6306ENG · Design of Real-Time Systems
Project B: Arcade Game

Joseph Simeon, s2966176
joseph.simeon@griffithuni.edu.au

1 Project outline and requirements

The purpose of the project is to create a 1980's style arcade game utilising a Field Programmable Gate Array (FPGA) Cyclone V with the hardware designed using Intel Quartus Prime software and soft-processor designed in Nios II Processor Software Build Tools (SBT) [integrated with Eclipse]. The requirements for the project include:

- The video game must follow all the rules of the game and show a clear resemblance to its 1908's counterpart. Although it may be customised.
- The video game must connect to an external monitor at a minimum resolution of 800x600 with a minimum colour range of 64 colours. Note that higher values can be used.
- The player controls must be implemented via a PS2 keyboard and play simple sounds.
- The seven-segment display should be used to display use messages to the user as example score, number of lives, energy etc. What is displayed depends upon the game chosen.

The game chosen for this project is the 80's game Tetris created by Alexey Pajitnov during the Soviet Union, while minimum objective of this project is to create a basic Tetris game the extended objective is to customise the game into a variation of Tetris while the game itself has an official guideline as of 2009 issues by developer Blue Planet Software [1]:

- Playfield is 10x40, where 20 are hidden or obstructed by the field (2002 guideline, it could be at least 22 height)
    - If hardware permits, pixel of row 21 will be visible
- Tetris must consist of tetrominoes, geometric shapes composed of four squares connected orthogonally.
- Tetromino shapes have associate colours.
    - While monochrome screens or palettes, tetromino should have distinct hues and patterns.
- Tetromino have start locations depend on which shape it is.
- Shapes have initial rotation and movements dictated by the rotation system.
- Tetris has a standard mapping for computer keyboards to be followed.
- Sounds effects on rotation, movement, landing on surface, touching a wall, locking, line clear and game over.
- Game must have a ghost piece function.
- Designated soft drop and hard drop.
- Tetris uses a song called 'Korobeiniki', which is the default song.
- Use of a half second lock delay.
- Arcade variations must have 15 moves/rotations before lock.

While the game may have a guideline, it will be followed as much as possible however unless limited by hardware and time which will be discussed through-out the report.

2 System layout

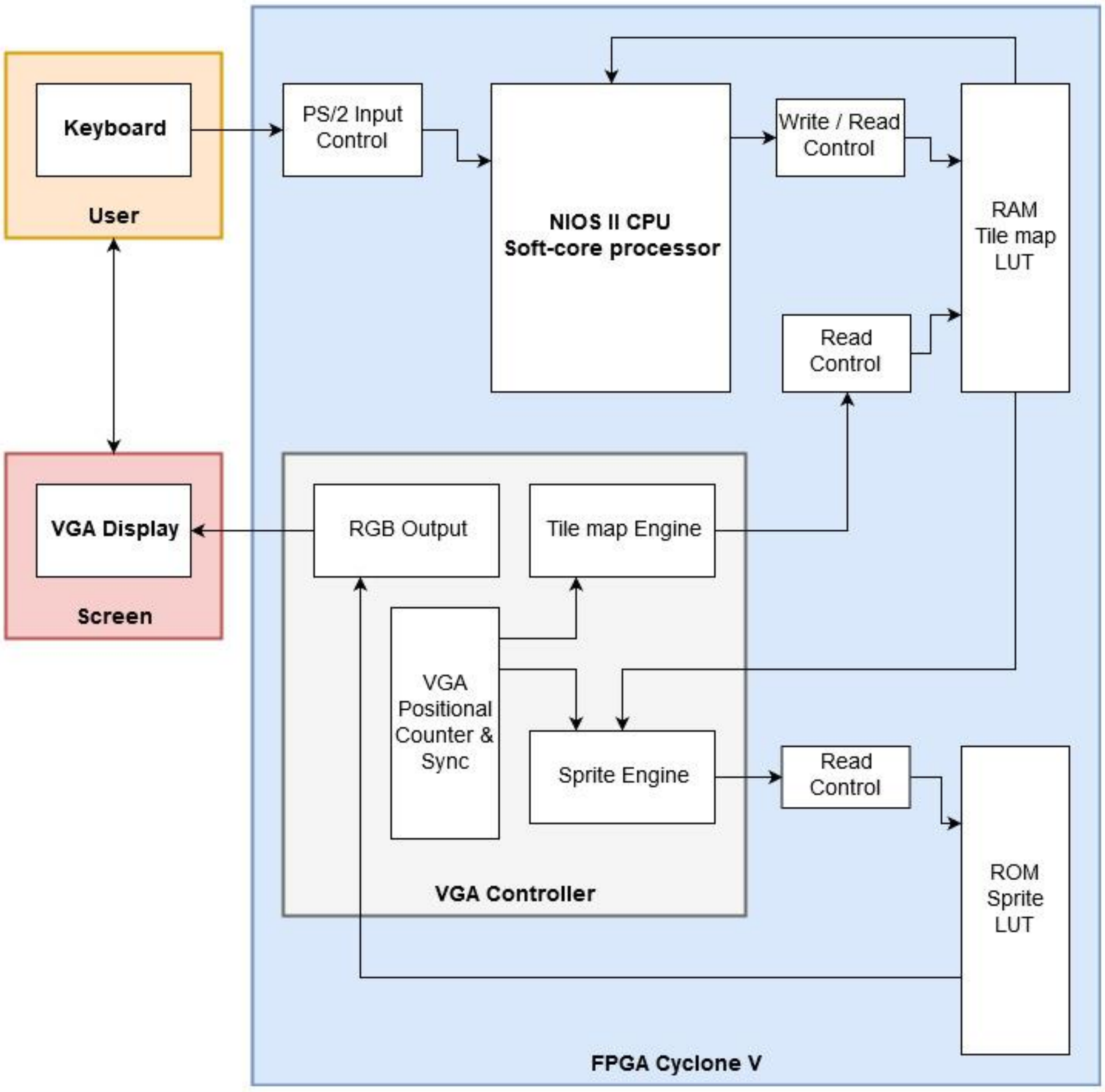


Figure 1: System structure and communication process of the project.

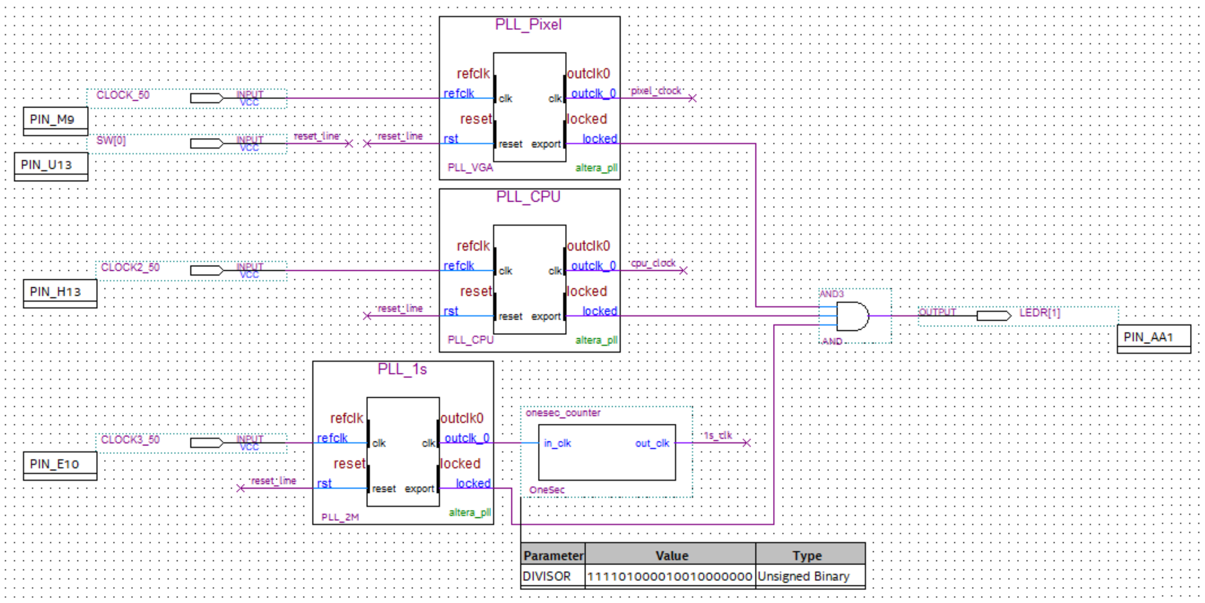
Figure 1 shows the communication process from the user's operation via a keyboard to the FPGA where the keyboard inputs will be processed by the FPGA's soft-core processor (designed via NIOS II Processor platform builder), the processed input will be written to the stored Tile map look-up-table (LUT) within the RAM using read & write control where the VGA Controller using 'Tile map engine' will be reading the tile map from positional counter generated by the controller to send the tile number and positional data to the 'Sprite engine' which will match the tile number to the sprite and positional data to which part of the sprite in the ROM Sprite look-up-table (LUT) which is stored as a Red-Green-Blue (RGB) value which will all be displayed via VGA to a screen, the report will be breaking down the system structure into sections that consist of relational parts such as the PLL clocks, VGA controller and memory interaction, Stored data within the RAM & ROM, PS/2 input control, and Game engine (NIOS II CPU) and memory interaction.

3 PLL clocks

Clock speeds used in the FPGA Cyclone V is at 50 MHz, for the project the clock speeds used are outputted phase-locked loops (PLL) specifically the Video Graphics Array (VGA) utilises a specific frequency relating to the active screen display at a the screen's frequency (640x480, 60 Hz) which can be viewed in Table 1 under pixel clock at 25.175 MHz [2] while the approach for the soft-core processor was to input a clock speed that as high as possible still in stabled at around 100 MHz and since the CPU needs a one second triggering for a clock a PLL is used to generate a lower frequency which is fed into a counter module to generate a one second pulse, the Verilog code can be found in Appendix under onesec_counter module.

Table 1: VGA information associated with the format of 640x480 at 60Hz monitor.

Format	Pixel Clock (MHz)	Horizontal (in Pixels)				Vertical (in Lines)			
		Active Video	Front Porch	Sync Pulse	Back Porch	Active Video	Front Porch	Sync Pulse	Back Porch
640x480, 60Hz	25.175	640	16	96	48	480	10	2	33



Parameter	Value	Type
DIVISOR	11110100001001000000	Unsigned Binary

Figure 2: Schematic view of the PLL for the pixel clock, CPU clock and one second generator.

Table 2: Truth table for 3 inputs of the PLL with an LED output.

PLL_Pixel	PLL_CPU	PLL_1s	LEDR[1]
OFF	OFF	OFF	OFF
ON	OFF	OFF	OFF
OFF	ON	OFF	OFF
OFF	OFF	ON	OFF
ON	ON	OFF	OFF
OFF	ON	ON	OFF
ON	OFF	ON	OFF
ON	ON	ON	ON

Figure 2 shows the block diagram of the two PLL for the pixel clock that will output 25.175 MHz from a reference clock of 50 MHz and a CPU clock of 100 MHz also using a reference clock of 50 MHz, both export outputs are connected to an AND gate that outputs to the LED at position one on the FPGA. **Error! Reference source not found.** shows the truth table of the AND with the output of the LED in relation to each PLL working (on or off), this is a test point with the LED being on as an indicator that both PLL are working and the LED being off as an indicator that either one PLL is working or both are not.

4 VGA controller and memory interaction

This section will discuss the video graphics array (VGA) connection to the FPGA and the controlling module associated with VGA as it communicates with the RAM and ROM that stores the tile map and sprite map for the video game.

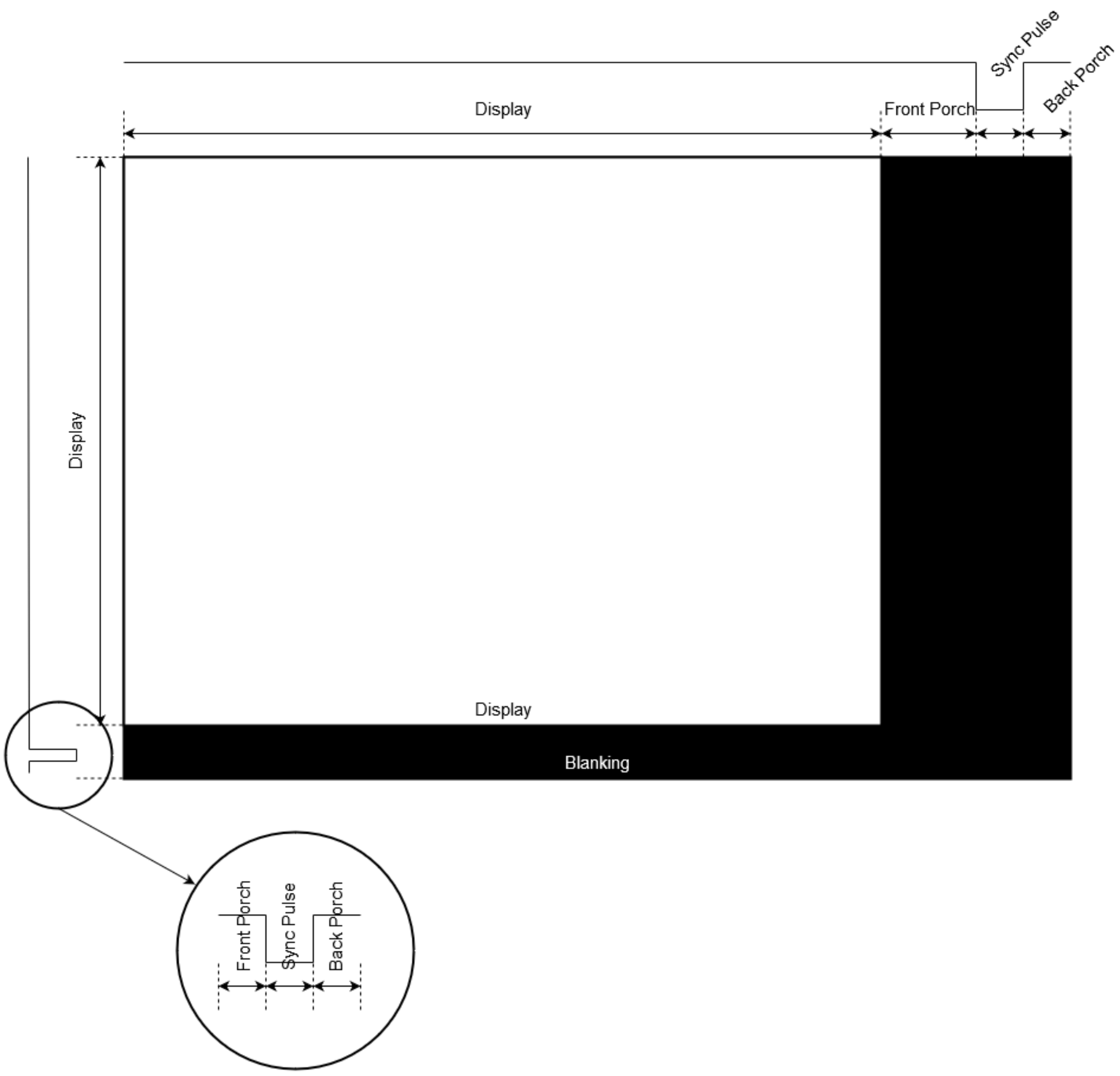


Figure 3: VGA display and blanking section.

VGA is an analogue video standard which doesn't require high clock speeds thus the FPGA project will be using VGA connection as the source of displaying the video game, Figure 3 shows the display and blanking areas which are determined by the position movement, the blanking area is due to the development of VGA and CRT monitors needing time for the screen to stabilise and for the electron gun to return to that starting line position with the horizontal sync denoting a line and the vertical sync denoting a frame. Table 1 from PLL clocks shows the needing frequency of the pixel clock as well as active screen area, porch and sync values related to Figure 3.

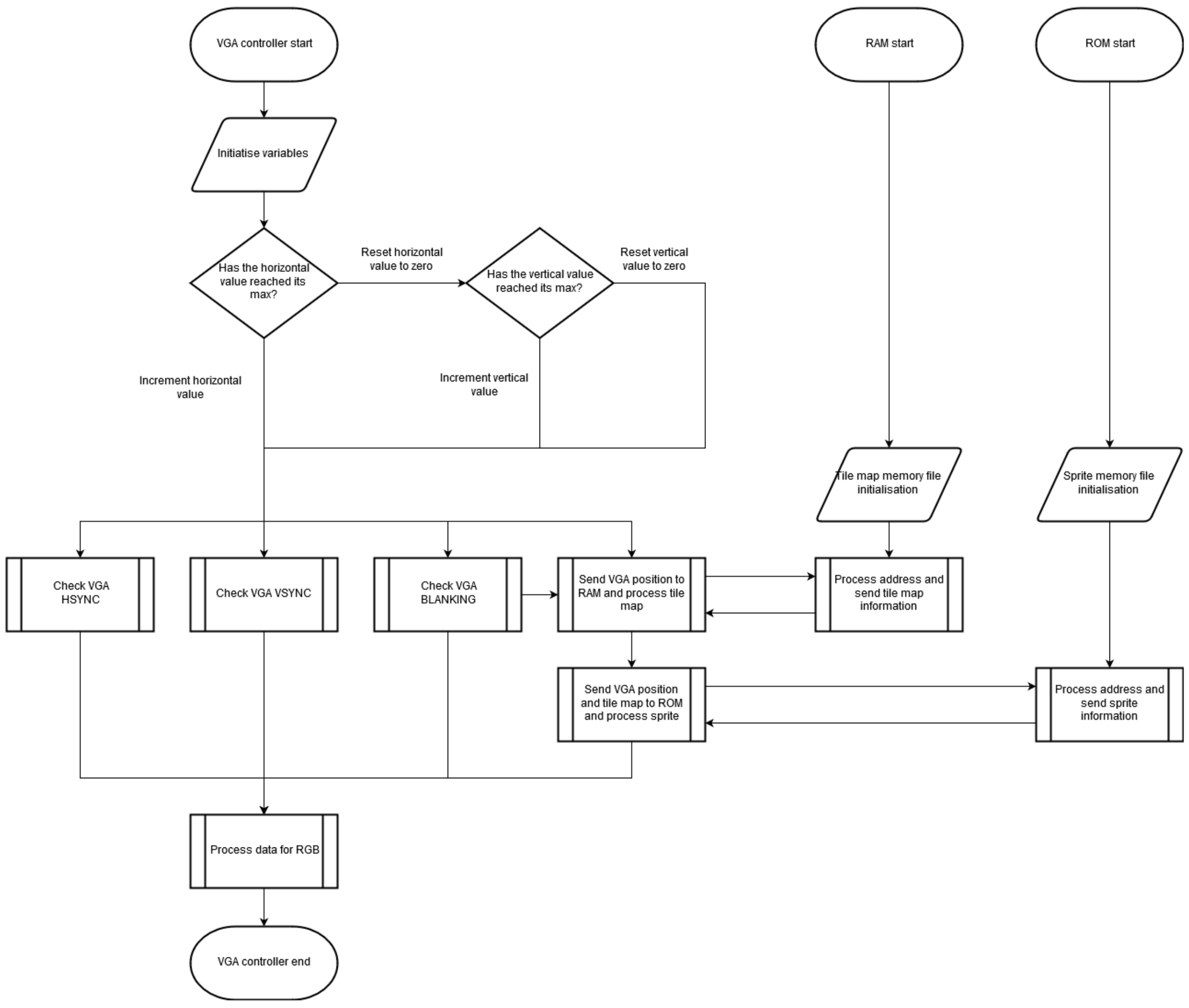

VGA controller start
Initiatise variables
Has the horizontal value reached its max?
Reset horizontal value to zero
Has the vertical value reached its max?
Reset vertical value to zero
Increment horizontal value
Increment vertical value
RAM start
ROM start
Tile map memory file initialisation
Sprite memory file initialisation
Check VGA HSYNC
Check VGA VSYNC
Check VGA BLANKING
Send VGA position to RAM and process tile map
Process address and send tile map information
Send VGA position and tile map to ROM and process sprite
Process address and send sprite information
Process data for RGB
VGA controller end

Figure 4: Flow chart of VGA controller module as it processes increments through screen location and transform data to addressing to retrieve information from RAM and ROM memory that will be sent via RGB after checks.

Figure 4 shows a flow chart for the VGA controller module alongside the ROM and RAM which will hold the memory files for the sprite and tile map data respectively which the VGA will access to print the appropriate sprite RGB pixel in the appropriate position of the screen according to the tile map layout that is read from the RAM, the RAM also writeable from the CPU but that will be discussed later in the report while the ROM is readable only for the purpose of the RGB pixel values that make up each sprite used in the video game. The VGA controller will move through the positioning of the screen horizontally and vertically until each frame is completed in which it will start all over again sending the RGB value to each pixel within the display view, the controller will assess whether VGA is within the horizontal sync (HSYNC), vertical sync (VSYNC), blanking area (When to send the RGB data) and retrieval of the sprite number according to the position of the screen and what the RGB value is according to the position of the sprite, these values are held in the RAM and ROM respectively initialised with memory files known as MIF (Memory Initialization File).

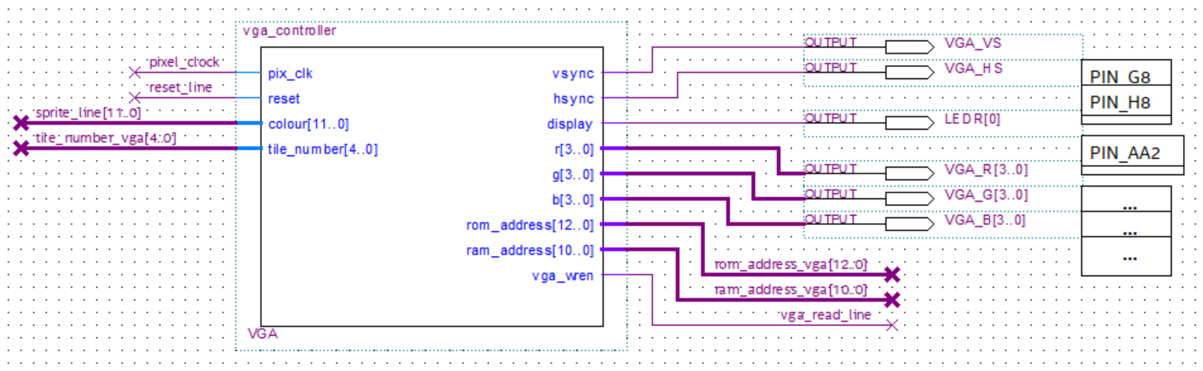


Figure 5: VGA controller schematic.

Figure 5 is the block diagram of the VGA controller runs on the pixel clock of 25.175 MHz and processes the horizontal and vertical position of each pixel across the frame which is used in the communication of the syncing of the horizontal and vertical, active display flags, and handling the process to retrieved the tile number from the RAM's tile map look-up-table (LUT) which will be used to transfer the correct colouring for the RGB output via the sprite number determined by the tile number and the position of the pixel relating the to position within the sprite by modulus of the horizontal and vertical position. The LED output for the VGA controller is a testing point as the LED will be on during the active display section and off during the blanking section shown in Figure 3 but due the frequency of the displaying of each frame the LED will be visually seen as dimmed rather than blinking. The code for the VGA module can be view in Appendix under vga_controller module.

4.1 VGA signals

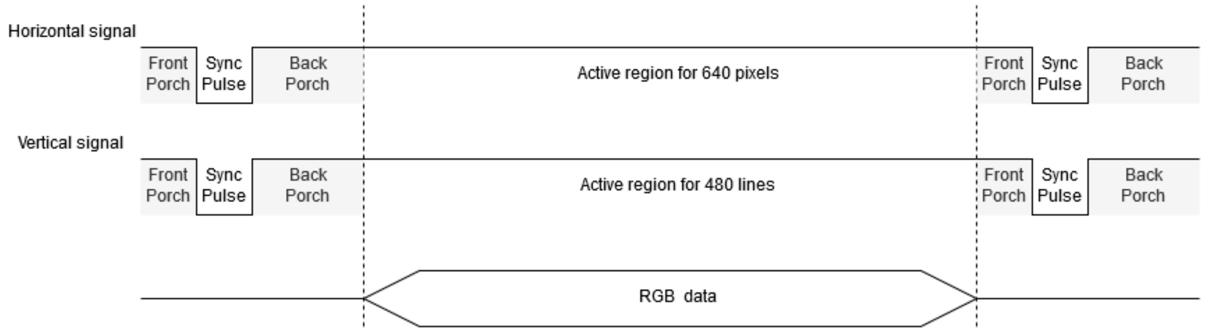


Figure 6: VGA signal representation.

4.2 VGA algorithms

The position data used within the VGA controller utilises a horizontal and vertical counter which moves through the horizontal value from left to right before moving between each vertical value top to bottom.

Equation 1: RAM address algorithm for the tile engine.

$$RAM\ address = \left(\frac{vertical\ value}{width\ of\ sprite} * \#of\ horizontal\ tiles\right) + \left(\frac{horizontal\ value}{length\ of\ sprite}\right)$$

Equation 2: ROM address algorithm for the sprite engine.

$$\begin{aligned} ROM\ address = {} & (tile\ number * length\ of\ sprite * width\ of\ sprite) \\ & + \big((vertical\ value \,\%\, width\ of\ sprite) * width\ of\ sprite\big) \\ & + (horizontal\ value \,\%\, length\ of\ sprite) \end{aligned}$$

5 Stored data in the RAM & ROM

The project utilises two main types of memory, Random-Access-Memory (RAM) that will be used to store read and write-able data and Read-Only-Memory (ROM) that will be used to store read-only data. The project's tile-map is read and write-able data and holds the number that relates to a sprite number which is stored in the RAM and the sprite is a 16 by 16 pixel image thus the data for the sprite does not need to be write-able which ROM is the best solution in storing the data; both storage methods for the tile map and sprites are in Memory-Initialization-File (MIF), the generation of the MIF files were processed through GNU Octave and both generating code can be found in Appendix under tilemap_mif_generator and sprite_mif_generator.

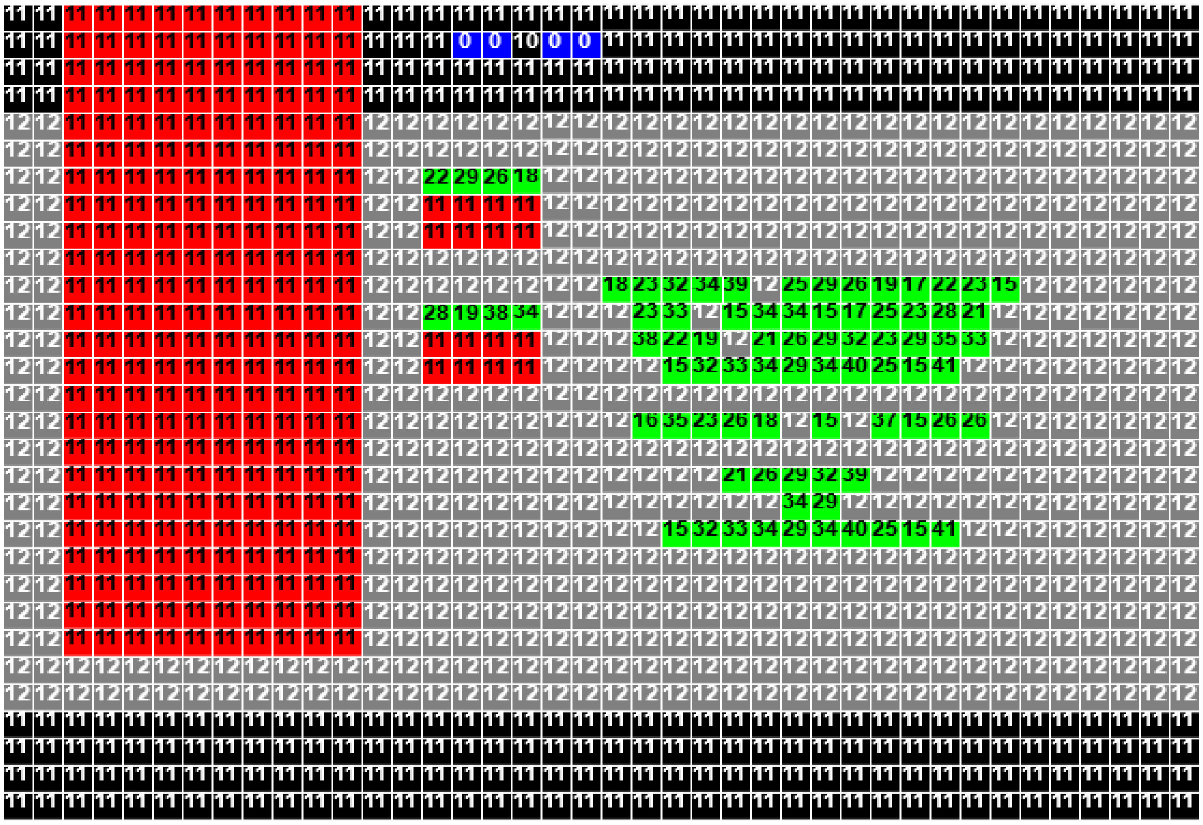


Figure 7: Visual representation of the tile-map.

Figure 7 is a visual representation of the tile map with numbers associated with the sprites, the tile map represents the screen that the operator will see which is sectioned into tiles by length and width of a sprite. Within Figure 7 most of the map is static with the highlighted red and blue tiles indicating that those tiles will be written over, the red tiles is the play area in which the tetromino blocks will be active for the user to interact with while the blue tiles is the clock and will change numerically in decimal as time increases naturally through state of play and the green highlighted relating to the alphabet.

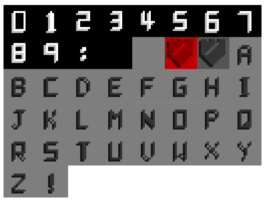


Figure 8: Sprites associated with the tile map.

Figure 8 shows the 16x16 pixel sprites used within the project and each sprite relates to the number associated with the tile map, the tile map data is a value between 0 to 41, and that relates to the sprite data as each sprite is takes up 256 pixels which is an individual address place in the memory file while each address holds a 12-bit value which the VGA controller will break into three 4-bit values each will be transmitted as RGB data line seen in Figure 6 with each sprite is off-set by 256.

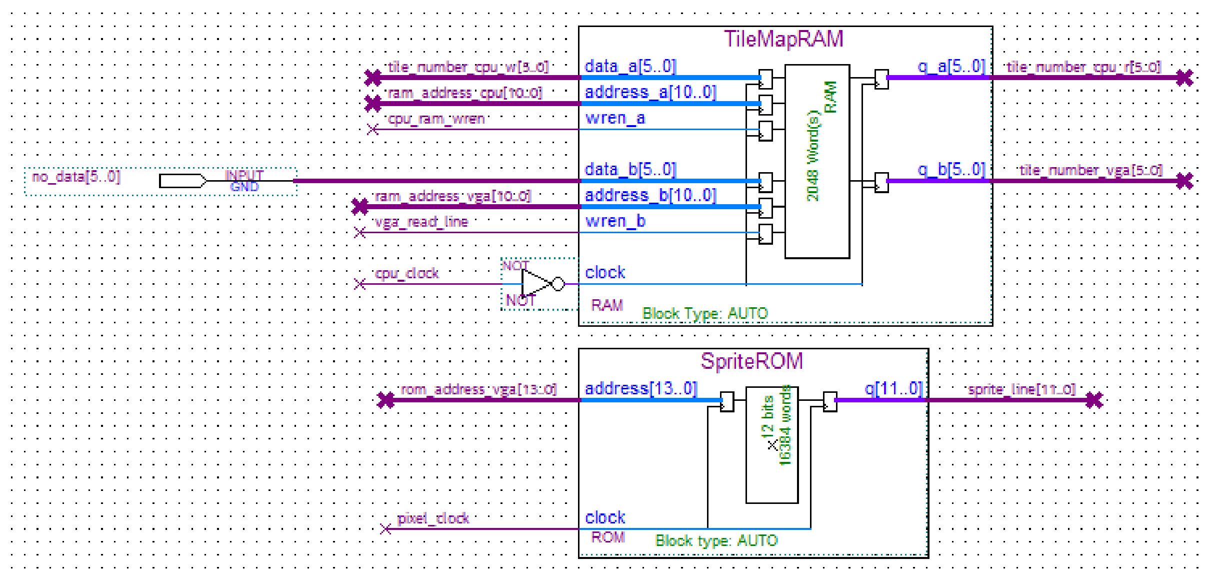


Figure 9: Schematic of the RAM and ROM.

Figure 9 shows the block diagram of the RAM and ROM; the project uses a single port ROM accessed by the VGA controller to retrieve the data that will be used displaying the RGB value and dual-port RAM accessed by both the CPU and VGA in retrieving the tile number from the mapped data however the VGA is read-only as the write enable line is set to read-only and write data line is linked to an input that is connected to GND and cannot be accessed by the user while the CPU will read and write to the RAM via the write data line and write enable line will be either write or read. The RAM and ROM use different clock speeds with the ROM utilising the pixel clock (25.175 MHz) and RAM utilising the CPU clock (100 MHz) as well as the RAM using a not gate to create a phase shift allowing the pixel data to stabilise properly to the screen.

5.1 ROM and RAM algorithms

Equation 3: Number of tiles within a line.

$$\#\ of\ tiles\ length = \frac{length\ of\ display}{sprite\ length}$$

Equation 4: Number of tile lines.

$$\#\ of\ tile\ lines = \frac{width\ of\ display}{sprite\ width}$$

Equation 5: RAM address size.

$$RAM\ address\ size = \#\ of\ tiles\ length * \#\ of\ tile\ lines$$

Equation 6: Accessing the data of the RAM.

$$Access\ RAM\ data = \left(\left(\frac{vertical\ position}{sprite\ width}\right) * \#\ of\ tiles\ length\right) + \left(\frac{horizontal\ position}{sprite\ lentgh}\right)$$

Equation 7: Sprite size.

$$Sprite\ size = sprite\ length * sprite\ width$$

Equation 8: ROM address size.

$$ROM\ address\ size = sprite\ size * \#\ of\ sprites$$

Equation 9: Accessing the data of the ROM.

$$\begin{aligned}\mathit{Access\ ROM\ data} &= (\mathit{tile\ number} * \mathit{sprite\ size}) \\ &+ \big((\mathit{vertical\ position}\ \%\ \mathit{sprite\ width}) * \mathit{sprite\ width}\big) \\ &+ (\mathit{horizontal\ position}\ \%\ \mathit{sprite\ length})\end{aligned}$$

6 PS/2 input control

Table 3: Button controls used in the modern Tetris.

Control	Button
Move left	Left-arrow
Move right	Right-arrow
Rotate Clockwise	Up-arrow
Drop: Soft	Down-arrow
Drop: Hard	Space bar
Hold piece	C

The project uses a PS/2 keyboard input to control the tetromino with the modern control scheme for Tetris shown in Table 3 while the user will be given the options to move the tetromino left and right only within the play area as well as given the option to rotate the tetromino clockwise but not counter clockwise and the ability to drop the piece at a faster fall rate referred to as a soft drop, the early versions of Tetris did not include the ability hold a piece until the user is ready to use and a different drop type known as the hard drop which instead of a piece fall at a faster rate it instantly drops to the bottom of the well.

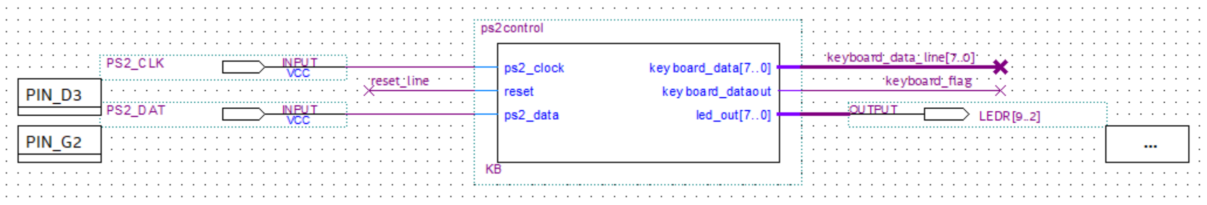


Figure 10: Schematic of the PS/2 keyboard module.

Figure 10 shows the schematic of the PS/2 control module that will take on the inputs from the PS/2 clock and PS/2 data line, the PS/2 control module will filter out the appropriate code relating to the keyboard buttons seen in Figure 11, the filter data will be sent to the CPU along with a notification flag which will indicate to the CPU to receive a keyboard command. The LED output allows to visually examine which keyboard buttons are pressed as a testing method, the code for the PS2 control module can be found in Appendix under ps2control module.

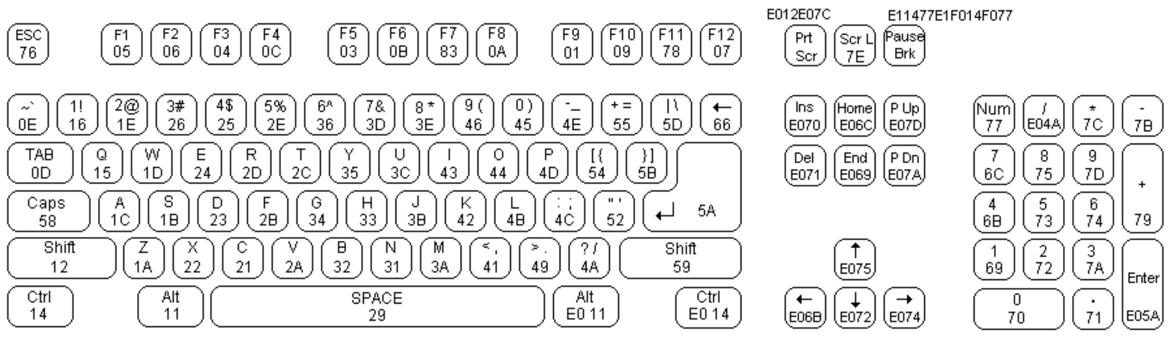


Figure 11: Keyboard layout with PS/2 values, image sourced from [3].

6.1 PS/2 keyboard signal

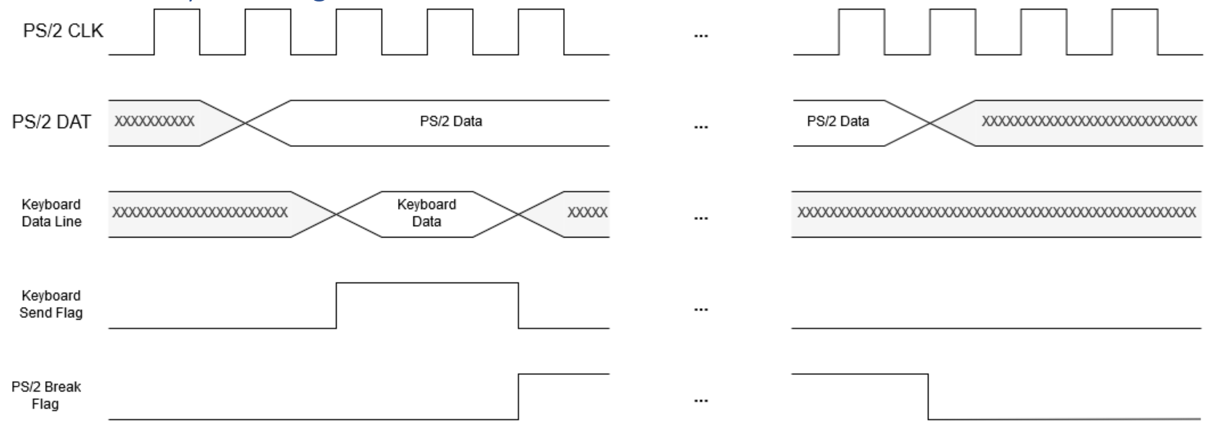


Figure 12: Signal of the PS2 control module.

7 Game engine (Nios II Soft Processor) and memory interaction

The central processing unit (CPU) used within the project is a soft processor created using the Nios II Soft Processor Platform Designer. The CPU is the brain of the project which is the game engine as it will be processing the user inputs from the keyboard and depending on the button presses actions will be taken regarding block movement and telegraphed to the VGA display through the VGA controller by writing the changed information to the tile map LUT.

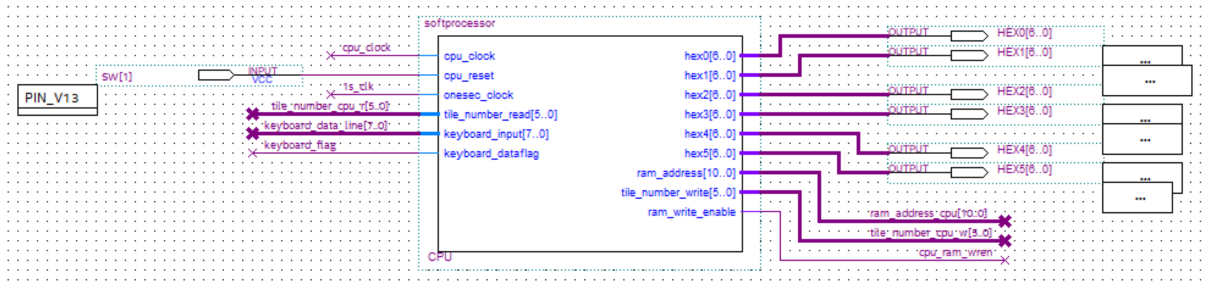


Figure 13: Schematic of the CPU designed in the NIOS II Processor platform builder.

Figure 13 shows the block diagram of the CPU for the project, the inputs involved are the CPU clock, one second clock trigger, reading of the tile map LUT data, the data for the PS2 keyboard, flag for when the keyboard data is sent, and a reset connected to a switch which requires manual flicking of on and off for the CPU to start or another method is use a ground connected input that will automatically trigger the CPU's software to be connected to the hardware. Outputs for the CPU utilise all of the hex seven segment displays for the hidden scoring system, and three outputs relating to the RAM writing with the address, data to be written and a write enable line (if the write enable line is 1 then the CPU is ready to write and if the it is 0 then the CPU is ready to read).

8 Software design

The game setting for the software is based on Lucas Pope's "Papers, Please"; in terms of naming convention for the for fictional location and history between fictional places thus the game setting is '*Fictional Eastern Europe country Arstotzka is being attacked by the even more fictional Kolechia, their fabled enemy. You must save Arstotzka by building a wall however be aware that Kolechia will destroy your blocks, fill the rows to set the blocks and save the glorious state of Arstotzka*'. For the game setting to succeed in code the blocks will be generated at a certain sprite and as the rows are filled the sprite will change to a different hue of the same sprite to show that the block has been 'set'. After a certain timeframe the fictional enemy will 'attack' and all unset blocks will be destroyed within the well for the process to begin all over again until the game is finished the same way Tetris is finished until the generation block cannot fall because the well is filled even if the well has not been completely filled. At finishing the game an internal score is generated on the hex output depending on how long the game occurred for, the amount of set, unset and blank tile positions within the well and whether the fictional enemy invaded the country.

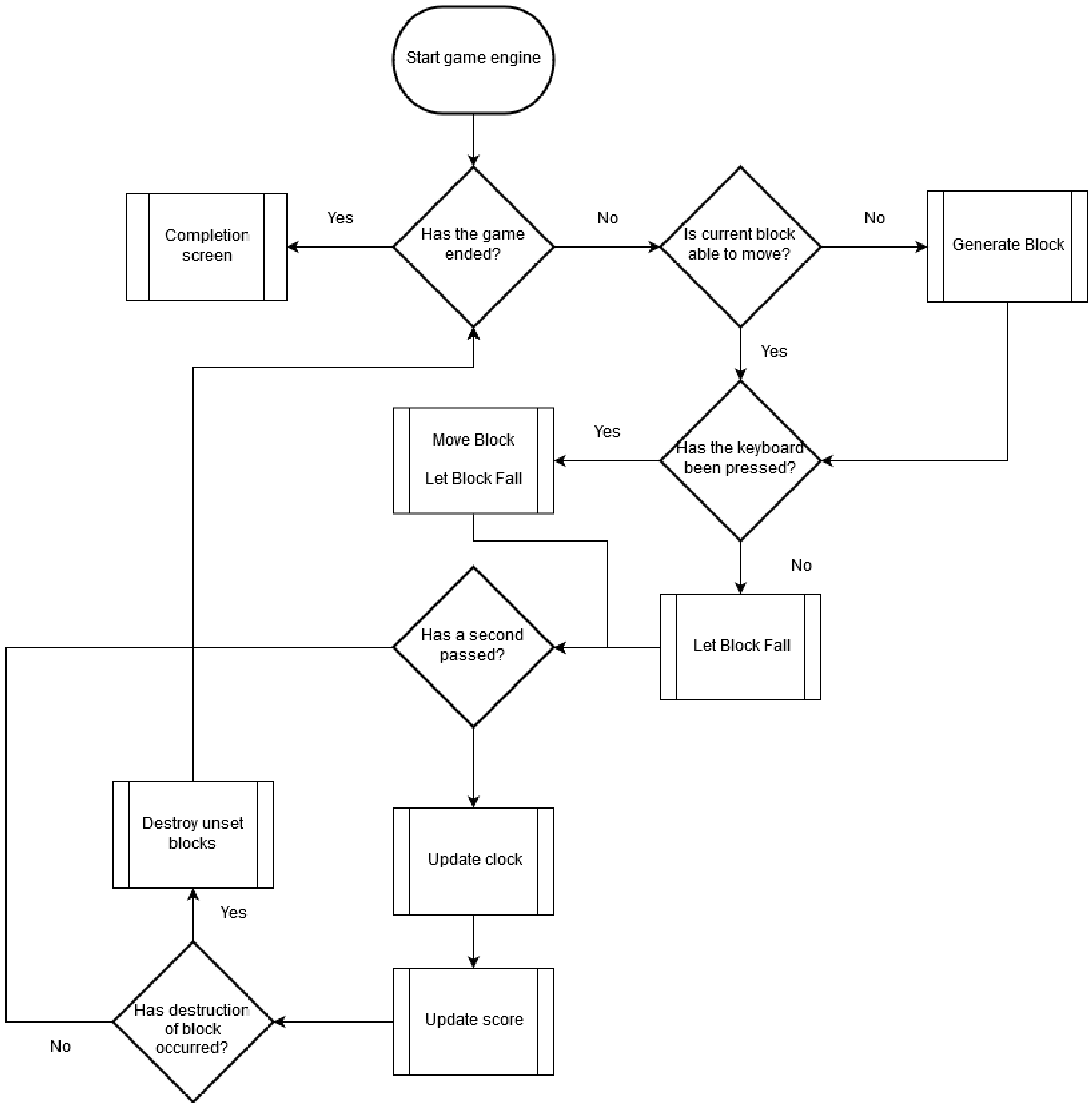
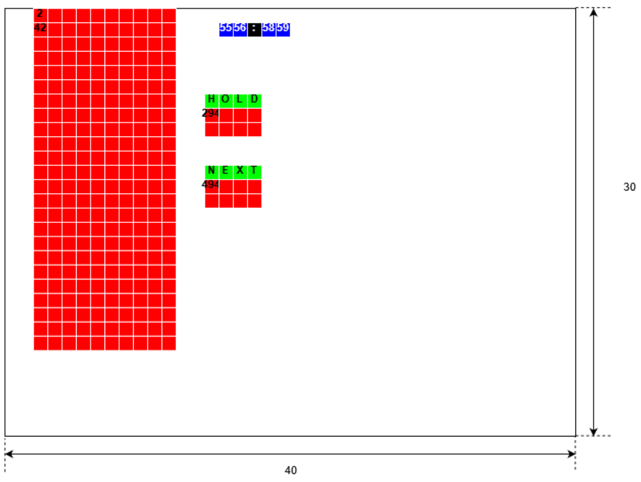
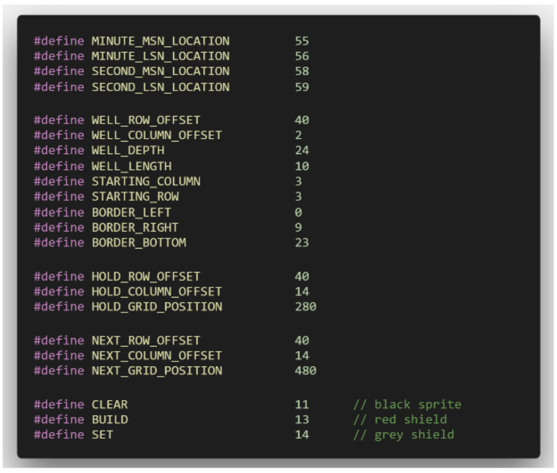


Figure 14: Software design flowchart.

There a four designated areas in which the CPU will need to be attentive too, Figure 7 shows the tile map LUT in a visual design and within that figure the areas highlighted in red and blue are the attentive areas with the red zones relating to the block (generation and movement) and the blue zones relating to the real-time clock that the user will observe incremental. These areas determine that the CPU does not need to read in the entire tile map LUT but because these areas are always the same an array variables can be used to hold the data and depending the position of that array along with off-sets can added to reflect the position of the same tile on the LUT limiting the amount of times the CPU has to communicate to the RAM while the sprites used in the red zones only need to be cleared, built and set by three different sprites and the blue zones reflect a clock which will be utilising 10 different sprites from numbers zero to nine seen in Figure 8, since the sprite number reflects the number represented it will be easier writing the correct sprite number to the tile map LUT because sprite zero is the number zero and sprite nine is the number nine which is also helpful that C-language number system starts at zero. The clock uses four different variables related to the most significant and least significant numbers associated with seconds and minutes that will check that each variable reaches its maximum number and increases the next variables and resets the current variable.




```
#define MINUTE_MSN_LOCATION        55
#define MINUTE_LSN_LOCATION        56
#define SECOND_MSN_LOCATION        58
#define SECOND_LSN_LOCATION        59

#define WELL_ROW_OFFSET            40
#define WELL_COLUMN_OFFSET         2
#define WELL_DEPTH                 24
#define WELL_LENGTH                10
#define STARTING_COLUMN            3
#define STARTING_ROW               3
#define BORDER_LEFT                0
#define BORDER_RIGHT               9
#define BORDER_BOTTOM              23

#define HOLD_ROW_OFFSET            40
#define HOLD_COLUMN_OFFSET         14
#define HOLD_GRID_POSITION         280

#define NEXT_ROW_OFFSET            40
#define NEXT_COLUMN_OFFSET         14
#define NEXT_GRID_POSITION         480

#define CLEAR                      11      // black sprite
#define BUILD                      13      // red shield
#define SET                        14      // grey shield
```

Figure 15: Attentive areas of the tile map and the deconstructed positions in terms of off sets.

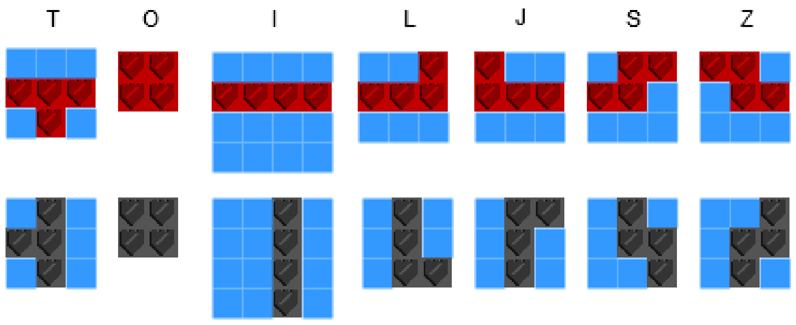


Figure 16: Red shield generation of tetromino at starting rotation and grey shield set block at first clockwise rotation.

Figure 16 shows the blocks generated in terms of the red shield and the set version of the block as the grey shield with the first clockwise rotation with the light blue blocks are the checks involved to successfully rotate each block. The blocks will be randomly generated above the well and will fall down due to gravity which in this project is at a arbitrary number of 55 milliseconds as it is approximately half a second which is the time that will be the flagging system to allow the user to move and update the well to the VGA display.

As the clock updates, the real-time clock will be visible on the VGA display and internally a random timer is waiting to finally occur between 15 to 45 seconds all the unset blocks will be destroyed increasing difficulty in completing the game as well as the longer the time increases to finish the more decreasing of points due to time taken. The code for the software design can be found in the Appendix under tetris C file.

8.1 Software design algorithms

While hex5 and hex4 for the seven-segment display would have shown the received keyboard value, hex0 to hex3 would have been used for the display at score with the maximum game have a maximum playtime of 5 minutes, the score will deplete per second.

Equation 10: Initialised score.

$$Score = 2^{16}$$

Equation 11: Score depletion equation.

$$Score = Score - \frac{2^{16}}{300}$$

9 Results and discussion

Due to occurrences of errors within the construction of the project between hardware and software communication as well as the scheduling issues regarding completing the University’s semester the project was not able to succeed in the software design section, while the hardware section allowed for communication between on the FPGA cemented in certainty with testing points associated with each module.

The error was a timing and reset issue as the switch associated with the CPU needed to be triggered by flicking of the switch for the CPU to initialise however the solution finding was focused on the platform designer to search for a possible error in constructing and then trying to resolve a possible timing issue between the hardware components, software design and internal functions to no avail. Ultimately the testing for the software was not completed and did not succeed if getting basic functionally even though fixing the reset issue had working code but flags were not thought out properly with the CPU entering parts of the software multiple times when the entering and exiting were to occur every one second.

10 Conclusion and future improvement

Better scheduling for the project would allow appropriate timing to finish the project properly as well as better process regarding solution finding, the improvements for the project other than scheduling for a successful completion improvements that could be added to the project are multiple use of frames in terms of a beginning & final screen with two separate endings, implementation of a defaulting depending on how to natural progression of the gameplay (if it is too easy), sound implementation that would play the required theme of Tetris along with noises regarding collisions.

11 Reference

[1] Blue Planet Software, “2009 Tetris Design Guideline”, Blue Planet Software, Japan, Vol. 1, 2009, Accessed 1/10/2019: https://github.com/frankkopp/Tetris/blob/master/2009%20Tetris%20Design%20Guideline.pdf [Online].

[2] M. Hinner. “VGA Timings”. Martin.Hinner. Online: http://martin.hinner.info/vga/timing.html (Accessed: 1/10/2019).

[3] The Computer Engineering Research Group. “PS/2 Controller”. The Computer Engineering Research Group. Online: http://www.eecg.toronto.edu/~jayar/ece241_08F/AudioVideoCores/ps2/ps2.html (Accessed: 1/10/2019).

12 Appendix

12.1 onesec_counter module

```
/*
    Joseph Simeon, s2966176
    Griffith University
    6303ENG - Design of Real-Time Systems

    Module design: One Second Counter

    Input PLL at lowest possible frequency therefore module
    outputs a one second signal.
*/

module onesec_counter(
    input in_clk,
    output reg out_clk
);
    parameter DIVISOR = 21'd2000000;
    reg [20:0] counter = 0;

    always@(posedge in_clk)
    begin
        if(counter >= (DIVISOR-1))
        begin
            out_clk <= ~out_clk;
            counter <= 0;
        end
        else
            counter <= counter + 1;
    end
endmodule
```

12.2 vga_controller module

```
/*
    Joseph Simeon, s2966176
    Griffith University
    6306ENG - Design of Real-Time Systems

    Module design: VGA Controller

    Handles VGA output, tile engine and sprite engine

    VGA Controller 640x480@60Hz
    Pixel clk at 25.17 MHz
    Horizontal:     800 Pixel = 640 + 48 + 16 + 96
    Vertical:       525 Pixel = 480 + 10 + 33 + 2
*/
```

```
module vga_controller(
    input pix_clk, reset,
    input [11:0] colour,
    input [5:0] tile_number,
    output reg vsync, hsync, display,
    output reg [3:0] r, g, b,
    output reg [13:0] rom_address,
    output reg [10:0] ram_address,
    output reg vga_wren
);
    // local parameters for 640x480@60Hz
    localparam H_DISPLAY              = 640;
    localparam H_FRONTPORCH           =  16;
    localparam H_SYNCPULSE            =  96;
    localparam H_BACKPORCH            =  48;
    localparam H_MAX                      = H_DISPLAY + H_FRONTPORCH + H_SYNCPUL
SE + H_BACKPORCH - 1;
    localparam START_H_PULSE          = H_DISPLAY + H_FRONTPORCH;
    localparam END_H_PULSE            = H_DISPLAY + H_FRONTPORCH + H_SYNCPULSE -
 1;

    localparam V_DISPLAY                  = 480;
    localparam V_FRONTPORCH           =  10;
    localparam V_SYNCPULSE            =   2;
    localparam V_BACKPORCH            =  33;
    localparam V_MAX                      = V_DISPLAY + V_FRONTPORCH + V_SYNCPUL
SE + V_BACKPORCH - 1;
    localparam START_V_PULSE          = V_DISPLAY + V_FRONTPORCH;
    localparam END_V_PULSE            = V_DISPLAY + V_FRONTPORCH + V_SYNCPULSE -
 1;

    localparam SPRITE_LENGTH          = 16;
    localparam SPRITE_WIDTH           = 16;

    localparam TILE_WIDTH             = H_DISPLAY / SPRITE_WIDTH;
    localparam TILE_LENGTH            = V_DISPLAY / SPRITE_LENGTH;

    reg [9:0] posx, posy;

    // checking position of x,y counter against display parameters
    always@(posedge pix_clk or posedge reset)
    begin
        if(reset)
        begin
            posx <= 0;
            posy <= 0;
        end
        else
        begin
            if(posx < H_MAX)
                posx <= posx + 1;
            else
            begin
                posx <= 0;
                if(posy < V_MAX)
                    posy <= posy + 1;
                else
                    posy <= 0;
            end
        end
    end
    // assigned values of hsync, vsync & display
    always@(posx or posy)
    begin
        if((posx >= START_H_PULSE) && (posx <= END_H_PULSE))
            hsync <= 1;
        else
            hsync <= 0;
    end
    always@(posx or posy)
    begin
        if((posy >= START_V_PULSE) && (posy <= END_V_PULSE))
            vsync <= 1;
        else
            vsync <= 0;
    end
    always@(posx or posy)
    begin
        if((posx < H_DISPLAY) && (posy < V_DISPLAY))
            display <= 1;
        else
            display <= 0;
    end

    // sending for tile map
    always@(posx or posy)
    begin
        if(display)
        begin
            vga_wren <= 0;
            ram_address <= ((posy/SPRITE_WIDTH)*TILE_WIDTH)+(posx/SPRITE_LENGT
H);
        end
    end
    // sending for sprite
    always@(tile_number)
    begin
        if(display)
        begin
            rom_address <= (tile_number*SPRITE_LENGTH*SPRITE_WIDTH)+((posy%SPR
ITE_WIDTH)*SPRITE_WIDTH)+(posx%SPRITE_LENGTH)-1;
        end
    end

    // rgb transmission
    always@(negedge pix_clk or posedge reset)
    begin
        if(reset)
        begin
            r <= 4'd0;
            b <= 4'd0;
            b <= 4'd0;
        end
        else
        begin
            r <= (display) ? colour[11:8] : 4'd0;       // data[11:8]
            g <= (display) ? colour[7:4] : 4'd0;        // data[7:4]
            b <= (display) ? colour[3:0] : 4'd0;        // data[3:0]
        end
    end
endmodule
```

12.3 tilemap_mif_generator

```
clc;
clear all;
close all;

pkg load io

src = xlsread('TileMap.xlsx','Sheet1','A1:AN30');

[n,m] = size(src);
bit = 16;

fid = fopen("tilemap.mif", "w");

fprintf(fid, ['WIDTH = %i;\n'], bit);
fprintf(fid, ['DEPTH = %i;\n\n'], n*m);

fprintf(fid, ['ADDRESS_RADIX=UNS;\n']);
fprintf(fid, ['DATA_RADIX=UNS;\n\n']);
fprintf(fid, ['CONTENT BEGIN\n']);
for i = 1:n
  for j = 1:m
    address = (40*(i-1))+(j-1);
    fprintf(fid, ['\t%i'], address);
      if(address < 10)
        fprintf(fid, ['     :']);
        elseif(address < 100)
        fprintf(fid, ['    :']);
        elseif(address < 1000)
        fprintf(fid, ['   :']);
        elseif(address < 10000)
        fprintf(fid, ['  :']);
      endif
      fprintf(fid, ['   %d'], src(i,j));
      fprintf(fid, [';\n']);
  endfor
endfor
fprintf(fid, ['END;']);

fclose(fid);
```

12.4 sprite_mif_generator

```
clc;
clear all;
close all;

picname = {'0.png','1.png','2.png','3.png','4.png','5.png','6.png','7.png','8.png','9.png','10.png','11.png','12.png','13.png','14.png','15.png','16.png','17.png','18.png','19.png','20.png','21.png','22.png','23.png','24.png','25.png','26.png','27.png','28.png','29.png','30.png','31.png','32.png','33.png','34.png','35.png','36.png','37.png','38.png','39.png','40.png','41.png'};
src = imread('0.png');
R = src(:,:,1);
[d,n] = size(picname);
[l,w] = size(R);

fid = fopen("sprites.mif", "w");

fprintf(fid, ['WIDTH = %i;\n'], w);
fprintf(fid, ['DEPTH = %i;\n\n'], n*l*w);

fprintf(fid, ['ADDRESS_RADIX=UNS;\n']);
fprintf(fid, ['DATA_RADIX=UNS;\n\n']);

fprintf(fid, ['CONTENT BEGIN\n']);
for i = 1:n
  string = char(picname(i));
  src = imread(string);

  R = src(:,:,1);
  G = src(:,:,2);
  B = src(:,:,3);

  for j = 1:l
    for k = 1:w
      address = (l*w*(i-1))+(l*(j-1))+(k-1);
      temp_R = bitshift(uint8(R(j,k)),-4);
      temp_G = bitshift(uint8(G(j,k)),-4);
      temp_B = bitshift(uint8(B(j,k)),-4);

      rgb = (256*double(temp_R))+(16*double(temp_G))+(double(temp_B));

      fprintf(fid, ['\t%i'], address);
      if(address < 10)
        fprintf(fid, ['     :']);
        elseif(address < 100)
        fprintf(fid, ['    :']);
        elseif(address < 1000)
        fprintf(fid, ['   :']);
        elseif(address < 10000)
        fprintf(fid, ['  :']);
        elseif(address < 100000)
        fprintf(fid, [' :']);
      endif
      fprintf(fid, ['   %d'], rgb);
      fprintf(fid, [';\n']);
    endfor
  endfor
endfor
fprintf(fid, ['END;\n']);

fclose(fid);
```

12.5 ps2control module

```
/*
    Joseph Simeon, s2966176
    Griffith University
    6306ENG - Design of Real-Time Systmes

    Module design: Keyboard interface via ps2
*/

module ps2control(
    input ps2_clock, reset, ps2_data,
    output reg [7:0] keyboard_data,
    output reg keyboard_dataout,
    output reg [7:0] led_out
);
    reg [4:0] count = 0;
    reg [10:0] data = 0;
    reg flag = 0;

    always@(negedge ps2_clock or posedge reset)
    begin
        if(reset)
        begin
            count <= 0;
            data <= 0;
            keyboard_dataout <= 0;
            keyboard_data <= 0;
            flag <= 0;
        end
        else
        begin
            // obtaining data from keyboard
            keyboard_dataout <= 0;
            case(count)
                0 : begin
                    // start bit
                       data[0] <= ps2_data;
                end
                1 : begin
                    // data
                       data[1] <= ps2_data;
                end
                2 : begin
                    // data
                       data[2] <= ps2_data;
                end
                3 : begin
                    // data
                       data[3] <= ps2_data;
                end
                4 : begin
                    // data
                       data[4] <= ps2_data;
                end
                5 : begin
                    // data
                       data[5] <= ps2_data;
                end
                6 : begin
                    // data
```

```
                        data[6] <= ps2_data;
                end
                7 : begin
                    // data
                        data[7] <= ps2_data;
                end
                8 : begin
                    // data
                        data[8] <= ps2_data;
                end
                9 : begin
                    // parity bit
                        data[9] <= ps2_data;
                end
                10 : begin
                    // end bit
                        data[10] <= ps2_data;
                end
            endcase
            if(count == 10)
            begin
                // reset count
                count <= 0;
                // data has been recieved
                // if a break is received the flag will be alerted
                // flag break will skip over any code that is sent when key is
released
                if(flag == 1)
                    flag <= 0;
                else
                begin
                    // check received data, emit EO & FO breaks
                    // when F0 is reached, flag is set
                    case(data[8:1])
                        'hE0 : begin
                            flag <= 0;
                        end
                        'hF0 : begin
                            flag <= 1;
                        end
                        default : begin
                            keyboard_dataout <= 1;
                            keyboard_data <= data[8:1];
                            led_out <= data[8:1];
                            flag <= 0;
                        end
                    endcase
                end
            end
            else
            begin
                // increment count
                count <= count + 1;
            end
        end
    end
endmodule
```

12.6 softprocessor module

```
module softprocessor(
    input cpu_clock, cpu_reset, onesec_clock,
    input [5:0] tile_number_read,
    input [7:0] keyboard_input,
    input keyboard_dataflag,
    output [6:0] hex0, hex1, hex2, hex3, hex4, hex5,
    output [10:0] ram_address,
    output [5:0] tile_number_write,
    output ram_write_enable
);

        tetriscpu2 u0 (
        .clk_clk                                      (cpu_clock),
      //  clk.clk
        .reset_reset_n                                (cpu_reset),
      //  reset.reset_n
        .onesec_clock_external_connection_export      (onesec_clock),
      // onesec_clock_external_connection.export
        .tile_number_read_external_connection_export  (tile_number_read[5:0]),
      // tile_number_read_external_connection.export
        .keyboard_dataflag_external_connection_export (keyboard_dataflag),
      // keyboard_dataflag_external_connection.export
        .keyboard_data_external_connection_export     (keyboard_input[7:0]),
      // keyboard_data_external_connection.export
        .hex0_external_connection_export              (hex0[6:0]),
      // hex0_external_connection.export
        .hex1_external_connection_export              (hex1[6:0]),
      // hex1_external_connection.export
        .hex2_external_connection_export              (hex2[6:0]),
      // hex2_external_connection.export
        .hex3_external_connection_export              (hex3[6:0]),
      // hex3_external_connection.export
        .hex4_external_connection_export              (hex4[6:0]),
      //  hex4_external_connection.export
        .hex5_external_connection_export              (hex5[6:0]),
      //  hex5_external_connection.export
```

```
        .ram_write_enable_external_connection_export  (ram_write_enable),
      //  ram_write_enable_external_connection.export
        .ram_address_external_connection_export       (ram_address[10:0]),
      //  ram_address_external_connection.export
        .tile_number_write_external_connection_export (tile_number_write[5:0])
      //  tile_number_write_external_connection.export
    );
endmodule
```

12.7 tetris C file