Space Invaders Final Report

Kevin Hsu
Andrew Pozo
December 10, 2009

ABSTRACT

A simplified version of *Space Invaders* has been implemented using a PIC18F4520 microcontroller and a SPARTAN-3 FPGA. The microcontroller is used to receive user control inputs and calculate the state of the game: invader position/status, tank position, bullet position, score, and win or lose. The FPGA is implemented as a graphics engine, to display the game on a VGA monitor. A single level of the game with ten space invaders on the screen was implemented in the final version.

INTRODUCTION

This report describes the implementation of *Space Invaders LITE*, based off of the game *Space Invaders*, originally developed by Tomohiro Nishikado in 1978. A player uses a Nintendo Entertainment System (NES) controller to control the tank's movement and shooting. The controller is powered by a separate 3.3V power source, to provide the proper logic levels to input ports configured for digital I/O on the Harris board.

The PIC takes the inputs from the controller and updates the tank position on the screen, while moving the space invaders down the screen on a set trajectory. If the user decides to shoot, the PIC generates a single bullet based off the current position of the tank. The user is only allowed to shoot again once a space invader is destroyed or the bullet has gone off screen. The game ends when either all the space invaders have been eliminated or have reached the tank. This data, collectively called the game state, is sent in a 16-bit sequence to the FPGA via Ports B, C, and D.

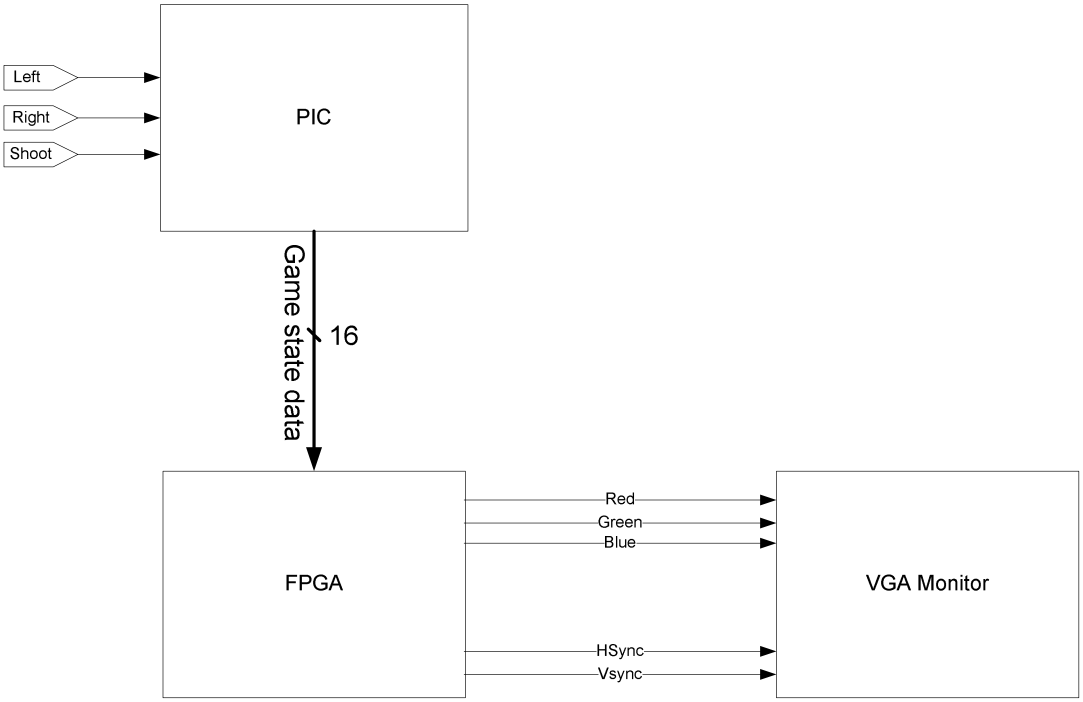


Figure 1. System block diagram

The FPGA decodes data sent from the PIC, determines which pixels are on and drives the VGA monitor. The FPGA outputs the three color bits as well as HSync and VSync for the monitor.

NEW HARDWARE

VGA Monitor

The game is displayed using a VGA monitor. A VGA monitor makes images with an electron gun that scans across the screen one row of pixels at a time until it reaches the bottom of the screen and resets back at the top to paint another screen. The scanning is governed by a clock running at $25.175\ MHz$ and by two synchronization signals:

- **Hsync**: This horizontal synchronizing signal resets the monitor's electron gun to begin scanning a new line by pulsing for $3.77\ \mu s$ at a frequency of $31.47\ KHz$. $Hsync$ has a negative polarity.
- **Vsync**: This vertical synchronization signal resets the monitors electron gun to begin scanning a new screen by pulsing for $63.555\ \mu s$ at a frequency of $59.94\ Hz$. $Vsync$ has a negative polarity.

The color of each pixel is determined by the voltage applied to three analog color signals: red, green and blue. The color signals must be asserted with proper timing to paint all desired pixels as the electron gun scans over the screen. The timing of the color signals was achieved by counting rows and columns within the FPGA, so that the pixel the electron gun is painting is known at all times.

More information on controlling a VGA monitor with a FPGA and schematics of the VGA pinouts can be found in the MicroToys VGA monitor document [1].

NES Controller

A NES controller is used to control the movement and shooting of the tank. A standard NES controller uses a small integrated circuit to serially transmit the state of all the buttons when cued. For the *Space Invaders* game, the NES controller is used just as housing for the move and shoot buttons. The IC was removed and wires were directly soldered to one end of the desired buttons through the holes left after removing the IC, as well as to the common power and ground planes of the controller's PCB. A simplified controller schematic is shown as part of the bread board schematic in *Appendix A*.

MICROPROCESSOR DESIGN

The PIC controls the game state of the space invaders and player. The variables associated with the game states are saved as global variables. Positions of the space invaders, tank and bullet are stored on the PIC as global integers, and the status of the space invaders are stored on the PIC as a global array of 1's and 0's where a 1 represents an alive space invader and a 0 represents a dead space invader.

The PIC uses the TIMER0 overflow interrupt to update the game state on the FPGA. On the interrupt, the PIC transfers a 16 bit sequence to the FPGA, containing the game state.

6 Code bits	10 Data bits

The top 6 bits of the sequence are code bits, which the FPGA uses to identify the type of data it is receiving. The lower 10 bits store relevant data, like the x-y position of the space invaders or tank, or whether a space invader is alive or dead. This 16-bit coded sequence is sent to the FPGA through PORTB[0:2], PORTC[3:7], and PORTD[0:7] (listed in MSB to LSB order). Inputs are received on the three LSBs of PORTC.

The following are the functions used to implement *Space Invaders LITE*:

Function	Description
`void isr(void)`	TIMER0 interrupt handler. On TIMER0 overflow, reset values of TIMER0, and sends game state data to the FPGA using `send_int` and `send_array`. Checks for the four different run screens possible: go, game, win, or lose. Depending on the current game state, the function will send a control signal to the FPGA choosing which screen to display. It also updates the game states on the PIC and looks for inputs to the PIC for left movement, right movement, or a shot.
`void send_int(unsigned char code, unsigned int data)`	Takes in "code" and "data" and sends it to the FPGA. Data sent through this function are: • Invader x-y position • Bullet x-y position • Tank x-y position
`void send_array(unsigned char code, unsigned int *data)`	Takes in the "code" for an invader array (an array of 0's and 1's indicating the status of a specific invader on the screen) and "data" array to the FPGA.
`void invader_traj(void)`	Gives the path the space invaders will traverse during the course of the game.
`void move_tank(void)`	Reads in inputs from the two LSB of PORTC and updates the horizontal position of the tank on the PIC. Error checks for the user input so that the tank does not run off the screen.
`void shoot(void)`	Gives the initial position and velocity of the bullet, once a shot is detected through an input port.
`int collision_detect(void)`	Detects collisions between the bullet and a space invader. Hit boxes for the invaders are a 22x16 pixel rectangle enclosing the entire space invader.
`int checkwin(void)`	Checks if the invader array is empty. If the array it is, returns 1 for win, otherwise returns 0.
`void shot_reset(void)`	Sets the position of the bullet, and the velocity of the bullet to 0.
`void play(void)`	Runs the game loop, and checks for the win and lose conditions of the game. Uses `checkwin` to determine if the user has won.
`void reset(void)`	Resets all parameters necessary to run the game.

FPGA DESIGN

The FPGA is used as the VGA and graphics driver for the game. It takes in the 6 code bits and 10 data bits and translates them into properly timed signals for display on the VGA monitor. The FPGA does this through the use of three main blocks: the digital clock time manager, a generate syncs module and a generate display signals module.

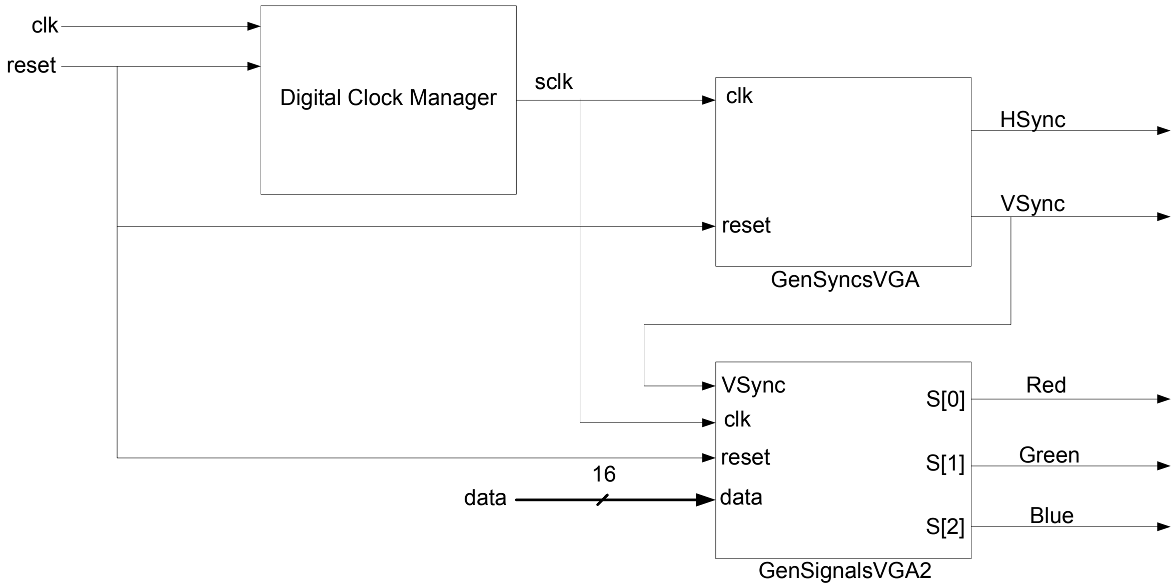


Figure 2. FPGA module block diagram

Digital Clock Manager (DCM)
The DCM is used to create a 25 MHz clock used for timing the VGA monitor's electron gun with the necessary outputs from the FPGA. The DCM was programmed through Xilinx CoreGen and its clock is distributed to all other modules.

GenSyncsVGA
This module generates the *HSync* and *VSync* signals for the VGA monitor. The *GenSyncsVGA* module uses the *DCM*'s slower clock along with counters to keep track of where the VGA monitor's electron gun is and when it needs to be moved to a new row or reset back to the top corner.

GenSignalsVGA2
To get the information of where to paint the tank, bullet or space invader array, this *GenSignalsVGA2* module has a data decoder that take in the 16-bit code sequence received from the PIC, separates the most significant 6 bits and according to their value assigns the least significant 10 bits of data to proper wires that are used by large logic blocks to generate the color signals for the VGA monitor. A row and column counter sub-module keeps track of what pixel the VGA monitor's electron gun is painting at any particular moment. The logic takes this row and column information along with the position data taken from the bottom 10 bits and generates

a signal for the red, green and blue VGA inputs that is asserted only when a particular pixel is supposed to be painted.
To avoid the timing issues generated by asynchronously sending data from the PIC to the FPGA, a *dataReady* bit is sent from the PIC to the FPGA after it has asserted and stabilized the output data. The *dataReady* bit is registered and then tied to the enable of another register that receives the data from the PIC. This set up ensures that data from the PIC is passed to the FPGA VGA logic only on a rising clock edge, when the *dataReady* bit is asserted and consequently when the data being sent is no longer changing.

RESULTS

A reliable *Space Invaders LITE* video game has been implemented. There are start, win and lose game screens, there are ten space invaders slowly working their way down the screen and the user can move the tank and shoot the space invaders with a NES controller.

One problem encountered during the project was a PIC reset that seemed to occur after a set amount of time. It was discovered, with the help of other E155 students during the presentation, that the PIC's watchdog timer was resetting the game. The game is essentially an infinite while loop that sends data when interrupted by TIMER0, because of this infinite loop the watchdog timer was triggered and the PIC was reset. The watchdog timer was disabled by setting its configuration register, WDTCON, to 0x00.

Another problem encountered was the drawing of detailed sprites in an efficient manner. In the current implementation the FPGA uses logic to determine whether or not a pixel at a certain row and column address needs to be painted. This technique is good for drawing simple shapes but the necessary logic increases drastically when coding detailed sprites. An elegant solution for drawing detailed sprites was never developed. Text drawing techniques were explored but proved inadequate. Drawing text would have been implemented by partitioning the monitor into multiple cells, like a chess board. The character each cell was to display would be held in a screen buffer and a ROM library would hold the pixel pattern of all characters. Though defining a space invader as a character in the library and drawing one in desired cells would have allowed for displaying tens of space invaders, it would have restricted their movement to increments of cells, not pixels, as required.

The final major problem encountered was screen glitches. As more information was being sent to the FPGA to be drawn, the images began to glitch randomly across the screen. It was discovered that the information being sent from the PIC was not synchronized with screen drawing. Before synchronizing the data transfer and drawing, it was possible for the FPGA to draw data that was not stable. The synchronization was accomplished as described in the FPGA implementation section.

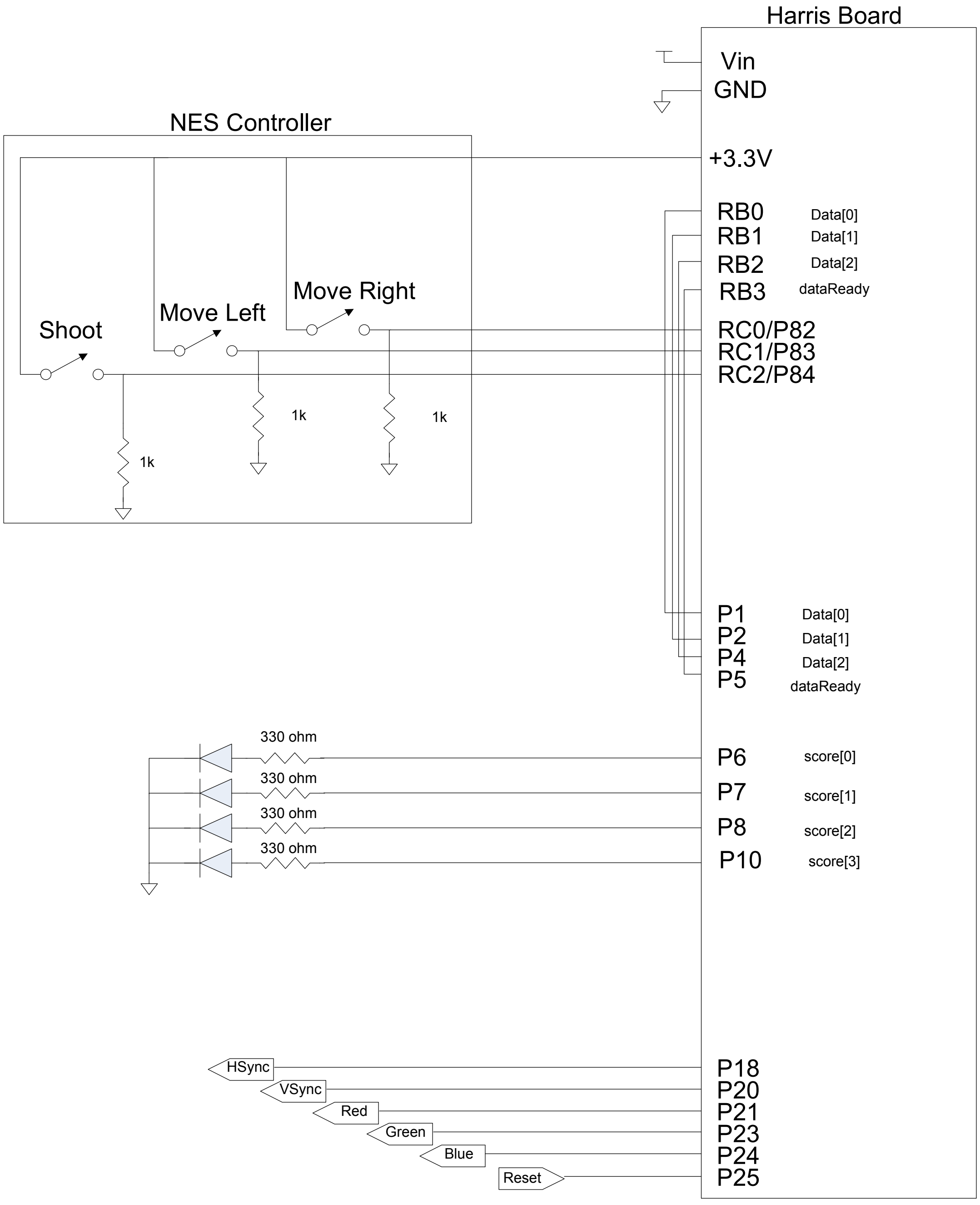
Parts List

NES Controller:	$11.49

Sources

[1] Rinzler, D. "MicroToys Guide: VGA Monitor." Apr. 2005. Web. Nov. 2009.

Appendix A – Bread Board Circuit Schematic



Appendix B – C Code

```c
/*
Programmed By: Kevin Hsu, Andrew Pozo
Contact: khsu@hmc.edu, apozo@hmc.edu

Program Description:
This code runs the full game of Space Invaders (Light).  It holds
all variables for the game state in the PIC and sends the updated
game states to the FPGA to be drawn on screen.
*/

#include <p18F4520.h>
#include <timers.h>

#define DATA_HI_MASK    0x03  //Masks upper 2 bits of data sent
                              //to FPGA.  Data sent to FPGA is 10-bits.
#define DATA_LOW_MASK   0xff  //Masks lower 8 bits of data sent
                              //to FPGA.
#define DATA_LOW_SIZE   8       //bitlength of lower bits sent to FPGA.
#define UPDATE_DELAY    60

//Encodings for data sent to FPGA.
#define CODE_NONE       0x00
#define CODE_INVADER_X  0x04
#define     CODE_INVADER_Y    0x08
#define CODE_TANK_X           0x10
#define CODE_INVADER_R1 0x0C
#define CODE_BULLET_X   0x14
#define CODE_BULLET_Y   0x24
#define CODE_STATUS           0x28
#define CODE_SCORE            0x48

//Initial values and limits
#define INIT_INVADER_X  120
#define INIT_INVADER_Y  50
#define RXLIM                 200
#define LXLIM                 40

// GLOBAL VARIABLES
int invaderx;
int invadery;
int   tankx;
int   tanky;
int bulletx;
int bullety;
int bullet_vel;
int shot;
int game_status;
int invaderR1[10];
int rmove;
int lmove;
int right;
int left;
int invader_move;
int score;
```

```
//Interrupt handler.  On interrupt, updates the gamestate.
void isr(void);

//Allows for digital output to FPGA from pic.  Sends a single
//integer PIC, along with a code describing what the sent data
//is.
//
//Used to send positions values for tank, bullet, space invaders
//and the game status.
void send_int(unsigned char code, unsigned int data);

//Allows for digital output to FPGA from pic.  Sends an array
//of 1-bit integers to the FPGA along with a code describing what
//the sent data is.
//
//Used to send the status of the space invader array to the FPGA.
void send_array(unsigned char code, unsigned int *data);

//Updates the position of the space invaders.  Programmed for
//space invader trajectory.
void invader_traj(void);

//Allows user to move tank through input ports: PORTC[0], PORTC[1].
void move_tank(void);

//Allows user to shoot bullet through input port PORTC[2].  Sets
//"shot" variable to 1, indicating the bullet has been shot.
void shoot(void);

//Checks for collisions between bullet and a space invader.
//Returns 0, if no collision is detected and 1 if a collision is detected.
int  collision_detect(void);

//Resets the "shot" variable to 0, allows for the user to shoot
//again.
void shot_reset(void);

//Checks to see if the invader array is empty.  Returns 0 if invaders
//stille exist.  Returns 1 if no more invaders left.
int  checkwin(void);

//Runs the play game once the user enters the "game" state.  Checks to see
//if user has won, or if invaders have arrived on earth.
void play(void);

//Resets entire game.
void reset(void);

#pragma code high_vector = 0x8

void high_interrupt(void)
{
      _asm
            GOTO isr
      _endasm
}
```

```
#pragma code

void main(void)
{
	unsigned short long p;
	int counter;
	ADCON1 = 0xFF;	//Turn off A/D Converter on PORTB to allow
			//digital I/O through that port
	reset();		//Initialize game

	//Run game, allow for reset when game ends.
	while(1)
	{
		play();
		reset();
	}

}

// interrupt handler, runs this code when TIMER0 Overflows
#pragma interrupt isr
void isr(void)
{
	//reset TIMER0 overflow
	if(INTCONbits.TMR0IF)
	{
		TMR0H = (0xffff - UPDATE_DELAY) >> 8;
		TMR0L = (0xffff - UPDATE_DELAY)&&0xff;
		INTCONbits.TMR0IF = 0;
	}
	PORTBbits.RB3 = 1;
	send_int(CODE_STATUS, game_status);

	//if game status is go, send GO to FPGA to display GO on screen.
	if(game_status == 0)
		game_status = 0;


	//run game if game status is GAME
	else if(game_status == 1)
	{
		// if no bullet is currently moving, allow the PIC to pole user
		// for shots
		if(!shot)
		{
			if(PORTCbits.RC2)
				shoot();
		}
		else // update the bullet position if bullet has been shot
		{
			if(invader_move%2==0)
				bullety -= bullet_vel;
			if(collision_detect())
				shot_reset();
			send_int(CODE_BULLET_X, bulletx);
			send_int(CODE_BULLET_Y, bullety);
```

```
            }

            send_int(CODE_TANK_X, tankx);
            PORTBbits.RB3 = 1;
            send_array(CODE_INVADER_R1, invaderR1);
            PORTBbits.RB3 = 1;
            send_int(CODE_INVADER_X, invaderx);
            PORTBbits.RB3 = 1;
            send_int(CODE_INVADER_Y, invadery);
            PORTBbits.RB3 = 1;
            send_int(CODE_SCORE, score);
            PORTBbits.RB3 = 1;

            if(bullety < 0) //if bullet runs off screen, reset shot
                  shot_reset();

            invader_move++;
            move_tank(); // pole user for tank movement
            if (invader_move == 20)
            {
                  invader_traj();
                  invader_move = 0;
            }
            game_status=1;
      }
      // if game status is WIN, send WIN to FPGA to display WIN on screen
      else if(game_status == 2)
            game_status = 2;


      // if game status is LOSE, send WIN to FPGA to display LOSE on screen
      else
            game_status = 3;


}

void send_int(unsigned char code, unsigned int data)
{

      PORTD = CODE_NONE; //sets PORTD to 0 to clear the port

      //LOWER 8 BITS OF DATA SENT TO FPGA
      PORTBbits.RB0 = data & 0x01;
      PORTBbits.RB2 = (data & 0x02)>>1;  //LSB
      PORTBbits.RB1 = (data & 0x04)>>2;
      PORTCbits.RC3 = (data & 0x08)>>3;
      PORTCbits.RC4 = (data & 0x10)>>4;
      PORTCbits.RC5 = (data & 0x20)>>5;
      PORTCbits.RC6 = (data & 0x40)>>6;
      PORTCbits.RC7 = (data & 0x80)>>7;  //MSB

      //HIGHER 8 BITS OF DATA SENT TO FPGA
      PORTD = code | ((data >> DATA_LOW_SIZE)&DATA_HI_MASK);
}

void send_array(unsigned char code, unsigned int *data)
```

```
{
      unsigned char helper;
      PORTD = CODE_NONE; //sets PORTD to 0 to clear the port

      //LOWER 8 BITS OF DATA SENT TO FPGA
      PORTBbits.RB0 = data[0];
      PORTBbits.RB2 = data[1];
      PORTBbits.RB1 = data[2];
      PORTCbits.RC3 = data[3];
      PORTCbits.RC4 = data[4];
      PORTCbits.RC5 = data[5];
      PORTCbits.RC6 = data[6];
      PORTCbits.RC7 = data[7];

      //HIGHER 8 BITS OF DATA SENT TO FPGA
      if(data[9]&&data[8])
            helper = 0b11;
      else if(!data[9]&&data[8])
            helper = 0b01;
      else if(data[9]&&!data[8])
            helper = 0b10;
      else
            helper = 0b00;

      PORTD = code | helper;
}

void invader_traj(void)
{
      int rightmost;
      int leftmost;
      int counter;
      int trajhelper;

      if(rmove) //invaders moving right
      {
            //allows invaders to move to the right until they reach the
            //the right of the screen
            if(invaderx < (int)RXLIM)//+rightmost*45)
                  invaderx += 5;
            else //sets the movement to the left
            {
                  rmove = 0;
                  lmove = 1;
                  invadery +=16;
            }
      }
      else //invaders moving left
      {
            //allows invaders to move to the left until they reach the
            //the right of the screen
            if(invaderx > (int)LXLIM)//-leftmost*45)
                  invaderx -=5;
            else //sets movement to the right
            {
                  rmove = 1;
                  lmove = 0;
```

```
                invadery +=16;
            }
        }
}

void move_tank(void)
{
        if(PORTCbits.RC0 == 1)  //poles PORTC bit 0 for movement to the left
            if(tankx <=540)    //allows tank movement to left end.
                tankx += 1;
        if(PORTCbits.RC1 == 1)  //poles PORTC bit 1 for movement to the right
            if(tankx >=5)      //allows tank movement to right end.
                tankx -=1;
}

void shot_reset(void)
{
        //resets shot and allows for another shot
        shot = 0;
        bullet_vel = 0;
        bullety = -1;
        bulletx = -1;
}

int collision_detect(void)
{
        int i;

        for(i = 0; i < 10; i++)
        {

        if((bulletx+2>invaderx+i*40)&&(bulletx<invaderx+22+i*40)&&invaderR1[i])
            {
            //checks if bullet is within the bounds of each invader in array.
            //invaders are spaced 40 pixels apart, and have width 22 pixels.
                if((bullety<invadery+16)&&(bullety-
2>invadery)&&(invaderR1[i]!=0))
                {
                    invaderR1[i] = 0;
                    score+=1;
                    return 1;
                    shot_reset();
                }
            }
        }
        return 0;
}

void shoot(void)
{
        //initialize bullet position to tip of tank cannon
        bulletx = tankx+11;
        bullety = tanky;
        bullet_vel = 2; //set bullet velocity
        shot = 1;       //set bullet shot
}
```

```
int checkwin(void)
{
	int counter;
	for(counter=0; counter<10; counter++)
	{
		//user has not won if anything exists in invader array
		if(invaderR1[counter]==1)
			return 0;
	}
	return 1;
}

void play(void)
{
	//allows the game to continue while there are space invaders on the
	//screen, and while the invaders have yet to reach the tank.
	while((invadery < tanky-20)&&(!checkwin()))
	{
	}
	while(!PORTCbits.RC2)
	{
		if(checkwin())
			game_status = 2;
		else
			game_status = 3;
	}
}
void reset(void)
{
	int counter;
	TRISD = 0x00;		//PORTD OUTPUT PORT
	TRISC = 0x07;		//PORTC[2:0] INPUT PORTS
				//PORTC[7:3] OUTPUT PORTS
	TRISB = 0x00;		//PORTB OUTPUT PORTS
	WDTCON = 0x00;		//TURN OFF WATCHDOG TIMER TO ALLOW FOR GAME LOOP
	T0CON = 0x87;		//ENABLE TIMER0, 16 BIT, CLK0, x, PSA, by 256
	INTCON = 0xA0;		//ENABLE INTERRUPTS, INT on TMR0 OVERFLOW
	PORTBbits.RB3 = 0;

	game_status = 0;  // initial GO Screen
	while(!PORTCbits.RC2)
	{
		//Show "GO" screen until user presses "B" button
	}
	game_status = 1; // set screen to GAME

	//Initialize space invader trajectory parameters.  Trajectory
	//begins moving to the right.
	rmove = 1;
	lmove = 0;

	//Initialize invader move counter to 0.  Invader traverses
	//trajectory only when INVADER_MOVE reaches a constant.
	invader_move = 0;
	invaderx = INIT_INVADER_X;  //Initial invader position
	invadery = INIT_INVADER_Y;
```

```
      //Initialize tank positions
      tankx = 320;
      tanky = 460;

      //Initialize bullet positions
      shot = 0; //0 for bullet not yet fired.
      bulletx = -1;
      bullety = -1;


      //Initialize score
      score = 0;
      for(counter = 0; counter <10; counter++)
            invaderR1[counter] = 1;

}
```

Appendix C – Verilog Code

```
/////////////////////////////////////////////////////////////////////////////
// Kevin Hsu and Andrew Pozo, Fall 2009
// 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
// Further modified Jonathan Beall and Austin Katzin, Fall 2006
/////////////////////////////////////////////////////////////////////////////

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

      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
      output [3:0]    score;
      input           dataReady;
      wire            sclk;           //25Mhz clock after DCM
      wire            clkdv,clkfx, clklock; // Unused DCM signals
      wire            dataValid;      // Is there valid data being sent?

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

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

      // Generate Signal to monitor
      GenSignalVGA2 GenSignal(VSync, dataValid, signal, sclk, reset, data,
                         score, dataReady);

endmodule
```

```
/////////////////////////////////////////////////////////////////////////////
// Kevin Hsu and Andrew Pozo, Fall 2009
// 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
// Further modified 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,VSync,reset,dataValid);
input             clk;
input             reset;
output            HSync;
output            VSync;
output            dataValid; //High when according to HSync and VSync 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 VSync data is ready to flow
reg         VSync;

//this always block determines when Hsync should be driven low to start a new
//line
always @ (posedge clk)
    begin
        slowdownforHsync = slowdownforHsync + 1;

        //check if you’ve counted to the end of the screen or if
        //youre resetting
        if((slowdownforHsync == 10'd800) || (reset == 1))
            slowdownforHsync = 0;

        //check if you need to set the Hsync low for next line
        if((slowdownforHsync >= 10'd8)&&(slowdownforHsync < 10'd104))
            HSync = 0;
        else
            HSync = 1;

        //check if youre in a draw able part of the screen, Hdata used later
        if((slowdownforHsync >= 10'd152) && (slowdownforHsync < 10'd792))
            HData = 1;
        else
            HData = 0;
    end
```

```
//this always block determines when VSync should be driven low, indicating
// the start of a new screen
always @ (negedge HSync)
    begin
        slowdownforVsync = slowdownforVsync + 1;

        //see if youre off the screen or if you want to reset
        if ((slowdownforVsync == 10'd525) || (reset == 1))
            slowdownforVsync = 0;

        //see if you need to set Vsync low to start a new screen
        if((slowdownforVsync >= 10'd2) && (slowdownforVsync < 10'd4))
            VSync = 0;
        else
            VSync = 1;

        //see if youre in a draw able area of the screen
        if((slowdownforVsync >= 10'd37) && (slowdownforVsync < 10'd517))
            VData = 1;
        else
            VData = 0;
    end

//a signal that is asserted if the pixel is in a draw able area
assign dataValid = HData && VData;

endmodule
```

```
//////////////////////////////////////////////////////////////////////////////
// Use a RowColCounter to keep track of rows and columns.
// Read in positions of the shapes from the PIC.
// Instantiate space invaders, tank, bullet, GO, WIN, LOSE for purposes of
// the game.
//////////////////////////////////////////////////////////////////////////////

module GenSignalVGA2(VSync, dataValid, signal, clk, reset, data, score,
dataReady);
    input               VSync;
    input               dataValid;
    output  [2:0]       signal;
    input               clk;
    input               reset;
    input   [15:0]      data;
    output  [3:0]       score;
    input               dataReady;
    wire    [9:0]       col;        // Horizontal coordinate
    wire    [9:0]       row;        // Vertical coordinate
    wire                inInvader;
    wire                inTank;
    wire                inBullet;
    wire                inGo;
    wire                inWin;
    wire                inLose;
    wire    [9:0]       invaderX;
    wire    [9:0]       invaderY;
    wire    [9:0]       tankX;
    wire    [9:0]       bulletY;
    wire    [9:0]       bulletX;
    wire    [9:0]       invaderR1;
    wire    [9:0]       status;
    wire                out;
    reg     [15:0]      regData;
    reg                 regDataReady;

    parameter tankY = 10'd460;
    parameter wX    = 10'd320;
    parameter wY    = 10'd240;

    always @ (posedge clk)
       regDataReady <= dataReady;

    always @ (posedge clk)
       if (regDataReady)
            regData <= data;

    // Decodes data sent from PIC
    dataDecoder decoder(clk, reset, regData, invaderX, invaderY, invaderR1,
                        tankX, bulletX, bulletY, status, score);

    // Keep track of current coordinates.
    RowColCounter rcCount(VSync, dataValid, col, row, clk, reset);

    //draw the space invader
    drawInvader invader(col, row, inInvader, invaderX, invaderY,
                          invaderR1);
```

```
       //draw the tank
       drawTank tank(col, row, inTank, tankX, tankY);

       //draw the bullet
       drawBullet bullet(col, row, inBullet, bulletX, bulletY);

       //draw GO
       drawGo go(col, row, inGo, wX, wY);

       //draw WIN
       drawWin win(col,row,inWin,wX,wY);

       //draw LOSE
       drawLose lose(col,row,inLose,wX,wY);

       //registered mux to draw screen depending on game state
       c4 choose(clk, status, inBullet, inInvader, inTank, inGo, inWin,
                 inLose, out);

       assign signal[0] = out;
       assign signal[1] = out;
       assign signal[2] = out;

endmodule

//looks at 6MSB of data taken from the PIC.  Decodes then stores into wires
//for drawing purposes

module dataDecoder(clk, reset, data, invaderX, invaderY, invaderR1, tankX,
                   bulletX, bulletY, status, score);
       input                clk;
       input                reset;
       input          [15:0]data;
       output  reg    [9:0] invaderX;
       output  reg    [9:0] invaderY;
       output  reg    [9:0] invaderR1;
       output  reg    [9:0] tankX;
       output  reg    [9:0] bulletX;
       output  reg    [9:0] bulletY;
       output  reg    [9:0] status;
       output  reg    [3:0] score;

       parameter CODE_INVADER_X  = 6'b000001;
       parameter CODE_INVADER_Y  = 6'b000010;
       parameter CODE_INVADER_R1 = 6'b000011;
       parameter CODE_TANK_X     = 6'b000100;
       parameter CODE_BULLET_X   = 6'b000101;
       parameter CODE_BULLET_Y   = 6'b001001;
       parameter CODE_STATUS     = 6'b001010;
       parameter CODE_SCORE      = 6'b010010;

       always @ (posedge clk, posedge reset)
         begin
            if (reset)
             begin
                  invaderX  <= 10'd0;
```

```
                  invaderY     <= 10'd0;
                  invaderR1    <= 10'b11_1111_1111;
                  tankX        <= 10'd320;
                  bulletX      <= 10'd0;
                  bulletY      <= 10'd0;
                  status       <= 10'd0;
                  score        <= 10'd0;
             end
            else
                  case (data[15:10])
                        CODE_INVADER_X:  invaderX  <= data[9:0];
                        CODE_INVADER_Y:  invaderY  <= data[9:0];
                        CODE_INVADER_R1: invaderR1 <= data[9:0];
                        CODE_TANK_X:     tankX     <= data[9:0];
                        CODE_BULLET_X:   bulletX   <= data[9:0];
                        CODE_BULLET_Y:   bulletY   <= data[9:0];
                        CODE_STATUS:        status <= data[9:0];
                        CODE_SCORE:         score  <= data[9:0];
                        default:;
                  endcase
         end
endmodule

//gets the col and row of the electron gun
module RowColCounter(VSync, dataValid, col, row, clk, reset);

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

    reg [9:0] temp;

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

    // This lets us know when we're at the next row.
    always @ (posedge clk)
      begin
        if (reset)
            begin
              temp<= 0;
              row <= 0;
            end
        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
endmodule

module drawInvader(col, row, in, x, y, R1);
       input [9:0] col;
       input [9:0] row;
       output      in;
       input [9:0] x;
       input [9:0] y;
       input [9:0] R1;

       //combinational logic for space invaders on screen
       assign in = … //combinational logic left out for conserving paper
endmodule

module drawTank(col, row, in, x, y);
       input [9:0] col;
       input [9:0] row;
       output      in;
       input [9:0] x;
       input [9:0] y;

       //combinational logic for tank
       assign in = … //combinational logic left out to save paper
endmodule

module drawBullet(col, row, in, x, y);
       input [9:0] col;
       input [9:0] row;
       output      in;
       input [9:0] x;
       input [9:0] y;

       //combinational logic for bullet
       assign in = ((col>=x)&&(col<=x+1)&&(row==y))||
                   ((col>=x)&&(col<=x+1)&&(row==y+1));
endmodule

module drawGo(col, row, in, x, y);
       input [9:0] col;
       input [9:0] row;
       output      in;
       input [9:0] x;
       input [9:0] y;

       //combinational logic
       assign in = … //combinational logic left out to save paper
```

```
endmodule

module drawWin(col,row,in,x,y);
       input [9:0] col;
       input [9:0] row;
       output            in;
       input [9:0] x;
       input [9:0] y;

       //combinational logic
       assign in = … //combinational logic left out to save paper
endmodule

module drawLose(col, row, in, x, y);
       input [9:0] col;
       input [9:0] row;
       output      in;
       input [9:0] x;
       input [9:0] y;

       //combinational logic
       assign in = … //combinational logic left out to save paper
endmodule

// mux chooses what to output to screen based on status received from PIC
module c4(clk, status, inBullet, inInvader, inTank, inGo, inWin, inLose,
           out);
      input       clk;
      input [9:0] status;
      input       inBullet;
      input       inInvader;
      input       inTank;
      input       inGo;
      input       inWin;
      input       inLose;
      output reg  out;

      parameter GO   = 2'b00;
      parameter GAME = 2'b01;
      parameter WIN  = 2'b10;
      parameter LOSE = 2'b11;
      wire game;

      assign game = (inBullet|inInvader|inTank);

      always @ ( posedge clk )
            case(status[1:0])
                  GO:   out <= inGo;
                  GAME: out <= game;
                  WIN:  out <= inWin;
                  LOSE: out <= inLose;
                  default: out <= game;
            endcase
endmodule
```

```
/////////////////////////////////////////////////////////////////////////////
///
// Copyright (c) 1995-2008 Xilinx, Inc.  All rights reserved.
/////////////////////////////////////////////////////////////////////////////
///
//   ____  ____
//  /   /\/   /
// /___/  \  /    Vendor: Xilinx
// \   \   \/     Version : 10.1.03
//  \   \         Application : xaw2verilog
//  /   /         Filename : digitalCM.v
// /___/   /\     Timestamp : 11/09/2009 20:03:18
// \   \  /  \
//  \___\/\___\
//
//Command: xaw2verilog -st H:\E155\Final_project\space_invaders\digitalCM.xaw
//H:\E155\Final_project\space_invaders\digitalCM
//Design Name: digitalCM
//Device: xc3s400-tq144-5
//
// Module digitalCM
// Generated by Xilinx Architecture Wizard
// Written for synthesis tool: XST
// Period Jitter (unit interval) for block DCM_INST = 0.03 UI
// Period Jitter (Peak-to-Peak) for block DCM_INST = 1.23 ns
`timescale 1ns / 1ps

module digitalCM(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 GND_BIT;

   assign GND_BIT = 0;
   assign CLK0_OUT = CLKFB_IN;
   BUFG CLKDV_BUFG_INST (.I(CLKDV_BUF),
                         .O(CLKDV_OUT));
   BUFG CLKFX_BUFG_INST (.I(CLKFX_BUF),
                         .O(CLKFX_OUT));
   BUFG CLK0_BUFG_INST (.I(CLK0_BUF),
                        .O(CLKFB_IN));
   DCM DCM_INST (.CLKFB(CLKFB_IN),
                 .CLKIN(CLKIN_IN),
```

```
                  .DSSEN(GND_BIT),
                  .PSCLK(GND_BIT),
                  .PSEN(GND_BIT),
                  .PSINCDEC(GND_BIT),
                  .RST(RST_IN),
                  .CLKDV(CLKDV_BUF),
                  .CLKFX(CLKFX_BUF),
                  .CLKFX180(),
                  .CLK0(CLK0_BUF),
                  .CLK2X(),
                  .CLK2X180(),
                  .CLK90(),
                  .CLK180(),
                  .CLK270(),
                  .LOCKED(LOCKED_OUT),
                  .PSDONE(),
                  .STATUS());
   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 = 25.000;
   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'h8080;
   defparam DCM_INST.PHASE_SHIFT = 0;
   defparam DCM_INST.STARTUP_WAIT = "FALSE";

endmodule
```