Touch Pong

CSEE 4840 Embedded System Design
Final Report

Xiang Zhou	xz2266
Hao Zheng	hz2256
Ran Zheng	rz2228
Younggyun Cho	yc2704

Contents

1 Abstract

This project is conducted using the Altera DE2 development board. We are aiming at implementing a touch-screen ping pong game. It will be a player vs. player game with a specified rule. Player serves and receives the ball by touching the screen connected to DE2 board. To do so, we need to set up the interface between touch screen and DE2 board.

2 Introduction

The Ping Pong game is an extension of the real Ping Pong game. We set a few new rules for the game for example, the ball can bounce on the two horizontal sides of the screen and once the ball hits the perpendicular side of the screen, the game is over. In terms of movement of the bat, it can move in 2D screen by following the moving trajectory of the hand on the touch screen. The horizontal rebound velocity of the ball depends on the direction of the moving racket when batting occurs.

To implement the Ping pong game, the project will involve both hardware set up and software programming. Especially, due to the control of the touch screen, the hardware set up will take the most of the work.

For the hardware part, the major workload is to set up the touch screen and interface. Moreover, the display of the game graphics can also take some efforts.

For the software part, the difficulty lies in how we realize the algorithm of the Ping pong Game. What's more, we need add interruption to transmit the coordinate of racket and ball.

3 Architecture

In this project, there are two major hardware devices: FPGA board and LTM touch screen.

Incorporate VGA display with the TRDB_LTM Kit to develop the application using a digital touch panel on an Altera DE2 board.

- VHDL (compiled with Quartus 7.2 and Nios II) will be used for the inter-connections of hardware.
- C will be also employed to handle the hardware implementation.
- The Terasic LCD Touch Panel Module (LTM) board is a displayer and a controller.

- A 40-pin IDE cable will be used for connecting between the LTM and the DE2 board.

The figure below is the block diagram of the Touch Screen Processor architecture:

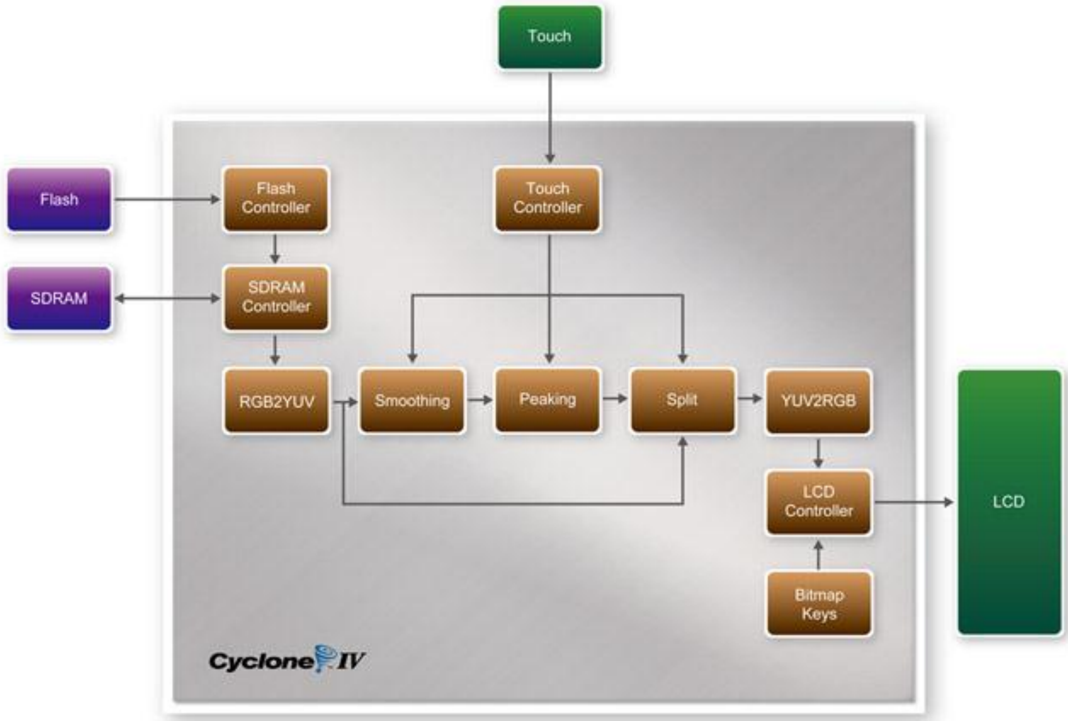


Figure 3-1 Touch Screen Processor

The figure below shows the block diagram of the photo demonstration:

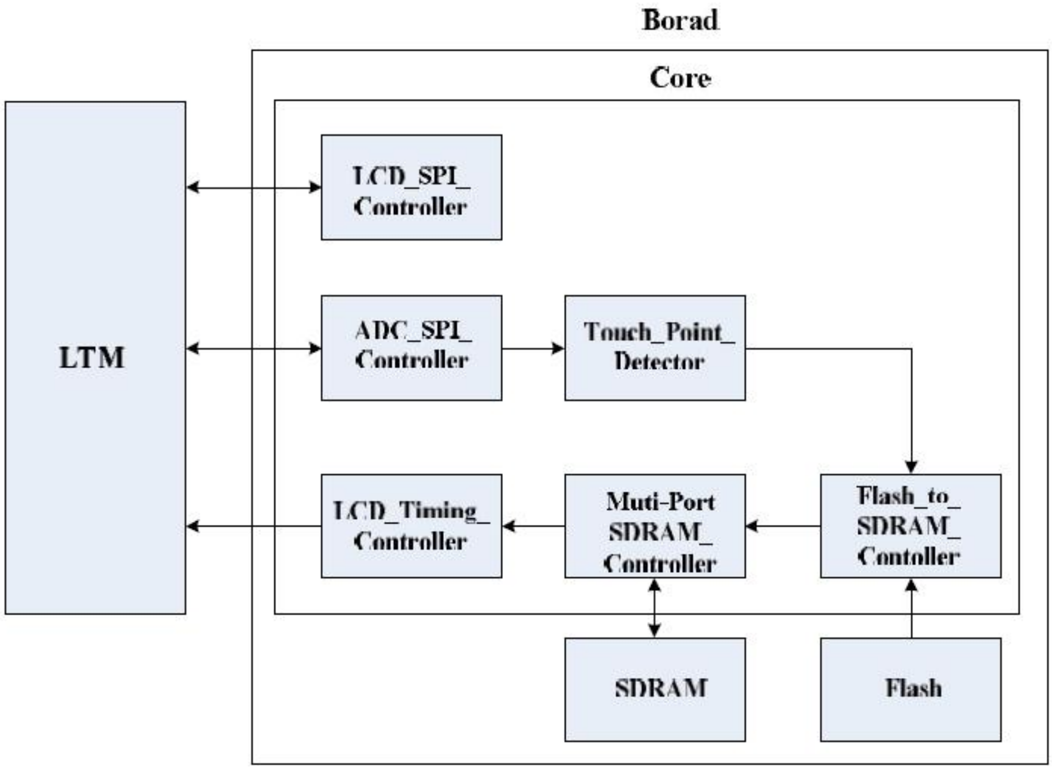


Figure 3-2 block diagram of the photo demonstration

As soon as the bit stream is downloaded into the FPGA, the register values of the LCD driver IC using to control the LCD display function will be configured by the LCD_SPI_Controller block, which uses the serial port interface to communicate with the LCD driver IC. Meanwhile, the Flash_to_SDRAM_Controller block will read the RGB data of one picture stored in the Flash, and then write the data into SDRAM buffer. Accordingly, both the synchronous control signals and the picture data stored in the SDRAM buffer will be sent to the LTM via the LCD_Timing_Controller block.

When users touch LTM screens, the x and y coordinates of the touch point will be obtained by the ADC_SPI_Controller block through the ADC serial port interface. Then the Touch_Point_Detector block will determine whether these coordinates are in a specific range. If the coordinates fit the range, the Touch_Point_Detector block will control the Flash_to_SDRAM_Controller block to read the next or previous picture's data from the Flash and repeat the steps as mentioned before to command the LTM to display the next or previous picture.

The block diagram of the system is listed below:

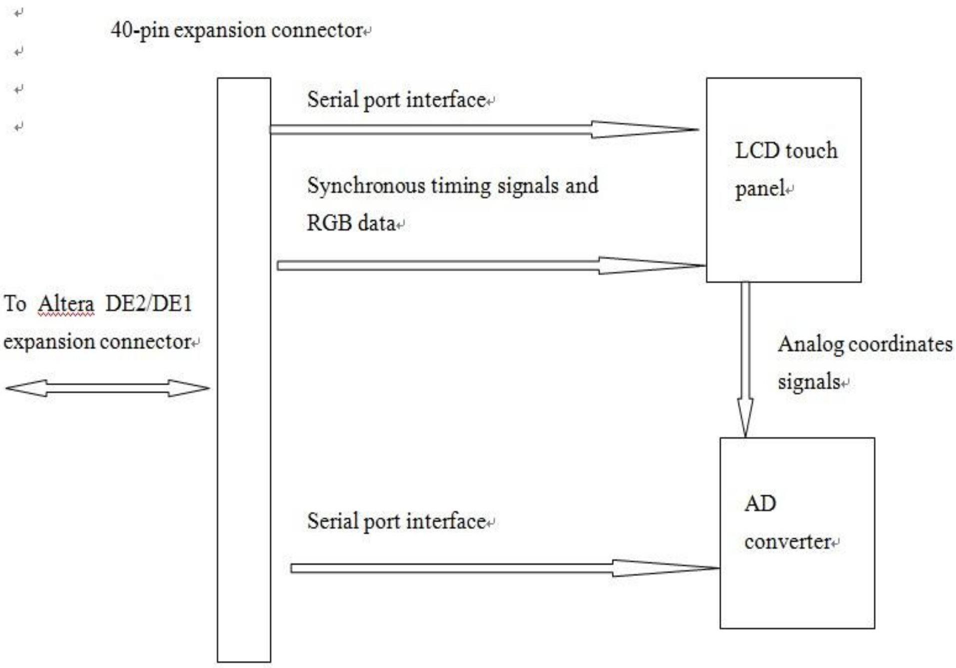


Figure 3-3 System Structure

The LTM consists of three major components: LCD touch panel module, AD converter, and 40-pin expansion header. All of the interfaces on the LTM are connected to Altera DE2 board via the 40-pin expansion connector. The LCD and touch panel module will take the control signals provided directly from FPGA as input and display images on the LCD panel. Finally, the AD converter will convert the coordinates of the touch point to its corresponding digital data and output to the FPGA via the expansion header.

The feature set of the LTM is listed below:

1. Equipped with Toppoly TD043MTEA1 active matrix color TFT LCD module.
2. Support 24-bit parallel RGB interface.
3. 3-wire register control for display and function selection.
4. Built-in contrast, brightness, and gamma modulation.
5. Converting the X/Y coordination of the touch point to its corresponding digital data via the Analog Devices AD7843 AD converter.
6. The general specifications of the LTM are listed below:

Item	Description	Unit
Display Size (Diagonal)	4.3	Inch
Aspect ratio	15:9	-
Display Type	Transmissive	-
Active Area (HxV)	93.6 x 56.16	mm
Number of Dots (HxV)	800 x RGB x480	dot
Dot Pitch (HxV)	0.039 x 0.117	mm
Color Arrangement	Stripe	-
Color Numbers	16Million	-

Table 3-1 general specifications of the LTM

4 Design

4.1 Game Logic

4.1.1 Introduction

The game is an extension of the real Ping Pong game. There are two players fighting against each other. Players would use their finger to control ping pong bat through touching on the screen and the racket in the game would move along with movement of the touching trace. The ball would bounce when hitting the upper and down wall or the rackets just like the "bouncing ball" in lab3, while when the ball hit the left and right side of the wall, that round of game will be over, and the ball and bats would get back to the default position.

4.2.2 Playing rules

1. In order to be fair for both players, the players would serve alternately by touching any point of the panel, and the initial moving angle of the ball would be 45 degree.

2. The player can only move their rackets in his own half side of the table and players have several chances to hit the ball before the ball runs out of the boundary.

3. Racket can only be moved when the ball get into the corresponding side of the table.

4. The one who misses the ball through letting it run out of the boundary in his own side would lose that round, and the opposite side would gain one point

5. The one who gains 11 points first would win the whole game.

6. If the players want to continue playing, they just need to touch the left corner of the panel, and the score would be set to zero.

4.2 Hardware

4.2.1 LTM Controller

The LCD and touch panel module on the LTM is equipped with a LCD driver IC to support three display resolutions and with functions of source driver, serial port interface, timing controller, and power supply circuits. To control these functions, users can use FPGA to configure the registers in the LCD driver IC via serial port interface.

Also, there is an analog to digital converter (ADC) on the LTM to convert the analog X/Y coordinates of the touch point to digital data and output to FPGA through the serial port interface of the ADC. Both LCD driver IC and ADC serial port interfaces are connected to the FPGA via the 40-pin expansion header and IDE cable.

Because of the limited number of I/O on the expansion header, the serial interfaces of the LCD driver IC and ADC need to share the same clock (ADC_DCLK) and chip enable (SCEN) signal I/O on the expansion header. To avoid both the serial port interfaces may interfere with each other when sharing the same clock and chip enable signals, the chip enable signal (CS), which is inputted into the ADC will come up with a logic inverter as shown in Figure 4-1. Users need to pay attention controlling the shared signals when designing the serial port interface controller. The detailed register maps of the LCD driver IC are listed in appendix chapter. The specifications of the serial port interface of the LCD driver IC are described below.

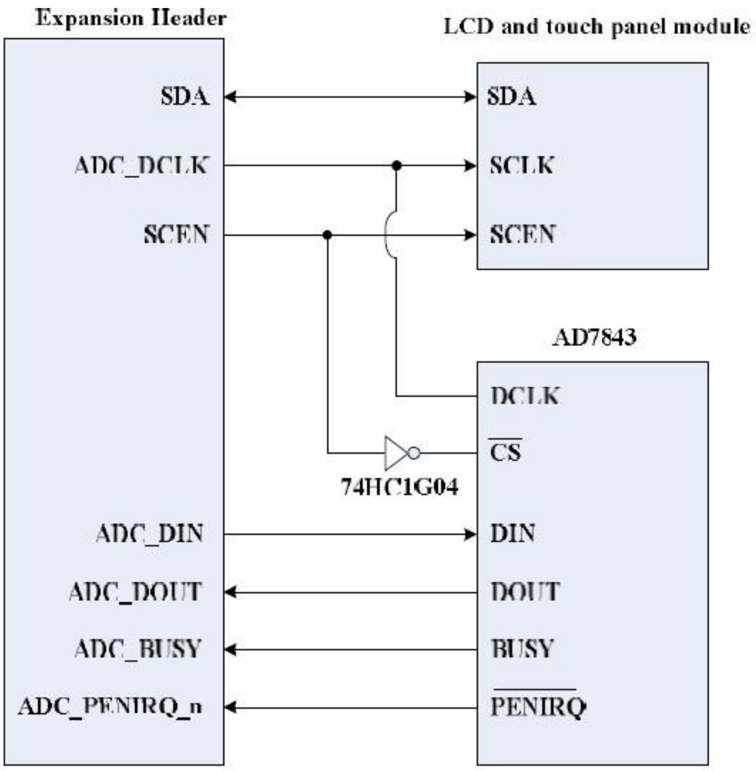


Figure4-1 Serial interface of the LCD touch panel module and AD7843

Timing Control

1 The Serial Port Interface of the LCD Driver IC

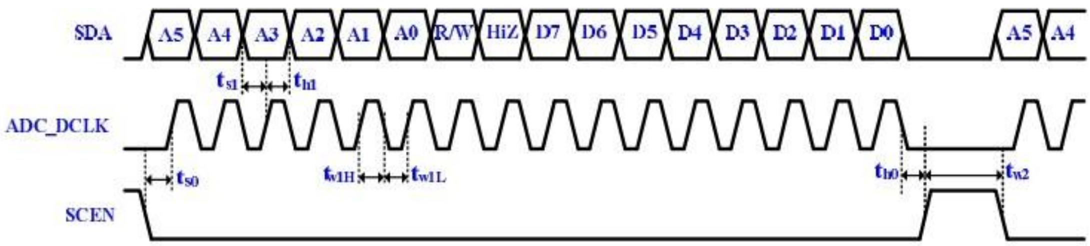


Figure4-2 Frame format and timing diagram of the serial port interface

The figure above shows the frame format and timing diagram of the serial port interface. The LCD driver IC recognizes the start of data transfer on the falling edge of SCEN input and starts data transfer. When setting instruction, theTPG110 inputs the setting values via SDA on the rising edge of input SCL.

The first 6 bits (A5~A0) specify the address of the register. The next bit means Read/Write command. “0” is write command. “1” is read command. Then, the next cycle is turn-round cycle. Finally, the last 8 bits are for Data setting (D7 ~ D0). The address and data are transferred from the MSB to LSB sequentially. The data is written to the register of assigned address when “End of transfer” is detected after the 16th SCL rising cycles. Data is not accepted if there are less or more than 16 cycles for one transaction.

2 Input timing of the LCD panel display function

This section will describe the timing specification of the LCD synchronous signals and RGB data.

Figure below illustrates the basic timing requirements for each row (horizontal) that is displayed on the LCD panel. An active-low pulse of specific duration (time t_{hpw} in the figure) is applied to the horizontal synchronization (HD) input of the LCD panel, which signifies the end of one row of data and the start of the next. The data (RGB) inputs on the LCD panel are not valid for a time period called the hsync back porch (t_{hbp}) after the hsync pulse occurs, which is followed by the display area (t_{hd}). During the data display area the RGB data drives each pixel in turn across the row being displayed. Also, during the period of the data display area, the data enable signal (DEN) must be driven to logic high. Finally, there is a time period called the hsync front porch (t_{hfp}) where the RGB signals are not valid again before the next hsync pulse can occur.

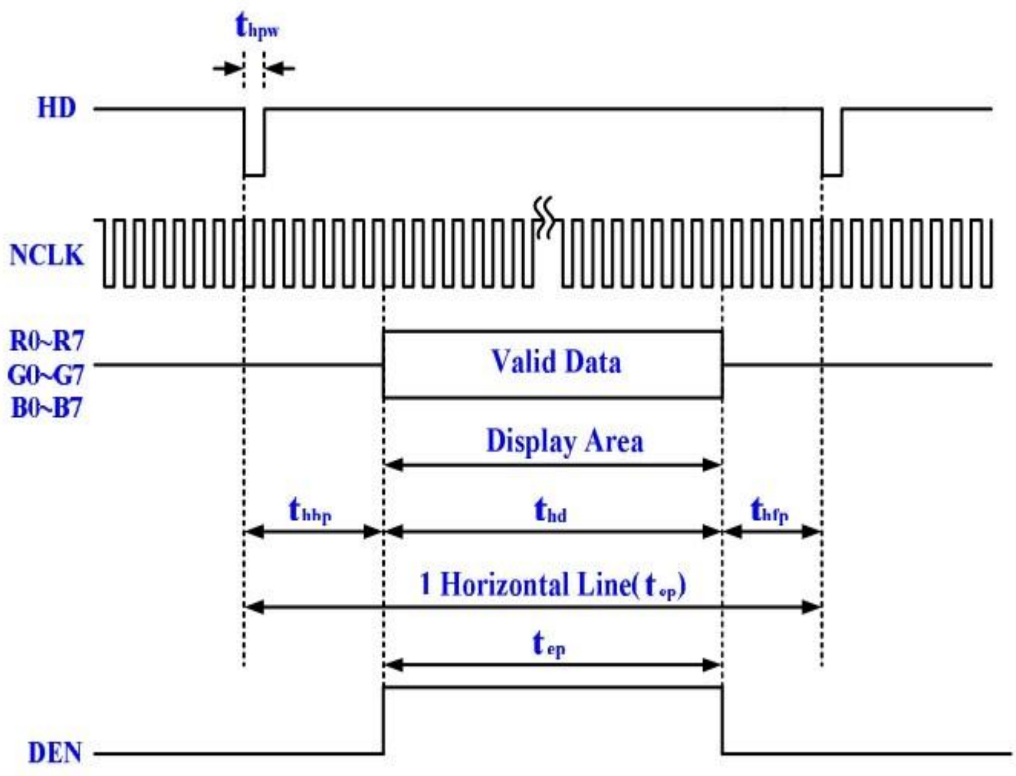


Figure4-3 LCD horizontal timing specification

The timing of the vertical synchronization (VD) is the same as shown in Figure 4-4, except that a vsync pulse signifies the end of one frame and the start of the next, and the data refers to the set of rows in the frame (horizontal timing). Tables 3.2 and 3.3 in reference (LTM_User_Manual) show for different resolutions, the durations of time periods t_{hpw}, t_{hbp}, t_{hd}, and t_{hfp} for both horizontal and vertical timing. Finally, the timing specification of the synchronous signals is shown in the Table 3.4.

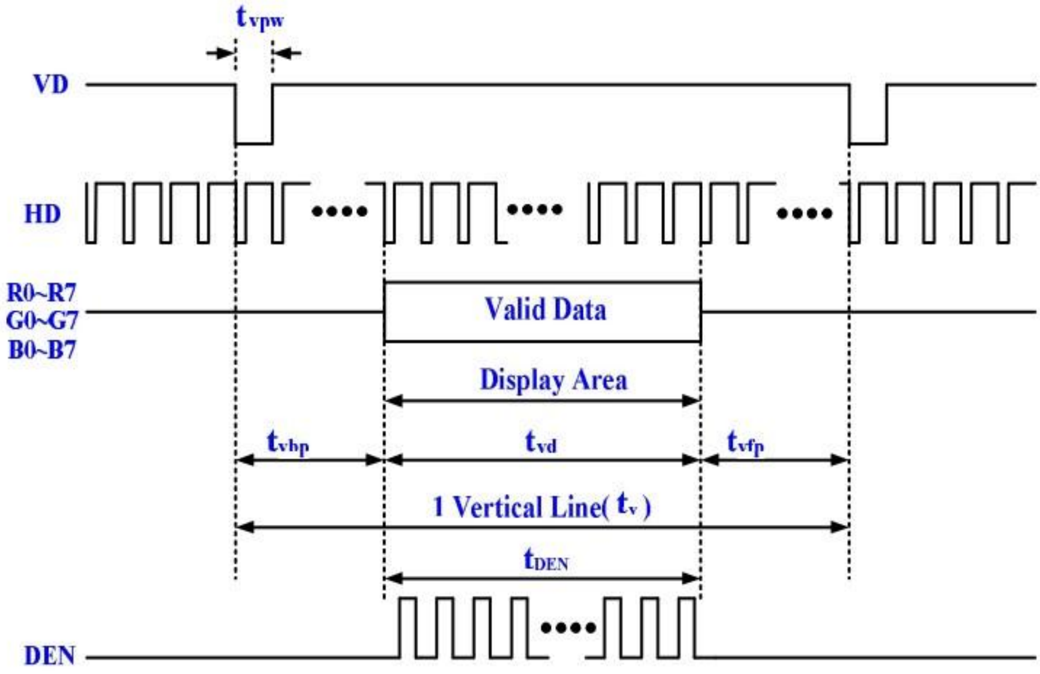


Figure4-4 LCD vertical timing specification

3 The serial interface of the AD converter

This section will describe how to obtain the X/Y coordinates of the touch point from the AD converter. The LTM also equipped with an Analog Devices AD7843 touch screen digitizer chip. The AD7843 is a 12-bit analog to digital converter (ADC) for digitizing x and y coordinates of touch points applied to the touch screen.

To obtain the coordinate from the ADC, the first thing users need to do is monitor the interrupt signal ADC_PENIRQ_n outputted from the ADC. By connecting a pull high resistor, the ADC_PENIRQ_n output remains high normally. When the touch screen connected to the ADC is touched via a pen or finger, the ADC_PENIRQ_n output goes low, initiating an interrupt to a FPGA that can then instruct a control word to be written to the ADC via the serial port interface. The control word provided to the ADC via the DIN pin is shown in reference.

The control word provided to the ADC via the DIN pin is shown in Table 3.5 in reference (LTM_User_Manual). This provides the conversion start, channel addressing, ADC conversion resolution, configuration, and power-down of the ADC. The detailed information on the order and description of these control bits can be found from the datasheet of the ADC in the DATASHEET folder on the LTM System CD-ROM.

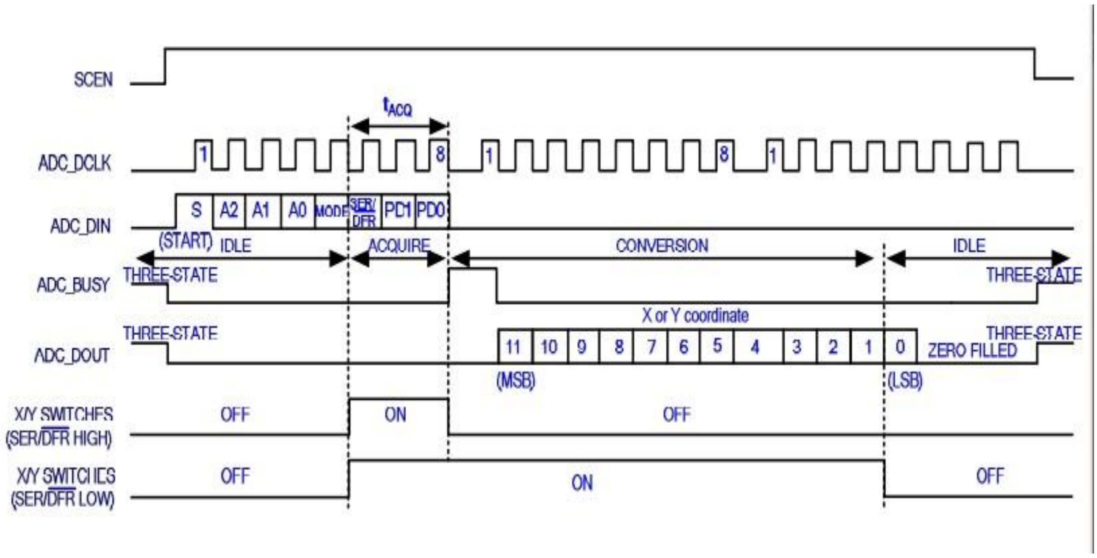


Figure4-5 Conversion timing of the serial port interface

Figure 4-5 shows the typical operation of the serial interface of the ADC. The serial clock provides the conversion clock and also controls the transfer of information to and from the ADC. One complete conversion can be achieved with 24 ADC_DCLK cycles. The detailed behavior of the serial port interface can be found in the datasheet of the ADC. Note that the clock (ADC_DCLK) and chip enable signals (SCEN) of the serial port interface SHRAE the same signal I/O with LCD driver IC. Users should avoid controlling the LCD driver IC and ADC at the same time when designing the serial port interface controller. Also, because the chip enable signal (SCEN) inputted to the ADC comes up with a logic inverter, the logic level of the SCEN should be inverse when it is used to control the ADC. ADC_DIN is pattern control signal of AD converter and ADC_DOUT is the coordinate of X or Y. Data can be transmitted when signal ADC_PENIRQ_n falls. ADC_BUSY controls the pattern of AD converter which enables to receive data when it keeps low.

4.2.2 DE2 Controller

4.2.2.1 Loading background into the Flash

1. Make sure the USB-Blaster download cable is connected into the host PC
2. Load the Control Panel bit stream (DE2_USB_API/ DE1_USB_API) into the FPGA. Please also refer to Chapter 3 DE2/DE1 Control Panel in the Altera DE2/DE1 User Manual for more details in the Control Panel Software
3. Execute the Control Panel application software
4. Open the USB port by clicking Open > Open USB Port 0. The DE2/DE1 Control Panel application will list all the USB ports that connect to DE2/DE1 board
5. Switch to FLASH page and click on the “Chip Erase(40 Sec)” bottom to erase Flash data

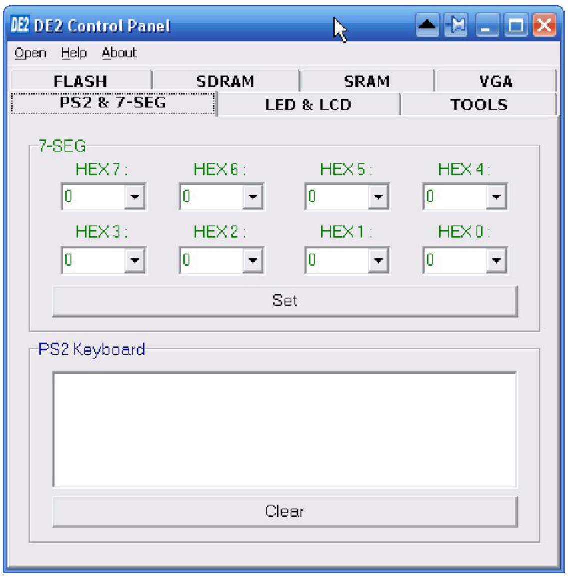


Figure4-6 Loading picture

6. Click on the “File Length” checkbox to indicate that you want to load the entire file

7. Click on the “Write a File to FLASH” bottom. When the Control Panel responds with the standard Windows dialog box and asks for the source file, select the “tab222_2.bmp” file in the “Photo” directory

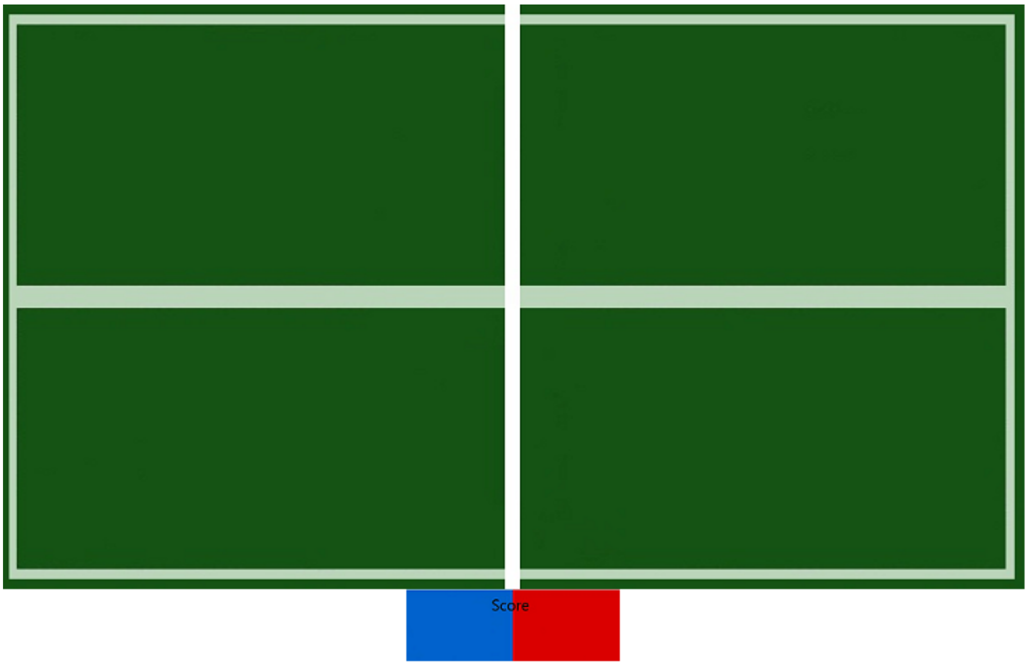


Figure4-7 Background

4.2.2.2. Generate mif file

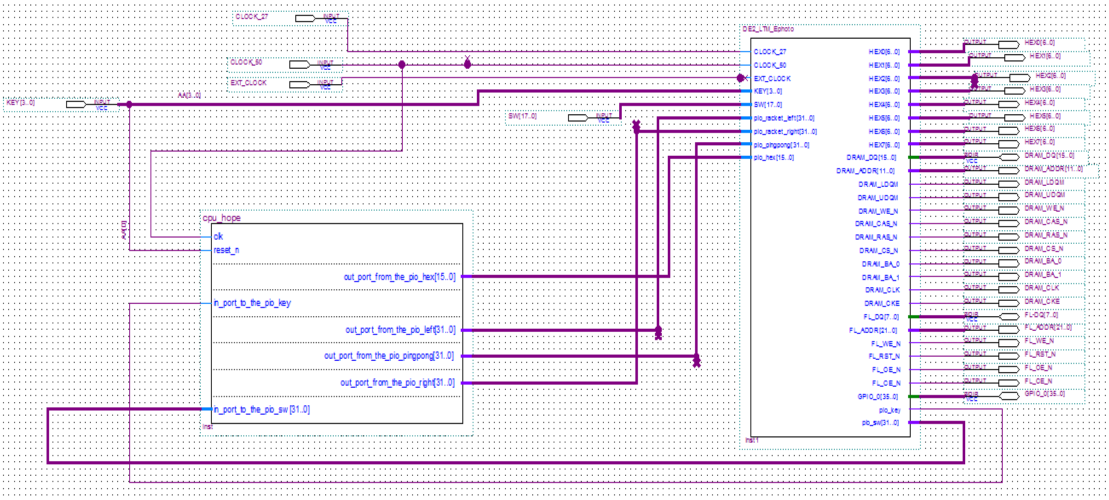
A memory Initialization File (.mif) is an ASCII text file (with the extension .mif) that specifies the initial content of a memory block (CAM, RAM, or ROM), that is, the initial values for each address. This file is used during Quartus project compilation and/or simulation.

The MIF file serves as an input file for memory initialization in the Quartus compiler and simulator. You can also use a Hexadecimal Intel-Format File (.hex) to provide memory initialization data.

MATLAB code:

```
Img=imread('PINGPONG.BMP');
BW = Img;
R=BW(:,:,1);
```

4.2.2.3 Generate block diagram of system

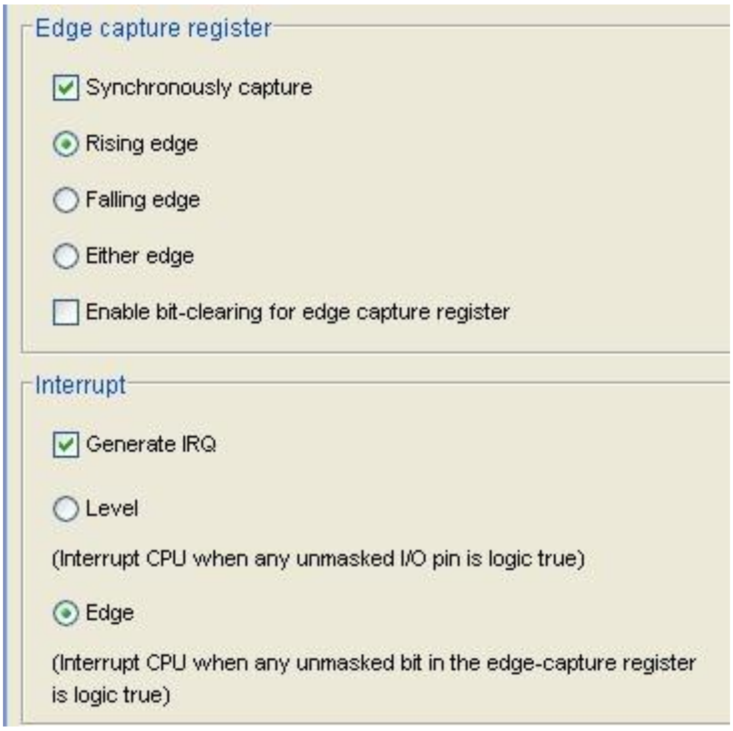


4.3 Software

In the whole project, the most important part is the FPGA and the touch panel part. Hence, we didn't put our main effort on the software part. Although we just designed the basic function of the gain, it still took loads of efforts. Our software part can be divided into two parts: interrupt part game control part.

4.3.1 Interruption Design

In order to acknowledge the touch on touch panel, we write this the interrupt code. Actually, there existing a transform mechanism in DE2, which can transfer the interrupt from the touch panel to the PIO interrupt. Thus, using interrupt from PIO ports is indirectly use interrupt from the touch panel, which makes the design work much easier. In this code, we referenced the interrupt of using key to control LED.



Interruption setting of PIO

edgecapture Register

Bit *n* in the edgecapture register is set to 1 whenever an edge is detected on input port *n*. An Avalon-MM master peripheral can read the edgecapture register to determine if an edge has occurred on any of the PIO input ports. If the option Enable bit-clearing for edge capture register is turned off, writing any value to the edgecapture register clears all bits in the register. Otherwise, writing a 1 to a particular bit in the register clears only that bit.

Edgecapture Register

The interrupt code can be divided into three parts. First part is KeyDown_interrupts. when the key is pressed, the function will set a flag to inform the outside code. Second part is InitPIO. One function of it is to initialize the PIO_KEY as the input and PIO_LED as output. Another function is opening interruption and clear the edge capturing register. The third part is main function. It is used to wait for key interrupt and output signal to control the LED. Their existing a "while" loop in this code to keep detecting if a interrupt is happening. What’s more, we can only use sprintf() to debug the interruption not printf().

4.3.2 Game Control Design

The basic purpose for the c code is control the movement of the ball and rackets, thereby realizing the rule of the game.

Firstly, we show all the important parameters in this paper:

volatile alt_u32 flag // Indicate the beginning and end of the whole game
volatile alt_u32 x_y_pingpong // pingpong's coordinate for transmitting
volatile alt_u32 x_pingpong //pingpoing's x axis coordinate

```
volatile alt_u32 y_pingpong    //pingpoing's y axis coordinate
volatile alt_u32 x_count         // direction and speed of the ball in x axis
volatile alt_u32 y_count        // direction and speed of the ball in y axis
volatile alt_u32 center_x      // ball default value in x axis
volatile alt_u32 center_y      // ball default value in y axis
volatile alt_u32 count_l        // score of the left side
volatile alt_u32 count_r        // score of the right side
volatile alt_u32 right_x        // x axis position of right racket
volatile alt_u32 right_y        // y axis position of right racket
volatile alt_u32 left_x         // x axis position of left racket
volatile alt_u32 reft_y         // y axis position of left racket
volatile alt_u32 ltm_x          // x axis position of either racket
volatile alt_u32 ltm_y          // y axis position of either racket
volatile alt_u32 ltm_y_x      // rackets' coordinates get from Verilog code
alt_busy_sleep()                  // delay for the ball
```

The first thing code should do is to get the coordinates of both the ball and the rackets. Already do the signal transformation in the Verilog part, we can just use the coordinates of x_ y_pingpong and ltm_y_x. But the format of these coordinates which is 20 bits long with the x and y combined together is different from the nomal coordinates.

```
x_ y_pingpong=y_pingpong*2048+x_pingpong;
ltm_y_x=ltm_y*2048+ltm_x;
```

By using the equations above, we can conveniently transform the original coordinates to the ones we use in codes.

Then, a big problem come into our eyesight: we only get one coordinate from the touch panel at a moment, how can we decide which one is for the left rackets, which one is for the other racket? In order to settle this problem, we set a rule for the game, racket can't move until the ball and the touch point reach the corresponding side. The code is as follows:

```
if (x_pingpong>center_x && lem_x>center_x)
{       right_x=ltm_x;
        right_y=ltm_y;      }
```

What we need do now is the easy part: designing the rules of game. As we already know, the trace of rackets is the same with the touch position and what's left is the movement of the ball. By setting the movement step and the direction of the ball, and then adding then to the previous position, we can get the instant position of the ball. For example, for the ball moving towards left direction, we have:

```
if(x_pingpong>(right_x-20)  &&  x_pingpong<(right_x+20)  &&  y_pingpong>(right_y-20)&&
y_pingpong< right_y+20);
```

```
x_count=-1;
```

By considering all the situations the ball would move towards left, we can get the direction needed for realizing the ball's trace. Then, just by adding the data with the the previous x axis position, we can get the x_pingpong which is shown below:

```
x_poingpong=x_pingpong+x_count;
y_poingpong=y_pingpong+y_count;
```

In terms of the score of the game, we just need to count the number of ball being out of boundary on each side. Whoever get the 11 points would win this game.

The final part of the C code is to transmit the controlled data from the nios system to the LCD_Timing_Controller, so that the reprocessed pictures can be sent to the touch panel. The code below shows how we transform the needed data out.

```
IOWR_ALTERA_AVALON_PIO_DATA(PIO_LED_BASE, score);
IOWR_ALTERA_AVALON_PIO_DATA(PIO_PINGPONG_BASE, x_y_pingpong);
IOWR_ALTERA_AVALON_PIO_DATA(PIO_LEFT_BASE, left_y_x);
IOWR_ALTERA_AVALON_PIO_DATA(PIO_RIGHT_BASE, right_y_x);
```

5. Conclusion

5.1 Responsibilities

Ran Zheng: drafted original proposal; researched guide book and helped with whole system construction.

Hao Zheng: developed all aspects of hardware (modified Verilog code); built NIOS system and set up interruption; contributed to presentation slides and final report.

Xiang Zhou: developed algorithm; wrote software for control of game. Write part of the final report. Helped set hardware.

Younggyun Cho: helped write game logic; researched and initially implemented displaying image using ROM; loaded background into the DE2 Flash; Detected and fixed bugs; contributed to final report.

Actually, as a team, we work together. It's pretty hard to tell exactly what a single team member did in this project.

5.2 Lessons Learned

This game was successfully implemented. Although this was a simple game and we believe it was a success, it was definitely a lot harder to implement than we imagined.

In displaying image step, it took a long time to figure out that using ROM to store picture is better. We need to think from hardware perspective. In system building step, it is difficult to learn Verilog alone, understand communication protocol and build Nios system from the beginning instead of using the work already done by professor. We use interruption function given by PIO which makes later work more convenient. Even though we thought we finished the hardware, we had to keep going back to debug it as we implemented more software features.

We've learned a lot from project. Thanks for Prof. Edwards and our TA Shangru Li for all the help and suggestion!

6.Codes

C code

```
#include <stdio.h>

#include "system.h"

#include "altera_avalon_pio_regs.h"

#include "alt_types.h"

#include "sys/alt_irq.h"

#include "priv/alt_busy_sleep.h"

#define LEDCON 0x01

#define KEYCON 0x01

#define left 5
#define right 700

#define down    50
#define    up 450


#define center_x 340
#define center_y 250

#define speed_normal 5000;
#define speed_high 2500;
#define speed_low    10000;


volatile alt_u32 done = 0;          //flag£ºinform the occurrance of an interrupt

    volatile alt_u32    x_pingpong=400;
    volatile alt_u32    y_pingpong=150;
    volatile alt_u32    x_y_pingpong=0;

   volatile alt_u32      ltm_x;
   volatile alt_u32      ltm_y;
   volatile alt_u32      ltm_y_x;
```

```
volatile alt_u32      left_x=100;
volatile alt_u32      left_y=150;
volatile alt_u32      left_y_x;


volatile alt_u32      right_x=600;
volatile alt_u32      right_y=150;
volatile alt_u32      right_y_x;


volatile alt_u32      score1=0;
volatile alt_u32      score2=0;
volatile alt_u32      score3=0;
volatile alt_u32      score4=0;


volatile alt_u32      score;



volatile alt_u32      x_count;
volatile alt_u32      y_count;

volatile alt_u32      flag;
     volatile alt_u32      flag1;

volatile alt_u32    count_r=0;
volatile alt_u32    count_l=0;    // score of the left side


volatile alt_u32    speed=speed_normal;


#define PIO_LED_BASE 0x00101020

static void KeyDown_interrupts(void* context, alt_u32 id)

{        IOWR_ALTERA_AVALON_PIO_EDGE_CAP(PIO_KEY_BASE, ~KEYCON); // clear the edge
capturing register

     ltm_y_x=IORD_ALTERA_AVALON_PIO_DATA(PIO_SW_BASE);
     ltm_x=(ltm_y_x>>12)*800/4095;
```

```
     ltm_y=(ltm_y_x & 0xfff)*480/4095;


     if (x_pingpong<center_x && ltm_x<center_x)
        {
          left_x=ltm_x-20;
          left_y=ltm_y;
        }


      if (x_pingpong>center_x && ltm_x>center_x)
        {
          right_x=ltm_x;
          right_y=ltm_y;
        }


        if (flag==0)
             {
               flag=1;
               score1=0;
               score2=0;
               score3=0;
               score4=0;
               count_r=0;
               count_l=0;
             }

        flag1=1;

   if (ltm_x>0 && ltm_x<100 && ltm_y>0 && ltm_y<60)
        speed=speed_low;

   if (ltm_x>(center_x-30) && ltm_x<(center_x+80) && ltm_y>0 && ltm_y<60)
        speed=speed_normal;

   if (ltm_x>right && ltm_x<(right+60) && ltm_y>0 && ltm_y<60)
        speed=speed_high;

}

void InitPIO(void)

{       /*initializing the PIO_KEY as the input and PIO_LED as output */
```

```
	IOWR_ALTERA_AVALON_PIO_DIRECTION(PIO_KEY_BASE, ~KEYCON); //0 means input

	IOWR_ALTERA_AVALON_PIO_DIRECTION(PIO_LED_BASE, LEDCON); //1 means output

	IOWR_ALTERA_AVALON_PIO_IRQ_MASK(PIO_KEY_BASE, KEYCON); // open PIO_KEY interrupt

	IOWR_ALTERA_AVALON_PIO_EDGE_CAP(PIO_KEY_BASE, ~KEYCON);//clear the edge capturing register

	/* register the interrupts */

	alt_irq_register(PIO_KEY_IRQ, NULL, KeyDown_interrupts);

}

void main(void)

{	flag=0;
	 x_count=-1;
	 y_count=-1;
	   int start=0;                         //beging of the whole game
	   int beginL=0,beginR=0;             // flag for the serve side
	     int play=1;                          // flag of the whole game


	volatile alt_u32 key_state, old_state, new_state;


	old_state = KEYCON;

	IOWR_ALTERA_AVALON_PIO_DATA(PIO_LED_BASE, old_state); //initializing LED and extinguish it

	InitPIO();
	right_y_x=right_x+right_y*2048;
	left_y_x=left_x+left_y*2048;
	score=score1+(score2<<4)+(score3<<8)+(score4<<12);
	          x_pingpong=center_x;
	         y_pingpong=center_y;

	while(1)
```

```
        {
            right_y_x=ltm_x+ltm_y*2048;
            left_y_x=left_x+left_y*2048;
            score=score1+(score2<<4)+(score3<<8)+(score4<<12);
            x_y_pingpong=y_pingpong*2048+x_pingpong;



                alt_busy_sleep(50000);          //delay5ms

                key_state = IORD_ALTERA_AVALON_PIO_DATA(PIO_KEY_BASE)&KEYCON;

                if(key_state == 0xFF)           //interrupt caused by pulse

                    continue;                   //remove keyboard jitter

                new_state = ~(old_state^key_state); //get the new state

                old_state = new_state;          //save the status of LED

                IOWR_ALTERA_AVALON_PIO_DATA(PIO_LED_BASE, score);
                IOWR_ALTERA_AVALON_PIO_DATA( PIO_PINGPONG_BASE,x_y_pingpong);
                IOWR_ALTERA_AVALON_PIO_DATA( PIO_LEFT_BASE,left_y_x);
                IOWR_ALTERA_AVALON_PIO_DATA( PIO_RIGHT_BASE,right_y_x);



            while(flag && flag1)
            {

                x_pingpong=x_pingpong+x_count;
                y_pingpong=y_pingpong+y_count;

          //      if (right <=x_pingpong)
            //          x_count=-1;

                if (x_pingpong>(right_x-20 )&& x_pingpong <(right_x+20) && 
y_pingpong>(right_y-20)&& y_pingpong<right_y+20)
                    x_count=-1;
            // if (left>=x_pingpong)
            //        x_count=1;

                if (x_pingpong>(left_x-20) && x_pingpong<(left_x+20) && 
y_pingpong>(left_y-20)&& y_pingpong<left_y+20)
```

```
      x_count=1;

  if (up<=y_pingpong)
    y_count=-1;
  if (down >=y_pingpong)
    y_count=1;

  right_y_x=right_x+right_y*2048;
  left_y_x=left_x+left_y*2048;
score=score1+(score2<<4)+(score3<<8)+(score4<<12);
x_y_pingpong=y_pingpong*2048+x_pingpong;

IOWR_ALTERA_AVALON_PIO_DATA(PIO_LED_BASE, score);
IOWR_ALTERA_AVALON_PIO_DATA( PIO_PINGPONG_BASE,x_y_pingpong);
IOWR_ALTERA_AVALON_PIO_DATA( PIO_LEFT_BASE,left_y_x);
IOWR_ALTERA_AVALON_PIO_DATA( PIO_RIGHT_BASE,right_y_x);
alt_busy_sleep(speed);    /// sudu

  if (x_pingpong<=left)
      {
          count_r++;
          x_pingpong=center_x;
          y_pingpong=center_y;
          x_count=1;
          y_count=-1;
          flag1=0;

      }

if (x_pingpong>=right)
{
          count_l++;
        x_pingpong=center_x;
          y_pingpong=center_y;
          x_count=-1;
          y_count=-1;
          flag1=0;

}

if (count_r>=10)
    {
          score3=1;
          score4=count_r-10;
```

```
            }
        else
          {
                score3=0;
                score4=count_r;
            }

      if (count_l>=10)
            {
                score1=1;
                score2=count_l-10;
            }
      else
          {
                score1=0;
                score2=count_l;
            }



          if( count_r==11 || count_l==11)
          {
                flag=0;
                x_pingpong=center_x;
                y_pingpong=center_y;

                right_x=650;
                right_y=150;

                left_x=100;
                left_y=150;
          }
    }
}
}
```

lcd_spi_controller

```
module lcd_spi_cotroller (//  Host Side
                              iCLK,
                              iRST_n,
                              //    3wire interface side
                              o3WIRE_SCLK,
                              io3WIRE_SDAT,
                              o3WIRE_SCEN,
```

```
                          o3WIRE_BUSY_n
                          );
//=============================================================================
// PARAMETER declarations
//=============================================================================

parameter    LUT_SIZE =    20; // Total setting register numbers
//=============================================================================
// PORT declarations
//=============================================================================
//    Host Side
output       o3WIRE_BUSY_n;
input        iCLK;
input        iRST_n;
//    3wire interface side
output       o3WIRE_SCLK;
inout        io3WIRE_SDAT;
output       o3WIRE_SCEN;
//    Internal Registers/Wires
//=============================================================================
// REG/WIRE declarations
//=============================================================================
reg               m3wire_str;
wire      m3wire_rdy;
wire      m3wire_ack;
wire      m3wire_clk;
reg  [15:0]       m3wire_data;
reg  [15:0]       lut_data;
reg  [5:0] lut_index;
reg  [3:0] msetup_st;
reg               o3WIRE_BUSY_n;
wire      v_reverse; // display Vertical reverse function
wire      h_reverse; // display Horizontal reverse function
wire [9:0] g0;
wire [9:0] g1;
wire [9:0] g2;
wire [9:0] g3;
wire [9:0] g4;
wire [9:0] g5;
wire [9:0] g6;
wire [9:0] g7;
wire [9:0] g8;
wire [9:0] g9;
wire [9:0] g10;
```

```
wire [9:0] g11;

//==============================================================================
// Structural coding
//==============================================================================

assign      h_reverse = 1'b0;
assign      v_reverse = 1'b1; // enable vertical reverse display function

three_wire_controller    u0   (      //     Host Side
                              .iCLK(iCLK),
                              .iRST(iRST_n),
                              .iDATA(m3wire_data),
                              .iSTR(m3wire_str),
                              .oACK(m3wire_ack),
                              .oRDY(m3wire_rdy),
                              .oCLK(m3wire_clk),
                              //     Serial Side
                              .oSCEN(o3WIRE_SCEN),
                              .SDA(io3WIRE_SDAT),
                              .oSCLK(o3WIRE_SCLK)
                         );
//////////////////////      Config Control ///////////////////////////
always@(posedge m3wire_clk or negedge iRST_n)
begin
    if(!iRST_n)
    begin
        lut_index <=    0;
        msetup_st      <=    0;
        m3wire_str     <=    0;
        o3WIRE_BUSY_n    <=    0;
    end
    else
    begin
        if(lut_index<LUT_SIZE)
        begin
        o3WIRE_BUSY_n    <=    0;
            case(msetup_st)
            0:    begin
                      msetup_st      <=    1;
                  end
            1:    begin
                      msetup_st      <=    2;
                  end
```

```
			2:	begin
					m3wire_data	<=	lut_data;
					m3wire_str	<=	1;
					msetup_st	<=	3;
				end
			3:	begin
					if(m3wire_rdy)
					begin
						if(m3wire_ack)
						msetup_st	<=	4;
						else
						msetup_st	<=	0;
						m3wire_str	<=	0;
					end
				end
			4:	begin
					lut_index <=	lut_index+1;
					msetup_st	<=	0;
				end
			endcase
		end
		else		o3WIRE_BUSY_n	<=	1;
	end
end

assign	g0 =106;
assign	g1 =200;
assign	g2 =289;
assign	g3 =375;
assign	g4 =460;
assign	g5 =543;
assign	g6 =625;
assign	g7 =705;
assign	g8 =785;
assign	g9 =864;
assign	g10 = 942;
assign	g11 = 1020;

/////////////////////	Config Data LUT	//////////////////////////
always
begin
	case(lut_index)
```

```
		0		:	lut_data	<=	{6'h11,2'b01,g0[9:8],g1[9:8],g2[9:8],g3[9:8]};
		1		:	lut_data	<=	{6'h12,2'b01,g4[9:8],g5[9:8],g6[9:8],g7[9:8]};
		2		:	lut_data	<=	{6'h13,2'b01,g8[9:8],g9[9:8],g10[9:8],g11[9:8]};
		3		:	lut_data	<=	{6'h14,2'b01,g0[7:0]};
		4		:	lut_data	<=	{6'h15,2'b01,g1[7:0]};
		5		:	lut_data	<=	{6'h16,2'b01,g2[7:0]};
		6		:	lut_data	<=	{6'h17,2'b01,g3[7:0]};
		7		:	lut_data	<=	{6'h18,2'b01,g4[7:0]};
		8		:	lut_data	<=	{6'h19,2'b01,g5[7:0]};
		9		:	lut_data	<=	{6'h1a,2'b01,g6[7:0]};
		10		:	lut_data	<=	{6'h1b,2'b01,g7[7:0]};
		11		:	lut_data	<=	{6'h1c,2'b01,g8[7:0]};
		12		:	lut_data	<=	{6'h1d,2'b01,g9[7:0]};
		13		:	lut_data	<=	{6'h1e,2'b01,g10[7:0]};
		14		:	lut_data	<=	{6'h1f,2'b01,g11[7:0]};
		15		:	lut_data	<=	{6'h20,2'b01,4'hf,4'h0};
		16		:	lut_data	<=	{6'h21,2'b01,4'hf,4'h0};
		17		:	lut_data	<=	{6'h03, 2'b01, 8'hdf};
		18		:	lut_data	<=	{6'h02, 2'b01, 8'h07};
		19		:	lut_data	<=	{6'h04, 2'b01, 6'b000101,!v_reverse,!h_reverse};
		default	:	lut_data	<=	16'h0000;
		endcase
end
////////////////////////////////////////////////////////////////////

endmodule
```

DE2_LTM_Ephoto

```
module DE2_LTM_Ephoto
	(
		//////////////////// Clock Input	 	////////////////////
		CLOCK_27,						//	27 MHz
		CLOCK_50,						//	50 MHz
		EXT_CLOCK,						//	External Clock
		//////////////////// Push Button		////////////////////
		KEY,						//	Pushbutton[3:0]
		//////////////////// DPDT Switch		////////////////////
		SW,							//	Toggle Switch[17:0]
		//////////////////// 7-SEG Dispaly	////////////////////
		HEX0,							//	Seven Segment Digit 0
		HEX1,							//	Seven Segment Digit 1
		HEX2,							//	Seven Segment Digit 2
		HEX3,							//	Seven Segment Digit 3
```

```
HEX4,                           //   Seven Segment Digit 4
HEX5,                           //   Seven Segment Digit 5
HEX6,                           //   Seven Segment Digit 6
HEX7,                           //   Seven Segment Digit 7


////////////////////    SDRAM Interface      ////////////////
DRAM_DQ,                        //   SDRAM Data bus 16 Bits
DRAM_ADDR,                      //   SDRAM Address bus 12 Bits
DRAM_LDQM,                      //   SDRAM Low-byte Data Mask
DRAM_UDQM,                          //   SDRAM High-byte Data Mask
DRAM_WE_N,                      //   SDRAM Write Enable
DRAM_CAS_N,                         //   SDRAM Column Address Strobe
DRAM_RAS_N,                         //   SDRAM Row Address Strobe
DRAM_CS_N,                      //   SDRAM Chip Select
DRAM_BA_0,                      //   SDRAM Bank Address 0
DRAM_BA_1,                      //   SDRAM Bank Address 0
DRAM_CLK,                       //   SDRAM Clock
DRAM_CKE,                       //   SDRAM Clock Enable
//////////////////// Flash Interface        ////////////////
FL_DQ,                          //   FLASH Data bus 8 Bits
FL_ADDR,                    //   FLASH Address bus 22 Bits
FL_WE_N,                        //   FLASH Write Enable
FL_RST_N,                       //   FLASH Reset
FL_OE_N,                    //   FLASH Output Enable
FL_CE_N,                    //   FLASH Chip Enable

GPIO_0,

        //////////////////// SRAM Interface        ////////////////
SRAM_DQ,                        //   SRAM Data bus 16 Bits
SRAM_ADDR,                      //   SRAM Address bus 18 Bits
SRAM_UB_N,                      //   SRAM High-byte Data Mask
SRAM_LB_N,                      //   SRAM Low-byte Data Mask
SRAM_WE_N,                      //   SRAM Write Enable
SRAM_CE_N,                      //   SRAM Chip Enable
SRAM_OE_N,                      //   SRAM Output Enable


  pio_racket_left,
  pio_racket_right,
pio_pingpong,
pio_hex,
```

```
    pio_sw,
    pio_key



    );
//==========================================================================
// PORT declarations
//==========================================================================
////////////////////////	Clock Input	 	////////////////////////
input		CLOCK_27;		//	27 MHz
input		CLOCK_50;		//	50 MHz
input		EXT_CLOCK;		//	External Clock
////////////////////////	Push Button		////////////////////////
input	[3:0]	KEY;		//	Pushbutton[3:0]
////////////////////////	DPDT Switch		////////////////////////
input	[17:0]	SW;		//	Toggle Switch[17:0]
////////////////////////	7-SEG Dispaly	////////////////////////
output	[6:0]	HEX0;		//	Seven Segment Digit 0
output	[6:0]	HEX1;		//	Seven Segment Digit 1
output	[6:0]	HEX2;		//	Seven Segment Digit 2
output	[6:0]	HEX3;		//	Seven Segment Digit 3
output	[6:0]	HEX4;		//	Seven Segment Digit 4
output	[6:0]	HEX5;		//	Seven Segment Digit 5
output	[6:0]	HEX6;		//	Seven Segment Digit 6
output	[6:0]	HEX7;		//	Seven Segment Digit 7

////////////////////////	SDRAM Interface	////////////////////////
inout	[15:0]	DRAM_DQ;		//	SDRAM Data bus 16 Bits
output	[11:0]	DRAM_ADDR;		//	SDRAM Address bus 12 Bits
output		DRAM_LDQM;		//	SDRAM Low-byte Data Mask
output		DRAM_UDQM;		//	SDRAM High-byte Data Mask
output		DRAM_WE_N;		//	SDRAM Write Enable
output		DRAM_CAS_N;		//	SDRAM Column Address Strobe
output		DRAM_RAS_N;		//	SDRAM Row Address Strobe
output		DRAM_CS_N;		//	SDRAM Chip Select
output		DRAM_BA_0;		//	SDRAM Bank Address 0
output		DRAM_BA_1;		//	SDRAM Bank Address 0
output		DRAM_CLK;		//	SDRAM Clock
output		DRAM_CKE;		//	SDRAM Clock Enable
////////////////////////	Flash Interface////////////////////////
inout	[7:0]	FL_DQ;		//	FLASH Data bus 8 Bits
output	[21:0]	FL_ADDR;		//	FLASH Address bus 22 Bits
output		FL_WE_N;		//	FLASH Write Enable
```

```
output            FL_RST_N;               //   FLASH Reset
output            FL_OE_N;            //   FLASH Output Enable
output            FL_CE_N;            //   FLASH Chip Enable


//////////////////////// GPIO     ////////////////////////////////

inout   [35:0]    GPIO_0;                 //   GPIO Connection 0

//////////////////////// SRAM Interface     ////////////////////////
inout   [15:0]    SRAM_DQ;                //   SRAM Data bus 16 Bits
output  [17:0]    SRAM_ADDR;              //   SRAM Address bus 18 Bits
output            SRAM_UB_N;              //   SRAM High-byte Data Mask
output            SRAM_LB_N;              //   SRAM Low-byte Data Mask
output            SRAM_WE_N;              //   SRAM Write Enable
output            SRAM_CE_N;              //   SRAM Chip Enable
output            SRAM_OE_N;              //   SRAM Output Enable



////////////////////////////
//   All inout port turn to tri-state
assign  DRAM_DQ         =   16'hzzzz;
assign  OTG_DATA    =   16'hzzzz;
assign  LCD_DATA    =   8'hzz;
assign  SD_DAT      =   1'bz;
assign  ENET_DATA   =   16'hzzzz;
assign  AUD_ADCLRCK     =   1'bz;
assign  AUD_DACLRCK     =   1'bz;
assign  AUD_BCLK    =   1'bz;
assign  GPIO_1      =   36'hzzzzzzzzz;
//=============================================================================
// REG/WIRE declarations
//=============================================================================
// Touch panel signal //
wire [7:0] ltm_r;       //   LTM Red Data 8 Bits
wire [7:0] ltm_g;       //   LTM Green Data 8 Bits
wire [7:0] ltm_b;       //   LTM Blue Data 8 Bits
wire        ltm_nclk; //   LTM Clcok
wire        ltm_hd;
wire        ltm_vd;
wire        ltm_den;
wire            adc_dclk;
wire            adc_cs;
```

```
wire                    adc_penirq_n;
wire                    adc_busy;
wire                    adc_din;
wire                    adc_dout;
wire                    adc_ltm_sclk;
wire            ltm_grst;
// LTM Config//
wire            ltm_sclk;
wire            ltm_sda;
wire            ltm_scen;
wire                    ltm_3wirebusy_n;

wire [11:0]     x_coord;
wire [11:0]     y_coord;
wire            new_coord;
wire [2:0] photo_cnt;
// clock
wire                    F_CLK;// flash read clock
reg   [31:0]    div;
// sdram to touch panel timing
wire            mRead;
wire [15:0]     Read_DATA1;
wire [15:0]     Read_DATA2;
//    flash to sdram sdram
wire [7:0]      sRED;// flash to sdram red pixel data
wire [7:0] sGREEN;// flash to sdram green pixel data
wire [7:0] sBLUE;// flash to sdram blue pixel data
wire            sdram_write_en; // flash to sdram write control
wire            sdram_write; // sdram write signal
// system reset
wire            DLY0;
wire            DLY1;
wire            DLY2;

//=============================================================================
// Structural coding
//=============================================================================


////////////////////////////////////////
assign     adc_penirq_n    =GPIO_0[0];
assign     adc_dout        =GPIO_0[1];
assign     adc_busy        =GPIO_0[2];
assign     GPIO_0[3]       =adc_din;
```

```
assign    GPIO_0[4]     =adc_ltm_sclk;
assign    GPIO_0[5]     =ltm_b[3];
assign    GPIO_0[6]     =ltm_b[2];
assign    GPIO_0[7]     =ltm_b[1];
assign    GPIO_0[8]     =ltm_b[0];
assign    GPIO_0[9]     =ltm_nclk;
assign    GPIO_0[10]    =ltm_den;
assign    GPIO_0[11]    =ltm_hd;
assign    GPIO_0[12]    =ltm_vd;
assign    GPIO_0[13]    =ltm_b[4];
assign    GPIO_0[14]    =ltm_b[5];
assign    GPIO_0[15]    =ltm_b[6];
assign    GPIO_0[16]    =ltm_b[7];
assign    GPIO_0[17]    =ltm_g[0];
assign    GPIO_0[18]    =ltm_g[1];
assign    GPIO_0[19]    =ltm_g[2];
assign    GPIO_0[20]    =ltm_g[3];
assign    GPIO_0[21]    =ltm_g[4];
assign    GPIO_0[22]    =ltm_g[5];
assign    GPIO_0[23]    =ltm_g[6];
assign    GPIO_0[24]    =ltm_g[7];
assign    GPIO_0[25]    =ltm_r[0];
assign    GPIO_0[26]    =ltm_r[1];
assign    GPIO_0[27]    =ltm_r[2];
assign    GPIO_0[28]    =ltm_r[3];
assign    GPIO_0[29]    =ltm_r[4];
assign    GPIO_0[30]    =ltm_r[5];
assign    GPIO_0[31]    =ltm_r[6];
assign    GPIO_0[32]    =ltm_r[7];
assign    GPIO_0[33]    =ltm_grst;
assign    GPIO_0[34]    =ltm_scen;
assign    GPIO_0[35]    =ltm_sda;

////////////////////////////////////////
assign ltm_grst           = KEY[0];
assign F_CLK          = div[3];
assign adc_ltm_sclk= ( adc_dclk & ltm_3wirebusy_n )    |    ( ~ltm_3wirebusy_n & ltm_sclk );
always @( posedge CLOCK_50 )
    begin
        div <= div+1;
    end

/////////////////////////////////////////////////////////////////
/*******************************************************out*****************
```

```
***********/
output    [31:0]pio_racket_left;
output    [31:0] pio_racket_right;
output [31:0] pio_pingpong;
output    [15:0]    pio_hex;

output [31:0]pio_sw;
output pio_key;




hope      jjj(
                    // 1) global signals:
                      .clk(CLOCK_50),
                      .reset_n(KEY[0]),

                    // the_pio_hex
                      .out_port_from_the_pio_hex(pio_hex),

                    // the_pio_key
                      .in_port_to_the_pio_key(pio_key),

                    // the_pio_left
                      .out_port_from_the_pio_left(pio_racket_left),

                    // the_pio_pingpong
                      .out_port_from_the_pio_pingpong(pio_pingpong),

                    // the_pio_right
                      .out_port_from_the_pio_right(pio_racket_right),

                    // the_pio_sw
                      .in_port_to_the_pio_sw(pio_sw),

                    // the_sram_16bit_512k_0
                      .SRAM_ADDR_from_the_sram_16bit_512k_0(SRAM_ADDR),
                      .SRAM_CE_N_from_the_sram_16bit_512k_0(SRAM_CE_N),
                      .SRAM_DQ_to_and_from_the_sram_16bit_512k_0(SRAM_DQ),
                      .SRAM_LB_N_from_the_sram_16bit_512k_0(SRAM_LB_N),
                      .SRAM_OE_N_from_the_sram_16bit_512k_0(SRAM_OE_N),
                      .SRAM_UB_N_from_the_sram_16bit_512k_0(SRAM_UB_N),
                      .SRAM_WE_N_from_the_sram_16bit_512k_0(SRAM_WE_N)
                  )
```

```
;



assign pio_sw={8'd0,y_coord,x_coord};
assign pio_key=new_coord;

lcd_spi_cotroller        u1       (
                                        // Host Side
                                        .iCLK(CLOCK_50),
                                        .iRST_n(DLY0),
                                        // 3 wire Side
                                        .o3WIRE_SCLK(ltm_sclk),
                                        .io3WIRE_SDAT(ltm_sda),
                                        .o3WIRE_SCEN(ltm_scen),
                                        .o3WIRE_BUSY_n(ltm_3wirebusy_n)
                                        );

adc_spi_controller  u2            (
                                        .iCLK(CLOCK_50),
                                        .iRST_n(DLY0),
                                        .oADC_DIN(adc_din),
                                        .oADC_DCLK(adc_dclk),
                                        .oADC_CS(adc_cs),
                                        .iADC_DOUT(adc_dout),
                                        .iADC_BUSY(adc_busy),
                                        .iADC_PENIRQ_n(adc_penirq_n),
                                        .oX_COORD(x_coord),
                                        .oY_COORD(y_coord),
                                        .oNEW_COORD(new_coord),
                                         );

touch_point_detector    u3    (
                                        .iCLK(CLOCK_50),
                                        .iRST_n(DLY0),
                                        .iX_COORD(x_coord),
                                        .iY_COORD(y_coord),
                                        .iNEW_COORD(new_coord),
                                        .iSDRAM_WRITE_EN(sdram_write_en),
                                        .oPHOTO_CNT(photo_cnt),
                                        );

flash_to_sdram_controller     u4          (
```

```
                                        .iPHOTO_NUM(2),
                                        .iRST_n(DLY1) ,
                                        .iF_CLK(F_CLK),
                                        .FL_DQ(FL_DQ)    ,
                                        .oFL_ADDR(FL_ADDR) ,
                                        .oFL_WE_N(FL_WE_N) ,
                                        .oFL_RST_n(FL_RST_N),
                                        .oFL_OE_N(FL_OE_N) ,
                                        .oFL_CE_N(FL_CE_N) ,
                                        .oSDRAM_WRITE_EN(sdram_write_en),
                                        .oSDRAM_WRITE(sdram_write),
                                        .oRED(sRED),
                                        .oGREEN(sGREEN),
                                        .oBLUE(sBLUE),
                                        );

SEG7_LUT_8                  u5          (
                                        .oSEG0(HEX0),
                                        .oSEG1(HEX1),
                                        .oSEG2(HEX2),
                                        .oSEG3(HEX3),
                                        .oSEG4(HEX4),
                                        .oSEG5(HEX5),
                                        .oSEG6(HEX6),
                                        .oSEG7(HEX7),
                                        .iDIG({4'h0,x_coord,4'h0,y_coord}),
                                        .ON_OFF(8'b01110111)
                                        );

lcd_timing_controller       u6    (
                                        .iCLK(ltm_nclk),
                                        .iRST_n(DLY2),
                                        // sdram side
                                        .iREAD_DATA1(Read_DATA1),
                                        .iREAD_DATA2(Read_DATA2),
                                        .oREAD_SDRAM_EN(mRead),
                                        // lcd side
                                        .oLCD_R(ltm_r),
                                        .oLCD_G(ltm_g),
                                        .oLCD_B(ltm_b),
                                        .oHD(ltm_hd),
                                        .oVD(ltm_vd),
                                        .oDEN(ltm_den),
```

```
                                   .pio_racket_left(pio_racket_left),
                                   .pio_racket_right(pio_racket_right),
                                   .pio_pingpang(pio_pingpong),
                                   .pio_hex(pio_hex)
                                   );

//    SDRAM frame buffer
Sdram_Control_4Port    u7    (     //    HOST Side
                                   .REF_CLK(CLOCK_50),
                                   .RESET_N(1'b1),
                                   //    FIFO Write Side 1
                                   .WR1_DATA({sRED,sGREEN}),
                                   .WR1(sdram_write),
                                   .WR1_FULL(WR1_FULL),
                                   .WR1_ADDR(0),
                                   .WR1_MAX_ADDR(800*480),
                                   .WR1_LENGTH(9'h80),
                                   .WR1_LOAD(!DLY0),
                                   .WR1_CLK(F_CLK),
                                   //    FIFO Write Side 2

                                   .WR2_DATA({8'h0,sBLUE}),
                                   .WR2(sdram_write),
                                   .WR2_ADDR(22'h100000),
                                   .WR2_MAX_ADDR(22'h100000+800*480),
                                   .WR2_LENGTH(9'h80),
                                   .WR2_LOAD(!DLY0),
                                   .WR2_CLK(F_CLK),

                                   //    FIFO Read Side 1
                                   .RD1_DATA(Read_DATA1),
                                   .RD1(mRead),
                                   .RD1_ADDR(0),
                                   .RD1_MAX_ADDR(800*480),
                                   .RD1_LENGTH(9'h80),
                                   .RD1_LOAD(!DLY0),
                                   .RD1_CLK(ltm_nclk),
                                   //    FIFO Read Side 2

                                   .RD2_DATA(Read_DATA2),
                                   .RD2(mRead),
                                   .RD2_ADDR(22'h100000),
                                   .RD2_MAX_ADDR(22'h100000+800*480),
```

```
                    .RD2_LENGTH(9'h80),
                    .RD2_LOAD(!DLY0),
                    .RD2_CLK(ltm_nclk),

                    //    SDRAM Side
                    .SA(DRAM_ADDR),
                    .BA({DRAM_BA_1,DRAM_BA_0}),
                    .CS_N(DRAM_CS_N),
                    .CKE(DRAM_CKE),
                    .RAS_N(DRAM_RAS_N),
                    .CAS_N(DRAM_CAS_N),
                    .WE_N(DRAM_WE_N),
                    .DQ(DRAM_DQ),
                    .DQM({DRAM_UDQM,DRAM_LDQM}),
                    .SDR_CLK(DRAM_CLK),
                    .CLK_33(ltm_nclk)
                    );

Reset_Delay         u8      (.iCLK(CLOCK_50),
                    .iRST(KEY[0]),
                    .oRST_0(DLY0),
                    .oRST_1(DLY1),
                    .oRST_2(DLY2)
                    );
endmodule
```

flash_to_sdram_controller

```
module flash_to_sdram_controller(
            iRST_n,
            iPHOTO_NUM,
            // Flash side
            iF_CLK,
            FL_DQ,
            oFL_ADDR,
            oFL_WE_N,
            oFL_RST_n,
            oFL_OE_N,
            oFL_CE_N,
            // Sdram side
            oSDRAM_WRITE_EN,
            oSDRAM_WRITE,
            oRED,
            oGREEN,
```

```
                    oBLUE,
                    );
//==============================================================================
// PARAMETER declarations
//==============================================================================
parameter DISP_MODE = 800*480;
//=============================================================================
// PORT declarations
//=============================================================================
input               iRST_n;                 //  System reset
input       [3:0]   iPHOTO_NUM;             //  Picture status
input               iF_CLK;                 //  Flash read clcok
inout       [7:0]   FL_DQ;              //  FLASH Data bus 8 Bits
output      [22:0]  oFL_ADDR;               //   FLASH Address bus 22 Bits
output              oFL_WE_N;               //   FLASH Write Enable
output              oFL_RST_n;              //   FLASH Reset
output              oFL_OE_N;               //   FLASH Output Enable
output              oFL_CE_N;               //   FLASH Chip Enable
output          oSDRAM_WRITE_EN;        //  SDRAM write enable control signal
output              oSDRAM_WRITE;           //  SDRAM write signal
output      [7:0]   oRED;                   //  Image red color data to sdram
output      [7:0]   oGREEN;                 //  Image green color data to sdram
output      [7:0]   oBLUE;                  //  Image blue color data to sdram
//===============================================================================
// REG/WIRE declarations
//===============================================================================
reg             oSDRAM_WRITE_EN;
reg                 oSDRAM_WRITE;
reg   [1:0]     flash_addr_cnt;
reg   [7:0]     fl_dq_delay1;
reg   [7:0]     fl_dq_delay2;
reg   [7:0]     fl_dq_delay3;
reg   [18:0]    write_cnt ;
reg         [7:0]   oRED;
reg         [7:0]   oGREEN;
reg         [7:0]   oBLUE;
reg         [22:0]  flash_addr_o;
wire        [22:0]  flash_addr_max;
wire        [22:0]  flash_addr_min;
reg         [2:0]   d1_photo_num;
reg         [2:0]   d2_photo_num;
reg                 photo_change;
reg                 rgb_sync;
reg                 mrgb_sync;
```

```
//=============================================================================
// Structural coding
//=============================================================================

assign    oFL_WE_N    = 1;
assign    oFL_RST_n = 1;
assign    oFL_OE_N    = 0;
assign    oFL_CE_N    = 0;
assign    oFL_ADDR    = flash_addr_o;
assign    flash_addr_max = 54 + 3*DISP_MODE * (d2_photo_num+1) ; //54(bmp file header)+ 3
x 800x480 (3 800x480 pictures)
assign    flash_addr_min = 54 + 3*DISP_MODE * iPHOTO_NUM;

///////////////////////////////////////////////////


always@(posedge iF_CLK or negedge iRST_n)
    begin
        if (!iRST_n)
            begin
                d1_photo_num <= 0;
                d2_photo_num <= 0;
            end
        else
            begin
                d1_photo_num <= iPHOTO_NUM;
                d2_photo_num <= d1_photo_num;
            end
    end
// This is photo change detection
always@(posedge iF_CLK or negedge iRST_n)
    begin
        if (!iRST_n)
            photo_change <= 0;
        else if (d1_photo_num != iPHOTO_NUM)
            photo_change <= 1;
        else
            photo_change <= 0;
    end
// If changing photo , flash_addr_min &    flash_addr_max    & flash_addr_owill chagne ,
// if flash_addr_o    < flash_addr_max , starting read flash data
always @(posedge iF_CLK or negedge iRST_n)
    begin
        if ( !iRST_n )
```

```
			flash_addr_o <= flash_addr_min ;
		else if (photo_change)
			flash_addr_o <= flash_addr_min ;
		else if ( flash_addr_o    <    flash_addr_max )
			flash_addr_o <= flash_addr_o + 1;
	end

////////////////////// Sdram write enable control    ///////////////////////////
always@(posedge iF_CLK or negedge iRST_n)
	begin
		if (!iRST_n)
			oSDRAM_WRITE_EN <= 0;
		else if ( (flash_addr_o    <    flash_addr_max-1)&&(write_cnt < DISP_MODE) )
			begin
				oSDRAM_WRITE_EN <= 1;
			end
		else
			oSDRAM_WRITE_EN <= 0;
	end
////////////////////// delay flash data    for aligning RGB data///////////////
always@(posedge iF_CLK or negedge iRST_n)
	begin
		if (!iRST_n)
			begin
				fl_dq_delay1 <= 0;
				fl_dq_delay2 <= 0;
				fl_dq_delay3 <= 0;
			end
		else
			begin
				fl_dq_delay1 <= FL_DQ;
				fl_dq_delay2 <= fl_dq_delay1;
				fl_dq_delay3 <= fl_dq_delay2;
			end
	end


always@(posedge iF_CLK or negedge iRST_n)
	begin
		if (!iRST_n)
			flash_addr_cnt <= 0;
		else if ( flash_addr_o    <    flash_addr_max )
		begin
			if (flash_addr_cnt == 2)
```

```
				flash_addr_cnt <= 0;
			else
				flash_addr_cnt <=flash_addr_cnt + 1;
		end
		else
			flash_addr_cnt <= 0;
	end

always@(posedge iF_CLK or negedge iRST_n)
	begin
		if (!iRST_n)
			begin
				write_cnt <= 0;
				mrgb_sync <= 0;
			end
			else if (oSDRAM_WRITE_EN)
				begin
				if (flash_addr_cnt == 1)
				begin
					write_cnt <= write_cnt + 1;
					mrgb_sync <= 1;
				end
				else
					mrgb_sync <= 0;
			end

			else
			begin
				write_cnt <= 0;
				mrgb_sync <= 0;
			end
	end


always@(posedge iF_CLK or negedge iRST_n)
	begin
		if (!iRST_n)
			rgb_sync <= 0;
		else
			rgb_sync <= mrgb_sync;
	end

always@(posedge iF_CLK or negedge iRST_n)
	begin
```

```
		if (!iRST_n)
			begin
				oSDRAM_WRITE <= 0;
				oRED <= 0;
				oGREEN <= 0;
				oBLUE <= 0;
			end
		else if (rgb_sync)
			begin
				oSDRAM_WRITE <= 1;
				oRED	<= fl_dq_delay1;
				oGREEN	<= fl_dq_delay2;
				oBLUE	<= fl_dq_delay3;
			end
		else
			begin
				oSDRAM_WRITE <= 0;
				oRED	<= 0;
				oGREEN	<= 0;
				oBLUE	<= 0;
			end
	end
endmodule
```

three_wire_controller

```
module three_wire_controller(	//	Host Side
						iCLK,
						iRST,
						iDATA,
						iSTR,
						oACK,
						oRDY,
						oCLK,
						//	Serial Side
						oSCEN,
						SDA,
						oSCLK	);
//	Host Side
input			iCLK;
input			iRST;
input			iSTR;
input	[15:0]	iDATA;
output			oACK;
output			oRDY;
```

```
output              oCLK;
//  Serial Side
output              oSCEN;
inout               SDA;
output              oSCLK;
//  Internal Register and Wire
reg                 mSPI_CLK;
reg         [15:0]  mSPI_CLK_DIV;
reg                 mSEN;
reg                 mSDATA;
reg                 mSCLK;
reg                 mACK;
reg         [4:0] mST;

parameter   CLK_Freq =  50000000;   //  50  MHz
parameter   SPI_Freq  =  20000;     //  20  KHz

//  Serial Clock Generator
always@(posedge iCLK or negedge iRST)
begin
    if(!iRST)
    begin
        mSPI_CLK<=  0;
        mSPI_CLK_DIV<=  0;
    end
    else
    begin
        if( mSPI_CLK_DIV    < (CLK_Freq/SPI_Freq) )
        mSPI_CLK_DIV<=  mSPI_CLK_DIV+1;
        else
        begin
            mSPI_CLK_DIV<=  0;
            mSPI_CLK    <=  ~mSPI_CLK;
        end
    end
end
//  Parallel to Serial
always@(negedge mSPI_CLK or negedge iRST)
begin
    if(!iRST)
    begin
        mSEN    <=  1'b1;
        mSCLK   <=  1'b0;
        mSDATA  <=  1'bz;
```

```
		mACK	<=	1'b0;
		mST	<=	4'h00;
	end
	else
	begin
		if(iSTR)
		begin
			if(mST<17)
			mST <=	mST+1'b1;
			if(mST==0)
			begin
				mSEN	<=	1'b0;
				mSCLK	<=	1'b1;
			end
			else if(mST==8)
			mACK	<=	SDA;
			else if(mST==16 && mSCLK)
			begin
				mSEN	<=	1'b1;
				mSCLK	<=	1'b0;
			end
			if(mST<16)
			mSDATA	<=	iDATA[15-mST];
		end
		else
		begin
			mSEN	<=	1'b1;
			mSCLK	<=	1'b0;
			mSDATA	<=	1'bz;
			mACK	<=	1'b0;
			mST	<=	4'h00;
		end
	end
end

assign	oACK		=	mACK;
assign	oRDY		=	(mST==17)	?	1'b1 :	1'b0;
assign	oSCEN		=	mSEN;
assign	oSCLK		=	mSCLK	&	mSPI_CLK;
assign	SDA =	(mST==8) ?	1'bz :
				(mST==17)	?	1'bz :
							mSDATA	;
assign	oCLK		=	mSPI_CLK;
```

```
endmodule
```

lcd_timing_controller

```
module lcd_timing_controller (
                iCLK,                // LCD display clock
                iRST_n,          // systen reset
                // SDRAM SIDE
                iREAD_DATA1,         // R and G  color data form sdram
                iREAD_DATA2,     // B color data form sdram
                oREAD_SDRAM_EN,        // read sdram data control signal
                //LCD SIDE
                oHD,             // LCD Horizontal sync
                oVD,             // LCD Vertical sync
                oDEN,                // LCD Data Enable
                oLCD_R,              // LCD Red color data
                oLCD_G,              // LCD Green color data
                oLCD_B,              // LCD Blue color data


                pio_racket_left,
                pio_racket_right,
                pio_pingpang,
                pio_hex
                );


input [31:0]pio_racket_left;
input [31:0]pio_racket_right;
input [31:0]pio_pingpang;
input [15:0]pio_hex;
//=============================================================================
// PARAMETER declarations
//=============================================================================
parameter H_LINE = 1056;
parameter V_LINE = 525;
parameter Hsync_Blank = 216;
parameter Hsync_Front_Porch = 40;
parameter Vertical_Back_Porch = 35;
parameter Vertical_Front_Porch = 10;
//===========================================================================
// PORT declarations
//===========================================================================
input           iCLK;
input           iRST_n;
```

```
input       [15:0]      iREAD_DATA1;
input       [15:0]      iREAD_DATA2;
output                  oREAD_SDRAM_EN;
output      [7:0] oLCD_R;
output     [7:0]  oLCD_G;
output     [7:0]  oLCD_B;
output                  oHD;
output                  oVD;
output                  oDEN;
//=============================================================================
// REG/WIRE declarations
//=============================================================================
reg         [10:0]    x_cnt;
reg         [9:0] y_cnt;
wire [7:0] read_red;
wire [7:0] read_green;
wire [7:0] read_blue;
wire              display_area;
wire              oREAD_SDRAM_EN;
reg                     mhd;
reg                     mvd;
reg                     oHD;
reg                     oVD;
reg                     oDEN;
reg         [7:0] oLCD_R;
reg         [7:0] oLCD_G;
reg         [7:0] oLCD_B;
//=============================================================================
// Structural coding
//=============================================================================

// This signal control reading data form SDRAM , if high read color data form sdram    .
assign      oREAD_SDRAM_EN = (   (x_cnt>Hsync_Blank-2)&&
                                 (x_cnt<(H_LINE-Hsync_Front_Porch-1))&&
                                 (y_cnt>(Vertical_Back_Porch-1))&&
                                 (y_cnt<(V_LINE - Vertical_Front_Porch))
                             )?   1'b1 : 1'b0;

// This signal indicate the lcd display area .
assign      display_area = ((x_cnt>(Hsync_Blank-1)&& //>215
                            (x_cnt<(H_LINE-Hsync_Front_Porch))&& //< 1016
                            (y_cnt>(Vertical_Back_Porch-1))&&
                            (y_cnt<(V_LINE - Vertical_Front_Porch))
                            ))    ? 1'b1 : 1'b0;
```

```
/*******************pingpong********************************************/
wire [10:0]x_cord_pong;
wire [9:0]y_cord_pong;
assign x_cord_pong=pio_pingpang[10:0]+215;//11'd400;
assign y_cord_pong=pio_pingpang[20:11]+35;//10'd150;

wire en_pong;

assign en_pong=((x_cnt>=x_cord_pong) && (x_cnt<=x_cord_pong+30) &&
(y_cnt>=y_cord_pong)&&(y_cnt<=y_cord_pong+30))?1'b1:1'b0;


wire [9:0]addr_pong;

assign addr_pong=(x_cnt-x_cord_pong)+(y_cnt-y_cord_pong)*30;

wire [7:0]red_pong;


rom_pong    ii (
            .address(addr_pong),
            .clock(iCLK),
            .q(red_pong)
            );

//////////////////////////////////////////////////////////////////////////////////////////////
//////////////////////


/*******************************************score1************************
************************************/

wire [10:0]x_cord_score1;
wire [9:0]y_cord_score1;
assign x_cord_score1=11'd520;
assign y_cord_score1=10'd50;

wire en_score1;

assign en_score1=((x_cnt>=x_cord_score1) && (x_cnt<=x_cord_score1+20) &&
```

```
(y_cnt>=y_cord_score1)&&(y_cnt<=y_cord_score1+20))?1'b1:1'b0;


wire [9:0]addr_score1;

assign addr_score1=(x_cnt-x_cord_score1)+(y_cnt-y_cord_score1)*20;


/***************************score2*****************************************/

wire [10:0]x_cord_score2;
wire [9:0]y_cord_score2;
assign x_cord_score2=11'd540;
assign y_cord_score2=10'd50;

wire en_score2;

assign en_score2=((x_cnt>=x_cord_score2) && (x_cnt<=x_cord_score2+20) &&
(y_cnt>=y_cord_score2)&&(y_cnt<=y_cord_score2+20))?1'b1:1'b0;


wire [9:0]addr_score2;

assign addr_score2=(x_cnt-x_cord_score2)+(y_cnt-y_cord_score2)*20;


/***************************score3*****************************************/

wire [10:0]x_cord_score3;
wire [9:0]y_cord_score3;
assign x_cord_score3=11'd600;
assign y_cord_score3=10'd50;

wire en_score3;

assign en_score3=((x_cnt>=x_cord_score3) && (x_cnt<=x_cord_score3+20) &&
(y_cnt>=y_cord_score3)&&(y_cnt<=y_cord_score3+20))?1'b1:1'b0;


wire [9:0]addr_score3;

assign addr_score3=(x_cnt-x_cord_score3)+(y_cnt-y_cord_score3)*20;
```

```
/***************************score3*****************************************/

wire [10:0]x_cord_score4;
wire [9:0]y_cord_score4;
assign x_cord_score4=11'd620;
assign y_cord_score4=10'd50;

wire en_score4;

assign en_score4=((x_cnt>=x_cord_score4) && (x_cnt<=x_cord_score4+20) &&
(y_cnt>=y_cord_score4)&&(y_cnt<=y_cord_score4+20))?1'b1:1'b0;


wire [9:0]addr_score4;

assign addr_score4=(x_cnt-x_cord_score4)+(y_cnt-y_cord_score4)*20;


wire [15:0]HEX;
assign HEX=pio_hex;//16'H1234;

reg [9:0]addr_score;
reg    [3:0]num;

reg en_score;
always @(*)
     if (en_score1)
         begin
             addr_score<=addr_score1;
             en_score<=1;
             num<=HEX[3:0];
         end
    else if (en_score2)
           begin
             addr_score<=addr_score2;
             en_score<=1;
             num<=HEX[7:4];
            end
    else if (en_score3)
             begin
             addr_score<=addr_score3;
             en_score<=1;
             num<=HEX[11:8];
```

```
                end
      else if (en_score4)
                begin
                addr_score<=addr_score4;
                en_score<=1;
                num<=HEX[15:12];
                end

      else
             begin
            addr_score<=0;
             en_score<=0;
             end


   wire [7:0]reg_score;



choose_display ii3(
                        . clk(iCLK),
                         .rstn(iRST_n),

                         .num(num),
                         .addr(addr_score),

                         .data_out(reg_score)
                        );





      /**********************************************racket*********************
********************/
      /******************racket******************************************/
wire [10:0]x_cord_racket1;
wire [9:0]y_cord_racket1;
assign x_cord_racket1=pio_racket_left[10:0]+215;//11'd300;
assign y_cord_racket1=pio_racket_left[20:11]+35;//10'd150;

wire en_racket1;

assign      en_racket1=((x_cnt>=x_cord_racket1)      &&      (x_cnt<=x_cord_racket1+50)      &&
```

```
(y_cnt>=y_cord_racket1)&&(y_cnt<=y_cord_racket1+50))?1'b1:1'b0;


wire [14:0]addr_racket1;

assign addr_racket1=(x_cnt-x_cord_racket1)+(y_cnt-y_cord_racket1)*50;

wire [10:0]x_cord_racket2;
wire [9:0]y_cord_racket2;
assign x_cord_racket2=pio_racket_right[10:0]+215;//11'd800;
assign y_cord_racket2=pio_racket_right[20:11]+35;//10'd150;

wire en_racket2;

assign en_racket2=((x_cnt>=x_cord_racket2) && (x_cnt<=x_cord_racket2+50) &&
(y_cnt>=y_cord_racket2)&&(y_cnt<=y_cord_racket2+50))?1'b1:1'b0;


wire [14:0]addr_racket2;

assign addr_racket2=(x_cnt-x_cord_racket2)+(y_cnt-y_cord_racket2)*50;


wire en_racket=en_racket1 | en_racket2;

reg [14:0]addr_racket;
always @(*)
    if (en_racket1)
        addr_racket<=addr_racket1;
    else
        addr_racket<=addr_racket2;


wire [14:0]red_racket;


qiupai_rom i14(
            .address(addr_racket),
            .clock(iCLK),
            .q(red_racket)
            );
```

```
/////////////////////////////////////////////////////////////////////////////////////////
///////////

assign read_red =
display_area ?( en_pong ?red_pong:(en_score?(reg_score+iREAD_DATA1[15:8]):(en_racket?(iREA
D_DATA1[15:8] | red_racket):iREAD_DATA1[15:8] ))): 8'b0;
assign read_green = display_area ?(en_score?(reg_score |
iREAD_DATA1[7:0]):( en_racket?(iREAD_DATA1[7:0] | red_racket):iREAD_DATA1[7:0])): 8'b0;
assign read_blue = display_area ?(en_score?(reg_score |
iREAD_DATA2[7:0]):(en_racket?(iREAD_DATA2[7:0] | red_racket):iREAD_DATA2[7:0])): 8'b0;

///////////////////////// x    y counter    and lcd hd generator //////////////////
always@(posedge iCLK or negedge iRST_n)
    begin
        if (!iRST_n)
        begin
            x_cnt <= 11'd0;
            mhd    <= 1'd0;
        end
        else if (x_cnt == (H_LINE-1))
        begin
            x_cnt <= 11'd0;
            mhd    <= 1'd0;
        end
        else
        begin
            x_cnt <= x_cnt + 11'd1;
            mhd    <= 1'd1;
        end
    end

always@(posedge iCLK or negedge iRST_n)
    begin
        if (!iRST_n)
            y_cnt <= 10'd0;
        else if (x_cnt == (H_LINE-1))
        begin
            if (y_cnt == (V_LINE-1))
                y_cnt <= 10'd0;
            else
                y_cnt <= y_cnt + 10'd1;
        end
    end
```

```
////////////////////////////// touch panel timing /////////////////

always@(posedge iCLK    or negedge iRST_n)
	begin
		if (!iRST_n)
			mvd    <= 1'b1;
		else if (y_cnt == 10'd0)
			mvd    <= 1'b0;
		else
			mvd    <= 1'b1;
	end


always@(posedge iCLK or negedge iRST_n)
	begin
		if (!iRST_n)
			begin
				oHD <= 1'd0;
				oVD <= 1'd0;
				oDEN <= 1'd0;
				oLCD_R <= 8'd0;
				oLCD_G <= 8'd0;
				oLCD_B <= 8'd0;
			end
		else
			begin
				oHD <= mhd;
				oVD <= mvd;
				oDEN <= display_area;
				oLCD_R <= read_red;
				oLCD_G <= read_green;
				oLCD_B <= read_blue;
			end
	end
endmodule
```

touch_point_detector

```
module touch_point_detector       (
					iCLK,
					iRST_n,
					iX_COORD,
					iY_COORD,
					iNEW_COORD,
					iSDRAM_WRITE_EN,
```

```
                oPHOTO_CNT,
                );

//==========================================================================
// PARAMETER declarations
//==========================================================================

parameter   PHOTO_NUM = 3;  // total photo numbers
parameter   NEXT_PIC_XBD1 = 12'h0;
parameter   NEXT_PIC_XBD2 = 12'h300;
parameter   NEXT_PIC_YBD1 = 12'he00;
parameter   NEXT_PIC_YBD2 = 12'hfff;
parameter   PRE_PIC_XBD1 = 12'hd00;
parameter   PRE_PIC_XBD2 = 12'hfff;
parameter   PRE_PIC_YBD1 = 12'h000;
parameter   PRE_PIC_YBD2 = 12'h200;
//==========================================================================
// PORT declarations
//==========================================================================
input               iCLK;               // system clock 50Mhz
input               iRST_n;                 // system reset
input       [11:0]  iX_COORD;               // X coordinate form touch panel
input       [11:0]  iY_COORD;               // Y coordinate form touch panel
input               iNEW_COORD;                 // new coordinates indicate
input               iSDRAM_WRITE_EN;        // sdram write enable
output      [2:0]   oPHOTO_CNT;         // displaed photo number
//==========================================================================
// REG/WIRE declarations
//==========================================================================
reg                 mnew_coord;
wire            nextpic_en;
wire            prepic_en;
reg                 nextpic_set;
reg                 prepic_set;
reg         [2:0]   photo_cnt;
//==========================================================================
// Structural coding
//==========================================================================

// if incoming x and y coordinates fit next picture command area , nextpic_en goes high
assign      nextpic_en = ((iX_COORD > NEXT_PIC_XBD1) && (iX_COORD <   NEXT_PIC_XBD2)   &&
                        (iY_COORD > NEXT_PIC_YBD1) && (iY_COORD <   NEXT_PIC_YBD2))
                        ?1:0;
// if incoming x and y coordinates fit previous picture command area , nextpic_en goes high
```

```
assign      prepic_en = ((iX_COORD > PRE_PIC_XBD1) && (iX_COORD <    PRE_PIC_XBD2)    &&
                         (iY_COORD > PRE_PIC_YBD1) && (iY_COORD <    PRE_PIC_YBD2))
                         ?1:0;

always@(posedge iCLK or negedge iRST_n)
    begin
        if (!iRST_n)
            mnew_coord<= 0;
        else
            mnew_coord<= iNEW_COORD;
    end

always@(posedge iCLK or negedge iRST_n)
    begin
        if (!iRST_n)
            nextpic_set <= 0;
        else if (mnew_coord && nextpic_en &&(!iSDRAM_WRITE_EN))
            nextpic_set <= 1;
        else
            nextpic_set <= 0;
    end

always@(posedge iCLK or negedge iRST_n)
    begin
        if (!iRST_n)
            prepic_set <= 0;
        else if (mnew_coord && prepic_en && (!iSDRAM_WRITE_EN))
            prepic_set <= 1;
        else
            prepic_set <= 0;
    end

always@(posedge iCLK or negedge iRST_n)
    begin
        if (!iRST_n)
                photo_cnt <= 0;
        else
            begin
                if (nextpic_set)
                    begin
                        if(photo_cnt == (PHOTO_NUM-1))
                            photo_cnt <= 0;
                        else
                            photo_cnt <= photo_cnt + 1;
```

```
                    end
                if (prepic_set)
                    begin
                        if(photo_cnt == 0)
                            photo_cnt <= (PHOTO_NUM-1);
                        else
                            photo_cnt <= photo_cnt - 1;
                    end
            end
    end

assign      oPHOTO_CNT = photo_cnt;
endmodule
```

adc_spi_controller

```
module adc_spi_controller    (
                    iCLK,
                    iRST_n,
                    oADC_DIN,
                    oADC_DCLK,
                    oADC_CS,
                    iADC_DOUT,
                    iADC_BUSY,
                    iADC_PENIRQ_n,
                    oX_COORD,
                    oY_COORD,
                    oNEW_COORD,

                    );

//=============================================================================
// PARAMETER declarations
//=============================================================================

parameter SYSCLK_FRQ  = 50000000;
parameter ADC_DCLK_FRQ    = 1000;
parameter ADC_DCLK_CNT    = SYSCLK_FRQ/(ADC_DCLK_FRQ*2);

//============================================================================
// PORT declarations
//============================================================================
input               iCLK;
input               iRST_n;
input               iADC_DOUT;
```

```
input			iADC_PENIRQ_n;
input			iADC_BUSY;
output			oADC_DIN;
output			oADC_DCLK;
output			oADC_CS;
output	[11:0]	oX_COORD;
output	[11:0]	oY_COORD;
output			oNEW_COORD;
//=============================================================================
// REG/WIRE declarations
//=============================================================================
reg			d1_PENIRQ_n;
reg			d2_PENIRQ_n;
wire		touch_irq;
reg	[15:0]	dclk_cnt;
wire		dclk;
reg			transmit_en;
reg	[6:0]	spi_ctrl_cnt;
wire		oADC_CS;
reg			mcs;
reg			mdclk;
wire [7:0]	x_config_reg;
wire [7:0]	y_config_reg;
wire [7:0]	ctrl_reg;
reg	[7:0]	mdata_in;
reg			y_coordinate_config;
wire		eof_transmition;
reg	[5:0]	bit_cnt;
reg			madc_out;
reg	[11:0]	mx_coordinate;
reg	[11:0]	my_coordinate;
reg	[11:0]	oX_COORD;
reg	[11:0]	oY_COORD;
wire		rd_coord_strob;
reg			oNEW_COORD;
reg	[5:0]	irq_cnt;
reg	[15:0]	clk_cnt;
//=============================================================================
// Structural coding
//=============================================================================
assign	x_config_reg = 8'h92;
assign	y_config_reg = 8'hd2;

always@(posedge iCLK or negedge iRST_n)
```

```
	begin
		if (!iRST_n)
			madc_out <= 0;
		else
			madc_out <= iADC_DOUT;
	end

///////////////      pen irq detect    ////////
always@(posedge iCLK or negedge iRST_n)
	begin
		if (!iRST_n)
			begin
				d1_PENIRQ_n <= 0;
				d2_PENIRQ_n <= 0;
			end
		else
			begin
				d1_PENIRQ_n <= iADC_PENIRQ_n;
				d2_PENIRQ_n <= d1_PENIRQ_n;
			end
	end

// if iADC_PENIRQ_n form high to low , touch_irq goes high
assign		touch_irq = d2_PENIRQ_n & ~d1_PENIRQ_n;

// if touch_irq goes high , starting transmit procedure ,transmit_en goes high
// if end of transmition and no penirq , transmit procedure stop.

always@(posedge iCLK or negedge iRST_n)
	begin
		if (!iRST_n)
			transmit_en <= 0;
		else if (eof_transmition&&iADC_PENIRQ_n)
			transmit_en <= 0;
		else if (touch_irq)
			transmit_en <= 1;
	end

always@(posedge iCLK or negedge iRST_n)
	begin
		if (!iRST_n)
			dclk_cnt <= 0;
		else if (transmit_en)
			begin
```

```
                if (dclk_cnt == ADC_DCLK_CNT)
                    dclk_cnt <= 0;
                else
                    dclk_cnt <= dclk_cnt + 1;
            end
        else
            dclk_cnt <= 0;
    end

assign  dclk =  (dclk_cnt == ADC_DCLK_CNT)? 1 : 0;

always@(posedge iCLK or negedge iRST_n)
    begin
        if (!iRST_n)
            spi_ctrl_cnt <= 0;
        else if (dclk)
            begin
                if (spi_ctrl_cnt == 49)
                    spi_ctrl_cnt <= 0;
                else
                    spi_ctrl_cnt <= spi_ctrl_cnt + 1;
            end
    end

always@(posedge iCLK or negedge iRST_n)
    begin
        if (!iRST_n)
            begin
                mcs     <= 1;
                mdclk   <= 0;
                mdata_in <= 0;
                y_coordinate_config <= 0;
                mx_coordinate <= 0;
                my_coordinate <= 0;
            end
        else if (transmit_en)
            begin
                if (dclk)
                    begin
                        if (spi_ctrl_cnt == 0)
                            begin
                                mcs     <= 0;
                                mdata_in <= ctrl_reg;
                            end
```

```
					else if (spi_ctrl_cnt == 49)
						begin
							mdclk 	<= 0;
							y_coordinate_config <= ~y_coordinate_config;

							if (y_coordinate_config)
								mcs 		<= 1;
							else
								mcs 		<= 0;
						end
					else if (spi_ctrl_cnt != 0)
						mdclk 	<= ~mdclk;
					if (mdclk)
						mdata_in <= {mdata_in[6:0],1'b0};
					if (!mdclk)
						begin
							if(rd_coord_strob)
								begin
									if(y_coordinate_config)
										my_coordinate 				<=
{my_coordinate[10:0],madc_out};
									else
										mx_coordinate 				<=
{mx_coordinate[10:0],madc_out};
								end
						end
				end
			end
	end

assign 	oADC_CS 	= mcs;
assign 	oADC_DIN = 	mdata_in[7];
assign 	oADC_DCLK = mdclk;
assign 	ctrl_reg = y_coordinate_config ? y_config_reg : x_config_reg;

assign 	eof_transmition = (y_coordinate_config & (spi_ctrl_cnt == 49) & dclk);

assign 	rd_coord_strob = ((spi_ctrl_cnt>=19)&&(spi_ctrl_cnt<=41)) ? 1 : 0;

always@(posedge iCLK or negedge iRST_n)
	begin
		if (!iRST_n)
			begin
				oX_COORD <= 0;
```

```
                oY_COORD <= 0;
            end
        else if (eof_transmition&&(my_coordinate!=0))
            begin
                oX_COORD <= mx_coordinate;
                oY_COORD <= my_coordinate;
            end
    end

always@(posedge iCLK or negedge iRST_n)
    begin
        if (!iRST_n)
            oNEW_COORD <= 0;
        else if (eof_transmition&&(my_coordinate!=0))
            oNEW_COORD <= 1;
        else
            oNEW_COORD <= 0;
    end

endmodule
```

Sdram_Control_4Port/Sdram_Control_4Port

```
module Sdram_Control_4Port(
        //  HOST Side
        REF_CLK,
        RESET_N,
        CLK,
        CLK_33,
        //  FIFO Write Side 1
        WR1_DATA,
        WR1,
        WR1_ADDR,
        WR1_MAX_ADDR,
        WR1_LENGTH,
        WR1_LOAD,
        WR1_CLK,
        WR1_FULL,
        WR1_USE,
        //  FIFO Write Side 2
        WR2_DATA,
        WR2,
        WR2_ADDR,
        WR2_MAX_ADDR,
        WR2_LENGTH,
```

```
		WR2_LOAD,
		WR2_CLK,
		WR2_FULL,
		WR2_USE,
		//	FIFO Read Side 1
		RD1_DATA,
		RD1,
		RD1_ADDR,
		RD1_MAX_ADDR,
		RD1_LENGTH,
		RD1_LOAD,
		RD1_CLK,
		RD1_EMPTY,
		RD1_USE,
		//	FIFO Read Side 2
		RD2_DATA,
		RD2,
		RD2_ADDR,
		RD2_MAX_ADDR,
		RD2_LENGTH,
		RD2_LOAD,
		RD2_CLK,
		RD2_EMPTY,
		RD2_USE,
		//	SDRAM Side
		SA,
		BA,
		CS_N,
		CKE,
		RAS_N,
		CAS_N,
		WE_N,
		DQ,
		DQM,
		SDR_CLK,
		);


`include		"Sdram_Params.h"
//	HOST Side
input					REF_CLK;		//System Clock
input					RESET_N;		//System Reset
//	FIFO Write Side 1
input	[`DSIZE-1:0]		WR1_DATA;		//Data input
```

```
input                               WR1;                        //Write Request
input    [`ASIZE-1:0]            WR1_ADDR;                 //Write start address
input    [`ASIZE-1:0]            WR1_MAX_ADDR;             //Write max address
input    [8:0]                   WR1_LENGTH;               //Write length
input                               WR1_LOAD;                   //Write register load & fifo
clear
input                               WR1_CLK;                    //Write fifo clock
output                              WR1_FULL;                   //Write fifo full
output   [15:0]                     WR1_USE;                    //Write fifo usedw
//   FIFO Write Side 2
input   [`DSIZE-1:0]              WR2_DATA;                  //Data input
input                               WR2;                        //Write Request
input    [`ASIZE-1:0]            WR2_ADDR;                 //Write start address
input    [`ASIZE-1:0]            WR2_MAX_ADDR;             //Write max address
input    [8:0]                   WR2_LENGTH;               //Write length
input                               WR2_LOAD;                   //Write register load & fifo
clear
input                               WR2_CLK;                    //Write fifo clock
output                              WR2_FULL;                   //Write fifo full
output   [15:0]                     WR2_USE;                    //Write fifo usedw
//   FIFO Read Side 1
output  [`DSIZE-1:0]              RD1_DATA;                  //Data output
input                               RD1;                      //Read Request
input    [`ASIZE-1:0]            RD1_ADDR;                 //Read start address
input    [`ASIZE-1:0]            RD1_MAX_ADDR;             //Read max address
input    [8:0]                   RD1_LENGTH;               //Read length
input                               RD1_LOAD;                   //Read register load & fifo
clear
input                               RD1_CLK;                  //Read fifo clock
output                              RD1_EMPTY;                  //Read fifo empty
output   [15:0]                     RD1_USE;                    //Read fifo usedw
//   FIFO Read Side 2
output  [`DSIZE-1:0]              RD2_DATA;                  //Data output
input                               RD2;                      //Read Request
input    [`ASIZE-1:0]            RD2_ADDR;                 //Read start address
input    [`ASIZE-1:0]            RD2_MAX_ADDR;             //Read max address
input    [8:0]                   RD2_LENGTH;               //Read length
input                               RD2_LOAD;                   //Read register load & fifo
clear
input                               RD2_CLK;                  //Read fifo clock
output                              RD2_EMPTY;                  //Read fifo empty
output   [15:0]                     RD2_USE;                    //Read fifo usedw
//   SDRAM Side
output   [11:0]                      SA;                          //SDRAM address output
```

```
output    [1:0]              BA;              //SDRAM bank address
output    [1:0]              CS_N;            //SDRAM Chip Selects
output                       CKE;             //SDRAM clock enable
output                       RAS_N;           //SDRAM  Row  address
Strobe
output                       CAS_N;           //SDRAM  Column  address
Strobe
output                       WE_N;            //SDRAM write enable
inout     [`DSIZE-1:0]       DQ;              //SDRAM data bus
output    [`DSIZE/8-1:0]     DQM;             //SDRAM data mask lines
output                       SDR_CLK;         //SDRAM clock
output                       CLK_33;          //LCD clock
output                       CLK;
//    Internal Registers/Wires
//    Controller

reg       [`ASIZE-1:0]       mADDR;           //Internal address
reg       [8:0]              mLENGTH;         //Internal length
reg       [`ASIZE-1:0]       rWR1_ADDR;       //Register write address

reg       [`ASIZE-1:0]       rWR1_MAX_ADDR;   //Register max write address

reg       [8:0]              rWR1_LENGTH;     //Register write length
reg       [`ASIZE-1:0]       rWR2_ADDR;       //Register write address

reg       [`ASIZE-1:0]       rWR2_MAX_ADDR;   //Register max write address

reg       [8:0]              rWR2_LENGTH;     //Register write length
reg       [`ASIZE-1:0]       rRD1_ADDR;       //Register read address
reg       [`ASIZE-1:0]       rRD1_MAX_ADDR;   //Register max read address
reg       [8:0]              rRD1_LENGTH;     //Register read length
reg       [`ASIZE-1:0]       rRD2_ADDR;       //Register read address
reg       [`ASIZE-1:0]       rRD2_MAX_ADDR;   //Register max read address
reg       [8:0]              rRD2_LENGTH;     //Register read length
reg       [1:0]              WR_MASK;         //Write port active mask
reg       [1:0]              RD_MASK;         //Read port active mask
reg                          mWR_DONE;        //Flag  write  done,  1  pulse
SDR_CLK
reg                          mRD_DONE;        //Flag  read  done,  1  pulse
SDR_CLK
reg                          mWR,Pre_WR;      //Internal    WR    edge
capture
reg                          mRD,Pre_RD;      //Internal RD edge capture
reg   [9:0]                  ST;              //Controller status
```

```
reg     [1:0]              CMD;              //Controller command
reg                        PM_STOP;          //Flag page mode stop
reg                        PM_DONE;          //Flag page mode done
reg                        Read;             //Flag read active
reg                        Write;            //Flag write active
reg     [`DSIZE-1:0]       mDATAOUT;         //Controller Data output
wire    [`DSIZE-1:0]       mDATAIN;          //Controller Data input
wire    [`DSIZE-1:0]       mDATAIN1;         //Controller Data input 1
wire    [`DSIZE-1:0]       mDATAIN2;         //Controller Data input 2
wire                       CMDACK;           //Controller command
acknowledgement
//  DRAM Control
reg     [`DSIZE/8-1:0]     DQM;              //SDRAM data mask lines
reg     [11:0]             SA;               //SDRAM address output
reg     [1:0]              BA;               //SDRAM bank address
reg     [1:0]              CS_N;             //SDRAM Chip Selects
reg                        CKE;              //SDRAM clock enable
reg                        RAS_N;            //SDRAM Row address
Strobe
reg                        CAS_N;            //SDRAM Column address
Strobe
reg                        WE_N;             //SDRAM write enable
wire    [`DSIZE-1:0]       DQOUT;            //SDRAM data out link
wire    [`DSIZE/8-1:0]     IDQM;             //SDRAM data mask lines
wire    [11:0]             ISA;              //SDRAM address output
wire    [1:0]              IBA;              //SDRAM bank address
wire    [1:0]              ICS_N;            //SDRAM Chip Selects
wire                       ICKE;             //SDRAM clock enable
wire                       IRAS_N;           //SDRAM Row address
Strobe
wire                       ICAS_N;           //SDRAM Column address
Strobe
wire                       IWE_N;            //SDRAM write enable
//  FIFO Control
reg                        OUT_VALID;        //Output data request to
read side fifo
reg                        IN_REQ;           //Input data request to
write side fifo
wire [15:0]                write_side_fifo_rusedw1;
wire [15:0]                read_side_fifo_wusedw1;
wire [15:0]                write_side_fifo_rusedw2;
wire [15:0]                read_side_fifo_wusedw2;
//  DRAM Internal Control
wire    [`ASIZE-1:0]       saddr;
```

```
wire                                    load_mode;
wire                                    nop;
wire                                    reada;
wire                                    writea;
wire                                    refresh;
wire                                    precharge;
wire                                    oe;
wire                               ref_ack;
wire                               ref_req;
wire                               init_req;
wire                               cm_ack;
wire                               active;



Sdram_PLL sdram_pll1   (
                    .inclk0(REF_CLK),
                    .c0(CLK),
                    .c1(SDR_CLK),
                    .c2(CLK_33)
                    );

control_interface control1 (
                    .CLK(CLK),
                    .RESET_N(RESET_N),
                    .CMD(CMD),
                    .ADDR(mADDR),
                    .REF_ACK(ref_ack),
                    .CM_ACK(cm_ack),
                    .NOP(nop),
                    .READA(reada),
                    .WRITEA(writea),
                    .REFRESH(refresh),
                    .PRECHARGE(precharge),
                    .LOAD_MODE(load_mode),
                    .SADDR(saddr),
                    .REF_REQ(ref_req),
                    .INIT_REQ(init_req),
                    .CMD_ACK(CMDACK)
                    );

command command1(
                    .CLK(CLK),
```

```
                        .RESET_N(RESET_N),
                        .SADDR(saddr),
                        .NOP(nop),
                        .READA(reada),
                        .WRITEA(writea),
                        .REFRESH(refresh),
                        .LOAD_MODE(load_mode),
                        .PRECHARGE(precharge),
                        .REF_REQ(ref_req),
                        .INIT_REQ(init_req),
                        .REF_ACK(ref_ack),
                        .CM_ACK(cm_ack),
                        .OE(oe),
                        .PM_STOP(PM_STOP),
                        .PM_DONE(PM_DONE),
                        .SA(ISA),
                        .BA(IBA),
                        .CS_N(ICS_N),
                        .CKE(ICKE),
                        .RAS_N(IRAS_N),
                        .CAS_N(ICAS_N),
                        .WE_N(IWE_N)
                        );

sdr_data_path data_path1(
                        .CLK(CLK),
                        .RESET_N(RESET_N),
                        .DATAIN(mDATAIN),
                        .DM(2'b00),
                        .DQOUT(DQOUT),
                        .DQM(IDQM)
                        );

Sdram_WR_FIFO           write_fifo1(
                        .data(WR1_DATA),
                        .wrreq(WR1),
                        .wrclk(WR1_CLK),
                        .aclr(WR1_LOAD),
                        .rdreq(IN_REQ&WR_MASK[0]),
                        .rdclk(CLK),
                        .q(mDATAIN1),
                        .wrfull(WR1_FULL),
                        .wrusedw(WR1_USE),
                        .rdusedw(write_side_fifo_rusedw1)
```

```
                );

Sdram_WR_FIFO   write_fifo2(
                .data(WR2_DATA),
                .wrreq(WR2),
                .wrclk(WR2_CLK),
                .aclr(WR2_LOAD),
                .rdreq(IN_REQ&WR_MASK[1]),
                .rdclk(CLK),
                .q(mDATAIN2),
                .wrfull(WR2_FULL),
                .wrusedw(WR2_USE),
                .rdusedw(write_side_fifo_rusedw2)
                );

assign    mDATAIN =    (WR_MASK[0])    ?    mDATAIN1    :
                                            mDATAIN2    ;

Sdram_RD_FIFO   read_fifo1(
                .data(mDATAOUT),
                .wrreq(OUT_VALID&RD_MASK[0]),
                .wrclk(CLK),
                .aclr(RD1_LOAD),
                .rdreq(RD1),
                .rdclk(RD1_CLK),
                .q(RD1_DATA),
                .wrusedw(read_side_fifo_wusedw1),
                .rdempty(RD1_EMPTY),
                .rdusedw(RD1_USE)
                );

Sdram_RD_FIFO   read_fifo2(
                .data(mDATAOUT),
                .wrreq(OUT_VALID&RD_MASK[1]),
                .wrclk(CLK),
                .aclr(RD2_LOAD),
                .rdreq(RD2),
                .rdclk(RD2_CLK),
                .q(RD2_DATA),
                .wrusedw(read_side_fifo_wusedw2),
                .rdempty(RD2_EMPTY),
                .rdusedw(RD2_USE)
                );
```

```
always @(posedge CLK)
begin
	SA		<= (ST==SC_CL+mLENGTH)		?	12'h200	:	ISA;
	BA		<= IBA;
	CS_N		<= ICS_N;
	CKE		<= ICKE;
	RAS_N	<= (ST==SC_CL+mLENGTH)		?	1'b0 :	IRAS_N;
	CAS_N	<= (ST==SC_CL+mLENGTH)		?	1'b1 :	ICAS_N;
	WE_N		<= (ST==SC_CL+mLENGTH)		?	1'b0 :	IWE_N;
	PM_STOP<= (ST==SC_CL+mLENGTH)		?	1'b1 :	1'b0;
	PM_DONE	<= (ST==SC_CL+SC_RCD+mLENGTH+2)	?	1'b1 :	1'b0;
	DQM		<= ( active && (ST>=SC_CL) )	?	(	((ST==SC_CL+mLENGTH)	&&	Write)?
	2'b11	:	2'b00	)	:	2'b11	;
	mDATAOUT<= DQ;
end

assign	DQ = oe ? DQOUT : `DSIZE'hzzzz;
assign	active	=	Read | Write;

always@(posedge CLK or negedge RESET_N)
begin
	if(RESET_N==0)
	begin
		CMD			<=	0;
		ST			<=	0;
		Pre_RD		<=	0;
		Pre_WR		<=	0;
		Read			<=	0;
		Write		<=	0;
		OUT_VALID	<=	0;
		IN_REQ		<=	0;
		mWR_DONE	<=	0;
		mRD_DONE	<=	0;
	end
	else
	begin
		Pre_RD	<=	mRD;
		Pre_WR	<=	mWR;
		case(ST)
		0:	begin
				if({Pre_RD,mRD}==2'b01)
				begin
					Read		<=	1;
					Write	<=	0;
```

```
			CMD		<=	2'b01;
			ST		<=	1;
		end
		else if({Pre_WR,mWR}==2'b01)
		begin
			Read		<=	0;
			Write		<=	1;
			CMD		<=	2'b10;
			ST		<=	1;
		end
	end
1:	begin
		if(CMDACK==1)
		begin
			CMD<=2'b00;
			ST<=2;
		end
	end
default:
	begin
		if(ST!=SC_CL+SC_RCD+mLENGTH+1)
		ST<=ST+1;
		else
		ST<=0;
	end
endcase

if(Read)
begin
	if(ST==SC_CL+SC_RCD+1)
	OUT_VALID	<=	1;
	else if(ST==SC_CL+SC_RCD+mLENGTH+1)
	begin
		OUT_VALID	<=	0;
		Read		<=	0;
		mRD_DONE	<=	1;
	end
end
else
mRD_DONE	<=	0;

if(Write)
begin
	if(ST==SC_CL-1)
```

```
			IN_REQ	<=	1;
			else if(ST==SC_CL+mLENGTH-1)
			IN_REQ	<=	0;
			else if(ST==SC_CL+SC_RCD+mLENGTH)
			begin
				Write	<=	0;
				mWR_DONE<=	1;
			end
		end
		else
		mWR_DONE<=	0;

	end
end
//	Internal Address & Length Control
always@(posedge CLK or negedge RESET_N)
begin
	if(!RESET_N)
	begin
		rWR1_ADDR		<=	0;
		rWR1_MAX_ADDR	<=	800*480;
		rWR2_ADDR		<=	22'h100000;
		rWR2_MAX_ADDR	<=	22'h100000+800*480;

		rRD1_ADDR		<=	0;
		rRD1_MAX_ADDR	<=	800*480;
		rRD2_ADDR		<=	22'h100000;
		rRD2_MAX_ADDR	<=	22'h100000+800*480;


		rWR1_LENGTH		<=	128;
		rRD1_LENGTH		<=	128;
		rWR2_LENGTH		<=	128;
		rRD2_LENGTH		<=	128;
	end
	else
	begin
		//	Write Side 1
		if(WR1_LOAD)
		begin
			rWR1_ADDR	<=	WR1_ADDR;
			rWR1_LENGTH	<=	WR1_LENGTH;
		end
		else if(mWR_DONE&WR_MASK[0])
```

```
begin
	if(rWR1_ADDR<rWR1_MAX_ADDR-rWR1_LENGTH)
	rWR1_ADDR	<=	rWR1_ADDR+rWR1_LENGTH;
	else
	rWR1_ADDR	<=	WR1_ADDR;
end
//	Write Side 2
if(WR2_LOAD)
begin
	rWR2_ADDR	<=	WR2_ADDR;
	rWR2_LENGTH		<=	WR2_LENGTH;
end
else if(mWR_DONE&WR_MASK[1])
begin
	if(rWR2_ADDR<rWR2_MAX_ADDR-rWR2_LENGTH)
	rWR2_ADDR	<=	rWR2_ADDR+rWR2_LENGTH;
	else
	rWR2_ADDR	<=	WR2_ADDR;
end
//	Read Side 1
if(RD1_LOAD)
begin
	rRD1_ADDR	<=	RD1_ADDR;
	rRD1_LENGTH	<=	RD1_LENGTH;
end
else if(mRD_DONE&RD_MASK[0])
begin
	if(rRD1_ADDR<rRD1_MAX_ADDR-rRD1_LENGTH)
	rRD1_ADDR	<=	rRD1_ADDR+rRD1_LENGTH;
	else
	rRD1_ADDR	<=	RD1_ADDR;
end
//	Read Side 2
if(RD2_LOAD)
begin
	rRD2_ADDR	<=	RD2_ADDR;
	rRD2_LENGTH	<=	RD2_LENGTH;
end
else if(mRD_DONE&RD_MASK[1])
begin
	if(rRD2_ADDR<rRD2_MAX_ADDR-rRD2_LENGTH)
	rRD2_ADDR	<=	rRD2_ADDR+rRD2_LENGTH;
	else
	rRD2_ADDR	<=	RD2_ADDR;
```

```
			end
		end
end
//	Auto Read/Write Control
always@(posedge CLK or negedge RESET_N)
begin
	if(!RESET_N)
	begin
		mWR		<=	0;
		mRD	<=	0;
		mADDR	<=	0;
		mLENGTH	<=	0;
	end
	else
	begin
		if( (mWR==0) && (mRD==0) && (ST==0) &&
			(WR_MASK==0)	&&	(RD_MASK==0) &&
			(WR1_LOAD==0)	&&	(RD1_LOAD==0) &&
			(WR2_LOAD==0)	&&	(RD2_LOAD==0) )
		begin
			//	Read Side 1
			if( (read_side_fifo_wusedw1 < rRD1_LENGTH) )
			begin
				mADDR	<=	rRD1_ADDR;
				mLENGTH	<=	rRD1_LENGTH;
				WR_MASK	<=	2'b00;
				RD_MASK	<=	2'b01;
				mWR		<=	0;
				mRD	<=	1;
			end
			//	Read Side 2
			else if( (read_side_fifo_wusedw2 < rRD2_LENGTH) )
			begin
				mADDR	<=	rRD2_ADDR;
				mLENGTH	<=	rRD2_LENGTH;
				WR_MASK	<=	2'b00;
				RD_MASK	<=	2'b10;
				mWR		<=	0;
				mRD	<=	1;
			end
			//	Write Side 1
			else if( (write_side_fifo_rusedw1 >= rWR1_LENGTH) && (rWR1_LENGTH!=0) )
			begin
				mADDR	<=	rWR1_ADDR;
```

```
				mLENGTH		<=	rWR1_LENGTH;
				WR_MASK		<=	2'b01;
				RD_MASK		<=	2'b00;
				mWR			<=	1;
				mRD		<=	0;
			end
			//	Write Side 2
			else if( (write_side_fifo_rusedw2 >= rWR2_LENGTH) && (rWR2_LENGTH!=0) )
			begin
				mADDR	<=	rWR2_ADDR;
				mLENGTH		<=	rWR2_LENGTH;
				WR_MASK		<=	2'b10;
				RD_MASK		<=	2'b00;
				mWR			<=	1;
				mRD		<=	0;
			end
		end
		if(mWR_DONE)
		begin
			WR_MASK		<=	0;
			mWR			<=	0;
		end
		if(mRD_DONE)
		begin
			RD_MASK		<=	0;
			mRD		<=	0;
		end
	end
end
endmodule
```

Sdram_Control_4Port/sdr_data_path

```
module sdr_data_path(
		CLK,
		RESET_N,
		DATAIN,
		DM,
		DQOUT,
		DQM
		);

`include		"Sdram_Params.h"

input					CLK;				// System Clock
```

```
input                                   RESET_N;                        // System Reset
input     [`DSIZE-1:0]                  DATAIN;                         // Data input from the host
input     [`DSIZE/8-1:0]                DM;                             // byte data masks
output    [`DSIZE-1:0]                  DQOUT;
output    [`DSIZE/8-1:0]                DQM;                            // SDRAM data mask ouputs
reg       [`DSIZE/8-1:0]                DQM;



// Allign the input and output data to the SDRAM control path
always @(posedge CLK or negedge RESET_N)
begin
          if (RESET_N == 0)
          DQM           <= `DSIZE/8-1'hF;
          else
          DQM           <=    DM;
end

assign DQOUT = DATAIN;

endmodule
```

Sdram_Control_4Port/control_interface

```
module control_interface(
          CLK,
          RESET_N,
          CMD,
          ADDR,
          REF_ACK,
          INIT_ACK,
          CM_ACK,
          NOP,
          READA,
          WRITEA,
          REFRESH,
          PRECHARGE,
          LOAD_MODE,
          SADDR,
          REF_REQ,
          INIT_REQ,
          CMD_ACK
          );

`include            "Sdram_Params.h"
```

```
input                               CLK;                    // System Clock
input                               RESET_N;                // System Reset
input      [2:0]                    CMD;                    // Command input
input      [`ASIZE-1:0]             ADDR;                   // Address
input                               REF_ACK;                // Refresh request
acknowledge
input                               INIT_ACK;               // Initial request
acknowledge
input                               CM_ACK;                 // Command acknowledge
output                              NOP;                    // Decoded NOP command
output                              READA;                  // Decoded READA
command
output                              WRITEA;                 // Decoded WRITEA
command
output                              REFRESH;                // Decoded REFRESH
command
output                              PRECHARGE;              // Decoded PRECHARGE
command
output                              LOAD_MODE;              // Decoded LOAD_MODE
command
output     [`ASIZE-1:0]             SADDR;                  // Registered version of ADDR
output                              REF_REQ;                // Hidden refresh request
output                              INIT_REQ;               // Hidden initial request
output                              CMD_ACK;                // Command acknowledge



reg                                 NOP;
reg                                 READA;
reg                                 WRITEA;
reg                                 REFRESH;
reg                                 PRECHARGE;
reg                                 LOAD_MODE;
reg        [`ASIZE-1:0]             SADDR;
reg                                 REF_REQ;
reg                                 INIT_REQ;
reg                                 CMD_ACK;

// Internal signals
reg        [15:0]                   timer;
reg        [15:0]                   init_timer;
```

```
// Command decode and ADDR register
always @(posedge CLK or negedge RESET_N)
begin
	if (RESET_N == 0)
	begin
		NOP			<= 0;
		READA			<= 0;
		WRITEA			<= 0;
		SADDR			<= 0;
	end

	else
	begin

		SADDR  <=  ADDR;				//  register  the
address to keep proper
								//  alignment  with
the command

		if (CMD == 3'b000)				// NOP command
			NOP <= 1;
		else
			NOP <= 0;

		if (CMD == 3'b001)				// READA command
			READA <= 1;
		else
			READA <= 0;

		if (CMD == 3'b010)				// WRITEA command
			WRITEA <= 1;
		else
			WRITEA <= 0;

	end
end


//	Generate CMD_ACK
always @(posedge CLK or negedge RESET_N)
begin
	if (RESET_N == 0)
		CMD_ACK <= 0;
```

```
		else
			if ((CM_ACK == 1) & (CMD_ACK == 0))
				CMD_ACK <= 1;
			else
				CMD_ACK <= 0;
end


// refresh timer
always @(posedge CLK or negedge RESET_N) begin
		if (RESET_N == 0)
		begin
			timer			<= 0;
			REF_REQ			<= 0;
		end
		else
		begin
			if (REF_ACK == 1)
			begin
			    timer <= REF_PER;
			    REF_REQ <=0;
			end
			else if (INIT_REQ == 1)
			begin
			    timer <= REF_PER+200;
			    REF_REQ <=0;
			end
			else
			    timer <= timer - 1'b1;

			if (timer==0)
			    REF_REQ	<= 1;

		end
end

// initial timer
always @(posedge CLK or negedge RESET_N) begin
		if (RESET_N == 0)
		begin
			init_timer		<= 0;
			REFRESH			<= 0;
			PRECHARGE		<= 0;
			LOAD_MODE		<= 0;
```

```
		INIT_REQ	<= 0;
	end
	else
	begin
		if (init_timer < (INIT_PER+201))
			init_timer	<= init_timer+1;

		if (init_timer < INIT_PER)
		begin
			REFRESH		<=0;
			PRECHARGE	<=0;
			LOAD_MODE	<=0;
			INIT_REQ	<=1;
		end
		else if(init_timer == (INIT_PER+20))
		begin
			REFRESH		<=0;
			PRECHARGE	<=1;
			LOAD_MODE	<=0;
			INIT_REQ	<=0;
		end
		else if(	(init_timer == (INIT_PER+40))	||
				(init_timer == (INIT_PER+60))	||
				(init_timer == (INIT_PER+80))	||
				(init_timer == (INIT_PER+100))	||
				(init_timer == (INIT_PER+120))	||
				(init_timer == (INIT_PER+140))	||
				(init_timer == (INIT_PER+160))	||
				(init_timer == (INIT_PER+180))	)
		begin
			REFRESH		<=1;
			PRECHARGE	<=0;
			LOAD_MODE	<=0;
			INIT_REQ	<=0;
		end
		else if(init_timer == (INIT_PER+200))
		begin
			REFRESH		<=0;
			PRECHARGE	<=0;
			LOAD_MODE	<=1;
			INIT_REQ	<=0;
		end
		else
		begin
```

```
                REFRESH     <=0;
                PRECHARGE   <=0;
                LOAD_MODE   <=0;
                INIT_REQ<=0;
            end
        end
end
endmodule
```

Sdram_Control_4Port/command

```
module command(
        CLK,
        RESET_N,
        SADDR,
        NOP,
        READA,
        WRITEA,
        REFRESH,
        PRECHARGE,
        LOAD_MODE,
        REF_REQ,
        INIT_REQ,
        PM_STOP,
        PM_DONE,
        REF_ACK,
        CM_ACK,
        OE,
        SA,
        BA,
        CS_N,
        CKE,
        RAS_N,
        CAS_N,
        WE_N
        );

`include        "Sdram_Params.h"

input                       CLK;                // System Clock
input                       RESET_N;            // System Reset
input   [`ASIZE-1:0]        SADDR;              // Address
input                       NOP;                // Decoded NOP command
input                       READA;              //  Decoded  READA
command
```

```
input                    WRITEA;                // Decoded WRITEA
command
input                    REFRESH;               // Decoded REFRESH
command
input                    PRECHARGE;             // Decoded PRECHARGE
command
input                    LOAD_MODE;             // Decoded LOAD_MODE
command
input                    REF_REQ;               // Hidden refresh request
input                    INIT_REQ;              // Hidden initial request
input                    PM_STOP;               // Page mode stop
input                    PM_DONE;               // Page mode done
output                   REF_ACK;               // Refresh request
acknowledge
output                   CM_ACK;                // Command acknowledge
output                   OE;                    // OE signal for data path
module
output   [11:0]          SA;                    // SDRAM address
output   [1:0]           BA;                    // SDRAM bank address
output   [1:0]           CS_N;                  // SDRAM chip selects
output                   CKE;                   // SDRAM clock enable
output                   RAS_N;                 // SDRAM RAS
output                   CAS_N;                 // SDRAM CAS
output                   WE_N;                  // SDRAM WE_N


reg                      CM_ACK;
reg                      REF_ACK;
reg                      OE;
reg      [11:0]          SA;
reg      [1:0]           BA;
reg      [1:0]           CS_N;
reg                      CKE;
reg                      RAS_N;
reg                      CAS_N;
reg                      WE_N;



// Internal signals
reg                      do_reada;
reg                      do_writea;
reg                      do_refresh;
reg                      do_precharge;
```

```
reg                                    do_load_mode;
reg                                    do_initial;
reg                                    command_done;
reg        [7:0]                       command_delay;
reg        [1:0]                       rw_shift;
reg                                    do_act;
reg                                    rw_flag;
reg                                    do_rw;
reg        [6:0]                       oe_shift;
reg                                    oe1;
reg                                    oe2;
reg                                    oe3;
reg                                    oe4;
reg        [3:0]                       rp_shift;
reg                                    rp_done;
reg                                    ex_read;
reg                                    ex_write;

wire       [`ROWSIZE - 1:0]            rowaddr;
wire       [`COLSIZE - 1:0]            coladdr;
wire       [`BANKSIZE - 1:0]           bankaddr;

assign   rowaddr    = SADDR[`ROWSTART + `ROWSIZE - 1: `ROWSTART];          //
assignment of the row address bits from SADDR
assign   coladdr    = SADDR[`COLSTART + `COLSIZE - 1:`COLSTART];           // assignment
of the column address bits
assign   bankaddr   = SADDR[`BANKSTART + `BANKSIZE - 1:`BANKSTART];        //
assignment of the bank address bits



// This always block monitors the individual command lines and issues a command
// to the next stage if there currently another command already running.
//
always @(posedge CLK or negedge RESET_N)
begin
        if (RESET_N == 0)
        begin
                do_reada        <= 0;
                do_writea       <= 0;
                do_refresh      <= 0;
                do_precharge    <= 0;
                do_load_mode    <= 0;
                do_initial      <= 0;
```

```
                command_done        <= 0;
                command_delay       <= 0;
                rw_flag             <= 0;
                rp_shift            <= 0;
                rp_done             <= 0;
                ex_read             <= 0;
                ex_write            <= 0;
        end

        else
        begin

//  Issue the appropriate command if the sdram is not currently busy
            if( INIT_REQ == 1 )
            begin
                do_reada            <= 0;
                do_writea           <= 0;
                do_refresh          <= 0;
                do_precharge        <= 0;
                do_load_mode        <= 0;
                do_initial          <= 1;
                command_done        <= 0;
                command_delay       <= 0;
                rw_flag             <= 0;
                rp_shift            <= 0;
                rp_done             <= 0;
                ex_read             <= 0;
                ex_write            <= 0;
            end
            else
            begin
                do_initial          <= 0;

                if ((REF_REQ == 1 | REFRESH == 1) & command_done == 0 & do_refresh == 0
& rp_done == 0              // Refresh
                        & do_reada == 0 & do_writea == 0)
                        do_refresh <= 1;
                else
                        do_refresh <= 0;

                if ((READA == 1) & (command_done == 0) & (do_reada == 0) & (rp_done == 0)
& (REF_REQ == 0))       // READA
                begin
                        do_reada <= 1;
```

```
				ex_read <= 1;
			end
			else
				do_reada <= 0;

			if ((WRITEA == 1) & (command_done == 0) & (do_writea == 0) & (rp_done ==
0) & (REF_REQ == 0))    // WRITEA
			begin
				do_writea <= 1;
				ex_write <= 1;
			end
			else
				do_writea <= 0;

			if ((PRECHARGE == 1) & (command_done == 0) & (do_precharge == 0))
// PRECHARGE
				do_precharge <= 1;
			else
				do_precharge <= 0;

			if ((LOAD_MODE == 1) & (command_done == 0) & (do_load_mode == 0))
// LOADMODE
				do_load_mode <= 1;
			else
				do_load_mode <= 0;

// set command_delay shift register and command_done flag
// The command delay shift register is a timer that is used to ensure that
// the SDRAM devices have had sufficient time to finish the last command.

			if ((do_refresh == 1) | (do_reada == 1) | (do_writea == 1) | (do_precharge ==
1)
			   | (do_load_mode == 1))
			begin
				command_delay <= 8'b11111111;
				command_done    <= 1;
				rw_flag <= do_reada;
			end

			else
			begin
				command_done                    <=     command_delay[0];
// the command_delay shift operation
				command_delay           <= (command_delay>>1);
```

```
                        end


 // start additional timer that is used for the refresh, writea, reada commands
                        if (command_delay[0] == 0 & command_done == 1)
                        begin
                                rp_shift <= 4'b1111;
                                rp_done <= 1;
                        end
                        else
                        begin
                                if(SC_PM == 0)
                                begin
                                        rp_shift    <= (rp_shift>>1);
                                        rp_done         <= rp_shift[0];
                                end
                                else
                                begin
                                        if( (ex_read == 0) && (ex_write == 0) )
                                        begin
                                                rp_shift    <= (rp_shift>>1);
                                                rp_done         <= rp_shift[0];
                                        end
                                        else
                                        begin
                                                if( PM_STOP==1 )
                                                begin
                                                        rp_shift    <= (rp_shift>>1);
                                                        rp_done         <= rp_shift[0];
                                                        ex_read         <= 1'b0;
                                                        ex_write  <= 1'b0;
                                                end
                                        end
                                end
                        end
                end
        end
end


// logic that generates the OE signal for the data path module
// For normal burst write he duration of OE is dependent on the configured burst length.
// For page mode accesses(SC_PM=1) the OE signal is turned on at the start of the write
command
```

```
// and is left on until a PRECHARGE(page burst terminate) is detected.
//
always @(posedge CLK or negedge RESET_N)
begin
        if (RESET_N == 0)
        begin
                oe_shift <= 0;
                oe1      <= 0;
                oe2      <= 0;
                OE       <= 0;
        end
        else
        begin
                if (SC_PM == 0)
                begin
                        if (do_writea == 1)
                        begin
                                if (SC_BL == 1)                         // Set the
shift register to the appropriate
                                        oe_shift <= 0;                  // value based
on burst length.
                                else if (SC_BL == 2)
                                        oe_shift <= 1;
                                else if (SC_BL == 4)
                                        oe_shift <= 7;
                                else if (SC_BL == 8)
                                        oe_shift <= 127;
                                oe1 <= 1;
                        end
                        else
                        begin
                                oe_shift <= (oe_shift>>1);
                                oe1   <= oe_shift[0];
                                oe2   <= oe1;
                                oe3   <= oe2;
                                oe4   <= oe3;
                                if (SC_RCD == 2)
                                        OE <= oe3;
                                else
                                        OE <= oe4;
                        end
                end
                else
                begin
```

```
                              if  (do_writea  ==  1)                                        //
OE generation for page mode accesses
                                        oe4      <= 1;
                              else  if  (do_precharge  ==  1  |  do_reada  ==  1  |  do_refresh==1  |
do_initial == 1 | PM_STOP==1 )
                                        oe4      <= 0;
                              OE <= oe4;
                    end

          end
end




// This always block tracks the time between the activate command and the
// subsequent WRITEA or READA command, RC.    The shift register is set using
// the configuration register setting SC_RCD. The shift register is loaded with
// a single '1' with the position within the register dependent on SC_RCD.
// When the '1' is shifted out of the register it sets so_rw which triggers
// a writea or reada command
//
always @(posedge CLK or negedge RESET_N)
begin
          if (RESET_N == 0)
          begin
                    rw_shift <= 0;
                    do_rw        <= 0;
          end

          else
          begin

                    if ((do_reada == 1) | (do_writea == 1))
                    begin
                              if  (SC_RCD  ==  1)                              //  Set  the  shift
register
                                        do_rw <= 1;
                              else if (SC_RCD == 2)
                                        rw_shift <= 1;
                              else if (SC_RCD == 3)
                                        rw_shift <= 2;
                    end
                    else
```

```
			begin
				rw_shift <= (rw_shift>>1);
				do_rw       <= rw_shift[0];
			end
		end
end

// This always block generates the command acknowledge, CM_ACK, signal.
// It also generates the acknowledge signal, REF_ACK, that acknowledges
// a refresh request that was generated by the internal refresh timer circuit.
always @(posedge CLK or negedge RESET_N)
begin

		if (RESET_N == 0)
		begin
			CM_ACK      <= 0;
			REF_ACK    <= 0;
		end

		else
		begin
			if (do_refresh == 1 & REF_REQ == 1)                               // Internal refresh
timer refresh request
				REF_ACK <= 1;
			else  if  ((do_refresh  ==  1)  |  (do_reada  ==  1)  |  (do_writea  ==  1)  |
(do_precharge == 1)      // externa    commands
				  | (do_load_mode))
				CM_ACK <= 1;
			else
			begin
				REF_ACK <= 0;
				CM_ACK    <= 0;
			end
		end
end




// This always block generates the address, cs, cke, and command signals(ras,cas,wen)
//
```

```
always @(posedge CLK ) begin
        if (RESET_N==0) begin
                SA       <= 0;
                BA       <= 0;
                CS_N    <= 1;
                RAS_N <= 1;
                CAS_N <= 1;
                WE_N    <= 1;
                CKE      <= 0;
        end
        else begin
                CKE <= 1;

// Generate SA
                if (do_writea == 1 | do_reada == 1)       // ACTIVATE command is being issued,
so present the row address
                        SA <= rowaddr;
                else
                        SA <= coladdr;                          // else alway present column
address
                if ((do_rw==1) | (do_precharge))
                        SA[10] <= !SC_PM;                    // set SA[10] for autoprecharge
read/write or for a precharge all command
                                                                  // don't set it if the controller
is in page mode.
                if (do_precharge==1 | do_load_mode==1)
                        BA <= 0;                                  // Set BA=0 if performing a
precharge or load_mode command
                else
                        BA <= bankaddr[1:0];                 // else set it with the appropriate
address bits

                if (do_refresh==1 | do_precharge==1 | do_load_mode==1 | do_initial==1)
                        CS_N <= 0;                                                          // Select
both chip selects if performing
                else                                                                          // refresh,
precharge(all) or load_mode
                begin
                        CS_N[0] <= SADDR[`ASIZE-1];                           // else set the
chip selects based off of the
                        CS_N[1] <= ~SADDR[`ASIZE-1];                         // msb address
bit
                end
```

```
                if(do_load_mode==1)
                SA        <= {2'b00,SDR_CL,SDR_BT,SDR_BL};


//Generate the appropriate logic levels on RAS_N, CAS_N, and WE_N
//depending on the issued command.
//
                if ( do_refresh==1 ) begin                              // Refresh: S=00,
RAS=0, CAS=0, WE=1
                        RAS_N <= 0;
                        CAS_N <= 0;
                        WE_N   <= 1;
                end
                else if ((do_precharge==1) & ((oe4 == 1) | (rw_flag == 1))) begin          //
burst terminate if write is active
                        RAS_N <= 1;
                        CAS_N <= 1;
                        WE_N   <= 0;
                end
                else if (do_precharge==1) begin                         // Precharge All: S=00,
RAS=0, CAS=1, WE=0
                        RAS_N <= 0;
                        CAS_N <= 1;
                        WE_N   <= 0;
                end
                else if (do_load_mode==1) begin                         // Mode Write: S=00,
RAS=0, CAS=0, WE=0
                        RAS_N <= 0;
                        CAS_N <= 0;
                        WE_N   <= 0;
                end
                else if (do_reada == 1 | do_writea == 1) begin   // Activate: S=01 or 10,
RAS=0, CAS=1, WE=1
                        RAS_N <= 0;
                        CAS_N <= 1;
                        WE_N   <= 1;
                end
                else if (do_rw == 1) begin                              // Read/Write: S=01 or
10, RAS=1, CAS=0, WE=0 or 1
                        RAS_N <= 1;
                        CAS_N <= 0;
                        WE_N   <= rw_flag;
                end
                else if (do_initial ==1) begin
```

```
                RAS_N <= 1;
                CAS_N <= 1;
                WE_N    <= 1;
            end
            else  begin                                          //  No  Operation:
RAS=1, CAS=1, WE=1
                RAS_N <= 1;
                CAS_N <= 1;
                WE_N    <= 1;
            end
        end
end
endmodule
```