“2048 game”

made by:

Peshkov Alexey

Maksunova Ekaterina

Timofeev Marat

# GAME RULES

The playing field is a 4x4 field divided into 16 cells in total. To simplify the perception we will assume that the outermost cells (1,1), (1,2), (1,3), (1,4), (2,1), (2,4), (3, 1), (3,4), (4, 1), (4, 2), (4, 3), (4, 4) have so-called outer walls.

Each cell can contain either nothing (then the cell is free) or a tile with a number starting from 2 and ending with 2048.

The number tiles can be moved relative to their original position in four different directions: up, down, right, left.

At the beginning of the game, a tile with the value “2” (with 80% probability) or “4” (with 20% probability) appears.

When a move event occurs in any direction, all cells move in the selected direction along the free cells until they collide with a wall or a tile of a different denomination. If two tiles with the same number “meet”, they are combined into one with the sum of their values.

After each successful move, a new tile (2 or 4 with the same odds) is added to a vacant space.

If there are several tiles of the same value in a row or column, they are merged starting from the side to which they are moved. For example, a row of tiles (4, 4, 4, 4) will become (8, 4) when moved to the left, and (4, 8) when moved to the right. Joining is pairwise: for example, in a row of (8, 4, 4, 4), moving left will join the tiles into a pair, giving (8, 8) rather than (16). Thus, one tile (16) can only be obtained on the next move.

If a tile with the value 2048 appears, the game is considered won and ends. The game is lost when there are no possible moves or combinations after the next move.

  
  
  
  
  
An example of the web version of the game and the situation when the game is over:

# OVERVIEW

In this project we do not aim to make a game “2048”, where all the moves are thought out and made by the player. In this case it is only required to process the field in time (by pressing buttons), but this does not have a large load on the processor.

Our task is to try to create a machine algorithm that calculates the best chains of moves and can potentially end the game by creating the last cell with the value 2048.

Elements that the player will interact with:

* playing field
* “artificial intelligence”, which can select the most successful game moves and will reflect them on the display

To make it all work, all elements will be simulated in Logisim and linked to each other, to logic elements from the standard program libraries, and to the off-the-shelf Cdm-16 processor that was proposed for use in the project.

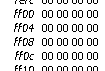
# HARDWARE DESIGN

## Display

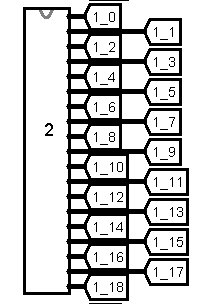
To show the player the changes taking place in the processor memory, we presented information about the state of cells (empty or with value) on 19x19 displays taken from Logisim library.

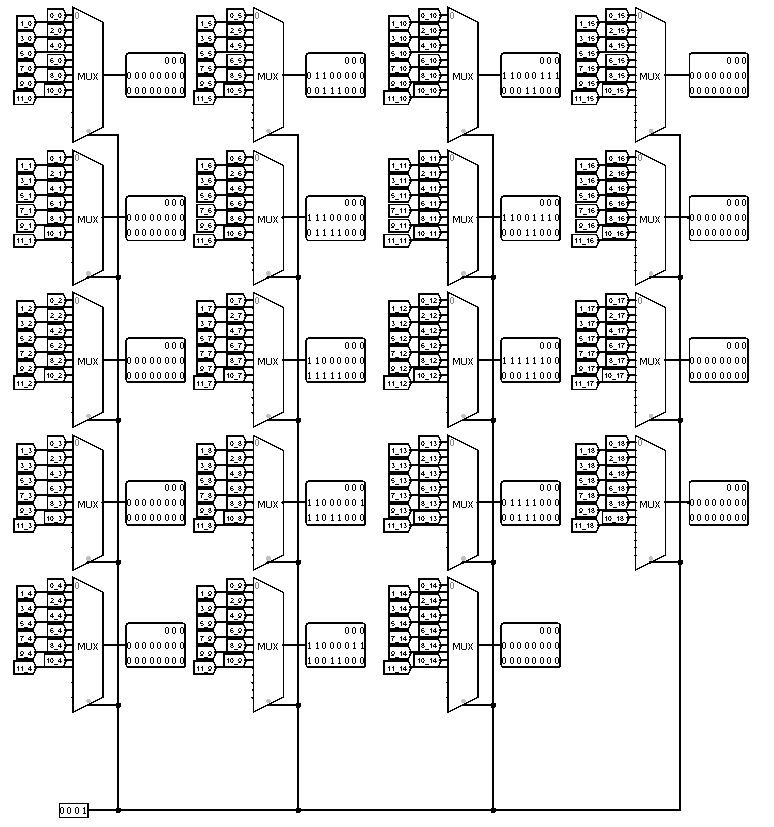
The 19x19 matrix dimension was chosen because of convenience and sufficiency when drawing four-digit numbers, then 4 pixels (16) + 3 indents of 1 pixel each are allocated for each digit.

It is known in advance that one of the preset twelve values may appear in the cell.



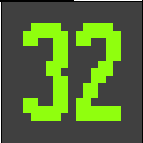
All values of the main field are stored in the processor memory at address 0xff00 to 0xff0f.

Address 0xff00 in the memory image is chosen for a reason. It is necessary to simplify the interaction with the matrix. For example, if you need to make the processor address the 13th tile, i.e. the 13th display, you should translate its number into a hexadecimal system (13 => d) and add it to address 0xff00.

With the help of the multiplexer and the memory cells connected to it, the displays broadcast their current state.

# DESCRIPTION OF GAME BASE

## Numbers

Since all values are obtained by adding identical numbers and calculations start from two, the values in the matrix will show the current degree of two in this cell. Let's reserve the value 0 for an empty cell.

Thus, in order to write a two into the matrix, we need to leave a one in the selected field, since 2 to the 1st degree gives a two. This simplifies the processor (avoiding storing large numbers and expensive operations for them) and is still simple enough to understand and track changes

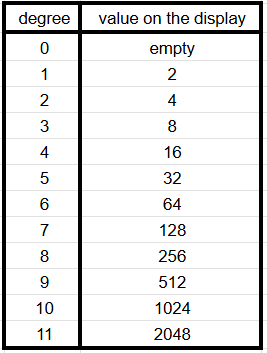
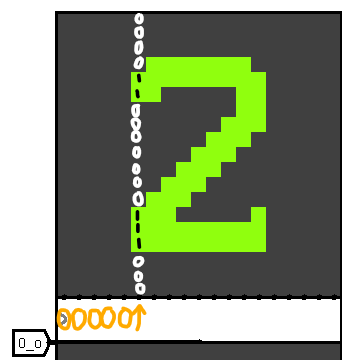


Table of correspondence of the degree value in the matrix with the value on the display:

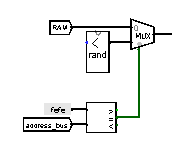
All required numbers were drawn on a 19x19 matrix.

For example, to reflect a completely empty cell, we need to connect nineteen 19-bit numbers, each bit of which is 0, to the display in sequence. The screen will be “completely blank”.

And if it is necessary to draw a double, the units contained in the transmitted number will be responsible for switching on the necessary pixels.

The sixth number transmitted will be 0b0000110000000111000 in binary

## Random

To add a new cell, the randomizer is used 2 times: first, when the cell value is selected, and second, when its position is determined.

The numbers to be selected must be in the range from 0 to 16, which the Logisim component of random can handle perfectly.

However, if we use random with 4 bits, we will have a wire size mismatch, so to eliminate this problem we add extra bits to the 4-bit random in the rand component, which is used in this project.

# DESCRIPTION OF THE GAME BASE

## Storing numbers and reproduction them

To store numbers in memory, they decided to use their representation in the form of powers of two, because thanks to this approach, the largest number (2048) was able to fit into just one byte. This made it possible to halve the amount of memory required to store one game board. And, since a computer player needs to simultaneously work with five fields, the total memory size was 640 bits instead of 1280. In addition, this method of storage made it much easier to display numbers in the Logisim program, as discussed earlier in the HARDWARE section.

## Movement

It would have been possible to write just one function for left movement to handle all movements, and for the other movements simply mirror/transpose the playing field before calling this function. But, since the gain would only be in the amount of code, while the loss would be in performance, separate functions were written for all four directions to speed up the program.

Let's look at the processing of moves for an example:

1. First, we move all cells as tightly to the left as possible so that there are no empty cells between two non-zero cells.
2. Compare the neighboring cells: if their values are equal, increase the value of the left cell by one (we store degrees, so this is equivalent to multiplying by 2), and clear the right cell.
3. Once again, we shift the cells to the left.

To speed things up, we check to see if there were any changes in the first step. If not, we can skip the third step.

In the code sections move\_left, move\_right, move\_down, move\_up 4 different field states are calculated, then they are written to addresses 0xff10-0xff1f, 0xff20-0xff2f, 0xff30-0xff3f, 0xff40-0xff4f respectively.

## Placing new tile

At the beginning of the game, two cells are added to the field with a random value and in a random free space. Also a new cell is added after each move.

The algorithm of adding a new cell works as follows.

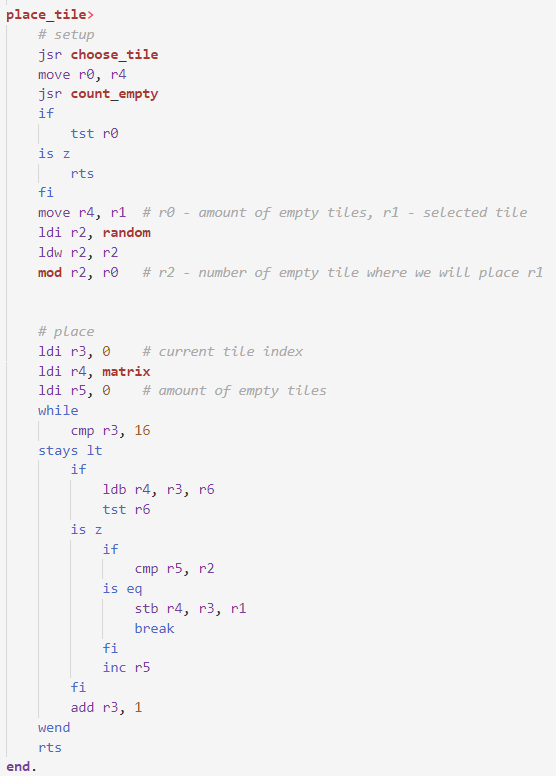
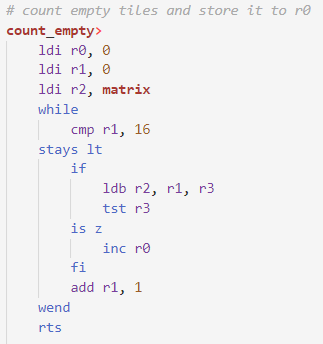
First, the value for the new cell is selected

For this we take a random number from 0 to 16 and if it is greater than or equal to 13, then the value of the cell is 4, otherwise it is 2.

The choice of number in such ranges is due to the chances of tiles of denomination 2 and 4 appearing. In our conditions this chance for 2 is 81.25%, for 4 - 18.75%.

After that, we count empty cells = free\_count, choose another random number = random.

Then we count the position of the new cell as n = random % free\_count.

And we place the new cell in the n'th empty cell

# ROBOTIC PLAYER

## Algorithm of calculating the state

To select the optimal move, we evaluate the state of the playing field after each possible action. The evaluation is based on five key parameters, each of which is considered with a certain weight.

The key evaluation parameters are:

Free cells (free)

The number of empty cells on the field. The more free cells, the higher the chance to continue the game.

Maximum number position (edge)

A check to see if the highest number is in one of the corners of the playing field. Corner positions are considered more favorable.

Smoothness

The sum of the moduli of the differences between the values of neighboring cells (horizontally and vertically). The result is multiplied by -1: smaller differences between numbers improve the score.

Merge capability

The number of pairs of neighboring cells that can be merged on the next move.

Monotonicity (mono)

For each row and column, counts how many numbers are in ascending or descending order. This helps to line up the numbers in a “ladder” to the corner.

The estimation formula:

The coefficients are chosen as degrees of two (e.g., 1, 2, 4, 8, etc.). This simplifies the computation at the hardware level, since multiplication by degree of two can be replaced by a bit shift.

This performance is considered sufficient for a computer player within the project.

## Choosing the next move

The computer player requires field evaluation functions, and we have two of them: eval\_individual and eval\_collective.

eval\_individual evaluates the field by criteria 1 and 2, since this can be done in one loop.

eval\_collective evaluates the field on the remaining criteria, since these 3 remaining evaluations can also be done in one loop.

The evaluation results for each possible move are written to addresses 0xf50, 0xf52, etc.

After all the game fields have been evaluated, the choose\_move function is called, which, based on the evaluations obtained earlier, chooses the best move and clears the memory for auxiliary fields for the computer player's work.

# MAIN GAME CYCLE

The following things happen in the main game cycle:

1. All possible moves are emulated and recorded at the appropriate addresses
2. All received game fields are evaluated according to all criteria
3. The function choose move selects the best move and writes it to address 0xff00 - 0xf0f, the cells in this memory location are displayed on the game board.
4. A new cell is placed on a random free space

# END OF THE GAME

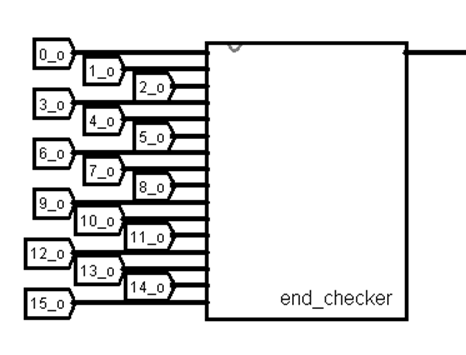
Finally, the game ends under one of the conditions:

* there are no free cells and no two neighboring cells can be connected.
* a cell with the value 2048 has appeared on the field.

To handle all of these situations, a Logisim component was created that checks the state of the field to see if it is impossible to make a move. After evaluation, if the field really cannot be moved or the desired end number has appeared, the component writes the value 0 at address 0xfefc (global section - is\_game\_over), which is processed at each iteration of the main loop. Now the loop condition will not be met and the game will end.

The input is the degrees of twos recorded in the cells responsible for storing the field values.

With the help of a small comparator we compare all neighboring cells (Vertical neighbors, Horizontal neighbors), and if all are different and not empty, the state of the global section is\_game\_over is switched from 1 to 0.

  
  
  
Also, in parallel, the cells of the field are checked for degree 11, representing the shortened 2048 entry. If at least one cell contains the given value, the game ends the same way as in the previous condition.