ECE540

Final Project
Battle Tanks
Theory of Operation

Team Members:

Mark Chernishoff

Hai Dang

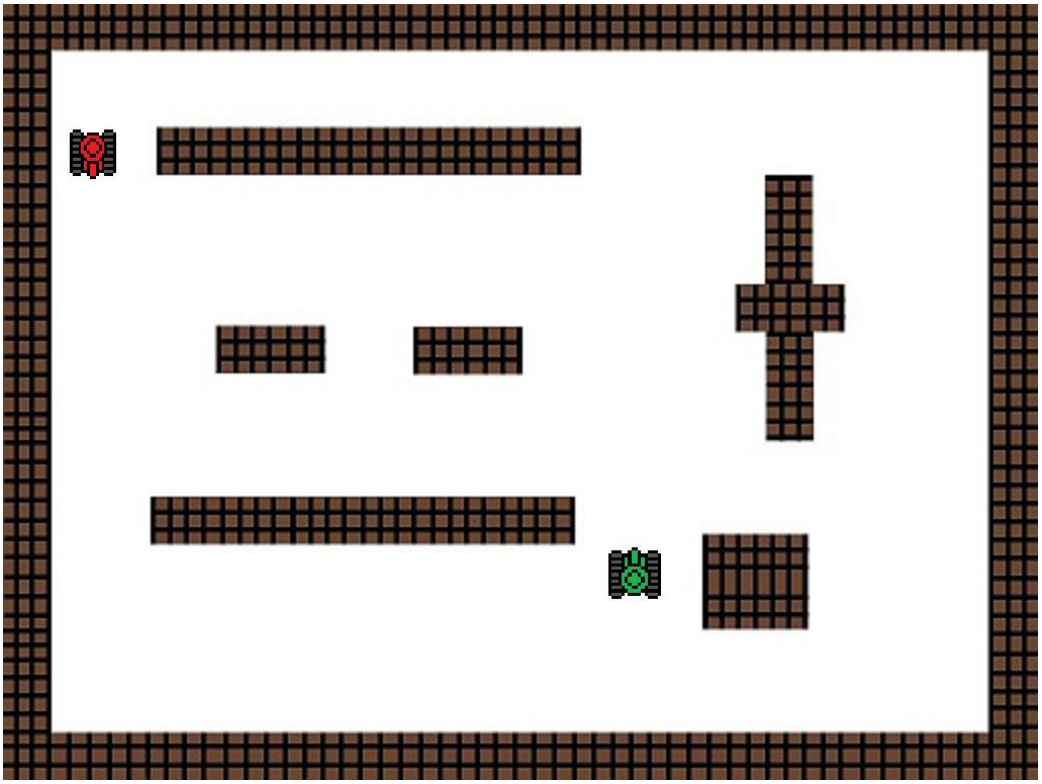
Aditya Pawar

12/6/16

Portland State University

I) Introduction

The game of Battle Tanks consists of two tanks who are trying to shoot each other. The player 1 (green tank) tries to shoot player 2 (red tank) with bullet fired from the tanks. At the beginning to game the tanks are placed at different positions in map. The tank will fire only one bullet when space key (for player 1) or enter key (player 2) is pressed. The map consists of multiple brick walls which help the players to hide from each other or to dodge the shell.



Instructions:
Player 1 and Player 2 can move in 4 directions with the help of keyboard keys.

The keys for Player 1 are:
W = up direction
A = left direction
S = down direction
D = right direction

The keys for player 2 are:
I = up direction
J = left direction
K = down direction
L = right direction

Player 1 can use space bar for firing the bullet and Player 2 can use enter key.

Game Rules:
If any player 1 is able to destroy another player 2 for 3 times or vice versa, then player 1 or player 2 wins.

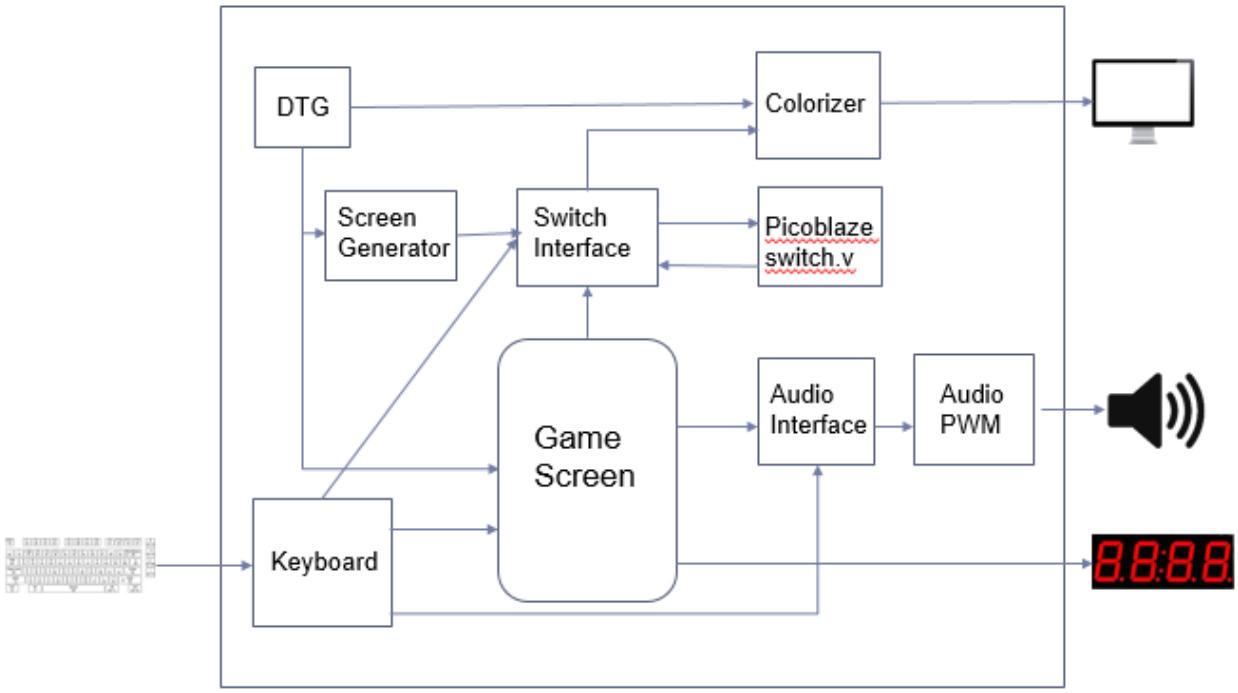
This document describes the theory of operation, design, implementation of hardware and firmware with gameplay logic.

Hardware: Digilent Nexys™4 DDR Artix-7 FPGA Board, VGA monitor for display, keyboard for game control, speakers for Audio output.

Soft Processor Core: Picoblaze KCPSM6

EDA Tools: Xilinx Vivado Suite 2016.2 (for Synthesis, Place and Route and downloading the implemented design to the FPGA Board).

II) Block diagram:



The above block diagram shows the overall integration of all the modules used to design this project. It gives a general overview of the whole system. This project uses keyboard to control two tanks direction and their bullets. The Screen Generator regarding to first screen, and 2 game over screens take coe and output from DTG as the input. These coe files are generated by using a MATLAB script which converts any image into 12 bits coe. Screen interface take output from Screen Generator as a parameter input. It is implemented through an interface software Picoblaze. The output of interface is driven into Colorizer module to display on a 1024x768 resolution VGA display. Game Screen implements game logic battle in which it contains animation and score. The score is displayed on seven-segment led. Output game screen also makes the song for all animation in game logic. The overall diagram gives a detail idea of how the integration is done. Each of the above module is explained below with brief description

III) Keyboard interface:

Hardware:

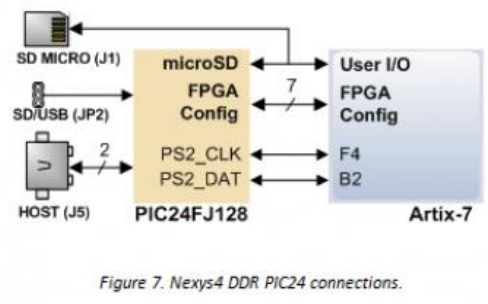


Figure 7. Nexys4 DDR PIC24 connections.

The auxiliary function microcontroller (Microchip PIC24FJ128) provides the Nexys4 DDR with USB embedded HID host capability. Once the FPGA is programmed, the microcontroller switches to application mode, which is USB HID host in this case. Firmware in the microcontroller can drive a keyboard attached to the type A USB connector at J5. The PIC24 drives several signals into the FPGA to implement a standard PS/2 interface for communication with a keyboard.

PS2 communication:

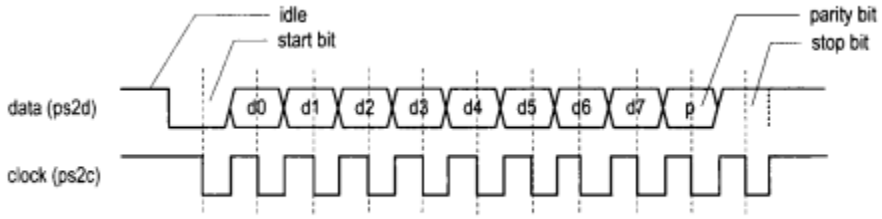


Figure 9.1 Timing diagram of a PS2 port.

The communication of the PS2 port is bidirectional and the host can send a command to the keyboard to set certain parameters. The PS2 port contains two wires for communication purpose. One wire is for data, which is transmitted in a serial stream. The other wire is for the clock information, which specifies when the data is valid and can be retrieved. The information is transmitted as an 11-bit packet that contains a start bit (‘0’), 8 data bits (scan code LSB first), an odd parity bit, and a stop bit (‘1’).

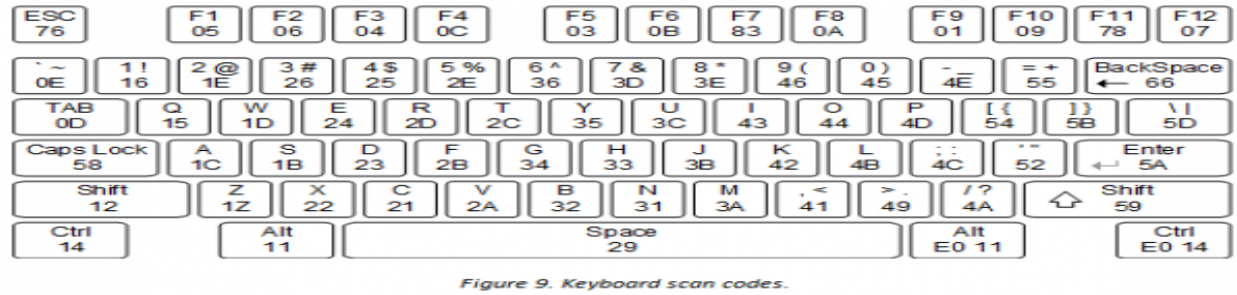


Figure 9. Keyboard scan codes.

PS/2 keyboards use scan codes to communicate key press data. Each key is assigned a code that is sent whenever the key is pressed. If the key is held down, the scan code will be sent repeatedly. When a key is

released, an F0 key-up code is sent, followed by the scan code of the released key. For example, when we press and release the A key, the keyboard first transmits its make code and then the break code:

1C F0 1C

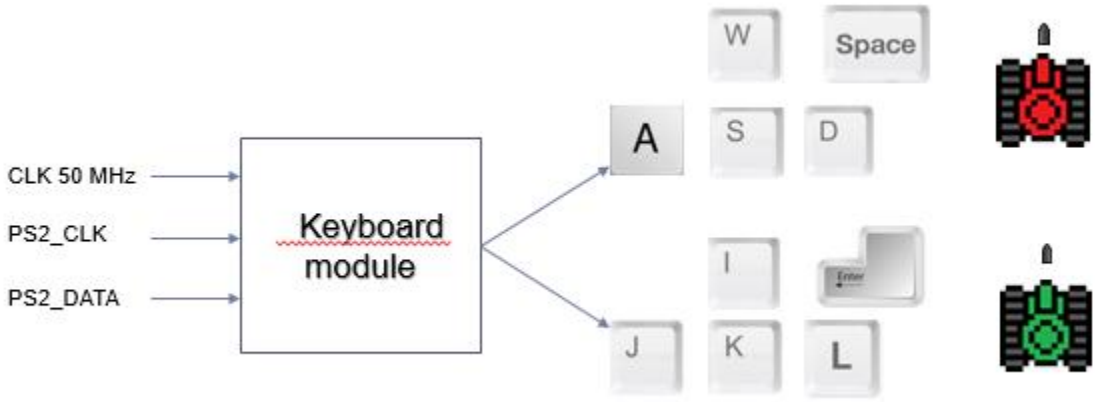
If we hold the key down for a while before releasing it, the make code will be transmitted multiple times:

1C 1C 1C … 1C F0 1C.

Multiple keys can be pressed at the same time. For example, we can first press the A key, and then the S key, and release the S key and then release the A key. The transmitted code sequence follows the make and break codes of the two keys:

1C 1B F0 1B F0 1C

Block diagram:



The input for module is clock with 50 MHz, PS2_CLK, and PS2_DATA. The output are signal to control tank direction and fire in terms of:

Red tank:

W → Up

S → Down

A → Left

D → Right

Space → Fire

Green tank:

I → Up

K → Down

J → Left

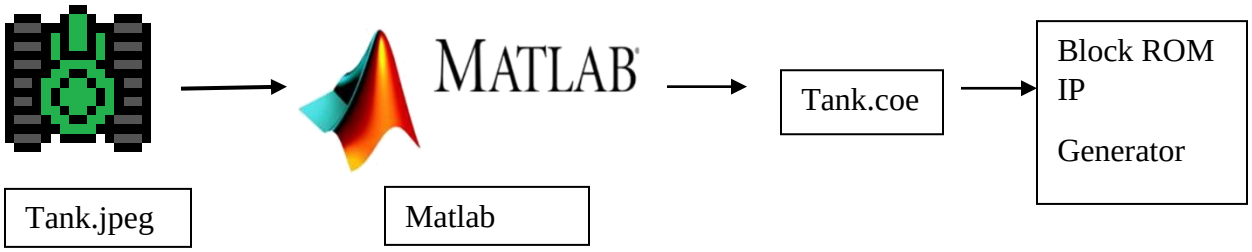
L → Right

Enter → Fire

IV) Image Generation

The images were made in paint in RGB format. The idea behind making the images was use very few colors so that we can save the memory. Also fewer colors mean that less gradation at boundary between different colors.

All the screens were made in paint in JPEG format. Matlab script (*mifgen.m*) was used generate the *image.mif* file. This file was later saved as *image.coe* file which contained the RGB values of image encoded in decimal values. Each pixel was encoded in 4 decimal values in decimal format.



For the first screen, player 1 screen and player 2 screen, we have used 128x96 resolution. The game screen was of resolution of 592x384. The green and red tank icons are of 32x32 pixel in up, down, left right orientation. Bullet coming from tank was made of 8x15 pixels for all directions.

The *.coe* files was inserted in BROM IP generator and BROM blocks were made for all the screens and icon

The first screen, player 1 screen and player 2 screen was scaled to 1024x768 by shifting the pixel_row and pixel_col by 3 and feeding them as address to BRAM. The game screen was scaled to 1024x768 by shifting the pixel_row and pixel_col by 1 and feeding them as address to BRAM.

```
//Zooming for background from 128x96 to 1024x768
assign pixel_row_B =  {3'b000,pPixel_row [9:3]};
assign pixel_column_B =  {3'b000,pPixel_column [9:3]};

reg [13:0] rAddr_first = 14'd0;

//Out from the Block RAM
wire [11:0] rDout_first;

// Block ROM to store the image information of background
blk_mem_gen_2 first_page (
  .clka(pClk),    // input wire clka
  .addra(rAddr_first),  // input wire [13 : 0] addra
  .douta(rDout_first)  // output wire [11 : 0] douta
);

//Address to be passed to the Block RAM


always @(posedge pClk2)begin
    if (pReset)
        first_screen_out <= 12'd0;
    else
        begin
            //Where ever is the pixel pointer, take the rgb value from the BRAM and print it.
            rAddr_first   <= {pixel_row_B[6:0],pixel_column_B[6:0]};
            first_screen_out  <= rDout_first;
        end
end
```

V) game_screen.v

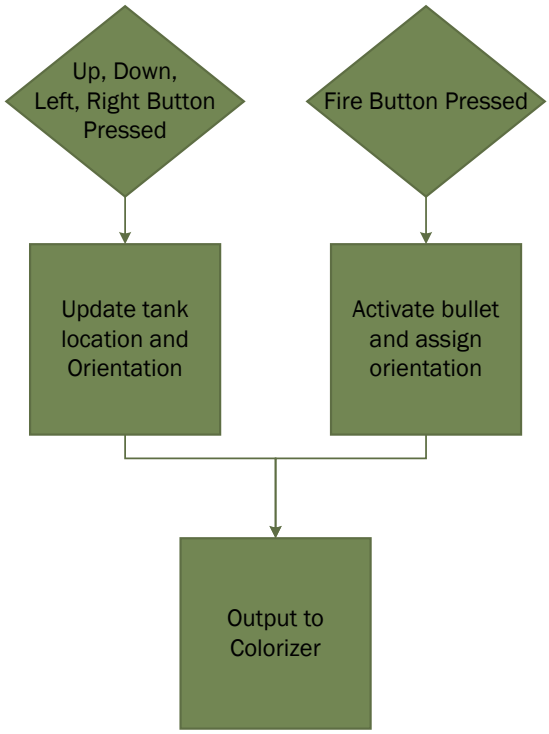


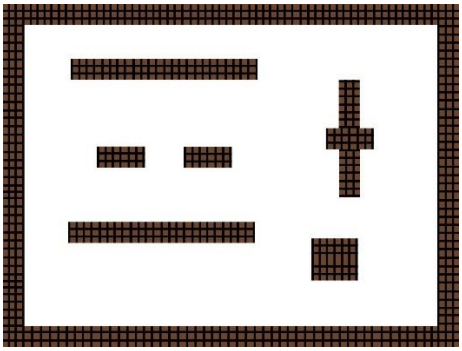
Figure 1 – Basic Block Diagram of game_screen

This module contains logic for displaying the game_screen, both red and green tanks, and bullets. This also keeps track of tank and bullet orientation and location. All the icons for tanks, bullets, and explosion are held in block ROM. Block ROM's hold color information that will eventually be output to the Colorizer.

a) Tank direction, Tank speed, and bullet speed:

The game_screen starts out by assigning an orientation to the tank whenever a button is pressed. When either up, down, left, or right are pressed for the tanks it is assigned a binary value from 2'b00 to 2'b11. ICON_UP is 2'b00, ICON_DOWN is 2'b01, ICON_LEFT is 2'b10, and ICON_RIGHT is 2'b11. If the buttons keep being pressed the location of the tank will increment along that orientation. This applies to both red and green tanks. At the bottom of the Verilog code, there is 2 counters. These counters are used to control the speed that the locations of the tanks and bullets can be incremented, because if the locations were incremented at the speed of the clock you would not be able to see the tanks move. So, a counter was implemented to count up every clock cycle and when it reached a count of 500,000 for tank a flag would be set. If that flag was set and the button push to move the tank are high are same time, the tank will increment once. The bullet needs to travel a bit faster than the tank, so the bullet_flag was set to a count of 100,000. At the moment the fire button is pressed the bullet will get the current tanks orientation and location and its location will increment along the path until it either hits a wall or another tank, at which point the bullet will disappear and reset its location.

b) Background Map:



To display the background map, the address is incremented according to the current pixel location output by the DTG module. The background map is always output unless the pixel location is over a red or green tank, a red or green bullet, or an explosion that happens. This is done by outputting icon data and incrementing the block ROM address by 1 everytime the `pPixel_row` and `pPixel_column` is within a 32x32 range of the tank or explosion animation. The range for the bullet is 8x15. This strategy is implemented 4 times. Once for the red tank which is labeled as `tank_start_locY` and `tank_start_locX`, once for the green tank labeled as `green_tank_locX` and `green_tank_locY`, once for the red bullet labeled as `red_bullet_locX` and `red_bullet_locY`, and finally once for the green bullet labeled as `green_bullet_locY` and `green_bullet_locX`.

c) Animation between bullet and wall, animation between tank and wall:

To stop the bullets from being able to travel through a wall, the rDout_first data for the background is read and if it is not equal to 12'd4095, which is white, the bullet is deactivated, locations for them are reset to 0, and will no longer be displayed. A similar strategy was implemented to stop the tanks from being able to travel through the walls. When the tanks reach a position where the background data, rDout_first, is no longer white or 12'd4095, at the tanks location the tank needs to be decremented by a single pixel, so it is no longer over a wall. After this happens then you will be able to move the tank again. There are 4 situations between tank and wall: Wall is above, below, left-side, and right-side tank.

+ *Tank hits the wall:*

```
else if ( rDout_first != 12'd4095 && (pPixel_row >= tank_start_locX ) && ( pPixel_row <= (tank_start_locX + 10'd31)) &&
                                    (pPixel_column >= tank_start_locY ) && ( pPixel_column <= (tank_start_locY + 10'd31)))
                begin
                     red_stop <= 1'd1;            // if at wall stop the movement of tank
                 end
else if ((flag > 0) )          // changing red tank location when button pushed and flag is set high
    begin
        if (red_stop == 1'd0)
            begin
                case (btns)         // change tank location if buttons pressed
                    UP:     tank_start_locX <= tank_start_locX - 1'd1;
                    DOWN:   tank_start_locX <= tank_start_locX + 1'd1;
                    LEFT:   tank_start_locY <= tank_start_locY - 1'd1;
                    RIGHT:  tank_start_locY <= tank_start_locY + 1'd1;
                    default:    begin
                                    tank_start_locY <= tank_start_locY;
                                    tank_start_locX <= tank_start_locX;
                                end
                endcase
            end
```

+ *Tank does not hit the wall:*

```
    else if (red_stop >= 1'd1)       // stop is set high, reverse 1 location and clear the stop
         begin
         red_stop <= 1'd0;           // clear the stop flag
             case (btns)
                 UP:         tank_start_locX <= tank_start_locX + 1'd1;
                 DOWN:       tank_start_locX <= tank_start_locX - 1'd1;
                 LEFT:       tank_start_locY <= tank_start_locY + 1'd1;
                 RIGHT:      tank_start_locY <= tank_start_locY - 1'd1;
                 default:     begin
                                  tank_start_locY <= tank_start_locY;
                                  tank_start_locX <= tank_start_locX;
                              end
              endcase
         end
    else begin
             tank_start_locX <= tank_start_locX;
             tank_start_locY <= tank_start_locY;
             red_stop <= 1'd0;
         end

end
```

d) Explotion:

When a tank is destroyed an explosion icon will appear for about a second. After which, the X and Y locations of the tank are reset to their initial positions. Red tank is reset to pixel location 60x60. The green tank is reset to location 560x560.

e) Score and Output signal:

Finally, a few signals are output to other varying modules. They would be the red_score, the green_score, player_screen, and explosion_act. The red_score and green_score are simple signals, they just keep track of each tanks score and are output to SSD. The player_screen is a 2-bit selector that is output to the SwitchInterface.v whenever a red_score or green_score reaches 3. This is used to display to the VGA whichever player won. player_screen has a value of 2'b01 when red player wins, and 2'b10 when green player wins. explosion_act is used to output to the AudioInterface to output sound whenever a tank is destroyed.

VI) SwitchInterface.v

The switch interface is just used to control the communication with the Picoblaze switch.v program. The SwitchInterface inputs are the FirstScreen, GameScreen, Player1Wins, and Player2Wins data outputs to VGA. There is also inputs from the keyboard and player_screen that are used to control which screen needs to be output to the display. To achieve this, whenever the player_screen or keyboard data is greater than 0, an interrupt is sent to the picoblaze switch.v program. Depending on what the state of the game is in it will either use the player_screen selector bits to output a Player 1 or 2 Wins screen. Or if it will use keyboard 'Y' or 'N' buttons to change to GameScreen or FirstScreen. When the switch.v decides what screen to select a binary value of 2'b00 to 2'b11 is assigned to screenout variable. 2'b00 is used to output the FirstScreen, 2'b01 is used to output GameScreen, 2'b10 is used to output Player1Wins, and finally 2'b11 is used to output Player2Wins screen. This is all implemented using a case statement where screenout is the parameter. This case statement is also used to reset the player_screen selector bit in game_screen.v to 0, so that the GameScreen is not always output.

Picoblaze Switch.v Program:

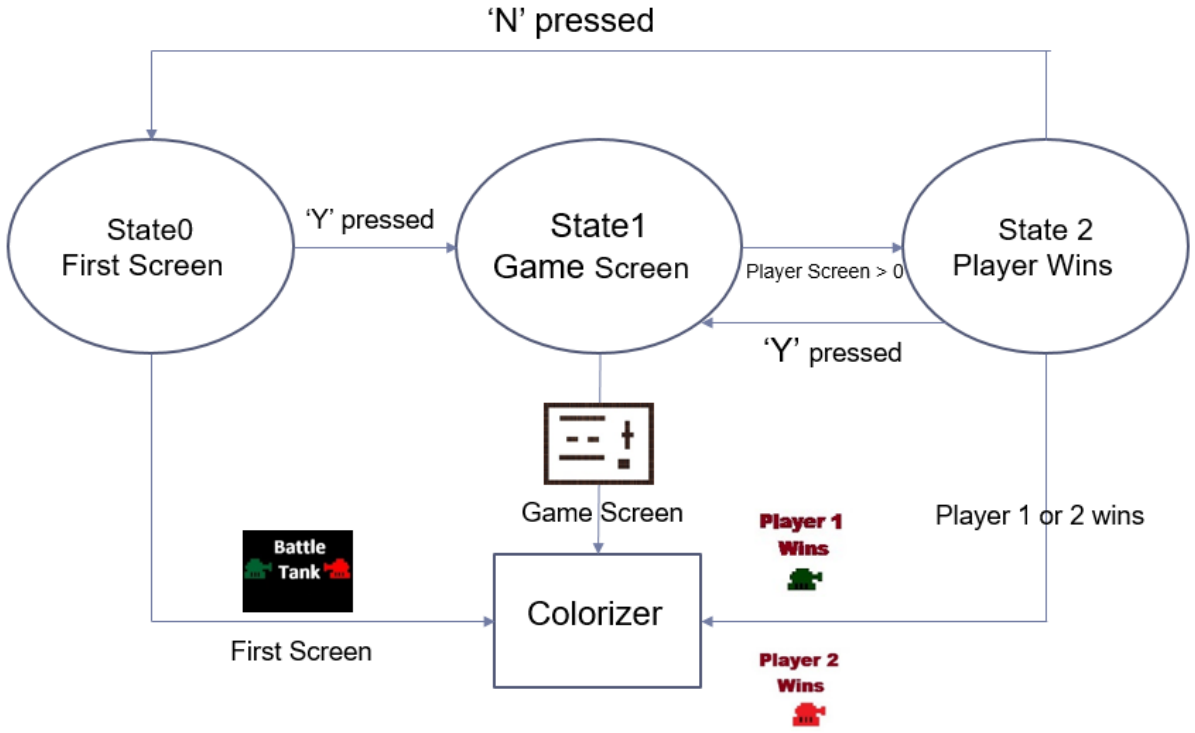


Figure 2 - State Machine for picoblaze program

The picoblaze program is just a state machine. All the screen changes happen when an interrupt occurs. In the main loop, main_l0, to kill time a delay_100ms is called to delay some time then complement the value of the LED[0] on the board. Complementing the LED is used as a watchdog timer so that if a problem were to occur in the program I could see if it was still running.

To cause an interrupt, the player_screen value from the game_screen.v or a keyboard 'Y' or 'N' button need to be pressed. It is initialized to state0 which just outputs the FirstScreen. An interrupt occurs whenever a letter 'Y' or 'N' on the keyboard is pressed. In this case the state only changes when 'Y' is pressed. When in state1 the output is the GameScreen. The output only will change when either player reaches a score of 3. Depending on the player, the player_screen value will either be a 2'b01 or 2'b10 and now player_screen value is greater than 0 and an interrupt will occur and it will change screens. If it is 2'b01, Player1Wins screen will be displayed. Alternatively, Player2Wins screen will be displayed if the value is 2'b10. In either case, it is now in state2 and can go back to state0 or state1. The program will have to wait for another interrupt to occur to either play again, a 'Y' key press on the keyboard, or quit, which is a 'N' key press on the keyboard.

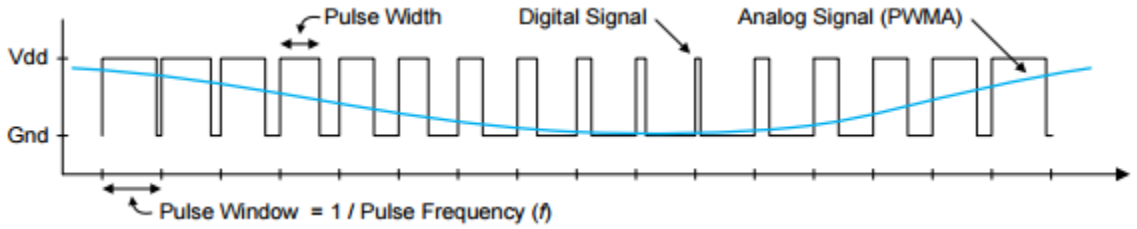
VII) Mono Audio interface:

a) Hardware:

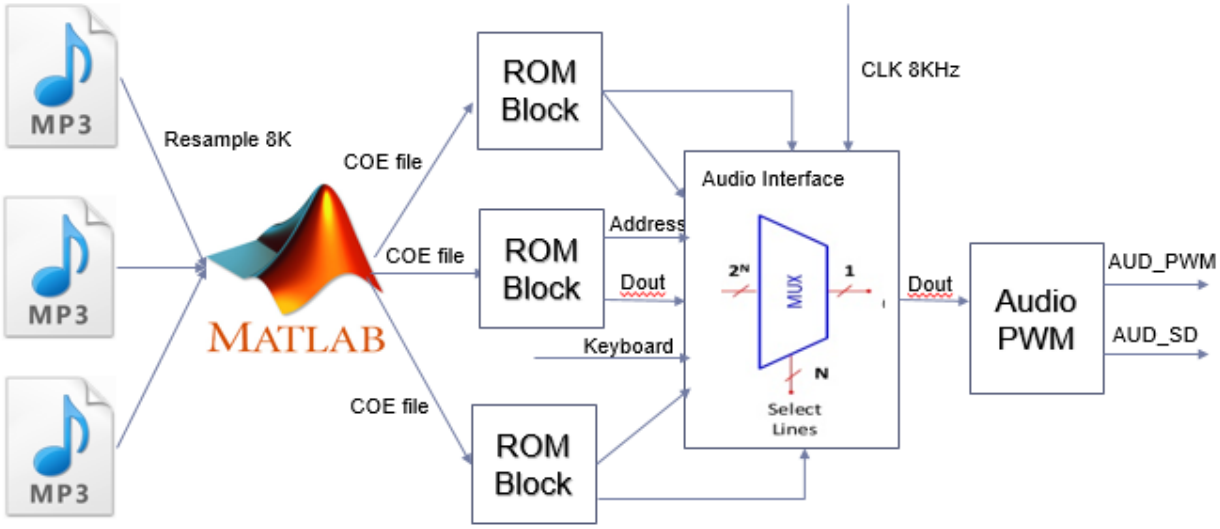
The on-board audio jack (J8) is driven by a Sallen-Key Butterworth Low-pass 4th Order Filter that provides mono audio output. The input of the filter (AUD_PWM) is connected to the FPGA pin A11. A digital input will typically be a pulse width modulated (PWM) signal or pulse density modulated (PDM) signal produced by the FPGA.

b) Pulse-Width Modulation:

A pulse-width modulated (PWM) signal is a chain of pulses at some fixed frequency, with each pulse potentially having a different width. This digital signal can be passed through a simple low-pass filter that integrates the digital waveform to produce an analog voltage proportional to the average pulse width over some interval. For example, if the pulses are high for an average of 10% of the available pulse period, then an integrator will produce an analog value that is 10% of the Vdd voltage. The below figure shows a waveform represented as a PWM signal.



c) Block diagram:



Matlab script *makecoe.m* resample audio file (mp3 format) to 8KHz frequency, and convert them to .coe file respectively. These coe files are inserted the block ROM to generate Addr and 8-bit data (Dout). Addr and 8-bit Dout are passed as the input for *Audio Interface* module. The *Audio Interface* operates like a multiplexer with control signal coming from keyboard and output from *Game Screen* module. For example, when we press the button to control the direction tank, it produces the background song. When we press the button 'space' (red tank fires), it produces the bump song. In addition, when the output signal explosion from *Game Screen* module is high, it produces the song for explosion respectively.

The 8-bit digital information (Dout) from *Audio Interface* module is passed as a parameter input for *AudioPWM* module at sampling frequency of 8Khz. This module will convert these samples into PWM information with duty cycle proportional to the magnitude of 8-bit digital data. And then, this signal goes into an analog filter which ultimately arrives at Mono audio output jack on Nexys4 DDR board.

VIII)Challenges, Enhancement and Improvement:

a) Challenges:

1) Bram issue: because vga part and audio interface use ROM block (coe file), it consumes too much memory space. If we use the original picture with resolution 1024x768, we cannot load coe file into ROM block because of the lack of Bram. Similarly, if we take a mp3 song too long, we do not have enough memory space to load coe file into ROM block.

   Solution: For image generation, each image was painted in photoshop in which each pixel is zoomed 8 times. Only game screen is zoomed 2 times because we use it for game animation. For audio, we only used a small section of song rather than using the entire song.

2) Animation: because game screen is zoomed 2 times, it has some noise pixel in white screen. Consequently, it has a strong negative effect on my algorithm for tank moving.

   Solution: We enhance the algorithm by using flag stop signal to tell the tank stop and go back the white screen. Although the tanks slowly move when they hit wall, it enhances the serious bugs animation in game. In particular, tank cannot pass the wall.

3) Keyboard interface: we need to detect situation in which the user holds a button or presses and then releases button.

   Solution: we enhance the algorithm by using break signal F0 to detect this situation.

b) Enhancement:

In proposal, we use buttons and switch to control the tank. In design, we use PS2 keyboard instead. In addition, we use audio and make channel for audio song. It is not mention in proposal.

c) Improvement:

Using Android app instead of keyboard.

Create bonus items (Ex. Increase speed tank, change type of bullets, …).

Create more maps and let player choose.

Change style game by writing AI bot (2 tanks fights these AI bot).

REFERENCES

[1] Nexys4 DDR™ FPGA Board Reference Manual retrieved from:

https://reference.digilentinc.com/reference/programmable-logic/nexys-4-ddr/reference-manual

[2] FPGA Prototyping By Verilog Examples, Pong P. Chu

[3] Matlab script: makecoe.m retrieved from:

https://github.com/gajjanag/6111_Project/blob/master/assets/audio_convert/makecoe.m

[4] Kcpsm6_design_template.v included in the Picoblaze download from Xilinx

[5] Picoblaze for Virtex-6 and 7-Series (KCPSM6) User Guide, Release 9 by K.Chapman

Link video demo:

https://www.youtube.com/watch?v=r9YjkmzVmh0&feature=youtu.be