An nVGA Controller

for Integrating Graphics into Microcontroller Projects

William Koven
Drew Macrae

December 11, 2009

Abstract

We designed and demonstrated a system to allow a Spartan 3 FPGA to command a range of drawing operations on a monitor connected through a VGA cable. The Graphics Controller can draw single dots or clear the whole screen, and it can use 4 different points on the color-depth resolution tradeoff. The demo seeks to emulate an Etch-A-Sketch, but other applications could now be designed quickly, using appropriate serial commands to color individual dots on the screen.

Introduction

The considerable effort required to use a PIC or an FPGA to perform even rudimentary control of a VGA monitor motivated us to construct a model of a VGA controller. Traditional VGA controllers use a 256KB memory to store a set of images and allow a computer to update the screen continuously without requiring the Central Processing Unit's constant attention. We set out to produce and demonstrate a controller that facilitated this kind of management of a monitor and would hopefully be useable in the future by anyone interested in using a microcontroller to control a monitor. Over the course of this project we came across a system much like what we set out to build. It can be found at
http://www.sparkfun.com/commerce/product_info.php?products_id=8541
It has a well defined instruction set for serial command of writes, and we used a similar serial instruction set to assure practicality and hopefully expandability. Its instructions are constrained to using 8bit words, so there is a fair amount of wasted signaling, but the design certainly simplifies the hardware required for reading in serial instructions.

Without an external memory, the Spartan 3 on the Harrisboard is capable of storing about 1/8th the data used in a traditional VGA system. This led us to reduce the resolution significantly, and Table 1 shows the required Resolution-ColorDepth tradeoff to ensure the design could be implemented with the available memory.

Table 1: The limits of the Spartan 3 frame buffer enforce a Resolution-ColorDepth tradeoff.

Resolution	Color Depth
320x480	1 Bit (Black and White)
320x240	2 Bit (4 levels of gray)
160x240	4 Bit (16 Colors)
160x120	8 Bit (256 Colors)

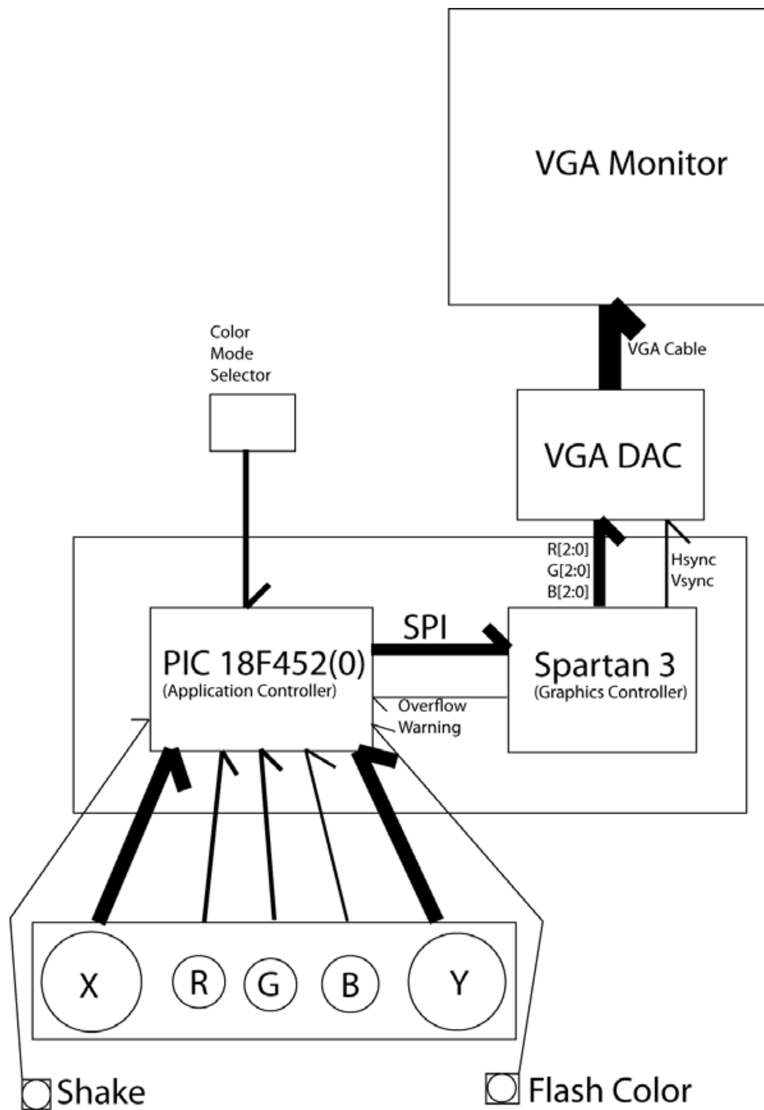


Figure 1: A schematic of signaling and devices in our Etch-A-Sketch System. Signal line width indicates bus width.

In order to demonstrate the power of our graphics controller we decided to implement an Etch-A-Sketch with four different color modes, so we could show that an FPGA system can achieve either high resolution display operations or a decent color depth. Figure 1 shows the main components for this system, where the PIC contains the application layer, and sends serial instructions to the FPGA which

performs the draws and maintains the screen. In this way the FPGA is acting like the graphics card on a computer, while the PIC is acting as a CPU.

Timing

The Harrisboard's clock operates the PIC at 20 MHz. VGA requires something closer to 25MHz, which we produced from a 20MHz signal using the FPGA's Digital Clock Manager. The signaling from the PIC must therefore be asynchronously passed to the FPGA. The bus is mediated by a parallelizer that reads entire bytes over the SPI and outputs them over a parallel bus asserting a valid byte while the output is sure to be stable.

Signal Specifications

We imagine that our model VGA system is more interesting if other designers can use it as a general display controller. The commands for it are included below. They should be sent over the PIC SPI.

Draw Instructions

1. Dot Draw

Drawing a single dot is performed by sending the following string of Bytes:
0x50
X position (16 Bits)
Y position (16 Bits)
Color (8 Bits)
Our design includes a Dot Draw module capable of drawing 10 dots per frame. The X and Y position must be valid for the current color mode to allow for proper tracking of which dots have been written.
When dot draws are approaching overflow, an external signal is raised to indicate to the PIC that a new dot draw will prevent a previously commanded draw from occurring.

2. Clear Screen

Clearing the screen is performed by sending the following string of Bytes.
0x45
Color (8 Bits)
The screen can be cleared once every two frames.

3. Change Color Mode

The color mode can be changed by sending the following string of Bytes.
0x59
Color Mode (8 bits)

Supported color modes are:
0x00: 1 black and white achieves 320x480 resolution
0x01: 2 bit grayscale at a 320x240 resolution (square pixel groups)

0x02: 4 bit color at a 160x240 resolution 1bR,2bG,1bB
0x03: 8 bit color at a 160x120 resolution 3bR,3bG,2bB (square pixel groups)

Changing the color mode doesn't clear the screen automatically

4. No-Op

The Command Interpreter will only read bytes from the SPI receiver while the next byte is being sent. Therefore, after the last instruction is sent over the SPI a No-Op should be sent to guarantee that all bytes of an instruction are properly interpreted. A No-Op is executed by sending the byte
0x00.

Overflow Warning Flag

The FPGA uses a binary flag to signal to the PIC that it is not prepared for a command. The flag is raised when the Dot Draw module is filled and undrawn dots would be overwritten. These are passed back to the PIC over port D pin 7 so that the user and the designer can observe this behavior.

New Hardware

The Hardware for this project is similar to that of the CrazyGame as implemented on the PIC by Dayringer and Weiner. There are several elements that we believe we've improved:

VGA DAC for the Spartan 3

The VGA standard uses a 0-0.7V analog signal on three different conductors to indicate each of the Red, Green and Blue color values to be displayed. We used a simple set of 4 resistors to act as a Digital to Analog Converter to allow multiple digital outputs to control each of these analog signals.

Our VGA controller used 8 bits to represent a color at the greatest color depth, so that red and green will have 3 bits specifying 8 possible levels, and blue will has two bits specifying four possible levels. For the sake of simplicity, the circuit to achieve the blue output is the same as red and green, but the three pins only achieve four different colors.

Computing the set of resistors to achieve the desired output impedance is a reasonably simple bit of circuit design. We start by assuming a value for the output impedance of the FPGA pins. We guessed 30Ω so that ideally we would have the output being run into resistors of at least ten times that value such that the load of the resistor wouldn't affect the output significantly.

We built our DAC by using four different resistors in the configuration shown in Figure 4. The first three have a resistance of R_1, $2R_1$ and $4R_1$ so that each pin will have twice the effect on the current into the load resistor, R_2. Our system must then satisfy two constraints. First, the output must be 0.7 when all the pins are pulled High(3.3V). Second, the output impedance should match that of the VGA cable, (75Ω). The output impedance is the parallel addition of R_1, $2R_1$, $4R_1$ and R_2, and the

output voltage is effectively the result of dividing the input voltage across the parallel addition of R_1, $2R_1$ and $4R_1$ and R_2. By solving for these resistances, we get a set of resister values for the DAC. Then we simply subtract the resistance already present in the FPGA output pins to achieve the final circuit. We built and tested its ability to produce the desired output before connecting it to a monitor through the VGA port.

Figure 2 shows the completed board, which slides into a set of female header pins placed on the edge of the Harrisboard. It occupies a total of 12 pins of the Harrisboard's FPGA IOs.

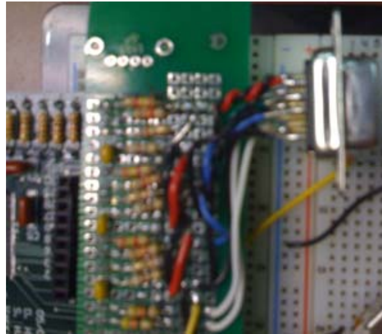


Figure 2: A photo of the completed DAC. It sits on a set of female headers that have replaced the male headers on the edge of the Harrisboard.

User Interface

The interface for this circuit is quite simple. We connected a set of five potentiometers from ground to 3.3V in order to vary output values across the entire PIC ADC range. We then connected two buttons and four switches so that they have the appropriate pull-ups or pull-downs to allow them to send 3.3V or 0V to the PIC input pins. When we became frustrated by potentiometers becoming disconnected, we built a panel to ensure that they are held in place. To improve the grip and appearance of our UI we made knobs of different sizes to control color and position. Figure 3 shows the layout and the labeling of the knobs.

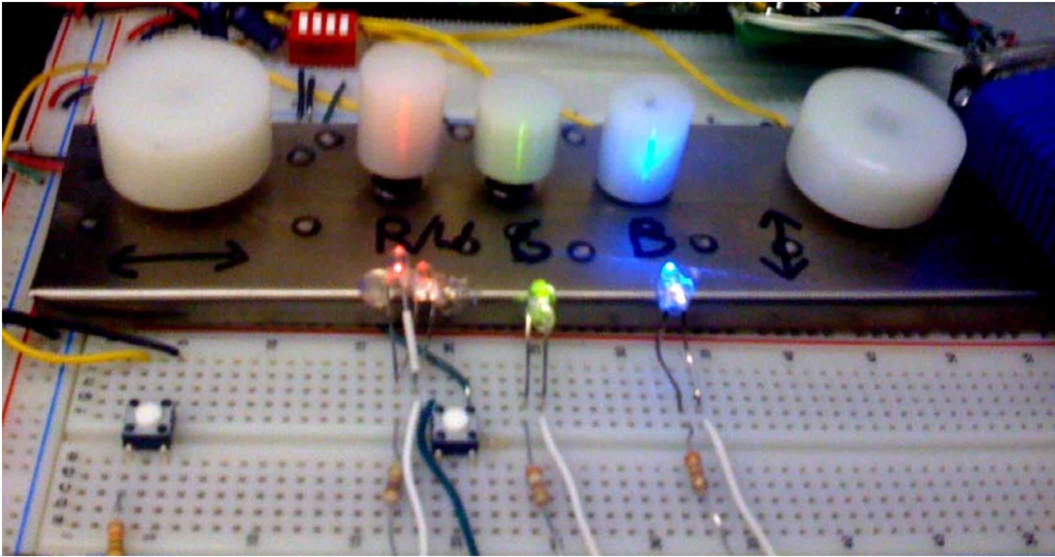


Figure 3: The user input interface for the Etch-A-Sketch demo, the shake button is on the left, the flashing color button is on the right.

Schematic of constructed circuits

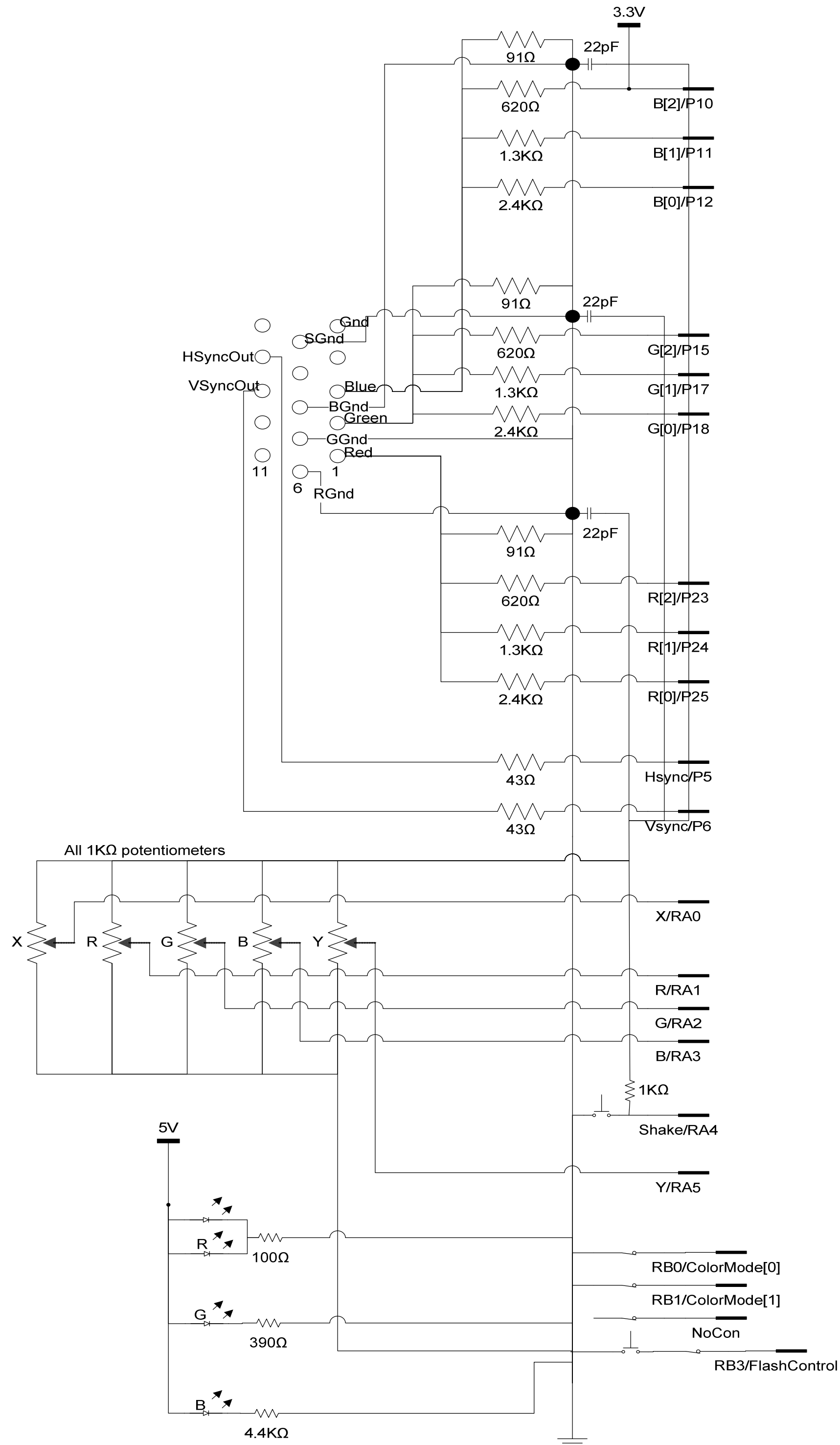


Figure 4: A Schematic for the breadboarded circuit

Microcontroller design

The PIC microcontroller manages the FPGA display controller and sends commands to perform and facilitate single pixel draws and full screen clears. By emulating an Etch-A-Sketch, the PIC demonstrates the utility of a well defined frame buffer and its ability to store onscreen information until it is overwritten.

Controls

We use the PIC ADC on port A to interface with the 5 potentiometers used to control the system. A bank of DIP switches selects graphics modes and a pushbutton "shakes" the system. We configured a second button to provide the user with a flashing color draw. The PIC interprets the motion of the knobs and switching of the digital pins and commands the FPGA to draw the appropriate updates to the screen.

Serial interface

The SPI module on the PIC sends data asynchronously to the FPGA. The data is formatted as dot draw, shake and color mode instructions as shown in the section on signals in the system overview.

Mediating User Input

User input will be collected by the PIC using PORTA's ADC capabilities. The PIC will perform periodic analog to digital conversions on 5 channels and then send the results to the FPGA Graphics controller in the form of a dot draw. It will also send color mode changes and command the clearing of the screen with serial instructions.

Interfacing with the FPGA

The FPGA is where the brunt of the merit for this project lies. The PIC uses the SPI port in master mode to clock data out to the FPGA at up to 5MbPS. This corresponds to a few full screens' worth of single pixel draws per second.

Behavior

The knobs in the order they are interpreted are: X Position, Red Value, Green Value, Blue Value, and Y Position. Each knob is configured so that the full left is a 0 and full right is the full range value so it will nicely map to the space. In addition there is one pushbutton that will clear the screen and 4 dipswitches to configure draw modes. The available draw modes are described in the signals portion of the introduction.

PIC Helper functions

A designer who wishes to use the FPGA designs developed here would likely want to use the following helper functions to send commands to their graphical controller.

WriteChar(char Data)

This function manages the Serial Peripheral Interface with the FPGA. It can be used to send draw commands to the FPGA in a simple project that wishes to gain video capabilities.

SPIinit()

This function initializes the SPI to communicate with the FPGA it should be called before WriteChar, and only needs to be called once.

FPGA Design

The FPGA consists of an SPI Receiver, Command Interpreter, Memory Controller, VGA Controller, and graphics accelerator units which currently include Dot Draw and Solid Draw. The SPI Receiver receives serial data from the PIC and outputs it in parallel to the Command Interpreter, which in turn issues instructions to the graphics accelerator units. The Memory Controller is responsible for managing the screen buffer and generating the X and Y scan signals as well as arbitrating between different graphics units. The VGA Controller is responsible for drawing black during porches and generating HSync and VSync signals.

SPI Receiver

The SPI Receiver is designed to accept 8 bit words over a pair of pins connected to the PIC. On every rising edge of the SPI clock, SPI data is read into a set of registers, and every time 8 bits have been read in, they are transmitted to a register that makes them available in parallel form to the 25 MHz portion of the circuit. While bits 2-4 are being read in, the SPI Receiver raises a valid flag to indicate that the parallel bits are valid and aren't subject to imminent change.

Command Interpreter

The Command Interpreter looks for the rising edge of the signal that indicates the parallel data is valid, and whenever new parallel data becomes valid, it uses its values to drive a state machine, setting up a set of control signals for the draw mode that is currently being invoked and raising a flag to indicate that a draw is to start.

Memory controller

The memory controller generates X and Y coordinates incrementing each clock cycle, or 25 Hz, and outputs both the current X, Y, and Color signals as well as the next X and Y coordinates that will be drawn. The controller uses a simple dual ported 8 bit x 32,768 memory. Whenever the memory controller moves to a new word in memory, it writes the displayed values to the most recently displayed region.

The memory is always addressed using the 7 most significant Y coordinate bits, or Y[8:2], and the 8 most significant X coordinate bits, or X[9:2] where X and Y refer to the position of the pixel on the monitor that will be drawn. In 8 bit color mode the color information is stored in the entire byte, and 16 pixels all use that

byte to determine their color. In 4 bit color mode the color is stored in either the top or bottom nibble addressed according to Y[1], here groups of 8 pixels share colors. In 2 bit color mode the color information is stored in two bit segments of the memory byte according to Y[1] and X[1], and four pixels share a color. Finally, in 1 bit color mode each bit of the memory byte holds color information addressed according to Y[1:0] and X[1] so that color is shared by only two pixels.

There are three stages of pipeline in the memory controller, one before the memory, one after the memory, and a final stage holding all the current X, Y, and Color and memory information. The three stages are necessary because there is a one cycle delay between the address going into the memory and the output value becoming valid. The first cycle X and Y are advertised to all the graphics acceleration modules and using combinational logic any module that wishes to draw raises a request flag. The address for the memory location pertaining to X and Y also goes into the memory on the first cycle. On the second cycle X and Y advance to XMemOut and YMemOut while the memory address advances to MemAddrOut and MemOut becomes the valid memory byte pertaining to XMemOut and YMemOut. A single graphics unit that requested a draw is then enabled and by combinational logic a valid color is put on the ColorBus from the graphics unit to the memory unit. On the third cycle Enable advances to EnableNow while XMemOut and YMemOut advance to XNow and YNow, and MemAddrOut becomes MemAddrNow. If MemAddrOut and MemAddrNow are pointing to different memory locations, then the current modified memory byte is written back into memory at MemAddrNow and MemNow gets the value from MemOut. Otherwise, the current modified memory byte is stored back into MemNow. The modified memory byte as well as the current color are generated using combinational logic involving the current memory byte, the current BusColor, BusColowNow, and the color mode. See Appendix D for a timing diagram.

VGA controller

The VGA controller takes in the current X scan and Y scan positions as well as the color, and either outputs the color if the current scan position is in the viewable screen area or outputs black. The VGA controller also outputs the HSync and VSync signals.

Acceleration modules

The Dot Draw module takes in X Coordinates, Y Coordinates, and a Color value. Dot Draw stores the data in a collection of shift registers that can hold up to 10 different dots. When the screen reaches the coordinate of any of the dots, a draw is requested and the dot is marked as written. If the modules ninth dot is currently unwritten, it raises a buffer full flag to signal to other modules that it cannot hold any more dots at the moment.

The Solid draw uses enabled registers to hold a color value, and once it is told to start the color register is un-enabled. Solid Draw then requests to draw its color for

at least a full frame by looking for a negative edge, positive edge, and then another negative edge of VSync before re-enabling the color register.

Results

The system achieves the target resolutions and color depths. It allows for the expedient design of microcontroller graphics subject to the constraint of the size of the frame buffer.

References

4dSystems uVGA Picasso
http://www.4dsystems.com.au/prod.php?id=15

"MicroToys Guide: VGA Monitor" D. Rinzler
http://www4.hmc.edu:8001/Engineering/microtoys/VGA/MicroToys%20VGA.pdf

Video Graphics Array
http://en.wikipedia.org/wiki/Video_Graphics_Array

"PIC Crazygame.exe with Wireless Controller" M. Dayringer, M. Weiner
http://www3.hmc.edu/~sharris/class/e155/Projects_2007/CrazyGame.pdf

Extra Parts List

The following components were important for the completion of our project and were not readily available in the lab

- Knobs (manufactured from soft face scrap)
- Red, Green and Blue LEDs (found in VLSI lab component bins)
- Panel (manufactured from tooltray scrap)
- Female, Soldercup VGA connector (ordered from Mouser)
- Protoboard PCB. (obtained from stockroom)

Future Work

This project has tried to facilitate later expansion and extension to ultimately convert it to something that acts as a useful peripheral for integration with a PIC. In the future we intend to do the following:

Web publishing

To save people from reinventing the wheel we'd like to publish a tutorial that explains the design as well as providing it to anyone who sees fit to use it.

Extending Drawing Modes

VGA gives a computer many acceleration modes. Though we only implemented three and only demonstrated two, the usefulness of this system would be improved by extending the number of draw modes by adding useful functions.

Rectangle Draws

We implemented a simple Rectangle drawing engine that draws a rectangle of a specified color. It's not particularly interesting for an Etch-A-Sketch, but many other projects for this class use rectangular graphical elements quite often.

Text Draws

Printing text to a display is perhaps the most useful single drawing task that a display controller can perform. In fact, many displays can only display text. Implementation of a Text Draw mode requires some extension, but the current serial interface should allow printing of text over the serial port with little effort, and once a font set has been programmed on the FPGA and a FSM is programmed to track target position, Text Draws become a reasonable task.

Read Pixel

Certain operations require the reading of the color value at a certain pixel. If the serial interface were made bidirectional, a command could be written to instruct the FPGA to transmit the color at a certain pixel.

Line Draws

Diagonal lines are an interesting problem, and programming a digital system to quickly draw lines is an interesting challenge. This goal is optimistic.

Further Color Modes

The four color modes allow for some adaptability in the uses of the display. Further work could add additional modes to allow the system to emulate some of the functions often used in a modern architecture.

Double Buffering

Animation often requires the use of two or four pieces of video memory, one of which is currently being displayed while the others are being edited. This sacrifices some resolution or color depth to enable some fidelity in the timing of events onscreen, or to allow for the undoing of certain draws.

Color Pallet

256 colors is a lot and often a designer would prefer to use a smaller subset of colors. A color pallet allows the designer to select a subset of these colors that they will use in a display, and to store the color value as a shorter piece of data that points towards the address of the color on a pallet rather than to specify the entire color. For example, this could allow the designer to use any 16 Colors of the 256 possible colors that the DAC can produce while in the 4 bit color mode.

Appendix A PIC Microcontroller code

```
Y:\2009-2010\ESketch\Working_ESketch_5_Dec_2009_1930pm\PICSoftware_6_12_09\main.c

//Andrew Macrae
//VGAPIC
//December 6, 2009
//drewmacrae@gmail.com

#include <p18f452.h>

    //these chars let the user developer ADCON0 quickly
    //select input channel and configure ADCON0
    //32TOSC                                            (0(10))
    //select input chanel ADC(0)                        (000)-(100)
    //set go                                            (0)
    //unimplemented                                     (0)
    //ADON                                              (1)
#define UseAD0 0b10000001
#define UseAD1 0b10001001
#define UseAD2 0b10010001
#define UseAD3 0b10011001
#define UseAD4 0b10100001//this is pin RA5


//not really the height or widths, really the maximum address value
#define CM00W 319
#define CM00H 478//this is large enough that the FIR overestimates by one @ max
//this value tweaked^

#define CM01W 319
#define CM01H 239

#define CM10W 160
//this value tweaked^ it was small enough the FIR underestimates by one at max
#define CM10H 239

#define CM11W 160
#define CM11H 120
//both these values tweaked^ small enough the FIR underestimates by one at max

//gain is skewed 0.5% to avoid low level steady state errors due to rounding
#define IIRGain 50.25//vary gain and satisfy below
#define IIRRem  0.95//1000-Gain = 990 = Rem*1000



int get_AN_V(char channelConf)//uses the ADC to monitor the PIC's output
{
    char i;

    ADCON0=channelConf;//configure the channel using the argument

    for(i=0;i<40;i++)//this will allow the cap to charge
    {}

    //start conversion
    ADCON0bits.GO=1;

    while(ADCON0bits.GO)//wait till done
    {}

    return ADRESH*256+ADRESL;//return an 10 bit reading of analog voltage
}

void writeChar(char output)//make writes a bit easier
{
    SSPCON1bits.WCOL = 0;//set collision bit low
    SSPBUF = output;      //try a write
    while(SSPCON1bits.WCOL == 1)
    {                     //keep trying till it worked
        SSPCON1bits.WCOL = 0;   //set collision bit low
        SSPBUF = output;        //try a write
    }

                                      1
```

```
}

void SPIinit(void)//initialize the SPI for this project
{
    TRISC=0x00;     //set PORTC as outputs
    PORTCbits.RC4 = 1;
    //configure SSPSTAT
    //Sample bit sample at middle                     (0)
    //Transmit at rising edge                         (1)
    //don't care                                      (000000)
    SSPSTAT = 0b01000000;

    //configure SSPCON1
    //no collision                                    (0)
    //don't care                                      (0)
    //enable the ssp                                  (1)
    //clock idles low                                 (0)
    //SPI Master clock = FOSC/4                       (0000)
    SSPCON1 = 0b00100000;
    PORTCbits.RC4 = 0;
}

void ADinit(void) //initialize the ADC for this project
{
    TRISA=0xFF;     //set PORTA as inputs
    //configure ADCON1
    //Right Justify                                   (1)
    //32TOSC                                          ((0)10)
    //unimplemented                                   (00)
    //configure AN 0-4 to be ADCS                     (0010)
    ADCON1=0b10000000;
}

void main(void) {
    long int IIRmx;//x*1000 and filtered to reduce jitter
    long int IIRmy;//y*1000 and filtered to reduce jitter
    long int x;    //temporary variable for x
    long int y;    //temporary variable for y
    long int lastX;//the slew limiter needs to know last output
    long int lastY;

    unsigned int i;//temporary iterator

    unsigned long int width;//the color mode is used to set these
    unsigned long int height;

    char colorMode;//store the color mode to detect changes and send new one

    int R;//read in color values
    int G;
    int B;

    char color;//concatenated color byte

    TRISB=0xFF;     //set PORTB as inputs
    INTCON2bits.RBPU = 0;//pull up port b pins

    TRISD=0xFF;     //set PORTD as inputs

    ADinit();//initialize SFRs to access ADCs
    SPIinit();//initialize SFRs to access SPIs

    while(1) {
        if((PORTB&0b00000011)!=colorMode)
        {//detected a color mode change
            colorMode=(PORTB&0b00000011);

            //update color mode on PIC and FPGA
            writeChar(0x59);       //change color mode
            writeChar(colorMode);  //write colorMode
            if(colorMode==0)//update widths and heights co compute x and y
```

```
    {
        width=CM00W;
        height=CM00H;
    }
    if(colorMode==1)
    {
        width=CM01W;
        height=CM01H;
    }
    if(colorMode==2)
    {
        width=CM10W;
        height=CM10H;
    }
    if(colorMode==3)
    {
        width=CM11W;
        height=CM11H;
    }

    x = get_AN_V(UseAD0);//read x and y
    x =(x*width)>>10;     //then scale to writable range
    y = get_AN_V(UseAD4);
    y =(y*(height))>>10;

    lastY=y;//initialize slew rate limiting filter
    lastX=x;

    IIRmy=y*1000;//initialize the infinite impulse response filter
    IIRmx=x*1000;

    writeChar(0x45);//fill screen with
    writeChar(color);//dialed in color
}

if(!PORTAbits.RA4)//if someone presses the shake button
{
    writeChar(0x45);//fill screen with
    writeChar(color);//dialed in color
}

if(PORTBbits.RB3)//if the flash button isn't held, read in a color
{
    R = (get_AN_V(UseAD1)>>2)&0b11100000;
    G = (get_AN_V(UseAD2)>>5)&0b00011100;
    B = (get_AN_V(UseAD3)>>8)&0b00000011;

    color = R|G|B;//concatenate the colors
}
else
    color++;//otherwise flash the color

x = get_AN_V(UseAD0);//read and scale x and y
x =(x*width)>>10;
y = get_AN_V(UseAD4);
y =(y*(height))>>10;

IIRmx = IIRmx * IIRRem + x * IIRGain;//apply the IIR low pass filter
IIRmy = IIRmy * IIRRem + y * IIRGain;//it has a fixed precision at 1000x

x = IIRmx*0.001;//scale value back to writable range
y = IIRmy*0.001;

if(PORTDbits.RD7==0)//if the overflow flag is low
{
    if(lastY<y)//allow the pixel to move one step
        {lastY++;}
    if(lastY>y)
        {lastY--;}
    if(lastX<x)
        {lastX++;}
```

```
            else if(lastX>x)
                {lastX--;}

            writeChar(0x50);//pixel put instr

            writeChar(lastX>>8);//MSBs X
            writeChar(lastX);   //LSBs

            writeChar(lastY>>8);//MSBs Y
            writeChar(lastY);   //LSBs

            writeChar(color);   //all the colors
        }
    }
}
```

Appendix B Verilog FPGA code

VGAGraphicsManager.v   Fri Dec 11 05:29:36 2009

```
1   `timescale 1ns / 1ps
2   ////////////////////////////////////////////////////////////////////////////////
3   // Designers:     William Koven          and Drew Macrae
4   // Contact:       william.koven@gmail.com     drewmacrae@gmail.com
5   // Create Date:    15:36:14 11/22/2009
6   // Module Name:    VGAGraphicsManager
7   // Project Name:   uPs final project
8   // Target Devices: XILINX SPARTAN XC3S400
9   // Description:    This project leverages the Spartan's high speed parallel nature
10  // to manage acceleration and maintnance of draws to a VGA monitor
11  //
12  // Revision:
13  // Revision 0.01 - File Created
14  // Revision 0.02 - Done flashing screen, beginning to update based on PIC data
15  // Revision 0.03 - Draws, in need of optimization
16  // Revision 0.04 - Full color draw works well. moving on to other modes
17  // Revision 0.05 - December 6, 2009 no known issues
18  ////////////////////////////////////////////////////////////////////////////////
19  module VGAGraphicsManager( input SPIClk,
20                             input SPID,
21                             input Reset,
22                             output Ready,
23                             output VGAClk,
24                             output HSync,       //VGA Outputs
25                             output VSync,
26                             output [2:0] R,
27                             output [2:0] G,
28                             output [2:0] B,
29                             output [7:0] DLED, //Debug LED's
30                             input PICClk,
31                             input IOReset);
32
33      wire CLKFB;//throwaway feedback signal for DCM
34      wire Waiting;
35
36      wire       Valid; //it's safe to read parallelized data from PIC
37      wire [7:0] ParD;  //parallelized data from the PIC
38
39      wire [15:0] XCoordinate; //Current X Coordinate of the cursor
40      wire [15:0] YCoordinate; //Current Y Coordinate of the cursor
41      wire [7:0]  Color;       //Current color of the cursor
42      wire [2:0]  ColorMode;
43      wire        DDGo,CSGo,RDGo;  //Dot Draw Go, Clear Go, Rectangle Draw Go
44
45      wire BuffFull;      //whether the dot draw buffer is full
46
47      wire [15:0] XNext, YNext; // Next X and Y coordinate being drawn to screen
48      wire        DrawDot, DotEnable;  // Dot request and dot enable signals
49      wire        SolidReq, SolidEnable; // Solid draw request and solid enable
50      wire        RectReq, RectEnable;  // Rect draw and request
51      wire [15:0] UpperX, UpperY, LowerX, LowerY;
52      wire [7:0]  BusColor;
53      reg  [3:0]  Counter;
54
55      wire [15:0] XScan, YScan; //The current position of the VGA scan line
56      wire [7:0]  DrawReq;      //The bus holding requests for the ColorBus
57
58
59      wire [7:0]  Enable;       //The bus holding enable lines for the ColorBus
60
61      wire [7:0]  OutColor;     //The color being drawn out to the VGA cont
```

Page 1

```
62          wire writ;
63
64          VGAClockGen vgaclock(PICClk,Reset,VGAClk,CLKFB);//25MHz PixelClock for VGA
65
66          SPIPar spipar(SPIClk,IOReset,SPID,Ready,Valid,ParD);
67
68          //this command interpreter will be a state machine that interprets SPI Data
69          ComInt comint(VGAClk,IOReset,Valid,ParD,Waiting,XCoordinate,YCoordinate,
70                              Color,ColorMode,DDGo,CSGo,RDGo);
71
72          //These give debug information on the LEDs
73          //assign DLED[0]   = DDGo;
74          //assign DLED[1]   = CSGo;
75          //assign DLED[3]   = RDGo;
76          //assign DLED[6:4] = ColorMode;
77
78          //if we want to debug the databus we can monitor it
79          //assign DLED[6:0] = ParD[6:0];
80
81          //if we want to debug position we can monitor the bits that change often
82          //assign DLED[3:0] = XCoordinate[3:0];
83          //assign DLED[6:4] = YCoordinate[2:0];
84
85          //this LED lights when the buffer is full
86          assign DLED[7]   = BuffFull;
87
88          assign SolidColor = Color;
89          assign RectColor  = Color;
90          assign UpperX     = XCoordinate;
91          assign LowerX     = 16'h00a0;
92          assign UpperY     = YCoordinate;
93          assign LowerY     = 16'h0078;
94
95          //this is graphical accelerator draws dots when
96          //commanded by the interpreter
97          DotDraw dotdraw(VGAClk,Reset,XCoordinate,YCoordinate,ColorMode,Color,DDGo,
98                              XNext,YNext,DrawDot,DotEnable,BusColor,BuffFull);
99
100         //this fills the screen when commanded by the interpreter
101         SolidColor solid(VGAClk,Reset,SolidEnable,Color,SolidReq,BusColor,CSGo,VSync);
102
103         // NOTE: Rectange Draw works, however it is currently unused in the project
104         RectangleDraw rect(VGAClk,Reset,UpperX,UpperY,LowerX,LowerY,XNext,YNext,
105                                Color,RectEnable,RDGo,RectReq,RDReady,BusColor);
106
107         assign DrawReq = {5'b0,RectReq,SolidReq,DrawDot};
108         assign {RectEnable,SolidEnable,DotEnable} = Enable[2:0];
109
110         //control the memory and the pointer for the scanline and the peripherals
111         MemCont memcont(VGAClk,Reset,BusColor,ColorMode,DrawReq,XNext,YNext,XScan,
112                             YScan,Enable,OutColor,writ);
113
114         //controls the VGA port based on scanlines from MemCont
115         //and data passed by memory controller
116         VGACont vgacont(VGAClk,Reset,OutColor,XScan,YScan,HSync,VSync,R,G,B,writ);
117
118     endmodule
119
120     module ComInt( input VGAClk, Reset, Valid,
121                    input [7:0] ParD,
122                    output Waiting,
```

```
123                  output reg [15:0] XCoordinate, YCoordinate,
124                  output reg [7:0]  OutColor,
125                  output reg [2:0]  ColorMode,
126                  output reg DDGo,
127                  output reg CSGo,
128                  output reg RDGo);
129         reg [7:0] Color;
130         reg [7:0] State;
131         wire SyncValid;
132         wire SyncValidLast;
133         wire VEdge;
134
135         FlopR vsynchronizer(VGAClk, Reset, Valid, SyncValid);//Syncronize valid bit
136         FlopR vdelay(VGAClk, Reset, SyncValid,SyncValidLast);//delay valid one more
137
138         assign VEdge = SyncValid&~SyncValidLast;   //indicate data is valid and new
139         assign Waiting = State == 8'h00;//Waiting says that writes will be in frame
140
141         always@(*)
142         begin
143           case(ColorMode[1:0])                          //remix and zero fill colors
144              2'b00:OutColor={7'b0,Color[7]};                  //2 color is 0000_000W
145              2'b01:OutColor={6'b0,Color[7:6]};                //4 color is 0000_00WW
146              2'b10:OutColor={4'b0,Color[7],Color[4:3],Color[1]};//16 " "  0000_RGGB
147              2'b11:OutColor=Color;                          //256 color is RRRG_GGBB
148           endcase
149         end
150
151         always@(posedge VGAClk, posedge Reset)
152         begin
153           if(Reset)
154           begin
155              State       <= 8'b0;               //make sure all flops are resettable
156              ColorMode   <= 3'b0;
157              Color       <= 8'b0;
158              XCoordinate <= 16'b0;
159              YCoordinate <= 16'b0;
160           end
161           else
162           begin
163              case(State)  //this is an FSM for reading commands from shift register
164                 8'h00: begin                    //Original state is waiting for data
165
166                    DDGo <= 1'b0;  //clear out either of the bits triggerring a draw
167                    CSGo <= 1'b0;
168                    if(VEdge)                      //if there is new valid data:
169                    begin
170                       case(ParD)
171                          8'h59: State <= 8'h01;  //trigger a color mode change
172                          8'h50: State <= 8'h02;  //trigger a dot draw
173                          8'h45: State <= 8'h07;  //clear or overwrite entire screen
174                          default: State <= 8'h00;//ignore anything else
175                       endcase
176                    end
177                    else
178                       State<=8'h00;              //state only changes if new data
179                 end
180
181                 8'h01:begin                               //ColorMode change state
182
183                    if(VEdge)            //if new data read it in as the color mode
```

```
184                     begin
185                        State     <= 8'h00;
186                        ColorMode <= ParD[2:0];
187                     end
188                     else
189                        State<=8'h01;      //otherwise wait
190                  end
191
192                  8'h02:begin                                //DotDraw first state
193                     if(VEdge)//read first byte of new data as upper x address
194                     begin
195                        State<=8'h03;
196                        XCoordinate[15:8]<=ParD;
197                     end
198                     else
199                        State<=8'h02; //otherwise wait
200                  end
201
202                  8'h03:begin                                //DotDraw second state
203                     if(VEdge)//read second byte of new data as lower x address
204                     begin
205                        State<=8'h04;
206                        XCoordinate[7:0]<=ParD;
207                     end
208                     else
209                        State<=8'h03; //otherwise wait
210                  end
211
212                  8'h04:begin                                //DotDraw third state
213                     if(VEdge)//read third byte of new data as upper y address
214                     begin
215                        State<=8'h05;
216                        YCoordinate[15:8]<=ParD;
217                     end
218                     else
219                        State<=8'h04; //otherwise wait
220                  end
221
222                  8'h05:begin                                //DotDraw fourth state
223                     if(VEdge)//read fourth byte of new data as the lower y address
224                     begin
225                        State<=8'h06;
226                        YCoordinate[7:0]<=ParD;
227                     end
228                     else
229                        State<=8'h05; //otherwise wait
230                  end
231
232                  8'h06:begin                                //DotDraw fifth state
233                     if(VEdge)//read fifth byte as the color
234                     begin
235                        State <= 8'h00;
236                        Color <= ParD;
237                        DDGo  <= 1'b1; //then raise a flag to run a dot draw
238                     end
239                     else
240                        State<=8'h06; //otherwise wait
241                  end
242
243                  8'h07:begin                                //clear screen state
244                     if(VEdge)//read first byte as color and tell clearscreen to go
```

```
245                     begin
246                        State <= 8'h00;
247                        Color <= ParD;
248                        CSGo  <= 1'b1;
249                     end
250                     else
251                        State<=8'h07; //if no new data wait
252                  end
253
254                  default: State<=8'h00;//it shouldnt ever get here
255
256               endcase
257            end
258         end
259      endmodule
260
261      //I store data from the SPI
262      module SPIPar( input SPIClk,Reset,
263                     input SPID,
264                     output Ready,
265                     output Valid,
266                     output [7:0] ParD);
267
268
269                     wire   [7:0] SREGOut;
270                     wire   [2:0] bitCount;
271
272         assign Ready = bitCount == 0;//this raises a flag saying data is in frame
273         assign Valid = (bitCount > 1) & (bitCount < 4);
274         //the above comparisons say data is safe to read
275
276         SReg8R  sreg(SPIClk, Reset, SPID, SREGOut);//collects data from SPI
277
278         //stores parallel data when ready
279         Reg8enR regout(SPIClk,Reset,Ready,SREGOut,ParD);
280
281         //counts 8 clockedges to know when to enable above reg
282         Count3b counter(SPIClk,Reset,bitCount);
283      endmodule
284
285      //Structural resettable flop
286      module FlopR(  input Clk, Reset, D,
287                     output reg Q);
288         always@(posedge Clk, posedge Reset)
289         begin
290            if(Reset)
291               Q<=1'b0;
292            else
293               Q<=D;
294         end
295      endmodule
296
297      //I count to 8
298      module Count3b(input Clk, Reset,
299                     output reg [2:0] Q);
300         always@(posedge Clk, posedge Reset)
301         begin
302            if(Reset)
303               Q<=3'b0;
304            else
305               Q<=Q+1;
```

```
306            end
307         endmodule
308
309         //I store D and shift Q whenever I see a rising edge of Clk
310         module SReg8R( input Clk, Reset,D,
311                        output reg [7:0] Q);
312            always@(posedge Clk, posedge Reset)
313            begin
314               if(Reset)
315                  Q <= 7'b0;
316               else
317                  Q <= {Q[6:0],D};
318            end
319         endmodule
320
321         //I store D whenever I see En and the rising edge of a clock
322         module Reg8enR(input Clk, Reset, En,
323                        input  [7:0] D,
324                        output reg [7:0] Q);
325            always@(posedge Clk, posedge Reset)
326            begin
327               if(Reset)
328                  Q <= 7'b0;
329               else
330                  if(En)
331                     Q <= D;
332            end
333         endmodule
334
335         /*
336          * A Dot Draw module that can hold up to 10 draw requests at a time.
337          */
338         module DotDraw(input VGAClk, Reset,
339                        input [15:0] XCoordinate, YCoordinate,
340                        input ColorMode,
341                        input [7:0]  Color,
342                        input DDGo,
343                        input [15:0] XNext, YNext,
344                        output DrawDot,
345                        input DotEnable,
346                        output [7:0] BusColor,
347                        output buffFull);
348
349                        reg  [89:0] XCoords;
350                        reg  [89:0] YCoords;
351                        reg  [79:0] Colors;
352                        reg  [9:0]  Unwritten;
353                        wire [9:0]  StillUnwritten;
354                        wire [9:0]  WriteRequest;
355                        reg  [9:0]  Writing;
356
357            //if second to last dot hasn't been drawn predict that full soon
358            assign buffFull = Unwritten[1];
359
360            // See if any dot module wants to draw
361            assign WriteRequest =
362            {
363               (XNext == XCoords[8:0] & YNext == YCoords[8:0]),
364               (XNext == XCoords[17:9] & YNext == YCoords[17:9]),
365               (XNext == XCoords[26:18] & YNext == YCoords[26:18]),
366               (XNext == XCoords[35:27] & YNext == YCoords[35:27]),
```

```
367             (XNext == XCoords[44:36] & YNext == YCoords[44:36]),
368             (XNext == XCoords[53:45] & YNext == YCoords[53:45]),
369             (XNext == XCoords[62:54] & YNext == YCoords[62:54]),
370             (XNext == XCoords[71:63] & YNext == YCoords[71:63]),
371             (XNext == XCoords[80:72] & YNext == YCoords[80:72]),
372             (XNext == XCoords[89:81] & YNext == YCoords[89:81])
373          };
374
375          //If any dot module wants to draw, request the ColorBus
376          assign DrawDot = ~(WriteRequest==10'b0);
377
378          assign StillUnwritten = Unwritten&~Writing;
379
380          always@(posedge VGAClk, posedge Reset)
381          begin
382             if (Reset)
383             begin
384                XCoords   <= 90'b0;
385                YCoords   <= 90'b0;
386                Colors    <= 80'b0;
387                Unwritten <= 10'b0;
388                Writing   <= 10'b0;
389             end
390
391             else
392             begin
393                Writing <= WriteRequest;
394
395                // if told to draw, shift all the draws down one dot module
396                if(DDGo)
397                begin
398                   XCoords   <= {XCoordinate[8:0],XCoords[89:9]};
399                   YCoords   <= {YCoordinate[8:0],YCoords[89:9]};
400                   Colors    <= {Color[7:0],Colors[79:8]};
401                   Unwritten <= {1'b1,StillUnwritten[9:1]};
402                end
403                // otherwise keep the values until they're drawn
404                else
405                   Unwritten <= StillUnwritten;
406             end
407          end
408
409          //use the Writing signal to use the right color to drive color bus
410          assign BusColor = ~(DotEnable) ? 8'bzzzz_zzzz :
411             Writing[9] ? Colors[79:72]:
412             Writing[8] ? Colors[71:64]:
413             Writing[7] ? Colors[63:56]:
414             Writing[6] ? Colors[55:48]:
415             Writing[5] ? Colors[47:40]:
416             Writing[4] ? Colors[39:32]:
417             Writing[3] ? Colors[31:24]:
418             Writing[2] ? Colors[23:16]:
419             Writing[1] ? Colors[15:8]:
420             Writing[0] ? Colors[7:0]: 8'bzzzz_zzzz ;
421       endmodule
422
423       /*
424        * Draws a solid color as long as it's told to.
425        * Should be changed just draw the screen a solid color.
426        */
427       module SolidColor(input VGAClk, Reset,
```

```
428                     input SolidEnable,
429                     input [7:0] ColorIn,
430                     output SolidReq,
431                     output [7:0] BusColor,
432                     input Go,
433                     input Vsync);
434
435        reg [7:0] DrawColor;
436        reg [1:0] State;
437
438        //if there's a request to draw a solid color
439
440        assign SolidReq = ~(State == 2'b00);
441        assign BusColor = (SolidEnable) ? DrawColor : 8'bzzzz_zzzz;
442
443        always@(posedge VGAClk, posedge Reset)
444        begin
445           if (Reset)
446           begin
447              State <= 2'b01;                       //on reset do a draw
448              DrawColor <= 8'b0;                    //clear DrawColor
449           end
450           else
451           begin   //this state machine tracks whether whole screen has been printed
452              case(State)
453                 2'b00:
454                    begin
455                       if(Go)
456                       begin
457                          State <= 2'b01;                        //waiting state
458                          DrawColor <= ColorIn;                    //This color
459                       end
460                    end
461                 2'b01: State <= ~Vsync ? 2'b10 : 2'b01;             //I'm drawing
462                 2'b10: State <= Vsync ? 2'b11 : 2'b10;      //I've seen one negedge
463                 2'b11: State <= ~Vsync ? 2'b00 : 2'b11;//Im looking for second edge
464              endcase
465           end
466        end
467
468     endmodule
469
470     /*
471      * Will draw a rectange specified by the given upper left hand corner
472      * and the lower right hand corner. Each instance of the module can only
473      * draw one rectangle at a time. The Ready signal goes high when the
474      * module can take another rectangle input.
475      */
476     module RectangleDraw(input VGAClk, Reset,
477                          input [15:0] UpperX, UpperY,
478                          input [15:0] LowerX, LowerY,
479                          input [15:0] XNext, YNext,
480                          input [7:0] RectColor,
481                          input RectEnable,
482                          input Go,
483                          output RectReq,
484                          output reg Ready,
485                          output [7:0] BusColor);
486
487        reg SeenFirst, SeenLast; //Checks to see if the full rectangle was drawn
488
```

```
489         // Registers to keep hold start values until the rectange is drawn
490         reg GoReg;
491         reg [7:0] ColorReg;
492         reg [15:0] UpperXReg, UpperYReg, LowerXReg, LowerYReg;
493         reg [15:0] LastUpperX, LastUpperY, LastLowerX, LastLowerY;
494
495         always@(posedge VGAClk, posedge Reset)
496         begin
497            if(Reset)
498            begin
499               GoReg <= 1'b0;
500               SeenFirst <= 1'b0;
501               SeenLast <= 1'b0;
502               UpperXReg <= 16'b0;
503               UpperYReg <= 16'b0;
504               LowerXReg <= 16'b0;
505               LowerYReg <= 16'b0;
506               Ready <= 1'b1;
507               ColorReg <= 8'b0;
508            end
509            else
510            begin
511               //Enabled on ready registers
512               if (Ready) //If ready, take new values
513               begin
514                  UpperXReg <= UpperX;
515                  UpperYReg <= UpperY;
516                  LowerXReg <= LowerX;
517                  LowerYReg <= LowerY;
518                  ColorReg <= RectColor;
519               end
520               // Keep drawing for a full frame and guarranty that the entire
521               // rectangle gets drawn. Enabled on !Go registers
522               if (GoReg)
523               begin
524                  Ready <= 1'b0;
525                  if (XNext == UpperXReg & YNext == UpperYReg)
526                     SeenFirst <= 1'b1;
527                  else
528                     SeenFirst <= SeenFirst;
529                  if (XNext == LowerXReg & YNext == LowerYReg)
530                     SeenLast <= 1'b1;
531                  else
532                     SeenLast <= SeenLast;
533
534                  if (SeenFirst & SeenLast)
535                     GoReg <= 1'b0;
536               end
537               //Otherwise look for new values for go and set Ready
538               //and reset SeenFirst and SeenLast.
539               else
540               begin
541                  GoReg <= Go;
542                  Ready <= 1'b1;
543                  SeenFirst <= 1'b0;
544                  SeenLast <= 1'b0;
545               end
546            end
547         end
548
549         assign RectReq = (XNext > UpperXReg &
```

```
550                          XNext < LowerXReg &
551                          YNext > UpperYReg &
552                          YNext < LowerYReg & GoReg);
553        assign BusColor = (RectEnable) ? ColorReg : 8'bzzzz_zzzz;
554
555     endmodule
556
557
558     /*
559      * The Memory Controller module. This module manages memory operations
560      * and sends output colors to the VGA controller.
561      * it also outputs the X and Y coordinates for upcoming draws
562      */
563     module MemCont(input VGAClk, Reset,
564                    input  [7:0]  BusColor,
565                    input  [1:0]  ColorMode,
566                    input  [7:0]  DrawReq,
567                    output reg [15:0] XOut,YOut,
568                    output reg [15:0] XScan,YScan,
569                    output [7:0] Enable,
570                    output [7:0]  OutColor,
571                    input writ);
572
573
574                    wire [7:0] MemOut;       // The next memory value
575                    wire [7:0] MemWriteBack; // The value to be written to memory
576                    reg [7:0] MemNow;        // The current memory value
577
578                    //Scan values adjusted for first writable position to be at 0
579                    //There is a one cycle delay on the memory values, so there
580                    //is one pipeline stage going into the memory, another just
581                    //after the memory, and a final stage where all values are
582                    //current for the pixel about to be written. Y values
583                    //are constant (and change during porches) so Y scan location
584                    //does not need as much pipeline.
585                    wire [15:0] XScanMemIn,YScanMemIn;
586                    reg  [7:0]  XScanMemOut;
587                    reg  [15:0] XScanNow,YScanNow;
588
589                    //The three stages of memory address as taken from (X,Y)Scan
590                    wire [14:0] MemAddrIn;
591                    reg  [14:0] MemAddrOut;
592                    reg  [14:0] MemAddrNow;
593
594                    //Memory Write Enable
595                    wire MemWE;
596
597                    //Either holds the MemWriteBack values for the current memory
598                    //byte or the next memory value
599                    wire [7:0] TempMemVal;
600
601                    //Holds the current bus color
602                    reg  [7:0]  BusColorNow;
603
604                    // A register to keep last cycles requests for selMaskSig to
605                    // decide whether to use the mem location or the CurBusColor
606                    reg [7:0] EnableNow;
607
608                    //Holds either the current memory color value
609                    //or the BusColor value
610                    wire [7:0] Color2Mask;
```

```
611
612                 //The current color from memory based on color mode
613                 reg [7:0] MemColorNow;
614
615                 //The write back and color values for the different colore modes
616                 wire [7:0] MemWB1, MemWB2, MemWB4;
617                 wire [7:0] Color1, Color2, Color4;
618
619         /***************************************************************************
620                        BEGIN ASSIGNMENTS AND SUB-MODULES
621         ***************************************************************************/
622
623         // The X and Y Scan for memory addressing
624         // adjust to make top left corner 0,0
625         assign XScanMemIn = XScan - 136;
626         assign YScanMemIn = YScan - 28;
627
628         assign MemAddrIn = {YScanMemIn[8:2],XScanMemIn[9:2]};
629
630         //If the current memory address is different from the address
631         //beign read, then MemNow should take next memory value,
632         //otherwise it should take the updated memory byte
633         assign TempMemVal = (MemAddrOut != MemAddrNow) ? MemOut : MemWriteBack;
634
635         //If the current memory address is different from the address
636         //beign read, then write back the updated memory byte
637         assign MemWE = (MemAddrOut != MemAddrNow) & writ;
638
639         //An 8bx32786 dual ported RAM, not big enough for VGA
640         //but big enough for something close
641         VideoMem videomem(VGAClk,MemWriteBack,MemAddrNow,MemWE,
642                           VGAClk,MemAddrIn,MemOut);
643
644         //Construct the appropriate MemoryColorNow depending on the color Mode
645         always@(*)
646         begin
647            case (ColorMode)
648               2'b00:case({XScanNow[1],YScanNow[1:0]})
649                        3'b000:MemColorNow <= {7'b0,MemNow[0]};
650                        3'b001:MemColorNow <= {7'b0,MemNow[1]};
651                        3'b010:MemColorNow <= {7'b0,MemNow[2]};
652                        3'b011:MemColorNow <= {7'b0,MemNow[3]};
653                        3'b100:MemColorNow <= {7'b0,MemNow[4]};
654                        3'b101:MemColorNow <= {7'b0,MemNow[5]};
655                        3'b110:MemColorNow <= {7'b0,MemNow[6]};
656                        3'b111:MemColorNow <= {7'b0,MemNow[7]};
657                     endcase
658               2'b01:case({XScanNow[1],YScanNow[1]})
659                        2'b00:MemColorNow <= {6'b0,MemNow[1:0]};
660                        2'b01:MemColorNow <= {6'b0,MemNow[3:2]};
661                        2'b10:MemColorNow <= {6'b0,MemNow[5:4]};
662                        2'b11:MemColorNow <= {6'b0,MemNow[7:6]};
663                     endcase
664               2'b10:case(YScanNow[1])
665                        1'b0:MemColorNow <= {4'b0,MemNow[3:0]};
666                        1'b1:MemColorNow <= {4'b0,MemNow[7:4]};
667                     endcase
668               2'b11:MemColorNow <= MemNow;
669            endcase
670         end
671
```

```
672         //If the ColorBus was enabled, take the BusColor,
673         //otherwise select the MemNowColor to mask and output
674         Mux1_8   selMaskSig(MemColorNow,BusColorNow,
675                         ((|EnableNow)!= 1'b0),Color2Mask);
676
677         //Add a stage of pipeline and output the enable signals
678         //for the given draw requests using a priority decoder
679         ColorBusEnable selectPeripheral(VGAClk,Reset,DrawReq,Enable);
680
681         //Combinational logic that outputs the current color
682         //to the VGA controller and the modified byte to write back to
683         //the memory for each color mode
684         ColorMask1b mask1b(MemNow,Color2Mask,XScanNow[1],
685                         YScanNow[1:0],MemWB1,Color1);
686         ColorMask2b mask2b(MemNow,Color2Mask,XScanNow[1],YScanNow[1],MemWB2,Color2);
687         ColorMask4b mask4b(MemNow,Color2Mask,YScanNow[1],MemWB4,Color4);
688
689         //Select the correct color and write back values depending on ColorMode
690         Mux2_8 selectWB(MemWB1,MemWB2,MemWB4,Color2Mask,ColorMode,MemWriteBack);
691         Mux2_8 selectC(Color1,Color2,Color4,Color2Mask,ColorMode,OutColor);
692
693         //Combinational logic that assigns the X and Y output coordinates
694         //of the next pixel depending on the color mode
695         always@(*)
696         begin
697            case(ColorMode)
698               2'b11:begin
699                  // XOut is between 0 and 160
700                  XOut = (XScan > 136) ? {8'b0,XScanMemIn[9:2]} : 16'b0;
701                  // YOut is between 0 and 120
702                  YOut = (YScan > 27) ? {9'b0,YScanMemIn[8:2]} : 16'b0;
703               end
704               2'b10:begin
705                  // XOut is between 0 and 160
706                  XOut = (XScan > 136) ? {8'b0,XScanMemIn[9:2]} : 16'b0;
707                  // YOut is between 0 and 240
708                  YOut = (YScan > 27) ? {8'b0,YScanMemIn[8:1]} : 16'b0;
709               end
710               2'b01:begin
711                  // XOut is between 0 and 320
712                  XOut = (XScan > 136) ? {7'b0,XScanMemIn[9:1]} : 16'b0;
713                  // YOut is between 0 and 240
714                  YOut = (YScan > 27) ? {8'b0,YScanMemIn[8:1]} : 16'b0;
715               end
716               2'b00:begin
717                  // XOut is between 0 and 320
718                  XOut = (XScan > 136) ? {7'b0,XScanMemIn[9:1]} : 16'b0;
719                  // YOut is between 0 and 480
720                  YOut = (YScan > 27) ? {7'b0,YScanMemIn[8:0]} : 16'b0;
721               end
722            endcase
723         end
724
725         // Resetable Registers
726         always@(posedge VGAClk, posedge Reset)
727         begin
728            if(Reset)
729            begin
730            //reset state of memCont
731               MemNow          <= 8'b0;
732               MemAddrOut      <= 8'b0;
```

```
733                 MemAddrNow      <= 15'b0;
734                 XScanMemOut     <= 8'b0;
735                 XScanNow        <= 16'b0;
736                 YScanNow        <= 16'b0;
737                 XScan           <= 16'b0;
738                 YScan           <= 16'b0;
739                 BusColorNow     <= 8'b0;
740                 EnableNow       <= 8'b0;
741              end
742              else
743              begin
744              //Clk edge, registers get new values
745                 MemAddrOut <= MemAddrIn;
746                 MemAddrNow <= MemAddrOut;
747
748                 MemNow <= TempMemVal;
749
750                 EnableNow <= Enable;
751
752                 XScanMemOut <= XScanMemIn;
753                 XScanNow <= XScanMemOut;
754                 YScanNow <= YScanMemIn;
755
756                 BusColorNow <= BusColor;
757
758
759           /***********************************************
760                 Manage Scan Lines for VGA Controller
761           ***********************************************/
762                 if (XScan < 800)
763                    XScan <= XScan+1;
764                 else
765                 begin
766                    XScan <= 0;
767                    if (YScan < 525)
768                       YScan <= YScan+1;
769                    else
770                    begin
771                       YScan <= 0;
772                    end
773                 end
774              end
775           end
776        endmodule
777
778        module Mux3_1( input  [7:0] D,
779                       input  [2:0] Sel,
780                       output reg Q);
781           //3 select lines and 1 output bit
782           always@(*)
783           begin
784              case(Sel)
785                 3'b000:Q<=D[0];
786                 3'b001:Q<=D[1];
787                 3'b010:Q<=D[2];
788                 3'b011:Q<=D[3];
789                 3'b100:Q<=D[4];
790                 3'b101:Q<=D[5];
791                 3'b110:Q<=D[6];
792                 3'b111:Q<=D[7];
793              endcase
```

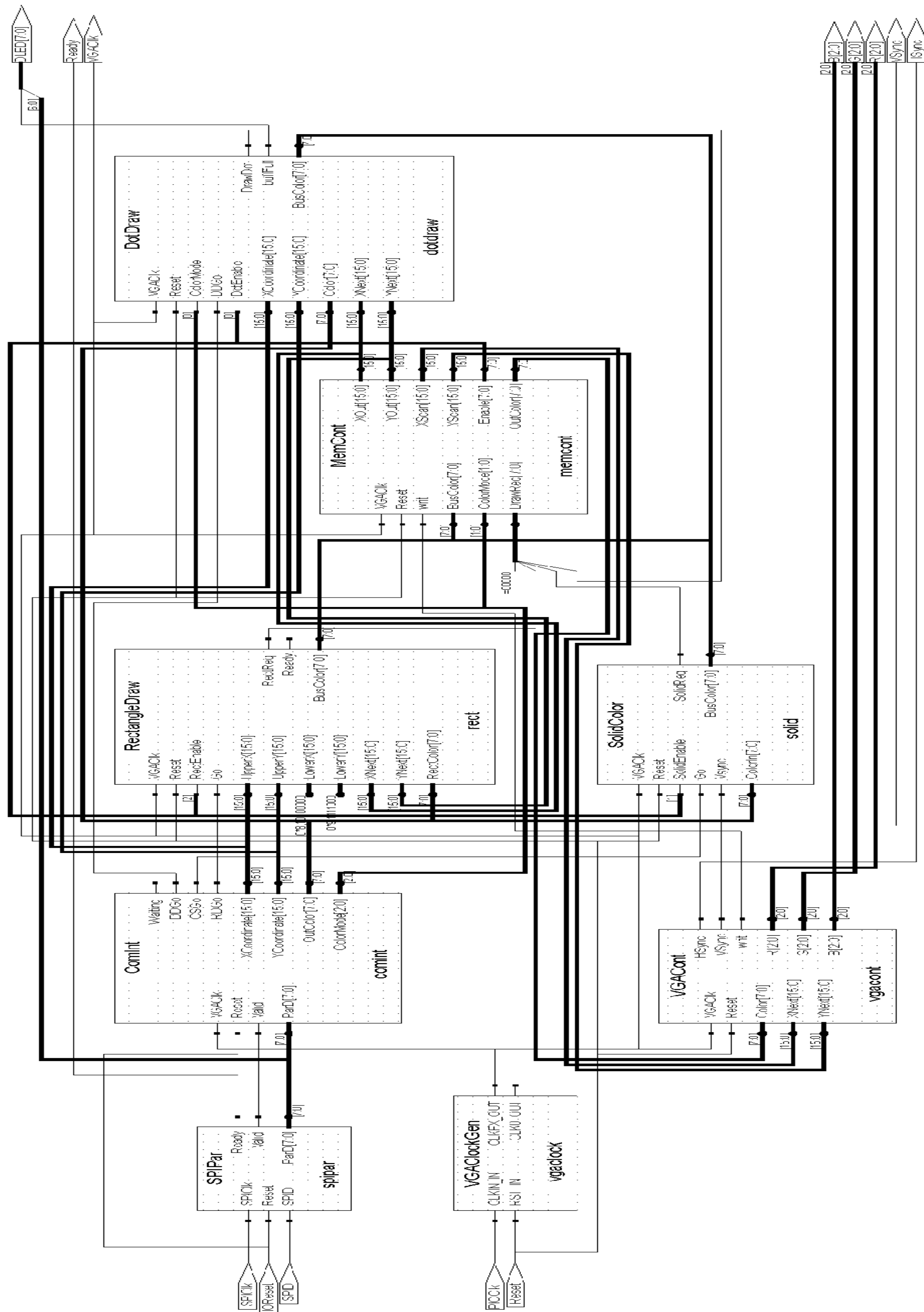
```
794         end
795       endmodule
796
797       //2 select lines and 2 output bits
798       module Mux2_2( input   [7:0] D,
799                      input   [1:0] Sel,
800                      output reg [1:0]Q);
801
802         always@(*)
803         begin
804            case(Sel)
805               2'b00:Q<=D[1:0];
806               2'b01:Q<=D[3:2];
807               2'b10:Q<=D[5:4];
808               2'b11:Q<=D[7:6];
809            endcase
810         end
811       endmodule
812
813       //1 select line and 4 output bits
814       module Mux1_4( input   [7:0] D,
815                      input   Sel,
816                      output reg [3:0]Q);
817
818         always@(*)
819         begin
820            case(Sel)
821               1'b0:Q<=D[3:0];
822               1'b1:Q<=D[7:4];
823            endcase
824         end
825       endmodule
826
827       //1 select line and 8 output bits
828       module Mux1_8( input   [7:0] D0,
829                      input   [7:0] D1,
830                      input   Sel,
831                      output reg [7:0] Q);
832
833         always@(*)
834         begin
835            case(Sel)
836               1'b0:Q<=D0;
837               1'b1:Q<=D1;
838            endcase
839         end
840       endmodule
841
842       //2 select lines and 8 output bits
843       module Mux2_8( input   [7:0] D0,
844                      input   [7:0] D1,
845                      input   [7:0] D2,
846                      input   [7:0] D3,
847                      input   [1:0] Sel,
848                      output reg [7:0] Q);
849
850         always@(*)
851         begin
852            case(Sel)
853               2'b00:Q<=D0;
854               2'b01:Q<=D1;
```

```
855                 2'b10:Q<=D2;
856                 2'b11:Q<=D3;
857              endcase
858           end
859        endmodule
860
861
862        /*
863         * Takes the draw requests and outputs the appropriate
864         * enable signals using a priority decoder to give
865         * enable the highest priority request.
866         */
867        module ColorBusEnable(input VGAClk, Reset,
868                              input [7:0] DrawReq,
869                              output reg [7:0] Enable);
870
871           always@(posedge VGAClk, posedge Reset)
872           begin
873              if(Reset)
874                 Enable <= 8'h00;
875              else
876              begin
877                 casez(DrawReq)
878                    8'b1???_????: Enable <= 8'h80;
879                    8'b01??_????: Enable <= 8'h40;
880                    8'b001?_????: Enable <= 8'h20;
881                    8'b0001_????: Enable <= 8'h10;
882                    8'b0000_1???: Enable <= 8'h08;
883                    8'b0000_01??: Enable <= 8'h04;
884                    8'b0000_001?: Enable <= 8'h02;
885                    8'b0000_0001: Enable <= 8'h01;
886                    default: Enable <= 8'h00;
887                 endcase
888              end
889           end
890
891        endmodule
892
893        //Wordmap
894        //Word is broken down 4 different ways.
895        //each word represents a 4x4 grid of monitor pixels
896        //that can be further broken down to
897        //2x1 tinyPixels,
898        //2x2 littlePixels,
899        //4x2 mediumPixels,
900        //or left as 4x4 largePixels
901
902        module ColorMask1b(  input [7:0] MCByte,
903                             input [7:0] CByte,
904                             input xPos,
905                             input [1:0] yPos,
906                             output reg[7:0] DatWB,
907                             output [7:0] DatC);
908
909                             wire OutSig;
910
911           //select the appropriate signal to output
912           assign OutSig = CByte[0];
913           //Mux3_1 outSelMux(CByte,{xPos,yPos},OutSig);
914
915           //mix to Black White
```

```
916         assign DatC = {OutSig,OutSig,OutSig,OutSig,OutSig,OutSig,OutSig,OutSig};
917
918         always@(*)
919         begin
920            case({xPos,yPos})//mask and shift to reconstruct memory word
921               3'b000:DatWB<={MCByte[7:1],OutSig};
922               3'b001:DatWB<={MCByte[7:2],OutSig,MCByte[0]};
923               3'b010:DatWB<={MCByte[7:3],OutSig,MCByte[1:0]};
924               3'b011:DatWB<={MCByte[7:4],OutSig,MCByte[2:0]};
925               3'b100:DatWB<={MCByte[7:5],OutSig,MCByte[3:0]};
926               3'b101:DatWB<={MCByte[7:6],OutSig,MCByte[4:0]};
927               3'b110:DatWB<={MCByte[7],OutSig,MCByte[5:0]};
928               3'b111:DatWB<={OutSig,MCByte[6:0]};
929            endcase
930         end
931      endmodule
932
933
934      module ColorMask2b(  input [7:0] MCByte,
935                           input [7:0] CByte,
936                           input xPos,
937                           input yPos,
938                           output reg [7:0] DatWB,
939                           output [7:0] DatC);
940
941                           wire [1:0] OutSig;
942         //Mux2_2 outSelMux(CByte,{xPos,yPos},OutSig);
943
944         //select appropriate output
945         assign OutSig = CByte[1:0];
946
947         //grayScale output
948         assign DatC = {OutSig,OutSig[1],OutSig,OutSig[1],OutSig};
949
950         always@(*)
951         begin
952            case({xPos,yPos})//mask and shift to reconstruct memory word
953               2'b00:DatWB<={MCByte[7:2],OutSig};
954               2'b01:DatWB<={MCByte[7:4],OutSig,MCByte[1:0]};
955               2'b10:DatWB<={MCByte[7:6],OutSig,MCByte[3:0]};
956               2'b11:DatWB<={OutSig,MCByte[5:0]};
957            endcase
958         end
959      endmodule
960
961
962      module ColorMask4b(  input [7:0] MCByte,
963                           input [7:0] CByte,
964                           input yPos,
965                           output reg[7:0] DatWB,
966                           output [7:0] DatC);
967
968                           wire [3:0] OutSig;
969
970         //select appropriate output
971         assign OutSig = CByte[3:0];
972         //Mux1_4 outSelMux(CByte,yPos,OutSig);
973
974         //4bColor out 1'bR,2'bG,1'bB
975         assign DatC = {OutSig[3],OutSig[3],OutSig[3],OutSig[2:1],OutSig[1],
976                           OutSig[0],OutSig[0]};
```

```
977
978         always@(*)
979         begin
980            case(yPos) //mask and shift to reconstruct memory word
981               1'b0:DatWB<={MCByte[7:4],OutSig};
982               1'b1:DatWB<={OutSig,MCByte[3:0]};
983            endcase
984         end
985      endmodule
986
987      //Note that no masking or writeback reconstruction is required for 8b'Color
988
989      module VGACont(input VGAClk,
990                     input Reset,
991                     input [7:0] Color,
992                     input [15:0] XNext, YNext,
993                     output reg HSync, VSync,
994                     output reg [2:0] R, G, B,
995                     output writ);
996
997                     wire [2:0] writable;
998
999            //are we in the writable area?
1000           assign writable[0] = (XNext > 137) & (XNext < 778) &
1001                                (YNext > 27) & (YNext < 508);
1002           assign writable[1] = writable[0];
1003           assign writable[2] = writable[1];
1004           assign writ = writable[0];
1005
1006           always@(posedge VGAClk, posedge Reset)
1007           begin
1008              if(Reset)
1009              begin//reset the state
1010                 HSync <= 1'b0;
1011                 VSync <= 1'b0;
1012                 R     <= 3'b0;
1013                 G     <= 3'b0;
1014                 B     <= 3'b0;
1015              end
1016              else
1017              begin
1018                 HSync <= ~(XNext < 96);//negative polarity syncs.
1019                 VSync <= ~(YNext < 2);
1020                 R     <= Color[7:5]&writable;//make our porches and such.
1021                 G     <= Color[4:2]&writable;
1022                 B     <= {Color[1:0],Color[1]}&writable;//using Color1 will make blues brigh
1023              end
1024           end
1025     endmodule
1026
```

Appendix C Top Level of Graphics Controller



Appendix D Memory Controller Timing Diagram

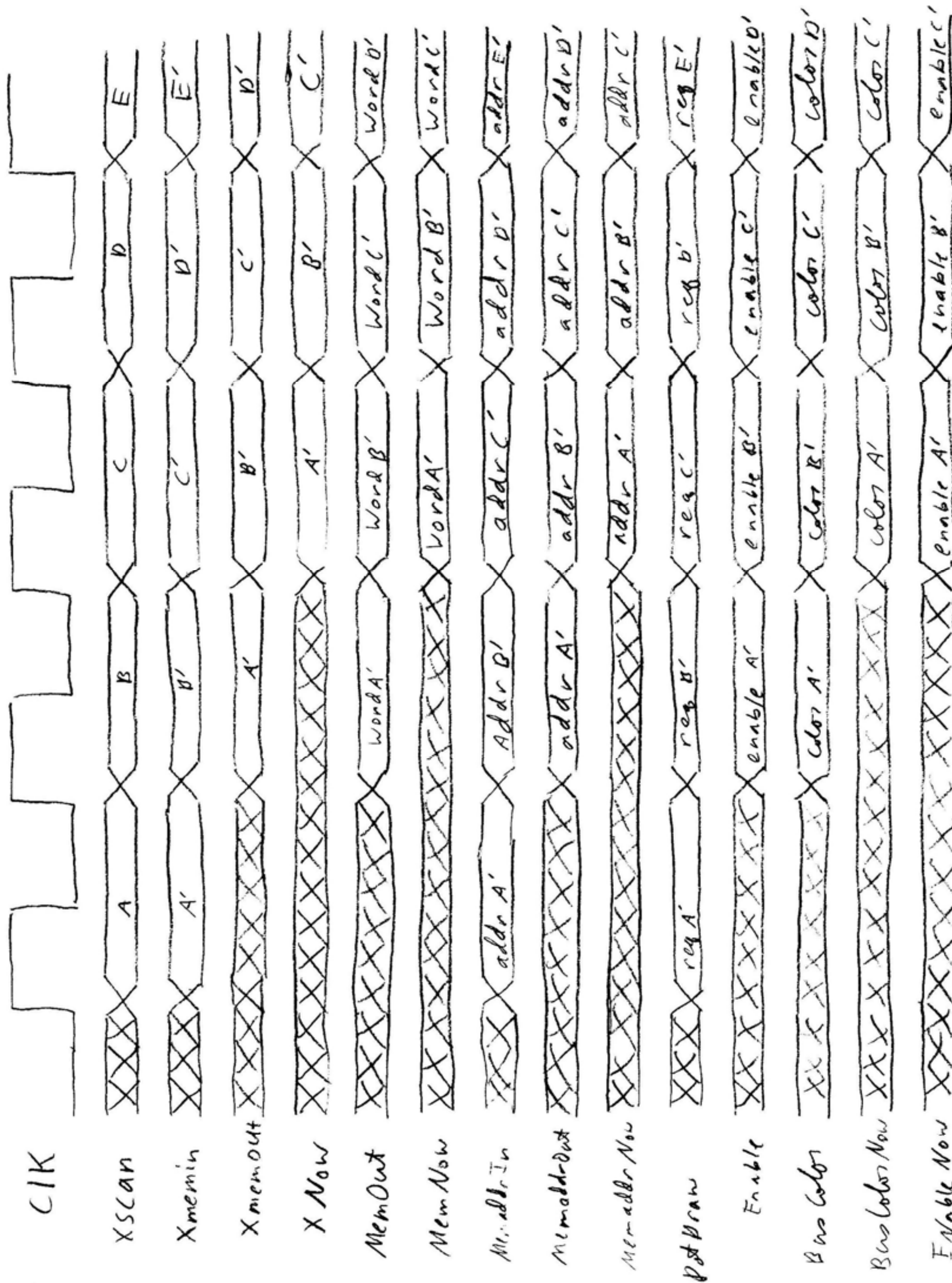


Figure 5: Timing diagram for signals moving through the Memory Controller. Signal names are as specified by the Verilog file in Appendix B.