Valeriia Okhmak

07.12.2023

Designing the Video Game on FPGA

Project Report

Introduction

Design of the video games can be quite challenging and time consuming.

Creating a game on a Field-Programmable Gate Array (FPGA) offers several advantages, although i also comes with its own set of challenges. Potential advantages of using FPGA for the game implementation are: parallelism of the processes on the FPGA, energy efficiency, reconfiguration possibilit. Despite these advantages, it's important to note that developing games on FPGAs also poses challenges. FPGA development requires specialized knowledge in hardware design languages (such as Verilog) and an understanding of hardware architecture.

For this project the game Sokoban was chosen.

Game: Sokoban

Figure 1. Game

Game description

Sokoban is a classic puzzle game that originated in Japan in the early 1980s. The word "Sokoban" translates to "warehouse keeper" in Japanese. The game was created by Hiroyuki Imabayashi and was later published by Thinking Rabbit. Over the years, Sokoban has gained widespread popularity and has become a timeless puzzle-solving challenge.

Game Concept:

The purpose of the game is to win.

The objective of Sokoban is straightforward yet deceptively challenging. Players control a character (often represented by a warehouse worker or a character pushing boxes) within a maze-like environment consisting of walls and open floor spaces. The goal is to push all the boxes onto designated storage locations (usually marked by targets (goals)).

Game Elements:

Student Character - Agent: The player controls an agent that can move horizontally and vertically across the grid.

Boxes: The main puzzle elements are **mushrooms** that need to be pushed to specific target locations.

Targets: Designated spots on the game board where boxes must be placed to solve the puzzle.

Walls: Immobile barriers that restrict the movement of the worker and boxes.

Rules:

1. The agent can move horizontally or vertically, one square at a time.
2. The agent can push one box at a time, but cannot pull them.
3. Boxes can only be pushed, and they cannot be pushed into walls or other boxes.
4. The puzzle is solved when all boxes are on the target locations.

Challenge:

What makes Sokoban challenging is the need for careful planning and foresight. Pushing boxes into corners or against walls without an escape route can lead to unsolvable situations, requiring players to rewind or restart the puzzle!

Program description

Current implementation of the game includes one level of complexity. This configuration of the ropm allows the program to be atrachtable for players while remaining its difficulties. Nevertheless, inserting the number of additional levels to the game is an easy task - that is why it was not put on the practice of this project. The game was developed on the FPGA board NEXUS A7 with hardware design languages - Verilog.

In order to visualise the elements of the program the following block diagram was designed.

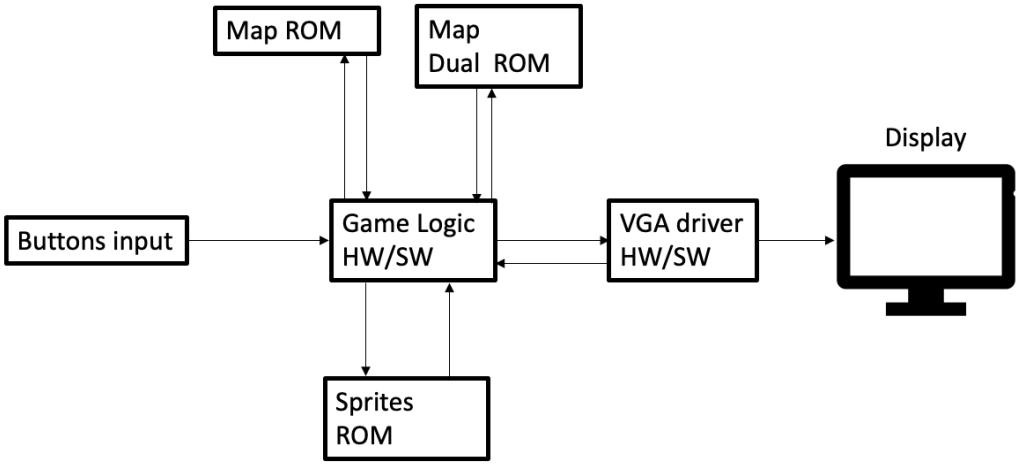


Figure 2. Block diagram of the program

Buttont input - represents the block that reads the input signals from the command device - fortunately the board (NEXYS A7) includes the press buttons - right, left, up, down and the middle button, fig. 3.

Game Logic - block that describes the control logic. It is connected to the two memory blocks, VGA driver input and reads the signal from buttons. The Game logic block basically represents the top module in the program that includes submodules.

VGA driver - is a block - main function of which is to read the input color data from the Game Logic and visualise one pixel at a clock. This is an important part of the program as it calibrates the currently drawn signal with respect to the synchonisxation.

Map ROM - block that represent the memory where the layout of the game level is saved.

Sprite Dual ROM - block that represents the memory where the sprite color data is stored. The dual configuration was chosen in order to access the data of the moving sprites and stationary sprites for the layout separately - it is an important adjustment.

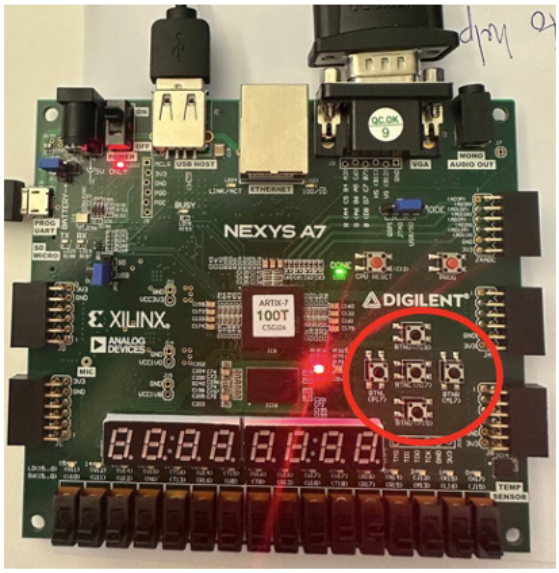


Figure 3. NEXYS A7 board and input buttons

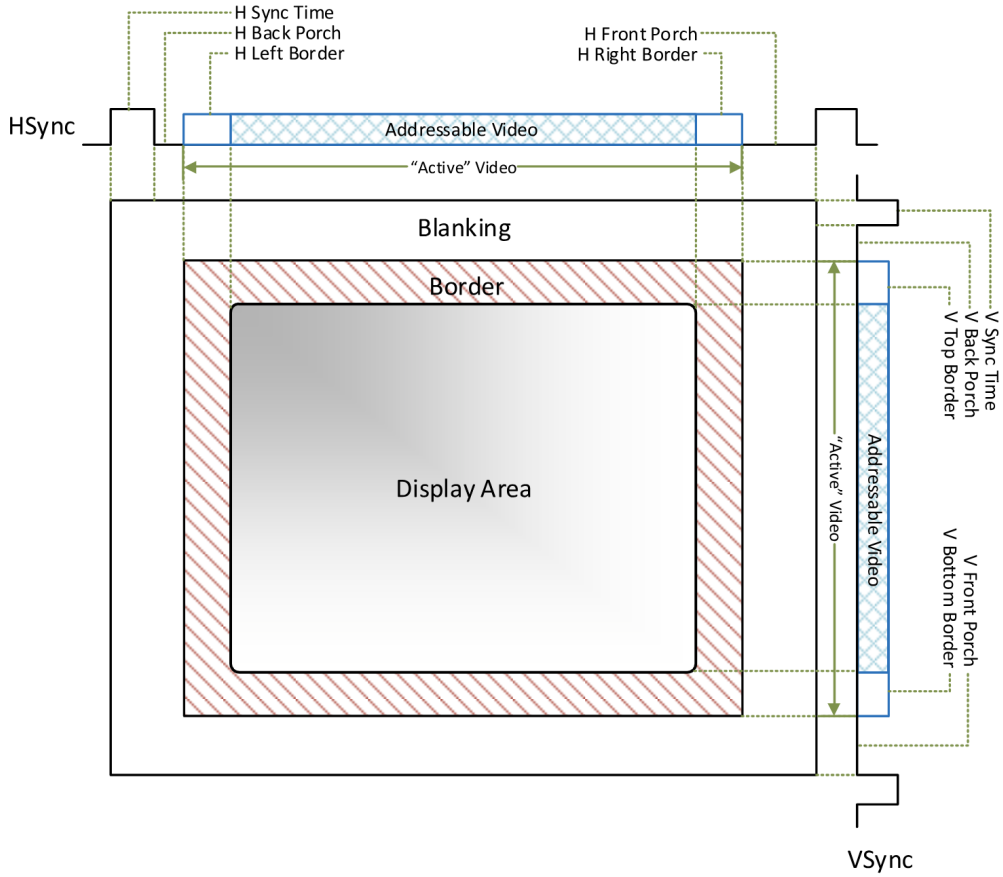
The above diagram is the simplified representation of the program and the more detailed explanations will be provided in the further sections of the report.

Displays and VGA

On an electronic display each individual pixel is usually composed of three sub-pixels, one each of red, green and blue. These are the primary colors which can be combined in various portions to form other colors. By varying the brightness of each sub-pixel (controlling how much of each red, green and blue light is produced) different colors are perceived on the display [1].

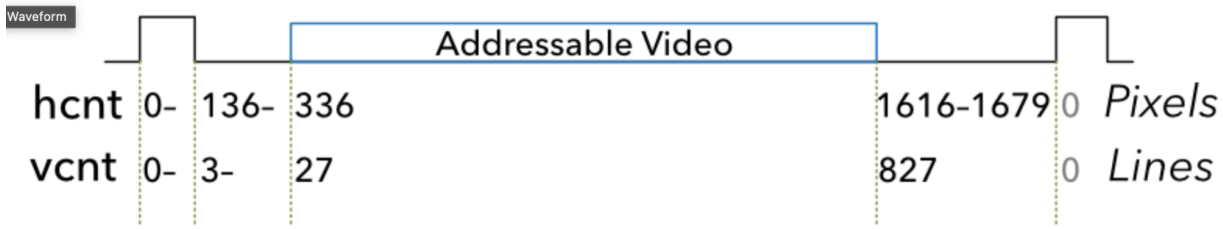
The function of VGA module in the game is to translate coordinate and pixel settings information received from the graphic display module and the text display module into VGA control signal, and then correctly display the image information by a monitor.

As it was discussed in the class the VGA output consists of three 4-bit outputs for red, green, and blue pixel intensity plus two control pulse signals for horizonatal and vertical syncing. In each clock cycle a single pixel output is produced, so to draw a complete frame of pixels takes width×height clock cycles. To generate the required control pulses and output coloured pixels to the screen, it is required to build two timers, one horizontal and the other vertical. The screen active pixel area and the visualized data would be configured respectively to the signals of synchronisation.



In the proposed implementation of the game the VGA driver outputs two signals curr_x and curr_y that indicate the current visible pixel being drawn.

The addressable area of the display is as described below:



It is important to notice that the indices for the x and y (current pixels drawn) start at the top left corner of the display and increment to the right and bottom respectively[2].

The resolution of the display supposed to be 1280x800 for the current implementation [3].

VGA driver module fully correlates with proposed implementation of the class project.

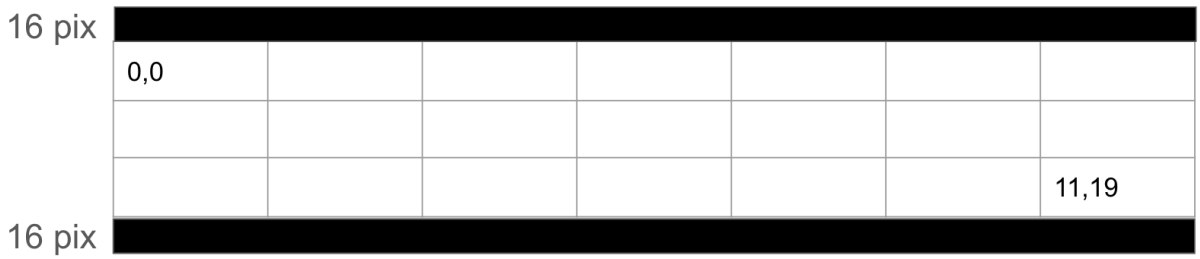
Design description

The game layout is represented as a grid of the 240 cells - 12 rows and 20 columns. As the area of the visible pixels is 1280x800 the top and the bottom 16 rows of pixels are visualized as black lines.

This size was chosen in order to fit the whole screen with sprites of the size 64x64 pix that refers to the image of 8x8 pix.

Display/Screen grid representation:

240 - whole cells
0.5 cell remaining 32 pix at the y. 800/64 = 12.5, 16 pix at the top and bottom of the screen



In order to create a map of 240 cells filled by the sprites the simple symbolic layout is mapped to the corresponding sprite ID. An example of map layout:

Size 12X20

```
1   ====================
2   =====############===
3   ==###----#------#===
4   ===#-------------#==
5   =#------###------#==
6   ###,------#-----#===
7   =#=-----------#====
8   ==#-------,-----#===
9   ===#---##------#====
10  =====##---#-##=======
11  ======####==========
12  ====================
```

The sprites of the game can be categorised as the stable and movable. The stable sprites correspond to the layout of the game level and the moving sprites are dedicated to the objects that supposed to be

manipulated. The idea of the sprites placement is to mask the level layout that contains stable sprites with the movable.

The symbolic layout contains only information that represent a game level and don't assign the placement for the moving sprites.

The memory block that contains the color data of every pixel of the sprite has the following instantiation:

```
blk_mem_gen_0 sprite_mem (
  .clka(clko),    // input wire clka
  .addra(address),  // input wire [5 : 0] addra
  .douta(value)  // output wire [11 : 0] douta
);
```

It receives the address of the current pixel being drawn and outputs the 12 bit color data.

For the current implementation the Dual ROM was chosen in order to save separately the stable and moving sprites.

As it is already mentioned the VGA output provides the input to the game logic - curr_x and curr_y, that informate about the index of the pixel that has to be visualised. Nevertheless, it is obvious that in order to “draw” the image of 8x8 pix in the area 64x64 the same color data has to be repeated 8 times for every pixel of the 8x8 image.

In order to visualise the 8x8 pix picture as a 64x64 pix - every pixel has to be plotted as a little block 8x8. The pixel can be repeated 8 times with the use of the shift right by 3 (2^3).

The memory block that contains the data about the location of every sprite at the grid has the following instantiation:

```
blk_mem_gen_1 map_mem (
  .clka(clko),    // input wire clka
  .addra(square_address),  // input wire [8 : 0] addra
  .douta(square_id)  // output wire [2 : 0] douta
);
```

It accepts the address of the data in the .coe file and provides the corresponding value.

The same data of the grid map is saved at the Dual ROM memory - further discussion of the reasons why the Dual ROM was exploit is provided in memory section.

The program reads the sprite id that has to be drawn every 64 pixels - this procedure is also done with the shift right by 6 (2^6).

Sprites:

Stable sprites

The following snipped of the module drawcon gives the better understanding of how the data is visualised:

```
if ( curr_y >15 && curr_y <783 )  begin
        square_x = curr_x>>6; // shift by 64, grid divider
        square_y = (curr_y-15)>>6; // shift by 64, grid divider
        carachter_overlap=1;
        square_address = square_x+square_y*20; // address of the sprite id
        x = (curr_x-64*square_x)>>3; // shift by 8
        y = ((curr_y-15)-64*square_y)>>3; // shift by 8
```

The square_address is passed to the block memory of the map that outputs the square id that has to be drawn at every 64x64 cell of the grid.
Then the values of address is assigned by the value of the x and y with respect of the base address of every sprite.

```
case (square_id)
            wall: address = x + y*8 +wall_offset;
            floor: address = x + y*8 + floor_offset;
            target: address = x + y*8 + target_offset;
            background: address = x + y*8 + background_offset;
        endcase
```

This mapping is performed for the stable sprites.

Moving sprites

For the moving sprites the assignment is simple and can be seen in the following snipped

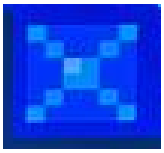
```
if (square_x==obj_x && square_y==obj_y) // check the value of the current
        begin                           // x and y of the grid
        address = x + y*8 +128 ;        // if it matches the location
                                        // of moving obj - assig the addres
```

More detailed code explanation can be found in the code files attached and appendix to the report.

Every sprite has a unique binary ID:

```
translate_dict = {
  "#": "000",  # wall
  "!": "001",  # crate
  "@": "010",  # agent
  "-": "011",  # floor
  ",": "100",  # goal
  "=": "110"   # grey
}
```

Pictures of the sprites 8x8 pix. The pictures in the table are 80x80 pix for the better perception.

ID	picture	binary ID
0		000
1		001
2		010
3		011
4		100
5		101
6		110

The block diagram of the modules of the program is presented below.

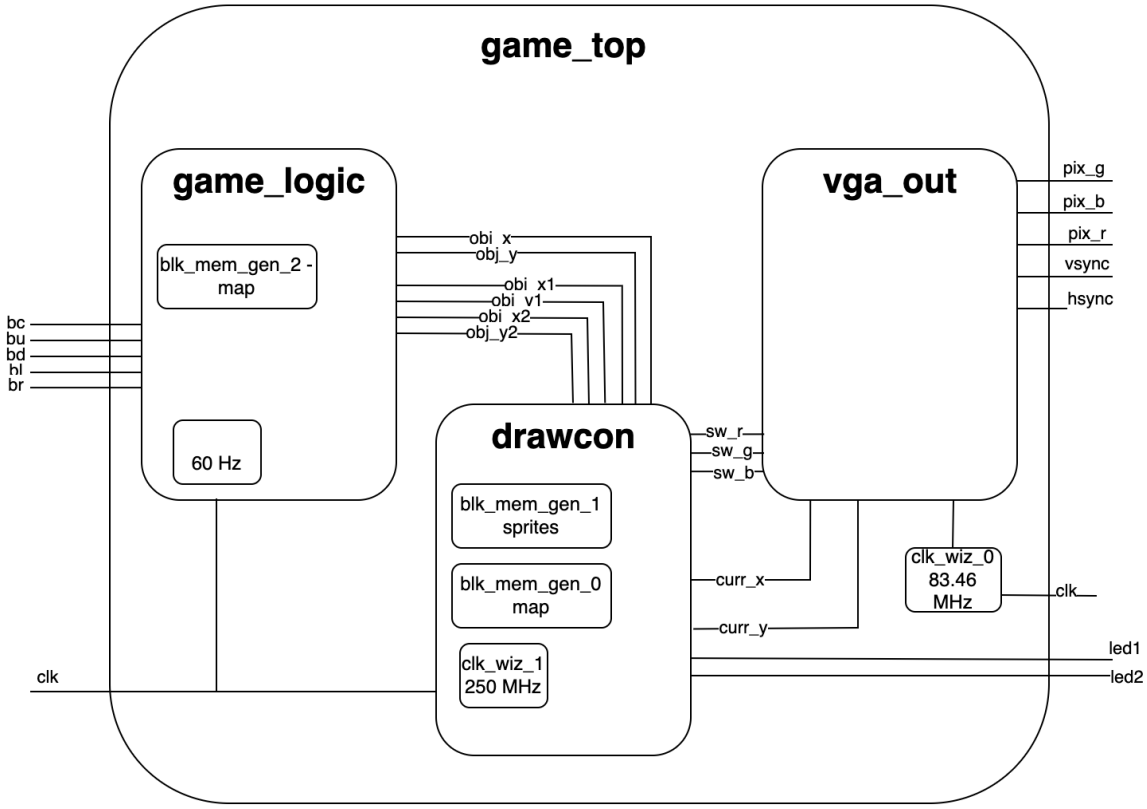


Figure 4. Block diagram of the functional blocks

Inputs: bc (center), bu(up), bd(down), bl(left), br(right) - input buttons

Output: pix_g, pix_b, pix_r, - color data: green, blue and red respectively: vsync, hsync - synchronisation signals: vertical and horizontal; led1, led2 - led signals.

game_top - the top module of the game, main function of which is to collect all of the submodules, this module contains user defined clock of 60 Hz.

game_logic - module that is responsible for the logic of the game, the main function of this module is to output the coordinates of the moving objects of the game.

drawcon - module main function of which is grid map and sprites placement arrangement.

vga_out - VGA driver, module main function of which is to arrange the correct pixel representation on the display in accordance with the input color data.

Clock

Clock arrangement in the program is the most important part of the design. At the current design there are four clocks.

clk - the on-board clock runs at 100 MHz.

clk_wiz_0 - the clock for the every pixel drawn 83.46 MHz.

clk_wiz_1 - the clock for the sprite color data to be extracted from the mem block 250 MHz

clk_60hz - the clock for the control inputs reading.

For instance - the clock for the sprite color data has to be 3 times faster than the clock for the VGA output - so at every clock cycle the color data is already extracted from the memory block and ready to be visualised.

Memory

The memory arrangement of the program has an interesting adjustment of accessing the map data in two blocks.

Single ROM is responsible for visualising the layout of the game and the Dual ROM is present for the validation of the game logic conditions, the Dual ROM was a better option comparing to the Single one as it gives the data of the next two blocks simultaneously - at the same clock.

Game logic

Game logic of the program follows the rules of the game.

The basic idea of the game logic is to check the current state of the agent and at every state accumulate information about following blocks (one or two) in all four directions. I would like to notice that whether the agent wants to know what the second block is depends on the first block.

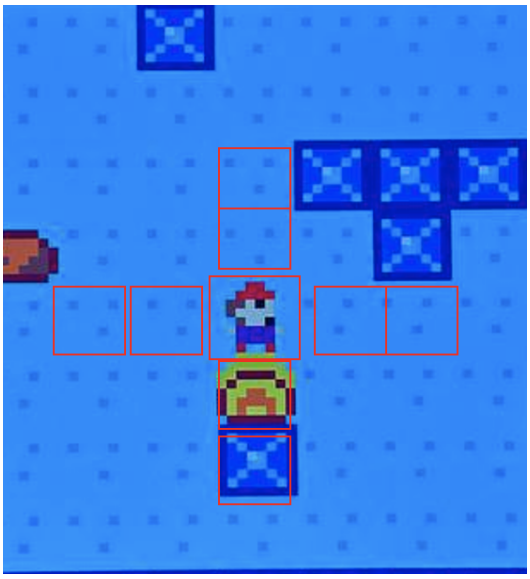


Figure. 5. Agent possible movements space

The program checks at first which button was pressed and according to the pressed button collects the grid index of the next 2 blocks.

Here is the example of the pressed button logic for the button - Up

```
        if (bu)
```

```
            begin
                addra <= obj_x + (obj_y-1)*20; //address of the next cell
                addrb <= obj_x + (obj_y-2)*20; //address of the next cell+1
```

The next step is to identify whether the address of the block corresponds to the location of mushrooms, the appropriate conditions are raised for the next part of the code to be executed.

```
                if (addra == obj_x1 + (obj_y1)*20 )//check for mushroom 1
                    cond = 1;
                else if (addra == obj_x2 + (obj_y2)*20 )//check for mushroom2
                    cond = 2;
                else
                    cond = 0;
                if (addrb == obj_x1 + (obj_y1)*20 || addrb == obj_x2 +
(obj_y2)*20 )
                    cond1 = 1;
                else
                    cond1 = 0;
            end
```

The next procedure is to validate if the agent can pass next cells. The cells have to be either floor or the goal for the agent to freely pass.

```
if (bu)
    begin
            if ((douta == floor || douta == goal ) && cond == 0 ) // Check for
the empty space and goal
            begin
                obj_y <= obj_y-1;// Move the agent up
            end
            else if ( cond != 0 && cond1 != 1) //check conditions
                    if (doutb == floor || doutb == goal) //check 2nd cell
                    begin
                        if (cond == 1 ) begin
                                obj_y <= obj_y-1;// move the agent up
                                obj_y1 <= obj_y1-1;// move the mushroom 1 up
                            end else if (cond == 2) begin
                                obj_y <= obj_y-1;// move the agent up
```

```
                                obj_y2 <= obj_y2-1;// move the mushroom 2 up
                            end
                        end
    end
```

Once the mushrooms are moved to the designated areas the game is finished and the RGB led is changed from red to green, fig. 6., this feature exploits the usage of extra I/O of the board.

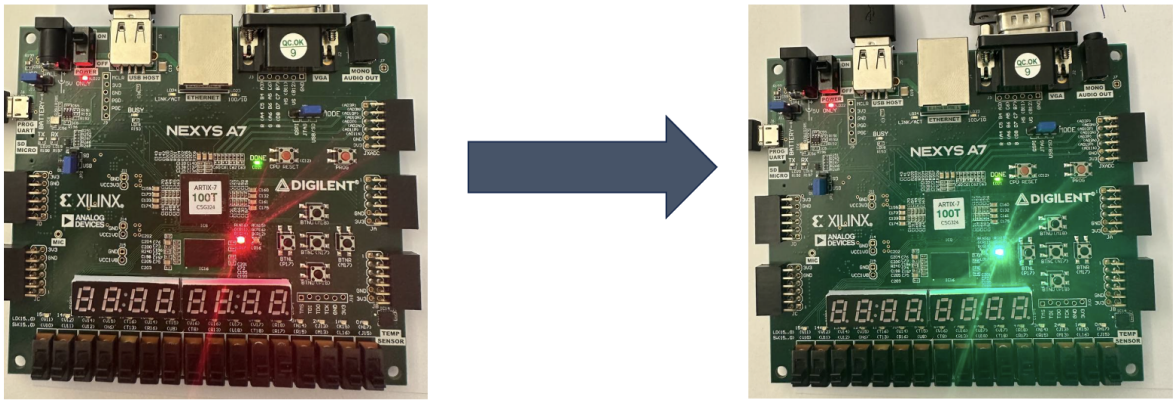


Figure 6. Board RGB Led changed from the red to green

The final game over state of the game is captured at the fig. 7

Figure 7. The Game over state of the game

Testing description

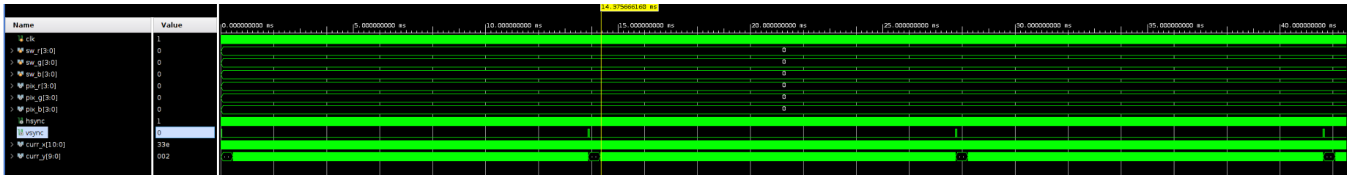
The testing part of the project was performed at most by the validation of the bitstream files at the board.

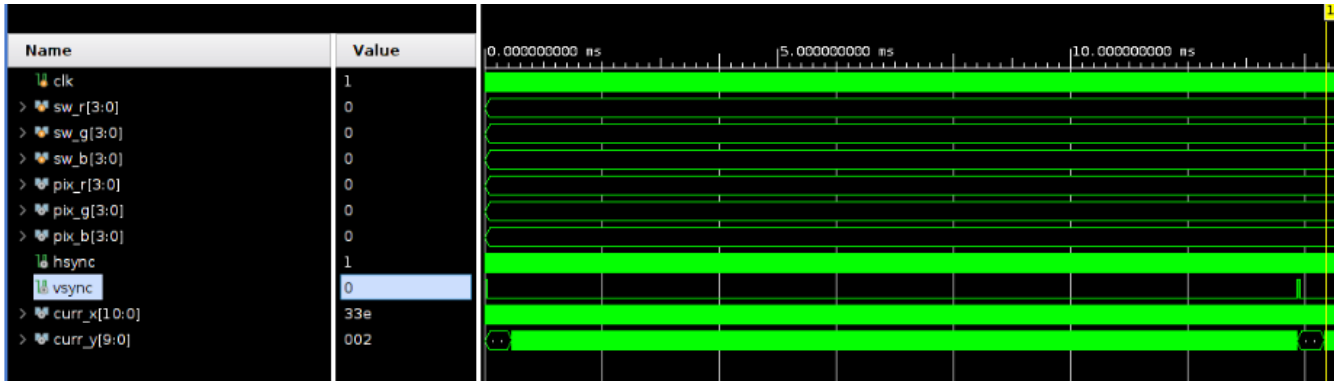
For the modules **vga_out** and **drawcon** the testbenches are provided in the Appendix of this report, as well as an attachment.

vga_out has the set of the parameters like follow

```
// Test scenario
  initial begin
      sw_r = 4'b0000; // Set switch inputs to desired values
      sw_g = 4'b0000;
      sw_b = 4'b0000;
```

The diagram for the testbench of the the vga_out module.





As it can be seen the data of the vsync has a pic values for the notadressable video of the display and at the same time the values of the curr_y is low.

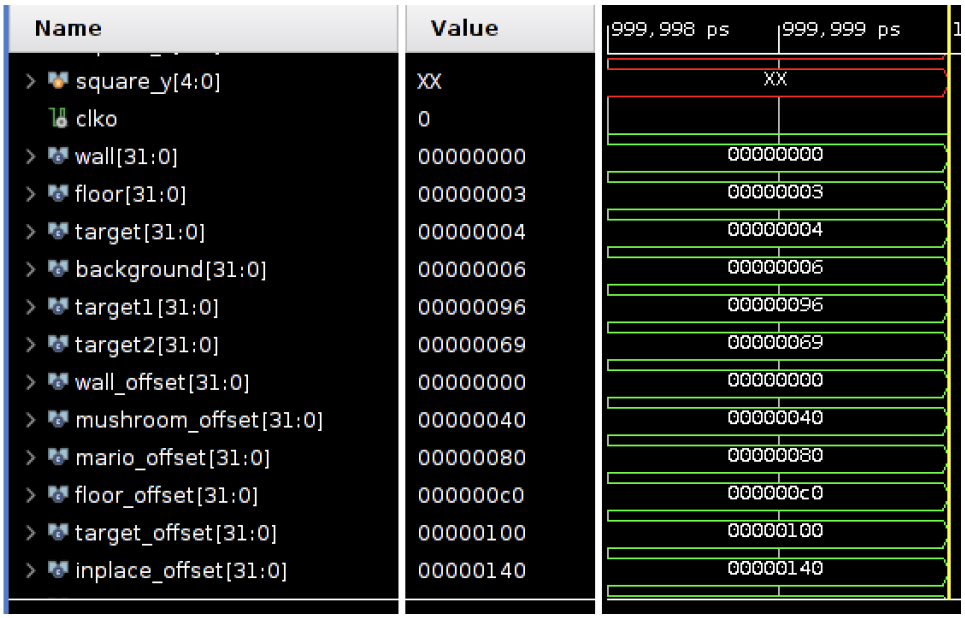
The performed test of the vga_poy completely satisfies the designated criteria for the module. It can be concluded that both module vga_out and the testbench for it has been setted up correctly. The output of the drawcon testbench has the following results which is completely satisfies the design criteria.

drawcon

For the drawcon module the test has shown the following results

Objects | Protocol Instances

Name	Value	Data Ty...
square_y[4:0]	XX	Array
clko	0	Logic
wall[31:0]	0	Array
floor[31:0]	3	Array
target[31:0]	4	Array
background[31:0]	6	Array
target1[31:0]	150	Array
target2[31:0]	105	Array
wall_offset[31:0]	0	Array
mushroom_offset[31:0]	64	Array
mario_offset[31:0]	128	Array
floor_offset[31:0]	192	Array
target_offset[31:0]	256	Array
inplace_offset[31:0]	320	Array
background_offset[31:0]	384	Array



Reflection section

The implemented game logic adheres to Sokoban concepts and defined rules. I enjoyed working on both the game logic development and sprite design aspects of the project. However, a notable drawback was the limited availability of physical boards per student group, with only one board dedicated to each group.

One noteworthy aspect of the game is the efficient use of memory. Memory storage is optimized based on sprite size, with each memory dedicated to a sprite storing values for the 8x8 pixel size rather than 64x64 pixels. Additionally, each sprite has only one color data memory location. I believe this approach to efficient memory management and optimization can be further extended and improved, potentially reaching NES-level efficiency.

I anticipate that the experience gained from this project will prove highly beneficial for my future research path.

References

[1] "How Computer Screens and Printers Show Images." Tekeye Computing, [Online], Available: https://tekeye.uk/computing/how-computer-screens-and-printers-show-images., Accessed on: December 7, 2023.

[2] "Prototyping with FPGAs - Final Project - Pong Game," Element14 Community. [Online]. Available:https://community.element14.com/challenges-projects/project14/digitalfever/b/blog/posts/prototyping-with-fpgas---final-project---pong-game. Accessed on: December 7, 2023.

[3] "Graphics display resolution," Wikipedia. [Online]. Available: https://en.wikipedia.org/wiki/Graphics_display_resolution.Accessed on: December 7, 2023.

Appendix

Code scripts for the modules and the testbenches

game_top

```
`timescale 1ns / 1ps

// Define a Verilog module named game_top with input and output ports
module game_top(
    input clk,          // Clock input
    input bc, bu, bd, bl, br, // Button inputs
    output [3:0] pix_r, // Output for red pixel
    output [3:0] pix_g, // Output for green pixel
    output [3:0] pix_b, // Output for blue pixel
    output hsync,       // Horizontal synchronization signal
    output vsync,       // Vertical synchronization signal
    output led1,        // Output for LED 1
    output led2         // Output for LED 2
);

    parameter black = 0; // Define the black color

    wire clk_i;
    clk_wiz_0 clk_mod (.clk_out1(clk_i), .clk_in1(clk)); // Clock module
instantiation

    wire [10:0] curr_x;    // Current X-coordinate
    wire [9:0] curr_y;     // Current Y-coordinate
    wire draw ;            // Flag for drawing
    wire [3:0] chr_red, chr_green, chr_blue; // Character color

    wire [4:0] obj_x, obj_x1, obj_x2, obj_y, obj_y1, obj_y2; // Object
positions
    wire [3:0] out_red, out_green, out_blue; // Output colors
    wire led1;             // LED 1 signal
    wire led2;             // LED 2 signal

    // VGA output module instantiation
    vga_out uut (
        .clk(clk_i),
```

```
        .sw_r(out_red),
        .sw_g(out_green),
        .sw_b(out_blue),
        .pix_r(pix_r),
        .pix_g(pix_g),
        .pix_b(pix_b),
        .hsync(hsync),
        .vsync(vsync),
        .curr_x(curr_x),
        .curr_y(curr_y)
    );

    // Draw controller module instantiation
    drawcon dr (
      .clk(clk),
      .curr_x(curr_x), .curr_y(curr_y),
      .obj_x(obj_x), .obj_x1(obj_x1), .obj_x2(obj_x2),
      .obj_y(obj_y), .obj_y1(obj_y1), .obj_y2(obj_y2),
      .carachter_overlap(draw),
      .r(chr_red),
      .g(chr_green),
      .b(chr_blue),
      .led1(led1),
      .led2(led2)
    );

    // Game logic module instantiation
    game_logic logic (
         .clk(clk),
         .bc(bc), .bu(bu), .bd(bd), .bl(bl), .br(br),
         .obj_x(obj_x), .obj_x1(obj_x1), .obj_x2(obj_x2),
         .obj_y(obj_y), .obj_y1(obj_y1), .obj_y2(obj_y2)
    );

    // Assign output colors based on the draw flag
    assign out_red = (draw == 0) ? black : chr_red;
    assign out_green = (draw == 0) ? black : chr_green ;
    assign out_blue = (draw == 0) ? black : chr_blue;

endmodule
```

drawcon

```
// Define a Verilog module named drawcon with input and output ports
module drawcon(
    input clk,              // Clock input
    input [10:0] curr_x,    // Current X-coordinate
    input [9:0] curr_y,     // Current Y-coordinate
    input [4:0] obj_x, obj_x1, obj_x2, obj_y, obj_y1, obj_y2, // Object
positions
    output reg carachter_overlap, // Output to indicate character overlap
    output reg [3:0] r, g, b,    // Output colors
    output reg led1, led2       // Output LEDs
);

    reg [7:0] x;            // X-coordinate within a square
    reg [7:0] y;            // Y-coordinate within a square
    reg game_over;          // Game over flag

    // Initialize X and Y coordinates
    initial begin
        x = 0;
        y = 0;
    end

    reg [8:0] address;       // Address for sprite memory
    wire [11:0] value;       // Value read from sprite memory

    // Sprite memory module instantiation
    blk_mem_gen_0 sprite_mem (
      .clka(clko),    // Input clock
      .addra(address), // Input address
      .douta(value)    // Output data
    );

    wire [2:0] square_id;         // Square ID from map memory
    reg [8:0] square_address ;    // Address for map memory

    // Map memory module instantiation
    blk_mem_gen_1 map_mem (
      .clka(clko),            // Input clock
      .addra(square_address), // Input address
      .douta(square_id)        // Output data
    );
```

```
    reg [4:0] square_x;     // X-coordinate of the current square
    reg [4:0] square_y;     // Y-coordinate of the current square

    // Parameters for different sprite types and offsets
    parameter wall=0, floor=3, target=4, background=6;
    parameter target1=150, target2=105;
    parameter wall_offset=0, mushroom_offset=64, mario_offset= 128,
floor_offset= 192, target_offset=256, inplace_offset=320,
background_offset=384;

    always @(posedge clko)
    begin
        // Check if the character is within the playable area
        if ( curr_y > 15 && curr_y < 783) begin
            square_x = curr_x >> 6;
            square_y = (curr_y - 15) >> 6;
            carachter_overlap = 1;
            square_address = square_x + square_y * 20;
            x = (curr_x - 64 * square_x) >> 3;
            y = ((curr_y - 15) - 64 * square_y) >> 3;

            // Check if the character is on the same square as an object
            if (square_x == obj_x && square_y == obj_y) begin
                address = x + y * 8 + mario_offset;
            end
            else if (square_x == obj_x1 && square_y == obj_y1) begin
                if (obj_x1 + 20 * obj_y1 == target2 || obj_x1 + 20 * obj_y1
== target1 )
                    address = x + y * 8 + inplace_offset;
                else
                    address = x + y * 8 + mushroom_offset;
            end
            else if (square_x == obj_x2 && square_y == obj_y2) begin
                if (obj_x2 + 20 * obj_y2 == target2 || obj_x2 + 20 * obj_y2
== target1 )
                    address = x + y * 8 + inplace_offset;
                else
                    address = x + y * 8 + mushroom_offset;
            end
            else begin
                // Read the square ID from the map memory and set the
address accordingly
```

```
                case (square_id)
                    wall: address = x + y * 8 + wall_offset;
                    floor: address = x + y * 8 + floor_offset;
                    target: address = x + y * 8 + target_offset;
                    background: address = x + y * 8 + background_offset;
                endcase
            end
        end
        else
            carachter_overlap = 0; // Character is outside the playable
area
    end

    // Assign RGB values from the value read from sprite memory
    assign {r, g, b} = value;

    // Check conditions for LED1 and LED2 based on object positions
    assign {led1, led2 } = ((obj_x1 + 20 * obj_y1 == target2 || obj_x1 + 20
* obj_y1 == target1 ) &&
                            (obj_x2 + 20 * obj_y2 == target2 || obj_x2 +
20 * obj_y2 == target1 )) ? 2'b10 : 2'b01;

    // Clock module instantiation
    clk_wiz_1 instance_name (
        .clk_out1(clko),     // Output clock
        .clk_in1(clk)        // Input clock
    );

endmodule
```

game_logic

```
// Define a Verilog module named game_logic with input and output ports
module game_logic(
        input clk,               // Clock input
        input bc, bu, bd, bl, br, // Button inputs
        output reg [4:0] obj_x, obj_x1, obj_x2, obj_y, obj_y1, obj_y2, //
Object positions
        output led                // Output LED
    );
```

```
    reg [7:0] addra;              // Address for memory A
    reg [7:0] addrb;              // Address for memory B
    wire [2:0] douta;             // Data output from memory A
    wire [2:0] doutb;             // Data output from memory B
    reg  [3:0] counter;           // Counter for clock generation

    // Memory module instantiation
    blk_mem_gen_2 map_memory (
        .clka(clk),    // Clock input for memory A
        .addra(addra), // Address input for memory A
        .douta(douta), // Data output from memory A
        .clkb(clk),    // Clock input for memory B
        .addrb(addrb), // Address input for memory B
        .doutb(doutb)  // Data output from memory B
    );

    // Initial values for object positions
    initial begin
        obj_x =  5;
        obj_y =  6;
        obj_x1 = 7;
        obj_y1 = 5;
        obj_x2 = 8;
        obj_y2 = 6;
    end

    parameter floor = 3, goal = 4; // Define constants for floor and goal

    reg [26:0] count; // 27 bits to represent 100,000,000
    reg clk_60hz;     // 60Hz clock signal

    // Clock generation for 60Hz
    always @(posedge clk) begin
        if (count == 45 ) begin // Divide the clock frequency to get 60Hz
```

```
   count <= 0;
            clk_60hz <= ~clk_60hz; // Toggle the 60Hz clock
        end
        else begin
            count <= count + 1;
        end
```

```
    end

    reg [2:0] cond;   // Condition variable
    reg cond1;        // Additional condition variable

    // Button event detection and object position calculation
    always @ (posedge clk) begin
        if (bu) begin
            addra <= obj_x + (obj_y-1)*20;
            addrb <= obj_x + (obj_y-2)*20;

            // Check if object position matches predefined conditions
            if (addra == obj_x1 + (obj_y1)*20 )
                cond = 1;
            else if (addra == obj_x2 + (obj_y2)*20 )
                cond = 2;
            else
                cond = 0;

            // Check additional condition for address B
            if (addrb == obj_x1 + (obj_y1)*20 || addrb == obj_x2 +
(obj_y2)*20 )
                cond1 = 1;
            else
                cond1 = 0;
        end

        // Similar logic for other button inputs (bd, br, bl)...

    end

    // Object movement logic based on button presses
    always @ (posedge clk_60hz) begin
        if (bu) begin
            if ((douta == floor || douta == goal ) && cond == 0 )
                obj_y <= obj_y-1;
            else if ( cond != 0 && cond1 != 1)
                if (doutb == floor || doutb == goal) begin
                    if (cond == 1 ) begin
                        obj_y <= obj_y-1;
                        obj_y1 <= obj_y1-1;
                    end
                    else if (cond == 2) begin
```

```
                        obj_y <= obj_y-1;
                        obj_y2 <= obj_y2-1;
                    end
                end
        end
        // Similar logic for other button inputs (bd, br, bl)...
    end

endmodule
```

vga_out

```
// Define a Verilog module named vga_out with input and output ports
module vga_out(
    input clk,                    // Clock input
    input [3:0] sw_r,             // Switch input for red color
    input [3:0] sw_g,             // Switch input for green color
    input [3:0] sw_b,             // Switch input for blue color
    output reg [3:0] pix_r,       // Red pixel output
    output reg [3:0] pix_g,       // Green pixel output
    output reg [3:0] pix_b,       // Blue pixel output
    output hsync,                 // Horizontal synchronization output
    output vsync,                 // Vertical synchronization output
    output reg [10:0] curr_x,     // Current X position output
    output reg [9:0] curr_y       // Current Y position output
);

// Define parameters for VGA timing
parameter HCOUNT_MAX = 1679,
          HCOUNT_INIT = 0,
          VCOUNT_MAX = 826,
          VCOUNT_INIT = 0,
          CURRX_INIT = 0,
          CURRY_INIT = 0,
          HCOUNT_1 = 336,
          HCOUNT_2 = 1615,
          VCOUNT_1 = 27,
          HSYNC_LIMIT = 135,
          VSYNC_LIMIT = 2;

// Declare internal registers for horizontal and vertical counters
```

```
reg [10:0] hcount;
reg [9:0] vcount;

// Initialize counters and current position at the start
initial begin
    hcount <= HCOUNT_INIT;
    vcount <= VCOUNT_INIT;
    curr_x <= CURRX_INIT;
    curr_y <= CURRY_INIT;
end

// VGA timing and position calculation logic
always @(posedge clk) begin
    if (vcount == VCOUNT_MAX) begin
        vcount <= VCOUNT_INIT;
        curr_y <= CURRY_INIT;
    end else begin
        if (hcount == HCOUNT_MAX) begin
            hcount <= HCOUNT_INIT;
            vcount <= vcount + 1'b1;
            if (vcount >= VCOUNT_1 && vcount <= VCOUNT_MAX) begin
                curr_y <= curr_y + 1'b1;
            end
        end else begin
            hcount <= hcount + 1'b1;
            if (hcount >= HCOUNT_1 && hcount <= HCOUNT_2) begin
                curr_x <= curr_x + 1'b1;
            end
            if (hcount == HCOUNT_2) begin
                curr_x <= CURRX_INIT;
            end
        end
    end
end

// Horizontal and vertical synchronization signals generation
assign hsync = (hcount <= HSYNC_LIMIT) ? 0 : 1;
assign vsync = (vcount <= VSYNC_LIMIT) ? 1 : 0;

// Pixel color assignment based on VGA timing
always @* begin
    pix_r <= ((hcount >= HCOUNT_1) && (hcount <= HCOUNT_2) && (vcount >=
VCOUNT_1) && (vcount <= VCOUNT_MAX)) ? sw_r : 4'b0000;
```

```
    pix_g <= ((hcount >= HCOUNT_1) && (hcount <= HCOUNT_2) && (vcount >=
VCOUNT_1) && (vcount <= VCOUNT_MAX)) ? sw_g : 4'b0000;
    pix_b <= ((hcount >= HCOUNT_1) && (hcount <= HCOUNT_2) && (vcount >=
VCOUNT_1) && (vcount <= VCOUNT_MAX)) ? sw_b : 4'b0000;
end

endmodule
```

Testbenches

vga_out_tb

```
module testbench;

    // Declare signals for the testbench
    reg clk;
    reg [3:0] sw_r, sw_g, sw_b;
    wire [3:0] pix_r, pix_g, pix_b;
    wire hsync, vsync;
    wire [10:0] curr_x;
    wire [9:0] curr_y;

    // Instantiate the module under test
    vga_out dut (
        .clk(clk),
        .sw_r(sw_r),
        .sw_g(sw_g),
        .sw_b(sw_b),
        .pix_r(pix_r),
        .pix_g(pix_g),
        .pix_b(pix_b),
        .hsync(hsync),
        .vsync(vsync),
        .curr_x(curr_x),
        .curr_y(curr_y)
    );

    // Clock generation
    initial begin
        clk = 0;
        // Provide a clock with a period of 5 time units (adjust as needed)
        forever #5 clk = ~clk;
```

```
    end

    // Test scenario
    initial begin
        sw_r = 4'b0000; // Set switch inputs to desired values
        sw_g = 4'b0000;
        sw_b = 4'b0000;

        // Release reset after a few clock cycles
        // Insert any other test sequences or scenarios here
        // Terminate simulation after a certain time (adjust as needed)
        #1000 $finish;
    end

endmodule
```

Drawcon_tb

```
module my_creature_Testbench;

// Inputs
reg [10:0] curr_x;
reg [9:0] curr_y;

// Outputs
wire carachter_overlap;
wire [3:0] r, g, b;

// Instantiate the my_creature module
drawcon uut (
    .curr_x(curr_x),
    .curr_y(curr_y),
    .carachter_overlap(carachter_overlap),
    .r(r),
    .g(g),
    .b(b)
);
integer i;
// Stimulus
initial begin
    // Initialize inputs
```

```
    curr_x = 0;
    curr_y = 0;

    // Provide stimulus for curr_x from 0 to 200


    for (i = 0; i <= 200; i = i + 1) begin
        curr_x = i;
        $display("curr_x: %d, carachter_overlap: %b, r: %b, g: %b, b: %b",
curr_x, carachter_overlap, r, g, b);
        #10; // Simulate for some time after each value change
    end

    // End simulation
    #1000 $finish;
end

endmodule
```