Jerry Wang jyz.wang@mail.utoronto.ca 1000009457
Soon Chee Loong cheeloong.soon@mail.utoronto.ca 999295793

ECE241 Project Fall 2013: The Motion Tracking Based Game

A) INTRODUCTION

In this project, our goal was to apply real-time computer vision and image processing algorithms on a game using the Altera DE2 board. Our motivation came primarily from Digital Canvas, the only past ECE241 project posted that uses both a camcorder and a Visual Graphics Display (VGA). We also wanted to work on a challenging project to compete for bonus marks.

Our project, *The Motion Tracking Based Game* is a game where a player controls a white square to avoid collisions with incoming coloured squares. The player wins when the player is able to pass all ten levels, whereby each level increases the number of squares that appear concurrently on the screen as shown in Figure 2.

This project is unique because the player controls the white square by moving a green object of any size in front of a HDR-CX260V Sony Camcorder. The camcorder tracks the motion of the green object and our algorithm accurately moves the white square on the screen, regardless of the distance and size of the green object with respect to the camcorder. Refer to *Appendix A* for *GamePlay Instructions*.

B) DESIGN

Top Level Design

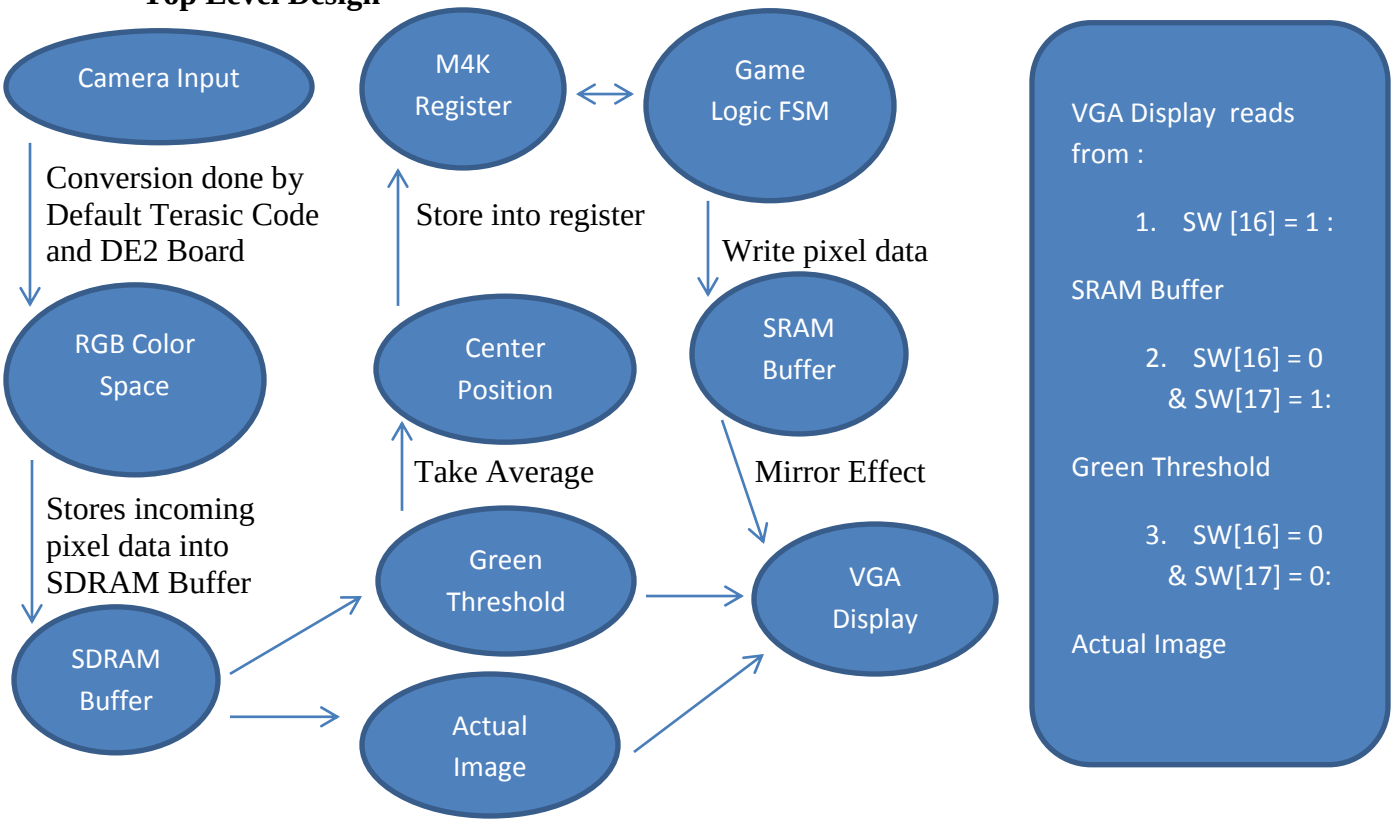

Figure 1: RTL Level Design of Entire System

Module 1: SDRAM Buffer

This module handles storing pixel data from the camera input through the DE2 board's ITU-R 656 Decoder into the SDRAM Buffer. As taught in lecture, the SDRAM buffer is able to store a lot of information but has to be refreshed continuously. This fits well with its application in our project where the SDRAM has to continuously update its data real time from the camera.

Module 2: Green Threshold

This module detects only a certain color of green threshold set by SW[15:0] and outputs it as a white color into the display. If the color of any pixel does not pass this threshold requirement, it is outputted as a black color on the display. This is done by FSM 1 (Finite State Machine) in the DE2_TV.v file. We chose green to be our threshold as this is the color that is most sensitive to the human eye. Thresholding is shown in Figure 3.

Module 3: Center Position

This module captures all pixel data outputted from the Green Threshold. If the pixel passes the requirement, it adds that pixels current x and y position to a sum and updates a counter that counts the total number of pixels out of the 640 * 320 pixels that are checked periodically. At the end of all 640 * 320 pixels, this module divides the total sum of the x and y position by the total number of pixels that passes the Green Threshold requirement. As a result, it outputs the center position of the green threshold. This is done by FSM 2 in the DE2_TV.v file.

Module 4: SRAM Buffer

The SRAM buffer is updated by the Game Logic FSM located in sketch.v file. The SRAM buffer contains actual pixel information of the white square as well as all other colliding obstacle squares of the game. As the VGA display only updates itself every 60Hz, the ability for the SRAM Buffer to continuously store the same information without being needed to be refreshed fits well into this application. In addition, the mirror effect by the camera module where a player moves the green object to the left results in the display moving the white square to the right is handled before the white square pixel information is stored into the SRAM buffer. This allows the player to react by moving the green object in the proper direction as seen on the VGA display.

Module 5: Game State/Object Manipulation (GameObjManip)

This module handles the state of the game as well as the status and locations of all the objects on the screen. This module is comprised of three smaller components: the LFSR (linear feedback shift register), the main FSM (finite state machine), and a clock frequency converter. The inputs to this module are a clock for the module to run at, an active high start signal (*startgame*) and an active high *gameover* signal to reset the FSM to the idle state. The 2D memory registers which were used to improve the clarity of the code is flattened into 1D to be synthesizable into hardware. The register is 180 bits which corresponds to 18 bits per object for 10 different objects. For each object, the first 9 bits correspond to the x-coordinates of the object followed by 7 bits of its y-coordinates. The

final bit represents whether or not the object is enabled to be printed. There is also an *imagedone* signal to indicate to the upper module that the positions of the objects have been updated.

Module 6: Linear Feedback Shift Register (LFSR)

The LFSR is used as a pseudo random number generator used to produce pseudo random y-positions for the obstacles flying across the screen. To implement a generator with a larger randomness pool, we used a 29 bits shift register. The linear feedback is created by XORing the 27^{th} and 29^{th} bits. Since the our VGA module only has only 240 y positions, the LSFR mods the 29 bits on the shift register by base 240.

Module 7: Clock Frequency Converter (Objspeed)

The clock frequency converter is used to achieve a lower frequency compared to the 50Mhz clock provided on the DE2 board. This lower frequency acts as the speed of the square obstacles flying across the screen. This module creates a pulse each time a number of 50Mhz clock ticks occur. This pulse would subsequently cause our obstacles to increase their x-positions by one pixel. In effect, our objects would have a speed of *1 pixel per pulse*. Our implementation only updates the VGA buffer after all our obstacles has their x-positions updated. This influences the rate at which the VGA buffer is updated every 60Hz. As a result, a different speed is seen by the naked eye on the VGA display.

Module 8: Game Logic Finite State Machine

The Game Logic FSM of this module is implemented using three always blocks, two of which run on the *posedge* clock and the latter using the wildcard '*' operator. The two always blocks running on the *posedge* clock could be combined into one because they run on the same clock but we separated them for code readability. The combinational always block sets the next state using the *nstate* reg and this depends on the current state of the FSM which is the reg *cstate*. While the system is idle, the state remains in the idle state ST_IDLE. Once the input *startgame* changes from a 0 to 1, the state changes to ST_INITALIZE and the clocked always blocks initializes all the pertinent registers. This leads immediately to the ST_GAME state which is the state in which obstacles are activated or deactivated and the x-positions of the objects are updated. Immediately after the FSM goes to the ST_GAME2 state which has the sole purpose of indicating to the upper module that the FSM has finished updating the objects' positions for the current clock cycle. Finally, if the FSM detects that all the objects have passed through the entire screen then the game goes into ST_WAIT to give the player a short amount of pause before the next level (one extra object) is initiated. The objects having all passed through the screen is indicated by the wire *objdone* which is assigned to the 18^{th} bit of all the objects OR'd together – this essentially means that if all the objects are disabled, then all the objects are done moving through the screen. The game ends if all 10 levels are completed or if a *gameover* signal is received.

Module 9: Sketch

This module gives the upper modules the pixel information for every pixel that needs to be updated. The inputs to this module are a clock for the module to run at, a *startgame* active high signal, and the center x and y coordinates of the player-controlled

object. In turn, this module outputs the x-coordinates, y-coordinates, colour, and write enable (*writeEn*) signals. This module uses three smaller modules recycled from lab 7 in order to output correct pixel information. The functions of the three modules will be outlined in more detail further down. This sketch module instantiates a *GameObjManip* module which provides the x and y coordinates of all the objects as well as an enable signal (*imagedone*) to trigger the process of outputting all the pixels. Once the *imagedone* active-low signal is received, the *clearscreen* module activates setting all the pixels in the memory to black effectively clearing the screen. Once this is done, a subsequent enable signal is sent to the first object drawing module pair composed of a *downcounter25* and *xyarith* module. These two modules work in conjunction to output pixel information that would result in a square on the screen. Once this square is finished drawing, it enables the next square to be drawing and continues in sequential fashion until all the squares are drawn.

As well, this module also contains the collision detection (*ColDETCT*) module. The detection module works by checking the x/y coordinates of the player-controlled object against the x/y coordinates of all the other objects. If the player-controlled square is within the obstacles square, collision is detected. The detection is all combinational logic to see whether or not any of the pixels are overlapping.

Module 10: DownCounter25 & XYarith

This module once enabled starts two counters x and y count. Both counters count to 5 with y counting once every time x counts to 5. These counters are passed to *xyarith* which subsequently adds the counter to the x/y coordinates of the object in order to produce the coordinates of all the pixels which form the square. While *downcounter25* is enabled, it also sets its *plot*/*writeEn* signal to 1 which is used to indicate to an upper module that there is a pixel to be written to memory. As well, depending on which *plot* signal is enabled (each square has its own) a different colour is chosen based on combinational logic and outputted giving each square a distinct colour. If the current square is no longer active within the *GameObjManip* module, the *nodraw* input which is linked to the 18^{th} bit of the objects' register is used to tell the counter to not output anything but merely produce an enable signal for the next square.

Module 11: Clearscreen

While it is possible to clear only a square to black for each object, I found it would be much simpler to clear the entire screen and print the objects immediately after. Since all these operations occur at 50 MHz, it should not have an effect on the 60Hz picture being outputted.

C) RESULTS

The project works as intended. The player is able to move any green size object by first setting the threshold before beginning to play the game from any distance from the camera. However, a small flaw would be the horizontal blue line that flickers from the bottom to the top of the screen as shown in Figure 2. This may be because none of our write enable signals are perfectly synchronized to the VGA adapter. Originally, our design was to update the memory during the HSYNC/VSYNC periods because we were only using a single port RAM rather than

double port and could only either read or write at a single clock cycle. We soon realized that we updated far too many pixels to fit all the memory update operations within the HSYNC/VSYNC periods. This resulted in the game running too slowly for a person to enjoy playing it Hence, we ignored synchronization and only updated if there was a write enable signal, the memory would only write and continue to write until the memory was done completely updating. Since this occurs much faster than 60Hz, it does not cause much trouble. However, for the periods where it is completely out of sync with the VGA adapter, it is possible that this is the reason for the blue line that flickers upwards on the screen.

Figure 2: Colliding objects increases as game level increases. Also shows blue flickering.

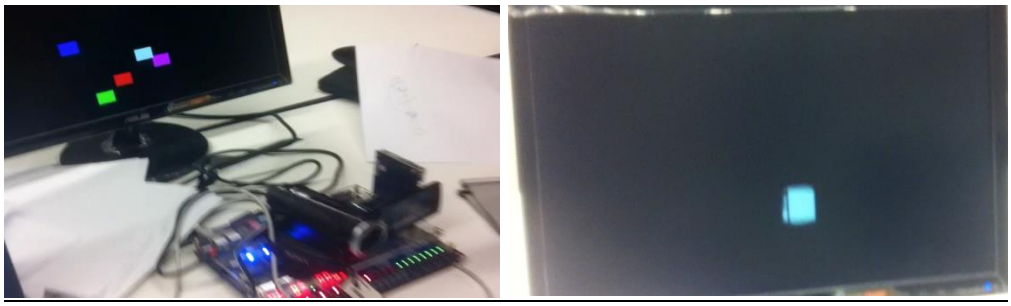

Figure 3: Camcorder setup and Thresholding on green object

D) FUTURE CHANGES

In the future, we would consider implementing a dual port RAM instead of synchronizing our currently implemented single port RAM to allow quicker access and also possibly get rid of the blue line flickering. Also, we would consider combining *downcounter25* and *xyarith* to clear individual blocks instead of the entire screen. This avoids updating unnecessary memory locations.

We would also consider working with the SOPC Builder and NIOS II in the future to reduce the amount of code written and to use some of the Intellectual Property Real-Time Image Processing MegaFunctions provided by Terasic. However, this adds largely to the compilation time and should only be done if we are very confident with our code. Our compilation time for this entire project was only below 45 seconds as we ensured that we use our memory resources wisely.

Finally, we will work on incorporating game sounds using the DE2 Board's Audio Codec so that the player gains a better gaming experience playing our *Motion Tracking Based Game*.

E) APPENDICES

Appendix A: GamePlay Instructions

Step 1)

Ensure camera is detecting images properly by setting SW [16] = 0 and SW [17] = 0. If the camera detects properly, move on to step 2. If not, try restarting the DE2 Board and reprogramming the game onto the DE2 board. Check the wires and connections.

Step 2)

Set the threshold level of the green object by setting SW[16] = 0 and SW[17] = 1. Play with SW [15:0] to ensure that the camera only detects your green object and not any other background. Once this is done, move on to Step3.

Step 3)

Go to the game display by setting SW[16] = 1. Refresh the game by hitting KEY[2] and start the game by hitting KEY[1]. The game should end after you pass all 10 levels or lost by colliding with any of the obstacles.

Appendix B: Verilog Code

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

Appendix B.1: DE2_TV.v

```
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//
=============================================================================
// Copyright (c) 2012 by Terasic Technologies Inc.
//
=============================================================================
//
// Permission:
//
//   Terasic grants permission to use and modify this code for use
//   in synthesis for all Terasic Development Boards and Altera Development
//   Kits made by Terasic.  Other use of this code, including the selling
//   , duplication, or modification of any portion is strictly prohibited.
//
// Disclaimer:
//
//   This VHDL/Verilog or C/C++ source code is intended as a design reference
//   which illustrates how these types of functions can be implemented.
//   It is the user's responsibility to verify their design for
//   consistency and functionality through the use of formal
```

```
//   verification methods.  Terasic provides no warranty regarding the use
//   or functionality of this code.
//
//
=============================================================================
//
//  Terasic Technologies Inc
//  9F., No.176, Sec.2, Gongdao 5th Rd, East Dist, Hsinchu City, 30070. Taiwan
//
//
//
//                     web: http://www.terasic.com/
//                     email: support@terasic.com
//
//
=============================================================================
//
// Major Functions:     DE2 TV_BOX
//
//
=============================================================================
//
// Revision History :
//
=============================================================================
//   Ver  :| Author           :| Mod. Date :| Changes Made:
//   V1.0 :| Joe Yang         :| 05/07/05   :| Initial Revision
//   V1.1 :| Johnny Chen      :| 05/09/05   :| Changed YCbCr2RGB Block,
//                                                                        RGB
output 8 Bits => 10 Bits
//   V1.2 :| Johnny Chen            :| 05/10/05   :| H_SYNC & V_SYNC Timing fixed.
//   V1.3 :| Johnny Chen      :| 05/11/16   :| Added FLASH Address FL_ADDR[21:20]
//   V1.4 :| Joe Yang         :| 06/07/20   :| Modify Output Color
//         V2.0 :| Johnny Chen     :| 06/11/20   :| New Version for DE2 v2.X PCB.
//   V3.0 :| Dee Zeng         :| 2012/01/30 :| change TD decoder chip from ADV7181 to
ADV7180. support PAL and NTSC Format
//   V3.1 :| Dee Zeng         :| 2012/04/20 :| add TV i2c software reset. small changes in
TD_detect.
//
=============================================================================
module DE2_TV
```

```
        (
        ////////////////////  Clock Input     ////////////////////
        CLOCK_27,                           //      On Board 27
MHz
        CLOCK_50,                           //      On Board 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
        //////////////////////    LED     //////////////////////
        LEDG,                               //      LED
Green[8:0]
        LEDR,                               //      LED Red[17:0]
        //////////////////////    UART    //////////////////////
        UART_TXD,                           //      UART
Transmitter, breaks parallel into serial
        UART_RXD,                           //      UART
Receiver, combines serial into parallel
        //////////////////////    IRDA    //////////////////////
        IRDA_TXD,                           //      IRDA
Transmitter
        IRDA_RXD,                           //      IRDA
Receiver
        ////////////////////  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 1
        DRAM_CLK,                               //  SDRAM
Clock
        DRAM_CKE,                               //  SDRAM
Clock Enable
        ////////////////// Flash Interface    ////////////////
        FL_DQ,                                  //
    FLASH Data bus 8 Bits
        FL_ADDR,                                //  FLASH
Address bus 20 Bits
        FL_WE_N,                                //  FLASH Write
Enable
        FL_RST_N,                               //  FLASH Reset
        FL_OE_N,                                //  FLASH
Output Enable
        FL_CE_N,                                //  FLASH Chip
Enable
        ////////////////// 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
        ////////////////////  ISP1362 Interface     ////////////////
        // Single Chip "Universal Serial Bus (USB) On The Go" Controller
        // On the Go means the USB can be used as peripheral or host
        OTG_DATA,                                   //  ISP1362 Data
bus 16 Bits
        OTG_ADDR,                                   //  ISP1362
Address 2 Bits
        OTG_CS_N,                                   //  ISP1362 Chip
Select
        OTG_RD_N,                                   //  ISP1362 Read
        OTG_WR_N,                                   //  ISP1362 Write
        OTG_RST_N,                                  //  ISP1362 Reset
        OTG_FSPEED,                                 //  USB
Full Speed,  0 = Enable, Z = Disable
        OTG_LSPEED,                                 //  USB
Low Speed,  0 = Enable, Z = Disable
        OTG_INT0,                                   //  ISP1362
Interrupt 0
        OTG_INT1,                                   //  ISP1362
Interrupt 1
        OTG_DREQ0,                                  //
    ISP1362 DMA Request 0
        OTG_DREQ1,                                  //
    ISP1362 DMA Request 1
        OTG_DACK0_N,                                //  ISP1362
DMA Acknowledge 0
        OTG_DACK1_N,                                //  ISP1362
DMA Acknowledge 1
        ////////////////////  LCD Module 16X2       ////////////////
        LCD_ON,                                     //  LCD
Power ON/OFF
        LCD_BLON,                                   //  LCD Back
Light ON/OFF
        LCD_RW,                                     //  LCD
Read/Write Select, 0 = Write, 1 = Read
        LCD_EN,                                     //  LCD
Enable
        LCD_RS,                                     //  LCD
Command/Data Select, 0 = Command, 1 = Data
```

```
        LCD_DATA,                                   //      LCD Data bus
8 bits
        ////////////////// SD_Card Interface     ////////////////
        SD_DAT,                                     //      SD
Card Data
        SD_WP_N,                                    //      SD Write
protect
        SD_CMD,                                     //      SD
Card Command Signal
        SD_CLK,                                     //      SD
Card Clock
        ////////////////// USB JTAG link         //////////////////
        TDI,                                        //      CPLD ->
FPGA (Data in)
        TCK,                                        //      CPLD ->
FPGA (Clock)
        TCS,                                        //      CPLD ->
FPGA (CS)
    TDO,                                            //      FPGA ->
CPLD (Data out)
        ////////////////// I2C           ////////////////////////////
        I2C_SDAT,                                   //      I2C Data
        I2C_SCLK,                                   //      I2C Clock
        ////////////////// PS2           ////////////////////////////
        PS2_DAT,                                    //      PS2 Data
        PS2_CLK,                                    //      PS2 Clock
        ////////////////// VGA           ////////////////////////////
        VGA_CLK,                                    //      VGA Clock
        VGA_HS,                                     //      VGA
H_SYNC
        VGA_VS,                                     //      VGA
V_SYNC
        VGA_BLANK,                                  //      VGA
BLANK
        VGA_SYNC,                                   //      VGA SYNC
        VGA_R,                                      //      VGA Red[9:0]
        VGA_G,                                      //      VGA
Green[9:0]
        VGA_B,                                      //      VGA Blue[9:0]
        ////////////    Ethernet Interface      ////////////////////////
        ENET_DATA,                                  //
    DM9000A DATA bus 16Bits
        ENET_CMD,                                   //      DM9000A
Command/Data Select, 0 = Command, 1 = Data
        ENET_CS_N,                                  //      DM9000A
Chip Select
```

```
        ENET_WR_N,                                          //
    DM9000A Write
        ENET_RD_N,                                  //      DM9000A
Read
        ENET_RST_N,                                         //
    DM9000A Reset
        ENET_INT,                                   //      DM9000A
Interrupt
        ENET_CLK,                                   //      DM9000A
Clock 25 MHz
        ////////////////    Audio CODEC             ////////////////////////
        AUD_ADCLRCK,                                //      Audio
CODEC ADC LR Clock
        AUD_ADCDAT,                                         //      Audio
CODEC ADC Data
        AUD_DACLRCK,                                //      Audio
CODEC DAC LR Clock
        AUD_DACDAT,                                         //      Audio
CODEC DAC Data
        AUD_BCLK,                                   //      Audio
CODEC Bit-Stream Clock
        AUD_XCK,                                    //      Audio
CODEC Chip Clock
        ////////////////    TV Decoder          ////////////////////////
        TD_DATA,                            //      TV Decoder Data bus
8 bits
        TD_HS,                                              //      TV
Decoder H_SYNC
        TD_VS,                                              //      TV
Decoder V_SYNC
        TD_RESET,                                   //      TV Decoder
Reset
        TD_CLK27,           //      TV Decoder 27MHz CLK
        ////////////////////  GPIO  ////////////////////////////
        GPIO_0,                                             //      GPIO
Connection 0
        GPIO_1                                              //      GPIO
Connection 1
    );

//---------------------------------------------------------------------------------------
// Defininig inputs and outputs
////////////////////////    Clock Input     ////////////////////////
input           CLOCK_27;                   //      On Board 27 MHz
input           CLOCK_50;                   //      On Board 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 Display    //////////////////////
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
////////////////////////    LED    ////////////////////////
output   [8:0]   LEDG;                         //    LED Green[8:0]
output   [17:0]  LEDR;                         //    LED Red[17:0]
////////////////////////    UART ////////////////////////
output             UART_TXD;                   //    UART Transmitter
input              UART_RXD;                   //    UART Receiver
////////////////////////    IRDA  ////////////////////////
output             IRDA_TXD;                   //    IRDA Transmitter
input              IRDA_RXD;                   //    IRDA Receiver
//////////////////////        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
//////////////////////    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 Low-
byte Data Mask
output          SRAM_LB_N;                              //      SRAM High-
byte Data Mask
output          SRAM_WE_N;                              //      SRAM Write
Enable
output          SRAM_CE_N;                              //      SRAM Chip
Enable
output          SRAM_OE_N;                              //      SRAM Output
Enable
//////////////////// ISP1362 Interface     //////////////////////
inout   [15:0] OTG_DATA;                //      ISP1362 Data bus 16 Bits
output [1:0]    OTG_ADDR;               //      ISP1362 Address 2 Bits
output          OTG_CS_N;                       //      ISP1362 Chip Select
output          OTG_RD_N;                       //      ISP1362 Write
output          OTG_WR_N;                       //      ISP1362 Read
output          OTG_RST_N;                      //      ISP1362 Reset
output          OTG_FSPEED;                             //      USB Full
Speed, 0 = Enable, Z = Disable
output          OTG_LSPEED;                             //      USB Low
Speed, 0 = Enable, Z = Disable
input           OTG_INT0;                       //      ISP1362 Interrupt 0
input           OTG_INT1;                       //      ISP1362 Interrupt 1
input           OTG_DREQ0;                              //      ISP1362
DMA Request 0
input           OTG_DREQ1;                              //      ISP1362
DMA Request 1
output          OTG_DACK0_N;                    //      ISP1362 DMA
Acknowledge 0
```

```
output          OTG_DACK1_N;                    //      ISP1362 DMA
Acknowledge 1
////////////////// LCD Module 16X2    ////////////////////////////
inout   [7:0]   LCD_DATA;               //      LCD Data bus 8 bits
output          LCD_ON;                         //      LCD Power
ON/OFF
output          LCD_BLON;               //      LCD Back Light
ON/OFF
output          LCD_RW;                         //      LCD
Read/Write Select, 0 = Write, 1 = Read
output          LCD_EN;                         //      LCD Enable
output          LCD_RS;                         //      LCD
Command/Data Select, 0 = Command, 1 = Data
////////////////// SD Card Interface  ////////////////////
inout   [3:0]   SD_DAT;                 //      SD Card Data
input           SD_WP_N;                //      SD write protect
inout           SD_CMD;                         //      SD Card
Command Signal
output          SD_CLK;                         //      SD Card
Clock
////////////////////    I2C     ////////////////////////////////
// Used to initialize ADV7181B chip
inout           I2C_SDAT;               //      I2C Data
output          I2C_SCLK;               //      I2C Clock
////////////////////    PS2     ////////////////////////////////
input           PS2_DAT;                //      PS2 Data
input           PS2_CLK;                //      PS2 Clock
////////////////// USB JTAG link      ////////////////////////////
input           TDI;                    // CPLD -> FPGA (data in)
input           TCK;                    // CPLD -> FPGA (clk)
input           TCS;                    // CPLD -> FPGA (CS)
output          TDO;                    // FPGA -> CPLD (data out)
////////////////////    VGA         ////////////////////////////
output          VGA_CLK;                //      VGA Clock
output          VGA_HS;                         //      VGA
H_SYNC
output          VGA_VS;                         //      VGA
V_SYNC
output          VGA_BLANK;                      //      VGA BLANK
output          VGA_SYNC;               //      VGA SYNC
output  [9:0]   VGA_R;                  //      VGA Red[9:0]
output  [9:0]   VGA_G;                  //      VGA Green[9:0]
output  [9:0]   VGA_B;                  //      VGA Blue[9:0]
///////////////     Ethernet Interface      ////////////////////////////
inout   [15:0]  ENET_DATA;              //      DM9000A DATA bus
16Bits
```

```
output          ENET_CMD;                       //      DM9000A
Command/Data Select, 0 = Command, 1 = Data
output          ENET_CS_N;                      //      DM9000A Chip
Select
output          ENET_WR_N;                      //      DM9000A
Write
output          ENET_RD_N;                      //      DM9000A Read
output          ENET_RST_N;                     //      DM9000A
Reset
input           ENET_INT;                       //      DM9000A Interrupt
output          ENET_CLK;                       //      DM9000A Clock 25
MHz
//////////////////  Audio CODEC             ////////////////////////////
inout           AUD_ADCLRCK;                    //      Audio CODEC ADC
LR Clock
input           AUD_ADCDAT;                     //      Audio
CODEC ADC Data
inout           AUD_DACLRCK;                    //      Audio CODEC DAC
LR Clock
output          AUD_DACDAT;                     //      Audio
CODEC DAC Data
inout           AUD_BCLK;                       //      Audio
CODEC Bit-Stream Clock
output          AUD_XCK;                        //      Audio CODEC Chip
Clock
//////////////////  TV Decoder          ////////////////////////////
input   [7:0]   TD_DATA;            //      TV Decoder Data bus 8 bits
input           TD_HS;                          //      TV Decoder
H_SYNC
input           TD_VS;                          //      TV Decoder
V_SYNC
output          TD_RESET;                       //      TV Decoder Reset
input       TD_CLK27;       //      TV Decoder 27MHz CLK
//////////////////////      GPIO    ////////////////////////////////
inout   [35:0]  GPIO_0;                         //      GPIO Connection 0
inout   [35:0]  GPIO_1;                         //      GPIO Connection 1
//-------------------------------------------------------------------------------------
// Wires & Registers

// For SRAM Buffer of VGA
reg [17:0] addr_reg; //memory address register for SRAM
reg [15:0] data_reg; //memory data register  for SRAM
reg we ;            //write enable for SRAM, it writes on active low
// but write enable for VGA wants to write on active high = writeEn

assign SRAM_DQ = 16'hzzzz;
```

```
// SRAM_control
assign SRAM_ADDR = addr_reg; // depends on position of current VGA_X and
VGA_Y position
assign SRAM_DQ = (we)? 16'hzzzz : data_reg ; // Depends on write enable
                                        // if we = 1 => read, let data be impedance, it should
read from proper values
                                        // if we = 0 => write, let it write whatever is in the
data register

assign SRAM_UB_N = 0;                                   // hi byte select enabled
assign SRAM_LB_N = 0;                                   // lo byte select enabled
assign SRAM_CE_N = 0;                                   // chip is enabled
assign SRAM_WE_N = we;                                  // write when ZERO
assign SRAM_OE_N = 0;                                   //output enable is overidden
by WE

//------------------------------------------------------------------------

// r4r3r2r1 g4g3g2g1 b4b3b2b1 xxxx // -> 12 bit color into 16 bit input
reg [15:0] colorSDRAM, colorSDRAM2; // this is color input into SDRAM
wire [9:0] mRedSRAM, mGreenSRAM, mBlueSRAM;
// temporary for understanding:
assign  mRedSRAM = {SRAM_DQ[15:12], 6'b0}; // The colors that output to the whole
vga every 60HZ is

        // Note: every pixel updates every few ns depending

        // on how fast HSYNC and YSYNC runs

        // but the whole screen is updated every 60 HZ
assign  mGreenSRAM = {SRAM_DQ[11:8], 6'b0}; // each color is 4 bits
assign  mBlueSRAM = {SRAM_DQ[7:4], 6'b0};
// color data written to SRAM is 12 bit color , but data is 16 bit
// SRAM_DQ[3:0] is used for
// but data register is 16'b0 => black
                                                        // 16'hFFFF => white

//------------------------------------------------------------------------

//      For Audio CODEC
wire            AUD_CTRL_CLK;   //      For Audio Controller

//      For ITU-R 656 Decoder
wire    [15:0]  YCbCr;
wire    [9:0]   TV_X;
```

```
wire            TV_DVAL;

//      For VGA Controller
wire    [9:0]   mRed;
wire    [9:0]   mGreen;
wire    [9:0]   mBlue;
wire    [10:0]  VGA_X;
wire    [10:0]  VGA_Y;
wire            VGA_Read;   //      VGA data request
wire            m1VGA_Read;     //      Read odd field
wire            m2VGA_Read;     //      Read even field

//      For YUV 4:2:2 to YUV 4:4:4
wire    [7:0]   mY;
wire    [7:0]   mCb;
wire    [7:0]   mCr;

//      For field select
wire    [15:0]  mYCbCr;
wire    [15:0]  mYCbCr_d;
wire    [15:0]  m1YCbCr;
wire    [15:0]  m2YCbCr;
wire    [15:0]  m3YCbCr;

//      For Delay Timer
wire            TD_Stable;
wire            DLY0;
wire            DLY1;
wire            DLY2;

//      For Down Sample
wire    [3:0]   Remain;
wire    [9:0]   Quotient;

wire            mDVAL;

wire    [15:0]  m4YCbCr;
wire    [15:0]  m5YCbCr;
wire    [8:0]   Tmp1,Tmp2;
wire    [7:0]   Tmp3,Tmp4;

wire        NTSC;
wire        PAL;

wire    i2c_av_config_reset_n;
wire    tv_detect_reset_n;
```

```
//-------------------------------------------------------------------------------------
//=====================================================================
===============
// Structural coding
//=====================================================================
===============
//      Flash
assign  FL_RST_N    =       1'b1; // assign Flash Memory REset to 1 bit

//      16*2 LCD Module
assign  LCD_ON              =       1'b1;   //      LCD ON
assign  LCD_BLON    =       1'b1;   //      LCD Back Light

//      All inout port turn to tri-state
assign  SD_DAT              =       4'bzzzz; // SD Data to tristate
assign  AUD_ADCLRCK     =       AUD_DACLRCK; // Audio codec ADC Clock =
Audio Codec DAC Clock

//      Disable USB speed select
assign  OTG_FSPEED      =       1'bz;
assign  OTG_LSPEED      =       1'bz;

// Set all IO Ports  (there are 2 36 pins IO ports)
assign  GPIO_0      =       36'hzzzzzzzzz; // Set all 36 pins of first IO ports to tristate
assign  GPIO_1      =       36'hzzzzzzzzz; // Set all 36 pins of second IO ports to
tristate

//      Enable TV Decoder , TD => TV Decoder
assign  TD_RESET    = 1'b1; // low active ...use name: TD_RESET_N? , never reset the
TV Decoder

//      For Audio CODEC
assign  AUD_XCK     =       AUD_CTRL_CLK;
// AUD_XCK = AUDIO CODEC Chip Clock,
// AUD_CTRL_CLK = A wire to control AUD_XCK

// Let RED LED show the state of wire VGA_X
// Let GREEN LED show the state of wire VGA_Y
assign  LEDG =      VGA_Y; // [10:0]
// assign       LEDR =      VGA_X; // [10:0]
// only uses the first LEDG0 to LEDG7 to display VGA_Y position cause not enough
LEDG
// LEDR is always on cause the clock goes too fast so it appears that all of them are on at
the same time.

// For video interlacing
```

```
// m1VGA_Read is wire to read odd file , m2VGA_Read is wire to read even field
// VGA_Y is the current Y position given out by the VGA
// Y incremements from 0 to ... from top of screen to btm.
// when y is an oddfield, it the first bit (in binary) will jump from 0 1 0 1 0 1 => 0 -> odd,
1 => even

// This determines which data is read from the SRAM BUFFER INTO THE YUV444 to
YUV 442 converter

// VGA_READ is a trigger generated by the VGA when it wants to read for X and Y.
// so basically if VGA_read is 0, you are also not reading from the sdram
assign  m1VGA_Read=     VGA_Y[0]   ?     1'b0          :     VGA_Read   ;
// odd field , if odd, m1VGA_READ is VGA_READ
assign  m2VGA_Read=     VGA_Y[0]          ?     VGA_Read   :     1'b0
        ; // even field

// Note: Either m1VGA_Read or m2VGA_read will be 1 at anytime, both depends on
status of VGA_Y[0]
// VGA_Y[0] = first bit of VGA_Y position
// VGA_Read is a wire for signaling VGA Data request

// Used for Field Select
// mYCbCr_d = a wire for field select, selects either from these 2 depending on NOT
VGA_Y[0]
// Every mY and m5Y thing here are wires

assign  mYCbCr_d   =     !VGA_Y[0]          ?     m1YCbCr     :  m2YCbCr;
assign  mYCbCr             =     m5YCbCr;

// These are used for deinterlacing from SDRAM before giving the interlaced version to
YUV422 to YUV444 converter
// mYCbCr_d is given by SDRAM directly from either odd or even field
// m3YCbCr is given from mYCbCr_d after going through line buffer 1, activates from
VGA_READ
// m4YCbCr is given from m3YCbCr after going through line buffer 2, activates from
VGA_READ
assign  Tmp1  =     m4YCbCr[7:0]+mYCbCr_d[7:0];          // adding 2 8 bits
gives a 9 bit
assign  Tmp2  =     m4YCbCr[15:8]+mYCbCr_d[15:8];  // adding 2 8 bits gives a 9 bit
assign  Tmp3  =     Tmp1[8:2]+m3YCbCr[7:1];                    // adding a 7
bit and a 7 bit gives a 8 bit
assign  Tmp4  =     Tmp2[8:2]+m3YCbCr[15:9];                   // adding a 7
bit and a 7 bit gives a 8 bit

// m5YCbCr = Tmp4 (first 8 bits) followed by Tmp3 (last 8 bits) => 16 bits
assign  m5YCbCr     =     {Tmp4,Tmp3}; // This concatenates both bits together
```

```
//      7 segment LUT for HEX Displays, controls all 7 HEX with 18 switches
// Note: Will only control HEX0 -HEX3 and a bit of HEX4 due to not enough switches
SEG7_LUT_8                  u0      (.oSEG0(HEX0),
                                    .oSEG1(HEX1),
                                    .oSEG2(HEX2),
                                    .oSEG3(HEX3),
                                    .oSEG4(HEX4),
                                    .oSEG5(HEX5),
                                    .oSEG6(HEX6),
                                    .oSEG7(HEX7),
                                    .iDIG(SW) );


//      TV Decoder Stable Check and if it is NTSC or PAL
TD_Detect                   u2      (.oTD_Stable(TD_Stable),
                                    .oNTSC(NTSC),
                                    .oPAL(PAL),
                                    .iTD_VS(TD_VS), // TD_VS = input
TV Decoder Vertical Sync
                                    .iTD_HS(TD_HS), // TD_HS = input
TV Decoder Horizontal Sync
                                    .iRST_N(KEY[0])); // Reset key is
KEY[0]

        // Note: When u click reset (KEY[0]), it might not detect camera anymore

        // need to reprogram de2 after turning on and off the board to work

        // when reprogram after restarting board , need make sure camcorder is off.




//      Reset Delay Timer, generates 3 different resets where reset 0 appears first,
follows by reset1 followed by reset2
// all 3 resets becomes 0 if TD_Stable becomes a 0 which happens when NTSC changes
to PAL or vice versa
// or if resetN from KEY[0] is given
Reset_Delay                 u3      (       .iCLK(CLOCK_50),
                                    .iRST(TD_Stable),
                                    // Note: At key[0] pressed, all resets
at same time
                                    // if not, it takes turns to reset
```

```
                                                .oRST_0(DLY0), // Reset delay for
DIV && Sdram_Control_4Port && YUV422_to_444
                                                .oRST_1(DLY1), // Reset delay for
ITU_656_Decoder && AUDIO_DAC
                                                .oRST_2(DLY2)); // Reset delay for
YCbCr2RGB


// the output of this is YCbCr which gets written into the SDRAM
// Input of this is 8 bit data coming from the TD_DATA
// this output is 16 bit, the 16 bit represents the color in YUV 4:2:2 mode
//        ITU-R 656 to YUV 4:2:2
ITU_656_Decoder              u4      (         //        TV Decoder Input
                                                .iTD_DATA(TD_DATA), // input
from TV Decoder, data bus 8 bits

          // its connected to pins, like CLOCK_50, its just a data given by TV
                                                //        Position Output
                                                .oTV_X(TV_X), // a [9:0] wire
representing TV_X's position
                                                //        YUV 4:2:2 Output
                                                .oYCbCr(YCbCr), // a [15:0] wire
output

        //  Data value to represent color of the YCbCr coordinate system
                                                .oDVAL(TV_DVAL), // a [0:0] wire
output from this decoder,

// tells SDRAM if data is currently valid to be stored

        // DVAL = Data Valid


                                                //        Control Signals
                                                .iSwap_CbCr(Quotient[0]), //
Quotient is a [9:0] wire  for down sample

                        // Its input is influenced by divide function at 27MHZ clock
                                                .iSkip(Remain==4'h0), // Remain is a
[3:0] wire for down sample

                        // Its input is influenced by divide function at 27MHZ clock

                        // It only cares if remainder is 0
                                                .iRST_N(DLY1),    // the reset1
delay from reset delay gives this the reset
```

```
        // it resets by after delay before DIV resets
                                                .iCLK_27(CLOCK_27));

// Megafunction:
// i) input numerator : 10 bits unsigned
// ii) input denominator: 4 bits unsigned
// iii) Output latency of 1 clock cycle
// iv) Always returns positive remainder
//      For Down Sample 720 to 640, this down samples the x position given by
// ITU_656_Decoder from 720 pixels wide to 640 pixels wide
// Wizard generated file for divide
// This divides input numerator by denominator and outputs quotient and its remainder
DIV                     u5      (       .aclr(!DLY0),  // this resets after a delay
before TD_Stable's negedge
                                                .clock(CLOCK_27),
                                                .denom(4'h9),   // input denominator
is always a constant which is 9
                                                .numer(TV_X), // input numerator
[9:0] which is outputted from ITU_656_Decoder

        // which is the TV's current X position
                                                .quotient(Quotient), // These are
output for quotient [9:0]
                                                .remain(Remain)); // This is an
output [3:0]

//      SDRAM frame buffer
Sdram_Control_4Port u6      (       //      HOST Side
                                        .REF_CLK(CLOCK_27), // Note: Runs at
27MHz clock
                                         .CLK_18(AUD_CTRL_CLK), // 18
MHZ clock
                                        .RESET_N(DLY0),
                                            //      FIFO Write Side 1
                                        .WR1_DATA(YCbCr), // The color value
in YCbCr from ITU_656_Decoder

        // [15:0] YCbCr


                                                // Writes whenever TV decoder tells
him its valid
                                                .WR1(TV_DVAL), // enable signal
by ITU_656_Decoder, lets you know if its time to write
```

```
        // Data Valid
                                                .WR1_FULL(),
                                                .WR1_ADDR(0), // start writing
frrom this memory address
                                                .WR1_MAX_ADDR(NTSC ?
640*507 : 640*576),             //      525-18 , the size of your write depending on tv type

        // stop writing on this memory address
                                                .WR1_LENGTH(9'h80), // this is 8
bits,
                                                .WR1_LOAD(!DLY0),
                                                .WR1_CLK(CLOCK_27),
                                                //      FIFO Read Side 1, First in
First Out
                                                .RD1_DATA(m1YCbCr),
                                        .RD1(m1VGA_Read), // tells to read odd field, this
will be 1 when VGA requests for a read

        // and this reads out into m1YCbCr which is 16 bits

        // Depending on whether it is even or add,

        // either one of m1YCbCr or m2YCbCr will be read out and

        // mYCbCr_d = m1YCbCr when VGAY[0] is 0

        // mYCbCr_d = m1YCbCr when VGAY[0] is 1

        // assign      mYCbCr_d    =     !VGA_Y[0]          ?      m1YCbCr     :
m2YCbCr             ;


        // this final myCbCr gets combine in some special way to m5YCbCr

        // and this m5YCbCr register is finally connected to wire mYCbCr

        // assign      mYCbCr              =     m5YCbCr;

        // mYCbCr is then connected to YUV422_to_444

        // to get YUV_444. look there for continuation
```

```
                                                //      Read odd field and bypass
blanking
                                        .RD1_ADDR(NTSC ? 640*13: 640*22),      // :
640*42),
                                                .RD1_MAX_ADDR(NTSC ?
640*253 : 640*262),//: 640*282),
                                                .RD1_LENGTH(9'h80), // WTF is
this?
                                        .RD1_LOAD(!DLY0),
                                                .RD1_CLK(CLOCK_27),
                                                //      FIFO Read Side 2
                                            .RD2_DATA(m2YCbCr),
                                                 // reads even field signal, this should
be turn on when read signal 1 is done
                                        .RD2(m2VGA_Read),
                                                //      Read even field and bypass
blanking
                                        .RD2_ADDR(NTSC ? 640*267 : 640*310), // :
640*330),
                                                .RD2_MAX_ADDR(NTSC ?
640*507 : 640*550),//: 640*570),
                                                .RD2_LENGTH(9'h80),
                                        .RD2_LOAD(!DLY0),
                                                .RD2_CLK(CLOCK_27),
                                                //      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)   );

//      Converts YUV 4:2:2 to YUV 4:4:4
YUV422_to_444           u7      (       //      YUV 4:2:2 Input
                                                .iYCbCr(mYCbCr), // from
Sdram_Control_4Port

// although converted from fields to mYCbCr, it is converted via

// wires so everything happens instantaneously.
                                                //      YUV   4:4:4 Output
```

```
                                                    .oY(mY),                    //
outputs 4:4:4 output of Y,Cb and Cr for YCbCr2RGB to
                                                    .oCb(mCb),                  //
convert to RGB
                                                    .oCr(mCr),
                                                    //      Control Signals
                                                    .iX(VGA_X-160),
                                                    .iCLK(CLOCK_27),
                                                    .iRST_N(DLY0));

// Converts     YCbCr 8-bit to RGB-10 bit
YCbCr2RGB                   u8      (       //      Output Side
                                                    .Red(mRed),   // outputs final red to
use
                                                    .Green(mGreen), // outputs final blue
to use
                                                    .Blue(mBlue), // outputs final green
to use
                                                    // It takes time for it to get the proper
mYmCbmCr, so
                                                    // when data is valid, you know
everything is done
                                                    .oDVAL(mDVAL), // Data Valid

        // it then gives out the colors for VGA
                                                    //      Input Side
                                                    .iY(mY), // input color
                                                    .iCb(mCb),
                                                    .iCr(mCr),
                                                    .iDVAL(VGA_Read), // it knows
that input is valid if vga_read wants to read

        // cause everything is connected in wires, so data comes instantaneously

        // from the SDRAM although it goes through all the processing.

        // which immediately generates mRed, mGreen, mBlue out.
                                                    //      Control Signal
                                                    .iRESET(!DLY2),   // This resets
after a delay after ITU656Decoder
                                                    .iCLK(CLOCK_27));

wire [9:0] mRed2, mGreen2, mBlue2; // This is connected to actual VGA
reg [9:0] mRed3, mGreen3, mBlue3, mRed4, mGreen4, mBlue4, mRed5, mGreen5,
mBlue5;
assign mRed2 = mRed5;
```

```
assign mGreen2 = mGreen5;
assign mBlue2 = mBlue5;
//assign LEDR = mGreen; // let LEDR show what the value of mGreen is

// If SW[16] is 1, it shows whatever that is stored in SRAM (actual game play)
// If SW[16] is 0, it shows whatever SW[17] determines
always @(*)
begin
        mRed5 = SW[16] ? mRed4: mRedSRAM;
        mGreen5 = SW[16] ? mGreen4: mGreenSRAM;
        mBlue5 = SW[16] ? mBlue4: mBlueSRAM;
end

// Only matters if SW[16] is 1
// If SW[17] is 1, it shows the original picture captured by camera
// If SW[17]is 0, it shows the green picture deduce by camera
// for anything that counts as 1 will be white
always @ (*)
begin
        mRed4 = SW[17]  ? mRed3: mRed;
        mGreen4 = SW[17] ? mGreen3: mGreen;
        mBlue4 = SW[17] ? mBlue3: mBlue;
end

always @ (*)
begin
        mRed3       =   ((mGreen >= SW[9:0])  && ( mRed < {SW[14:10],5'b11111}))?
9'b111111111 : 9'b0;
        mBlue3   =  ((mGreen >= SW[9:0])  && ( mRed < {SW[14:10],5'b11111})) ?
9'b111111111 : 9'b0;
        mGreen3  =   ((mGreen >= SW[9:0])  && ( mRed < {SW[14:10],5'b11111})) ?
9'b111111111 : 9'b0;
end

VGA_Ctrl                u9      (       //      Host Side,
// Proof: Since each time VGA wants data, it gets the right data, that means everything in
every module
// happens instantaneously once VGA_Read (the trigger in this file) is set
                                                        // When VGA wants to read, it reads
directly from SDRAM and not a temporary buffer.
//                                                      .iRed(mRed),
//                                                      .iGreen(mGreen),
//                                                      .iBlue(mBlue),
                                                        .iRed(mRed2), // always connect to
mRed2, if wanna connect to SRAM,
```

```
        // connect SRAM to mRedSRAM
                                                .iGreen(mGreen2),
                                                .iBlue(mBlue2),
                                                .oCurrent_X(VGA_X), // tells you
which position it is currently at,

          // updates itself every 27MHZ since not using PLL in this file
                                                .oCurrent_Y(VGA_Y),
                                                .oRequest(VGA_Read), // it requests
for reading
                                                //      VGA Side (dont have to
worry about this)
                                                .oVGA_R(VGA_R),
                                                .oVGA_G(VGA_G),
                                                .oVGA_B(VGA_B),
                                                .oVGA_HS(VGA_HS),
                                                .oVGA_VS(VGA_VS),
                                                .oVGA_SYNC(VGA_SYNC),
                                                .oVGA_BLANK(VGA_BLANK),
                                                .oVGA_CLOCK(VGA_CLK),
                                                //      Control Signal
                                                .iCLK(CLOCK_27),
                                                .iRST_N(DLY2));

// Line buffer allows you to do operation on the left to right bits!!!
// it is basically a buffer for a line of pixels!
// Megafunction // In this case, the parameters are:
// i) Distance between taps: 640
// ii) shiftin input bus:    16bits
// iii) shiftout output bus: 16bits
// altshift_taps megafunction is a parameterized shift register with
// taps. The taps provide data outputs from the shift register at certain
// points in the shift register chain. The tap points must be evenly spaced.
// Note: The camera is giving it an interlaced input
// This is used for deinterlacing from the camera input stored in the SDRAM before
// giving it to the
Line_Buffer u10      (      .aclr(!DLY0), // this only shifts every 640 bits
            .clken(VGA_Read),
                                .clock(CLOCK_27),
                                .shiftin(mYCbCr_d), // this is the result of the odd
or even from SDRAM
                                .shiftout(m3YCbCr)); // when VGA wants to read,
                                                                        //
mYCbCr_d automatically changes to this new m3YCbCr
```

```
// This is used for deinterlacing
Line_Buffer u11	(	.aclr(!DLY0), // this only shifts every 640 bits
		.clken(VGA_Read),
					.clock(CLOCK_27),
					.shiftin(m3YCbCr), // this automatically changes to
					.shiftout(m4YCbCr)); // m4YCbCr

AUDIO_DAC		u12	(	//	Audio Side
					.oAUD_BCK(AUD_BCLK),
					.oAUD_DATA(AUD_DACDAT),
					.oAUD_LRCK(AUD_DACLRCK),
					//	Control Signals
					.iSrc_Select(2'b01),
				.iCLK_18_4(AUD_CTRL_CLK),
					.iRST_N(DLY1));

// Initialize I2C to let it know you're using the I2C bus
//	Audio CODEC and video decoder setting
I2C_AV_Config	u1	(	//	Host Side
						.iCLK(CLOCK_50),
						.iRST_N(KEY[0]),
						//	I2C Side
						.I2C_SCLK(I2C_SCLK),
						.I2C_SDAT(I2C_SDAT)	);
//-------------------------------------------------------------------------------------------------------
----
// FSM1: To store color Data into SRAM whenever we is 1, and x and y is given
// Note: SDRAM only writes when its writeenable bit is 0
//-------------------------------------------------------------------------------------------------------
----
//Create SRAM to store whatever is in SDRAM
// Need to sync x and y that you give it with x and y that VGA wants from SDRAM
// addr_reg has to change depending on whether you are writing or reading from it
reg lock; // to detect synching

assign LEDR[8:0]=VGA_X[9:1];
assign LEDR[16:9]=CenterY;

assign reset = ~KEY[2]; // clear the SRAM buffer first before doing anything
assign resett = ~KEY[3];

wire [8:0] CenterY2;
assign CenterY2 = {1'b0, CenterY};

wire [8:0]xSUMed, ySUMed;
```

```
assign xSUMed = xSUM/(2*numGreenAdd);
assign ySUMed = ySUM/(2*numGreenAdd);

// SRAM doesnt depend on clock, just need wait 1 clock cycle before reading written data
always @ (posedge CLOCK_27)
begin
data_reg <= colorout;
	// only write to SRAM if synching
	// so that VGA won't read from it
	if (reset)		//synch reset assumes KEY0 is held down 1/60 second
	begin
		//clear the screen
		addr_reg <= {VGA_X[9:1],VGA_Y[9:1]} ;  // [17:0]
		we <= 1'b0;                                                    //write
some memory
		data_reg <= 16'b0;                                          //write all
zeros (black)
	end

else
	if(writeEn)
	begin
		we<=1'b0;
		addr_reg <= {9'd319-x,y2};
	end
	else
	/*if (~VGA_VS | ~VGA_HS)
	begin
			//we <= 1'b0; // tell u are writing something
			we<=1'b0;
			addr_reg <= {x,y};     // [17:0] // get address from where u need it
to be
			// write to SDRAM
			// Note: can only read this data after 1 clock cycle which is fine
			// cause you are currently in VGA sync mode -> VGA wont be
reading anything yet
	end
	else // if synching , VGA is reading from SDRAM */
	begin
			we<=1'b1;
			addr_reg <= {VGA_X[9:1], VGA_Y[9:1]} ; // [17:0] , access from
VGA access
		// do nothing
	end
end
```

```
// FSM to count number of greens which are one and add all the x and y positions up
divided
// by number of times you added them up to get middle pixel
//-------------------------------------------------------------
// FSM 2 : Determine center position
//-------------------------------------------------------------
// Note: Green pixels => anything that passes the given threshold and is outputted as 1
// numGreenAdd -> number of green pixels addd
// ySUM -> total sum of all positions of y sum belonging to green pixel added
// xSUM -> total sum of all positions of x sum belonging to green pixel added
// currY -> current y position of green pixel to be added
// currX -> current x position of green pixel to be added
// greenValid -> if current pixel is considered green
// resetN is tied to key0
// CenterX & CenterY is used in the 2nd FSM to draw the ball out to the screen

// clock is tied to whenever the data is read out from the SDRAM, VGA_Read in this case
// NOT SURE: DO I ASSUME VGA X is always equal to DATAX that i am reading in?

reg [8:0] CenterX ;
reg [7:0] CenterY ;
reg testtemp = 0;
// max. x sum if all bits are green is less than 640 x 480 x 640
// max. y sum if all bits are green is less than 480 x 480 x 640
reg [28:0] xSUM, ySUM;
wire [10:0] currX, currY;
wire greenValid;
// maximum number of greens add is 640 x 480
reg [18:0] numGreenAdd;
assign greenValid = ((mGreen >= SW[9:0]) && (mRed < {SW[14:10], 5'b11111}))? 1 :
0;
assign currX = VGA_X;
assign currY = VGA_Y;

reg [2:0] currState, nextState;
parameter
        sReset = 3'd0,
        sAdd = 3'd1,
        sIdle =3'd2,
        sCalc =3'd3,
        sDone =3'd4;

// State Operations
always @ (posedge CLOCK_27)
begin
```

```
	case(currState)
		sReset:
		begin
			numGreenAdd  <= 1;
			ySUM <= 0;
			xSUM  <= 0;
			CenterX <= 0;
			CenterY <= 0;
		end
		sAdd:
		begin
			ySUM <= ySUM + currY;
			xSUM <= xSUM + currX;
			numGreenAdd <= numGreenAdd + 1;
		end

		sIdle:
		begin
			// do nothing
		end

		sCalc:
		begin
			CenterX <= xSUMed;
			CenterY <= ySUMed;
		end

		sDone:
		begin
			numGreenAdd  <= 1;
			ySUM <= 0;
			xSUM  <= 0;
		end
	endcase
end

// State Transitions
	always@(*) // change during changes in VGA_X or VGA_Y or greenValid
	begin
		case (currState)
			sReset:
				if ((VGA_X[9:1] > 317) && (VGA_Y[9:1] > 239) &&
(numGreenAdd > 20))
					nextState = sCalc;
				else if (greenValid)
					nextState <= sAdd;
```

```
                        else
                                nextState <= sIdle;
                sAdd:
                        if ((VGA_X[9:1] > 317) && (VGA_Y[9:1] > 239) &&
(numGreenAdd > 20))
                                nextState = sCalc;
                        else if (greenValid)
                                nextState <= sAdd;
                        else
                                nextState <= sIdle;
                sIdle:
                        if ((VGA_X[9:1] > 317) && (VGA_Y[9:1] > 239) &&
(numGreenAdd > 20))
                                nextState = sCalc;
                        else if (greenValid)
                                nextState <= sAdd;
                        else
                                nextState <= sIdle;
                sCalc:
                                nextState <= sDone;
                sDone:
                        if (greenValid)
                                nextState <= sAdd;
                        else
                                nextState <= sIdle;
                default: nextState <= sReset; // initialize to sReset state.
        endcase
    end

// Note: You can't do any operations here if not will have multiple assignments errors
// To initialize values before entering a state such as resetN, need create a state
// just for initialization.
// Change State
    always@(posedge CLOCK_27 or posedge reset)
    begin
        if (reset) // reset = ~KEY[2]
        begin
            currState <= sReset;
        end
        else                    currState <= nextState;
    end

//-------------------------------------------------------------
// FSM 3 : Drawing the white square and the obstacle boxes
//-------------------------------------------------------------
wire writeEn;
```

```
wire [8:0]x;
wire [7:0]y;
wire [8:0]y2;
assign y2 = {1'b0,y};

wire [8:0]wirecx, wirecy;

assign wirecx = CenterX;
assign wirecy = CenterY;

wire [15:0]colorout;

// Located in sketch.v
sketch sk123(CLOCK_27, ~KEY[1],x, y, colorout, writeEn, wirecx, wirecy);


endmodule // End of main function
//---------------------------------------------------------------------
// Pulse to increase x position. Speed is 1 pixel/pulsetime
//---------------------------------------------------------------------
module tempclock(clock, reset, pulse);
input clock, reset;
output pulse;
reg [25:0]count = {26{1'b0}};
assign pulse = (count == 27'b110010110111001101010000000);

always@(posedge clock, negedge reset)
begin
if(!reset)
        count<={26{1'b0}};
else
        count <= pulse ? 0 : count+1'b1;
end
endmodule


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

Appendix B.2: GameObjManip.v

```
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// Responsible for dealing with coords of object (10 objects)

module GameObjManip(CLOCK_50, startgame, gameover, objCoordcomb, imagedone);

parameter[2:0] ST_IDLE = 0, ST_INITIALIZE = 1, ST_GAME = 2, ST_WAIT = 3,
ST_GAME2 = 4;
```

```
input startgame;
input gameover;
input CLOCK_50;
output [179:0]objCoordcomb;
output imagedone;

assign objCoordcomb =
{objCoord[9],objCoord[8],objCoord[7],objCoord[6],objCoord[5],objCoord[4],objCoord[
3],objCoord[2],objCoord[1],objCoord[0]};

reg [2:0]cstate, nstate;
reg [3:0]level;
reg [17:0]objCoord[9:0];                                    //10 objects,
16:9 is y, 8:0 is x, 17 is enable//10 objects, 16:9 is y, 8:0 is x, 17 is enable
wire objdone;
integer i;
reg [5:0]counter;
reg [3:0]currpos;
reg resetSClock;
reg firstw;
reg donewait;

wire objclock;
reg imagedone, gogame, gogame2;

//y random gen

reg randReq;
wire [7:0]randNum;

reg[28:0]y_rand = 29'h55555555;
wire seed_low_bit, y_low_bit;
assign y_low_bit=y_rand[26] ^ y_rand[28];

always@(posedge CLOCK_50)
begin
y_rand<={y_rand[27:0], y_low_bit};
end

assign randNum = y_rand % 240;

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

assign objdone =
objCoord[9][17]|objCoord[8][17]|objCoord[7][17]|objCoord[6][17]|objCoord[5][17]|objC
```

```
oord[4][17]|objCoord[3][17]|objCoord[2][17]|objCoord[1][17]|objCoord[0][17]; //Active
low

objspeed obj1(CLOCK_50, resetSClock, objclock);

always@(*)
begin

	case(cstate)
			ST_IDLE:
				begin
					if(startgame)
						nstate = ST_INITIALIZE;
					else
						nstate = ST_IDLE;
				end
			ST_INITIALIZE:
				begin
					nstate = ST_GAME;
				end
			ST_GAME:
				begin
					if(!objdone)
						nstate=ST_WAIT;
					else
						if(gogame2)
						nstate=ST_GAME2;
						else
						nstate=ST_GAME;
				end
			ST_GAME2:
				begin
					if(gogame)
					nstate=ST_GAME;
					else
					nstate=ST_GAME2;
				end
			ST_WAIT:
				begin
					if(donewait)
						if(level==10)
						nstate=ST_IDLE;
						else
						nstate=ST_INITIALIZE;
					else
						nstate=ST_WAIT;
```

```
                        end
                default:
                        begin
                        nstate = ST_IDLE;
                        end
                endcase

end

always@(posedge CLOCK_50)
begin
        if(gameover)
                cstate <= ST_IDLE;
        else
                cstate <= nstate;
end

always@(posedge CLOCK_50)
begin

                if(cstate == ST_IDLE)
                begin
                        level <= 0;
                end
                else if(cstate == ST_INITIALIZE)
                                begin
                                        for(i=0; i < 10; i = i + 1)
                                        begin
                                                if(i==0)
                                                begin
                                                        objCoord[i][17]<=1'b1;

        objCoord[i][16:9]<=randNum;
                                                        objCoord[i][8:0]<=0;
                                                        randReq <= 1;
                                                end
                                                else
                                                objCoord[i] <= 0;
                                                randReq <= 0;
                                        end
                                        currpos <= 0;
                                        counter <= 0;
                                        resetSClock <= 0;
                                        donewait <= 0;
                                        imagedone <=1;
                                end
```

```
        else if(cstate ==       ST_GAME)
                                begin
                                        resetSClock <= 1;
                                        gogame <= 0;
                                if(objclock)
                                        begin
                                        imagedone <= 0;
                                        gogame2 <= 1;
                                        for(i=0;i<10;i=i+1) // Add to x positions
                                        begin
                                                if(objCoord[i][17])
                                                begin

if(objCoord[i][8:0]<9'b101000000)

objCoord[i]<=objCoord[i]+1;
                                                        else
                                                                objCoord[i][17]<=0;
                                                end
                                        end

                                                if(counter == 19)
                                                begin
                                                        if(currpos<level)
                                                        begin
                                                                currpos <= currpos +1;

objCoord[(currpos+1)][17]<=1;

objCoord[(currpos+1)][16:9]<=randNum;
                                                                randReq<=1;
                                                        end
                                                        else
                                                        begin
                                                        randReq<=0;
                                                        end
                                                counter <= 0;
                                                end
                                                else
                                                        counter <= counter + 1;
                                        end
                                        else
                                                imagedone <= 1;
                                        firstw <= 1;
                                end
        else if(cstate == ST_GAME2)
```

```
                begin
                        gogame2<=0;
                        if(objclock)
                        begin
                                gogame<=1;
                        end
                        imagedone<=1;
                end
        else if(cstate == ST_WAIT)
                        begin
                                if(objclock)
                                begin
                                        if(firstw)
                                        begin
                                                level <= level + 1;
                                                counter <= 0;
                                                firstw <= 0;
                                        end
                                        else
                                        begin
                                                counter <= counter + 1;
                                        end
                                end
                                if(counter == 10)
                                donewait <= 1;
                        end

end

endmodule

//////////////////////////////////////////////////OBJSPEED
// Pulse to increase x position. Speed is 1 pixel/pulsetime
module objspeed(clock, reset, pulse);
input clock, reset;
output pulse;
reg [25:0]count = {26{1'b0}};
assign pulse = (count == 19'b1100101101110011010);//

always@(posedge clock, negedge reset)
begin
if(!reset)
        count<={26{1'b0}};
else
        count <= pulse ? 0 : count+1'b1;
end
```

```
endmodule
```

//

Appendix B.3: sketch.v

//

```
//Responsible for calculating positions/color of all objects & deals with collision detct.

module sketch
	(
		CLOCK_50,				//	Not necessarily 50Mhz, name should be changed. Clock freq is assigned in upper module INPUT

		startgame,				//	startgame
		x,
	// x coord out
		y,
	// y coord out
		color,					// color out
		writeEn,
	//writeEn

		centerX,				//	Camera x in INPUT
		centerY					//	Camera y in INPUT
	);

	input		CLOCK_50;		//	50 MHz
	input startgame;
	input [8:0]centerX;
	input [7:0]centerY;
	output [8:0]x;
	output [7:0]y;
	output [15:0]color;
	output writeEn;

	wire resetn;	//Useless for now.
	assign resetn = 1;

	wire enable;	//Used to link clearscreen to downcount -- after screen is cleared, pulse is given to confirm screen cleared and proceed to print next frame

	wire gameover;		//Reset Not enabled atm
```

```
	wire [179:0]objCoordcomb;   //Register containing info for all obj xy
	wire imagedone;                    //Not used yet, gives pulse after all obj xy
are updated

	GameObjManip gom(CLOCK_50, startgame, gameover, objCoordcomb,
imagedone);    //gameobj module

	// Create the color, x, y and writeEn wires that are inputs to the controller.


	//x
	wire [8:0] x;
	wire [8:0] xReset;
	wire [8:0] xNonReset[10:0];                    //x is for output to vga, xreset is
assigned to x when clearing, xnonreset is assigned to x when printing square
	//

	//y
	wire [7:0] y;
	wire [7:0] yReset;
	wire [7:0] yNonReset[10:0];                    //y same as x
	//


	wire [10:0]enablenext;
	wire [10:0]downCountPlot, arithPlot;
	wire doreset; //different enable triggers

	wire [3:0] xcount[10:0];
	wire [3:0] ycount[10:0];                                    //counters for drawing
square
	assign writeEn = (arithPlot[0]&downCountPlot[0]) | ~doreset
|(arithPlot[1]&downCountPlot[1])|(arithPlot[2]&downCountPlot[2])|(arithPlot[3]&down
CountPlot[3])|(arithPlot[4]&downCountPlot[4])|(arithPlot[5]&downCountPlot[5])|(arithP
lot[6]&downCountPlot[6])|(arithPlot[7]&downCountPlot[7])|(arithPlot[8]&downCountPl
ot[8])|(arithPlot[9]&downCountPlot[9])|(arithPlot[10]&downCountPlot[10]);
	reg [15:0]colorReg;                                                    //Entire
block is for implementation of animation



	integer i;
```

```
	wire pulse;
	//Pulse at 0.1s
	reg tempimage;

	wire [4:0]gameoverw;
	//CollisionDETCT gameover
	assign gameover = (gameoverw==4'b0000) ? 0 : 1;

	objspeed Hz601(CLOCK_50, resetn, pulse);//Pulse generator
	ColDETCT C1(CLOCK_50, imagedone, centerX, centerY, objCoordcomb,
gameoverw);   //CollisionDTCT module, feed it camera x y



	//Deals with drawing 10 objects, add remove as necessary

	downcounter25  d1(CLOCK_50, doreset, enable, downCountPlot[0], xcount[0],
ycount[0], enablenext[0], objCoordcomb[17]);
	xyarith                x1(xcount[0], ycount[0], objCoordcomb[8:0],
objCoordcomb[16:9], xNonReset[0], yNonReset[0], arithPlot[0]);

	downcounter25  d2(CLOCK_50, 1, enablenext[0], downCountPlot[1], xcount[1],
ycount[1], enablenext[1], objCoordcomb[35]);
	xyarith                x2(xcount[1], ycount[1], objCoordcomb[26:18],
objCoordcomb[34:27], xNonReset[1], yNonReset[1], arithPlot[1]);

	downcounter25  d3(CLOCK_50, 1, enablenext[1], downCountPlot[2], xcount[2],
ycount[2], enablenext[2], objCoordcomb[53]);
	xyarith                x3(xcount[2], ycount[2], objCoordcomb[44:36],
objCoordcomb[52:45], xNonReset[2], yNonReset[2], arithPlot[2]);

	downcounter25  d4(CLOCK_50, 1, enablenext[2], downCountPlot[3], xcount[3],
ycount[3], enablenext[3], objCoordcomb[71]);
	xyarith                x4(xcount[3], ycount[3], objCoordcomb[62:54],
objCoordcomb[70:63], xNonReset[3], yNonReset[3], arithPlot[3]);

	downcounter25  d5(CLOCK_50, 1, enablenext[3], downCountPlot[4], xcount[4],
ycount[4], enablenext[4], objCoordcomb[89]);
	xyarith                x5(xcount[4], ycount[4], objCoordcomb[80:72],
objCoordcomb[88:81], xNonReset[4], yNonReset[4], arithPlot[4]);

	downcounter25  d6(CLOCK_50, 1, enablenext[4], downCountPlot[5], xcount[5],
ycount[5], enablenext[5], objCoordcomb[107]);
	xyarith                x6(xcount[5], ycount[5], objCoordcomb[98:90],
objCoordcomb[106:99], xNonReset[5], yNonReset[5], arithPlot[5]);
```

```
	downcounter25  d7(CLOCK_50, 1, enablenext[5], downCountPlot[6], xcount[6], ycount[6], enablenext[6], objCoordcomb[125]);
	xyarith			x7(xcount[6], ycount[6], objCoordcomb[116:108], objCoordcomb[124:117], xNonReset[6], yNonReset[6], arithPlot[6]);

	downcounter25  d8(CLOCK_50, 1, enablenext[6], downCountPlot[7], xcount[7], ycount[7], enablenext[7], objCoordcomb[143]);
	xyarith			x8(xcount[7], ycount[7], objCoordcomb[134:126], objCoordcomb[142:135], xNonReset[7], yNonReset[7], arithPlot[7]);

	downcounter25  d9(CLOCK_50, 1, enablenext[7], downCountPlot[8], xcount[8], ycount[8], enablenext[8], objCoordcomb[161]);
	xyarith			x9(xcount[8], ycount[8], objCoordcomb[152:144], objCoordcomb[160:153], xNonReset[8], yNonReset[8], arithPlot[8]);

	downcounter25  d10(CLOCK_50, 1, enablenext[8], downCountPlot[9], xcount[9], ycount[9], enablenext[9], objCoordcomb[179]);
	xyarith			x10(xcount[9], ycount[9], objCoordcomb[170:162], objCoordcomb[178:171], xNonReset[9], yNonReset[9], arithPlot[9]);

	downcounter25  d11(CLOCK_50, ~downCountPlot[9], enablenext[9], downCountPlot[10], xcount[10], ycount[10], enablenext[10], 1);
	xyarith			x11(xcount[10], ycount[10], centerX, centerY, xNonReset[10], yNonReset[10], arithPlot[10]);

	//Clear screen module, is triggered first

	clearscreen		c1(CLOCK_50, imagedone, xReset, yReset, doreset, enable);	//3 modules for drawing square/clear screen

	//Color assignments
	always@(posedge CLOCK_50)
	begin
		if(!doreset)
			colorReg <= 16'h000F;
		else
			begin
			if(downCountPlot[0])
			colorReg <= 16'h00FF;
			else if(downCountPlot[1])
			colorReg <= 16'h0F0F;
			else if(downCountPlot[2])
			colorReg <= 16'hF00F;
			else if(downCountPlot[3])
			colorReg <= 16'h0FFF;
			else if(downCountPlot[4])
```

```
                        colorReg <= 16'hF0FF;
                        else if(downCountPlot[5])
                        colorReg <= 16'hFF0F;
                        else if(downCountPlot[6])
                        colorReg <= 16'h735F;
                        else if(downCountPlot[7])
                        colorReg <= 16'h892F;
                        else if(downCountPlot[8])
                        colorReg <= 16'h937F;
                        else if(downCountPlot[9])
                        colorReg <= 16'h283F;
                        else if(downCountPlot[10])
                        colorReg <= 16'hFFFF;
                        else
                        colorReg <= 16'h000F;
                        end
        end     //Select between clearing color (black) or square color

        //Use reset x coord or square x coord/Color set
        assign color = colorReg;
        assign x = {9{(~doreset)}}&xReset |
((xNonReset[10]&{9{downCountPlot[10]}})|{9{(doreset)}}&((xNonReset[0]&{9{down
CountPlot[0]}})|(xNonReset[1]&{9{downCountPlot[1]}})|(xNonReset[2]&{9{downCou
ntPlot[2]}})|(xNonReset[3]&{9{downCountPlot[3]}})|(xNonReset[4]&{9{downCountPl
ot[4]}})|(xNonReset[5]&{9{downCountPlot[5]}})|(xNonReset[6]&{9{downCountPlot[6
]}})|(xNonReset[7]&{9{downCountPlot[7]}})|(xNonReset[8]&{9{downCountPlot[8]}})
|(xNonReset[9]&{9{downCountPlot[9]}})));
        assign y = {8{(~doreset)}}&yReset |
((yNonReset[10]&{8{downCountPlot[10]}})|{8{(doreset)}}&((yNonReset[0]&{8{down
CountPlot[0]}})|(yNonReset[1]&{8{downCountPlot[1]}})|(yNonReset[2]&{8{downCou
ntPlot[2]}})|(yNonReset[3]&{8{downCountPlot[3]}})|(yNonReset[4]&{8{downCountPl
ot[4]}})|(yNonReset[5]&{8{downCountPlot[5]}})|(yNonReset[6]&{8{downCountPlot[6
]}})|(yNonReset[7]&{8{downCountPlot[7]}})|(yNonReset[8]&{8{downCountPlot[8]}})
|(yNonReset[9]&{8{downCountPlot[9]}})));



endmodule


////////////////////////////////////////////////////////////////////////////////////
//Downcounter                   Starts when enabled, plot = 1, xcount and ycount
increment towards 5x5 then ends
```

```
module downcounter25(clk, resetn, enable, plot, xcount, ycount, enablenext, nodraw);
input clk, resetn, enable, nodraw;
output plot, enablenext;
output [3:0]xcount, ycount;

reg [1:0]cstate, nstate;
reg write, write2, enablenext;
reg [3:0]xreg, yreg;

reg pulseextender = 0;

parameter[1:0] ST_IDLE = 0, ST_DOCOUNT = 1, ST_COUNTING = 2;

always@(*)
begin
	case(cstate)
		ST_IDLE:
			begin
				write = 0;
				if(enable)
					nstate = ST_DOCOUNT;
				else
				begin
					nstate = ST_IDLE;
				end
			end
		ST_DOCOUNT:
			begin
				write = 1;
				nstate = ST_COUNTING;
			end
		ST_COUNTING:
			begin
			if(xreg==15)
				begin
					if(yreg==15)
					begin
					nstate = ST_IDLE;
					end
					else
					begin
						nstate = ST_COUNTING;
					end
				end
			else
				begin
```

```
                                nstate = ST_COUNTING;
                        end
                end
        default:
                begin
                        nstate = ST_IDLE;
                        write = 0;
                end
        endcase
end

always@(posedge clk)
begin
        if(!resetn)
        begin
                cstate <= ST_IDLE;
        end
        else
                cstate <= nstate;
end

always@(posedge clk)
begin
        if(cstate == ST_IDLE)
        begin
                if(pulseextender)
                begin
                        pulseextender <= 0;
                        write2 <= 1;
                end
                else
                begin
                        enablenext <= 0;
                        write2 <= 1;
                end
        end
        else if(cstate == ST_DOCOUNT)
        begin
                xreg <= 0;
                yreg <= 0;
        end
        else if(cstate == ST_COUNTING)
        begin
                if(xreg == 15)
                begin
                        yreg <= yreg + 1;
```

```
                        xreg <= 0;
                        if(yreg==15)
                        begin
                        write2<=0;
                        enablenext <= 1;
                        pulseextender <= 1;
                        end
                end
                else
                begin
                        xreg <= xreg +1;
                end
        end
end


assign xcount = xreg;
assign ycount = yreg;
assign plot = write&write2&nodraw;

endmodule

//////////////////////////////////////////////////////////////////////////////////
//xyarith                               Adds current xcoord ycoord to the xycounters,
draws square


module xyarith(xadd, yadd, xin, yin, xout, yout, write);
input [3:0]xadd, yadd;
input [8:0]xin;
input [7:0]yin;
output [8:0]xout;
output [7:0]yout;
output write;

reg [8:0]xreg;
reg [7:0]yreg;
reg writetemp = 1;

always@(*)
begin
        xreg = xin + xadd;
        yreg = yin + yadd;
end

assign xout = xreg;
```

```
assign yout = yreg;
assign write = writetemp;

endmodule

////////////////////////////////////////////////////////////////////////////////////////
//clearscreen                    Clears the screen. Sends out enableother pulse at the
end to indicate finished



module clearscreen(clk, resetn, xout, yout, doreset, enableother);
input resetn, clk;
output [8:0]xout;
output [7:0]yout;
output doreset;
output enableother;
reg eother;

reg [1:0]cstate, nstate;
reg doresetreg, write2;
reg [8:0]xreg;
reg [7:0]yreg;

reg pulseextender = 0;

parameter[1:0] ST_IDLE = 0, ST_DOCOUNT = 1, ST_COUNTING = 2;

always@(*)
begin
        case(cstate)
                ST_IDLE:
                        begin
                                doresetreg = 1;
                                if(!resetn)
                                        nstate = ST_DOCOUNT;
                                else
                                begin
                                        nstate = ST_IDLE;
                                end
                        end
                ST_DOCOUNT:
                        begin
                                doresetreg = 0;
                                nstate = ST_COUNTING;
                        end
```

```
                ST_COUNTING:
                        begin
                        if(xreg==9'b100111111)
                                begin
                                        if(yreg==8'b11101111)
                                        begin
                                        nstate = ST_IDLE;
                                        end
                                        else
                                        begin
                                                nstate = ST_COUNTING;
                                        end
                                end
                        else
                                begin
                                        nstate = ST_COUNTING;
                                end
                        end
                default:
                        begin
                                nstate = ST_IDLE;
                                doresetreg = 1;
                        end
                endcase
end

always@(posedge clk)
begin
        cstate <= nstate;
end

always@(posedge clk)
begin
        if(cstate == ST_IDLE)
        begin
        if(pulseextender)
                pulseextender <= 0;
        else
                eother <= 0;
        end
        else if(cstate == ST_DOCOUNT)
        begin
                eother <= 0;
                xreg <= 0;
                yreg <= 0;
                write2<=0;
```

```
        end
        else if(cstate == ST_COUNTING)
        begin
                if(xreg == 9'b100111111)
                begin
                        yreg <= yreg + 1;
                        xreg <= 0;
                end
                else
                begin
                        xreg <= xreg +1;
                end
                if(xreg == 9'b100111111)
                begin
                        if(yreg == 8'b11101111)
                        begin
                        write2<=1;
                        eother <= 1;
                        pulseextender <= 1;
                        end
                end
        end
end


assign xout = xreg;
assign yout = yreg;
assign doreset = doresetreg&write2;
assign enableother = eother;

endmodule

//////////////////////////////////////////////////OBJSPEED
// Pulse to increase x position. Speed is 1 pixel/pulsetime
module Hz60(clock, reset, pulse);
input clock, reset;
output pulse;
reg [25:0]count = {26{1'b0}};
assign pulse = (count == 1'b1);

always@(posedge clock, negedge reset)
begin
if(!reset)
        count<={26{1'b0}};
else
        count <= pulse ? 0 : count+1'b1;
```

```
end

endmodule
//////////////////////////////////////////////////////////

module ColDETCT(clk, imagedone, xobj, yobj, xyreg, gameoverw);
input imagedone;
input clk;
input [8:0]xobj;
input [7:0]yobj;
input [89:0]xyreg;
output [4:0]gameoverw;
integer i;

reg [4:0]gameoverw;

always@(posedge clk)
begin fe:

if(~(gameoverw==4'b0000))
        gameoverw<=0;
else
if(imagedone)
for(i=0;i<5;i=i+1)
begin
if(xyreg[i*18+17])
if(xyreg[i*18+:9]>=xobj)
        begin
        if(xyreg[i*18+:9]-xobj<16)
                begin
                        if(xyreg[(i*18+9)+:8]>=yobj)
                                if(xyreg[(i*18+9)+:8]-yobj<16)
                                        gameoverw[i]<=1;
                                else
                                        gameoverw[i]<=0;
                        else
                                if(yobj-xyreg[(i*18+9)+:8]<16)
                                        gameoverw[i]<=1;
                                else
                                        gameoverw[i]<=0;
                end
        else
                gameoverw[i]<=0;
        end

else
```

```
        begin
        if(xobj-xyreg[i*18+:9]<16)
                begin
                        if(xyreg[(i*18+9)+:8]>=yobj)
                                if(xyreg[(i*18+9)+:8]-yobj<16)
                                        gameoverw[i]<=1;
                                else
                                        gameoverw[i]<=0;
                        else
                                if(yobj-xyreg[(i*18+9)+:8]<16)
                                        gameoverw[i]<=1;
                                else
                                        gameoverw[i]<=0;
                end
        else
                gameoverw[i]<=0;
        end

end

end

endmodule
```