OPEN HARDWARE 2015

ROACH SWEEPER
PROJECT REPORT

BOUSIAS DIMITRIS
MAGEIROPOULOS EVAGGELOS
TEAM NUMBER XIL-94875.

ROACH SWEEPER

A variation of the classic Minesweeper Game for two player local play, completely designed in VHDL.

It was designed on Xilinx Vivado for the Digilent Nexys 4 DDR board.

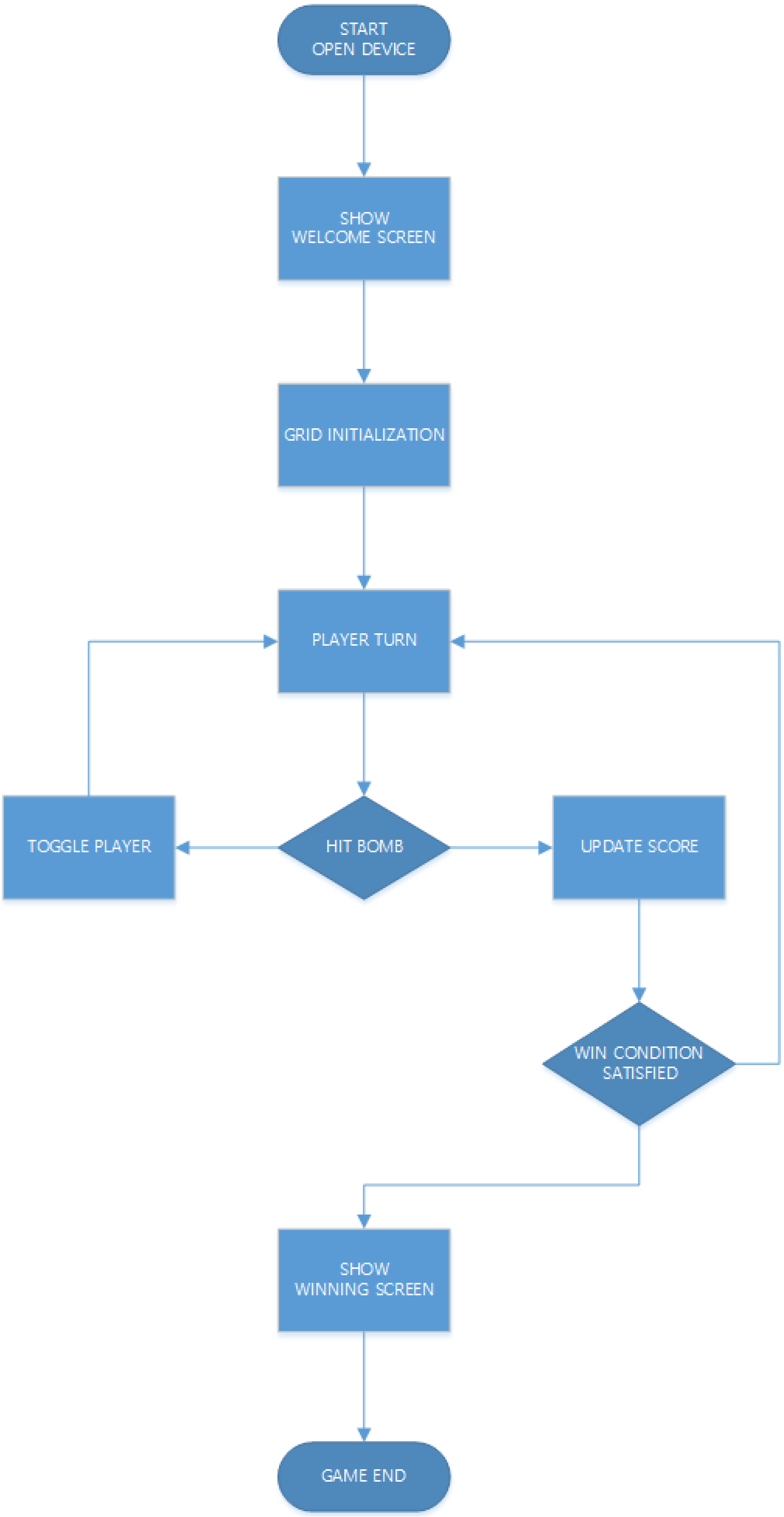
Our purpose was to explore our abilities on a register transfer level design, explore the FPGA capabilities all while creating something fun and educational.
The design was presented at the Microprocessor and Hardware Lab of the Technical Univercity of Crete on "Career Day" when local schools came visiting our Department, as well as on "Science Day" event held by the Univercity that introduced children into various scientific topics and fields.

THE GAME

The ground is infested! Something has to be Done!

In this game two players, seated closely enough to use the same controller (hot seat),
try to outsmart each other in finding more of the Roaches(mines) that hide on the Map.
After selecting their favorite Avatar from the Character Selection Menu, the Map is filled with randomly placed Roaches which the players try to uncover based on nearby clues.
As long as a player is succesful in finding roaches he keeps playing. When he fails to do so its the opponents turn.
The first one to find at least half of them is Declared Winner!

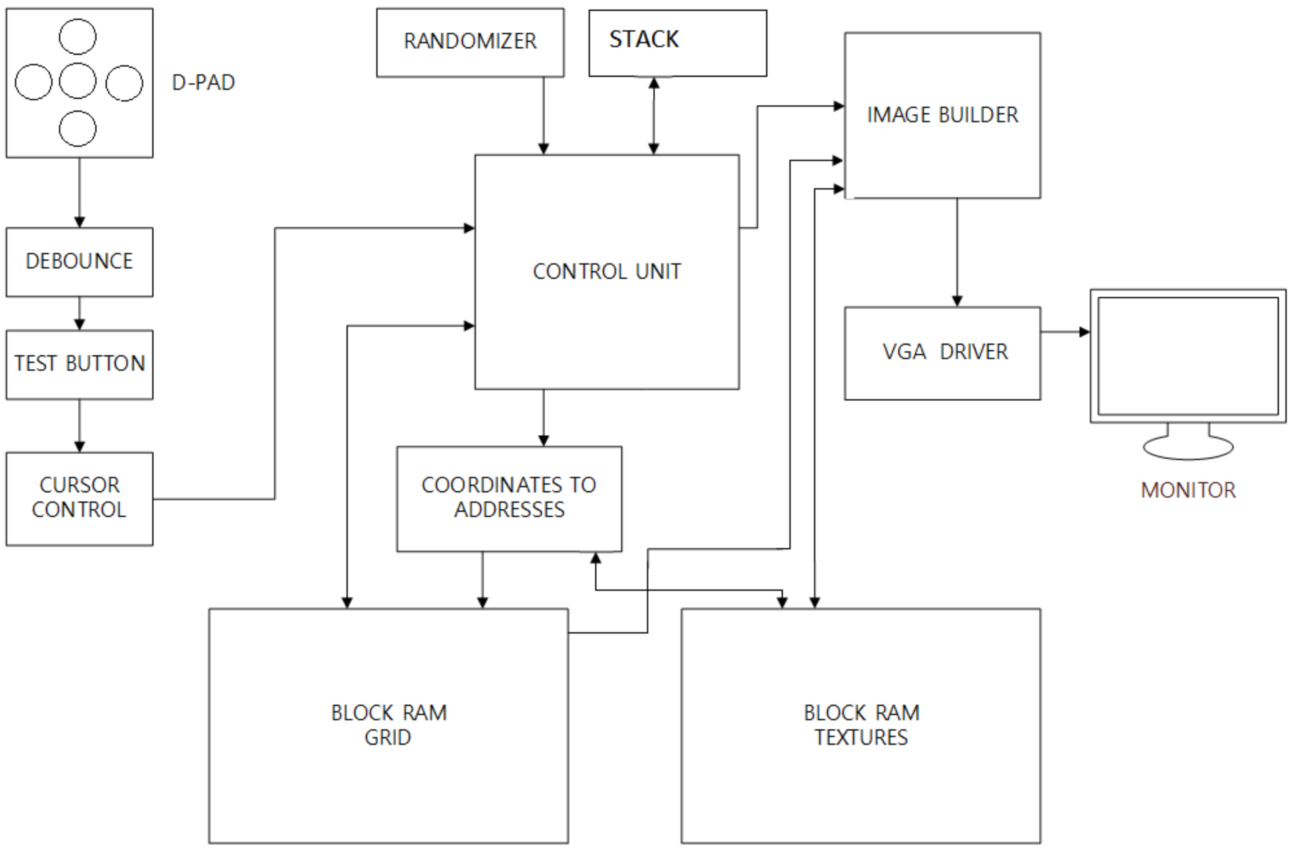
An Abstract but repressentative flow diagram follows.


START
OPEN DEVICE
SHOW
WELCOME SCREEN
GRID INITIALIZATION
PLAYER TURN
TOGGLE PLAYER
HIT BOMB
UPDATE SCORE
WIN CONDITION
SATISFIED
SHOW
WINNING SCREEN
GAME END

<><><><><><><><><><><><><><><><><><><><>
THE DESIGN

The whole functionality is completely modular, but due to the nature of the application it doesn't contain much parallelism.

Here is a somewhat simple and abstract Block Diagram of the Design
<><><><><><><><><><><><><><><><>><><><><><><><



Players use the D-pad like placed buttons of the Nexys 4 board to navigate through the menu and play the game.
Their Inputs are Normalized (button debouncing and pulse lasting for 1 clock cycle)

The Memory containing the Information about the Map state (grid) is shared by the Control Unit (where the game functions are implemented) and the Image Builder that depicts that information to be pressented to the players using the appropriate textures or hard coded graphics.
The Inital State of the Map is Random achieved with the use of an LFSR Galois.

The design also contains a Menu Controller , a Sever Segment Controller and Custom Memory Interfaces (not depicted in the diagram above).

MODULES (.VHD files)

Game Functional Modules

Cursor and Movement

The modules that control the cursor position on the game map.
It has Up, Down, Right, Left as inputs which are binded to the corresponding Push Buttons of the

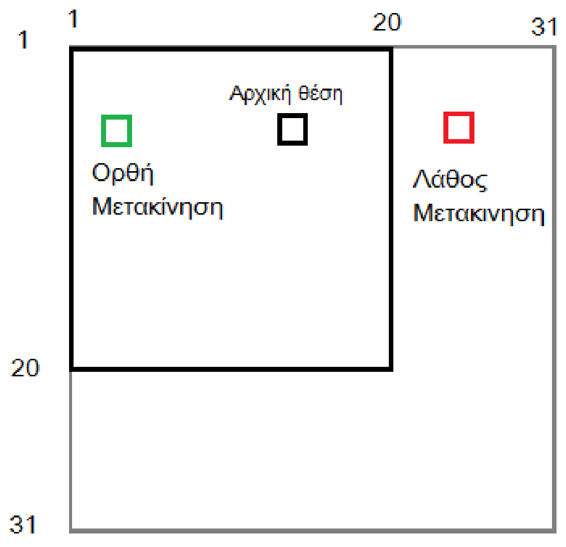
Nexys board or any controller that can be used.
It also contains 4 input signals binded to the 4 most right Switches of the Board that activate 4 Move flags which increase the number of tiles a player can move in one press of the desired direction button. Those flags correspond to increasing powers of 2 and help with the speed of the game flow.

They are concatenated into 1 single signal as follows
Move<=Moves3&Moves2&Moves1&Moves0;

e.g. if Moves3 and Moves0 are activated the player can move 2^3 + 2^0 additional tiles in the direction he selected.

The comparison is Synchronous so we can have only 1 cursor position per clock cycle.
We had to pay special attention to keep the position signals within the proper range of values.
Our Map contains 20x20 tiles so the signals had to be 5bits long thus we had to take care of "overflow" cases when adding or substracting coordinate positions.

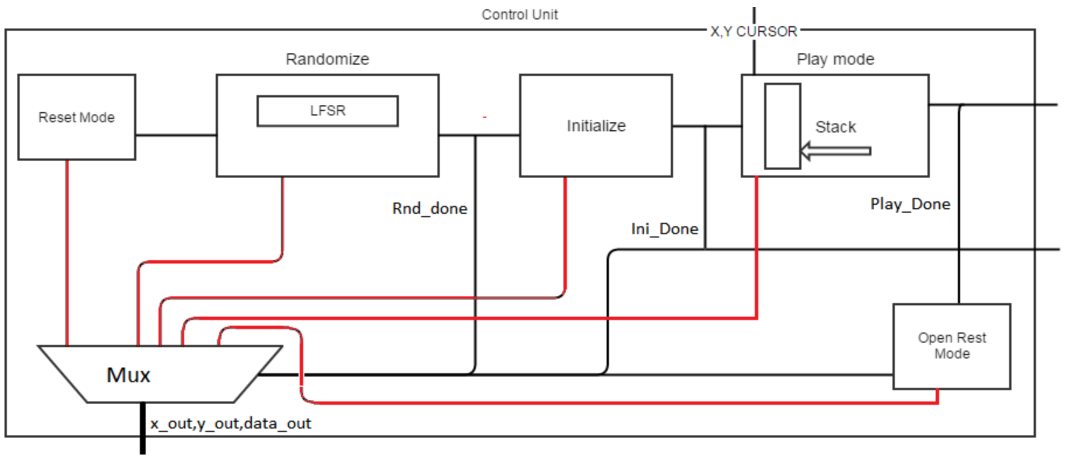
E.g. The user selects Right while the cursor is on a tile with coordinate y=15.
His Move flags add to 9.
Instead of positioning the cursor out of the grid the controller should consider tile 20 as final start over.



<><><>>>>>>>><><><><><><><><><>

Controller

The game control module that contains the 5 basic game functions: Randomize, Initialize, Play, Open Rest, Reset.



><><><><><><><><><><><><><><<><>>

The 5 functions mentioned above all interact with the Grid Memory but only one can be active at any time. Based on that we use a 3bit select Mux which based on the Done_flag signals it recieves from any module chooses which one will be sending Data(x,y coordinates, Tile content explored etc) to the memory.

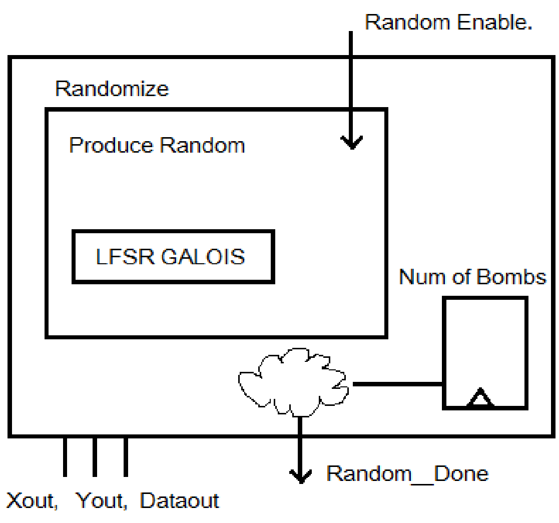
The Control module is also in charge of sending useful Data to the Image Builder such as Last_Hit of Players , Winner, etc.

Random_Mode

This module produces the 81 random coordinate pairs that correspond to the position of the Roach mines that will be placed on the map before the game begins.
Synchronous Logic FSM checks the number of already inserted Roaches with a proper number of Wait States for right synchronization.
Ones our threshold is exceeded (81 is an odd number so there can be no draw/tie in the game) it enables the Done flag signal so the sequential execution flow of the program continues.

It contains an LFSR Galois that starts when the board is powered on.
To battle the pseudo-randomness we only accept the LFSR's outputs after the users have navigated through the menu and made their choices, so that the sequence of numbers truly starts from a different point each time.

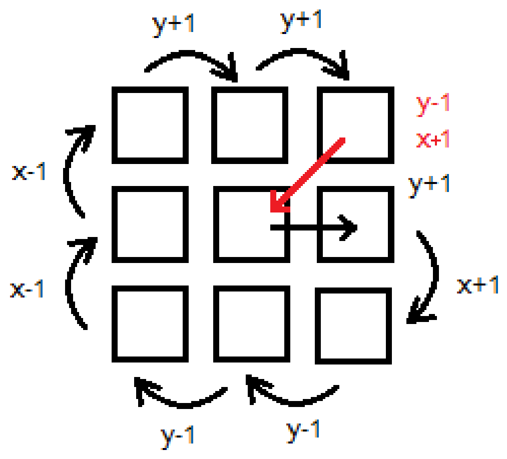
We check the produced nymbers and discard those that are not within the range of 1-20.
Thus the execution time of the algorithm may differ based on the beginning point of the LFSR sequence.



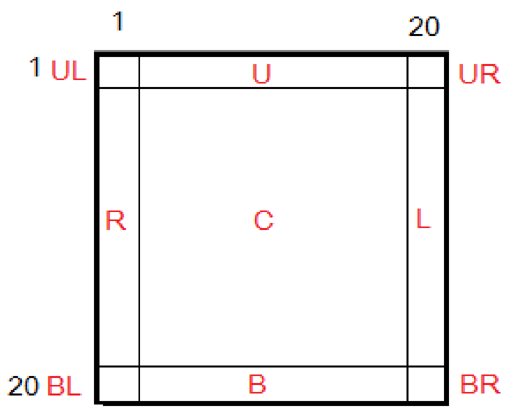
<><><><><><><><><><><><><><><><><><><><><<<

Initialize_Mode

After the execution of the Randomize Module(check with the Random_Done flag) another Synchronous FSM starts.
Its purpose is to access the whole map , to scan around each empty tile and assign to it its corresponding clue digit based on the number of Roaches that exist on the 8 tiles surrounding it.



This FSM is divided into 9 different “state-chains” based on the position type of each tile.
These chains are as follows Upper Left, Upper, Upper Right , Center Left, Center ,Center Right Bottom Left, Bottom, Bottom Right.
<><><><><><><><><><><><><><><><><><><><><><



The whole process finishes in a fixed number of clock cycles execution time.

Play_Mode and DigitsforScore

This module contains the main game function. It basically contains a synchronous FSM with a big number of states.
It starts after the initialization of the map has been finished and the Ini_Done signal flag is set 'high'.

Player 1(red) is the one to start first. When he selects 'Confirm' (N17 button) to exlore a tile's content we sent a request signal, along with the proper coordinates, to the BlockRam to inform us about its content. When we have the data(after one wait cycle) we check if the tile is already exlored or not. If it is we ignore the Confirm command and the same player is to play again.
If not we check if the tile contains a mine and if it does we change the unexplored mine to an explored one with the colour of the player that found it.
The score is increased and the whole process starts again for the player.
If the tile contains a number it is changed to explored and the player passes the turn.
The coordinates of the last tiles that were explored are kept into 10bit registered output signals to be used by the Image Builder.

If a tile contains nothing (empty) then all the surrounding ones should be opened recursively like the image below shows.
To do that we use a LIFO stack.
The steps of this procedure are as follows.
1. Explore the 8 neighboring tiles.
2. If any of them is empty Push its coordinates into the Stack
3. Pop the last entry of the stack. Move to those coordinates and repeat from step 1 until the stack is empty. Pass the turn to the opponent.

The game ends when any player reaches a score of 41.Assign the winner, Enable the Done flag signal and move to a stable state(can only be exited by resetting the FPGA).

The score is stored into a 12 bit registered output signal as a concatenation of the two individual 6 bit scores.
To accommodate its display on the Image Builder and SevenSegment Display we have to transform it into BCD format.
Thus its passed on 2 instances of the DigitsForScore module which is Basically a BCD decoder.
For example if a players score is 001111=15 the outputs of the DigitsForScore will be 2 signals with values 0001=1 , 0101=5.

Open_Rest_Mode

When a winner is declared the contents of the whole map shall be revealed.

In this module we can all 400 of tiles and we check its contents if any tile is still unexplored (Its most significant bit equals 0) its set to explored.
Then it moves to a final state which can only be exited by resetting.

Reset Mode

The map access and content overwriting is quite fast (2 clock cycles) so we decided that the time the Reset button is pressed by the user is enough for the whole map to be whipped clean.
We use a 2 State Fsm.One state to move to a specific tile and one to reset its content to either a value that represents the map border or an empty unexplored tile.

MEMORIES
For the purposes of our design we created 10 Block Memories using the IP Block Memory generator, as well as individual custom interfaces.

The main memory that contains the Map state and game critical information is a Block RAM called BlockRamGrid

Its a Dual Port RAM 5x512. It's content can be read by both Control and Image Builder but Control is the only one able to alterit.
We store 5 bits of Data for each game tile with the following encoding

	UNEXPLORED	EXPLORED
Empty	0 0 0 0 0	1 0 0 0 0
1	0 0 0 0 1	1 0 0 0 1
2	0 0 0 1 0	1 0 0 1 0
3	0 0 0 1 1	1 0 0 1 1
4	0 0 1 0 0	1 0 1 0 0
5	0 0 1 0 1	1 0 1 0 1
6	0 0 1 1 0	1 0 1 1 0
7	0 0 1 1 1	1 0 1 1 1
8	0 1 0 0 0	1 1 0 0 0
ROACH	0 1 1 1 1	1 1 1 1 0
		1 1 1 1 1
		1 1 0 1 0
BORDER		1 1 0 1 1

The Inputs are Clock, and Reset signals, X,Y coordinates that show which tile we would like to read/ edit, a Set signals that acts as Write Enable.
The coordinates are translated into proper addresses in the CoordToAddr module instances, using the transform Addr= X*22+Y

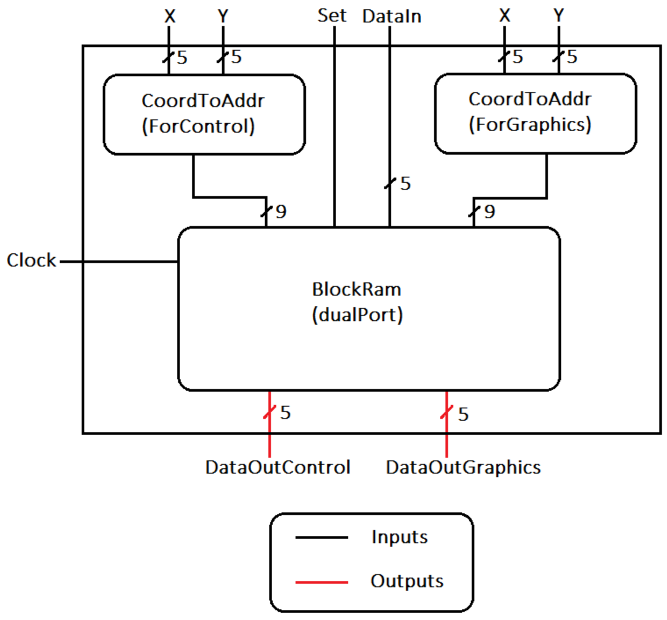
Address	X,Y
0	Border
1	Borde

On the left Matrix we can see the right order in which Data is stored in the Memory and on the right one the way that they will be presented.

	r
2	Borde r
.	Borde r
.	Borde r
20	Borde r
21	Borde r
22	Borde r
23	1.1
24	1.2
.	.
.	.
460	20.20
461	Borde r
.	Borde r
.	Borde r
461	Borde r

Borde r	Borde r	Borde r	Borde r	Borde r	Borde r	Borde r	Border
Borde r	1.1	1.2	.	.	1.19	1.20	Border
Borde r	2.1	2.2	.	.	2.19	2.20	Border
Borde r	3.1	3.2	.	.	3.19	3.20	Border
Borde r	4.1	4.2	.	.	4.19	4.20	Border
Borde r	5.1	5.2	.	.	5.19	5.20	Border
Borde r	.	.	.	.	.	.	Border
Borde r	.	.	.	.	.	.	Border
Borde r	19.1	19.2	.	.	19.19	19.20	Border
Borde r	20.1	20.2	.	.	20.19	20.20	Border
Borde r	Borde r	Borde r	Borde r	Borde r	Borde r	Borde r	Border

Here is a full Block Diagram of this memory:



ROMs
The rest of the memories are ROMs containing Textures in binary form.

Block Memory Textures

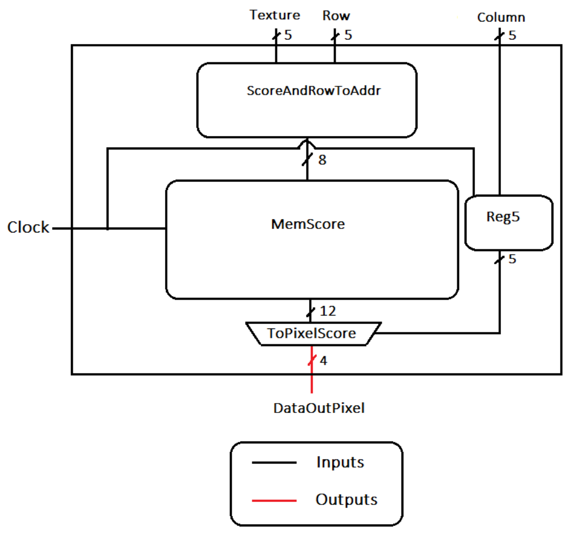
This memory has dimensions of 92x256 and contains the numbers and Roaches textures that will be drawed on the map during the game execution.
It communicates with the Image Builder who has only read access.
It is initialized with a .coe file.

Inputs are Clock, Texture (5bit signal indicating which texture we would like to draw) and Row, Column 5bit signals that correspond to the specific pixel we want from said texture.

For the proper addressing transformation we designed TextandRowToAddr the output of which points to a 92 bit line in the memory

This line is fed into a multiplexer which uses the Column as a 5 bit selection signal to give us the 4 bits we will need each time.
We also use a 5 bit register to delay the Column signal for 1 clock cycle.

empty	0 0 0 0
1	0 0 0 1
2	0 0 1 0
3	0 0 1 1
4	0 1 0 0
5	0 1 0 1
6	0 1 1 0
7	0 1 1 1
8	1 0 0 0
ROACH	1 0 0 1



The remaining 8 memories work all with the same principle but for different texture categories and different colour resolution. (the images used for the menu and the avatars are in 12 bit and 16 bit respectively).

Creating the .Coe Files

We started from the images we wanted to load on .jpg or .bmp format.
We used a matlab script we road to translate the colour values and save them on a .coe file.

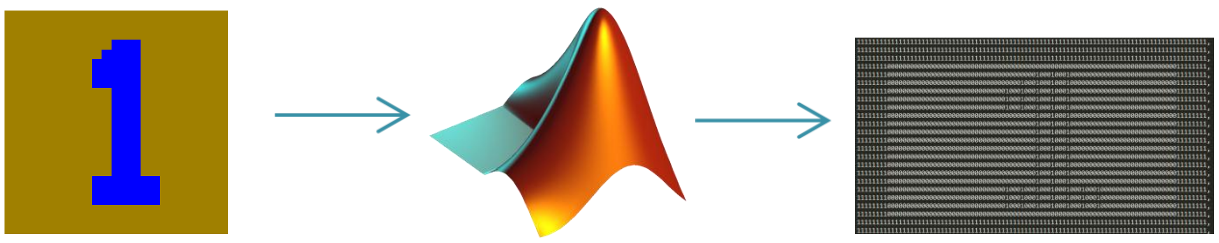
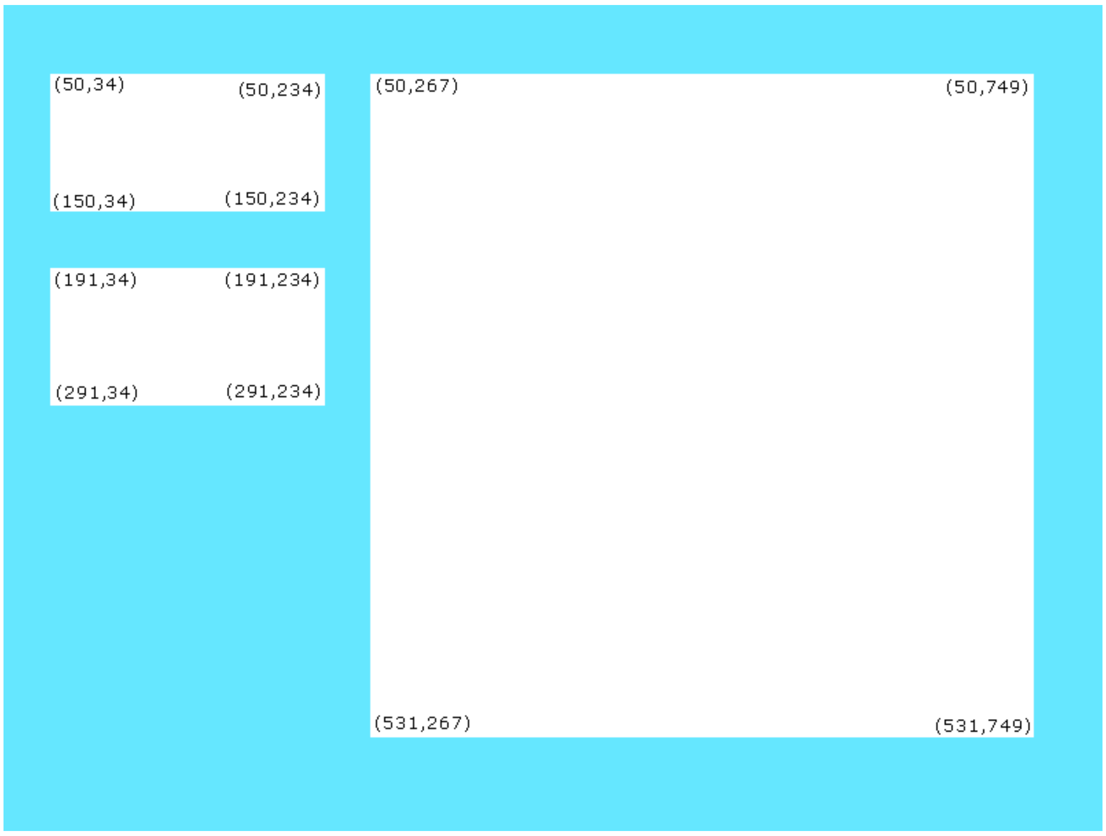


Image generator

The image generator (Imgur.vhd) uses a plethora of signals to communicate with other components of the project, namely the grid memory, the PlayMode and the Menu. Additionally, it has connections with many Block ROMs, that hold textures for the elements that appear on the screen. It also uses an implementation of the VGA protocol (taken from the demo project of the board), in order to properly communicate with an external monitor.

The working principle of the image generator is simple enough: It keeps track of the current column and row and selects which color to send based on their values, as well as the state of the game. Every element has been mapped to particular pixels while the project was being designed, in order to show the appropriate content on the corresponding pixel. Here’s an example of the game’s layout:



The main process of the image generator is the pixel_mapping: It checks for the current state of the game and assigns a state value to the variable ‘gun’. The state is resolved to 12-bit hex color in the process ‘pistol’ and the data vector is outputted to the monitor.

There are many processes that control the row and column inputs of the memories holding texture data. Those inputs are being constantly updated, even when there is no need to read from that memory. The pixel_mapping process decides if it must read the output of a texture memory when needed.

Some texture memories, for example the Face memory and the Score memory hold multiple textures: In this case, the input used to load a particular texture is controlled by a process that takes into account all the required data to select the right texture: E.g, the signal that selects a texture from the face memory is controlled by the process Face_select. This process takes into account whether the game has started or not. If it has started, it loads the first or second player’s character face, depending on the row’s and column’s current values, as well as each player’s selection. If it hasn’t, it loads each texture based on the row’s and column’s current values, in order to show them on the character selection menu.

Most texture data are encoded in order to minimize space consumption: The initial menu texture uses two bits per pixel, in order to render 4 colors and every other menu screen uses 1 bit per pixel. The ‘WINNER’ and score textures use 1 bit per pixel as well. The roach texture memory uses 4 bits per pixel that are converted to color according to whether the roach was found by the blue player, the red player or not at all (at the end of the game), in the process roach_coloring. The face texture memory uses 16 bits per pixel, that is then reduced to 12 bits per pixel, in order for the color to be VGA compatible.