Digital Lab Final Project Report

Tetris (rocker)

Group No. 10
Group members: 111511181吳毅恩、111511198鄭恆安

Introduction

This report presents the final project on Tetris game development by using Verilog and implementing it on an FPGA(Nexys DDR 4). Our project focuses on implementing a Tetris game with a unique feature called "rocker," which allows players to control the game using the FPGA board's tilting motion.

Traditional Tetris is controlled by buttons. What we have done is that we can control Tetris by not only the physical buttons on the board but also by tilting the FPGA board. By realizing this function, players now no longer need to keep their hands on the board forever; instead, players only need to control the button that rotates the Tetris, and other manipulation can be done by tilting the board.

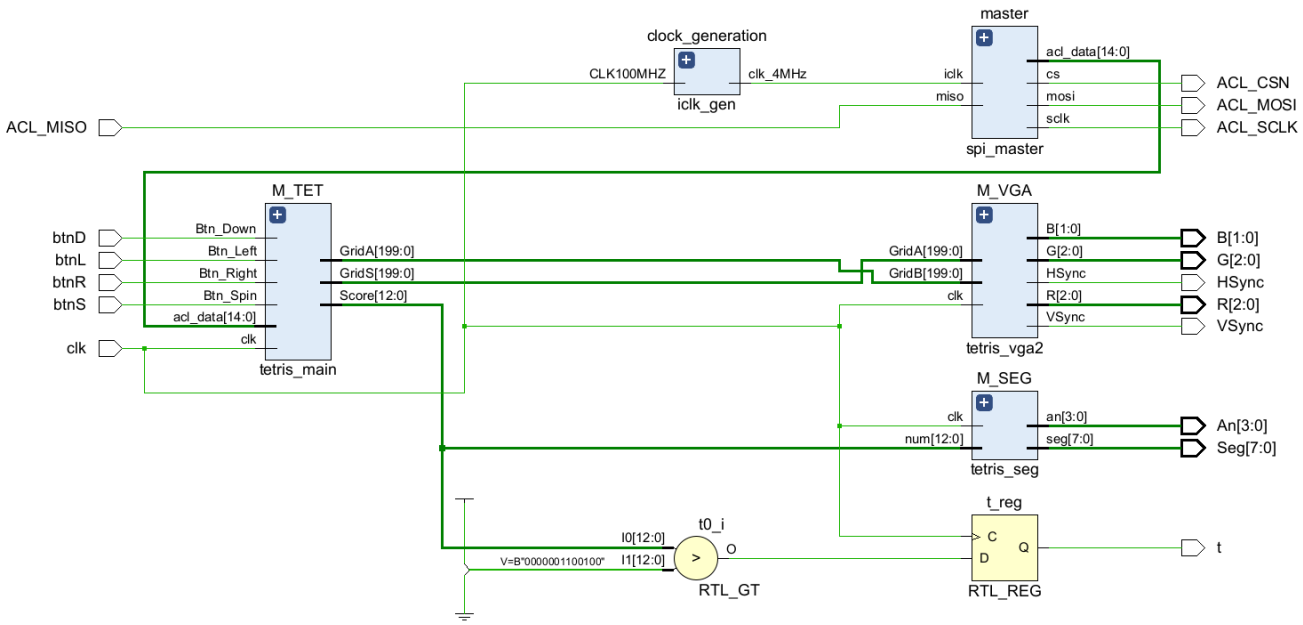
In this project, we integrate the built-in IMU and the built-in VGA output port to realize the Tetris game. The IMU will detect the acceleration of the three axes and pass the input to the main Tetris module. After the calculation, the main module will pass the data to the module responsible for VGA display and ultimately display on the LCD monitor.

Motivation

When we decided to choose Tetris as our final project, we initially planned to remake this game like its traditional way. However, when we first connected the VGA output to an LCD monitor, the default program in the FPGA board was then shown on the screen. We found out that the board has an built-in IMU which can perfectly detect the tilting angle of the board itself. With this observation, we decided to add a feature in our project and may become the highlight for the game. We want to control Tetris by tilting the board, just as most of the mobile racing games. By realizing this feature, players can free their hands and play the game more fluently.

Implementation

This diagram is the whole schematic of our project code, and we will illustrate the implementation of different modules below. The Tetris game we have designed is basically composed of three modules, Tetris game control module, accelerometer module, and VGA display module. Below is the hierarchy design of the Tetris game.




```
input clk, btnL, btnR, btnD, btnS;
input ACL_MISO;             // master in
output ACL_MOSI;            // master out
output ACL_SCLK;            // spi sclk
output ACL_CSN;             // spi ~chip select
output HSync, VSync;
output [2:0] R;
output [2:0] G;
output [1:0] B;
output [7:0] Seg;
output [3:0] An;
output reg t;
```

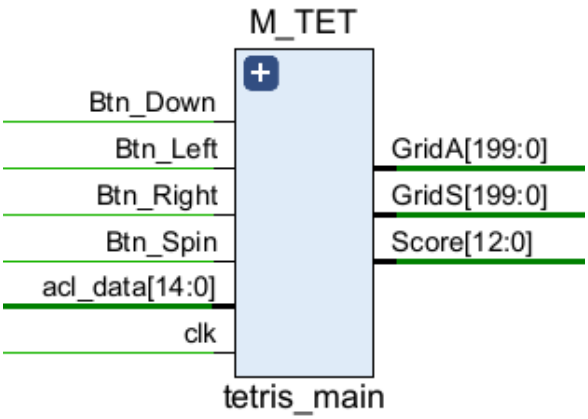
Tetris game function table

Button (Nexys DDR 4)	Function Description
btnL	Move blocks to the left
btnR	Move blocks to the right
btnD	Move blocks to the bottom
btnS	Rotate blocks / Game restart (when game ended)
rocker (the whole FPGA board)	Tilt the board to resemble the functions of the buttons. (tilt leftward to perform click on btnL, …)

part1: Tetris

The main concept of the Tetris module is a finite state machine. By defining all the conditions that might happen in the game, we can implement the game operation by assigning different states to the current state when different manipulations are made by the player.

Below is the schematic and the IOs of the Tetris module.




```
input clk, Btn_Left, Btn_Right, Btn_Down, Btn_Spin;
input [14:0]acl_data;
output wire [199:0] GridS;
output wire [199:0] GridA;
output reg [12:0] Score;
```

IOs	fucntion
clk	built-in input with a frequency of MHz.
Btn_Left, Btn_Right Btn_Down Btn_Spin	control the Tetris motion.
acl_data	retrieve the accelerometer's data
GridS, GridA	output ports for the display
Score	the current score according to the game state

The code processes accelerometer data (**acl_data**) to extract sign bits and 6 bits of axis data for each axis (**x_data**, **y_data**, **z_data**). It then performs binary to binary-coded decimal (BCD) conversion for each axis, storing the tens and units digits separately.

When we get the x, y, and z axis' acceleration, we can determine the tilting angle of the FPGA board, which will in turn simulate the button pressing operation.

```
wire [3:0] x_data, y_data, z_data;
assign x_data = x_sign==0 ? acl_data[13:10]:16-acl_data[13:10];
assign y_data = y_sign==0 ? acl_data[8:5]:16-acl_data[8:5];
assign z_data = z_sign==0 ? acl_data[3:0]:16-acl_data[3:0];
```

Registers to store the game process

We use registers (**AP_State_PS, AP_State_PSd, AP_State_SA, Game_State, Game_State_Counter, Row_Elim_Counter, Game_Clock_Counter, Game_Clock_Acc, Random_Counter, clk2_Counter**) and combinational logic to control the game state transitions, game clock, and various counters related to game logic and timing.

Active piece generation

The code generates the active Tetris piece grid based on the current piece state (AP_State_PS). It assigns the appropriate piece state data (AP_State_PSd) to Grid_Active, which represents the active piece within the game grid.

Each state represents a specific shape of Tetris, so we can make sure that different shapes of Tetris are generated each time.

```
if (Game_State_Counter == 0) begin
	case (Random_Counter)
		0: AP_State_PS <= PS_O_1;
		1: AP_State_PS <= PS_I_1;
		2: AP_State_PS <= PS_S_1;
		3: AP_State_PS <= PS_Z_1;
		4: AP_State_PS <= PS_L_1;
		5: AP_State_PS <= PS_J_1;
		default: AP_State_PS <= PS_T_1;
	endcase
	AP_State_SA <= BEGINNING_SA;
end
```

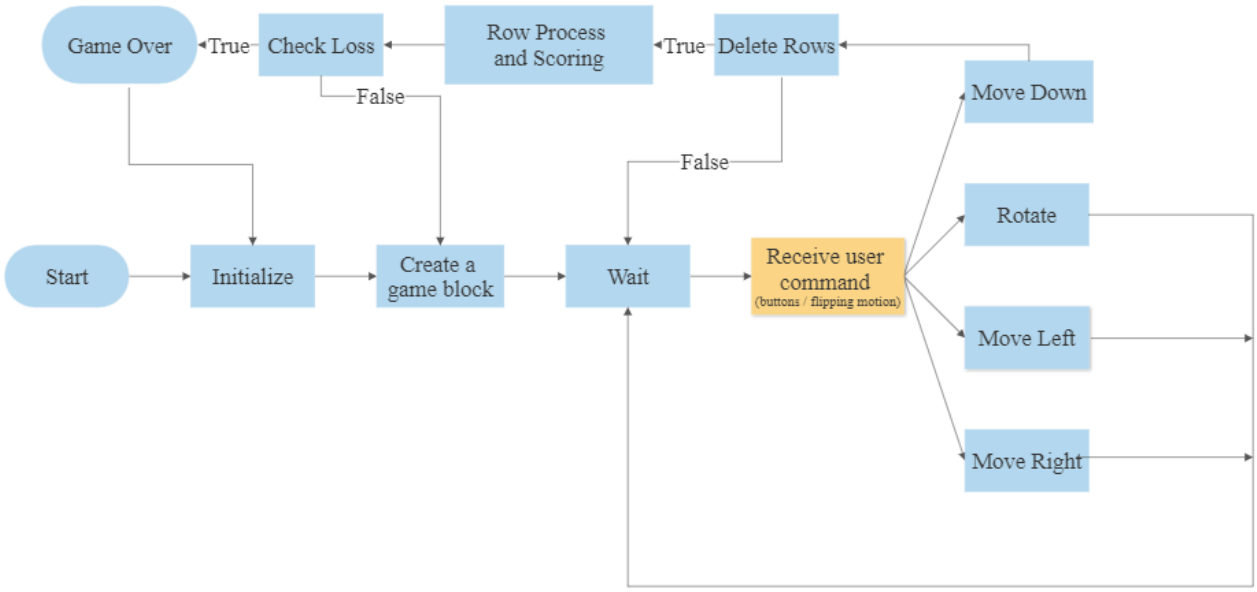
Main game state machine

We implement a main game state machine using a case statement. It handles various game states, such as starting a new game, initializing the game, generating a new active piece, handling user input, moving and rotating the active piece, and checking for game over conditions. The state transitions and operations depend on the current game state and other control variables.

Below are different states that we defined for the whole game. There are a total 13 states, including start and initialize operations. Besides, there are also MoveLeft and MoveRight states corresponding to the input Btn_Left and Btn_Right. With all of these states, we can realize the Tetris game with rocker features.

```
S_Start = 0,        // The very first state - wait for user to start a new game
S_Initialize = 1,   // this state initializes a new game
S_GenerateNewPiece = 2,    // Generate a new active piece
S_Idle = 3,                // main game state. Accepts tick event or user input
S_MoveLeft = 4,     // increments active piece shift amount
S_MoveRight = 5,    // decrements active piece shift amount
S_Spin = 6,         // spins piece by changing Active Piece PS
S_SpinCorrection = 7,      // Corrects active piece position after rotating piece
S_Wait = 8,         // just a wait state for debouncing input
S_Tick = 9,  // process a game tick (check for piece settling/dropping piece 1 row)
S_CleanFullRows = 10,            // clean all the complete rows
S_CleanEmptyRows = 11,     // remove empty rows and shift floating rows down
S_CheckLoss = 12,                // Check if the user has lost
S_GameOver = 13;    // Game over state - stop and wait for user to start new game
```

Below is the flowchart of how our Tetris works.



part2: accelerometer(IMU)

In this part, we will capture the accelerometer's data on x, y, z axis and determine whether the board is turning left, right or down.

The Nexys4 DDR board includes an ADXL362 accelerometer, which is a low-power 3-axis MEMS accelerometer. The accelerometer provides 12-bit output resolution and also offers 8-bit formatted data for more efficient single-byte transfers when lower resolution is sufficient. And the FPGA communicates with the ADXL362 via the **SPI interface**.

SPI Write to Put Sensor into Measurement Mode

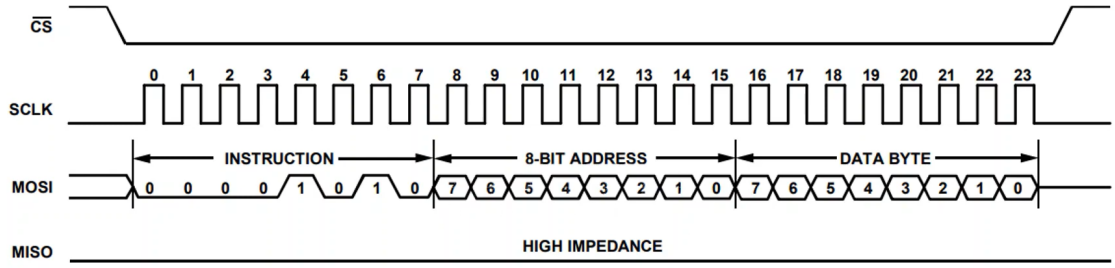


Figure 37. Register Write (Receive Instruction Only)

1. Wait at least 5ms after power-up
2. Send write instruction = 0x0A
3. Send register address 0x2D
4. Write byte 0x02 – turn on measurement mode
5. Go to reading data

Default data rate = 100Hz, period = 10ms

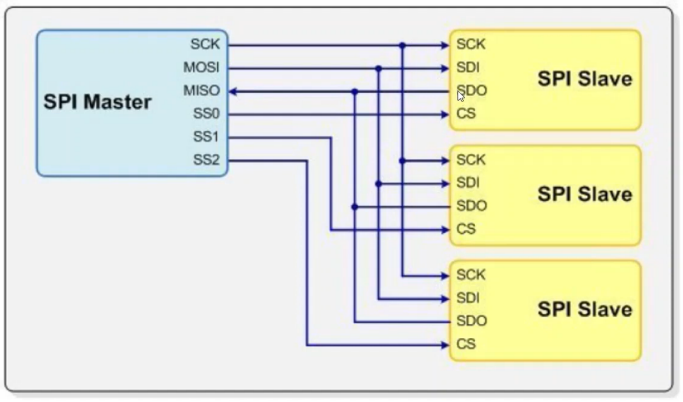
Measurement mode instruction to valid data = 40ms

Sensor defaults to standby mode 5ms after power up

SPI (Serial Peripheral Interface) is a synchronous serial communication protocol commonly used for communication between microcontrollers/microprocessors and peripheral devices. It allows for the exchange of data between a master device and one or more slave devices over short distances.

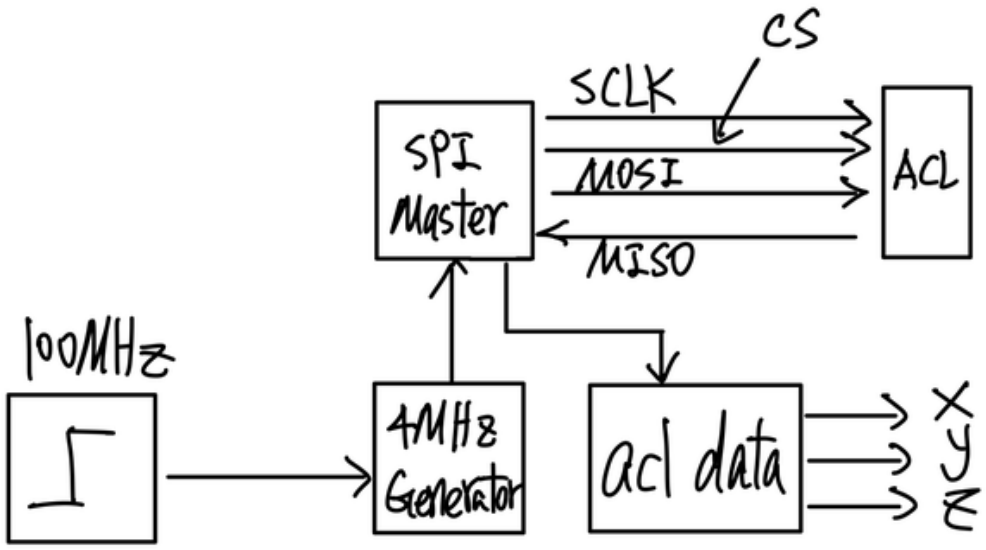
What is SPI?

Serial Peripheral Interface



https://www.corelis.com/education/tutorials/spi-tutorial/

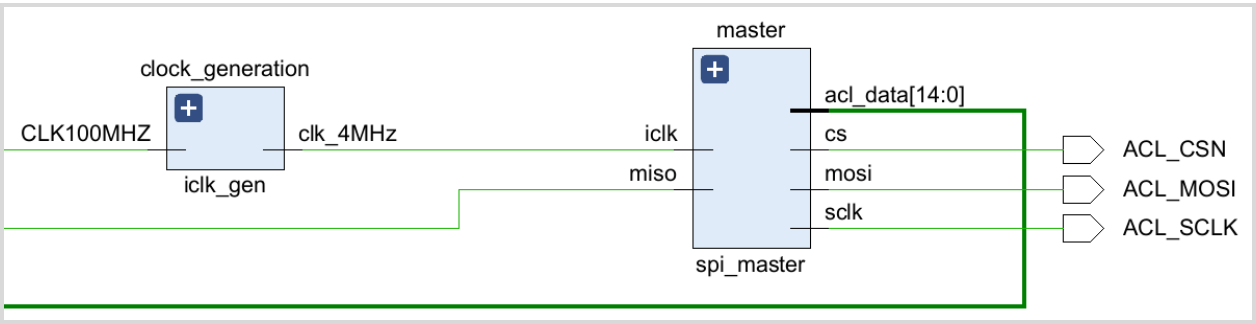
In Measurement Mode, the ADXL362 continuously measures and stores acceleration data in X-data, Y-data, and Z-data registers. The interface between the FPGA and accelerometer follows an SPI communication scheme, with a recommended clock frequency ranging from 1 MHz to 5 MHz(we choose 4MHz). The SPI operates in SPI mode 0 with CPOL = 0 and CPHA = 0. Configuration of the accelerometer is done by writing to control registers, and accelerometer data can be accessed by reading the device registers. And we can get the data of the x, y, z axis.



Below are the IOs for the accelerometer module:

```
input iclk; // 4Mhz clock
input miso; // master in
output sclk;//1MHz
output reg mosi = 1'b0; // master out
output reg cs = 1'b1;    // slave chip select
output [14:0] acl_data;    // 15 bit data, 5 each axis
```

IOs	fucntion
iclk	The internal clock of SPI master
miso, mosi	master in and out for SPI
sclk	sclk is the clock signal generated by the SPI master to synchronize data transfers.
acl_data	x, y, z axis data of accelerometer




```
reg [7:0] write_instr   = 8'h0A; // Sensor write instruction
reg [7:0] mode_reg_addr = 8'h2D; // Mode register address
reg [7:0] mode_wr_data  = 8'h02; // Data to write to mode register
reg [7:0] read_instr    = 8'h0B; // Sensor read instruction
reg [7:0] x_LSB_addr    = 8'h0E; // X data LSB address, auto-increments to
following 5 data registers
reg [14:0] temp_DATA = 15'b0;    // 15 bits, 5 bits for each axis
reg [15:0] X = 15'b0;                  // X data MSB and LSB from sensor
reg [15:0] Y = 15'b0;                  // Y data MSB and LSB from sensor
reg [15:0] Z = 15'b0;            // Z data MSB and LSB from sensor
reg [31:0] counter = 32'b0; // State machine sync counter 4MHz
wire latch_data;
```

And the process done by our code in SPI_master.v are shown below.

1. Initialization:
   - Initialize FPGA components, including GPIO (General Purpose Input/Output) pins and SPI (Serial Peripheral Interface) controller.
   - Configure SPI controller settings such as clock frequency, data order, and mode.

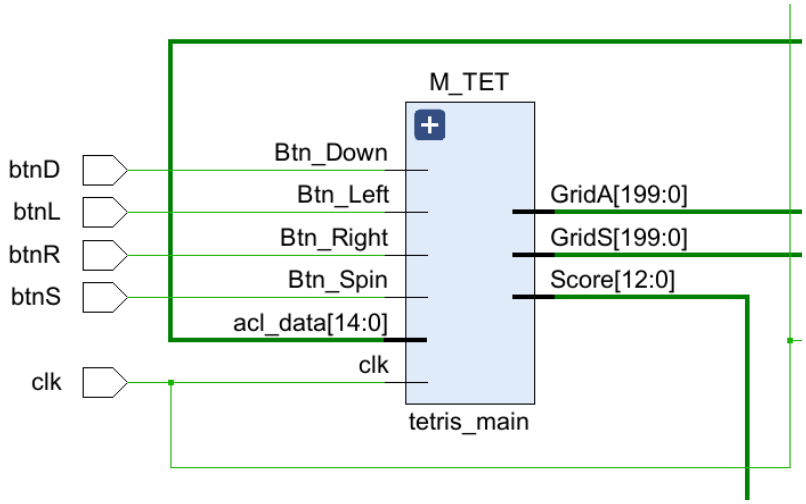
2. Configure Accelerometer:
   - Set up the SPI communication with the accelerometer.
   - Send configuration commands and settings to the accelerometer via SPI to configure its operating mode, range, and resolution.

3. Read Accelerometer Data:
   - Activate the SPI controller and select the accelerometer by asserting the chip select (CS) line.
   - Send a read command to the accelerometer to initiate data transfer.
   - Receive data bits from the accelerometer through the SPI controller.
   - Store received data in FPGA registers or memory for further processing.

4. Data Processing:
   - Perform necessary calculations or transformations on the received raw accelerometer data.
   - Apply filtering, scaling, or calibration techniques as required.
   - Convert the processed data to meaningful units (e.g., acceleration in g-force).

5. Utilize Accelerometer Data:
   - Utilize the processed accelerometer data for various purposes, such as:
     - Real-time monitoring of acceleration or vibration levels.
     - Implementing motion detection or gesture recognition algorithms.
     - Controlling other system components based on acceleration input.

6. Loop:
   - Repeat the above steps periodically or based on a defined sampling rate.
   - Continuously read and process accelerometer data in real-time.



After we collect the data from the accelerometer (acl_data), we will determine the data of the x and y axis to check if the board is turned left, right or down. Then we can control the tetris by our FPGA board as a Gamepad Joystick.

```
S_Idle: begin
        if (Btn_Spin) begin
          Game_State_Counter <= 1'b0;
          Game_State <= S_Spin;
        end
        else if (Btn_Left || (y_sign==0 && y_1>=3)) begin
          if ((Column_Left == 0) && (({1'b0,Grid_Settled[199:1]} & Grid_Active[229:30])==0))
            Game_State <= S_MoveLeft;
        end
        else if (Btn_Right  || (y_sign==1 && y_1>=4)) begin
          if ((Column_Right == 0) && (({Grid_Settled[198:0],1'b0} & Grid_Active[229:30])==0))
            Game_State <= S_MoveRight;
        end
        else if (Game_Tick || Btn_Down || (x_sign==0 && x_1>=5)) begin
          Game_State_Counter <= 1'b0;
          Game_State <= S_Tick;
        end
      end
```

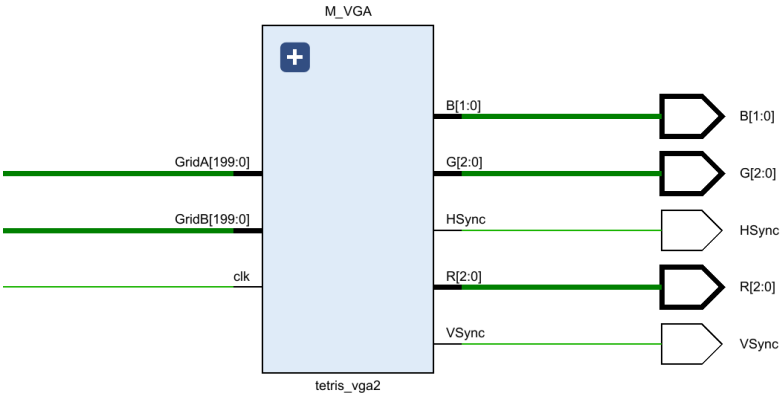
part3: VGA display

Below are the IOs for the VGA display module:

```
input clk; // 100Mhz clock
input [199:0] GridA; // blue grid
input [199:0] GridB; // red grid
output reg HSync;
output reg VSync;
output wire [2:0] R;
output wire [2:0] G;
output wire [1:0] B;
```

IOs	fucntion
clk	built-in input with a frequency of MHz.
GridA, GridB	Input ports representing the blue and color changing grid respectively.
HSync, Vsync	Output ports for horizontal and vertical synchronization signals.
R, G, B	Output ports for the red, green, and blue color components.

The VGA module in our Tetris game is responsible for displaying the data onto the connected LCD monitor. “HSynch” and “VSync” signals are pulses used for the start of the horizontal scan line and vertical scan line of the monitor.



The display is 640 by 480-pixel grid locations. The internal state comprises a 200-bit value that represents the arrangement of settled pieces and a 230-bit value that represents the grid of active pieces. When the active piece is shifted (left, right, down), the shift amount is adjusted accordingly. To bring a piece to the bottom of the VGA screen, the 200 most significant bits of

the active piece grid are combined with the settled piece grid using the bitwise OR operator. Collision detection can be performed using the bitwise AND operator.

The module defines various color parameters for different elements of the game. These colors are represented using 8 bits and are defined as binary value

```
parameter             COLOR_BACKGROUND = 8'b000_000_00,
                      COLOR_BORDER     = 8'b111_111_11,
                      COLOR_GRIDA      = 8'b000_000_11,
                      COLOR_GRIDB0     = 8'b000_110_00,
                      COLOR_GRIDB1     = 8'b111_000_00,
                      COLOR_GRIDB2     = 8'b000_110_10,
                      COLOR_GRIDB3     = 8'b011_000_10,
                      COLOR_GRIDB4     = 8'b100_100_10,
                      COLOR_GRIDB5     = 8'b001_011_10,
                      COLOR_BOTH       = 8'b111_000_00;
```

The clock_divider variable is a 2-bit register that increments on every positive edge of the clk input. The least significant bit of clock_divider is assigned to the pxl_clk wire.

```
reg [1:0] clock_divider;
      wire pxl_clk = clock_divider[1];

      always @(posedge clk) begin
            clock_divider <= clock_divider + 1'b1;
      end
```

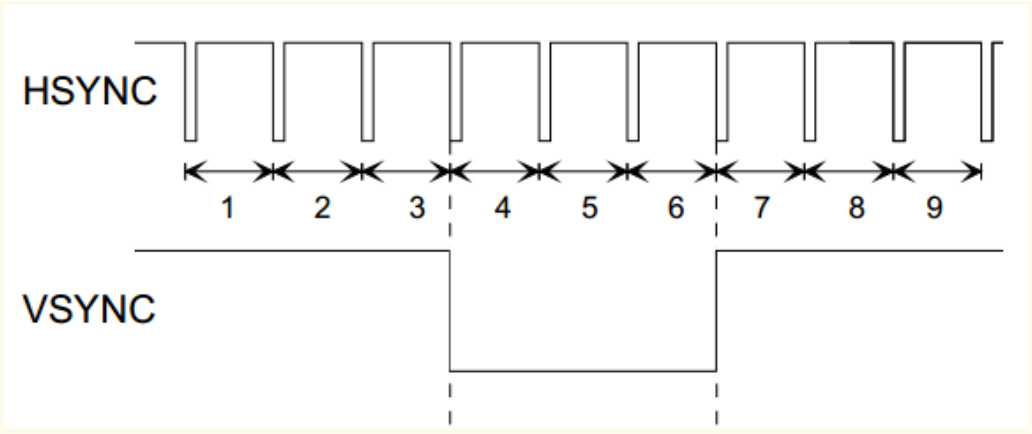
The module defines two 10-bit registers, HorizontalCounter and VerticalCounter, which are used to keep track of the horizontal and vertical positions of the display. The module contains logic to draw different parts of the game area on the VGA display. It uses the HorizontalCounter and VerticalCounter values to determine the current position and apply specific colors accordingly.

```
reg [9:0] HorizontalCounter;
reg [9:0] VerticalCounter;

always @(posedge pxl_clk) begin
            if (HorizontalCounter < 799)
                  HorizontalCounter <= HorizontalCounter + 1'b1;
            else
                  HorizontalCounter <= 0;
      end
      // HSync is active for 96 clock pulses
      always @(posedge pxl_clk) begin
```

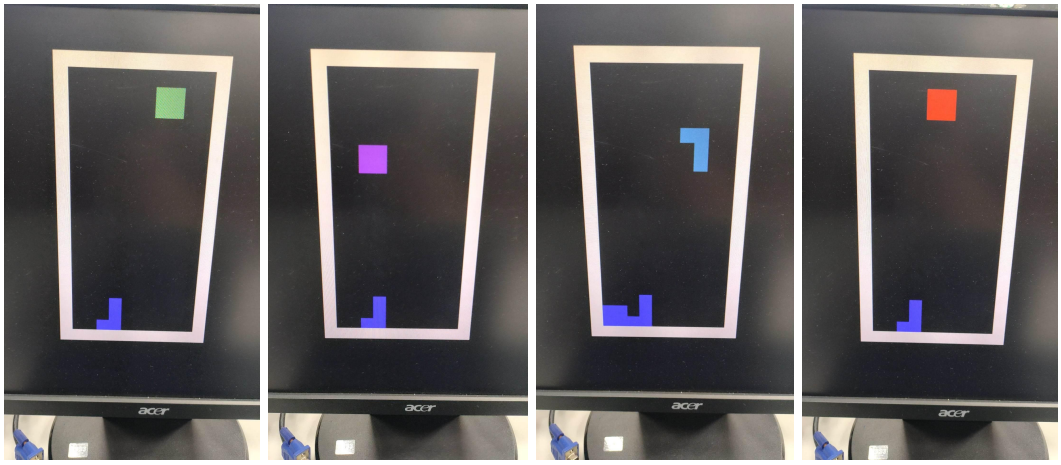
```
        if (HorizontalCounter < 96)
                HSync <= 1'b0;
        else
                HSync <= 1'b1;
end
// VerticalCounter should count HSync pulses from 0 to 524
always @(negedge HSync) begin
        if (VerticalCounter < 524)
                VerticalCounter <= VerticalCounter + 1'b1;
        else
                VerticalCounter <= 0;
end

// VSync is active for 2 lines
always @(posedge pxl_clk) begin
        if (VerticalCounter < 2)
                VSync <= 1'b0;
        else
                VSync <= 1'b1;
end
```



Inside the "Draw game" section, a case statement is used based on the value of the num variable and ShiftedHorizontalCounter. This determines the specific color assigned to the RGB signal. The color values are based on the values of the GridA and GridB inputs and the predefined colors in the parameters section.

Also, in order to realize the effect of the color changing Tetris, we used several cases to change the color of the Tetris while it's falling. The "num" variable will change according to specific frequency and therefore the color will change respectively, as we shown below.



```
case(num)
    0:begin
      case (ShiftedHorizontalCounter)
                            SHIFTED_HGAME_START:     RGB <= (a[9] & b[9]) ?
COLOR_BOTH : (a[9]) ? COLOR_GRIDA : (b[9]) ? COLOR_GRIDB0 : COLOR_BACKGROUND;
                            SHIFTED_HGAME_START + 1: RGB <= (a[8] & b[8]) ?
COLOR_BOTH : (a[8]) ? COLOR_GRIDA : (b[8]) ? COLOR_GRIDB0 : COLOR_BACKGROUND;
                            SHIFTED_HGAME_START + 2: RGB <= (a[7] & b[7]) ?
COLOR_BOTH : (a[7]) ? COLOR_GRIDA : (b[7]) ? COLOR_GRIDB0 : COLOR_BACKGROUND;
                            SHIFTED_HGAME_START + 3: RGB <= (a[6] & b[6]) ?
COLOR_BOTH : (a[6]) ? COLOR_GRIDA : (b[6]) ? COLOR_GRIDB0 : COLOR_BACKGROUND;
                            SHIFTED_HGAME_START + 4: RGB <= (a[5] & b[5]) ?
COLOR_BOTH : (a[5]) ? COLOR_GRIDA : (b[5]) ? COLOR_GRIDB0 : COLOR_BACKGROUND;
                            SHIFTED_HGAME_START + 5: RGB <= (a[4] & b[4]) ?
COLOR_BOTH : (a[4]) ? COLOR_GRIDA : (b[4]) ? COLOR_GRIDB0 : COLOR_BACKGROUND;
                            SHIFTED_HGAME_START + 6: RGB <= (a[3] & b[3]) ?
COLOR_BOTH : (a[3]) ? COLOR_GRIDA : (b[3]) ? COLOR_GRIDB0 : COLOR_BACKGROUND;
                            SHIFTED_HGAME_START + 7: RGB <= (a[2] & b[2]) ?
COLOR_BOTH : (a[2]) ? COLOR_GRIDA : (b[2]) ? COLOR_GRIDB0 : COLOR_BACKGROUND;
                            SHIFTED_HGAME_START + 8: RGB <= (a[1] & b[1]) ?
COLOR_BOTH : (a[1]) ? COLOR_GRIDA : (b[1]) ? COLOR_GRIDB0 : COLOR_BACKGROUND;
                            SHIFTED_HGAME_START + 9: RGB <= (a[0] & b[0]) ?
COLOR_BOTH : (a[0]) ? COLOR_GRIDA : (b[0]) ? COLOR_GRIDB0 : COLOR_BACKGROUND;
     default:                RGB <= COLOR_BACKGROUND;
       endcase
       end
```

Remark

- **completeness**

  We have developed a unique version of Tetris that incorporates tilting controls, allowing users to play the game with just one button while manipulating the board by tilting it. Additionally, we have added a dynamic color feature where the Tetris pieces change color as they fall. Once a piece touches the bottom, it transitions to blue, signaling that it can no longer be moved. With these innovations, we have introduced a fresh and exciting playing method to Tetris.

- **limits and future prospect**

  We have the capability to further enhance the score counting system by designing it in a more detailed and comprehensive manner. Additionally, we can introduce a feature that provides the player with a visual indication of where the Tetris piece will land. Furthermore, we can implement a function that displays the upcoming Tetris shapes, giving the player a preview of what to expect. These additions will enhance the gameplay experience and provide players with more strategic information.

- **demo**

  Our demo video link:
  https://www.youtube.com/watch?v=sbpdBV9Sn2o&ab_channel=HAC

- **thoughts**

  Completing the FPGA Tetris project has been an immensely fulfilling experience, especially considering that it was our first foray into FPGA development. Despite only having a few months of experience, we managed to successfully design and implement the game. Debugging and testing proved to be arduous yet crucial in achieving the desired level of accuracy.

  Utilizing online resources and studying example code greatly aided us in recreating the classic Tetris game. However, we wanted to go beyond replication and make our game stand out. During our research, we stumbled upon the presence of an accelerometer in the FPGA. This discovery sparked our imagination, and after extensive searching and investigation, we uncovered the key to turning the FPGA into a Gamepad Joystick for playing Tetris. Regrettably, we were unable to complete additional functions of the original Tetris game, which led to a slight sense of disappointment.

  Nevertheless, this project significantly expanded our knowledge of hardware-software co-design and optimization techniques, elevating our problem-solving skills to new heights. The experience has also ignited a deep curiosity within us to explore more intricate FPGA-based game designs and delve deeper into the realm of hardware-accelerated computing.

Overall, this project exemplifies the limitless possibilities of FPGA projects and underscores the potential for innovation within the realm of digital systems. It has motivated us to continue exploring and pushing the boundaries of FPGA development, driven by the desire to create innovative and captivating experiences.

Resources

https://ieeexplore.ieee.org/document/6201435

https://github.com/Digilent/digilent-xdc/blob/master/Nexys-4-DDR-Master.xdc

https://github.com/FPGADude/Digital-Design/tree/main/FPGA%20Projects/Nexys%20A7%203-Axis%20Accelerometer%20SPI

https://vanhunteradams.com/DE1/VGA_Driver/Driver.html