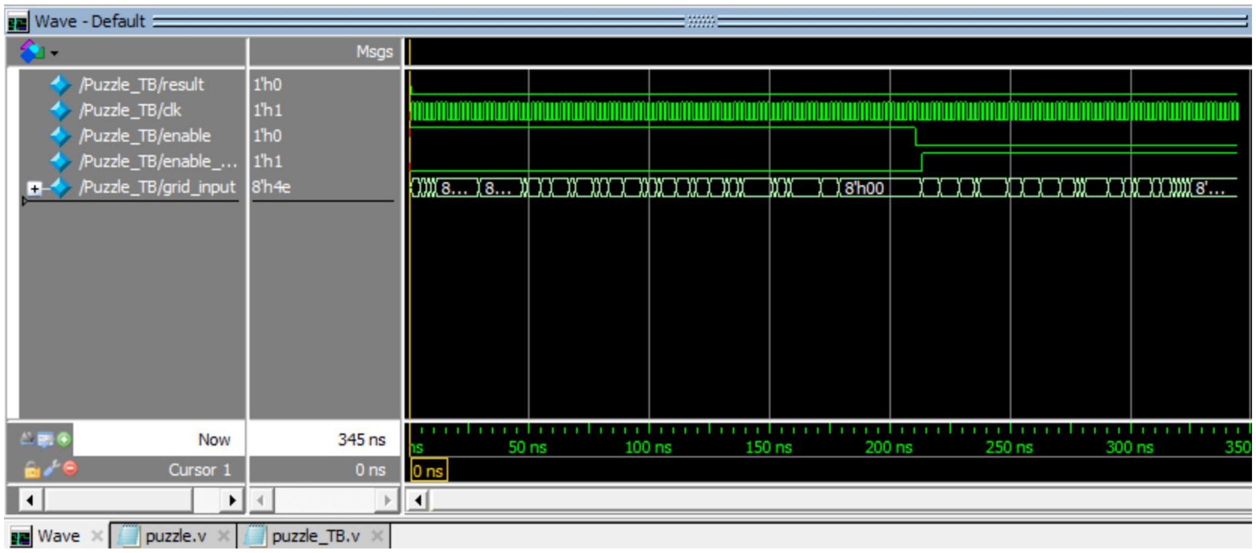


Figure 2.2 The result of a puzzle/non-solution pair. Notice the value result is never set to 1.

[Challenges Faced]

There were some difficulties manipulating the grid properly and converting between the load and solve modes. The main problem was that x values (a Verilog syntax convention) were passed to our puzzle solver and had to be filtered out. Another difficulty was keeping track of different grid variables. The two modes shared similar variables that could not be confused with each other. We had to pay attention not to use the incorrect variables depending on the mode that we were currently in.

Member Contributions:

Daniel Park: puzzle.v and puzzle_TB.v

Dennis Zang: Project Report and State Representation

Samuel Pando: C++ code and IO