Skyscrapers Puzzle with Altera FPGA DE1

Students: Andrea Giardini - Francesco Venturoli

Aim of the project

The project aim is to write a software to run the Skyscrapers puzzle on a Altera DE1 board [1], allowing users to interact with it using a keyboard. The final result will be an interactive logic game, where the user can input digits in the game or solve it automatically. The game needs to recognize when the puzzle is completed and if the solution provided is correct and satisfies all the constraints.

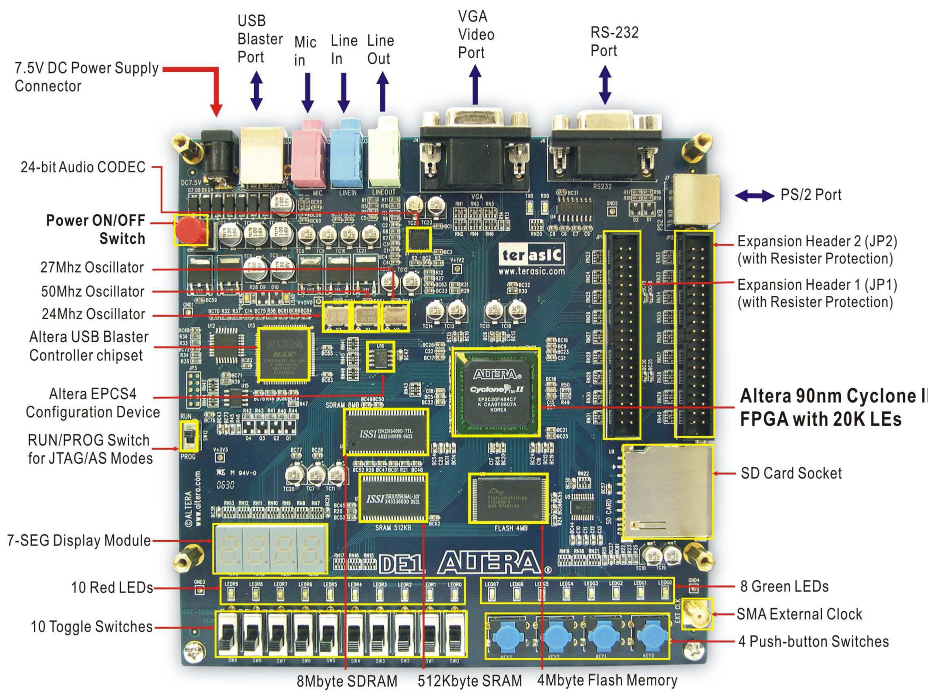

Figure 1: Altera DE1 FPGA Board

In the proposed game, the user needs to be able to move the cursor in the game using the arrows on the keyboard. Entering numbers in the matrix, always using the keyboard, the user tries to complete the puzzle filling all the blank spaces with a numeric value. If the user is not able to complete the game, the puzzle can automatically be solved by the algorithm implemented in the board. Once the puzzle is solved, the board should recognize the solution and show the user that he won.

Introduction to the puzzle

Each puzzle consists of an NxN grid, organized as a latin square, with some clues along its sides. The goal is to place a skyscraper in each square, with a height between 1 and N, so that no two skyscrapers in the same row or column have the same number of floors. In addition, the number of visible skyscrapers, as viewed from the direction of each clue, is equal to the value of the clue. Note that higher skyscrapers block the view of lower skyscrapers located behind them.

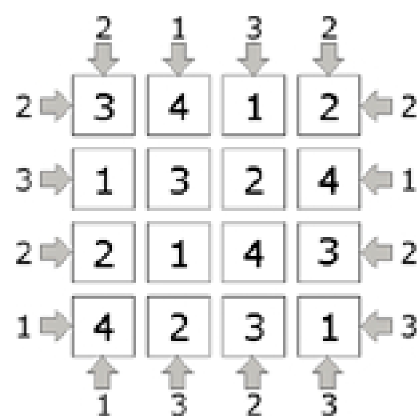

Figure 2: Solved 4x4 Skyscrapers puzzle

There are a number of intuitions that can be addressed immediately, filling some of the empty spaces with values, while other spaces need a combination of more constraints to identify the correct value. The game can be extended with a bigger matrix or to include parks (empty spaces, represented by buildings with zero height in the matrix). For this project, we decided to solve the classic version of the puzzle, a 4x4 matrix with no parks.

Control Unit

The control unit is used in this project to get the inputs from the user and interpret them. Following that, the interpreted signals are sent to the Datapath.

As we specified previously, our intention is to let the user use a keyboard to play the game. In order to do this we have to read the serial PS2 line and map the received data with the scan codes of the keys that we want to interpret.

In order to simplify this task and keep the Control Unit as clean as possible, we added a new class called *Skyscrapers_Puzzle_Keyboard*, which is responsible exclusively of reading the data from the keyboard and map them to scan codes. Using this class we can leave all the logic of translating the signals to actions to the Control Unit, as it is supposed to be.

The Control Unit uses the following signals:

```
CLOCK        : in std_logic;
keyboardData : in std_logic_vector (7 downto 0);
RESET_N      : in std_logic;
TIME_10MS    : in std_logic;

CURSOR_POS   : in CURSOR_POS_TYPE;

-- Connections with Data-Path
MOVE_RIGHT   : out std_logic;
MOVE_LEFT    : out std_logic;
MOVE_DOWN    : out std_logic;
MOVE_UP      : out std_logic;
NUMBER       : out std_logic_vector (3 downto 0);
SOLVE        : out std_logic;
CLEAN        : out std_logic;
```

The Control Unit reads the serial line *KEYBOARDDATA* and translates the scan codes received to actions, that are then sent to the Datapath. In order to avoid reading a single keypress as multiple keypresses (since the data clock is too fast) we are scanning the line periodically and not constantly.

Datapath

The Datapath takes care of the logic of the game, it maintains in memory the numbers that the user added to the matrix. The user is able to add new numbers to the matrix using the keyboard and all its changes are registered inside the Datapath. Moreover, the Datapath contains all the logic needed to solve the puzzle automatically.

```
CLOCK       : in std_logic;
RESET_N     : in std_logic;
MOVE_RIGHT  : in std_logic;
MOVE_LEFT   : in std_logic;
MOVE_DOWN   : in std_logic;
MOVE_UP     : in std_logic;
SOLVE       : in std_logic;
CLEAN       : in std_logic;

KEYS        : in std_logic_vector (3 downto 0);

MATRIX      : out MATRIX_TYPE;
CONSTRAINTS : out CONSTRAINTS_TYPE;
SOLUTIONS   : out SOLUTIONS_TYPE;
CURSOR_POS  : out CURSOR_POS_TYPE;
WINNER      : out std_logic
```

The Datapath is the main component of the project since it stores the matrix with the values, handles the inputs received from the Controller and contains the algorithm used to solve the puzzle.

Every cell in the schema is mapped as an array of N logic values (effectively mapping the game to a NxNxN matrix), each one representing the feasibility of a solution in a given cell. At the start of the game, all values are feasible in any cell.

When the user inputs a number or invokes the automatic solver, no values are set in the matrix: instead, other values are deemed infeasible, and thus removed from the list of feasible values. After this, if any cell is left with only one feasible solution, its value is assigned to it and displayed on screen.

View

The View is responsible for drawing on screen the elements of the Datapath and represent them in the best way for the user. In our case, the visual representation of the game is quite simple, but there were some challenges to be solved, especially with the drawing of numbers.

```
CLOCK        : in std_logic;
RESET_N      : in std_logic;

MATRIX       : in MATRIX_TYPE;
SOLUTIONS    : in SOLUTIONS_TYPE;
CONSTRAINTS  : in CONSTRAINTS_TYPE;
CURSOR_POS   : in CURSOR_POS_TYPE;

REDRAW       : in  std_logic;

FB_READY     : in  std_logic;
FB_CLEAR     : out std_logic;
FB_DRAW_RECT : out std_logic;
FB_DRAW_LINE : out std_logic;
FB_FILL_RECT : out std_logic;
FB_FLIP      : out std_logic;
FB_COLOR     : out color_type;
FB_X0        : out xy_coord_type;
FB_Y0        : out xy_coord_type;
FB_X1        : out xy_coord_type;
FB_Y1        : out xy_coord_type;
HEX0         : out std_logic_vector (6 downto 0);
HEX1         : out std_logic_vector (6 downto 0);
HEX2         : out std_logic_vector (6 downto 0);
HEX3         : out std_logic_vector (6 downto 0)
```

As we can see from the signals, we used the FrameBuffer technology to draw the frames of the game before printing them on the screen. The FrameBuffer is a portion of memory containing a Bitmap that is used to refresh a video display from a memory buffer containing a complete frame of data. Basically what happens is that we draw the whole frame to be displayed in memory before we display it on video. Once the frame is ready to be displayed and all the drawing operations are completed, we refresh the screen, drawing on it the new frame that we just created in memory.

The View is organized in different macro states:

```
type state_type is (IDLE, WAIT_FOR_READY, DRAWING);
```

In particular we will analyse the DRAWING state, which is made of a set of sub-states:

```
type substate_type is (CLEAR_SCENE, DRAW_BOARD_OUTLINE, DRAW_BOARD_BLOCKS,
    DRAW_BOARD_CONSTRAINTS, DRAW_BOARD_NUMBERS, FLIP_FRAMEBUFFER);
```

Every state draws a different object:

- CLEAR_SCENE

  Cleans the frame in memory, filling it with a black square which will be the background of our game.

- DRAW_BOARD_OUTLINE

  Draws the outline of our game board, just the perimetral square.

- DRAW_BOARD_BLOCKS

  Draws the blocks inside our game board and the position of the cursor.

- DRAW_BOARD_CONSTRAINTS

  Draws the constraints around our game board.

- DRAW_BOARD_NUMBERS

  Draws the numbers that have been inserted inside the board.

- FLIP_FRAMEBUFFER

  This state is reached when the drawing has been completed and we are ready to display it on the screen. After this, we go back to IDLE state, preparing for the creation of a new frame to display.

The most difficult part was drawing the numbers on screen, since the FrameBuffer library was not giving us any possibility of drawing custom images but only squares. To draw the numbers we had to define every digit as a sprite, which is an array of colors, representing pixel by pixel the color that has to be drawn, considering black as a background color and white as foreground. After that we had to draw a square of 1x1 pixel for each pixel of the number's sprite.

Solving algorithm

In order to solve the puzzle we had to implement in VHDL multiple methods to remove solutions from the board based on the constraints of the game. In the following section we will analyze the rules that we are applying to solve the problem and how they help us to remove infeasible solutions from the board.

Starting with the structure, we represented the board as a three-dimensional matrix. This matrix has dimensions 4 (length) x 4 (height) x 4 (possible solutions). We must consider that, when a number is inserted, that solution can be removed from the corresponding row and column, since we cannot have the same number twice on the same column or row.

During the analysis phase we determined that the most convenient technique to set the correct value for a cell is to remove from the set of possible solutions, all the values that are incorrect. This way, when a cell has only a possible value, we set that value.

Simple solvers

- If the constraint is **one**, the first element is **four**

  Since only a single skyscraper is visible from that side, it means that it has to be the tallest one. In this case the number *four* is added to the matrix in first position.

1	4			

- If the constraint is **two**, the second element cannot be **three**

  If the constraint is two and the second element is three it will not be possible to satisfy the constraint. In fact, if the second number is set to three, there are two possible outcomes, and we can see they are both invalid.

  In the first case, as shown in the table below, the tallest building is put before building three: therefore, we can only see one building and the constraint is invalid.

2	4	3		

In the second case, as shown in the table below, the tallest building is put after building three: therefore, we can see three buildings (the first one which is necessarily shorter than three, building number three and building number four) and the constraint is invalid.

2		3		4

- <u>If the constraint is **four**, the line contains all numbers in ascending order</u>

  Since the user is able to see all the skyscrapers, they can only be be put in ascending order.

4	1	2	3	4

- <u>Any constraint indicates the first valid position for number **four** in the line</u>

  Since the constraints tell us how many buildings we can see, the tallest building must be at least **constraint-1** positions away from the beginning of the line. For instance, if our constraint is **three**, the building number **four** must be at least in position **three**, to leave space for two more buildings to be seen before it.

  This rule leads us to a particular case, which can be used to guess the exact position of building **four**:

  - <u>If the sum of two opposite constraints is **five**, the number **four** is in the position specified by the leftmost constraint</u>

    It's easier to understand this rule with an example. As we can see, if we apply the previous rule to the schema in the table below, the leftmost constraint tells us that building number **four** can be put in the second, third or fourth space. If we apply the rule again, this time using the rightmost constraint, we can deduce that building number **four** can only be put in the third and fourth space from the right (which are the first and second space from the left). Combining these two deductions, we find that the only space which can contain number **four** is the second (as specified by the leftmost constraint, which is **two**).

2		4			3

    Of course, the same reasoning applies to all other cases in which the sum of the constraints is **five**. Here are some more examples:

3			4		2

1	4				4

4				4	1

Intuitive solver

The rules that we described previously can help us reduce the number of possible solutions in one cell. Unfortunately, just using those rules, our game is still not able to resolve some obvious cases, like for example:

3		2	4		2

3			4	1	2

In the examples, both empty cells have two possible solutions and, for this reason no number is assigned to them. We can also notice that, for each example, there are two possible solution and that only one of them is correct.

The intuitive rule tries to address this problem: when there are more than one possible combinations of results, the algorithm tries to verify if any of them is wrong. This rule allowed us to solve more complex schemas, where the intuition of the user would have been needed to complete the board.

The algorithm we designed works in two specific cases: when only one of the cells preceding building number four is empty and when there are only two empty spaces in the line and they are both before before building number four. Let's start by analyzing the first case, looking at the table below for an example.

3		2	4		2

As we can see, there is only one empty cell before the tallest building, which has two possible solution. In this case, our algorithm finds the empty cell and tries all possible values (in our case, /textbfone and /textbfthree). When trying value /textbfone the constraint is satisfied, so nothing happens. When trying value /textbfthree the constraint is not satisfied, so value /textbfthree is removed from the feasible values. This means that only value /textbfone is feasible now, so it is added into the cell. In turn, this means that value /textbfone is not valid anymore for the fourth cell, which now must contain value /textbfthree.

Let's now analyze our second case of interest in the table below:

3			4	1	2

Now we have two empty cells in our line, and both are located before the tallest building. Our algorithm now finds the two empty cells (they don't necessarily need to be adjacent) and tries all possible values combinations. It's worth noting that, since there are only two empty cells, they both have the same feasible values. Hence, we put the first feasible value in the first cell (in our case **two**) and the other one (**three**) in the second cell. The sequence we created satisfies our constraint, so, as in the previous case, nothing happens. Now we check the other value for the first cell (**three**), which implies a value of **two** for the second one. This particular sequence does not satisfy our constraint, so we set the value **three** as infeasible for the first cell and **two** as infeasible for the second cell. This way, both cells now have only one feasible value, which is added into the respective cell.

Conclusions

The game presented is complete and functional in all of its aspects. The algorithm studied to solve the game introduced a noticeable complexity from the logic point of view. Due to this we reached the logic limits of the board: the current project barely fits on it, but a heavy optimization was needed.

Unsurprisingly, the hardest part to fit was the solver logic; some more resources could be freed with a more extensive rewrite of its code, but we decided to aim for a good compromise between resources usage and code readability. We think we reached our goal, considering we ended up with 91% of the board's combinational functions used and a fairly readable code.

As for the available logic registers, careful use of the **range** parameter was enough to avoid occupying too many resources: we ended up with 4% logic registers used.

During the development, the biggest problem was the difficulty of debugging the project, being used to the ease of debugging sequential languages such as C or Java. To overcome this obstacle, we decided to use the onboard displays to view all possible solutions for the selected cell. Later we found this facility so useful that we decided to keep it in the final build of the project.

Future developments

Currently the limits of the board have been reached so the space for possible developments is quite small. However, with a heavier optimization of the code or using a board with more resources, the following improvements could be coded:

- Generate boards dynamically
- Generate and solve boards with more cells (higher N parameter)
- Let the user enter a specific constraints array for a "custom" board
- Solve boards with missing constraints

References

[1] http://de1.terasic.com/