

Abstract:

One of the first digital games created was a two-dimensional electronic table tennis game, which was named Pong. This project creates an implementation of Pong which is controlled by two players using knobs, and displays the game on a 640x480 pixel VGA monitor. The knobs control potentiometers, which send an analog signal to the analog ports of the microcontroller. The microcontroller handles the game mechanics, including tracking the position of the ball and paddles, and keeping score. Using parallel ports, the microcontroller communicates the positions and score to the FPGA, which sends the necessary signals to a VGA monitor in order to display the game.

Introduction:

This system is an implementation of the game of Pong on the PIC microcontroller and Xilinx II FPGA. This game is a two-dimensional digital table tennis game in which a ball bounces between two paddles, each one controlled by one of the players, and off the top and bottom of the rectangular playing area.

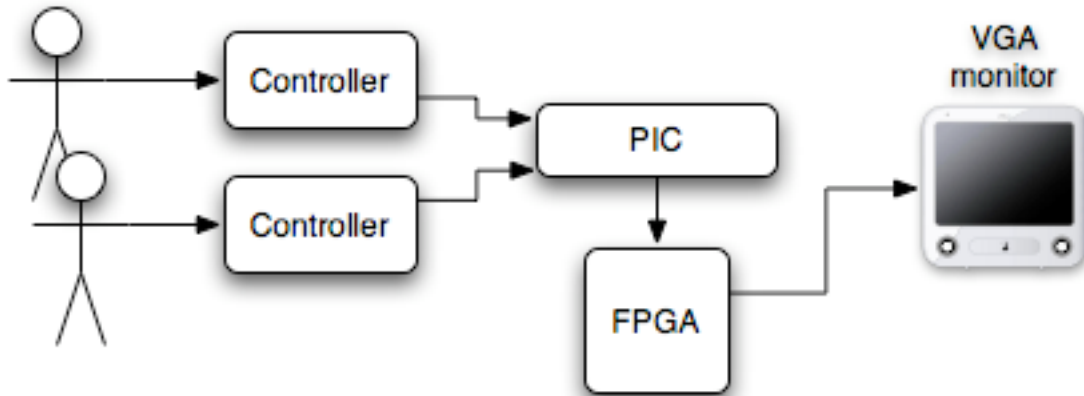
The players control their paddle using knobs attached to potentiometers. These potentiometers are powered by the 3.3 V output from the Harrisboard. The controllers are connected to the board across RJ-11 cables, in which we use 3 of the wires to carry the ground, high voltage, and potentiometer output. The output voltages from the potentiometers are directly connected to two of the analog ports of the PIC microcontroller.

The PIC then uses its analog to digital converter to determine the position of each of the paddles as directed by the current position of the knobs on the potentiometers. The PIC also determines the current position of the ball based on its previous position and its current velocity. The vertical velocity of the ball is modified by the movement of each paddle when the ball strikes them and is inverted whenever the ball strikes the top or bottom edge of the play area. The horizontal velocity of the ball is inverted and its increased in magnitude each time the ball strikes a paddle.

The PIC communicates the data for position and size of all play elements as well as the score of the game to the FPGA using Ports C and D for parallel communication. Of these sixteen pins, six of them are used to determine the use of the current value being sent, and the other ten carry the value. The FPGA uses these values to determine the color of each pixel of the screen as it needs to be displayed. The outputs of the FPGA are three bits of

color to the VGA monitor and two bits of HSync and VSync which are determined by the FPGA and are used to coordinate the FPGA and the monitor.

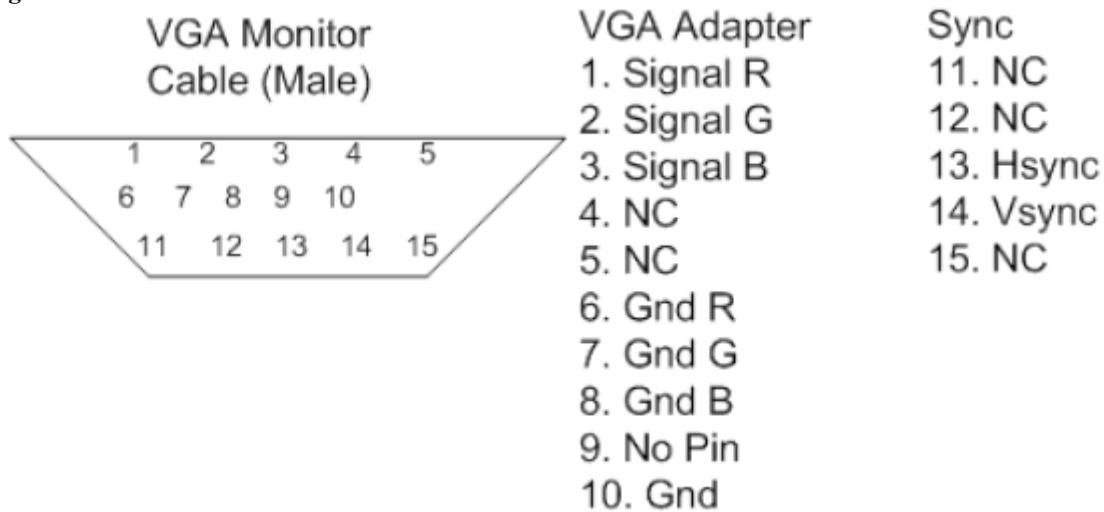
Figure 1 - System Block Diagram



New Hardware:

In this project, the team employed a VGA monitor to display the game. A VGA monitor uses a 15 pin connection. Four of these pins are ground pins, one for master ground, and three for color signal ground. Three pins are the analog color pins, one for red, another green, and one for blue. In addition to these pins, two pins are used for the horizontal and vertical sync signals. The rest of the fifteen pins are unused. Figure 2 shows a diagram of the pins on a male connector, and their usage. For this project, the three color pins and the two sync pins were directly connected to the digital output from the FPGA.

Figure 2 - VGA Pins



A 640x480 pixel VGA display, used for this project, takes pixel data at a rate of 25.175 MHz. Using the Digital Clock Manager (DCM) built into the FPGA, and a 40 MHz

clock signal, the 5/8 scaling produces a signal at 25 MHz. This is within the error permitted by a VGA monitor and so can be used to generate the signal.

HSync and VSync are two signals which control the rate of the monitor, and inform it when to begin a new row of pixels and a new screen. These signals need to occur at the frequencies of 31.47 KHz for HSync and 59.94 Hz for VSync. HSync should have a pulse length of 64 25.175 MHz clock periods, or 3.813 us, and VSync should have a pulse length equal to the time required to draw two rows of pixels, or 63.555 us. Timing information is given in Table 1.

Table 1 - VGA Timing Information

	25.175 MHz clock (VGA Standard)
[clock cycles] time	
Hsync period	[800] 31.778 us
H front porch	[8] 317.775 ns
Hsync pulse length	[96] 3.813 us
H back porch	[40] 1.589 us
H Border	[8] 317.775 ns
H active video	[640] 25.422 us
[scans] time	
Vsync period	[525] 16.683 ms
V front porch	[2] 63.555 us
Vsync pulse length	[2] 63.555 us
V back porch	[25] 794.439 us
V Border	[8] 254.220 us
V active video	[480] 15.253 ms
Vsync, Hsync polarity	-, -
H frequency	31.47 KHz
V frequency	59.94 Hz

More information on the use of a VGA monitor controlled by the FPGA can be found in the MicroToys VGA monitor documentation [1].

Schematics:

We connected two potentiometers as controllers through phone cord, with their outside pins connected to 3.3 volts and ground and the middle pins connected to Port A pins 0 and 1 on the PIC. The VGA connector is wired with pins 1, 2, and 3 connected to P1, P2, and P4 on the FPGA, pins 13 and 14 (HSync and VSync respectively) connected to pins P13 and P14 on the FPGA, and pins 6, 7, 8, and 10 tied to ground. No other connections or components were necessary.

Microcontroller Design:

Our project handles all of the game logic in the PIC microcontroller. The PIC takes the values of the two analog controllers as inputs on analog pins A₀ and A₁. It uses Port C and Port D as a sixteen bit parallel bus, ten bits of which are data and six of which are a control code (one bit of which is high to specify valid data). The PIC keeps track of the locations of the paddle, the location and velocity of the ball (represented as a fixed point number with four fractional bits to allow smooth ball movement), and the score. These values are transmitted to the FPGA for display at the appropriate times.

After some initialization, the PIC enters its main loop where it alternately reads the analog values in from each of the controllers. These values are then used to determine what the current location of the paddles should be. All other work is done at the interrupts.

The interrupt is where most of the game logic is called from. Although Timer0 is used to generate the interrupts, the game logic takes long enough that the timer will trigger almost immediately after the logic of the previous interrupt is complete. At each interrupt, the game will send new locations of the paddles and ball to the FPGA, move the ball, and check for collisions between the ball, sides, and paddles. If the ball collides with a vertical boundary or with the paddle, it is reflected and continues its motion. When reflecting from the paddle, the ball's velocity is increased slightly. The velocity of the paddle affects the change in the ball's vertical velocity, allowing the player some control over this bounce. If a player misses the ball and it hits the edge of the screen the other player then scores. The scoring routine updates the scores and pauses briefly (or slightly longer if one player has one and the game is resetting) before beginning the next serve.

FPGA Design:

We use the FPGA to generate the signals to drive a VGA monitor. Our code to do this is based off of the MicroToys VGA project [1], but with several improvements and corrected bugs. Our FPGA takes a 40 MHz clock and a sixteen bit parallel bus (ten bits of which are data and six of which are a control code (one bit of which is high to specify valid data)) as input, and generates the horizontal sync; vertical sync; and red, blue and green signals used to control a VGA monitor.

Our FPGA design takes the 40 MHz clock available on the HarrisBoard and uses the Digital Clock Manager to turn this into a 25 MHz clock that is close enough to the 25.175 MHz clock that the VGA monitor expects. The CoreGen & Architecture Wizard's Single DCM option will allow you to create a .xaw file from which verilog specifying the DCM can be generated. Checking CLKDV and CLKFX and specifying values of 8 and 5 for them respectively will multiply the clock by 5/8. This will take the CLKIN value of 40 Mhz and outputs a 25 Mhz signal on CLK0. Clicking on the xaw file will give you the option to generate an instance, and this Verilog module is instantiated in order to give access to the managed clock signal.

We also have a module GenSyncsVGA which takes this clock and generates the horizontal and vertical sync signals used by the VGA monitor using several counters. Note that the values used to generate this timing differ slightly from those found in the MicroToys tutorial, as theirs did not quite agree with the VGA standard [2] and produced a distorted output. This module also generates and output specifying if a valid pixel can currently be displayed.

Our module GenSignalVGA takes the timing signals from GenSyncsVGA and uses them to generate the output signals to display. It uses the module RowColCounter to keep track of which row and column are being displayed. The module DataRead is used to coordinate communication with the PIC and to remember the locations of the shapes to be drawn. Several instances of InCircle and InBox are also used to perform bounds checking on these shapes. These modules output can be or-ed together to determine if the current pixel is red, green, blue, or some combination, thus producing the output signal.

Results:

Our project resulted in a working implementation of the game Pong. The game displays the paddles on either side of the screen, with a circular ball bouncing between them. The play area on the screen is separated from the rest of the display by lines at the top and bottom. Above the play area, the score is displayed as two bars which fill toward the center of the screen when one player scores a point.

The hardest step of implementation was creating a round ball. We originally attempted to do this only using the ten bit multipliers on the FPGA to determine if the current bit was within the radius of the ball. This resulted in a large amount of overflow, which produced a pattern of shapes on the screen which made it difficult to determine the actual position of the ball. In order to solve this problem, we had to eliminate the ball signal outside of a box slightly larger than the ball itself.

Another difficulty we encountered was using the provided Microtoys code for displaying a box on a VGA monitor. In this code, we found a large number of errors where the previous team used numbers which were off by one. Also, since all of the numbers were originally coded in binary, which is hard for many people to easily read, some of them were incorrect by a factor of two. This resulted in a vertical compression of the screen which we corrected by coding the correct values, and we did so in decimal to make errors easily noticeable..

The game still has a few display errors that we were unable to fix. To the right of some of the boxes we draw, and at the very right of the screen, dim “ghost” lines appear on the monitor. Looking through the code, the team was unable to find an explanation for why these lines appear.

The team was also unable to create a container for the game, due to a lack of time. With more time, the game would be contained in a box with two RJ-11 ports for the

controllers, and a VGA port for the monitor. The controllers would also be contained in smaller boxes to make them easier to use.

References

[1] MicroToys VGA project,

<http://www4.hmc.edu:8001/Engineering/microtoys/VGA/MicroToys%20VGA.pdf>

[2] VGA timing information, http://www.epanorama.net/documents/pc/vga_timing.html

Parts List

We used several parts from RadioShack in addition to those available in the MicroPs lab.

Knobs	\$1.99
Female DB15 connector	\$1.89
Phone cord	\$2.97
RJ25 connectors	\$6.99
Crimping tool	\$10.49

Appendix 1: C Code

pong.c:

```
// Jonathan Beall and Austin Katzin, Fall 2006
// Game logic for a game of Pong.

#include <p18f452.h>
#include <timers.h>

// Constants.
#define DATA_HI_MASK    0x03
#define DATA_LO_MASK    0xff
#define DATA_LO_SIZE    8

#define CODE_NONE        0x00
#define CODE_PADDLE1X    0xe0
#define CODE_PADDLE2X    0xe4
#define CODE_PADDLE1Y    0xc0
#define CODE_PADDLE2Y    0xc4
#define CODE_BALLX        0xa0
#define CODE_BALLY        0xa4
#define CODE_PADDLE_WIDTH 0xc8
#define CODE_PADDLE_HEIGHT 0xcc
#define CODE_BALL_WIDTH  0xa8
#define CODE_BALL_HEIGHT 0xac
#define CODE_SCORE        0x90

#define PADDLE1_CHANNEL  0
#define PADDLE2_CHANNEL  1
#define AD_BITS           10

#define SCREEN_WIDTH      640
#define SCREEN_HEIGHT     480
#define TABLE_WIDTH      640
#define TABLE_HEIGHT     400
#define TABLE_X0        0
#define TABLE_Y0        78

#define PADDLE_WIDTH      5
#define PADDLE_HEIGHT     100
#define BALL_WIDTH        16
#define BALL_HEIGHT       16

#define PLAYER_1          0
#define PLAYER_2          1
#define PLAYER_RESET      2

#define UPDATE_DELAY      60

// Control how sensitive the ball-y-velocity change
// ..is to paddle velocity.
```

```

#define DELTA_DELAY      2
#define DELTA_SHIFT     0

// Global Variables
int paddle1;
int paddle2;
int oldPaddle1[DELTA_DELAY];
int oldPaddle2[DELTA_DELAY];
int ballx;           // This is floating point w/ 4 fractional bits.
int bally;          // This is floating point w/ 4 fractional bits.
int vx;             // This is floating point w/ 4 fractional bits.
int vy;             // This is floating point w/ 4 fractional bits.
unsigned char score1;
unsigned char score2;

// Prototypes
void isr(void);
unsigned int analogRead(unsigned char);
void send(unsigned char code, unsigned int data);
void score(unsigned char player);

#pragma code high_vector = 0x8
void high_interrupt(void)
{
    _asm
        GOTO isr
    _endasm
}

#pragma code
void main(void)
{
    unsigned short long p;
    score1 = score2 = 0;
    ballx = (TABLE_X0 + ((TABLE_WIDTH + BALL_WIDTH) >> 1)) << 4;
    bally = (TABLE_Y0 + ((TABLE_HEIGHT + BALL_HEIGHT) >> 1)) << 4;
    vx = 1 << 4;
    vy = 1 << 4;

    TRISD = 0;
    TRISC = 0;

    // FPGA is slow to start up, so we'll give it a bit before starting game.
    p = 0;           // Opportunistic reuse of unsigned short long p
    while (p < ((unsigned short long) 1) << 21) // Delay to let FPGA reset.
        p += 1;

    // Reset, and send the FPGA all the info.
    score(PLAYER_RESET);

    T0CON = 0x87;           // Timer0 on, 16 bit, CLK0, x, PSA, by 256
    INTCON = 0xa0;        // Enable Interrupts, Int on TMR 0 overflow.

```



```

while (1)
{
    // Alternately read controller values and update paddle locations.
    // Map 10 bit A/D value into whatever range of motion the paddles have.
    p = analogRead(PADDLE1_CHANNEL);
    p *= (TABLE_HEIGHT - PADDLE_HEIGHT);
    p >>= AD_BITS;
    p += TABLE_Y0;
    paddle1 = p;

    p = analogRead(PADDLE2_CHANNEL);
    p *= (TABLE_HEIGHT - PADDLE_HEIGHT);
    p >>= AD_BITS;
    p += TABLE_Y0;
    paddle2 = p;
}
}

#pragma interrupt isr
void isr(void)
{
    int ballxReal, ballyReal, delta1, delta2;
    unsigned char i;

    if (INTCONbits.TMR0IF)
    {
        // PIC isn't actually fast enough for timer value to matter.
        TMR0H = (0xffff - UPDATE_DELAY) >> 8;
        TMR0L = (0xffff - UPDATE_DELAY) && 0xff;
        INTCONbits.TMR0IF = 0; // Clear interrupt flag.

        // Keep track of prev. paddle position for velocity measurements.
        for (i = 0; i < DELTA_DELAY-1; i++)
        {
            oldPaddle1[i+1] = oldPaddle1[i];
            oldPaddle2[i+1] = oldPaddle2[i];
        }
        oldPaddle1[0] = paddle1;
        oldPaddle2[0] = paddle2;
        delta1 = paddle1 - oldPaddle1[DELTA_DELAY-1];
        delta2 = paddle2 - oldPaddle2[DELTA_DELAY-1];

        // Update the shapes on screen.
        send(CODE_PADDLE1Y, paddle1);
        send(CODE_PADDLE2Y, paddle2);
        send(CODE_BALLX, ballx >> 4);
        send(CODE_BALLY, bally >> 4);

        // Move the ball
        ballx += vx;
        bally += vy;
        ballxReal = ballx >> 4; // Ball pos. and vel. are fixed point
    }
}

```

```

    ballyReal = bally >> 4;    // ..fractions, so we get the truncated int.
    if (ballx < 0)             // Compensate for the logical shift.
        ballxReal |= 0xf000;
    if (bally < 0)             // Compensate for the logical shift.
        ballyReal |= 0xf000;

// Handle ball collision detection.
if (ballxReal <= (TABLE_X0 + PADDLE_WIDTH) && vx < 0)
{
    if (ballyReal + BALL_HEIGHT >= paddle1 &&
        ballyReal <= paddle1 + PADDLE_HEIGHT)
    {
        vx = -vx;
        vx += 0x4;
        vy += 0x4 * (delta1 >> DELTA_SHIFT);
    }
    else
        score(PLAYER_2);
}
if (ballxReal >= (TABLE_X0 + TABLE_WIDTH - PADDLE_WIDTH - BALL_WIDTH)
    && vx > 0)
{
    if (ballyReal + BALL_HEIGHT >= paddle2 &&
        ballyReal <= paddle2 + PADDLE_HEIGHT)
    {
        vx = -vx;
        vx -= 0x4;
        vy += 0x4 * (delta2 >> DELTA_SHIFT);
    }
    else
        score(PLAYER_1);
}
if (ballyReal <= TABLE_Y0 && vy < 0)
    vy = -vy;
if (ballyReal >= (TABLE_Y0 + TABLE_HEIGHT - BALL_HEIGHT) && vy > 0)
    vy = -vy;
}
}

// Setup A/D converter and perform an analog measurement on specified pin.
unsigned int analogRead(unsigned char pin)
{
    unsigned char i = 0xff;
    TRISA = 0xff;
//ADCON1 = 0x40;           // ADFM: left justified, FOSC/64, All A analog
ADCON1 = 0xc0;           // ADFM: right justified, FOSC/64, All A analog
ADCON0 = 0x81;           // AD clock: FOSC/64, AN1 channel, !GO, ADON
ADCON0 |= (pin & 7) << 3; // Select which analogue channel
while (i > 0)            // A busy wait delay loop to allow the charge
    i--;                  // ..holding capacitor to charge.
ADCON0bits.GO = 1;       // Initiate the conversion.
while (ADCON0bits.GO);   // Wait till conversion is complete.
}

```

```

    return ADRES;
}

// Send the data (with appropriate control code) to the FPGA.
void send(unsigned char code, unsigned int data)
{
    // Set code to NONE first to avoid screwups when data's read
    // ..halfway through an update, then set lower byte of data, then
    // ..set upper byte, including control code.
    PORTD = CODE_NONE;
    PORTC = data & DATA_LO_MASK;
    PORTD = code | ((data >> DATA_LO_SIZE) & DATA_HI_MASK);
}

// Update the score depending on who scored a point.  Reset at 5 points.
// Resend size data at each scored point to make sure FPGA isn't out of sync.
void score(unsigned char player)
{
    unsigned short long busyWait;
    ballx = (TABLE_X0 + ((TABLE_WIDTH + BALL_WIDTH) >> 1)) << 4;
    bally = (TABLE_Y0 + ((TABLE_HEIGHT + BALL_HEIGHT) >> 1)) << 4;
    vx = 1 << 4;
    vy = 1 << 4;
    if (player == PLAYER_1)
    {
        ++score1;
        vx = -vx;          // Return serve to other player.
    }
    else if (player == PLAYER_2)
        ++score2;
    else if (player == PLAYER_RESET)
        score1 = score2 = 0;

    send(CODE_PADDLE1X, TABLE_X0);
    send(CODE_PADDLE2X, TABLE_X0 + TABLE_WIDTH - PADDLE_WIDTH);
    send(CODE_PADDLE_WIDTH, PADDLE_WIDTH);
    send(CODE_PADDLE_HEIGHT, PADDLE_HEIGHT);
    send(CODE_BALL_WIDTH, BALL_WIDTH);
    send(CODE_BALL_HEIGHT, BALL_HEIGHT);

    send(CODE_SCORE, (score2 << 3) | score1);
    // If someone scored, wait a sec so they can see.
    for(busyWait = 0; busyWait < ((unsigned short long) 1)<<19; ++busyWait);
    if (score1 >= 5 || score2 >= 5)
    {
        // If someone won, wait a while so they can gloat.
        for(busyWait = 0; busyWait < ((unsigned short long) 1)<<21; ++busyWait);
        score1 = score2 = 0;
        send(CODE_SCORE, (score2 << 3) | score1);
    }
}
}

```

Appendix 2: Verilog Code

Toplevel.v

```
// Jonathan Beall and Austin Katzin, Fall 2006
// Based on code by Michael Cope and Philip Johnson 1999
// Modified by Dan Chan, Nate Pinckney and Dan Rinzler Spring 2005
// Further modified by Jonathan Beall and Austin Katzin, Fall 2006

module TopLevel(clk,sclk, HSync, VSync, signal, reset, data);

input      clk;           // 40Mhz input clock
input      reset;
output     HSync;        // Horizontal sync signal for monitor
output     VSync;        // Vertical sync signal for monitor
output [2:0] signal;     // RGB (R is output[0]) signal for monitor
input [15:0] data;       // Data input from PIC

wire       sclk;         //25Mhz clock after DCM
wire       clkdv,clkm, clklock; // Unused DCM signals
wire       dataValid;   // Is there valid data being sent?

// Use DCM to create 25Mhz clk
dcm vgaDCM(clk,reset,clkdv,clkm,sclk,clklock);

// Generate monitor timing signals
GenSyncsVGA GenSyncs1(sclk, HSync, VSync, reset, dataValid);

// Generate Signal to monitor
GenSignalVGA GenSignal1(VSync, dataValid, signal, sclk, reset, data);

Endmodule
```

dcm.v

```
/////////////////////////////////////////////////////////////////
/
// Copyright (c) 1995-2006 Xilinx, Inc. All rights reserved.
/////////////////////////////////////////////////////////////////
/
//
//   _____
//  /  ___  \  /
// /___/  \  /   Vendor: Xilinx
// \  \  \  \   Version : 8.2i
// \  \  \  \   Application : xaw2verilog
// /  /  /  /   Filename : dcm.v
// /___/  \  ^   Timestamp : 11/17/2006 16:41:28
// \  \  \  /  \
//  \___\  \___\
//
//Command: xaw2verilog -intstyle H:/MicroPs/pong/dcm.xaw -st dcm.v
//Design Name: dcm
//Device: xc3s400-4tq144
//
// Module dcm
// Generated by Xilinx Architecture Wizard
// Written for synthesis tool: SynplifyPro
`timescale 1ns / 1ps

module dcm(CLKIN_IN,
           RST_IN,
           CLKDV_OUT,
           CLKFX_OUT,
           CLK0_OUT,
           LOCKED_OUT);

    input CLKIN_IN;
    input RST_IN;
    output CLKDV_OUT;
    output CLKFX_OUT;
    output CLK0_OUT;
    output LOCKED_OUT;

    wire CLKDV_BUF;
    wire CLKFB_IN;
    wire CLKFX_BUF;
    wire CLK0_BUF;
    wire GND1;

    assign GND1 = 0;
    assign CLK0_OUT = CLKFB_IN;
    BUFG CLKDV_BUF_INST (.I(CLKDV_BUF),
                        .O(CLKDV_OUT));
    BUFG CLKFX_BUF_INST (.I(CLKFX_BUF),
                       .O(CLKFX_OUT));
    BUFG CLK0_BUF_INST (.I(CLK0_BUF),
```

```

        .0(CLKFB_IN));
// Period Jitter (unit interval) for block DCM_INST = 0.03 UI
// Period Jitter (Peak-to-Peak) for block DCM_INST = 1.23 ns
DCM DCM_INST (.CLKFB(CLKFB_IN),
              .CLKIN(CLKIN_IN),
              .DSSEN(GND1),
              .PSCLK(GND1),
              .PSEN(GND1),
              .PSINCDEC(GND1),
              .RST(RST_IN),
              .CLKDV(CLKDV_BUF),
              .CLKFX(CLKFX_BUF),
              .CLKFX180(),
              .CLK0(CLK0_BUF),
              .CLK2X(),
              .CLK2X180(),
              .CLK90(),
              .CLK180(),
              .CLK270(),
              .LOCKED(LOCKED_OUT),
              .PSDONE(),
              .STATUS())/* synthesis "CLK_FEEDBACK=1X, \
              CLKDV_DIVIDE=2.0, \
              CLKFX_DIVIDE=8, \
              CLKFX_MULTIPLY=5, \
              CLKIN_DIVIDE_BY_2=FALSE, \
              CLKIN_PERIOD=25.0, \
              CLKOUT_PHASE_SHIFT=NONE, \
              DESKEW_ADJUST=SYSTEM_SYNCHRONOUS, \
              DFS_FREQUENCY_MODE=LOW, \
              DLL_FREQUENCY_MODE=LOW, \
              DUTY_CYCLE_CORRECTION=TRUE, \
              FACTORY_JF=C080, \
              PHASE_SHIFT=0, \
              STARTUP_WAIT=FALSE" */;

// synopsys translate_off
defparam DCM_INST.CLK_FEEDBACK = "1X";
defparam DCM_INST.CLKDV_DIVIDE = 2.0;
defparam DCM_INST.CLKFX_DIVIDE = 8;
defparam DCM_INST.CLKFX_MULTIPLY = 5;
defparam DCM_INST.CLKIN_DIVIDE_BY_2 = "FALSE";
defparam DCM_INST.CLKIN_PERIOD = 25.0;
defparam DCM_INST.CLKOUT_PHASE_SHIFT = "NONE";
defparam DCM_INST.DESKEW_ADJUST = "SYSTEM_SYNCHRONOUS";
defparam DCM_INST.DFS_FREQUENCY_MODE = "LOW";
defparam DCM_INST.DLL_FREQUENCY_MODE = "LOW";
defparam DCM_INST.DUTY_CYCLE_CORRECTION = "TRUE";
defparam DCM_INST.FACTORY_JF = 16'hC080;
defparam DCM_INST.PHASE_SHIFT = 0;
defparam DCM_INST.STARTUP_WAIT = "FALSE";
// synopsys translate_on
endmodule

```

vgaSignals.v

```
// Jonathan Beall and Austin Katzin, Fall 2006
// Based on code by Michael Cope and Philip Johnson 1999
// Modified by Dan Chan, Nate Pinckney and Dan Rinzler Spring 2005
// Further modified by Jonathan Beall and Austin Katzin, Fall 2006
//
// This module takes the 25Mhz clock and steps it down to turn on
// HSync and VSync at the correct frequencies. It also determines when
// it is possible to send data for each pixel.

module GenSyncsVGA(clk,HSync,VSyn,reset,dataValid);
input      clk;
input      reset;
output     HSync;
output     VSyn;
output     dataValid; //High when according to HSync and VSyn data is ready
to flow

// 25 Mhz clk period = 40 ns
//Hsync = 31470Hz  Vsync = 59.94Hz
reg  [9:0] slowdownforHsync;
reg  [9:0] slowdownforVsync;
reg       HSync;
reg       HData; // High when according to HSync data is ready to flow
reg       VData; // High when according to VSyn data is ready to flow
reg       VSyn;

always @ (posedge clk)
begin
    slowdownforHsync = slowdownforHsync + 1;
    if((slowdownforHsync == 10'd800) || (reset == 1))
        slowdownforHsync = 0;

    if((slowdownforHsync >= 10'd8) && (slowdownforHsync < 10'd104))
HSync = 0;
    else
        HSync = 1;

    if((slowdownforHsync >= 10'd152) && (slowdownforHsync < 10'd792))
        HData = 1;
    else
        HData = 0;
end

//this always block determines when VSyn should be driven low, indicating the
//start of a new screen
always @ (negedge HSync)
begin
    slowdownforVsync = slowdownforVsync + 1;
    if ((slowdownforVsync == 10'd525) || (reset == 1))
        slowdownforVsync = 0;

    if((slowdownforVsync >= 10'd2) && (slowdownforVsync < 10'd4))
```

```
        VSync = 0;
    else
        VSync = 1;

        if((slowdownforVsync >= 10'd37) && (slowdownforVsync < 10'd517))
            VData = 1;
        else
            VData = 0;
    end

    assign dataValid = HData && VData;

endmodule
```


outputSignals.v

```
// Jonathan Beall and Austin Katzin, Fall 2006
// Based on code by Michael Cope and Philip Johnson 1999
// Modified by Dan Chan, Nate Pinckney and Dan Rinzler Spring 2005
// Further modified by Jonathan Beall and Austin Katzin, Fall 2006
//
// These modules take the monitor sync signals and use them to generate
// a signal to display many boxes and circles.

// Keep track of where on the screen we are drawing to.
module RowColCounter(VSync, dataValid, col, row, clk);

    input        VSync;
    input        dataValid;
    output reg   [9:0] col;    // Horizontal coordinate
    output reg   [9:0] row;    // Vertical coordinate
    input        clk;

    reg [9:0] temp;

    // This always block counts column values from 0 to 640
    always @ (posedge clk)
    begin
        if (dataValid)
            col <= col + 1;
        else
            col <= 0;
    end

    // temp also counts col values.
    // This lets us know when we're at the next row.
    always @ (posedge clk)
    begin
        if(!VSync)    // new screen
            begin
                temp <= 0;
                row <= 0;
            end
        else
            if (dataValid)
                temp <= temp + 1;
            if (temp == 10'd640)
                begin
                    row <= row + 1;
                    temp <= 0;
                end
            end
    end
endmodule

// Given coordinates, the upper left corner of the box, and the size,
// are the coordinates in the box?
```

```

module InBox(x, y, x1, y1, width, height, in);
    input [9:0] x, y;
    input [9:0] x1, y1;
    input [9:0] width, height;
    output in;

    wire [9:0] x2, y2;

    assign x2 = x1 + width;
    assign y2 = y1 + height;

    assign in = x >= x1 && x < x2 && y >= y1 && y < y2;
endmodule

// Given coordinates, the upper left corner of a circle's bounding box,
// and the diameter, are the coordinates in the circle?
module InCircle(x, y, x1, y1, d, in);
    input [9:0] x, y;
    input [9:0] x1, y1;
    input [9:0] d;
    output in;

    wire boxIn;
    wire [9:0] xa, ya, r, d1;

    wire [17:0] wx, wy, wxa, wya, wr;

    assign xa = x1 + r;
    assign ya = y1 + r;
    assign r = d >> 1;
    assign d1 = d + 1;

    // We've got 18 bit multipliers on the FPGA.
    // Use it all to avoid overflow.
    assign wx = {8'b0, x};
    assign wy = {8'b0, y};
    assign wxa = {8'b0, xa};
    assign wya = {8'b0, ya};
    assign wr = {8'b0, r};

    // Gotta check if the coordinates are also in the circles bounding box.
    // Otherwise we get overflow issues.
    InBox boundsCheck(x, y, x1, y1, d1, d1, boxIn);

    assign in = (((wx-wxa)*(wx-wxa)) + ((wy-wya)*(wy-wya)) <= wr * wr) & boxIn;
endmodule

// This module reads in data from the PIC every clock cycle.
module DataRead(clk, reset, dataIn, paddle1x, paddle2x, paddle1y, paddle2y,
    ballx, bally, paddleWidth, paddleHeight, ballWidth, ballHeight, score);
    input          clk;
    input          reset;
    input [15:0]   dataIn;

```

```

output reg [9:0]    paddle1x, paddle2x;
output reg [9:0]    paddle1y, paddle2y;
output reg [9:0]    ballx, bally;
output reg [9:0]    paddleWidth, paddleHeight;
output reg [9:0]    ballWidth, ballHeight;
output reg [9:0]    score;

```

```

parameter NONE      = 6'b000000;
parameter PADDLE1Y  = 6'b110000;
parameter PADDLE2Y  = 6'b110001;
parameter PADDLE1X  = 6'b111000;
parameter PADDLE2X  = 6'b111001;
parameter BALLX     = 6'b101000;
parameter BALLY     = 6'b101001;
parameter PADDLE_WIDTH = 6'b110010;
parameter PADDLE_HEIGHT = 6'b110011;
parameter BALL_WIDTH  = 6'b101010;
parameter BALL_HEIGHT = 6'b101011;
parameter SCORE      = 6'b100100;

```

```

always @ (posedge clk, posedge reset)
begin
    if (reset)
    begin
        paddle1x <= 10'd0;
        paddle2x <= 10'd635;
        paddle1y <= 10'd0;
        paddle2y <= 10'd0;
        paddleWidth <= 10'd5;
        paddleHeight <= 10'd100;
        ballWidth <= 10'd5;
        ballHeight <= 10'd10;
        score <= 10'd0;
        ballx <= 10'd320;
        bally <= 10'd240;
    end
    else
    begin
        case (dataIn[15:10])
            PADDLE1X:
                paddle1x <= dataIn[9:0];
            PADDLE2X:
                paddle2x <= dataIn[9:0];
            PADDLE1Y:
                paddle1y <= dataIn[9:0];
            PADDLE2Y:
                paddle2y <= dataIn[9:0];
            BALLX:
                ballx <= dataIn[9:0];
            BALLY:
                bally <= dataIn[9:0];
            PADDLE_WIDTH:
                paddleWidth <= dataIn[9:0];

```

```

        PADDLE_HEIGHT:
            paddleHeight <= dataIn[9:0];
        BALL_WIDTH:
            ballWidth <= dataIn[9:0];
        BALL_HEIGHT:
            ballHeight <= dataIn[9:0];
        SCORE:
            score <= dataIn[9:0];
        default: ;
    endcase
end
end
endmodule

// Use a RowColCounter to keep track of rows and columns.
// Read in positions of the shapes from the PIC.
// Instantiate some InBox and InCircles to represent shapes on the screen
// Or together the outputs of the InBox and InCircles to get output signals.
module GenSignalVGA(VSync, dataValid, signal, clk, reset, data);
    input          VSync;
    input          dataValid;
    output [2:0]   signal;
    input          clk;
    input          reset;
    input [15:0]   data;

    wire [9:0] col; // Horizontal coordinate
    wire [9:0] row; // Vertical coordinate
    wire inBall, inPaddle1, inPaddle2, inTopLine, inBottomLine,
        inCenterLine, inScore1, inScore2;
    wire [9:0] ballx, bally, ballWidth, ballHeight, paddle1y,
        paddle2y, paddle1x, paddle2x, paddleWidth,
        paddleHeight, score;
    wire [9:0] score1x, score1width, score2x, score2width;

    // Keep track of current coordinates.
    RowColCounter rcCount(VSync, dataValid, col, row, clk);
    // Keep track of shape locations.
    DataRead fromPic(clk, reset, data, paddle1x, paddle2x,
        paddle1y, paddle2y, ballx, bally,
        paddleWidth, paddleHeight, ballWidth, ballHeight,
        score);

    // All of the shape boundary checking.
    InCircle ball(col, row, ballx, bally, ballWidth, inBall);
    InBox paddle1(col, row, paddle1x, paddle1y,
        paddleWidth, paddleHeight, inPaddle1);
    InBox paddle2(col, row, paddle2x, paddle2y,
        paddleWidth, paddleHeight, inPaddle2);
    InBox topline(col, row, 10'd0, 10'd76, 10'd640, 10'd2, inTopLine);
    InBox bottomline(col, row, 10'd0, 10'd478, 10'd640, 10'd2, inBottomLine);
    InBox centerline(col, row, 10'd320, 10'd0, 10'd2, 10'd76, inCenterLine);

```

```

// Calculate size of score boxes based on score data
assign score1x = 10'd0;
assign score1width = {1'b0, score[2:0], 6'b0};
assign score2x = 10'd640 - score2width;
assign score2width = {1'b0, score[5:3], 6'b0};
InBox score1(col, row, score1x, 10'd20, score1width, 10'd40, inScore1);
InBox score2(col, row, score2x, 10'd20, score2width, 10'd40, inScore2);

// Assign signal value based on shape bounds check results.
// Red
assign signal[0] = (inPaddle1 || inPaddle2
    || inTopLine || inBottomLine || inCenterLine)
    && dataValid;
// Green
assign signal[1] = (inBall || inPaddle1 || inPaddle2
    || inTopLine || inBottomLine || inCenterLine)
    && dataValid;
// Blue
assign signal[2] = (inPaddle1 || inPaddle2
    || inTopLine || inBottomLine || inCenterLine
    || inScore1 || inScore2) && dataValid;

endmodule

```