TWO-DIMENSIONAL GRAPHICS CARD (GPU) ON AN ALTERA FPGA

A Design Project Report

Presented to the Engineering Division of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering (Electrical and Computer)

by

Peter Alexander Greczner

Project Advisor: Dr. Bruce Land

Degree Date: May, 2010

Abstract

Master of Electrical and Computer Engineering Program

Cornell University

Design Project Report

Project Title: Two-Dimensional Graphics Card (GPU) on an Altera FPGA

Author: Peter Alexander Greczner

Abstract:

This project realizes a simple graphics processing unit (GPU) on an FPGA which can be used as a GPU for Atmega microcontrollers to output graphics to a VGA display. An Altera Cyclone II FPGA will be used to implement the GPU. The FPGA and the host device will communicate via a RS-232 serial interface using a handshaking command structure to ensure accuracy of instruction delivery. Final output of the graphics will be displayed on a VGA compatible monitor. The result of the project is a GPU as well as VGA driver for the host device that allows a user to set up a graphical window to draw into and display on the VGA screen. The GPU supports multiple drawing functions such as text writing, line drawing, and rectangular fills. It also provides two different screen resolutions to draw into, 640x480 and 320x240, as well as capability for both single and double buffered graphics. The end product allows a simple method for one to use a microcontroller or other host device to create high quality graphical program without sacrificing memory, speed, or timing constraints of said microcontroller.

Report Approved by

Project Advisor: ____ Date: _______

1. Executive Summary

A simple two-dimensional graphics processing unit (GPU) was implemented on an Altera Cyclone II FPGA. The GPU is capable of understanding 80bit instructions sequentially sent to it via RS-232 serial communication. The only requirement being from a user side is that they have installed the proper C files and have a means of sending a byte over serial. As instruction requests are handled, they are appropriately written into SRAM and later displayed on a VGA compatible monitor.

This GPU supports an extensible number of drawing modules since a common data-flow scheme is inherent to all modules. Current modules include the functionality for the drawing of lines, circles, rectangles, and polygons. As well as the filling of both circles and rectangles, and that of writing text to the monitor.

The GPU is also able to support four different drawing modes. The modes are first divided into what resolution the monitor displays. These are 640x480 pixels and 320x240 pixels. The second level of division is that of either single or double buffering. A frame buffer was implemented on the GPU to allow for a user to draw into a frame and not have the result displayed until they specify the update. This process eliminates flickering that one would see from constantly refreshing the same buffer. The single buffer mode was still included because it offers simplicity in some applications, specifically low-latency applications.

Color depth is supported up to 12bit RGB in the single buffered 320x240 drawing mode. Single buffer 640x480 and double buffer 320x240 support 8bit RGB. Double buffer 640x480 supports 4bit RGB.

A hardware serial receiver and transmitter module was implemented at a baud rate of 115200 to provide the communication between FPGA and host device. The serial communication supports 8bit data, 1 start bit, and 1 stop bit. There is no flow control, and parity check is done via a simple CRC of the packet, not byte by byte.

With a few minor modifications the GPU will provide the ability to draw into sprites and support multiple fonts. One goal of the project was to implement a method of downloading instructions into an instruction RAM (IRAM) and having a dedicated processor handle the graphics operations directly on the GPU. At the time of this report, that functionality is only partially implemented and still being tested and debugged.

The specific FPGA hardware used was the Altera Cyclone II DE2 EP2C35F672C6 development board.

2. Design Problem and Specifications

2.1 Design Problem Description

This project stemmed from addressing the problem of providing high resolution color graphics for constrained embedded applications, such as a microcontroller. In the situation such as a microcontroller you often need extra hardware to provide color graphics, but then you are still limited to memory constraints of the microcontroller. In situations without extra hardware you may be limited to black and white low resolution graphics over NTSC signal. Since many microcontrollers provide support for RS-232 serial communication, a question was posed of: can the graphics operations be offloaded via serial to a GPU, and how would that GPU hardware be realized?

The proposed solution to this problem was to have an FPGA act as the GPU co-processor to the embedded application and they would communicate via RS-232 serial.

2.2 Design Specifications

In development of the GPU the issues to address can be split into a few categories: performance parameters, functionality, communication, and end-user design.

2.2-1 Performance Parameters

Performance Parameters includes what the user can expect to achieve from the GPU. These are things such as frames per second and available resolutions.

Performance Parameter	**Design Specification**
Frames per second	10 fps
Resolution 1	320x240
Resolution 2	640x480
Color Depth	16 predefined colors
Frame Buffers	2
Fonts	1 downloadable (MIF)

2.2-2 Drawing Functionality

- Shapes – filled and unfilled (Circles, Rectangles, Polygons, and lines)
- Character display – 8x8 pixel characters to be printable on the screen
- Sprites – using MIF's allow for drawing into this frame
- Frames – two full resolution frames
- Drawing queue – queue of instructions that can be programmed to automatically get executed each frame instead of the microcontroller continuingly specifying

Originally the design specified that there would be hardware polygon manipulation, but as the development wore on it became apparent that hardware polygon manipulation is not something that should be a primary goal of a two-dimensional graphics card. Due to complicated issues in implementation of other parts of the project, this idea was scrapped from the initial proposal, in favor of more completely developing the other aspects.

2.2-3 Communication

The communication between the FPGA and the microcontroller will take place over RS-232 serial protocol. The microcontroller will output a command to the FPGA who will receive and send either an ACK or NACK indicating successful or unsuccessful reception of the command.

2.2-4 End-user Design

The end-user will be able to write graphics by including a graphics module in their C program that contains all the code necessary to write to the FPGA. Graphics functions will have a simple English naming convention, such as fillRectangle(x1,y1,w,h,red,green,blue). An API will be available to explain in detail how to write to the FPGA and what methods are available.

3. Range of Solutions

3.1 Image Memory

With the goal of providing multiple resolution color graphics which could be displayed on a VGA compatible monitor, the choice of hardware solution was to be an FPGA. Other options such as color NTSC and an extra hardware chip were discounted as mentioned above. The next logical step was to decide on possible implementation approaches for aspects of the project.

One aspect of the project was where to store the image memory. The DE2 board allowed for four possible approaches to this image memory: SRAM, SDRAM, Flash, and M4K blocks. SRAM has 512Kbytes of memory, SDRAM has 8Mbytes, Flash has 4Mbytes, and there are 104 M4K (4Kbytes blocks) blocks. To support 16 colors (4 bits) in two frames at a resolution of 640x480 requires:

0.5bytes * 640 pixels * 480 pixels * 2 frames = 307.2Kbytes

Ultimately SRAM was chosen, and below is a description of the other possible approaches strengths and weaknesses.

Each type of memory option has enough memory capable for storing both image frames. The next criteria is how easy it will be to access each type of memory, since the graphics card will constantly be reading and writing to the memory.

Flash memory isn't memory that is used in constant write operations since it has a limited number of write-erase cycles before it can no longer be used. Therefore, Flash memory was discounted.

The M4K blocks have an advantage that they could be individual accessible, in a grid like fashion, so parallel computation could be achieved on each region of the image. This would work well in the case of rectangle fills, which take the most processing times. However, functions such as line drawing would still be sequential in nature, and figuring out which M4K block to select would be difficult. One other disadvantage to M4K blocks is that 77 blocks would need to be used, and that would only leave 27 other blocks for extensible features of the GPU, such as Fonts or Sprites. The complexity of the M4K blocks along with the extra hardware resources they suffocated discounted them as the image memory choice.

SDRAM provides the most available memory for the images. It can also be dual ported for reads and writes. However, the disadvantage is that the reads and writes have to be done in large blocks. The hardware implementation for this type of memory is also very complex and based on the proposed design of the drawing modules, it did not appear to be an appropriate choice.

SRAM was chosen as the best option for this project because it provided a sufficient amount of memory without adding too much excess memory. It interfaced nicely with the VGA controller and was simple to write to in that it supported single cycle reads and writes. One disadvantage was that it was only single ported. However, this was deemed an acceptable sacrifice based on the other advantages SRAM posed in terms of designing the GPU drawing modules.

3.2 Communication

The first communication choice to make was RS-232 versus USB. USB was discounted because it would require extra hardware attachments to most, if not all, microcontrollers as well as complicated hardware on the FPGA. However, the transmission rate would be very high and much less than the transmission overhead that serial communication provides. But ultimately, RS-232 was chosen because of the probability that most microcontrollers support some form of serial communication and it would require the least amount of overhead. In my prior work on FPGA's I had developed a RS-232 hardware receiver and transmitter that needed some modifications and an overhaul to be more robust and accurate for this project, but ultimately the baseline hardware was in place which made that option advantageous also.

3.3 Video Display

To display the images on a screen there were two different options, to use NTSC television signals or to use VGA signals. NTSC was discounted in favor of VGA because the original intent of the project was to provide a user developing graphical applications on a microcontroller, such as in Cornell Engineering ECE 4760, the ability to have color graphics. The lab in which this class is held has an abundance of VGA compatible monitors as opposed to color NTSC compatible televisions. VGA provided the most practical approach and it was in line with the goal of the project.

3.4 Draw Mode Specific Modules versus Single Data Writing Module

There was the option of having a different drawing module for each of the drawing functions and for each of the drawing modes. For example in that case there would be a line drawer for 640x480 single buffer, double buffer, and 320x240 single buffer and double buffer. Then each of these modules would have their output muxed based on if they should have been in use based on the current instruction. The advantage to this is that writing to SRAM can be done directly within each module since each module knows exactly how it needs to read and write its data based on drawing mode. Therefore, for something like a 320x240 single buffer mode in which no reads are needed to be done in order to write data, cycles are saved.

The second option was having universal drawing modules that output coordinates and data to a data writing module. The data writing module knows what drawing mode it is in and writes accordingly. The disadvantage here is that each drawing module needs to issue a signal indicating that the data writer should write, and then it needs to listen for the data writer to signal that a write has finished. This adds up to five extra states to each write operation. The advantage though, is that it allows for easier extensibility of adding more drawing modules, and for easier debugging of current drawing modules, since you only need to work about one line drawer having an error, not four or more different line drawers having an error.

Ultimately the multiple drawing modules per drawing function method of data writing was discounted due to unnecessary increased hardware and lack of extensibility. The method of using a single data writer, while adding more states, made it easiest to write data accurately based on the different modes, and provided a common interface for each drawing module to implement.

3.5 Purely Downloadable versus Serial Sequential

One option was to have the user download the entire graphics program into an instruction memory (IRAM) and have it run off of that. The benefit to this is that it would provide the fastest execution speed and possibly the best graphical results. However, the disadvantage is that of updating data on the fly while the microcontroller is running its applications. It was not clear how the GPU would handle interrupts and modify the program on the fly.

The alternative option was to have the graphics functions sent in line with the users host application. This is most similar to how graphics work on a PC application have the advantage of always having access to data used to create the graphics; i.e. no interrupts to change something in the DRAM of the processor, since the microcontroller/host sends the updated value within the instruction.

The end design of this project implemented the serial sequential approach and attempted to also implement the downloadable IRAM approach.

4. GPU Design

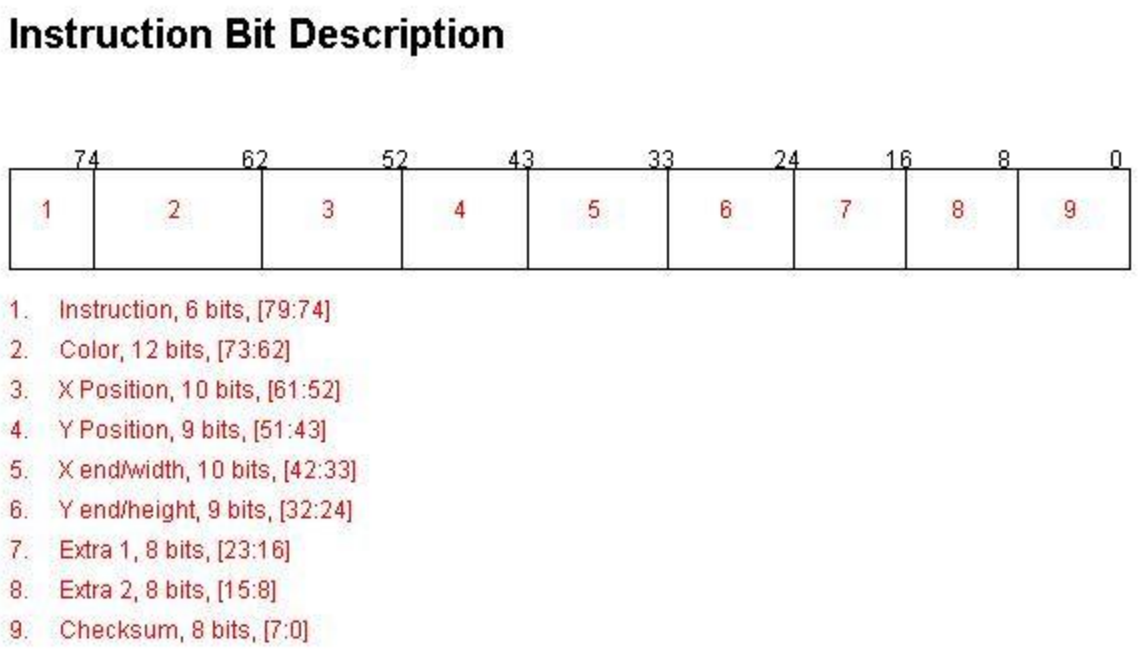
4.1 Overview

There are three levels of interaction in order to make the entire GPU application functional. There is the communication of instructions between host program and the GPU. There is the execution of the instruction within the GPU. Lastly, there is the displaying of the end result on the VGA monitor.

The general flow of information is that the host sends an instruction over serial to the GPU. The GPU processes the instruction, and sends a N/ACK back to the host. See Schematic #3 for an overview of the entire process.

4.2 Communication of Instructions

RS-232 was used as the method of communication of instructions between the host application and the GPU. The first issue that had to be resolved before any instructions could be sent was the issue of what comprises an instruction. The final instruction request is 80 bits long and a rationale for the length follows.



1. Instruction, 6 bits, [79:74]
2. Color, 12 bits, [73:62]
3. X Position, 10 bits, [61:52]
4. Y Position, 9 bits, [51:43]
5. X end/width, 10 bits, [42:33]
6. Y end/height, 9 bits, [32:24]
7. Extra 1, 8 bits, [23:16]
8. Extra 2, 8 bits, [15:8]
9. Checksum, 8 bits, [7:0]

The GPU set out to support the following drawing functions: fills for both circles and rectangles, draws for circles, rectangles, and lines, text writing, frame updating, and setting the draw mode. These comprise a total of eight instructions. However, when designing the instruction parameters I felt that possibly I would need more than just three bits to represent the eight instructions should I find a need to extend the functionalities. Therefore I chose to use six bits to represent the type of instruction to be executed. Color is transmitted as 12 bits with four bits for each of the red, green, and blue components.

The x position and x end/width are 10 bits in length which is the minimum number of bits to represent a range of 0-640. The y position and y end/height are 9 bits in length which is the minimum number of bits to represent a range of 0-480. The checksum is a simple XOR of all packets sent to quickly check if anything wrong in transmission has occurred. The reason it is so simple is because at these transmissions speeds and at such a short transmission length, error probability is very minimal. Lastly the two interesting components are the Extra 1 and Extra 2 bytes. I included those in the original design should I need to extend functionality of something that I had not originally planned. In fact I ended up using these to represent signals when attempting to implement the downloading of instructions to the IRAM, as well as some bits to represent what current drawing mode we were in. The main purpose of these extra bytes was to keep the design flexible to extra additions.

For the RS-232 serial communication I decided to use a baud rate of 115200. The data sent was 1 byte at a time (8 bits) with one start bit and one stop bit. This meant that each byte required 10 total bits to transmit. The rationale for this choice is that it is the highest supported baud rate on the PC that I was testing the GPU on. At a baud rate of 115200 with each instruction comprising 80 bits, this means each instruction takes (8+1+1)*10 = 100 bits to transmit. Minimum transmission of one instruction is 100/115200 seconds. To receive an acknowledgement of one byte takes (8+1+1)*1 = 10 bits / 115200 seconds. Maximum total instructions that can be sent per second equals 1047.3. Note that this means that no actual execution on the GPU took place, just instruction to be sent and immediately have an N/ACK follow. This value of 1047.3 instructions per second (IPS) will scale with the baud rate, and since higher IPS means better framerate and that one can do more with graphics, the highest possible baud rate was chosen.

The process for sending an instruction from host to the GPU is as follows. The host breaks up the instruction into 10 bytes to be sent over serial to the GPU. The GPU SerialReceiver module is constantly listening for new data on the serial port. Once a new byte comes in, it sets a flag to indicate that a new byte of data is available. Listening to this flag is a PacketManager. The PacketManager is responsible for taking each new byte of data and placing it into the appropriate spot in the packet; essentially, reorganizing the instruction as the bytes come in. After the PacketManager has recreated the entire instruction into a single 80-bit packet, it raises a flag to indicate that a new packet is available to process. Listening to this flag is the RequestManager. The RequestManager is responsible for taking in each new request that a user has sent and processing it according to the requests specifications. Once the request has been processed the RequestManager signals the SerialTransmitter to transmit back a N/ACK to the host to indicate either successful or unsuccessful processing. Refer to Schematic #1 in the appendix for a visual flow explanation of the process.

To synchronize the baud rates between the GPU and the host application a module was created to generate the baudrate that the SerialReceiver uses to listen to instructions. Using the 50MHz clock it is not possible to directly derive a baudrate of 115200 through the use of a counter. Therefore one can only approximate the baudrate to around 115200. Over time a skew will cause the receiver to miss a bit of data that is transmitted. To correct for this most receivers listen at a factor of eight or more times the baudrate for a start bit and then resynchronize to the standard baudrate. This ensures that no start bits are missed and the amount of skew that will be evident over the 10 bits required to receive one byte of data is negligible to caused missed data. Another baudrate generator is used to set the baudrate for the serial transmitter. This does not require a higher baudrate for listening, since the transmitter is only sending and it is the hosts duty to meet baudrate requirements.

4.3 Processing Instructions

4.3-1 Overview of Request Handling

Once an instruction is received it is given to the RequestManager which is responsible for managing the instruction request until completion. The RequestManager has nine states which it uses to process an instruction. Its primary state is listening for a new request. Once a new request is noticed it "grabs" the packet from the PacketManager and the PacketManager then lowers its new packet flag because that packet is now in use. After the packet is grabbed, the RequestManager parses the 80-bit instruction into its different component registers. The values of these 9 registers are outputted from the RequestManager into the top level module to be read by the drawing functions. At this point the RequestManager decides if the current instruction is a drawing instruction or a frame-update instruction. If it is a frame-update instruction it proceeds to the completion stage and tells the serial transmitter to send an ACK back to the host. Otherwise, it decides which type of drawing instruction is going to be executed and waits for that module to acknowledge that it is free for use. Once the wanted module has acknowledge that it is ready for an instruction, the RequestManager sends a start command to that module to begin its processing. All other drawing modules remain idle at this time. The RequestManager then listens for the selected drawing module's availability flag, and once the module indicates it is available again, the RequestManager understands that the drawing instruction has completed. At this point the RequestManager then signals the serial transmitter to send an ACK of completion to the host. Refer to Schematic #2 in the appendix for a visual representation of this process.

4.3-2 Processing a Drawing Instruction

Each drawing module has an inherent identification number that specifies which type of drawing module it is. This is the same number that when an instruction comes in the six bits that specify the instruction correspond to. The RequestManager starts the drawing by outputting which ID of the drawing module it wants to use along with a start command. Then, only the drawing module who's ID matches that from the RequestManager will process the start command. All modules have the requisite drawing information available to them, but only the module that is wanted will begin the drawing function and set its current availability to unavailable.

Each module has a different function, but all have the same inherent structure to drawing. There are three basic stages to performing a drawing function. The first is to listen for the start command. Once this is received it begins its drawing loop and continues within this loop until completion in which it exits back to the listening for a start command state. Also within the drawing loop is a stage that performs a write to memory. Each time the module wants to write to memory it enters a specific state in which is sets the data to be written to memory and choose the x and y addresses. It then remains in this state until a write is complete and then exits back to the loop. During the looping state the module sets itself as unavailable and in the listening for a start command state the module sets itself as available.

4.3-3 Handling a Data Write

To facilitate easier writing of data to memory based on what type of drawing mode is in use a DataWriter module was created. The purpose of this module is to properly pack the data into SRAM. Its other purpose is to separate memory functions from the drawing modules. This way drawing modules only need to worry about coordinates, in case memory specifications were to change, and thus they would not be affected in terms of functionality and performance.

Three muxes are implemented to feed the proper data into the DataWriter. These three muxes are for the X and Y Coordinates, and the color values. Each module sends a current value for each of these and the muxes select which one to output to the data writer based on which module the RequestManager has selected.

The DataWriter has four stages which include: waitForRequest, loadData, writeData, sendACK. In the waitForRequest state the DataWriter is listening for the drawing module to signal that it has data it wants to write. Once it receives this signal the DataWriter packages the color value appropriately so it can be written later into SRAM. In the loadData state, the DataWriter reads in from SRAM the current value of data at the address we want to write and modify data at. This is necessary since in some drawing modes data is packed more than one pixel per SRAM address. In the writeData state the DataWriter takes in the previously read data and modifies it with the data that the drawing module requested to be written and it writes this new data back into SRAM. Lastly, in the sendACK state, the DataWriter sends a signal back to the drawing module that the write has completed so the drawing module can continue on its operations.

The DataWriter is also responsible for refreshing the screen by listening for the coordinate request from the VGA controller.

4.4 Displaying to the VGA Monitor

To display to the VGA monitor a VGA controller written by Bruce Land is used to select which current pixel we want to read from SRAM and display on the screen. Based on the drawing mode in use and the pixel requested the data line is unpacked to get the correct pixel and then this fraction of data is repackaged and fed to the VGA red, green, and blue lines. The following code shows how each of the four drawing modes has their red, green, and blue color values unpacked from SRAM and repackaged based on the current Coord_X and Coord_Y (the outputs of the VGA controller) and then fed into the VGA color lines.

```
// Show SRAM on the VGA
//320x240 single frame
assign  mVGA_R1 = {SRAM_DQ[15:12], 6'b0} ;
assign  mVGA_G1 = {SRAM_DQ[11:8], 6'b0} ;
assign  mVGA_B1 = {SRAM_DQ[7:4], 6'b0} ;
//640x480 - single frame
assign  mVGA_R2 = (Coord_X[0]) ? {SRAM_DQ[15:13], SRAM_DQ[15:13],SRAM_DQ[15:13],1'b1} :
{SRAM_DQ[7:5], SRAM_DQ[7:5],SRAM_DQ[7:5],1'b1} ;
```

```
assign  mVGA_G2 = (Coord_X[0]) ? {SRAM_DQ[12:10], SRAM_DQ[12:10],SRAM_DQ[12:10],1'b1} :
{SRAM_DQ[4:2], SRAM_DQ[4:2],SRAM_DQ[4:2],1'b1};
assign  mVGA_B2 = (Coord_X[0]) ? {SRAM_DQ[9:8],
SRAM_DQ[9:8],SRAM_DQ[9:8],SRAM_DQ[9:8],SRAM_DQ[9:8]}  : {SRAM_DQ[1:0],
SRAM_DQ[1:0],SRAM_DQ[1:0],SRAM_DQ[1:0],SRAM_DQ[1:0]};
//320x240 - double frame
assign  mVGA_R3 = ~readFrame ? {SRAM_DQ[15:13], SRAM_DQ[15:13],SRAM_DQ[15:13],1'b1} :
{SRAM_DQ[7:5], SRAM_DQ[7:5],SRAM_DQ[7:5],1'b1} ;
assign  mVGA_G3 = ~readFrame ? {SRAM_DQ[12:10], SRAM_DQ[12:10],SRAM_DQ[12:10],1'b1} :
{SRAM_DQ[4:2], SRAM_DQ[4:2],SRAM_DQ[4:2],1'b1};
assign  mVGA_B3 = ~readFrame ? {SRAM_DQ[9:8],
SRAM_DQ[9:8],SRAM_DQ[9:8],SRAM_DQ[9:8],SRAM_DQ[9:8]}  : {SRAM_DQ[1:0],
SRAM_DQ[1:0],SRAM_DQ[1:0],SRAM_DQ[1:0],SRAM_DQ[1:0]};
//640x480 - double frame
assign mVGA_R4 = ~Coord_X[0] ? (readFrame ? ({10{SRAM_DQ[7]}}) : ({10{SRAM_DQ[3]}})) : (readFrame
? ({10{SRAM_DQ[11]}}) : ({10{SRAM_DQ[15]}}));
assign mVGA_G4 = ~Coord_X[0] ? (readFrame ? ({5{SRAM_DQ[6:5]}}) : ({5{SRAM_DQ[2:1]}})) :
(readFrame ? ({5{SRAM_DQ[10:9]}}) : ({5{SRAM_DQ[14:13]}}));
assign mVGA_B4 = ~Coord_X[0] ? (readFrame ? ({10{SRAM_DQ[4]}}) : ({10{SRAM_DQ[0]}})) : (readFrame
? ({10{SRAM_DQ[8]}}) : ({10{SRAM_DQ[12]}}));

assign mVGA_R = (drawMode == sixteenBit_single_320x240) ? mVGA_R1:
                              (drawMode == eightBit_single_640x480)        ? mVGA_R2:
                              (drawMode == eightBit_double_320x240)        ? mVGA_R3:
                              (drawMode == fourBit_double_640x480)  ? mVGA_R4: 10'd0;


assign mVGA_G = (drawMode == sixteenBit_single_320x240) ? mVGA_G1:
                              (drawMode == eightBit_single_640x480)        ? mVGA_G2:
                              (drawMode == eightBit_double_320x240)        ? mVGA_G3:
                              (drawMode == fourBit_double_640x480)  ? mVGA_G4: 10'd0;


assign mVGA_B = (drawMode == sixteenBit_single_320x240) ? mVGA_B1:
                              (drawMode == eightBit_single_640x480)        ? mVGA_B2:
                              (drawMode == eightBit_double_320x240)        ? mVGA_B3:
                              (drawMode == fourBit_double_640x480)  ? mVGA_B4: 10'd0;
```

4.5 Drawing Modules

All drawing modules follow the same basic "protocol" in terms of what data they receive, how they process and what they output.

Common to all modules are that they receive an initial (X,Y) coordinate starting point and they receive some sort of clarifier point or points. The clarifier could be a width and height for rectangles, a radius for the circles, and end (X,Y) point for a line, etc. They also take in color data for what color they need to draw presently. Each drawing module, as mentioned earlier, also has its own ID to identify itself. Also common to all modules are that they output an (X,Y) address for drawing as well as a data value to write to that (X,Y) address. The drawing modules output a signal to notify if they are currently available or not. Lastly, the drawing modules output a write clock for use of the DataWriter and take in a flag from the DataWriter indicating if a write has finished. See the appendix for further description on all inputs and outputs of each drawing module.

Below is a presentation of the different drawing algorithms.

4.5-1 Rectangle Fills and Draws

The rectangle fill works by starting at the starting coordinate (X,Y) and iterating across X and down Y until the coordinate (X+width,Y+height) is reached. Each pixel is written sequentially. In hindsight I realized I could have designed the 640x480 version to fill two pixels at once since it is double packed and that would cut fill completion time in half for that scenario of single buffered 640x480.

The rectangle draw works by iterating over each wall of the rectangle, north, south, east, and west.

These fills only support a single color fill, however an extension to the modules is possible if one were to use one of the extra fields of the instruction to specify an end color and a gradient effect could be achieved with minor modifications to the color data output of the modules.

4.5-2 Circle Fills and Draws

Circles are drawn using the Bresenham circle drawing algorithm which breaks up the circle into eight quadrants in which the edges of the circle can be formed. Then using symmetry it loops through the algorithm to form the exterior of the circle.

To fill a circle, the 8 different quadrants are paired to their symmetric partner across the vertical axis. The partner on the left specifies a start point, and since Bresenham is symmetric, the other partner specifies an end coordinate that is on the same Y line. Then the algorithm iterates from (X_{p1}, Y) to (X_{p2}, Y), where p1 and p2 are the partner pairing of the quadrants.

Originally I attempted to have the circle fill module output an (X_1, Y_1) and (X_2, Y_2) points to the line drawing module, but this made things unnecessarily complicated and introduced numerous communication and timing constraints and problems. Eventually I realized that because the quadrants were paired on the same horizontal, a simple loop would be sufficient for the fills.

4.5-3 Line Drawing

Lines are drawn using the integer Breshenham line drawing algorithm. This module was modified for this project and adapted from a laboratory exercise in ECE 5760. Refer to the appendix for pseudo-code of the line drawing algorithm.

4.5-4 Text Writing

The TextWriter module is responsible for writing characters to the string. A .MIF file containing 256 ASCII 8x8 characters is downloaded into ROM. The TextWriter addresses into this ROM based on which character the instruction specifies. It then loops across the 8x8 character which consists of 1's and 0's. A 1 indicates to draw a pixel, and a 0 indicates to ignore.

Future modification to this file could have the instruction specify the type of font being used and instead of looping across 8x8, it loops across WxH where W is the width and H is the height of the font in use.

4.5-5 Sprite Drawing (in progress)

Drawing into a sprite was attempted in this project. At the current time of this report that work is still in progress. The way it works is that a sprite is represented by a 32x32 pixel, 8 bit color RAM module. A drawing instruction that comes in specifies that you want to draw into a certain sprite and instead of feeding the X,Y coordinate and data into the DataWriter module, it muxes the information into the Sprite RAM. To draw the sprite on the screen the sprite must be enabled and have a valid (X,Y) screen coordinate. Then when the VGA controller is looping through the coordinates to display on the monitor, the sprite module checks to see if its boundaries corresponds to that pixel and if it does, it immediately draws on the screen instead of grabbing from SRAM.

4.6 Instruction Download and GPU Processor Mode

An original attempt of the project was to provide a processor to program and download the instructions directly into an instruction queue of the GPU. The idea was to eliminate the serial overhead, except for the one time downloading of instructions, and have the entire program run locally on the GPU, similar to the nVidia CUDA architecture. However, issues with the normal drawing methods delayed the start of this part of the project, and at the time of the report it is only partially implemented and not fully tested. However, an explanation of what was done follows.

The processor that was created was a simple single cycle microprocessor that supported functions such as branches, ALU ops, jumps, loads, stores. It had an IRAM and DRAM as well. A full list of all instructions is included in the appendix.

When the RequestManager received a request it is decided whether this is a real-time request (should be directly drawn on the screen) or if this is a processor specific request, such as an instruction download. If it is a download, the RequestManager loads the current request into IRAM at a PC specified by the instruction. The value of this PC was supposed to come from a “compiler” that I would have written. The RequestManager also listens for a start processor command and when it receives that it enables to GPU Processor mode and goes to the PC = 0 to start the downloaded program. Execution then continues within the processor until the user specified an exit point, if there was one.

The processor itself is a standard single cycle microprocessor with some modifications to adapt to the GPU’s drawing functions. The processor maintains its own PC and outputs that to IRAM which returns the instruction back into the processor. The processor then decodes the instruction. If it decides that the instruction is a non-drawing function, it continues to the ALU operation stage and then into the memory and writeback stage. However, if it is a drawing operation a flag is thrown to indicate that we need to stall the processor and execute a drawing instruction. The information for this instruction is then created and sent to the direct drawing method in which it is executed. At the end of execution a flag is returned to the processor indicated that all drawing has completed. The GPU processor is then enabled and continues on its operations.

A visual representation of this process is included in the appendix by the name of Schematic #4.

4.7 Graphics Library in C

A graphics library has been written in the C programming language to facilitate communication with the GPU over serial. It provides instructions for drawing each of the type of graphics functions and it packages up the instructions into the appropriate bytes and is able to send them out over serial. The only modification one needs to make to any of these files would be if the serial communication is different for your specific process. Mine uses methods like writeByte, where one may want to use something like fprintf instead. These are trivial changes and only need to be changed in one line of code in the send instruction method. A list of instructions is included in the appendix along with an explanation of how they work. A simple example of how to draw something to the GPU is as follows:

```
setDrawMode(M8bit_SINGLE_640x480);
int red;
int green;
int blue;

fillRectangle(0,0,640,480,0,15,0);

for(green = 0; green < 15; green ++)
        for(red = 0; red < 15; red ++)
                drawLine(green*15+red,green*15+red,0,240,red, green, 0);

for(green = 0; green < 4; green ++)
        for(blue = 0; blue < 15; blue ++)
                drawLine(green*15+blue+255,green*15+blue+255,0,240,0,green,blue);

drawString(100-8*6,140,"Hello World!",0,0,0);
```

This piece of code first sets the type of drawing mode that we want the GPU to run, and that is a single buffer 640x480 image. It then does a sequence of rectangular fills, line draws, and text writing. What is nice about the code is that as long as the communication is set up in the underlying graphics library, the user only has to worry about simple functions such as fillRectangle in their main application code.

4.8 Timing Issues

One of the major challenges of this project was numerous timing issues that I had to deal with. Almost every single module was clocked and not every module used the same clock to process its data. The serial modules ran on a clock equal to the baudrate. The drawing functions ran on the VGA control clock that is used to drive the VGA controller. Other clock signals were generated artificially from within some modules, such as the signal to grab a packet, or send a N/ACK to the transmitter. One more factor to complicate things is that the VGA controller needs to read from SRAM to refresh the screen, and when this is happening, no processing can be done in the form of writing to SRAM from one of the drawing modules. This loss of synch causes some modules to pause, while others need not pause because they don’t depend on VGA synch complications. This skew in some modules needing to pause while others were free to run caused some initial problems in having every module communicate smoothly. This is because sometimes an artificial clock signal would be lost during a loss of synch. These issues were resolved through careful debugging and testing of the code to find complications of this type. One thing that this project certainly helped me understand is how important meeting timing requirements and understanding the timing of a system is to system performance.

5. Results and Conclusions

5.1 Original Goal Completion

Of the original goals of the project this GPU was successful in implementation of the following:

- Multiple resolution support, 320x240 and 640x480
- Color depth support, at least 16 colors in worst case palette
- Two frame buffers
- Multiple drawing functions, lines, circles, rectangles, text, and polygons
- Font support, 1 8x8 downloadable font
- 10 FPS, successful in some applications; is application dependent

Of the original goals of the project this GPU was partially successful in implementation of the following:

- Sprite support, functionality was created but not finished
- Programmable Interface, the processor was created but not fully tested and implemented

5.2 Testing the GPU

The GPU performed a multitude of tests to check for accuracy of drawing algorithms, communication of instructions, and to measure overall performance (such as IPS and FPS).

5.2-1 Accuracy of Drawing Algorithms

Tests were done in each of the four drawing modes on each of the different drawing modules. The criterion in testing was to see if the modules drew their function correctly. For fills this meant that every interior point was filled, and for the text writer, this meant that the proper character was written to the string. These tests also checked that the objects were drawn at the correct location on the screen.

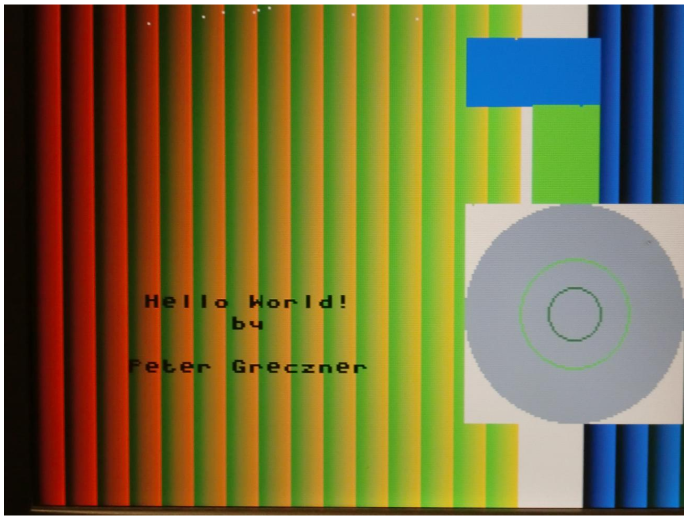


Figure – 320x240 Accuracy of Drawing Algorithm Test

There were two functions that did not work perfectly and a solution could not be found for these functions. The first is that the circle and rectangle fills would occasionally miss one pixel to draw. The likely explanation for this problem is a time/synchronization problem that results in one pixel not being written to SRAM. The other problem was with the text writing module which would occasionally write errant pixels to the screen. This problem was caused by the text writing module reading lines with a one cycle lag from the request. This would cause the last line of the previous character to be the first line of the next character. No problems occur when the last line is blank, but for a hanging letter such as ‘g’ or ‘j’, a bar would be drawn on top of the following character. This problem was resolved by introducing an extra line request when the command is sent from the RequestManager.

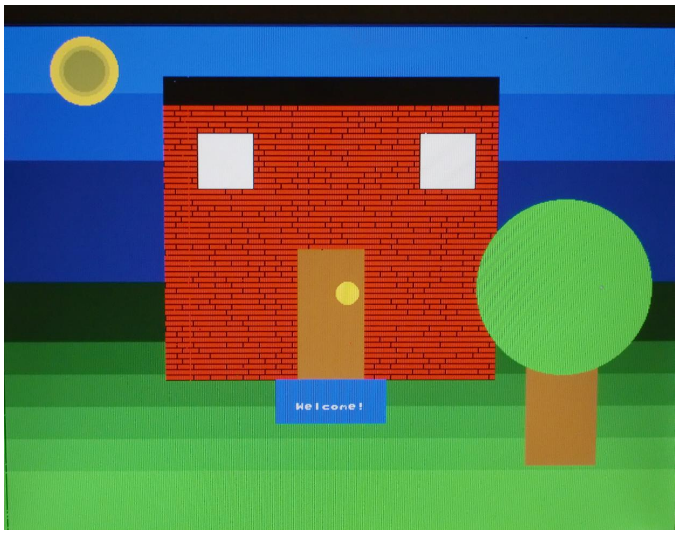


Figure – Scene created in 640x480 resolution to show off all drawing functions

5.2-2 Performance Tests

A few performance tests were run to see the speed at which the GPU could execute instructions. The theoretical max for best performance was 1047.3 instructions per second (IPS). To see what the real IPS is for the GPU the following tests were run and execution time was measured in all cases:

- 10,000 Lines – 10,000 random lines were drawn, used to represent where one might be drawing a lot of polygons
- Full Screen Fill – 100 full screen rectangle fills were performed, used to represent most extreme of drawing cases
- Random Fill – 250 rectangles of random size were filled on the screen, use to represent a high stress drawing application that constantly erasing parts of the screen
- Small Rectangles and Lines – 1000 rectangles and lines were drawn on the screen, use to represent an average type of drawing someone might do
- Point Draw – 5000 single point instructions, used to represent the least intensive of actual drawing instructions

- Frame Update – 5000 frame updates were performed, used to represent the equivalent of a NOP to find the real peak IPS for this GPU

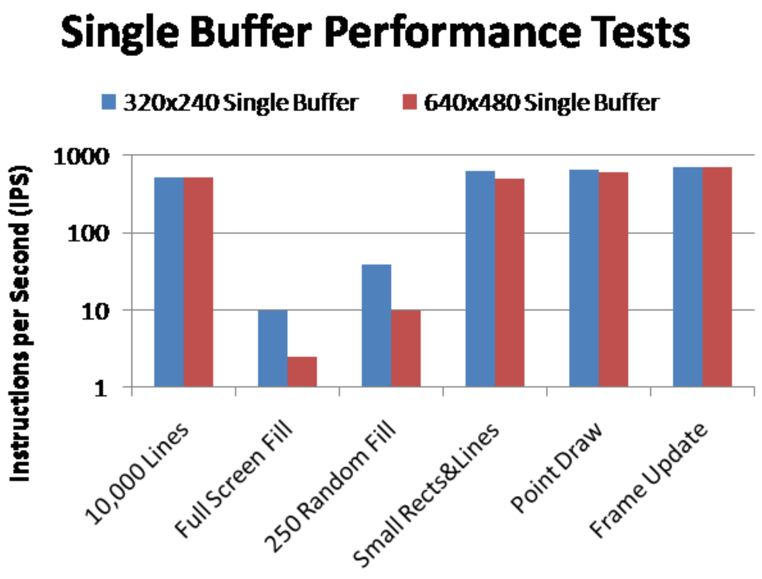


Figure – Shows the performance in terms of IPS for Single Buffer tests

The result of the test show that in most cases the IPS for the GPU peaks around the 800 IPS range. The outliers are the computationally heavy in terms of memory write tests of full screen rectangle fills and many random rectangle fills. However in the tests used to represent what an aggregate application might use, such as the Small Rects and lines, a high IPS was achieved.

To test the double buffered performance two simple tests were run, one in which a single bouncing ball was updated, and the second in which ten bouncing balls were updated and refreshed on the screen.

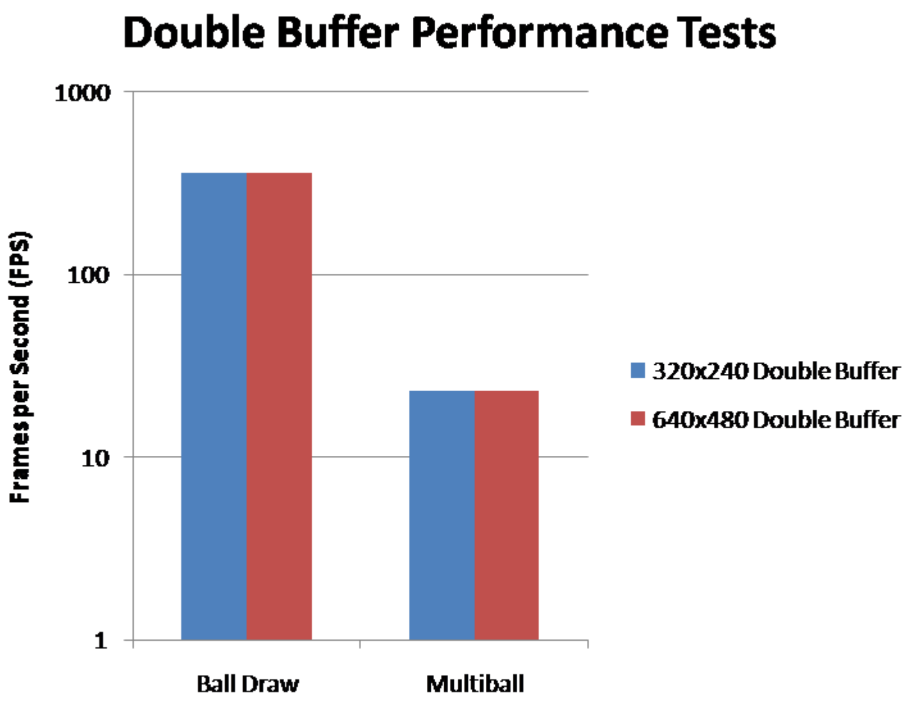


Figure – Double buffer FPS test

The result was that in the case of 10 balls bouncing on the screen an FPS of around 10 could be achieved. This was within the original goal of the project.

5.3 Conclusions

The GPU was extremely successful in completing many of the goals of the original proposal and made good progress towards completion of the last few goals. I believe that the current state of the GPU would work best for low-latency applications or those in which full screen fills do not occur often. Since IPS peaks around 800, this means that your FPS could significantly vary depending on the type of application you want to write. If your application requires 1000 instructions per frame to be executed, then it simply will not be possible to achieve an FPS better than 1, and hopefully the application is low-latency refresh, such as just printing data to the monitor. However, if your application is very simple, or you implement your graphics algorithm in an intelligent way (i.e. only refreshing areas that have been written too, instead of the whole screen) a high FPS is possible. Basically, the GPU cannot guarantee a certain minimum FPS as I had originally intended it too due to serial overhead limitations.

However, I am not sure that that is a bad thing at all. Leaving the design up to the user to decide how to best implement the graphics is not a bad component of the project. Also, although the GPU cannot guarantee an FPS of 10, that does not mean that an FPS of 10 is not possible; it certainly is.

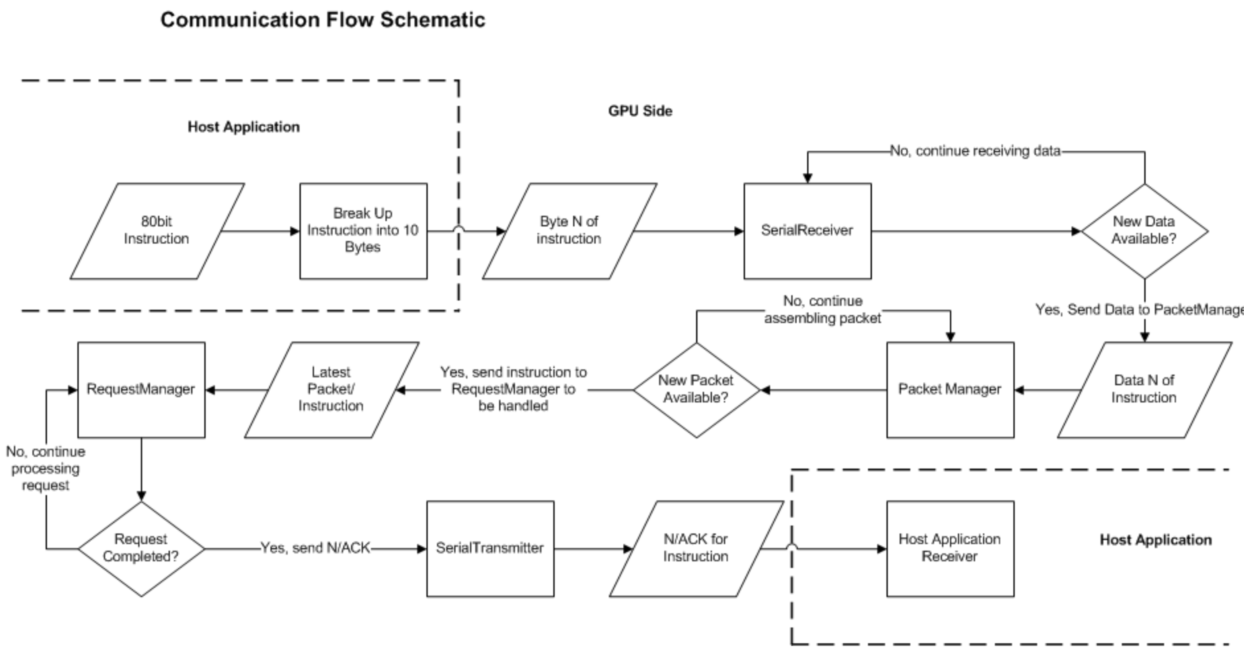
What I also believe works well in this project is the room for extension. The modules of the project were created in a way that could allow for easy future expansion of new drawing functions. From the beginning in which I added the two Extra 1 and Extra 2 fields should new functions arise, and the general extensible nature of the drawing modules in which they all follow a specific protocol, can make it easy for someone to take this project and make even more functionality.

Ultimately, this project did not implement the instruction queue and processor functionality to completion. Had that processor gotten to the working stage I truly believe the results would have been some high quality and high speed graphics. That isn't to take away from the serial sequential method which performs well, but future work in implementing the programmable processor could make this GPU an even more attractive option.

This project set out to realize a two-dimensional graphics card on an FPGA that could be used as a GPU for host devices such as microcontrollers in order to provide those applications with color graphics on a VGA compatible monitor. This goal was met.

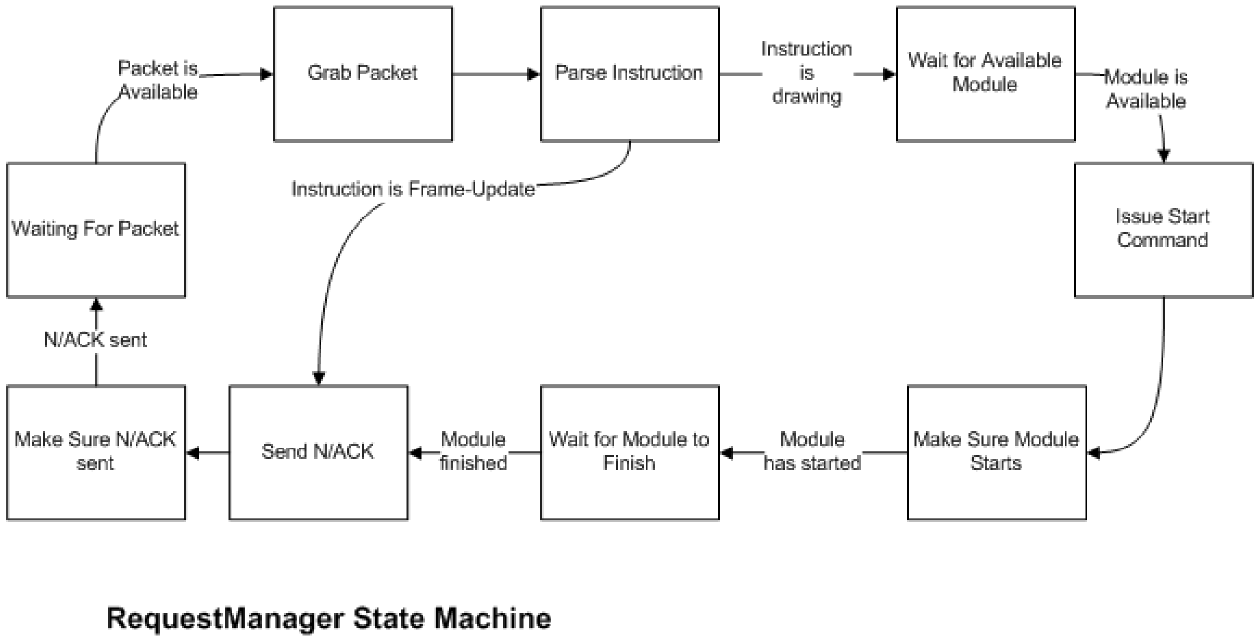
6. Appendix

6.1 Schematics

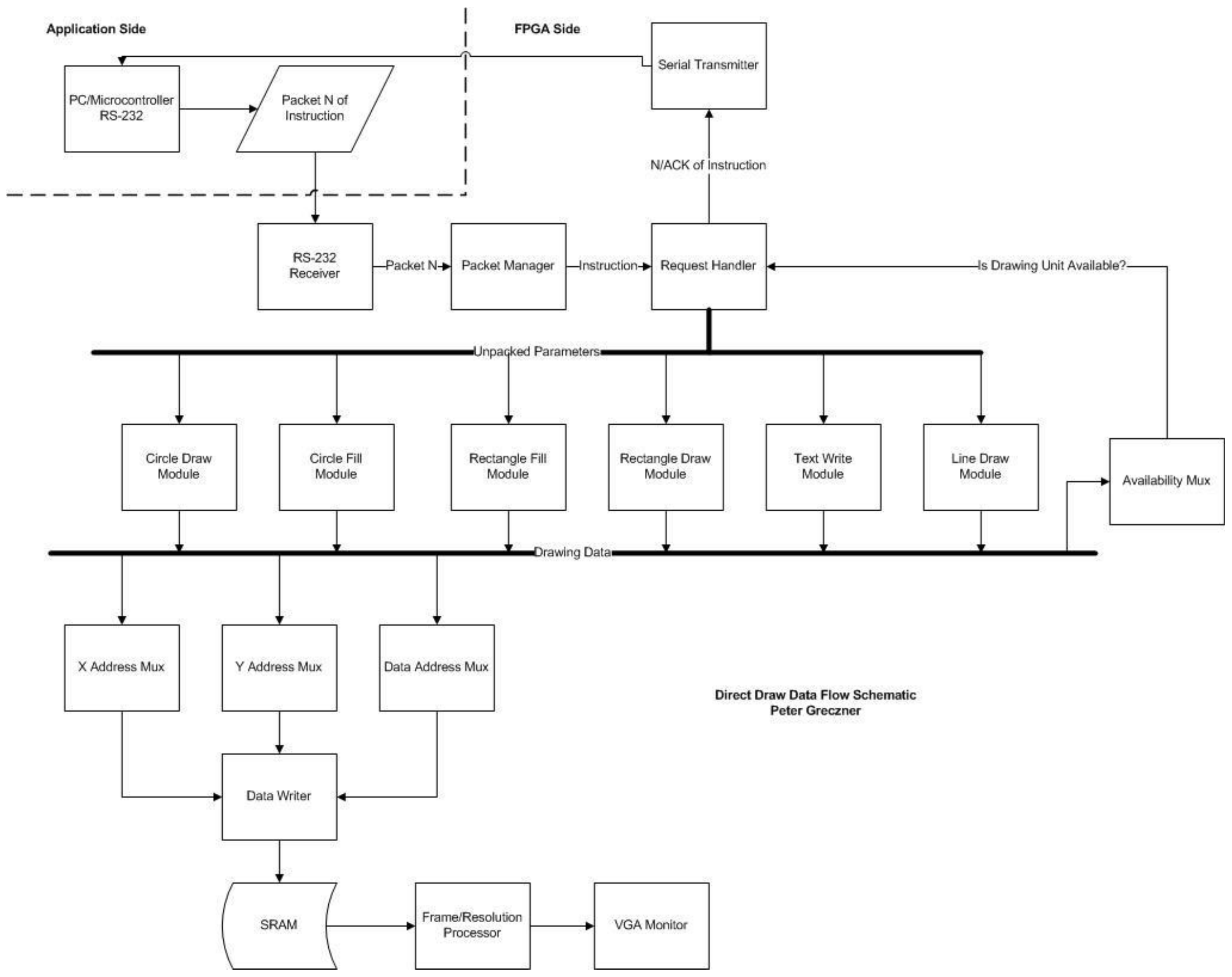


Communication Flow Schematic
Host Application
GPU Side
80bit Instruction
Break Up Instruction into 10 Bytes
Byte N of instruction
SerialReceiver
No, continue receiving data
New Data Available?
Yes, Send Data to PacketManager
Data N of Instruction
Packet Manager
No, continue assembling packet
New Packet Available?
Yes, send instruction to RequestManager to be handled
Latest Packet/ Instruction
RequestManager
No, continue processing request
Request Completed?
Yes, send N/ACK
SerialTransmitter
N/ACK for Instruction
Host Application Receiver
Host Application

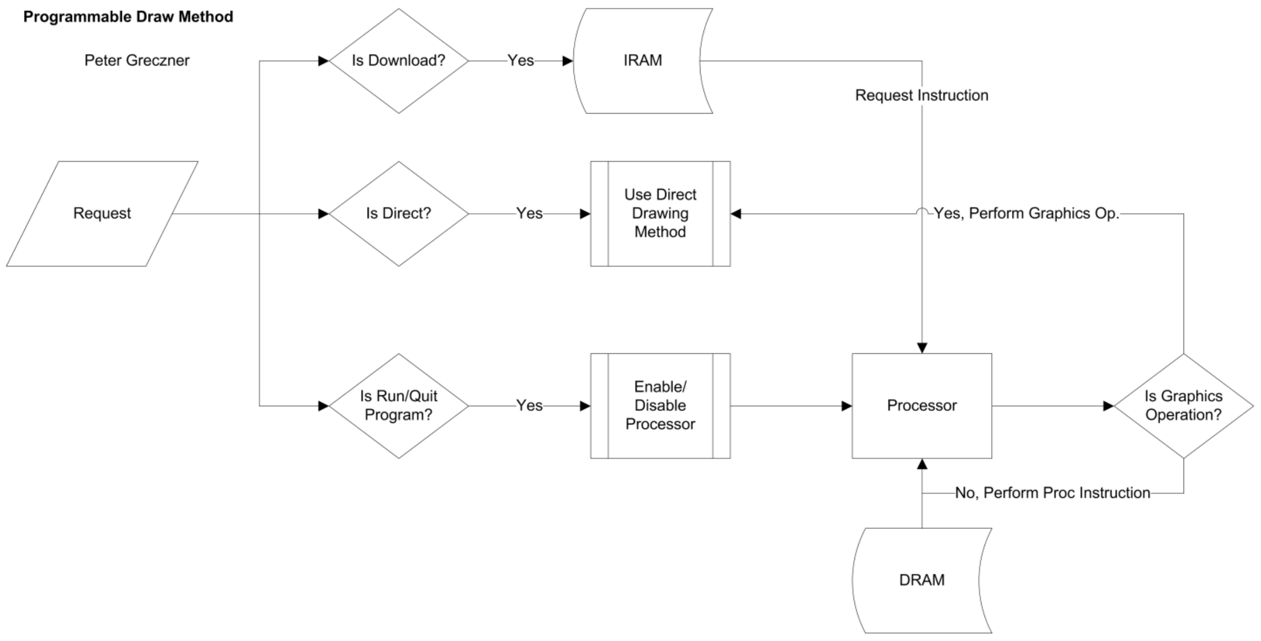
Schematic #1 – Overview of the communication between host and GPU



Schematic #2 – State diagram of how the RequestManager handles a request

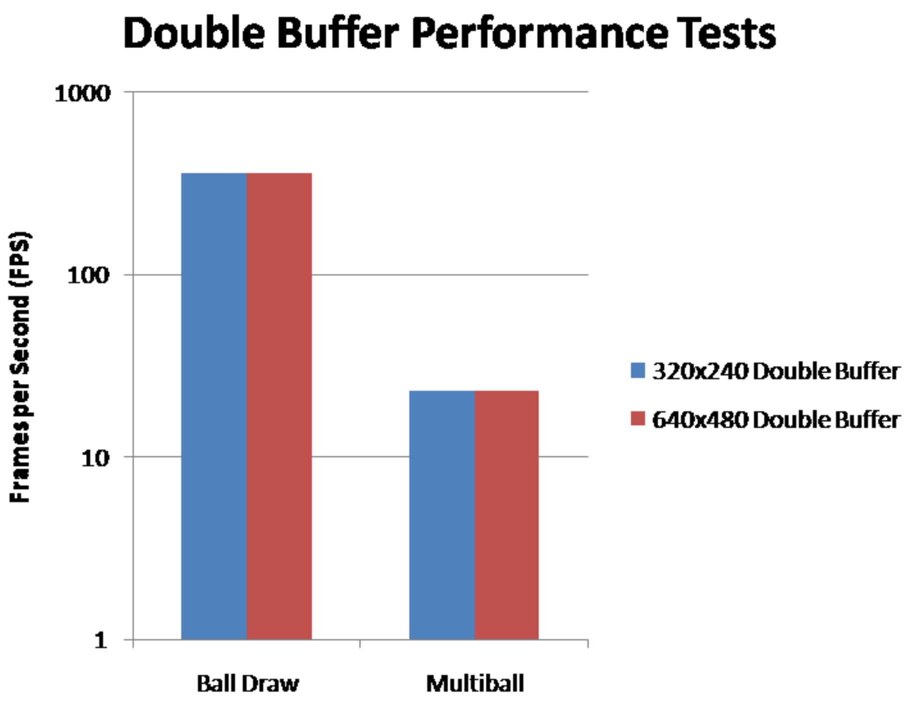


Schematic #3 – Direct Draw method schematic showing the general data flow from host to GPU to VGA

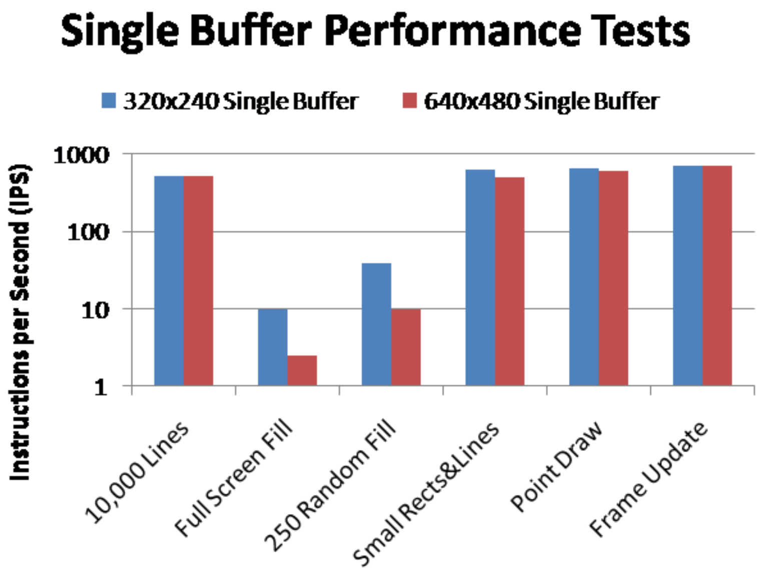


Schematic #4 – The programmable flow method data flow schematic

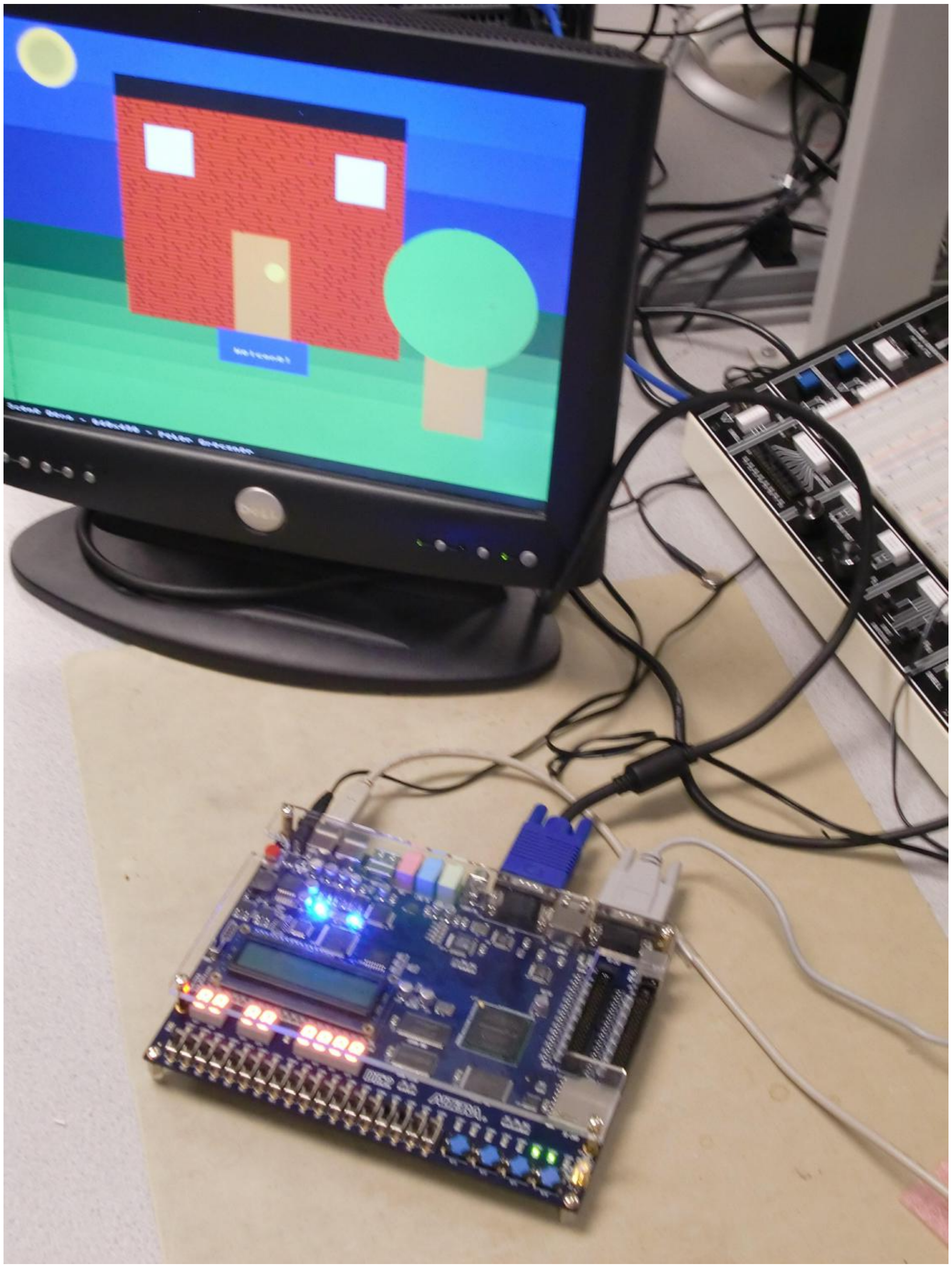
6.2 Images and Results



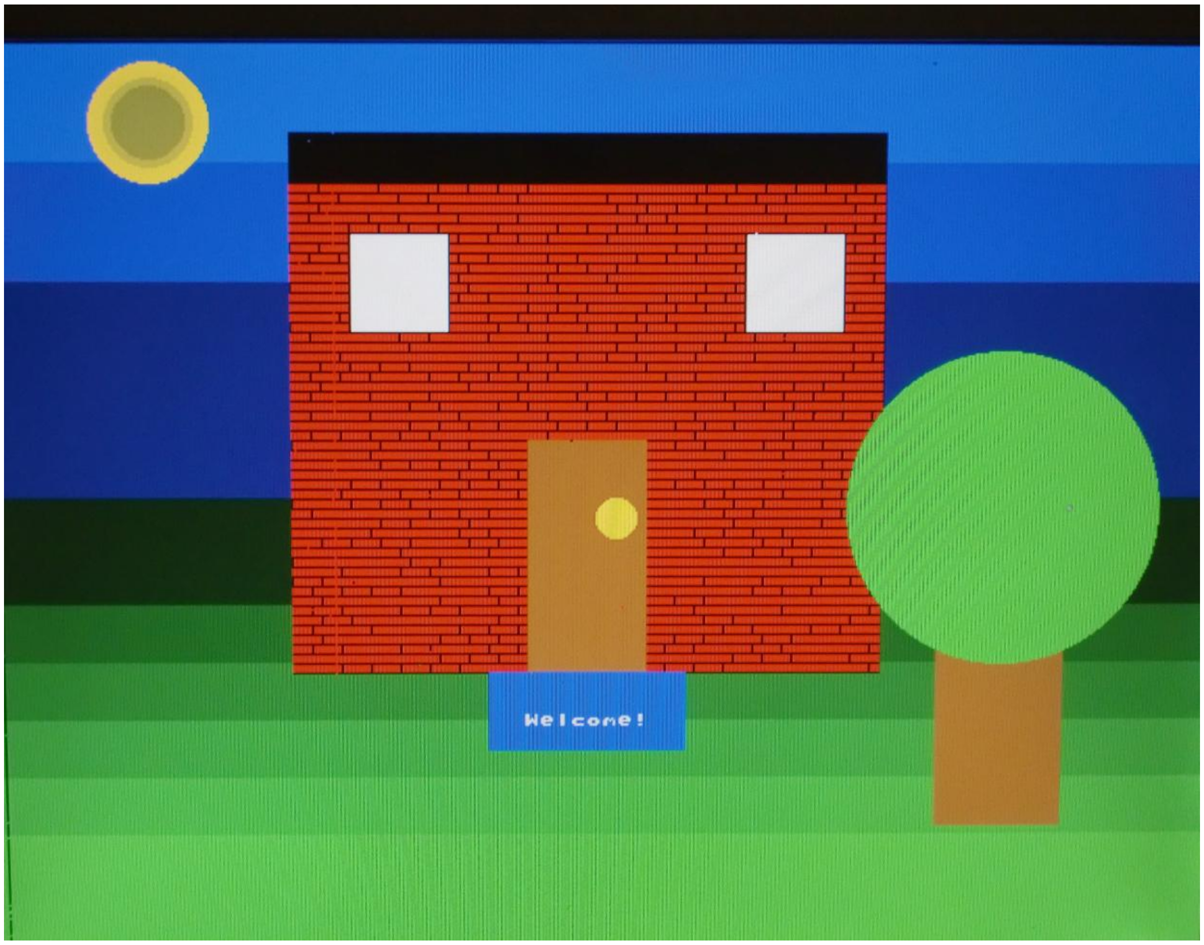
Double Buffer Performance Tests



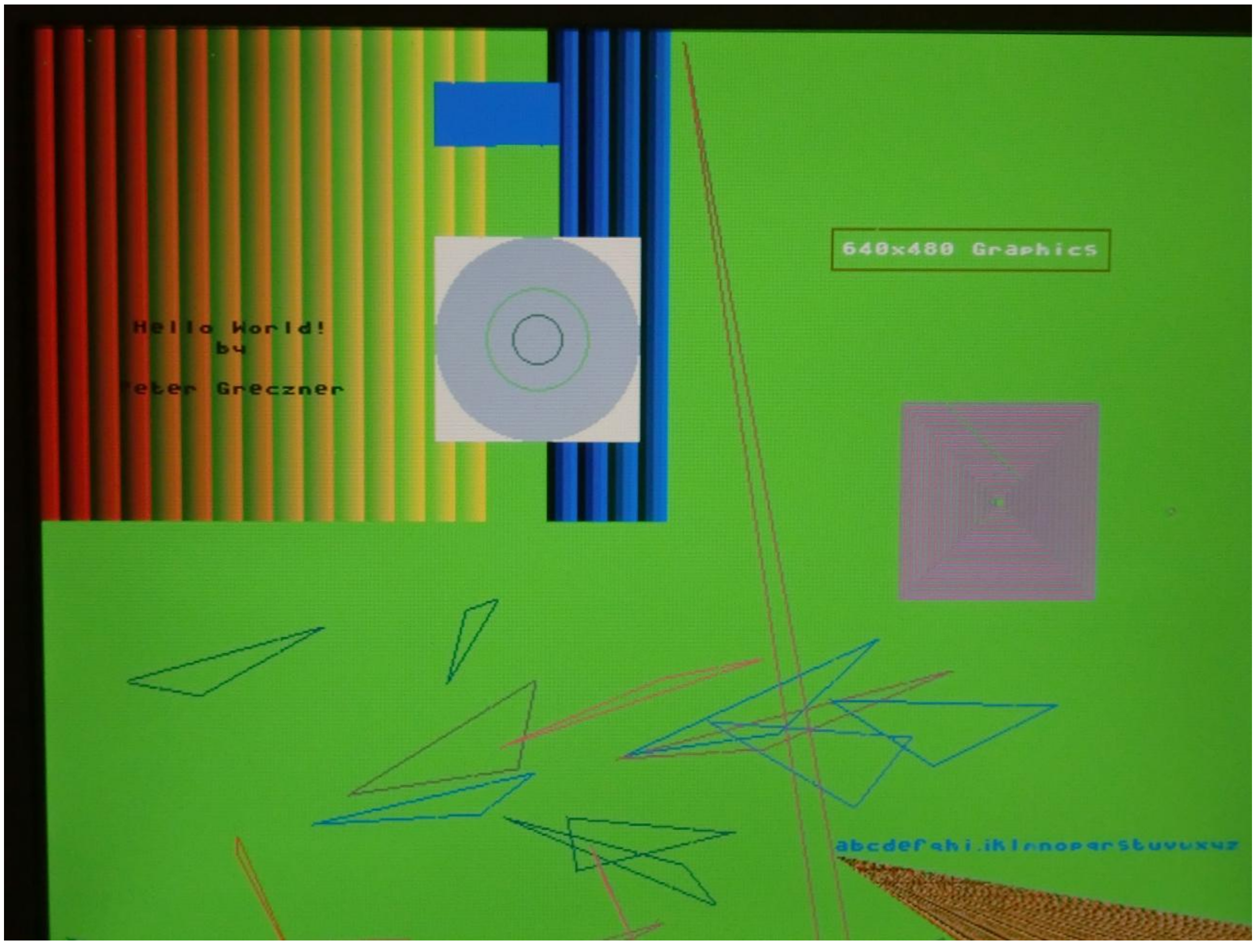
Single Buffer Performance Tests



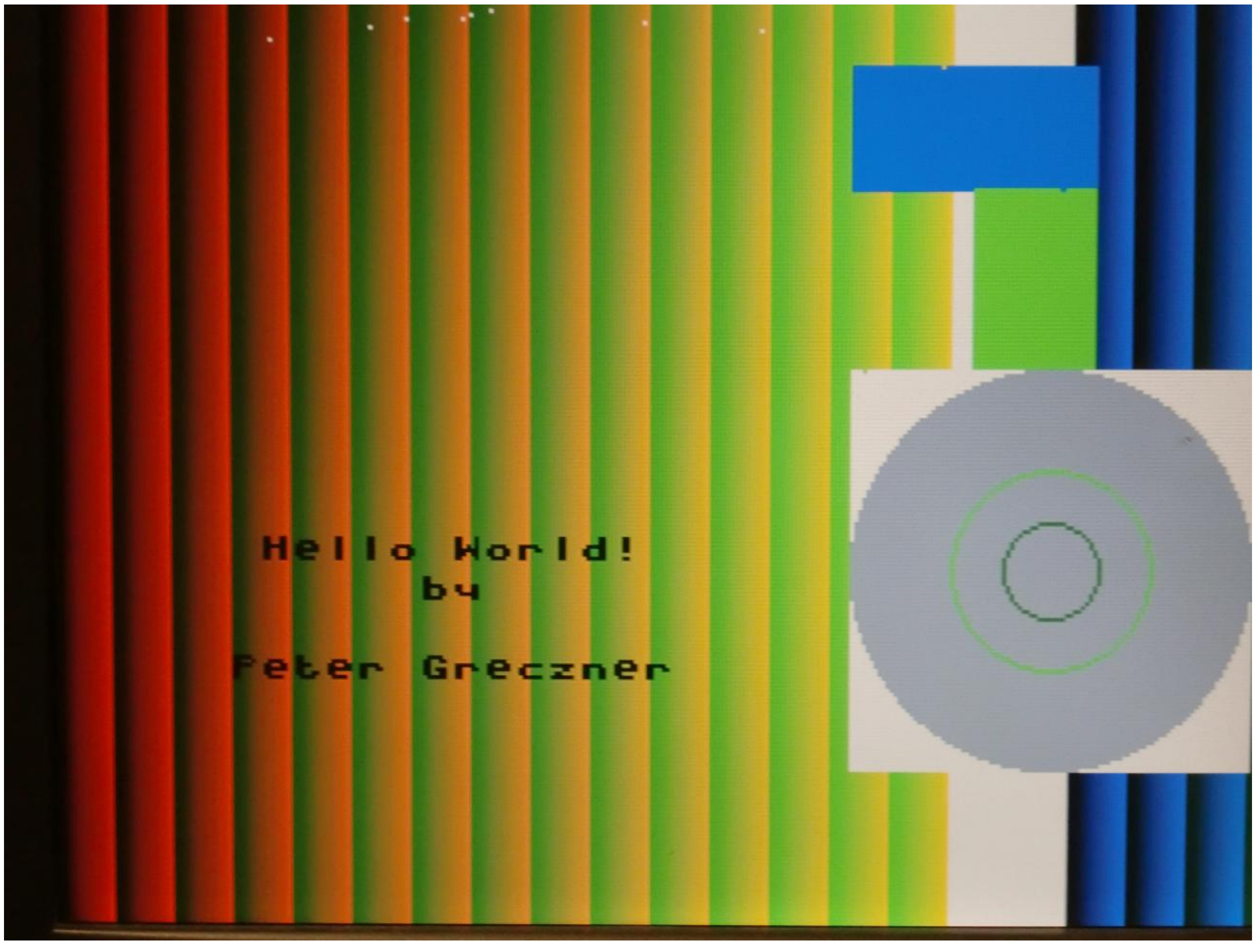
GPU running on FPGA and outputting to VGA monitor



640x480 Scene Test



640x480 Drawing function Test



320x240 Drawing Function Test

6.3 Module Descriptions

- *availableMux.v*: muxes the availability bits of all the drawing modules based on a select bit from the RequestManager
- *BaudGenerator.v:* generates the baud clock for the serial communication
- *BaudGenerator_8Times.*v: generates a baud clock at 8 times the required speed
- *circle_draw.v:* draws a circle on the screen
- *circle_fill_better.*v: fills a circle on the screen
- *Color_MUX*.v: muxes all of the color data from the drawing modules based on a select bit from the RequestManager
- *CRC.v:* computes the XOR of the packet to create a CRC
- *DataWriter.v*: writes the current data to SRAM
- *font1_rom*.v: the 8x8 font ROM
- *line_drawer*.v: draws a line on the screen
- *PacketManager_updated*.v: the packet manager that collects the instruction as it receives the individual parts from the SerialReceiver

- *RequestManager_updated*.v: manages all the requests that come in from the host device and processes them accordingly
- *SerialReceiver_improved_fastClock*.v: the serial receiver module
- *SerialTransmitter_updated*.v: the serial transmitter module
- *square_draw*.v: draws a square on the screen
- *square_fill.v:* fills a square on the screen
- *TextWriter.v:* writes text to the screen
- *TransmitClock.v:* artificial clock used to grab a packet
- *Xaddr_MUX.v:* mux for the x address data that muxes based on a select bit from the RequestManager
- *Yaddr_MUX.v:* mux for the yaddress data that muxes based on a select bit from the RequestManager

6.4 Software User Guide

6.4-1 Function Library and Description

The following functions have been created in the C graphics library:

void fillRectangle(int x, int y, int w, int h, int r, int g, int b);
Use: fills a rectangle on the screen with the specified color, width, and height at (x,y)
Parameters:
x = starting x coordinate
y = starting y coordinate
w = width of rectangle
h = height of rectangle
r = red color (0-15)
g = green color (0-15)
b = blue color (0-15)

void fillCircle(int x, int y, int r, int red, int green, int blue);
Use: fills a circle on the screen with the specified color, and radius at (x,y)
Parameters:
x = starting x coordinate
y = starting y coordinate
r = radius
red = red color (0-15)
green = green color (0-15)
blue = blue color (0-15)

void drawRectangle(int x, int y, int w, int h, int r, int g, int b);
Use: draws a rectangle on the screen with the specified color, width, and height at (x,y)
Parameters:

x = starting x coordinate
y = starting y coordinate
w = width of rectangle
h = height of rectangle
r = red color (0-15)
g = green color (0-15)
b = blue color (0-15)

void drawCircle(int x, int y, int r, int red, int green, int blue);
Use: draws a circle on the screen with the specified color, and radius at (x,y)
Parameters:
x = starting x coordinate
y = starting y coordinate
r = radius
red = red color (0-15)
green = green color (0-15)
blue = blue color (0-15)

void drawLine(int x1, int x2, int y1, int y2, int r, int g, int b);
Use: draws a line from (x1,y1) to (x2,y2) with the specified color
Parameters:
x1 = point 1 of line x coordinate
x2 = point 2 of line x coordinate
y1 = point 1 of line y coordinate
y2 = point 2 of line y coordinate
r = red color (0-15)
g = green color (0-15)
b = blue color (0-15)

void drawChar(int x1, int y1, int charVal, int r, int g, int b);
Use: draws a character on the screen at (x1,y1) with the specified color
Parameters:
x1 = starting x coordinate of character
y1 = starting y coordinate of character
charVal = the ascii character value (0-255)
r = red color (0-15)
g = green color (0-15)
b = blue color (0-15)

*void drawString(int x1, int y1, char *string, int r, int g, int b);*
Use: draws a string to the screen at (x1,y1) with specified color
Parameters:
x1 = starting x coordinate of string

y1 = starting y coordinate of string
*string = pointer to character array
r = red color (0-15)
g = green color (0-15)
b = blue color (0-15)

*void drawPolygon(int *x, int *y, int elems, int r, int g, int b);*
Use: draws a polygon on the screen with vertices *x and *y
Parameters:
*x = integer array of the x points of the polygon
*y = integer array of the y points of the polygon
elems = number of elements in *x and *y arrays
r = red color (0-15)
g = green color (0-15)
b = blue color (0-15)

void writeInstruction(int PC, char a, char b, char c, char d, char e, char f);
Use: to write an instruction into IRAM

void startProgram(void);
Use: to start execution of the IRAM program

void startDownload(void);
Use: to start downloading into IRAM

void finishDownload(void);
Use: to signal finish downloading into IRAM

void frameUpdate(void);
Use: to update to the current frame and switch to drawing into alternate frame

void setDrawMode(int dm);
Use: sets the drawing mode
Parameters:
dm = the drawing mode…
*M16bit_SINGLE_320x240: single buffer 320x240 image
*M8bit_SINGLE_640x480: single buffer 640x480 image
*M8bit_DOUBLE_320x240: double buffer 320x240 image
*M4bit_DOUBLE_640x480: double buffer 640x480 image

*void sendInstruction(int *instr);*
Use: sends an instruction over serial to the GPU
Parameters:
*instr = integer array of the instruction to be sent

*int computeChecksum(int *packet);*

Use: computes a checksum of the current instruction to be sent

Parameters:

*packet = current packet of instruction to be sent

6.4-2 Example Program

```
#include <stdio.h>
#include "rs232.h"
#include "globals.h"
#include "Graphics_Functions.h"
#include <time.h>

//This program draws a simple scene of a house in a field

int main(int argc, char **argv)
{

        int error;
        int test_num = -3;
        printf("args %i\n",argc);
        if(argc > 1)
                test_num = atoi(argv[1]);
        printf("Serial Communication in C\n");

        printf("Opening Comport: COM1\n");
        error = OpenComport(0, 115200);
        if(error) {
                printf("Failed!!! Exiting\n");
                return 0;
        }
        unsigned char result;
        result = 0;
        int gray = 1;
        while(gray == 1 && test_num != 24)
                gray = PollComport(0,&result,1); //to flush out initial data

    setDrawMode(M8bit_SINGLE_640x480);

    fillRectangle(0,0,640,480,0,0,0);
    int sky = 0;
    int skyc = 15;
    double dskyc = 1.0/230.0;
    for(sky = 1; sky < 230; sky ++)
    {
        if((sky % 15) == 0) skyc --;
        drawLine(0,640,sky,sky,0,0,max(skyc,4));
        printf("(int)skyc: %i\n",skyc);

    }

    int grass = 0;
    int grassc = 0;
    double dgrassc = 1.0/250.0;
    for(grass = 230; grass <= 479; grass ++){
        if((grass % 15) == 0) grassc ++;
        drawLine(0,640,grass,grass,0,max(grassc,2),0);

        }

    fillRectangle(320-150,320-250,300,250,12,0,0);
    int brick = 0;
    for(brick = 320; brick >= (320-250); brick -=5)
    {
        drawLine(170,470,brick,brick,0,0,0);
        int x;
        for(x = 170 + ((brick%2)*10); x < 470; x += 15)
```

```
            if((rand()/(float)RAND_MAX) <= .5)
            {
                if(brick <= 315)
                {
                    drawLine(x,x,brick,brick+5,0,0,0);
                }
            }
    }

    fillRectangle(320-150,320-250-25,300,25,0,0,0);
    fillRectangle(320-30,320-120,60,120,9,5,0);
    fillCircle(320+15,320-80,10,15,15,0);
    fillRectangle(320-120,320-225,50,50,15,15,15);
    drawRectangle(320-120,320-225,50,50,0,0,0);
    fillRectangle(320+80,320-225,50,50,15,15,15);
    drawRectangle(320+80,320-225,50,50,0,0,0);

    fillRectangle(500,400-165,65,165,9,5,0);
    fillCircle(533,400-165,80,0,12,0);

    fillRectangle(320-50,320,100,40,0,0,15);
    drawString(320-8*4,340,"Welcome!",15,15,8);


    int sun = 0;
    for(sun = 15; sun >= 8; sun --)
        fillCircle(100,40,sun*2,sun,sun,0);

    drawString(10,470,"Scene Demo - 640x480 - Peter Greczner",15,15,15);
}
```

6.5 Drawing Algorithm Reference

6.5-1 Bresenham Line Drawing Algorithm (pseudo-code)

```
function line(x0, x1, y0, y1)
     boolean steep := abs(y1 - y0) > abs(x1 - x0)
     if steep then
         swap(x0, y0)
         swap(x1, y1)
     if x0 > x1 then
         swap(x0, x1)
         swap(y0, y1)
     int deltax := x1 - x0
     int deltay := abs(y1 - y0)
     int error := deltax / 2
     int ystep
     int y := y0
     if y0 < y1 then ystep := 1 else ystep := -1
     for x from x0 to x1
         if steep then plot(y,x) else plot(x,y)
         error := error - deltay
         if error < 0 then
             y := y + ystep
             error := error + deltax
[1] http://en.wikipedia.org/wiki/Bresenham%27s_line_algorithm#Optimization
```

6.5-2 Bresenham Circle Drawing Algorithm (pseudo-code)

```
void circle(int xc, int yc, int r)
{
int x = 0;
int y = r;
int p = 3 - 2 * r;
   while (x <= y)
```

```
      {
         putpixel(xc + x, yc + y, getcolor());
         putpixel(xc - x, yc + y, getcolor());
         putpixel(xc + x, yc - y, getcolor());
         putpixel(xc - x, yc - y, getcolor());
         putpixel(xc + y, yc + x, getcolor());
         putpixel(xc - y, yc + x, getcolor());
         putpixel(xc + y, yc - x, getcolor());
         putpixel(xc - y, yc - x, getcolor());
         if (p < 0)
            p += 4 * x++ + 6;
         else
            p += 4 * (x++ - y--) + 10;
      }
   }
   [2] http://gamebub.com/cpp_algorithms.php#circle
```

6.6 Processor ISA Opcodes

Instruction Name	Op Code
Add	100000
Subtract	100001
Shift Left	100010
Shift Right	100011
OR	100100
NOR	100101
AND	100110
NAND	100111
XOR	101000
BGT	101001
BLT	101010
BGE	101011
BLE	101100
BEQ	101101
BNE	101110
BEZ	101111
BNEZ	110000
J	110001
JR	110010
LD	110011
ST	110100
Circle Fill	000000
Circle Draw	000001
Square Fill	000010
Square Draw	000011
Line Draw	000100
Text Write	000101
Frame Update	000110

ADDI	110101
SUBI	110110
SLI	110111
SRI	111000
NOT	111001
JAL	111010

6.7 Code Reference

```c
//globals.h

#define CIRCLE_FILL 0
#define CIRCLE_DRAW 1
#define SQUARE_FILL 2
#define SQUARE_DRAW 3
#define LINE_DRAW   4
#define TEXT_WRITE  5
#define FRAME_UPDATE 6
#define FRAME_SWITCH 6
#define DRAW_MODE 7
#define START_DOWNLOAD 8
#define FINISH_DOWNLOAD 15


//Drawing Modes
#define M16bit_SINGLE_320x240 0
#define M8bit_SINGLE_640x480 1
#define M8bit_DOUBLE_320x240 2
#define M4bit_DOUBLE_640x480 3

#define PACKET_LENGTH 10

#define DIRECT 0
#define LOAD 1
```

```c
//Graphics_Functions.c

#include "Graphics_Functions.h"
#include "globals.h"
#include "rs232.h"
#include <string.h>

int packet[] = {0,0,0,0,0,0,0,0,0};
int instruction[] = {0,0,0,0,0,0,0,0};

int frame_sel = 0;
int draw_mode = M16bit_SINGLE_320x240;


void fillRectangle(int x, int y, int w, int h, int r, int g, int b)
{
    instruction[0] = SQUARE_FILL;
    instruction[1] = r*256 + g*16 + b;
    instruction[2] = x;
    instruction[3] = y;
    instruction[4] = w;
    instruction[5] = h;
    instruction[6] = (draw_mode << 5) + (frame_sel << 3);
    instruction[7] = 0;

    sendInstruction(instruction);
}


void fillCircle(int x, int y, int r, int red, int green, int blue)
{
    instruction[0] = CIRCLE_FILL;
    instruction[1] = red*256 + green*16 + blue;
    instruction[2] = x;
    instruction[3] = y;
    instruction[4] = r;
    instruction[5] = 0;
    instruction[6] = (draw_mode << 5) + (frame_sel << 3);
    instruction[7] = 0;

    sendInstruction(instruction);
}


void drawRectangle(int x, int y, int w, int h, int r, int g, int b)
{
    instruction[0] = SQUARE_DRAW;
    instruction[1] = r*256 + g*16 + b;
    instruction[2] = x;
    instruction[3] = y;
    instruction[4] = w;
    instruction[5] = h;
    instruction[6] = (draw_mode << 5) + (frame_sel << 3);
    instruction[7] = 0;

    sendInstruction(instruction);
```

```
}


void drawCircle(int x, int y, int r, int red, int green, int blue)
{
    instruction[0] = CIRCLE_DRAW;
    instruction[1] = red*256 + green*16 + blue;
    instruction[2] = x;
    instruction[3] = y;
    instruction[4] = r;
    instruction[5] = 0;
    instruction[6] = (draw_mode << 5) + (frame_sel << 3);
    instruction[7] = 0;

    sendInstruction(instruction);
}


void drawLine(int x1, int x2, int y1, int y2, int r, int g, int b)
{
    instruction[0] = LINE_DRAW;
    instruction[1] = r*256 + g*16 + b;
    instruction[2] = x1;
    instruction[3] = y1;
    instruction[4] = x2;
    instruction[5] = y2;
    instruction[6] = (draw_mode << 5) + (frame_sel << 3);
    instruction[7] = 0;

    sendInstruction(instruction);
}

void drawPolygon(int *x, int *y, int elems, int r, int g, int b)
{
    int i;
    for(i = 1; i < elems; i ++)
    {
        drawLine(x[i],x[i-1],y[i],y[i-1], r, g, b);
        printf("drawing line from (%d,%d) to (%d,%d)\n",x[i],y[i],x[i-1],y[i-1]);
    }
    if(elems >= 2)
        drawLine(x[elems-1],x[0],y[elems-1],y[0], r, g, b);
    printf("drawing line from (%i,%i) to (%i,%i)\n",x[elems-1],y[elems-1],x[0],y[0]);
}

void drawChar(int x1, int y1, int charVal, int r, int g, int b)
{
    instruction[0] = TEXT_WRITE;
    instruction[1] = r*256 + g*16 + b;
    instruction[2] = x1;
    instruction[3] = y1;
    instruction[4] = charVal >> 2;
    instruction[5] = (charVal & 0x3) << 2;
    instruction[6] = (draw_mode << 5) + (frame_sel << 3);
    instruction[7] = 0;
    sendInstruction(instruction);
}

void writeInstruction(int PC, char a, char b, char c, char d, char e, char f)
{
    instruction[0] = a;
    instruction[1] = b;
    instruction[2] = c;
    instruction[3] = d;
    instruction[4] = e;
    instruction[5] = f;
    instruction[6] = PC >> 4;
    instruction[7] = (((PC&0xF) << 4) | 0x0F);
    sendInstruction(instruction);
}

void startProgram(void)
{
    int i;
    for(i = 0; i < 8; i ++) instruction[i] = 0xFF;
    sendInstruction(instruction);
}
void startDownload(void)
{
    instruction[0] = START_DOWNLOAD;
    sendInstruction(instruction);
}
void finishDownload(void)
{
    instruction[0] = FINISH_DOWNLOAD;
    sendInstruction(instruction);
}

void frameUpdate(void)
{
    instruction[0] = FRAME_SWITCH;
    sendInstruction(instruction);
    switch(draw_mode)
    {
        case M16bit_SINGLE_320x240:
            frame_sel = 0;
            break;
        case M8bit_SINGLE_640x480:
            frame_sel = 0;
            break;
        case M8bit_DOUBLE_320x240:
            frame_sel = (frame_sel == 0) ? 1 : 0;
            break;
        case M4bit_DOUBLE_640x480:
            frame_sel = (frame_sel == 0) ? 1 : 0;
            break;
    }
}

void setDrawMode(int dm)
{
    draw_mode = dm;
    instruction[0] = DRAW_MODE;
    instruction[6] = (draw_mode << 5) + (frame_sel << 3);
    //sendInstruction(instruction);
}

void drawString(int x1, int y1, char *s, int r, int g, int b)
{
    int i = 0;
    for(i = 0; i < strlen(s); i ++)
    {
        int charVal = s[i]*8 - 8*16;
        drawChar(x1+i*8,y1,charVal,r,g,b);
    }
}



void sendInstruction(int *instr)
{
    unsigned char result;

    int tries = 0;

    //int gray = 1;
    //while(gray == 1)
    //  gray = PollComport(0,&result,1); //to flush out initial data

    //do
    //printf("********::  %i   ::********\n",instr[4]);
    int i;
    {
        result = 0;
        packet[0] = ((instr[0] & 0x3F) << 2) | ((instr[1] & 0xC00) >> 10);
        packet[1] = ( instr[1] & 0x3FC) >> 2;
        packet[2] = ((instr[1] & 0x3) << 6) | ((instr[2] & 0x3F0) >> 4);
        packet[3] = ((instr[2] & 0xF) << 4) | ((instr[3] & 0x1E0) >> 5);
        packet[4] = ((instr[3] & 0x1F) << 3) | ((instr[4] & 0x380) >> 7);
        packet[5] = ((instr[4] & 0x7F) << 1) | ((instr[5] & 0x100) >> 8);
        packet[6] = ((instr[5] & 0xFF));
        packet[7] = instr[6] & 0xFF;
        packet[8] = instr[7] & 0xFF;
        packet[9] = 0;


        for(i = 0; i < (PACKET_LENGTH-1); i ++)
        {
            packet[PACKET_LENGTH-1] ^= packet[i];
            SendByte(0,(unsigned char)packet[i]);
            //printf("%i: %i\n",i,packet[i]);
        }

        SendByte(0,(unsigned char)packet[PACKET_LENGTH-1]);

        tries ++;
        int pollResult = 0;
        while(pollResult == 0){
            pollResult =PollComport(0,&result,1);
            //printf(".");
        }
        //printf("Got Ack: %d\n",result);
        //printf("tries: %d   result = %d, poll_res = %i\n",tries, result,pollResult);

    }
    /*int wait = 0;
    for(i = 0; i < 10000000; i ++)
        wait ++;
    i = wait;*/
    //while(result < 128);



}
```

```c
void fillRectangle(int x, int y, int w, int h, int r, int g, int b);
void fillCircle(int x, int y, int r, int red, int green, int blue);
void drawRectangle(int x, int y, int w, int h, int r, int g, int b);
void drawCircle(int x, int y, int r, int red, int green, int blue);
void drawLine(int x1, int x2, int y1, int y2, int r, int g, int b);
void drawChar(int x1, int y1, int charVal, int r, int g, int b);
void drawString(int x1, int y1, char *string, int r, int g, int b);
void drawPolygon(int *x, int *y, int elems, int r, int g, int b);
void writeInstruction(int PC, char a, char b, char c, char d, char e, char f);
void startProgram(void);
void startDownload(void);
void finishDownload(void);
void frameUpdate(void);
void setDrawMode(int dm);
void sendInstruction(int *instr);
int computeChecksum(int *packet);
```

```
//Peter Greczner, Instruction Decoder

module Decoder(instruction, ctrl_ALU, ctrl_BS, isBranch, isLoad, isStore, isJump, isDraw,
isIMMOpp, SA, SB, SD, IMM, WE);

input wire [79:0] instruction;

output reg [3:0] ctrl_ALU;
output reg [3:0] ctrl_BS;
output reg isBranch;
output reg isLoad;
output reg isStore;
output reg isJump;
output reg isDraw;
output reg isIMMOpp;

output reg [4:0] SA, SB, SD;
output reg [31:0] IMM;
output reg WE;

parameter   ADD = 6'b100000,
            SUB = 6'b100001,
            SL  = 6'b100010,
            SR  = 6'b100011,
            OR  = 6'b100100,
            NOR = 6'b100101,
            AND = 6'b100110,
            NAND= 6'b100111,
            XOR = 6'b101000,
            ADDI= 6'b110101,
            SUBI= 6'b110110,
            SLI = 6'b110111,
            SRI = 6'b111000,
            LD  = 6'b110011,
            ST  = 6'b110100,
            NOT = 6'b111001,
            BGT = 6'b101001,
            BLT = 6'b101010,
            BGE = 6'b101011,
            BLE = 6'b101100,
            BEQ = 6'b101101,
            BNE = 6'b101110,
            BEZ = 6'b101111,
            BNEZ= 6'b110000,
            J   = 6'b110001,
            JR  = 6'b110010,
            JAL = 6'b111010,
            CIRCLE_FILL = 6'b000000,
            CIRCLE_DRAW = 6'b000001,
            SQUARE_FILL = 6'b000010,
            SQUARE_DRAW = 6'b000011,
            LINE_DRAW = 6'b000100,
            TEXT_WRITE = 6'b000101,
            FRAME_UPDATE = 6'b000110;

parameter   BS_bgt = 4'd0,
            BS_blt = 4'd1,
            BS_bge = 4'd2,
            BS_ble = 4'd3,
            //BS_ble = 4'd4,
            BS_beq = 4'd5,
            BS_bne = 4'd6,
            BS_bez = 4'd7,
            BS_bnez = 4'd8,
            BS_j = 4'd9,
            BS_jr = 4'd10,
            BS_jal = 4'd11,
            BS_nop = 4'd12;

parameter   ALU_add = 4'd0,
            ALU_sub = 4'd1,
            ALU_sl  = 4'd2,
            ALU_sr  = 4'd3,
            ALU_and = 4'd4,
            ALU_or  = 4'd5,
            ALU_xor = 4'd6,
            ALU_nand= 4'd7,
            ALU_not = 4'd8,
            ALU_nor = 4'd9,
            ALU_nop = 4'd10;

always @(instruction)
begin
    case(instruction[79:74])
        ADD: begin
            SA = instruction[73:69];
            SB = instruction[68:64];
            SD = instruction[63:59];
            isBranch = 0;
            isLoad = 0;
            isStore = 0;
            isJump = 0;
            isDraw = 0;
            isIMMOpp = 0;
            WE = 1;
            ctrl_ALU = ALU_add;
            IMM = 0;
            ctrl_BS = BS_nop;
        end
        SUB: begin
            SA = instruction[73:69];
            SB = instruction[68:64];
            SD = instruction[63:59];
            isBranch = 0;
            isLoad = 0;
            isStore = 0;
            isJump = 0;
            isDraw = 0;
            isIMMOpp = 0;
            WE = 1;
            ctrl_ALU = ALU_sub;
            IMM = 0;
            ctrl_BS = BS_nop;
        end
        SL: begin
            SA = instruction[73:69];
            SB = instruction[68:64];
            SD = instruction[63:59];
            isBranch = 0;
            isLoad = 0;
            isStore = 0;
            isJump = 0;
            isDraw = 0;
            isIMMOpp = 0;
            WE = 1;
            ctrl_ALU = ALU_sl;
            IMM = 0;
            ctrl_BS = BS_nop;
        end
        SR: begin
            SA = instruction[73:69];
            SB = instruction[68:64];
            SD = instruction[63:59];
            isBranch = 0;
            isLoad = 0;
            isStore = 0;
            isJump = 0;
            isDraw = 0;
            isIMMOpp = 0;
            WE = 1;
            ctrl_ALU = ALU_sr;
            IMM = 0;
            ctrl_BS = BS_nop;
        end
        OR: begin
            SA = instruction[73:69];
            SB = instruction[68:64];
            SD = instruction[63:59];
            isBranch = 0;
            isLoad = 0;
            isStore = 0;
            isJump = 0;
            isDraw = 0;
            isIMMOpp = 0;
            WE = 1;
            ctrl_ALU = ALU_or;
            IMM = 0;
            ctrl_BS = BS_nop;
        end
        NOR: begin
            SA = instruction[73:69];
            SB = instruction[68:64];
            SD = instruction[63:59];
            isBranch = 0;
            isLoad = 0;
            isStore = 0;
            isJump = 0;
            isDraw = 0;
            isIMMOpp = 0;
            WE = 1;
            ctrl_ALU = ALU_nor;
            IMM = 0;
            ctrl_BS = BS_nop;
        end
        AND: begin
            SA = instruction[73:69];
            SB = instruction[68:64];
            SD = instruction[63:59];
            isBranch = 0;
            isLoad = 0;
            isStore = 0;
            isJump = 0;
            isDraw = 0;
            isIMMOpp = 0;
            WE = 1;
            ctrl_ALU = ALU_and;
            IMM = 0;
            ctrl_BS = BS_nop;
        end
        NAND: begin
            SA = instruction[73:69];
            SB = instruction[68:64];
            SD = instruction[63:59];
            isBranch = 0;
            isLoad = 0;
            isStore = 0;
            isJump = 0;
            isDraw = 0;
            isIMMOpp = 0;
            WE = 1;
            ctrl_ALU = ALU_nand;
            IMM = 0;
            ctrl_BS = BS_nop;
        end
        XOR: begin
            SA = instruction[73:69];
            SB = instruction[68:64];
            SD = instruction[63:59];
            isBranch = 0;
            isLoad = 0;
            isStore = 0;
            isJump = 0;
            isDraw = 0;
            isIMMOpp = 0;
            WE = 1;
            ctrl_ALU = ALU_xor;
            IMM = 0;
            ctrl_BS = BS_nop;
        end
        ADDI: begin
            SA = instruction[73:69];
            SD = instruction[68:64];
            SB = instruction[68:64];
            isBranch = 0;
            isLoad = 0;
            isStore = 0;
            isJump = 0;
            isDraw = 0;
            isIMMOpp = 1;
            WE = 1;
            ctrl_ALU = ALU_add;
            IMM = instruction[63:32];
            ctrl_BS = BS_nop;
        end
        SUBI: begin
            SA = instruction[73:69];
            SD = instruction[68:64];
            SB = instruction[68:64];
            isBranch = 0;
            isLoad = 0;
            isStore = 0;
            isJump = 0;
            isDraw = 0;
            isIMMOpp = 1;
            WE = 1;
            ctrl_ALU = ALU_sub;
            IMM = instruction[63:32];
            ctrl_BS = BS_nop;
        end
        SLI: begin
            SA = instruction[73:69];
            SD = instruction[68:64];
            SB = instruction[68:64];
            isBranch = 0;
            isLoad = 0;
            isStore = 0;
            isJump = 0;
            isDraw = 0;
            isIMMOpp = 1;
            WE = 1;
            ctrl_ALU = ALU_sl;
            IMM = instruction[63:32];
            ctrl_BS = BS_nop;
        end
        SRI: begin
            SA = instruction[73:69];
            SD = instruction[68:64];
            SB = instruction[68:64];
            isBranch = 0;
            isLoad = 0;
            isStore = 0;
            isJump = 0;
            isDraw = 0;
            isIMMOpp = 1;
            WE = 1;
            ctrl_ALU = ALU_sr;
            IMM = instruction[63:32];
            ctrl_BS = BS_nop;
        end
        ST: begin
```

```
        SA = instruction[73:69];
        SD = instruction[68:64];
        SB = instruction[68:64];
        isBranch = 0;
        isLoad = 0;
        isStore = 1;
        isJump = 0;
        isDraw = 0;
        isIMMOpp = 1;
        WE = 0;
        ctrl_ALU = ALU_add;
        IMM = instruction[63:32];
        ctrl_BS = BS_nop;
    end
    LD: begin
        SA = instruction[73:69];
        SD = instruction[68:64];
        SB = instruction[68:64];
        isBranch = 0;
        isLoad = 1;
        isStore = 0;
        isJump = 0;
        isDraw = 0;
        isIMMOpp = 1;
        WE = 1;
        ctrl_ALU = ALU_add;
        IMM = instruction[63:32];
        ctrl_BS = BS_nop;
    end
    NOT: begin
        SA = instruction[73:69];
        SB = instruction[68:64];
        SD = instruction[63:59];
        isBranch = 0;
        isLoad = 0;
        isStore = 0;
        isJump = 0;
        isDraw = 0;
        isIMMOpp = 0;
        WE = 1;
        ctrl_ALU = ALU_not;
        IMM = 0;
        ctrl_BS = BS_nop;
    end
    BGT: begin
        SA = instruction[73:69];
        SB = instruction[68:64];
        SD = instruction[63:59];
        isBranch = 1;
        isLoad = 0;
        isStore = 0;
        isJump = 0;
        isDraw = 0;
        isIMMOpp = 1;
        IMM = instruction[63:32];
        WE = 0;
```

```
            ctrl_ALU = ALU_add;
            ctrl_BS = BS_bgt;
        end
        BLT: begin
            SA = instruction[73:69];
            SB = instruction[68:64];
            SD = instruction[63:59];
            isBranch = 1;
            isLoad = 0;
            isStore = 0;
            isJump = 0;
            isDraw = 0;
            isIMMOpp = 1;
            IMM = instruction[63:32];
            WE = 0;
            ctrl_ALU = ALU_add;
            ctrl_BS = BS_blt;
        end
        BGE: begin
            SA = instruction[73:69];
            SB = instruction[68:64];
            SD = instruction[63:59];
            isBranch = 1;
            isLoad = 0;
            isStore = 0;
            isJump = 0;
            isDraw = 0;
            isIMMOpp = 1;
            IMM = instruction[63:32];
            WE = 0;
            ctrl_ALU = ALU_add;
            ctrl_BS = BS_bge;
        end
        BLE: begin
            SA = instruction[73:69];
            SB = instruction[68:64];
            SD = instruction[63:59];
            isBranch = 1;
            isLoad = 0;
            isStore = 0;
            isJump = 0;
            isDraw = 0;
            isIMMOpp = 1;
            IMM = instruction[63:32];
            WE = 0;
            ctrl_ALU = ALU_add;
            ctrl_BS = BS_ble;
        end
        BEQ: begin
            SA = instruction[73:69];
            SB = instruction[68:64];
            SD = instruction[63:59];
            isBranch = 1;
            isLoad = 0;
            isStore = 0;
            isJump = 0;
            isDraw = 0;
            isIMMOpp = 1;
            IMM = instruction[63:32];
            WE = 0;
            ctrl_ALU = ALU_add;
            ctrl_BS = BS_beq;
        end
        BNE: begin
            SA = instruction[73:69];
            SB = instruction[68:64];
            SD = instruction[63:59];
            isBranch = 1;
            isLoad = 0;
            isStore = 0;
            isJump = 0;
            isDraw = 0;
            isIMMOpp = 1;
            IMM = instruction[63:32];
            WE = 0;
            ctrl_ALU = ALU_add;
            ctrl_BS = BS_bne;
        end
        BEZ: begin
            SA = instruction[73:69];
            SB = instruction[68:64];
            SD = instruction[63:59];
            isBranch = 1;
            isLoad = 0;
            isStore = 0;
            isJump = 0;
            isDraw = 0;
            isIMMOpp = 1;
            IMM = instruction[63:32];
            WE = 0;
            ctrl_ALU = ALU_add;
            ctrl_BS = BS_bez;
        end
        BNEZ: begin
            SA = instruction[73:69];
            SB = instruction[68:64];
            SD = instruction[63:59];
            isBranch = 1;
            isLoad = 0;
            isStore = 0;
            isJump = 0;
            isDraw = 0;
            isIMMOpp = 1;
            IMM = instruction[63:32];
            WE = 0;
            ctrl_ALU = ALU_add;
            ctrl_BS = BS_bnez;
        end
        J: begin
            SA = instruction[73:69];
            SB = instruction[68:64];
            SD = instruction[63:59];
            isBranch = 0;
            isLoad = 0;
            isStore = 0;
            isJump = 1;
            isDraw = 0;
            isIMMOpp = 1;
            IMM = instruction[63:32];
            WE = 0;
            ctrl_ALU = ALU_add;
            ctrl_BS = BS_j;
        end
        JR: begin
            SA = instruction[73:69];
            SB = 0;
            SD = instruction[63:59];
            isBranch = 0;
            isLoad = 0;
            isStore = 0;
            isJump = 1;
            isDraw = 0;
            isIMMOpp = 0;
            WE = 0;
            ctrl_ALU = ALU_add;
            ctrl_BS = BS_jr;
            IMM = 0;
        end
        JAL: begin
            SA = instruction[73:69];
            SB = 0;
            SD = 5'd31;
            isBranch = 0;
            isLoad = 0;
            isStore = 0;
            isJump = 1;
            isDraw = 0;
            isIMMOpp = 0;
            IMM = instruction[63:32];
            WE = 1;
            ctrl_ALU = ALU_add;
            ctrl_BS = BS_jr;
        end
        CIRCLE_FILL: begin
            SA = instruction[73:69];
            SB = instruction[68:64];
            SD = instruction[63:59];
            isBranch = 0;
            isLoad = 0;
            isStore = 0;
            isJump = 0;
            isDraw = 1;
            isIMMOpp = 0;
            WE = 0;
            ctrl_ALU = ALU_nop;
            IMM = 0;
            ctrl_BS = BS_nop;
        end
```

```
        CIRCLE_DRAW: begin
            SA = instruction[73:69];
            SB = instruction[68:64];
            SD = instruction[63:59];
            isBranch = 0;
            isLoad = 0;
            isStore = 0;
            isJump = 0;
            isDraw = 1;
            isIMMOpp = 0;
            WE = 0;
            ctrl_ALU = ALU_nop;
            IMM = 0;
            ctrl_BS = BS_nop;
        end
        SQUARE_FILL: begin
            SA = instruction[73:69];
            SB = instruction[68:64];
            SD = instruction[63:59];
            isBranch = 0;
            isLoad = 0;
            isStore = 0;
            isJump = 0;
            isDraw = 1;
            isIMMOpp = 0;
            WE = 0;
            ctrl_ALU = ALU_nop;
            IMM = 0;
            ctrl_BS = BS_nop;
        end
        SQUARE_DRAW: begin
            SA = instruction[73:69];
            SB = instruction[68:64];
            SD = instruction[63:59];
            isBranch = 0;
            isLoad = 0;
            isStore = 0;
            isJump = 0;
            isDraw = 1;
            isIMMOpp = 0;
            WE = 0;
            ctrl_ALU = ALU_nop;
            IMM = 0;
            ctrl_BS = BS_nop;
        end
        LINE_DRAW: begin
            SA = instruction[73:69];
            SB = instruction[68:64];
            SD = instruction[63:59];
            isBranch = 0;
            isLoad = 0;
            isStore = 0;
            isJump = 0;
            isDraw = 1;
            isIMMOpp = 0;
            WE = 0;
            ctrl_ALU = ALU_nop;
            IMM = 0;
            ctrl_BS = BS_nop;
        end
        TEXT_WRITE: begin
            SA = instruction[73:69];
            SB = instruction[68:64];
            SD = instruction[63:59];
            isBranch = 0;
            isLoad = 0;
            isStore = 0;
            isJump = 0;
            isDraw = 1;
            isIMMOpp = 0;
            WE = 0;
            ctrl_ALU = ALU_nop;
            IMM = 0;
            ctrl_BS = BS_nop;
        end
        FRAME_UPDATE: begin
            SA = instruction[73:69];
            SB = instruction[68:64];
            SD = instruction[63:59];
            isBranch = 0;
            isLoad = 0;
            isStore = 0;
            isJump = 0;
            isDraw = 1;
            isIMMOpp = 0;
            WE = 0;
            ctrl_ALU = ALU_nop;
            IMM = 0;
            ctrl_BS = BS_nop;
        end
        default: begin
            SA = 0;
            SB = 0;
            SD = 0;
            isBranch = 0;
            isLoad = 0;
            isStore = 0;
            isJump = 0;
            isDraw = 0;
            isIMMOpp = 0;
            WE = 0;
            ctrl_ALU = ALU_and;
            IMM = 0;
            ctrl_BS = BS_nop;
        end
    endcase
end

endmodule



/*
```

```
Instruction - 79:74

Drawing Functions: 000XXX

Other Functions:
*Add, subtract, shift, or, nor, and, nand, xor
*BGT, BLT, BGTE, BLTE, BEQ, BNE, J, JR, BEZ, BNEZ
*LD, ST
*(SLT, SGT)









*/
```

```
//Peter Greczner, Data RAM

module DataRam(clock, reset, address, isStore, data_out, data_in);

input wire isStore;
input wire [31:0] data_in;
input wire clock, reset;
input wire [7:0] address;

reg [31:0] local_memory [0:255];

output reg [31:0] data_out;

always @(address)
begin
    data_out <= local_memory[address];
end


always @(posedge clock)
begin
    if(reset) begin

    end
    else begin
        if(isStore) begin
            local_memory[address] <= data_in;
        end
    end
end

endmodule
```

```
//Peter Greczner, Branch Select

module BranchSelect(isBranch, isJump, ctrl_BS, alu_EQ, alu_Z, alu_GT, alu_LT, takeBranch);

input wire isBranch, isJump, alu_EQ, alu_Z, alu_GT, alu_LT;
input wire [3:0] ctrl_BS;

output wire takeBranch;

assign takeBranch =     ~(isBranch | isJump)    ? 0                     :
                        (ctrl_BS == BS_bgt)     ? (alu_GT)              :
                        (ctrl_BS == BS_blt)     ? (alu_LT)              :
                        (ctrl_BS == BS_bge)     ? (alu_GT | alu_EQ)     :
                        (ctrl_BS == BS_ble)     ? (alu_LT | alu_EQ)     :
                        (ctrl_BS == BS_beq)     ? (alu_EQ)              :
                        (ctrl_BS == BS_bne)     ? (~alu_EQ)             :
                        (ctrl_BS == BS_bnez)    ? (~alu_Z)              :
                        (ctrl_BS == BS_j)       ? 1                     :
                        (ctrl_BS == BS_jr)      ? 1                     :
                        (ctrl_BS == BS_jal)     ? 1                     : 0;




parameter   BS_bgt = 4'd0,
            BS_blt = 4'd1,
            BS_bge = 4'd2,
            BS_ble = 4'd3,
            //BS_ble = 4'd4,
            BS_beq = 4'd5,
            BS_bne = 4'd6,
            BS_bez = 4'd7,
            BS_bnez = 4'd8,
            BS_j = 4'd9,
            BS_jr = 4'd10,
            BS_jal = 4'd11;


endmodule
```

```
//Peter Greczner, ALU

module ALU(control, A, B, C, isZero, isNegative, isEqual, isGT, isLT);

input wire [3:0] control;
input wire signed [31:0] A,B;
output reg [31:0] C;

output wire isZero, isNegative, isEqual, isGT, isLT;
assign isNegative = 0;

assign isZero = (C == 0) ? 1: 0;
assign isEqual = (A == B) ? 1 : 0;

assign isGT = (A > B) ? 1 : 0;
assign isLT = (A < B) ? 1 : 0;


parameter   add = 4'd0,
            subtract = 4'd1,
            shiftLeft = 4'd2,
            shiftRight = 4'd3,
            AND = 4'd4,
            OR = 4'd5,
            XOR = 4'd6,
            NAND = 4'd7,
            NOT = 4'd8,
            NOR = 4'd9,
            NOP = 4'd10;


always @ (control, A, B)
begin
    case(control)
        add: begin
            C = A + B;
        end

        subtract: begin
            C = A - B;
        end

        shiftLeft: begin
            C = A << B;
        end

        shiftRight: begin
            C = A >> B;
        end

        AND: begin
            C = A & B;
        end

        OR: begin
            C = A | B;
        end

        XOR: begin
            C = A ^ B;
        end

        NAND: begin
             C = ~(A & B);
        end

        NOT: begin
            C = ~A;
        end

        NOR: begin
            C = ~(A | B);
        end
        NOP: begin
            C = 0;
        end
        default: begin
            C = A | B;
        end
    endcase

end



endmodule
```

```
module TextWriter(  VGA_SYNC, VGA_CLK,
                    data_reg,
                    reset,
                    my_id, id_req,
                    command,
                    color,
                    font_select,
                    letter_addr, letterX, letterY,
                    available,

                    xAddr, yAddr,
                    write_clk,
                    is_write_finished
                    );


input wire VGA_SYNC, VGA_CLK;

output reg [15:0] data_reg; //memory data register  for SRAM
input wire [15:0] color;
input wire [1:0] command;
input wire reset;
input wire [3:0] my_id, id_req;
input wire [1:0] font_select;
input wire [11:0] letter_addr;
input wire signed [10:0] letterX, letterY;
output wire available;
reg available_reg;
assign available = available_reg;

output reg write_clk;
input wire is_write_finished;

output reg [9:0] xAddr;
output reg [8:0] yAddr;


reg [11:0] currLetterAddr;
reg [2:0] widthInd;
wire [7:0] currLetterData;

font1_rom f1r(currLetterAddr, letterClock, currLetterData);

reg [2:0] lineNumber;
reg [2:0] state, next_state;
reg letterClock;

parameter   [2:0]   waiting = 3'd0,
                    loadLetterLine = 3'd1,
                    printLetterLine = 3'd2,
                    looping = 3'd3,
                    write_state = 3'd4,
                    initial_clock = 3'd5;

always @ (posedge VGA_CLK) begin
    if(reset) begin
```

```
        data_reg <= 16'b0;                              //write all zeros (black)
        available_reg <= 1'b0;
        lineNumber <= 3'd0;
        state <= waiting;
        widthInd = 3'd0;
        letterClock <= 1'b0;
    end
    else if (VGA_SYNC) begin
        if(1'b1) begin
            case(state)

                write_state: begin
                    if(is_write_finished) begin
                        state <= next_state;
                        write_clk <= 1'b0;
                    end
                end

                waiting: begin
                    if(command[1] == 1'b0) begin available_reg <= 1'b1; end

                    if(command[1] == 1'b1 && (my_id == id_req)) begin
                        available_reg <= 1'b0;
                        lineNumber <= 3'd0;

                        state <= initial_clock;
                        letterClock <= 1'b1;
                        currLetterAddr <= letter_addr;
                        widthInd <= 3'd0;

                    end
                end

                initial_clock: begin
                    letterClock <= 1'b0;
                    state <= loadLetterLine;
                end

                loadLetterLine: begin
                    letterClock <= 1'b1;
                    state <= looping;
                end

                looping: begin
                    letterClock <= 1'b0;
                    available_reg <= 1'b0;


                    //Draw the Letter
                    if(currLetterData[~widthInd] == 1'b1) begin
                        data_reg <= color;

                        xAddr <= letterX[9:0] + widthInd;
                        yAddr <= letterY[8:0] + lineNumber;
                        state <= write_state;
                        write_clk <= 1'b1;
                    end
                    else begin //easy to implement w/background

                    end

                    //Loop Through Pixels
                    if(widthInd < 3'd7) begin
                        widthInd <= widthInd + 3'd1;
                        next_state <= looping;
                        if(~currLetterData[~widthInd] == 1'b1) begin state <= looping; end
                    end
                    else begin
                        if(lineNumber < 3'd7) begin
                            lineNumber <= lineNumber + 3'd1;
                            currLetterAddr <= letter_addr + {9'd0,(lineNumber+3'd1)};
                            widthInd <= 3'd0;

                            next_state <= loadLetterLine;
                            if(~currLetterData[~widthInd] == 1'b1) begin state <= loadLetterLine
; end
                        end
                        else begin

                            next_state <= waiting;
                            lineNumber <= 3'd0;
                            if(~currLetterData[~widthInd] == 1'b1) begin state <= waiting; end
                        end
                    end


                end

                default: begin
                    available_reg <= 1'b0;
                    lineNumber <= 3'd0;
                    state <= waiting;

                end

            endcase
        end
    end
    else
    begin

    end
end




endmodule
```

```verilog
module TransmitClock(tSend_clk_in, tSend_signal_out, tSend_reset_in);


input wire tSend_clk_in;
output wire tSend_signal_out;
reg tSend_out;

assign tSend_signal_out = tSend_out;

input wire tSend_reset_in;

//reg last_clk;

always @ (posedge tSend_clk_in, posedge tSend_reset_in) begin
    if(tSend_reset_in) begin
        //last_clk = 1'b0;
        tSend_out = 1'b0;
    end
    else begin
        if(tSend_clk_in) begin
            tSend_out = 1'b1;
        end
        else begin
            tSend_out = 1'b0;
        end
    end


end






endmodule
```

```
// megafunction wizard: %ALTPLL%
// GENERATION: STANDARD
// VERSION: WM1.0
// MODULE: altpll

// ============================================================
// File Name: VGA_Audio_PLL.v
// Megafunction Name(s):
//                      altpll
// ============================================================
// ************************************************************
// THIS IS A WIZARD-GENERATED FILE. DO NOT EDIT THIS FILE!
//
// 5.1 Build 176 10/26/2005 SJ Full Version
// ************************************************************


//Copyright (C) 1991-2005 Altera Corporation
//Your use of Altera Corporation's design tools, logic functions
//and other software and tools, and its AMPP partner logic
//functions, and any output files any of the foregoing
//(including device programming or simulation files), and any
//associated documentation or information are expressly subject
//to the terms and conditions of the Altera Program License
//Subscription Agreement, Altera MegaCore Function License
//Agreement, or other applicable license agreement, including,
//without limitation, that your use is for the sole purpose of
//programming logic devices manufactured by Altera and sold by
//Altera or its authorized distributors.  Please refer to the
//applicable agreement for further details.


// synopsys translate_off
`timescale 1 ps / 1 ps
// synopsys translate_on
module VGA_Audio_PLL (
	areset,
	inclk0,
	c0,
	c1,
	c2);

	input	  areset;
	input	  inclk0;
	output	  c0;
	output	  c1;
	output	  c2;

	wire [5:0] sub_wire0;
	wire [0:0] sub_wire6 = 1'h0;
	wire [2:2] sub_wire3 = sub_wire0[2:2];
	wire [1:1] sub_wire2 = sub_wire0[1:1];
	wire [0:0] sub_wire1 = sub_wire0[0:0];
	wire  c0 = sub_wire1;
	wire  c1 = sub_wire2;
	wire  c2 = sub_wire3;
```

```
    wire  sub_wire4 = inclk0;
    wire [1:0] sub_wire5 = {sub_wire6, sub_wire4};

    altpll  altpll_component (
                .inclk (sub_wire5),
                .areset (areset),
                .clk (sub_wire0)
                // synopsys translate_off
                ,
                .scanclk (),
                .pllena (),
                .sclkout1 (),
                .sclkout0 (),
                .fbin (),
                .scandone (),
                .clkloss (),
                .extclk (),
                .clkswitch (),
                .pfdena (),
                .scanaclr (),
                .clkena (),
                .clkbad (),
                .scandata (),
                .enable1 (),
                .scandataout (),
                .extclkena (),
                .enable0 (),
                .scanwrite (),
                .locked (),
                .activeclock (),
                .scanread ()
                // synopsys translate_on
                );
    defparam
        altpll_component.clk0_divide_by = 15,
        altpll_component.clk0_duty_cycle = 50,
        altpll_component.clk0_multiply_by = 14,
        altpll_component.clk0_phase_shift = "0",
        altpll_component.clk1_divide_by = 3,
        altpll_component.clk1_duty_cycle = 50,
        altpll_component.clk1_multiply_by = 2,
        altpll_component.clk1_phase_shift = "0",
        altpll_component.clk2_divide_by = 15,
        altpll_component.clk2_duty_cycle = 50,
        altpll_component.clk2_multiply_by = 14,
        altpll_component.clk2_phase_shift = "-9921",
        altpll_component.compensate_clock = "CLK0",
        altpll_component.inclk0_input_frequency = 37037,
        altpll_component.intended_device_family = "Cyclone II",
        altpll_component.lpm_type = "altpll",
        altpll_component.operation_mode = "NORMAL",
        altpll_component.pll_type = "FAST";


endmodule
```

```
// ============================================================
// CNX file retrieval info
// ============================================================
// Retrieval info: PRIVATE: ACTIVECLK_CHECK STRING "0"
// Retrieval info: PRIVATE: BANDWIDTH STRING "1.000"
// Retrieval info: PRIVATE: BANDWIDTH_FEATURE_ENABLED STRING "0"
// Retrieval info: PRIVATE: BANDWIDTH_FREQ_UNIT STRING "MHz"
// Retrieval info: PRIVATE: BANDWIDTH_PRESET STRING "Low"
// Retrieval info: PRIVATE: BANDWIDTH_USE_AUTO STRING "1"
// Retrieval info: PRIVATE: BANDWIDTH_USE_CUSTOM STRING "0"
// Retrieval info: PRIVATE: BANDWIDTH_USE_PRESET STRING "0"
// Retrieval info: PRIVATE: CLKBAD_SWITCHOVER_CHECK STRING "0"
// Retrieval info: PRIVATE: CLKLOSS_CHECK STRING "0"
// Retrieval info: PRIVATE: CLKSWITCH_CHECK STRING "1"
// Retrieval info: PRIVATE: CNX_NO_COMPENSATE_RADIO STRING "0"
// Retrieval info: PRIVATE: CREATE_CLKBAD_CHECK STRING "0"
// Retrieval info: PRIVATE: CREATE_INCLK1_CHECK STRING "0"
// Retrieval info: PRIVATE: CUR_DEDICATED_CLK STRING "c0"
// Retrieval info: PRIVATE: CUR_FBIN_CLK STRING "e0"
// Retrieval info: PRIVATE: DEVICE_SPEED_GRADE STRING "Any"
// Retrieval info: PRIVATE: DEV_FAMILY STRING "Cyclone II"
// Retrieval info: PRIVATE: DIV_FACTOR0 NUMERIC "1"
// Retrieval info: PRIVATE: DIV_FACTOR1 NUMERIC "6"
// Retrieval info: PRIVATE: DIV_FACTOR2 NUMERIC "1"
// Retrieval info: PRIVATE: DUTY_CYCLE0 STRING "50.00000000"
// Retrieval info: PRIVATE: DUTY_CYCLE1 STRING "50.00000000"
// Retrieval info: PRIVATE: DUTY_CYCLE2 STRING "50.00000000"
// Retrieval info: PRIVATE: EXT_FEEDBACK_RADIO STRING "0"
// Retrieval info: PRIVATE: GLOCKED_COUNTER_EDIT_CHANGED STRING "1"
// Retrieval info: PRIVATE: GLOCKED_FEATURE_ENABLED STRING "1"
// Retrieval info: PRIVATE: GLOCKED_MODE_CHECK STRING "0"
// Retrieval info: PRIVATE: GLOCK_COUNTER_EDIT NUMERIC "1048575"
// Retrieval info: PRIVATE: HAS_MANUAL_SWITCHOVER STRING "1"
// Retrieval info: PRIVATE: INCLK0_FREQ_EDIT STRING "27.000"
// Retrieval info: PRIVATE: INCLK0_FREQ_UNIT_COMBO STRING "MHz"
// Retrieval info: PRIVATE: INCLK1_FREQ_EDIT STRING "27.000"
// Retrieval info: PRIVATE: INCLK1_FREQ_EDIT_CHANGED STRING "1"
// Retrieval info: PRIVATE: INCLK1_FREQ_UNIT_CHANGED STRING "1"
// Retrieval info: PRIVATE: INCLK1_FREQ_UNIT_COMBO STRING "MHz"
// Retrieval info: PRIVATE: INT_FEEDBACK__MODE_RADIO STRING "1"
// Retrieval info: PRIVATE: LOCKED_OUTPUT_CHECK STRING "0"
// Retrieval info: PRIVATE: LONG_SCAN_RADIO STRING "1"
// Retrieval info: PRIVATE: LVDS_MODE_DATA_RATE STRING "Not Available"
// Retrieval info: PRIVATE: LVDS_MODE_DATA_RATE_DIRTY NUMERIC "0"
// Retrieval info: PRIVATE: LVDS_PHASE_SHIFT_UNIT0 STRING "deg"
// Retrieval info: PRIVATE: LVDS_PHASE_SHIFT_UNIT1 STRING "deg"
// Retrieval info: PRIVATE: LVDS_PHASE_SHIFT_UNIT2 STRING "ps"
// Retrieval info: PRIVATE: MIRROR_CLK0 STRING "0"
// Retrieval info: PRIVATE: MIRROR_CLK1 STRING "0"
// Retrieval info: PRIVATE: MIRROR_CLK2 STRING "0"
// Retrieval info: PRIVATE: MULT_FACTOR0 NUMERIC "1"
// Retrieval info: PRIVATE: MULT_FACTOR1 NUMERIC "5"
// Retrieval info: PRIVATE: MULT_FACTOR2 NUMERIC "1"
// Retrieval info: PRIVATE: NORMAL_MODE_RADIO STRING "1"
// Retrieval info: PRIVATE: OUTPUT_FREQ0 STRING "25.200"
// Retrieval info: PRIVATE: OUTPUT_FREQ1 STRING "18.000"
// Retrieval info: PRIVATE: OUTPUT_FREQ2 STRING "25.200"
// Retrieval info: PRIVATE: OUTPUT_FREQ_MODE0 STRING "1"
// Retrieval info: PRIVATE: OUTPUT_FREQ_MODE1 STRING "1"
// Retrieval info: PRIVATE: OUTPUT_FREQ_MODE2 STRING "1"
// Retrieval info: PRIVATE: OUTPUT_FREQ_UNIT0 STRING "MHz"
// Retrieval info: PRIVATE: OUTPUT_FREQ_UNIT1 STRING "MHz"
// Retrieval info: PRIVATE: OUTPUT_FREQ_UNIT2 STRING "MHz"
// Retrieval info: PRIVATE: PHASE_SHIFT0 STRING "0.00000000"
// Retrieval info: PRIVATE: PHASE_SHIFT1 STRING "0.00000000"
// Retrieval info: PRIVATE: PHASE_SHIFT2 STRING "-90.00000000"
// Retrieval info: PRIVATE: PHASE_SHIFT_UNIT0 STRING "deg"
// Retrieval info: PRIVATE: PHASE_SHIFT_UNIT1 STRING "deg"
// Retrieval info: PRIVATE: PHASE_SHIFT_UNIT2 STRING "deg"
// Retrieval info: PRIVATE: PLL_ADVANCED_PARAM_CHECK STRING "0"
// Retrieval info: PRIVATE: PLL_ARESET_CHECK STRING "1"
// Retrieval info: PRIVATE: PLL_AUTOPLL_CHECK NUMERIC "1"
// Retrieval info: PRIVATE: PLL_ENA_CHECK STRING "0"
// Retrieval info: PRIVATE: PLL_ENHPLL_CHECK NUMERIC "0"
// Retrieval info: PRIVATE: PLL_FASTPLL_CHECK NUMERIC "0"
// Retrieval info: PRIVATE: PLL_LVDS_PLL_CHECK NUMERIC "0"
// Retrieval info: PRIVATE: PLL_PFDENA_CHECK STRING "0"
// Retrieval info: PRIVATE: PLL_TARGET_HARCOPY_CHECK NUMERIC "0"
// Retrieval info: PRIVATE: PRIMARY_CLK_COMBO STRING "inclk0"
// Retrieval info: PRIVATE: SACN_INPUTS_CHECK STRING "0"
// Retrieval info: PRIVATE: SCAN_FEATURE_ENABLED STRING "0"
// Retrieval info: PRIVATE: SHORT_SCAN_RADIO STRING "0"
// Retrieval info: PRIVATE: SPREAD_FEATURE_ENABLED STRING "0"
// Retrieval info: PRIVATE: SPREAD_FREQ STRING "50.000"
// Retrieval info: PRIVATE: SPREAD_FREQ_UNIT STRING "KHz"
// Retrieval info: PRIVATE: SPREAD_PERCENT STRING "0.500"
// Retrieval info: PRIVATE: SPREAD_USE STRING "0"
// Retrieval info: PRIVATE: SRC_SYNCH_COMP_RADIO STRING "0"
// Retrieval info: PRIVATE: STICKY_CLK0 STRING "1"
// Retrieval info: PRIVATE: STICKY_CLK1 STRING "1"
// Retrieval info: PRIVATE: STICKY_CLK2 STRING "1"
// Retrieval info: PRIVATE: SWITCHOVER_COUNT_EDIT NUMERIC "1"
// Retrieval info: PRIVATE: SWITCHOVER_FEATURE_ENABLED STRING "1"
// Retrieval info: PRIVATE: USE_CLK0 STRING "1"
// Retrieval info: PRIVATE: USE_CLK1 STRING "1"
// Retrieval info: PRIVATE: USE_CLK2 STRING "1"
// Retrieval info: PRIVATE: USE_CLKENA0 STRING "0"
// Retrieval info: PRIVATE: USE_CLKENA1 STRING "0"
// Retrieval info: PRIVATE: USE_CLKENA2 STRING "0"
// Retrieval info: PRIVATE: ZERO_DELAY_RADIO STRING "0"
// Retrieval info: LIBRARY: altera_mf altera_mf.altera_mf_components.all
// Retrieval info: CONSTANT: CLK0_DIVIDE_BY NUMERIC "15"
// Retrieval info: CONSTANT: CLK0_DUTY_CYCLE NUMERIC "50"
// Retrieval info: CONSTANT: CLK0_MULTIPLY_BY NUMERIC "14"
// Retrieval info: CONSTANT: CLK0_PHASE_SHIFT STRING "0"
// Retrieval info: CONSTANT: CLK1_DIVIDE_BY NUMERIC "3"
// Retrieval info: CONSTANT: CLK1_DUTY_CYCLE NUMERIC "50"
// Retrieval info: CONSTANT: CLK1_MULTIPLY_BY NUMERIC "2"
// Retrieval info: CONSTANT: CLK1_PHASE_SHIFT STRING "0"
// Retrieval info: CONSTANT: CLK2_DIVIDE_BY NUMERIC "15"
// Retrieval info: CONSTANT: CLK2_DUTY_CYCLE NUMERIC "50"
// Retrieval info: CONSTANT: CLK2_MULTIPLY_BY NUMERIC "14"
// Retrieval info: CONSTANT: CLK2_PHASE_SHIFT STRING "-9921"
// Retrieval info: CONSTANT: COMPENSATE_CLOCK STRING "CLK0"
// Retrieval info: CONSTANT: INCLK0_INPUT_FREQUENCY NUMERIC "37037"
// Retrieval info: CONSTANT: INTENDED_DEVICE_FAMILY STRING "Cyclone II"
// Retrieval info: CONSTANT: LPM_TYPE STRING "altpll"
// Retrieval info: CONSTANT: OPERATION_MODE STRING "NORMAL"
// Retrieval info: CONSTANT: PLL_TYPE STRING "FAST"
// Retrieval info: USED_PORT: @clk 0 0 6 0 OUTPUT VCC "@clk[5..0]"
// Retrieval info: USED_PORT: @extclk 0 0 4 0 OUTPUT VCC "@extclk[3..0]"
// Retrieval info: USED_PORT: areset 0 0 0 0 INPUT GND "areset"
// Retrieval info: USED_PORT: c0 0 0 0 0 OUTPUT VCC "c0"
// Retrieval info: USED_PORT: c1 0 0 0 0 OUTPUT VCC "c1"
// Retrieval info: USED_PORT: c2 0 0 0 0 OUTPUT VCC "c2"
// Retrieval info: USED_PORT: inclk0 0 0 0 0 INPUT GND "inclk0"
// Retrieval info: CONNECT: @inclk 0 0 1 0 inclk0 0 0 0 0
// Retrieval info: CONNECT: c0 0 0 0 0 @clk 0 0 1 0
// Retrieval info: CONNECT: c1 0 0 0 0 @clk 0 0 1 1
// Retrieval info: CONNECT: c2 0 0 0 0 @clk 0 0 1 2
// Retrieval info: CONNECT: @inclk 0 0 1 1 GND 0 0 0 0
// Retrieval info: CONNECT: @areset 0 0 0 0 areset 0 0 0 0
// Retrieval info: GEN_FILE: TYPE_NORMAL VGA_Audio_PLL.v TRUE FALSE
// Retrieval info: GEN_FILE: TYPE_NORMAL VGA_Audio_PLL.inc FALSE FALSE
// Retrieval info: GEN_FILE: TYPE_NORMAL VGA_Audio_PLL.cmp FALSE FALSE
// Retrieval info: GEN_FILE: TYPE_NORMAL VGA_Audio_PLL.bsf FALSE FALSE
// Retrieval info: GEN_FILE: TYPE_NORMAL VGA_Audio_PLL_inst.v FALSE FALSE
// Retrieval info: GEN_FILE: TYPE_NORMAL VGA_Audio_PLL_bb.v FALSE FALSE
// Retrieval info: GEN_FILE: TYPE_NORMAL VGA_Audio_PLL_waveforms.html TRUE FALSE
// Retrieval info: GEN_FILE: TYPE_NORMAL VGA_Audio_PLL_wave*.jpg FALSE FALSE
```

```
module XAddr_MUX(
                    xCF, xCD, xSD, xSF, xLD, xTW, xIDLE,
                    clock, select,
                    xOut



);

input wire [9:0] xCF, xCD, xSD, xSF, xLD, xTW, xIDLE;
input wire clock;
input wire [3:0] select;
output wire [9:0] xOut;
reg [9:0] xReg;
assign xOut = xReg;

parameter   [3:0]   circle_fill = 4'd0,
                    circle_draw = 4'd1,
                    square_fill = 4'd2,
                    square_draw = 4'd3,
                    line_draw = 4'd4,
                    text_write = 4'd5,
                    frame_update = 4'd6,
                    idle = 4'd15;



always @(posedge clock) begin
    case(select)
        circle_fill: begin xReg <= xCF; end
        circle_draw: begin xReg <= xCD; end
        square_fill: begin xReg <= xSF; end
        square_draw: begin xReg <= xSD; end
        line_draw: begin xReg <= xLD; end
        text_write: begin xReg <= xTW; end
        idle: begin xReg <= xIDLE; end
        default: begin
            xReg <= 0;
        end
    endcase
end

endmodule
```

```verilog
module YAddr_MUX(
                    yCF, yCD, ySD, ySF, yLD, yTW, yIDLE,
                    clock, select,
                    yOut



);

input wire [8:0] yCF, yCD, ySD, ySF, yLD, yTW, yIDLE;
input wire clock;
input wire [3:0] select;
output wire [8:0] yOut;
reg [8:0] yReg;
assign yOut = yReg;

parameter   [3:0]   circle_fill = 4'd0,
                    circle_draw = 4'd1,
                    square_fill = 4'd2,
                    square_draw = 4'd3,
                    line_draw = 4'd4,
                    text_write = 4'd5,
                    frame_update = 4'd6,
                    idle = 4'd15;


always @(posedge clock) begin
    case(select)
        circle_fill: begin yReg <= yCF; end
        circle_draw: begin yReg <= yCD; end
        square_fill: begin yReg <= ySF; end
        square_draw: begin yReg <= ySD; end
        line_draw: begin yReg <= yLD; end
        text_write: begin yReg <= yTW; end
        idle: begin yReg <= yIDLE; end
        default: begin
            yReg <= 0;
        end
    endcase
end

endmodule
```

```
//Peter Greczner - Simple Instruction RAM

module SimpleIRAM(clock, write, instr_in, pc, pc_write, instr_out, isDraw, reset);

input wire clock, write, reset;
output wire isDraw;

input wire [11:0] pc, pc_write;

input wire [79:0] instr_in;
output reg [79:0] instr_out;

assign isDraw = (instr_out[79] == 1'b1) ? 0 : 1;

reg [79:0] instr_list [0:1024];

always @(pc)
begin
    instr_out <= instr_list[pc[9:0]];
end

always @(posedge clock)
begin
    if(write) begin
        instr_list[pc[9:0]] <= instr_in;
    end
end

endmodule
```

```
//Peter Greczner, Register File

module RegisterFile(A_addr, B_addr, D_addr, w_en, A_data, B_data, D_data, clk);

input wire [4:0] A_addr, B_addr, D_addr;
input wire w_en, clk;

output wire [31:0] A_data, B_data;
input wire [31:0] D_data;

assign A_data = registers[A_addr];
assign B_data = registers[B_addr];

reg [31:0] registers [0:31];

always @(posedge clk)
begin
    if(w_en)
    begin
        registers[D_addr] <= D_data;
    end
end






endmodule
```

```
//Peter Greczner, PC

module ProgramCounter(clock, enabled, PC_out, PC_in, reset, isDraw, isDrawDone);

input wire clock;
input wire enabled;
input wire reset;
input wire isDraw, isDrawDone;
input wire [11:0] PC_in;
output reg [11:0] PC_out;

always @ (posedge clock, posedge reset)
begin
    if(reset)
    begin
        PC_out <= 12'b0;
    end
    else if(enabled & ~(isDraw & isDrawDone)) begin
        PC_out <= PC_in;
    end
end

endmodule
```

```verilog
//Peter Greczner, PC

module Processor(
                    clock_50,
                    draw_complete,
                    reset,
                    enable,
                    instruction,
                    pc,
                    isDrawInstruction
                    );


input wire clock_50;
input wire draw_complete;
input wire reset;
input wire enable;

output wire isDrawInstruction;
assign isDrawInstruction = isDraw;

input wire [79:0] instruction;
output wire pc;
assign pc = pc_out;


wire pc_enable;
assign pc_enable = enable & draw_complete;
wire [11:0] pc_in;
wire [11:0] pc_out;
ProgramCounter pc1(
                    .clock(clock_50),
                    .reset(reset),
                    .enabled(pc_enable),
                    .PC_out(pc_out),
                    .PC_in(pc_in),
                    .isDraw(isDraw),
                    .isDrawDone(draw_complete)
                    );


wire [3:0] ctrl_ALU;
wire [3:0] ctrl_BS;
wire isBranch, isStore, isJump, isDraw, isIMMOpp, isLoad;
wire [4:0] SA, SB, SD;
wire [31:0] IMM;
wire WE;
Decoder dec1(   .instruction(instruction),
                .ctrl_ALU(ctrl_ALU),
                .ctrl_BS(ctrl_BS),
                .isBranch(isBranch),
                .isLoad(isLoad),
                .isStore(isStore),
                .isJump(isJump),
                .isDraw(isDraw),
                .isIMMOpp(isIMMOpp),
                .SA(SA),
                .SB(SB),
                .SD(SD),
                .IMM(IMM),
                .WE(WE)
                );

wire [31:0] A_data, B_data, D_data;
RegisterFile regfile1(
                        .A_addr(SA),
                        .B_addr(SB),
                        .D_addr(SD),
                        .w_en(WE),
                        .A_data(A_data),
                        .B_data(B_data),
                        .D_data(D_data),
                        .clk(clock_50)
                    );

wire [31:0] alu_res;
wire isZero, isNegative, isEqual, isGT, isLT;
wire [31:0] B_input;
assign B_input = isIMMOpp ? IMM : B_data;
ALU alu1(
            .control(ctrl_ALU),
            .A(A_data),
            .B(B_input),
            .C(alu_res),
            .isZero(isZero),
            .isNegative(isNegative),
            .isEqual(isEqual),
            .isGT(isGT),
            .isLT(isLT)
        );

wire [31:0] data_mem;
assign D_data = isLoad ? data_mem : alu_res;
DataRam dr1(
                .clock(clock_50),
                .reset(reset),
                .address(alu_res[7:0]),
                .isStore(isStore),
                .data_out(data_mem),
                .data_in(B_data)
            );

wire takeBranch;
BranchSelect bs1(
                    .isBranch(isBranch),
                    .isJump(isJump),
                    .ctrl_BS(ctrl_BS),
                    .alu_EQ(isEqual),
                    .alu_Z(isZero),
                    .alu_GT(isGT),
                    .alu_LT(isLT),
                    .takeBranch(takeBranch)
```

```
                );

PCUpdater pcu1(
                .ctrl_BS(ctrl_BS),
                .takeBranch(takeBranch),
                .alu_res(alu_res),
                .PC_now(pc_out),
                .PC_next(pc_in)
                );

endmodule
```

```verilog
//Peter Greczner, PC Updater

module PCUpdater(ctrl_BS, takeBranch, alu_res, PC_now, PC_next);

input wire [3:0] ctrl_BS;
input wire takeBranch;
input wire [31:0] alu_res;
input wire [11:0] PC_now;

output reg [11:0] PC_next;


always @(ctrl_BS, alu_res, PC_now, takeBranch)
begin
    if(~takeBranch)
    begin
        PC_next = PC_now + 1;
    end
    else begin
        PC_next = alu_res;
    end
end


endmodule
```

```
module GPU_Greczner
    (
        ////////////////////    Clock Input         ////////////////////
        CLOCK_27,                           //      27 MHz
        CLOCK_50,                           //      50 MHz
        EXT_CLOCK,                          //      External Clock
        ////////////////////    Push Button         ////////////////////
        KEY,                                //      Pushbutton[3:0]
        ////////////////////    DPDT Switch         ////////////////////
        SW,                                 //      Toggle Switch[17:0]
        ////////////////////    7-SEG Dispaly       ////////////////////
        HEX0,                               //      Seven Segment Digit 0
        HEX1,                               //      Seven Segment Digit 1
        HEX2,                               //      Seven Segment Digit 2
        HEX3,                               //      Seven Segment Digit 3
        HEX4,                               //      Seven Segment Digit 4
        HEX5,                               //      Seven Segment Digit 5
        HEX6,                               //      Seven Segment Digit 6
        HEX7,                               //      Seven Segment Digit 7
        //////////////////////// LED        ////////////////////////
        LEDG,                               //      LED Green[8:0]
        LEDR,                               //      LED Red[17:0]
        //////////////////////// UART ////////////////////////
        UART_TXD,                           //      UART Transmitter
        UART_RXD,                           //      UART Receiver
        //////////////////////// 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 0
        DRAM_CLK,                           //      SDRAM Clock
        DRAM_CKE,                           //      SDRAM Clock Enable
        ////////////////////    Flash Interface     ////////////////
        FL_DQ,                              //      FLASH Data bus 8 Bits
        FL_ADDR,                            //      FLASH Address bus 22 Bits
        FL_WE_N,                            //      FLASH Write Enable
        FL_RST_N,                           //      FLASH Reset
        FL_OE_N,                            //      FLASH Output Enable
        FL_CE_N,                            //      FLASH Chip Enable
        ////////////////////    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////////////////
OTG_DATA,                           //   ISP1362 Data bus 16 Bits
OTG_ADDR,                           //   ISP1362 Address 2 Bits
OTG_CS_N,                           //   ISP1362 Chip Select
OTG_RD_N,                           //   ISP1362 Write
OTG_WR_N,                           //   ISP1362 Read
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_DAT3,                            //   SD Card Data 3
SD_CMD,                             //   SD Card Command Signal
SD_CLK,                             //   SD Card Clock
////////////////////   USB JTAG link    ////////////////////
TDI,                                // CPLD -> FPGA (data in)
TCK,                                // CPLD -> FPGA (clk)
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
        ////////////////////  GPIO  ////////////////////////////
        GPIO_0,                             //  GPIO Connection 0
        GPIO_1                              //  GPIO Connection 1
    );

/////////////////////// Clock Input       ///////////////////////
input           CLOCK_27;                   //  27 MHz
input           CLOCK_50;                   //  50 MHz
input           EXT_CLOCK;                  //  External Clock
/////////////////////// Push Button       ///////////////////////
input   [3:0]   KEY;                        //  Pushbutton[3:0]
/////////////////////// DPDT Switch           ///////////////////////
input   [17:0]  SW;                         //  Toggle Switch[17:0]
/////////////////////// 7-SEG Dispaly     ///////////////////////
output  [6:0]   HEX0;                       //  Seven Segment Digit 0
output  [6:0]   HEX1;                       //  Seven Segment Digit 1
output  [6:0]   HEX2;                       //  Seven Segment Digit 2
output  [6:0]   HEX3;                       //  Seven Segment Digit 3
output  [6:0]   HEX4;                       //  Seven Segment Digit 4
output  [6:0]   HEX5;                       //  Seven Segment Digit 5
output  [6:0]   HEX6;                       //  Seven Segment Digit 6
output  [6:0]   HEX7;                       //  Seven Segment Digit 7
////////////////////////////  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 High-byte Data Mask
output          SRAM_LB_N;                      //    SRAM Low-byte Data Mask
output          SRAM_WE_N;                      //    SRAM Write Enable
output          SRAM_CE_N;                      //    SRAM Chip Enable
output          SRAM_OE_N;                      //    SRAM Output Enable
////////////////////    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           SD_DAT;                         //    SD Card Data
inout           SD_DAT3;                        //    SD Card Data 3
inout           SD_CMD;                         //    SD Card Command Signal
output          SD_CLK;                         //    SD Card Clock
//////////////////////// I2C        ////////////////////////////////
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         ////////////////////////////
output/*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 Devoder      ////////////////////////////
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
//////////////////////// GPIO ////////////////////////////////
inout   [35:0]  GPIO_0;                     //    GPIO Connection 0
inout   [35:0]  GPIO_1;                     //    GPIO Connection 1

////////////////////////////////////////////////////////////////////////
////////////////////////////////////
//DLA state machine variables
wire reset;
wire we; // we for SRAM
wire [17:0] addr_reg; //memory address register for SRAM
wire [15:0] data_reg; //memory data register  for SRAM
assign reset = ~KEY[0];
////////////////////////////////////////
////////////////////////////////////////////////////////////////////////

//    LCD ON
assign  LCD_ON      =   1'b0;
assign  LCD_BLON    =   1'b0;

//    All inout port turn to tri-state
assign  DRAM_DQ     =   16'hzzzz;
assign  FL_DQ       =   8'hzz;
assign  SRAM_DQ     =   16'hzzzz;
assign  OTG_DATA    =   16'hzzzz;
assign  SD_DAT      =   1'bz;
assign  ENET_DATA   =   16'hzzzz;
assign  GPIO_0      =   36'hzzzzzzzzz;
assign  GPIO_1      =   36'hzzzzzzzzz;
```

```verilog
wire        VGA_CTRL_CLK;
wire        AUD_CTRL_CLK;
wire [9:0]  mVGA_R, mVGA_R1, mVGA_R2, mVGA_R3, mVGA_R4;
wire [9:0]  mVGA_G, mVGA_G1, mVGA_G2, mVGA_G3, mVGA_G4;
wire [9:0]  mVGA_B, mVGA_B1, mVGA_B2, mVGA_B3, mVGA_B4;
wire [19:0] mVGA_ADDR;          //video memory address
wire [9:0]  Coord_X, Coord_Y;   //display coods
wire        DLY_RST;

assign  TD_RESET    =   1'b1;   //  Allow 27 MHz input

Reset_Delay         r0  (   .iCLK(CLOCK_50),.oRESET(DLY_RST)    );

VGA_Audio_PLL       p1  (   .areset(~DLY_RST),.inclk0(CLOCK_27),.c0(VGA_CTRL_CLK),.c1(
AUD_CTRL_CLK),.c2(VGA_CLK)    );


VGA_Controller      u1  (   //  Host Side
                            .iCursor_RGB_EN(4'b0111),
                            .oAddress(mVGA_ADDR),
                            .oCoord_X(Coord_X),
                            .oCoord_Y(Coord_Y),
                            .iRed(mVGA_R),
                            .iGreen(mVGA_G),
                            .iBlue(mVGA_B),
                            //  VGA Side
                            .oVGA_R(VGA_R),
                            .oVGA_G(VGA_G),
                            .oVGA_B(VGA_B),
                            .oVGA_H_SYNC(VGA_HS),
                            .oVGA_V_SYNC(VGA_VS),
                            .oVGA_SYNC(VGA_SYNC),
                            .oVGA_BLANK(VGA_BLANK),
                            //  Control Signal
                            .iCLK(VGA_CTRL_CLK),
                            .iRST_N(DLY_RST)    );

// SRAM_control
assign SRAM_ADDR = addr_reg;
assign SRAM_DQ = (we)? 16'hzzzz : data_reg ;
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

parameter [2:0] sixteenBit_single_320x240 = 3'd0,
                eightBit_single_640x480 = 3'd1,
                eightBit_double_320x240 = 3'd2,
                fourBit_double_640x480 = 3'd3,
                fiveBit_triple_320x240 = 3'd4,
                fourBit_quad_320x240 = 3'd5,
                twoBit_quad_640x480 = 3'd6,
                foreground_background = 3'd7;
```

```
// Show SRAM on the VGA
//320x240 single frame
assign  mVGA_R1 = {SRAM_DQ[15:12], 6'b0} ;
assign  mVGA_G1 = {SRAM_DQ[11:8], 6'b0} ;
assign  mVGA_B1 = {SRAM_DQ[7:4], 6'b0} ;
//640x480 - single frame
assign  mVGA_R2 = (Coord_X[0]) ? {SRAM_DQ[15:13], SRAM_DQ[15:13],SRAM_DQ[15:13],1'b1} : {SRAM_DQ
[7:5], SRAM_DQ[7:5],SRAM_DQ[7:5],1'b1} ;
assign  mVGA_G2 = (Coord_X[0]) ? {SRAM_DQ[12:10], SRAM_DQ[12:10],SRAM_DQ[12:10],1'b1} : {SRAM_DQ
[4:2], SRAM_DQ[4:2],SRAM_DQ[4:2],1'b1};
assign  mVGA_B2 = (Coord_X[0]) ? {SRAM_DQ[9:8], SRAM_DQ[9:8],SRAM_DQ[9:8],SRAM_DQ[9:8],SRAM_DQ[9
:8]}  : {SRAM_DQ[1:0], SRAM_DQ[1:0],SRAM_DQ[1:0],SRAM_DQ[1:0],SRAM_DQ[1:0]};
//320x240 - double frame
assign  mVGA_R3 = ~readFrame ? {SRAM_DQ[15:13], SRAM_DQ[15:13],SRAM_DQ[15:13],1'b1} : {SRAM_DQ[7
:5], SRAM_DQ[7:5],SRAM_DQ[7:5],1'b1} ;
assign  mVGA_G3 = ~readFrame ? {SRAM_DQ[12:10], SRAM_DQ[12:10],SRAM_DQ[12:10],1'b1} : {SRAM_DQ[4
:2], SRAM_DQ[4:2],SRAM_DQ[4:2],1'b1};
assign  mVGA_B3 = ~readFrame ? {SRAM_DQ[9:8], SRAM_DQ[9:8],SRAM_DQ[9:8],SRAM_DQ[9:8],SRAM_DQ[9:8
]}  : {SRAM_DQ[1:0], SRAM_DQ[1:0],SRAM_DQ[1:0],SRAM_DQ[1:0],SRAM_DQ[1:0]};
//640x480 - double frame
assign mVGA_R4 = ~Coord_X[0] ? (readFrame ? ({10{SRAM_DQ[7]}}) : ({10{SRAM_DQ[3]}})) : (
readFrame ? ({10{SRAM_DQ[11]}}) : ({10{SRAM_DQ[15]}}));
assign mVGA_G4 = ~Coord_X[0] ? (readFrame ? ({5{SRAM_DQ[6:5]}}) : ({5{SRAM_DQ[2:1]}})) : (
readFrame ? ({5{SRAM_DQ[10:9]}}) : ({5{SRAM_DQ[14:13]}}));
assign mVGA_B4 = ~Coord_X[0] ? (readFrame ? ({10{SRAM_DQ[4]}}) : ({10{SRAM_DQ[0]}})) : (
readFrame ? ({10{SRAM_DQ[8]}}) : ({10{SRAM_DQ[12]}}));

assign mVGA_R = (drawMode == sixteenBit_single_320x240) ? mVGA_R1:
                (drawMode == eightBit_single_640x480)   ? mVGA_R2:
                (drawMode == eightBit_double_320x240)   ? mVGA_R3:
                (drawMode == fourBit_double_640x480)    ? mVGA_R4: 10'd0;


assign mVGA_G = (drawMode == sixteenBit_single_320x240) ? mVGA_G1:
                (drawMode == eightBit_single_640x480)   ? mVGA_G2:
                (drawMode == eightBit_double_320x240)   ? mVGA_G3:
                (drawMode == fourBit_double_640x480)    ? mVGA_G4: 10'd0;


assign mVGA_B = (drawMode == sixteenBit_single_320x240) ? mVGA_B1:
                (drawMode == eightBit_single_640x480)   ? mVGA_B2:
                (drawMode == eightBit_double_320x240)   ? mVGA_B3:
                (drawMode == fourBit_double_640x480)    ? mVGA_B4: 10'd0;


wire LD_available, CD_available, CF_available, SD_available, SF_available, TW_available,
FU_available;

parameter   [3:0]   circle_fill = 4'd0,
                    circle_draw = 4'd1,
                    square_fill = 4'd2,
                    square_draw = 4'd3,
                    line_draw = 4'd4,
                    text_write = 4'd5,
                    frame_update = 4'd6;
```

```
line_drawer LD1(
            .x1(xstart_req),
            .y1(ystart_req),
            .x2(xend_width_req),
            .y2(yend_height_req),
            .VGA_SYNC(~VGA_VS | ~VGA_HS),
            .VGA_CLK(VGA_CTRL_CLK),
            .data_reg(cLD),
            .reset(reset),
            .color({color_req,4'h0}),
            .command({cmd_req,1'b0}),
            .available(LD_available),
            .my_id(line_draw),
            .id_req(avail_select),
            .xAddr(xLD),
            .yAddr(yLD),
            .is_write_finished(isWriteFinished),
            .write_clk(wLD)
            );


circle_draw CD1(
            .x1(xstart_req),
            .y1({1'b0,ystart_req}),
            .radius(xend_width_req),
            .VGA_SYNC(~VGA_VS | ~VGA_HS),
            .VGA_CLK(VGA_CTRL_CLK),
            .data_reg(cCD),
            .reset(reset),
            .color({color_req,4'h0}),
            .command({cmd_req,1'b0}),
            .available(CD_available),
            .my_id(circle_draw),
            .id_req(avail_select),
            .xAddr(xCD),
            .yAddr(yCD),
            .is_write_finished(isWriteFinished),
            .write_clk(wCD)
            );



square_draw SD1(
            .x1(xstart_req),
            .y1(ystart_req),
            .width(xend_width_req),
            .height(yend_height_req),
            .VGA_SYNC(~VGA_VS | ~VGA_HS),
            .VGA_CLK(VGA_CTRL_CLK),
            .data_reg(cSD),
            .reset(reset),
            .color({color_req,4'h0}),
            .command({cmd_req,1'b0}),
            .available(SD_available),
            .my_id(square_draw),
            .id_req(avail_select),
            .xAddr(xSD),
            .yAddr(ySD),
            .is_write_finished(isWriteFinished),
            .write_clk(wSD)
            );

square_fill SF1(
            .x1(xstart_req),
            .y1(ystart_req),
            .width(xend_width_req),
            .height(yend_height_req),
            .VGA_SYNC(~VGA_VS | ~VGA_HS),
            .VGA_CLK(VGA_CTRL_CLK),
            .data_reg(cSF),
            .reset(reset),
            .color({color_req,4'h0}),
            .command({cmd_req,1'b0}),
            .available(SF_available),
            .my_id(square_fill),
            .id_req(avail_select),
            .xAddr(xSF),
            .yAddr(ySF),
            .is_write_finished(isWriteFinished),
            .write_clk(wSF)
            );


circle_fill_better CFB1(
            .x1(xstart_req),
            .y1({1'b0,ystart_req}),
            .radius(xend_width_req),
            .VGA_SYNC(~VGA_VS | ~VGA_HS),
            .VGA_CLK(VGA_CTRL_CLK),
            .reset(reset),
            .color({color_req,4'h0}),
            .command({cmd_req,1'b0}),
            .data_reg(cCF),
            .available(CF_available),
            .my_id(circle_fill),
            .id_req(avail_select),
            .xAddr(xCF),
            .yAddr(yCF),
            .is_write_finished(isWriteFinished),
            .write_clk(wCF)
            );

TextWriter tw1(
                .VGA_SYNC(~VGA_VS | ~VGA_HS),
                .VGA_CLK(VGA_CTRL_CLK),
                .data_reg(cTW),
                .reset(reset),
                .my_id(text_write),
                .id_req(avail_select),
                .command({cmd_req,1'b0}),
                .color({color_req,4'h0}),
                .font_select(2'b0),
                .letter_addr({xend_width_req,yend_height_req[8:7]}),
                .letterX(xstart_req),
                .letterY(ystart_req),
                .available(TW_available),
                .xAddr(xTW),
                .yAddr(yTW),
                .is_write_finished(isWriteFinished),
                .write_clk(wTW)
                );




DataWriter dw1(
                    .x_addr(xAddr), //x coordinate of pixel to draw to
                    .y_addr(yAddr), //y coordinate of pixel to draw to
                    .color_data(cValue), //the color data
                    .isFinished(isWriteFinished), //flag if the write has completed
                    .clock(VGA_CTRL_CLK), //the clock to handle the writing
                    .sync(~VGA_VS | ~VGA_HS), //to keep us in sync with VGA
                    .reset(~KEY[0]), // a reset signal
                    .write_req_clk(write_clock), //clock that starts a new write request
                    .read_addr_out(addr_reg), //the address we want to read data from
                    .data_write_out(data_reg), //the new data that we will write to SRAM
                    .we_out(we), //the write enable signal (0 = write, 1 = read)
                    .read_data_in(SRAM_DQ), //the data that was read
                    //.currFrame_out(LEDR[17:16]),
                    .drawMode_sel(drawMode),
                    .frame_sel(writeFrame),
                    .Coord_X(Coord_X),
                    .Coord_Y(Coord_Y)

);
wire [2:0] drawMode;

assign drawMode = extra1_req[7:5];
//assign frame_sel = extra1_req[4:3];

wire [9:0] xCF, xCD, xSD, xSF, xLD, xTW, xIDLE, xAddr;
wire [8:0] yCF, yCD, ySD, ySF, yLD, yTW, yIDLE, yAddr;
wire [15:0] cCF, cCD, cSD, cSF, cLD, cTW, cIDLE, cValue;
wire isWriteFinished;
wire write_clock;
wire wCF, wCD, wSD, wSF, wLD, wTW, wIDLE;

assign write_clock = wCF | wCD | wSD | wSF | wLD | wTW;

XAddr_MUX xam1(
                    .xCF(xCF),
                    .xCD(xCD),
                    .xSD(xSD),
                    .xSF(xSF),
                    .xLD(xLD),
                    .xTW(xTW),
                    .xIDLE(xIDLE),
                    .clock(VGA_CTRL_CLK),
                    .select(avail_select),
                    .xOut(xAddr)
);

YAddr_MUX yam1(
                    .yCF(yCF),
                    .yCD(yCD),
                    .ySD(ySD),
                    .ySF(ySF),
                    .yLD(yLD),
                    .yTW(yTW),
                    .yIDLE(yIDLE),
                    .clock(VGA_CTRL_CLK),
                    .select(avail_select),
                    .yOut(yAddr)
);

Color_MUX cm1(
                    .cCF(cCF),
                    .cCD(cCD),
                    .cSD(cSD),
                    .cSF(cSF),
                    .cLD(cLD),
                    .cTW(cTW),
                    .cIDLE(cIDLE),
                    .clock(VGA_CTRL_CLK),
                    .select(avail_select),
                    .cOut(cValue)
);


wire avail_result;
wire [3:0] avail_select;

availableMUX AM1(   .clk(VGA_CTRL_CLK),
                        .availability_out(avail_result),
                        .CD_available(CD_available),
                        .CF_available(CF_available),
                        .SF_available(SF_available),
                        .SD_available(SD_available),
                        .LD_available(LD_available),
                        .TW_available(TW_available),
                        .FU_available(FU_available),
                        .select(avail_select),
                        .enable(1'b1)
                    );

//assign LEDR[15:9] = {CD_available, CF_available, SF_available, SD_available, LD_available, TW_available, FU_available};
//assign LEDR[3:0] = avail_select;
wire [11:0] color_req;
wire [9:0] xstart_req, xend_width_req;
wire [8:0] ystart_req, yend_height_req;
wire [7:0] extra1_req, extra2_req;
wire [5:0] instr_req;
wire cmd_req;
wire grabPacketClock;
wire sendDataClock;
wire readFrame;
wire writeFrame;
RequestManager_update rmu1(
//RequestHandler rh1(
                             .clock(VGA_CTRL_CLK),
                             .packet_in(currentPacket),//currentPacket),
                             .packet_crc_in(currentPacketCRC),
                             .packet_available(pavail),
                             .ocolor(color_req),
                             .oxstart(xstart_req),
                             .oystart(ystart_req),
                             .oxend_width(xend_width_req),
                             .oyend_height(yend_height_req),
                             .oextra1(extra1_req),
                             .oextra2(extra2_req),
                             .oinstr(instr_req),
                             .reset(~KEY[0]),
                             .grab_out(grabPacketClock),
                             .nodeWanted(avail_select),
                             .nodeWanted_isAvailable(avail_result),
                             .command_out(cmd_req),
                             .sync(~VGA_VS | ~VGA_HS),
                             .transmit_clock(sendDataClock),
                             .isTransmitting(currentlySendingData),
                             .my_state(LEDG[7:4]),
                             .packetID(packetID),
                             .frame_read(readFrame),
                             .frame_write(writeFrame),
                             //.SW(SW),
                             //.IRAM_out(IRAM_out),
                             //.drawMode(drawMode)//,
                             //.bug_check(LEDR[7:0])
                             );

wire took_packet;
wire packet_avail;
wire [79:0] IRAM_out;

assign LEDR[17:0] = SW[2:0] == 3'd0 ? IRAM_out[79:62] :
                    SW[2:0] == 3'd1 ? IRAM_out[61:44] :
                    SW[2:0] == 3'd2 ? IRAM_out[43:26] :
                    SW[2:0] == 3'd3 ? IRAM_out[25:8] :
                    SW[2:0] == 3'd4 ? {10'd0,IRAM_out[7:0]} : currentPacket[25:8];

sdram_pll neg_3ns (CLOCK_50, DRAM_CLK);


wire baudClock, baudClock_8Times, baudClock_fast8;
wire[7:0] lastSerialReceived;
wire [7:0] dataCounter;
wire [8:0] receive_error_check;
wire dataReady;
wire errorFlag;
wire packet_timeout;   // signal that the current packet has timedout
wire [79:0] packet1; //the current packet data
wire packet1_ready; //signal that the current packet is ready
wire packet1_crc_result; //the CRC result of the current packet
wire transmit_idle;
wire [79:0] last_packet; //the last packet received
wire [7:0] crc;
wire [7:0] lastCRC; //the last CRC result received
wire sendInProgress; //is the module currently sending

//The baud generator
BaudGenerator bg1(   .clock_in(CLOCK_50),
                     .baud_clock_out(baudClock)
                   );
//8 times baud generator
BaudGenerator_8Times bg8t1( .clock_in(CLOCK_50),
                            .baud_clock_out(baudClock_8Times)
                            );
//better 8 times as fast baud generator
BaudGenerator_fast bgf1(.clock_in(CLOCK_50),
                        .baud_clock_out(baudClock_fast8)
                        );
//the serial receiver
SerialReceiver_improved_fastClock sr_i1(
            .CLOCK_50(CLOCK_50),
            .UART_RXD(UART_RXD),
            .lastReceived(lastSerialReceived),
            .dataReady_out(dataReady),
            .errorFlag_out(errorFlag),
            .reset(~KEY[0]),
            .error_check(receive_error_check),
            .timeout(packet_timeout),
            .baud_clock(baudClock),
            .baud_8_clock(baudClock_fast8)
            );




//The Packet manager module
wire [79:0] currentPacket;
wire currentPacketCRC;
wire currentPacket_isAvailable;
wire [7:0] packetID;
PacketManager_updated pmu1(
                            .data_clock(dataReady),
                            .data_in(lastSerialReceived),
                            .latestPacket(currentPacket),
                            .packetCRC(currentPacketCRC),
                            .packetIsAvailable(currentPacket_isAvailable),
                            .reset(~KEY[0]),
                            .grab_clock(grabPacketClock),
                            .fast_clock(VGA_CTRL_CLK),
                            .PR(LEDG[1]),
                            .packetID(packetID)
```

```
                              );
wire pavail;
wire [79:0] pdata;
/*PacketHolder ph1(
                    .packet_in_avail(currentPacket_isAvailable),

                    .packet_out_avail(pavail),

                    .packet_request_in(grabPacketClock)//~KEY[2])
                    );*/

TransmitClock tc2(
                    .tSend_clk_in(currentPacket_isAvailable),//~KEY[3]),
                    .tSend_signal_out(pavail),
                    .tSend_reset_in(~KEY[0] | grabPacketClock)
                );

wire currentlySendingData;
SerialTransmitter_updated STU1(
                                .CLOCK_50(CLOCK_50),
                                .UART_TXD(UART_TXD),
                                .toTransmit({2'b0,currentPacket[79:74]}),
//{8{currentPacketCRC}}),//SW[7:0]),//
                                .reset(~KEY[0]),
                                .sendNew(transclk),//sendDataClock),
                                .baud_clock(baudClock),
                                .sendInProgress(gotSend)//currentlySendingData)
                                );

wire transclk;
wire gotSend;
//assign LEDR[11] = gotSend;
TransmitClock tc1(
                    .tSend_clk_in(sendDataClock | ~KEY[3]),//~KEY[3]),
                    .tSend_signal_out(transclk),
                    .tSend_reset_in(~KEY[0] | gotSend)
                    );

//assign LEDR[8:6] = drawMode;
//assign LEDR[5:0] = instr_req;
//assign LEDR[10:9] = {readFrame,writeFrame};
//assign LEDR[15:8] = packetID[7:0];
assign LEDG[0] = pavail;
//assign LEDG[1] = currentPacketCRC;
assign LEDG[2] = currentPacketCRC;


endmodule //top module
```

```
// megafunction wizard: %ROM: 1-PORT%
// GENERATION: STANDARD
// VERSION: WM1.0
// MODULE: altsyncram

// ============================================================
// File Name: font1_rom.v
// Megafunction Name(s):
//                      altsyncram
//
// Simulation Library Files(s):
//                      altera_mf
// ============================================================
// ************************************************************
// THIS IS A WIZARD-GENERATED FILE. DO NOT EDIT THIS FILE!
//
// 9.1 Build 222 10/21/2009 SJ Full Version
// ************************************************************


//Copyright (C) 1991-2009 Altera Corporation
//Your use of Altera Corporation's design tools, logic functions
//and other software and tools, and its AMPP partner logic
//functions, and any output files from any of the foregoing
//(including device programming or simulation files), and any
//associated documentation or information are expressly subject
//to the terms and conditions of the Altera Program License
//Subscription Agreement, Altera MegaCore Function License
//Agreement, or other applicable license agreement, including,
//without limitation, that your use is for the sole purpose of
//programming logic devices manufactured by Altera and sold by
//Altera or its authorized distributors.  Please refer to the
//applicable agreement for further details.


// synopsys translate_off
`timescale 1 ps / 1 ps
// synopsys translate_on
module font1_rom (
    address,
    clock,
    q);

    input   [11:0]  address;
    input     clock;
    output  [7:0]  q;

    wire [7:0] sub_wire0;
    wire [7:0] q = sub_wire0[7:0];

    altsyncram  altsyncram_component (
                .clock0 (clock),
                .address_a (address),
                .q_a (sub_wire0),
                .aclr0 (1'b0),
                .aclr1 (1'b0),
                .address_b (1'b1),
                .addressstall_a (1'b0),
                .addressstall_b (1'b0),
                .byteena_a (1'b1),
                .byteena_b (1'b1),
                .clock1 (1'b1),
                .clocken0 (1'b1),
                .clocken1 (1'b1),
                .clocken2 (1'b1),
                .clocken3 (1'b1),
                .data_a ({8{1'b1}}),
                .data_b (1'b1),
                .eccstatus (),
                .q_b (),
                .rden_a (1'b1),
                .rden_b (1'b1),
                .wren_a (1'b0),
                .wren_b (1'b0));
    defparam
        altsyncram_component.clock_enable_input_a = "BYPASS",
        altsyncram_component.clock_enable_output_a = "BYPASS",
        altsyncram_component.init_file = "eight.mif",
        altsyncram_component.intended_device_family = "Cyclone II",
        altsyncram_component.lpm_hint = "ENABLE_RUNTIME_MOD=NO",
        altsyncram_component.lpm_type = "altsyncram",
        altsyncram_component.numwords_a = 4096,
        altsyncram_component.operation_mode = "ROM",
        altsyncram_component.outdata_aclr_a = "NONE",
        altsyncram_component.outdata_reg_a = "CLOCK0",
        altsyncram_component.widthad_a = 12,
        altsyncram_component.width_a = 8,
        altsyncram_component.width_byteena_a = 1;


endmodule

// ============================================================
// CNX file retrieval info
// ============================================================
// Retrieval info: PRIVATE: ADDRESSSTALL_A NUMERIC "0"
// Retrieval info: PRIVATE: AclrAddr NUMERIC "0"
// Retrieval info: PRIVATE: AclrByte NUMERIC "0"
// Retrieval info: PRIVATE: AclrOutput NUMERIC "0"
// Retrieval info: PRIVATE: BYTE_ENABLE NUMERIC "0"
// Retrieval info: PRIVATE: BYTE_SIZE NUMERIC "8"
// Retrieval info: PRIVATE: BlankMemory NUMERIC "0"
// Retrieval info: PRIVATE: CLOCK_ENABLE_INPUT_A NUMERIC "0"
// Retrieval info: PRIVATE: CLOCK_ENABLE_OUTPUT_A NUMERIC "0"
// Retrieval info: PRIVATE: Clken NUMERIC "0"
// Retrieval info: PRIVATE: IMPLEMENT_IN_LES NUMERIC "0"
// Retrieval info: PRIVATE: INIT_FILE_LAYOUT STRING "PORT_A"
// Retrieval info: PRIVATE: INIT_TO_SIM_X NUMERIC "0"
// Retrieval info: PRIVATE: INTENDED_DEVICE_FAMILY STRING "Cyclone II"
// Retrieval info: PRIVATE: JTAG_ENABLED NUMERIC "0"
// Retrieval info: PRIVATE: JTAG_ID STRING "NONE"
// Retrieval info: PRIVATE: MAXIMUM_DEPTH NUMERIC "0"
// Retrieval info: PRIVATE: MIFfilename STRING "eight.mif"
// Retrieval info: PRIVATE: NUMWORDS_A NUMERIC "4096"
// Retrieval info: PRIVATE: RAM_BLOCK_TYPE NUMERIC "0"
// Retrieval info: PRIVATE: RegAddr NUMERIC "1"
// Retrieval info: PRIVATE: RegOutput NUMERIC "1"
// Retrieval info: PRIVATE: SYNTH_WRAPPER_GEN_POSTFIX STRING "0"
// Retrieval info: PRIVATE: SingleClock NUMERIC "1"
// Retrieval info: PRIVATE: UseDQRAM NUMERIC "0"
// Retrieval info: PRIVATE: WidthAddr NUMERIC "12"
// Retrieval info: PRIVATE: WidthData NUMERIC "8"
// Retrieval info: PRIVATE: rden NUMERIC "0"
// Retrieval info: CONSTANT: CLOCK_ENABLE_INPUT_A STRING "BYPASS"
// Retrieval info: CONSTANT: CLOCK_ENABLE_OUTPUT_A STRING "BYPASS"
// Retrieval info: CONSTANT: INIT_FILE STRING "eight.mif"
// Retrieval info: CONSTANT: INTENDED_DEVICE_FAMILY STRING "Cyclone II"
// Retrieval info: CONSTANT: LPM_HINT STRING "ENABLE_RUNTIME_MOD=NO"
// Retrieval info: CONSTANT: LPM_TYPE STRING "altsyncram"
// Retrieval info: CONSTANT: NUMWORDS_A NUMERIC "4096"
// Retrieval info: CONSTANT: OPERATION_MODE STRING "ROM"
// Retrieval info: CONSTANT: OUTDATA_ACLR_A STRING "NONE"
// Retrieval info: CONSTANT: OUTDATA_REG_A STRING "CLOCK0"
// Retrieval info: CONSTANT: WIDTHAD_A NUMERIC "12"
// Retrieval info: CONSTANT: WIDTH_A NUMERIC "8"
// Retrieval info: CONSTANT: WIDTH_BYTEENA_A NUMERIC "1"
// Retrieval info: USED_PORT: address 0 0 12 0 INPUT NODEFVAL address[11..0]
// Retrieval info: USED_PORT: clock 0 0 0 0 INPUT NODEFVAL clock
// Retrieval info: USED_PORT: q 0 0 8 0 OUTPUT NODEFVAL q[7..0]
// Retrieval info: CONNECT: @address_a 0 0 12 0 address 0 0 12 0
// Retrieval info: CONNECT: q 0 0 8 0 @q_a 0 0 8 0
// Retrieval info: CONNECT: @clock0 0 0 0 0 clock 0 0 0 0
// Retrieval info: LIBRARY: altera_mf altera_mf.altera_mf_components.all
// Retrieval info: GEN_FILE: TYPE_NORMAL font1_rom.v TRUE
// Retrieval info: GEN_FILE: TYPE_NORMAL font1_rom.inc FALSE
// Retrieval info: GEN_FILE: TYPE_NORMAL font1_rom.cmp FALSE
// Retrieval info: GEN_FILE: TYPE_NORMAL font1_rom.bsf FALSE
// Retrieval info: GEN_FILE: TYPE_NORMAL font1_rom_inst.v FALSE
// Retrieval info: GEN_FILE: TYPE_NORMAL font1_rom_bb.v TRUE
// Retrieval info: GEN_FILE: TYPE_NORMAL font1_rom_waveforms.html TRUE
// Retrieval info: GEN_FILE: TYPE_NORMAL font1_rom_wave*.jpg FALSE
// Retrieval info: LIB_FILE: altera_mf
```

```verilog
// Draws a line from (x1, y1) to (x2, y2) to SRAM when draw is asserted.
// Gives control of SRAM to VGA when VGA is not syncing (~VGA_SYNC)
module line_drawer(x1, y1, x2, y2, VGA_SYNC, VGA_CLK,
                                    data_reg, reset, color, command,
                                    available,
                                    my_id,
                                    id_req,
                                    xAddr, yAddr,
                                    write_clk,
                                    is_write_finished
                                    );

input wire signed [10:0] x1, y1, x2, y2;
input wire VGA_SYNC, VGA_CLK;

output reg [15:0] data_reg; //memory data register  for SRAM

input wire reset;
input wire [15:0] color;
input wire [1:0] command;
output wire available;
input wire [3:0] my_id, id_req;

output reg write_clk;
input wire is_write_finished;

output reg [9:0] xAddr;
output reg [8:0] yAddr;

//state machine
reg [2:0] state, next_state;

// absolute values of dx, dy, e
wire signed [10:0] absdx, absdy;
assign absdx = (x2>x1) ? (x2-x1) : (x1-x2);
assign absdy = (y2>y1) ? (y2-y1) : (y1-y2);

// initial value of e should be negative
wire signed [10:0] nege;
assign nege = ((absdy - absdx) < 0) ? (absdy-absdx) : (absdx-absdy);

// whether or not x is the DA
wire xDA;
assign xDA = (absdx > absdy);

// signs of dx and dy (0 = positive, 1 = negative)
wire dxsign, dysign;
assign dxsign = ((x2-x1) < 0);
assign dysign = ((y2-y1) < 0);

// value to increment e by
wire signed [10:0] einc;
assign einc = (xDA) ? (absdy) : (absdx);

// value to decrement e by
wire signed [10:0] edec;
```

```
assign edec = (xDA) ? (absdx) : (absdy);

// value to increment i and j by
wire signed [10:0] iinc, jinc;
assign iinc = (xDA) ? (dxsign ? (-1) : (1)) : (dysign ? (-1): (1));
assign jinc = (xDA) ? (dysign ? (-1) : (1)) : (dxsign ? (-1): (1));

// finish indicates when line is finished drawing
wire finish, xfinish, yfinish;
// if x-increment is negative, stop when i is less than x2, etc.
assign xfinish = (dxsign) ? (i < x2) : (i > x2);
// if y-increment is negative, stop when i is less than x2, etc.
assign yfinish = (dysign) ? (i < y2) : (i > y2);
// if DA is x, check x for done drawing, else check y
assign finish = (xDA) ? (xfinish) : (yfinish);

reg available_reg;
assign available = available_reg;





// for Bresenham's Algorithm
reg signed [10:0] dy, dx, e, j, i;

//state names
parameter
        setup       = 3'd0,  // initialize Bresenham's algorithm
        draw_line       = 3'd1,  // draw the line
        ignore = 3'd3,
        halt        = 3'd2,
        write_state = 3'd4;  // wait for next drawline command

always @ (posedge VGA_CLK)
begin

    if (reset)      //synch reset assumes KEY0 is held down 1/60 second
    begin

        data_reg <= 16'b0;                          //write all zeros (black)
        state <= halt;
        available_reg <= 1'b0;
        write_clk <= 1'b0;
        next_state <= write_state;
    end

    //modify display during sync
    else if (VGA_SYNC)
    begin
        case(state)


            write_state: begin
                if(is_write_finished) begin
                    state <= next_state;
                    write_clk <= 1'b0;
                end
            end

            //setup line drawing
            setup:
            begin
                available_reg <= 1'b0; //indicate that module is in use


                dy <= absdy;
                dx <= absdx;
                e <= nege;

                // i always gets increments
                // j gets incremented when e>0
                if (xDA)
                begin
                    i <= x1;
                    j <= y1;
                end
                else
                begin
                    i <= y1;
                    j <= x1;
                end

                state <= draw_line;
            end

            //draw the line
            draw_line:
            begin
                if (finish)
                begin
                    state <= halt;
                    //we <= 1'b1;
                end
                else
                begin
                    //we <= 1'b0;
                    // if x is DA, i is x, y is j
                    if (xDA)
                    begin

                        data_reg <= color;

                        xAddr <= i[9:0];
                        yAddr <= j[8:0];
                        next_state <= draw_line;
                        write_clk <= 1'b1;
                        state <= write_state;
                    end
                    // if y is DA, j is x, y is i
                    else
                    begin
```

```
                        data_reg <= color;

                        xAddr <= j[9:0];
                        yAddr <= i[8:0];
                        next_state <= draw_line;
                        write_clk <= 1'b1;
                        state <= write_state;
                    end

                    if (e >= 0)
                    begin
                        j <= j + jinc;
                        e <= e - edec + einc;
                        i <= i + iinc;
                    end
                    else
                    begin
                        i <= i + iinc;
                        e <= e + einc;
                    end
                end
            end
            ignore: begin
                available_reg <= 1'b1;
                if(~command[1]) begin
                    state <= halt;
                end
            end
            //wait for next draw line command
            halt:
            begin
                if (command[1] && (my_id == id_req))
                begin
                    state <= setup;
                    available_reg <= 1'b0;
                end

                //we <= 1'b1;
                if(~command[1])available_reg <= 1'b1;   //indicate that module is free for use
            end

            default:
            begin

                available_reg <= 1'b0;
            end

        endcase
    end
    else
    begin

    end
end

endmodule
```

```
module PacketManager_updated(
                            data_clock,
                            data_in,
                            latestPacket,
                            packetCRC,
                            packetIsAvailable,
                            reset,
                            grab_clock,
                            fast_clock,
                            PR,
                            packetID
                            );

input wire data_clock;
input wire [7:0] data_in;
input wire reset;
input wire grab_clock;
input wire fast_clock;

output wire [79:0] latestPacket;
output wire packetCRC;
output wire packetIsAvailable;
output wire PR;
assign PR = packetReady;


reg packetReady, packetIsAvail;
reg [79:0] currentPacket, lastPacket;
reg [3:0] packetIndex;
output reg [7:0] packetID;
reg CRC;

reg [2:0] state;

parameter [2:0]     waiting = 3'd0,
                    looping = 3'd1,
                    waitForGrab = 3'd2;


//assign packetIsAvailable = packetIsAvail ^ packetReady;
assign packetIsAvailable = packetReady;
assign latestPacket = lastPacket;
assign packetCRC = CRC;


always @(posedge fast_clock) begin
    if(reset | ~data_clock) begin
        packetIsAvail <= 1'b0;
    end
    else begin
        if(grab_clock) begin
            packetIsAvail <= 1'b1;
        end

    end
end
```

```
always @(posedge data_clock, posedge reset)
begin
    if(reset) begin
        currentPacket <= 80'b0;
        packetIndex <= 4'b0;
        packetReady <= 1'b0;
        lastPacket <= 80'b0;
        CRC <= 1'b0;
        state <= waiting;
        packetID <= 0;
    end
    else begin
        case(state)
            waiting: begin
                currentPacket <= currentPacket | {72'b0, data_in};
                packetIndex <= packetIndex + 1;
                state <= looping;
                packetReady <= 1'b0;
            end
            looping: begin
                if(packetIndex < 4'd9) begin
                    currentPacket <= ( (currentPacket << 8) | {72'b0,data_in} );
                    packetIndex <= packetIndex + 1;
                end
                else begin //the last packet to arrive
                    lastPacket <= ( (currentPacket << 8) | {72'b0,data_in} );
                    CRC <= (data_in == (currentPacket[7:0] ^ currentPacket[15:8] ^ currentPacket
[23:16]
                                ^ currentPacket[31:24] ^ currentPacket[39:32] ^ currentPacket[47:40]
                                ^ currentPacket[55:48] ^ currentPacket[63:56] ^ currentPacket[71:64
])) ?
                                    1'b1: 1'b0;
                    packetReady <= 1'b1;
                    state <= waiting;
                    currentPacket <= 80'b0;
                    packetIndex <= 4'd0;
                    if(packetID < 8'd255) begin
                        packetID <= packetID + 8'd1;
                    end
                    else begin
                        packetID <= 8'd0;
                    end

                end
            end
            waitForGrab: begin
                //if(grab_clock) begin
                    state <= waiting;
                    packetReady <= 1'b0;
                //end
            end
        endcase

    end
end
```

```
endmodule
```

```
//Request manager takes in the latest packet and processes it as a request
//This means it separates the packet data into the proper components
//It also sends an instruction to the serial transmitter on what to return to the sender
//It also checks to see if the available module is available, and if not waits on it
module RequestManager_update(
                            clock,
                            packet_in,
                            packet_crc_in,
                            packet_available,
                            oxstart,
                            oystart,
                            oxend_width,
                            oyend_height,
                            ocolor,
                            oinstr,
                            oextra1,
                            oextra2,
                            reset,
                            grab_out,
                            nodeWanted,
                            nodeWanted_isAvailable,
                            command_out,
                            sync,
                            transmit_clock,
                            isTransmitting,
                            my_state,
                            packetID,
                            frame_read,
                            frame_write,
                            bug_check
                            );


input wire clock;
input wire [79:0] packet_in;
input wire packet_crc_in;
input wire packet_available;
input wire reset;
input wire nodeWanted_isAvailable;
input wire sync;
input wire isTransmitting;
input wire [7:0] packetID;

output reg [5:0] oinstr;
output reg [11:0] ocolor;
output reg [9:0] oxstart, oxend_width;
output reg [8:0] oystart, oyend_height;
output reg [7:0] oextra1, oextra2;
output reg grab_out;
output reg [3:0] nodeWanted;
output reg command_out;
output reg transmit_clock;

output wire [3:0] my_state;
assign my_state = state;
```

```
output wire  frame_read, frame_write;
reg readFrame, writeFrame;
assign frame_read = readFrame;
assign frame_write = writeFrame;

reg [3:0] state;
reg [7:0] bugCounter;
output wire [7:0] bug_check = lastPacketID;

parameter   [3:0]   waitForPacket = 4'd0,
                    grabPacket = 4'd1,
                    processPacketInformation = 4'd2,
                    waitForAvailableNode = 4'd3,
                    sendCommand = 4'd4,
                    waitForNodeActive = 4'd5,
                    waitForNodeComplete = 4'd6,
                    sendCRCResultBack = 4'd7,
                    packetNotHere = 4'd8;

reg [7:0] lastPacketID;

always @(posedge clock) begin

    if(reset) begin
        state <= waitForPacket;
        oinstr <= 6'b0;
        ocolor <= 12'b0;
        oxstart <= 10'b0;
        oxend_width <= 10'b0;
        oystart <= 9'b0;
        oyend_height <= 9'b0;
        oextra1 <= 8'b0;
        oextra2 <= 8'b0;
        grab_out <= 1'b0;
        nodeWanted <= 4'b0;
        command_out <= 1'b0;
        transmit_clock <= 1'b0;
        lastPacketID <= 8'd255;
        readFrame <= 1'b0;
        writeFrame <= 1'b1;
    end
    else begin
    //if(sync | 1'b1) begin
        case(state)
            waitForPacket: begin
                if(packet_available) begin
                    state <= grabPacket;
                    grab_out <= 1'b0;
                end
                transmit_clock <= 1'b0;
            end

            grabPacket: begin
                grab_out <= 1'b1;
                    state <= processPacketInformation;
                    lastPacketID <= packetID;

            end

            processPacketInformation: begin
                oinstr <= packet_in[79:74];
                ocolor <= packet_in[73:62];
                oxstart <= packet_in[61:52]; //x_start
                oystart <= packet_in[51:43]; //y_start
                oxend_width <= packet_in[42:33]; //x_end/width
                oyend_height <= packet_in[32:24]; //y_end/height
                oextra1 <= packet_in[23:16]; //extra_1
                oextra2 <= packet_in[15:8]; //extra_2
                grab_out <= 1'b0;
                bugCounter <= 0;
                if(packet_in[79:74] == 6'd6) begin
                    state <= packetNotHere;
                    readFrame <= ~readFrame;
                    writeFrame <= ~writeFrame;
                    transmit_clock <= 1'b1;
                end
                else begin
                    state <= waitForAvailableNode;
                end

            end

            waitForAvailableNode: begin
                begin
                    nodeWanted <= oinstr[3:0];
                    if(nodeWanted_isAvailable) begin
                        state <= sendCommand;
                    end
                end
            end

            sendCommand: begin
                command_out <= 1'b1;
                state <= waitForNodeActive;
            end

            waitForNodeActive: begin
                if(~nodeWanted_isAvailable) begin
                    state <= waitForNodeComplete;
                    command_out <= 1'b0;
                end
            end

            waitForNodeComplete: begin
                if(nodeWanted_isAvailable) begin
                    state <= packetNotHere;
                    transmit_clock <= 1'b1;
                end
            end
```

```
            packetNotHere: begin
                if(packet_available == 1'b0) begin
                    state <= sendCRCResultBack;
                end
            end

            sendCRCResultBack: begin //


                    state <= waitForPacket;
                    transmit_clock <= 1'b0;

            end

            default: begin
                state <= waitForPacket;
            end
        endcase
    //end//end if(sync)
    end //end else statement

end




endmodule
```

```
module  Reset_Delay(iCLK,oRESET);
input       iCLK;
output reg  oRESET;
reg [19:0]  Cont;

always@(posedge iCLK)
begin
    if(Cont!=20'hFFFFF)
    begin
        Cont    <=  Cont+20'd1;//edit from +1 to + 20'd1
        oRESET  <=  1'b0;
    end
    else
    oRESET  <=  1'b1;
end

endmodule
```

```verilog
module SerialReceiver_improved_fastClock(
    input CLOCK_50,
    input  UART_RXD,
    output [7:0] lastReceived,
    input reset,
    output errorFlag_out,
    output dataReady_out,
    output [7:0] dataCounter,
    output [17:0] bigCounter,
    output [6:0] errs,
    output [3:0] state_out,
    output [8:0] error_check,
    input timeout,
    input baud_clock,
    input baud_8_clock
    );

    assign errs = errs_reg;
    assign bigCounter = bigCounter_reg;
    reg [6:0] errs_reg;
    reg [12:0] waiter = 13'b0000000000000;  //counter to reduce clock rate

    reg clk_new = 0;  //changes the state of an led every time a bit is sent

    reg [7:0] rxd;

    reg [3:0] currBit;
    reg [8:0] errorCheck;
    reg [3:0] errorBit;

    assign lastReceived = rxd;
    reg [17:0] bigCounter_reg;
    reg [3:0] receiveState;

    assign error_check = errorCheck;

    parameter [3:0] waitForStartBit = 4'd0,
                    collectData = 4'd1,
                    waitForStopBit = 4'd2,
                    error_1 = 4'd3;

    assign state_out = receiveState[3:0];

    reg errorFlag;
    reg dataReady;

    assign dataReady_out = dataReady;
    assign errorFlag_out = errorFlag;

    reg [7:0] dataCounter_reg;
    assign dataCounter = dataCounter_reg;

    reg [4:0] bitClock;
    wire internalBaudClock;

    assign internalBaudClock = (bitClock==4'd10);
```

```
always @(posedge CLOCK_50)
if(receiveState==waitForStartBit)
    bitClock <= 4'b0000;
else
if(baud_8_clock)
    bitClock <= {bitClock[2:0] + 4'b0001} | {bitClock[3], 3'b000};

reg [1:0] RxD_sync_inv;
always @(posedge CLOCK_50) if(baud_8_clock) RxD_sync_inv <= {RxD_sync_inv[0], ~UART_RXD};
// we invert RxD, so that the idle becomes "0", to prevent a phantom character to be received at startup

reg [1:0] RxD_cnt_inv;
reg RxD_bit_inv;

always @(posedge CLOCK_50)
if(baud_8_clock)
begin
    if( RxD_sync_inv[1] && RxD_cnt_inv!=2'b11) RxD_cnt_inv <= RxD_cnt_inv + 2'h1;
    else
    if(~RxD_sync_inv[1] && RxD_cnt_inv!=2'b00) RxD_cnt_inv <= RxD_cnt_inv - 2'h1;

    if(RxD_cnt_inv==2'b00) RxD_bit_inv <= 1'b1;//was 0
    else
    if(RxD_cnt_inv==2'b11) RxD_bit_inv <= 1'b0;//was 1
end



always @ (posedge CLOCK_50)
begin
    if(reset) begin
        receiveState <= 4'd0;
        dataReady <= 0;
        errorFlag <= 0;
        currBit <= 0;
        rxd <= 0;
        dataCounter_reg <= 0;
        bigCounter_reg <= 0;
        errs_reg <= 0;
    end
    else begin
        if(receiveState == waitForStartBit && baud_8_clock) begin
            if(~RxD_bit_inv) begin
                currBit <= 4'd0;
                receiveState <= collectData;
                errorFlag <= 0;
                dataReady <= 0;
            end
        end
        if(receiveState != waitForStartBit && internalBaudClock && baud_8_clock) begin
            case(receiveState)
                collectData: begin
                    rxd[currBit] <= RxD_bit_inv;
                    currBit <= currBit + 4'd1;
                    if(currBit == 4'd7) begin
```

```
                        receiveState <= waitForStopBit;
                    end
                end

                waitForStopBit: begin
                    if(RxD_bit_inv) begin
                        errorFlag <= 0;
                        receiveState <= waitForStartBit;
                        dataReady <= 1;
                        dataCounter_reg <= dataCounter_reg + 8'd1;
                        bigCounter_reg <= bigCounter_reg + 1;
                        errorCheck <= 0;
                        errorBit <= 0;
                    end
                    else begin
                        errorFlag <= 1;
                        errs_reg <= errs_reg + 1;
                        receiveState <= error_1;
                        errorCheck <= {rxd,RxD_bit_inv};
                        errorBit <= 0;
                        //dataReady <= 1;
                    end

                end
                error_1: begin
                    //it needs to resynch
                    errorCheck <= {errorCheck[7:0], RxD_bit_inv};

                    if(errorBit < 7) begin
                        errorBit <= errorBit + 1;
                        dataReady <= 0;
                    end
                    else begin
                        if(errorCheck == 9'h1FF) //all idle bits
                        begin
                            receiveState <= waitForStartBit;
                            errorFlag <= 0;
                            dataReady <= 1;
                        end
                        if(errorCheck[7] == 1'b0 && RxD_bit_inv == 1'b1) begin
                            receiveState <= waitForStartBit;
                            errorFlag <= 0;
                            dataReady <= 1;
                        end
                    end
                end
            endcase
        end
    end
end

endmodule
```

```
module SerialTransmitter_updated(
    input CLOCK_50,
    output  UART_TXD,
    input [7:0] toTransmit,
    input reset,
    input sendNew,
    input baud_clock,
    output sendInProgress
    );

    reg [7:0] txd, last_txd;
    reg [3:0] currBit;
    reg [3:0] transmitState;

    parameter [3:0] idleWait = 4'd0,
                    sendStartBit = 4'd1,
                    sendData = 4'd2,
                    sendStopBit = 4'd3;

    reg txd_out;
    reg should_send;
    reg sendInProgressReg;


    assign UART_TXD = txd_out;
    assign sendInProgress = sendInProgressReg;


    always @(posedge CLOCK_50)
    begin
        if(sendNew & ~reset & ((transmitState == idleWait) )) begin
            should_send <= 1'b1;
        end
        if(reset) begin
            transmitState <= 4'd0;
            currBit <= 0;
            txd <= 0;
            txd_out <= 1;
            should_send <= 0;
            sendInProgressReg <= 0;

        end
        else begin
            if (baud_clock) //9600 times a second (aprox)
            begin


                case(transmitState)
                    idleWait: begin
                        txd_out <= 1'b1;
                        sendInProgressReg <= 1'b0;
                        if(should_send) begin
                            txd <= toTransmit;
                            last_txd <= toTransmit;
                            transmitState <= sendStartBit;
                            should_send <= 0;
                        end
                    end

                    sendStartBit: begin
                        txd_out <= 1'b0; //the start bit
                        transmitState <= sendData;
                        currBit <= 4'd0;
                        sendInProgressReg <= 1'b1;
                    end

                    sendData: begin
                        txd_out <= txd[currBit];
                        currBit <= currBit + 4'd1;
                        if(currBit == 4'd7) begin
                            transmitState <= sendStopBit;
                        end
                    end

                    sendStopBit: begin
                        txd_out <= 1'b1;
                        sendInProgressReg <= 1'b0;
                        transmitState <= idleWait;
                    end

                endcase


            end

        end

    end
endmodule
```

```
module square_draw(x1,y1,width, height, VGA_SYNC, VGA_CLK,
                                    data_reg, reset, color, command,
                                    available,
                                    my_id,
                                    id_req,
                                    xAddr, yAddr,
                                    write_clk,
                                    is_write_finished
                                    );

input wire signed [10:0] x1,y1;
input wire signed [10:0] width, height;
input wire VGA_SYNC;
input wire VGA_CLK;
input wire [3:0] my_id, id_req;

output reg [15:0] data_reg; //memory data register  for SRAM

input wire reset;
input wire [15:0] color;
input wire [1:0] command;
output wire available;
reg available_reg;
assign available = available_reg;

output reg write_clk;
input wire is_write_finished;

output reg [9:0] xAddr;
output reg [8:0] yAddr;

reg signed  [10:0] x, x_end;
reg signed  [10:0] y, y_end;


reg [4:0] draw_state, next_state;
parameter start = 5'd0,
            loop = 5'd1,
            finish = 5'd2,
            north = 5'd3,
            south = 5'd4,
            east = 5'd5,
            west = 5'd6,
            write_state = 5'd7;

always @ (posedge VGA_CLK) begin
    if(reset) begin
        draw_state <= finish;

        data_reg <= 16'b0;                          //write all zeros (black)
        available_reg <= 1'b0;
        write_clk <= 1'b0;
    end
    else if (VGA_SYNC) begin
        case (draw_state)
```

```
write_state: begin
    if(is_write_finished) begin
        draw_state <= next_state;
        write_clk <= 1'b0;
    end
end

north: begin

    if(x < x_end) begin

        xAddr <= x[9:0];
        yAddr <= y[8:0];
        next_state <= north;
        draw_state <= write_state;
        write_clk <= 1'b1;
        data_reg <= color;
        x <= x + 11'd1;
    end
    else begin

        xAddr <= x[9:0];
        yAddr <= y[8:0];
        next_state <= south;
        draw_state <= write_state;
        write_clk <= 1'b1;
        data_reg <= color;
        y <= y_end;
        x <= x1;

    end
end

south: begin
    if(x < x_end) begin

        xAddr <= x[9:0];
        yAddr <= y[8:0];
        next_state <= south;
        draw_state <= write_state;
        write_clk <= 1'b1;
        data_reg <= color;
        x <= x + 11'd1;
    end
    else begin

        xAddr <= x[9:0];
        yAddr <= y[8:0];
        next_state <= east;
        draw_state <= write_state;
        write_clk <= 1'b1;
        data_reg <= color;

        x <= x1;
        y <= y1;
    end
```

```
        end

        east: begin
            if(y < y_end) begin

                xAddr <= x[9:0];
                yAddr <= y[8:0];
                next_state <= east;
                draw_state <= write_state;
                write_clk <= 1'b1;
                data_reg <= color;
                y <= y + 11'd1;
            end
            else begin

                xAddr <= x[9:0];
                yAddr <= y[8:0];
                next_state <= west;
                draw_state <= write_state;
                write_clk <= 1'b1;
                data_reg <= color;

                x <= x_end;
                y <= y1;
            end
        end

        west: begin
            if(y < y_end) begin

                xAddr <= x[9:0];
                yAddr <= y[8:0];
                next_state <= west;
                draw_state <= write_state;
                write_clk <= 1'b1;
                data_reg <= color;
                y <= y + 11'd1;
            end
            else begin

                xAddr <= x[9:0];
                yAddr <= y[8:0];
                next_state <= finish;
                draw_state <= write_state;
                write_clk <= 1'b1;
                data_reg <= color;

            end
        end

        finish: begin
            if(command[1] && (my_id == id_req)) begin
                draw_state <= north;
                x <= x1;
                y <= y1;
                x_end <= x1 + width;
```

```
                    y_end <= y1 + height;
                    available_reg <= 1'b0;
                end
                else begin
                    available_reg <= 1'b1;
                end

            end

            default: begin

                draw_state <= finish;
            end
        endcase
    end
    else
    begin

    end
end



endmodule
```

```verilog
module square_fill(x1,y1,width, height, VGA_SYNC, VGA_CLK,
                                    data_reg, reset, color, command,
                                    available,
                                    my_id,
                                    id_req,
                                    xAddr, yAddr,
                                    write_clk,
                                    is_write_finished
                                    );

input wire signed [10:0] x1,y1;
input wire signed [10:0] width, height;
input wire VGA_SYNC;
input wire VGA_CLK;
input wire [3:0] my_id, id_req;

output reg [15:0] data_reg; //memory data register  for SRAM

input wire reset;
input wire [15:0] color;
input wire [1:0] command;
output wire available;
reg available_reg;
assign available = available_reg;

output reg write_clk;
input wire is_write_finished;

output reg [9:0] xAddr;
output reg [8:0] yAddr;

reg signed  [10:0] x, x_end;
reg signed  [10:0] y, y_end;


reg [4:0] draw_state, next_state;
parameter start = 5'd0,
            loop = 5'd1,
            finish = 5'd2,
            north = 5'd3,
            south = 5'd4,
            east = 5'd5,
            west = 5'd6,
            write_state = 5'd7;

always @ (posedge VGA_CLK) begin
    if(reset) begin
        draw_state <= finish;

        data_reg <= 16'b0;                          //write all zeros (black)
        available_reg <= 1'b0;
        write_clk <= 1'b0;
    end
    else if (VGA_SYNC) begin
        case (draw_state)
```

```
write_state: begin
    if(is_write_finished) begin
        draw_state <= next_state;
        write_clk <= 1'b0;
    end
end

loop: begin

    if(x < x_end) begin

        xAddr <= x[9:0];
        yAddr <= y[8:0];
        write_clk <= 1'b1;
        draw_state <= write_state;
        next_state <= loop;
        data_reg <= color;
        x <= x + 11'd1;
    end
    else begin

        xAddr <= x[9:0];
        yAddr <= y[8:0];
        write_clk <= 1'b1;
        draw_state <= write_state;

        data_reg <= color;
        x <= x1;
        if(y < y_end) begin
            y <= y + 11'd1;
            next_state <= loop;
        end
        else begin

            next_state <= finish;
        end

    end
end

finish: begin
    if(command[1] && (my_id == id_req)) begin
        draw_state <= loop;
        x <= x1;
        y <= y1;
        x_end <= x1 + width;
        y_end <= y1 + height;
        available_reg <= 1'b0;

    end
    else begin
        available_reg <= 1'b1;
    end

end
```

```
            default: begin

                draw_state <= finish;
            end
        endcase
    end
    else
    begin

    end
end



endmodule
```

```
//Module Availability MUX

module availableMUX(select, availability_out,
                            LD_available,
                            CD_available, CF_available,
                            SD_available, SF_available,
                            TW_available,
                            FU_available,
                            clk,
                            enable);

input wire [3:0] select;
input wire enable;

output wire availability_out;
reg availablity_out_reg;
assign availability_out = availablity_out_reg;

input wire LD_available, CD_available, CF_available, SD_available, SF_available, TW_available,
FU_available;
input wire clk;

parameter   [3:0]   circle_fill = 4'd0,
                    circle_draw = 4'd1,
                    square_fill = 4'd2,
                    square_draw = 4'd3,
                    line_draw = 4'd4,
                    text_write = 4'd5,
                    frame_update = 4'd6,
                    idle = 4'd15;

always @ (posedge clk) begin
    if(enable) begin
        case(select)
            circle_fill: begin
                availablity_out_reg <= CF_available;
            end
            circle_draw: begin
                availablity_out_reg <= CD_available;
            end
            square_fill: begin
                availablity_out_reg <= SF_available;
            end
            square_draw: begin
                availablity_out_reg <= SD_available;
            end
            line_draw: begin
                availablity_out_reg <= LD_available;
            end
            text_write: begin
                availablity_out_reg <= TW_available;
            end
            frame_update: begin
                availablity_out_reg <= FU_available;
            end
            idle: begin
```

```
                availablity_out_reg <= 1'b0;
            end
            default: begin
                availablity_out_reg <= 1'b0;
            end
        endcase
    end
end


endmodule
```

```
module BaudGenerator( clock_in,
                        baud_clock_out);


input wire clock_in;
output wire baud_clock_out;

//Code Referenced from www.fpga4fun.com/SerialInterface2.html
parameter ClkFrequency = 50000000; // MHz
parameter Baud = 115200;
parameter BaudGeneratorAccWidth = 16;
//parameter BaudGeneratorInc = (Baud << BaudGeneratorAccWidth) / ClkFrequency;
parameter BaudGeneratorInc = ((Baud << (BaudGeneratorAccWidth - 4)) + (ClkFrequency>>5))/(
ClkFrequency>>4);

reg [BaudGeneratorAccWidth:0] BaudGeneratorAcc;
always @(posedge clock_in) begin
    BaudGeneratorAcc <= BaudGeneratorAcc[BaudGeneratorAccWidth-1:0] + BaudGeneratorInc;
end

wire BaudTick = BaudGeneratorAcc[BaudGeneratorAccWidth];

assign baud_clock_out = BaudTick;

endmodule
```

```
module BaudGenerator_8Times( clock_in,
                        baud_clock_out);


input wire clock_in;
output wire baud_clock_out;

//Code Referenced from www.fpga4fun.com/SerialInterface2.html
parameter ClkFrequency = 50000000; // MHz
parameter Baud = 307200;
parameter BaudGeneratorAccWidth = 16;
//parameter BaudGeneratorInc = (Baud << BaudGeneratorAccWidth) / ClkFrequency;
parameter BaudGeneratorInc = ((Baud << (BaudGeneratorAccWidth - 4)) + (ClkFrequency>>5))/(
ClkFrequency>>4);

reg [BaudGeneratorAccWidth:0] BaudGeneratorAcc;
always @(posedge clock_in) begin
    BaudGeneratorAcc <= BaudGeneratorAcc[BaudGeneratorAccWidth-1:0] + BaudGeneratorInc;
end

wire BaudTick = BaudGeneratorAcc[BaudGeneratorAccWidth];

assign baud_clock_out = BaudTick;

endmodule
```

```
module BaudGenerator_fast( clock_in,
                        baud_clock_out);


input wire clock_in;
output wire baud_clock_out;

//Code Referenced from www.fpga4fun.com/SerialInterface2.html
parameter ClkFrequency = 50000000; // MHz
parameter Baud = 921600;
parameter BaudGeneratorAccWidth = 16;
//parameter BaudGeneratorInc = (Baud << BaudGeneratorAccWidth) / ClkFrequency;
parameter BaudGeneratorInc = ((Baud<<(BaudGeneratorAccWidth-7))+(ClkFrequency>>8))/(ClkFrequency
>>7);


reg [BaudGeneratorAccWidth:0] BaudGeneratorAcc;
always @(posedge clock_in) begin
    BaudGeneratorAcc <= BaudGeneratorAcc[BaudGeneratorAccWidth-1:0] + BaudGeneratorInc;
end

wire BaudTick = BaudGeneratorAcc[BaudGeneratorAccWidth];

assign baud_clock_out = BaudTick;

endmodule
```

```verilog
module circle_draw(x1,y1,radius, VGA_SYNC, VGA_CLK,
                                    data_reg, reset, color, command,
                                    available,
                                    my_id,
                                    id_req,
                                    xAddr, yAddr,
                                    write_clk,
                                    is_write_finished
                                    );

input wire signed [10:0] x1,y1;
input wire signed [10:0] radius;
input wire VGA_SYNC;
input wire VGA_CLK;
input wire [3:0] my_id, id_req;
output reg [15:0] data_reg; //memory data register  for SRAM

input wire reset;
input wire [15:0] color;
input wire [1:0] command;
output wire available;
reg available_reg;
assign available = available_reg;

output reg write_clk;
input wire is_write_finished;

output reg [9:0] xAddr;
output reg [8:0] yAddr;

reg signed  [10:0] x;
reg signed  [10:0] y;
reg signed  [10:0] p;
reg signed [10:0] draw_x, draw_y;

reg [4:0] draw_state, next_state;
parameter start = 5'd0,
            loop = 5'd1,
            finish = 5'd2,
            calc = 5'd3,
            q1 = 5'd4,
            q2 = 5'd5,
            q3 = 5'd6,
            q4 = 5'd7,
            q5 = 5'd8,
            q6 = 5'd9,
            q7 = 5'd10,
            q8 = 5'd11,
            write_state = 5'd12;

always @ (posedge VGA_CLK) begin
    if(reset) begin
        draw_state <= finish;

        data_reg <= 16'b0;                              //write all zeros (black)
        available_reg <= 1'b0;
        write_clk <= 1'b0;
    end
    else if (VGA_SYNC) begin
        case (draw_state)

            write_state: begin
                if(is_write_finished) begin
                    draw_state <= next_state;
                    write_clk <= 1'b0;
                end
            end

            calc: begin

                if(x < y) begin
                    //put the pixels
                    draw_x <= x1 + x;
                    draw_y <= y1 + y;
                    draw_state <= q1;

                    //draw_state <= finish;
                end
                else begin
                    draw_state <= finish;
                    //we <= 1'b1;
                end

            end

            q1: begin

                draw_x <= x1 - x;
                draw_y <= y1 + y;

                xAddr <= draw_x[9:0];
                yAddr <= draw_y[8:0];
                next_state <= q2;
                draw_state <= write_state;
                write_clk <= 1'b1;

                data_reg <= color;

            end

            q2: begin

                draw_x <= x1 + x;
                draw_y <= y1 - y;

                xAddr <= draw_x[9:0];
                yAddr <= draw_y[8:0];
                next_state <= q3;
                draw_state <= write_state;
                write_clk <= 1'b1;
                data_reg <= color;
                //draw_state <= q3;
            end

            q3: begin

                draw_x <= x1 - x;
                draw_y <= y1 - y;

                xAddr <= draw_x[9:0];
                yAddr <= draw_y[8:0];
                next_state <= q4;
                draw_state <= write_state;
                write_clk <= 1'b1;
                data_reg <= color;
                //draw_state <= q4;
            end

            q4: begin

                draw_x <= x1 + y;
                draw_y <= y1 + x;

                xAddr <= draw_x[9:0];
                yAddr <= draw_y[8:0];
                next_state <= q5;
                draw_state <= write_state;
                write_clk <= 1'b1;
                data_reg <= color;

            end

            q5: begin

                draw_x <= x1 - y;
                draw_y <= y1 + x;

                xAddr <= draw_x[9:0];
                yAddr <= draw_y[8:0];
                next_state <= q6;
                draw_state <= write_state;
                write_clk <= 1'b1;
                data_reg <= color;
                //draw_state <= q6;
            end

            q6: begin

                draw_x <= x1 + y;
                draw_y <= y1 - x;

                xAddr <= draw_x[9:0];
                yAddr <= draw_y[8:0];
                next_state <= q7;
                draw_state <= write_state;
                write_clk <= 1'b1;
                data_reg <= color;
```

```
    end

    q7: begin

        draw_x <= x1 - y;
        draw_y <= y1 - x;

        xAddr <= draw_x[9:0];
        yAddr <= draw_y[8:0];
        next_state <= q8;
        draw_state <= write_state;
        write_clk <= 1'b1;
        data_reg <= color;

    end

    q8: begin

        xAddr <= draw_x[9:0];
        yAddr <= draw_y[8:0];
        next_state <= calc;
        draw_state <= write_state;
        write_clk <= 1'b1;
        data_reg <= color;
        if (p < 0) begin
            p <= p + x + x + x + x + 11'd6;
            x <= x + 11'd1;
        end
        else begin
            p <= p + x + x + x + x - y - y - y - y + 11'd10;
            x <= x + 11'd1;
            y <= y - 11'd1;
        end

    end


    finish: begin
        if(command[1] && (my_id == id_req)) begin
            draw_state <= calc;
            x <= 11'b0;
            y <= radius;
            p <= 11'd3 - radius - radius;
            available_reg <= 1'b0;
        end
        else begin
            available_reg <= 1'b1;
        end

    end

    default: begin

        draw_state <= finish;
    end
endcase
```

```
    end
    else
    begin

    end
end




endmodule
```

```
module circle_fill_better(x1,y1,radius, VGA_SYNC, VGA_CLK,
                                    reset, color, command,
                                    data_reg,
                                    available,
                                    my_id,
                                    id_req,
                                    xAddr, yAddr,
                                    write_clk,
                                    is_write_finished
                                    );

input wire signed [10:0] x1,y1;
input wire signed [10:0] radius;
input wire VGA_SYNC;
input wire VGA_CLK;

output reg write_clk;
input wire is_write_finished;

input wire [3:0] my_id, id_req;

output reg [9:0] xAddr;
output reg [8:0] yAddr;


reg signed [10:0] x1start, y1start, x2start, y2start;




output reg [15:0] data_reg; //memory data register  for SRAM

output wire available;
reg available_reg;
assign available = available_reg;


input wire reset;
input wire [15:0] color;
input wire [1:0] command;


reg signed  [10:0] x;
reg signed  [10:0] y;
reg signed  [10:0] p;
reg signed [10:0] draw_x, draw_y;

reg [4:0] draw_state;
reg [4:0] next_state;
parameter start = 5'd0,
            loop = 5'd1,
            finish = 5'd2,
            calc = 5'd3,
            q1 = 5'd4,
            q2 = 5'd5,
            q3 = 5'd6,
            q4 = 5'd7,
            q5 = 5'd8,
            q6 = 5'd9,
            q7 = 5'd10,
            q8 = 5'd11,
            wait12done = 5'd12,
            wait34done = 5'd13,
            wait56done = 5'd14,
            wait78done = 5'd15,
            set12 = 5'd16,
            set34 = 5'd17,
            set56 = 5'd18,
            set78 = 5'd19,
            write_state = 5'd20;

always @ (posedge VGA_CLK) begin
    if(reset) begin
        draw_state <= finish;

        data_reg <= 16'b0;                          //write all zeros (black)
        available_reg <= 1'b0;
        write_clk <= 1'b0;
    end
    else if (VGA_SYNC) begin
        case (draw_state)

            write_state: begin
                if(is_write_finished) begin
                    draw_state <= next_state;
                    write_clk <= 1'b0;
                end
            end

            calc: begin

                if(x <= y) begin
                    draw_state <= set12;

                end
                else begin
                    draw_state <= finish;
                end

            end

            set12: begin

                x1start <= x1 + x;
                y1start <= y1 + y;
                x2start <= x1 - x;
                y2start <= y1 + y;
                draw_state <= wait12done;
            end

            wait12done: begin

                if(x2start < x1start) begin
                    xAddr <= x2start[9:0];
                    yAddr <= y1start[8:0];
                    write_clk <= 1'b1;
                    draw_state <= write_state;
                    next_state <= wait12done;
                    data_reg <= color;
                    x2start <= x2start + 11'd1;
                end
                else begin

                    xAddr <= x2start[9:0];
                    yAddr <= y1start[8:0];
                    write_clk <= 1'b1;
                    draw_state <=  write_state;
                    next_state <= set34;
                    data_reg <= color;
                    //draw_state <= set34;
                end

            end

            set34: begin

                x1start <= x1 + x;
                y1start <= y1 - y;
                x2start <= x1-x;
                y2start <= y1-y;
                draw_state <= wait34done;

            end

            wait34done: begin

                if(x2start < x1start) begin

                    xAddr <= x2start[9:0];
                    yAddr <= y1start[8:0];
                    write_clk <= 1'b1;
                    draw_state <=  write_state;
                    next_state <= wait34done;
                    data_reg <= color;
                    x2start <= x2start + 11'd1;
                end
                else begin

                    xAddr <= x2start[9:0];
                    yAddr <= y1start[8:0];
                    write_clk <= 1'b1;
                    draw_state <=  write_state;
                    next_state <= set56;
                    data_reg <= color;

                end
            end
```

```
set56: begin

    x1start <= x1+y;
    y1start <= y1+x;
    x2start <= x1-y;
    y2start <= y1+x;
    draw_state <= wait56done;
end

wait56done: begin

    if(x2start < x1start) begin

        xAddr <= x2start[9:0];
        yAddr <= y1start[8:0];
        write_clk <= 1'b1;
        draw_state <=  write_state;
        next_state <= wait56done;
        data_reg <= color;
        x2start <= x2start + 11'd1;
    end
    else begin

        xAddr <= x2start[9:0];
        yAddr <= y1start[8:0];
        write_clk <= 1'b1;
        draw_state <=  write_state;
        next_state <= set78;
        data_reg <= color;

    end
end

set78: begin

    x1start <= x1+y;
    y1start <= y1-x;
    x2start <= x1-y;
    y2start <= y1-x;
    draw_state <= wait78done;
end

wait78done: begin

    if(x2start < x1start) begin

        xAddr <= x2start[9:0];
        yAddr <= y1start[8:0];
        write_clk <= 1'b1;
        draw_state <=  write_state;
        next_state <= wait78done;
        data_reg <= color;
        x2start <= x2start + 11'd1;
    end
    else begin
```

```
                xAddr <= x2start[9:0];
                yAddr <= y1start[8:0];
                write_clk <= 1'b1;
                draw_state <= write_state;
                next_state <= q8;
                data_reg <= color;
                //draw_state <= q8;
            end

        end

        q8: begin

            if (p < 0) begin
                p <= p + x + x + x + x + 11'd6;
                x <= x + 11'd1;
            end
            else begin
                p <= p + x + x + x + x - y - y - y - y + 11'd10;
                x <= x + 11'd1;
                y <= y - 11'd1;
            end
            draw_state <= calc;
        end


        finish: begin
            if(command[1] && (my_id == id_req)) begin
                draw_state <= calc;
                x <= 11'b0;
                y <= radius;
                p <= 11'd3 - radius - radius;
                available_reg <= 1'b0;
            end
            else begin
                available_reg <= 1'b1;
            end



        end

        default: begin
            draw_state <= finish;

        end
      endcase
    end
    else
    begin

    end

end

endmodule
```

```verilog
module Color_MUX(
                    cCF, cCD, cSD, cSF, cLD, cTW, cIDLE,
                    clock, select,
                    cOut



);

input wire [15:0] cCF, cCD, cSD, cSF, cLD, cTW, cIDLE;
input wire clock;
input wire [3:0] select;
output wire [15:0] cOut;
reg [15:0] cReg;
assign cOut = cReg;

parameter   [3:0]   circle_fill = 4'd0,
                    circle_draw = 4'd1,
                    square_fill = 4'd2,
                    square_draw = 4'd3,
                    line_draw = 4'd4,
                    text_write = 4'd5,
                    frame_update = 4'd6,
                    idle = 4'd15;


always @(posedge clock) begin
    case(select)
        circle_fill: begin cReg <= cCF; end
        circle_draw: begin cReg <= cCD; end
        square_fill: begin cReg <= cSF; end
        square_draw: begin cReg <= cSD; end
        line_draw: begin cReg <= cLD; end
        text_write: begin cReg <= cTW; end
        idle: begin cReg <= cIDLE; end
        default: begin
            cReg <= 16'hF000;
        end
    endcase
end

endmodule
```

```
//CRC Module

module CRC(
                    packet_in,
                    packet_ready_in,
                    crc_result_out,
                    reset,
                    crc_out
                    );

output wire crc_result_out;
input wire [79:0] packet_in;
input wire packet_ready_in;
input wire reset;

wire [7:0] xor_result;
output wire [7:0] crc_out;
wire one_true;

assign xor_result =
                                        packet_in[79:72]   ^ packet_in[71:64]   ^ packet_in[63:
56] ^
                                        packet_in[55:48]   ^ packet_in[47:40]   ^ packet_in[39:
32] ^
                                        packet_in[31:24]   ^ packet_in[23:16]   ^ packet_in[15:8
];



assign one_true = (xor_result == packet_in[7:0]) ? 1'b1 : 1'b0;


assign crc_result_out = one_true /*& two_true*/ & ~reset;
assign crc_out = xor_result;




endmodule
```

```
module DataWriter(
                    x_addr,  //x coordinate of pixel to draw to
                    y_addr,  //y coordinate of pixel to draw to
                    color_data,  //the color data
                    isFinished,  //flag if the write has completed
                    clock,  //the clock to handle the writing
                    sync,  //to keep us in sync with VGA
                    reset,  // a reset signal
                    write_req_clk,  //clock that starts a new write request
                    read_addr_out,  //the address we want to read data from
                    data_write_out,  //the new data that we will write to SRAM
                    we_out,  //the write enable signal (0 = write, 1 = read)
                    read_data_in,  //the data that was read
                    currFrame_out,
                    frame_sel,
                    drawMode_sel,
                    Coord_X,
                    Coord_Y

);

input wire clock, reset, sync;
input wire [9:0] Coord_X, Coord_Y;
input wire write_req_clk;

input wire [2:0] drawMode_sel;
input wire frame_sel;

input wire [9:0] x_addr;
input wire [8:0] y_addr;
input wire [15:0] color_data;
input wire [15:0] read_data_in;

output wire [1:0] currFrame_out;
assign currFrame_out = {frame_sel,frame_sel};

output reg isFinished;
output reg [17:0] read_addr_out;
output reg we_out;
output reg [15:0] data_write_out;


parameter [2:0] sixteenBit_single_320x240 = 3'd0,
                eightBit_single_640x480 = 3'd1,
                eightBit_double_320x240 = 3'd2,
                fourBit_double_640x480 = 3'd3,
                fiveBit_triple_320x240 = 3'd4,
                fourBit_quad_320x240 = 3'd5,
                twoBit_quad_640x480 = 3'd6,
                foreground_background = 3'd7;

reg [1:0] writeMode;


parameter [1:0] waitForRequest = 2'd0,
                loadData = 2'd1,
                  writeData = 2'd2,
                  sendACK = 2'd3;


reg [15:0] color_formated;
reg wasWriting;



always @(posedge clock) begin
    if(reset) begin
        isFinished <= 1'b1;
        writeMode <= waitForRequest;
        read_addr_out <= 0;
        we_out <= 1'b1;
        data_write_out <= 0;
        color_formated <= 0;
        case(drawMode_sel)
            sixteenBit_single_320x240: begin read_addr_out <= {Coord_X[9:1],Coord_Y[9:1]} ;end
            eightBit_single_640x480: begin read_addr_out <= {Coord_X[9:1],Coord_Y[8:0]
/*Coord_Y[9:1]*/} ;end
            eightBit_double_320x240: begin read_addr_out <= {Coord_X[9:1],Coord_Y[9:1]} ;end
            fourBit_double_640x480: begin read_addr_out <= {Coord_X[9:1],Coord_Y[8:0]/*Coord_Y[9:1]*/
} ;end
        endcase

            we_out <= 1'b0;
            data_write_out <= 16'hFFFF;
            wasWriting <= 1'b0;
    end
    else begin
        if(sync) begin
            case(writeMode)
                waitForRequest: begin
                    isFinished <= 1'b0;
                    if(write_req_clk | wasWriting) begin
                        writeMode <= loadData;
                        wasWriting <= 1'b1;
                        case(drawMode_sel)
                            sixteenBit_single_320x240:  begin color_formated <= color_data; end
                            eightBit_single_640x480:    begin color_formated <= {color_data[15:
13],color_data[11:9],color_data[7:6],color_data[15:13],color_data[11:9],color_data[7:6]}; end
//RRRGGGBB
                            eightBit_double_320x240:    begin color_formated <= {color_data[15:
13],color_data[11:9],color_data[7:6],color_data[15:13],color_data[11:9],color_data[7:6]}; end
//RRRGGGBB
                            fourBit_double_640x480:     begin color_formated <= {4{color_data[15
],color_data[11:10],color_data[7]}}; end //RGGB
                            fiveBit_triple_320x240:     begin color_formated <= {({3{color_data[
15:14],color_data[11:10],color_data[7]}}),1'b0}; end //RRGGB
                            fourBit_quad_320x240:       begin color_formated <= {4{color_data[15
],color_data[11:10],color_data[7]}}; end //RGGB
                            twoBit_quad_640x480:        begin color_formated <= 0; end
                            foreground_background:      begin color_formated <= 0; end
                            default:                    begin color_formated <= 16'h0F00; end
                        endcase
                    end
                end
                loadData: begin //TODO BOUND CHECK
                    we_out <= 1'b1; //we want to read
                    case(drawMode_sel)
                        sixteenBit_single_320x240:  begin read_addr_out <= {x_addr[8:0],y_addr[8
:0]}; end
                        eightBit_single_640x480:    begin read_addr_out <= {x_addr[9:1],y_addr[8
:0]}; end //{1'b0,y_addr[8:1]}}; end
                        eightBit_double_320x240:    begin read_addr_out <= {x_addr[8:0],y_addr[8
:0]}; end
                        fourBit_double_640x480:     begin read_addr_out <= {x_addr[9:1],y_addr[8
:0]}; end
                        fiveBit_triple_320x240:     begin read_addr_out <= {x_addr[8:0],y_addr[8
:0]}; end
                        fourBit_quad_320x240:       begin read_addr_out <= {x_addr[8:0],y_addr[8
:0]}; end
                        twoBit_quad_640x480:        begin read_addr_out <= {x_addr[9:1],{1'b0,
y_addr[8:1]}}; end
                        foreground_background:      begin read_addr_out <= {x_addr[8:0],y_addr[8
:0]}; end//CHANGE
                        default:                    begin read_addr_out <= 0; end
                    endcase
                    writeMode <= writeData;
                    wasWriting <= 1'b1;
                end
                writeData: begin
                    we_out <= 1'b0; //we want to write
                    case(drawMode_sel) //assuming color_data is already formatted
                        sixteenBit_single_320x240:  begin data_write_out <= color_formated; end
                        eightBit_single_640x480:    begin data_write_out <= (x_addr[0] == 1'b1)
? {(read_data_in & 16'hFF00) | (16'h00FF & color_formated)} : {(read_data_in & 16'h00FF) | (
16'hFF00 & color_formated)}; end
                        eightBit_double_320x240:    begin data_write_out <= (frame_sel == 1'b1)
? {(read_data_in & 16'hFF00) | (16'h00FF & color_formated)} : {(read_data_in & 16'h00FF) | (
16'hFF00 & color_formated)}; end
                        fourBit_double_640x480:     begin data_write_out <= x_addr[0] ?
                                                                                    (frame_sel ?
 ((read_data_in & 16'hFF0F) | (16'h00F0 & color_formated)) : ((read_data_in & 16'hFFF0) | (
16'h000F & color_formated)) ) :
                                                                                    (frame_sel ?
 ((read_data_in & 16'hF0FF) | (16'h0F00 & color_formated)) : ((read_data_in & 16'h0FFF) | (
16'hF000 & color_formated)) ); end
                        fiveBit_triple_320x240:     begin data_write_out <= color_data; end //TODO
                        fourBit_quad_320x240:       begin data_write_out <= color_data; end //TODO
                        twoBit_quad_640x480:        begin data_write_out <= color_data; end //TODO
                        foreground_background:      begin data_write_out <= color_data; end //TODO
                        default:                    begin data_write_out <= 16'h00F0; end
                    endcase
                    writeMode <= sendACK;
                    wasWriting <= 1'b0;
                end
                sendACK: begin
                    we_out <= 1'b1; //not writing anymore, so set to read to be safe
                    isFinished <= 1'b1;
                    writeMode <= waitForRequest;
                    wasWriting <= 1'b0;
                end
            endcase
        end
        else
        begin
            case(drawMode_sel)
                sixteenBit_single_320x240: begin read_addr_out <= {Coord_X[9:1],Coord_Y[9:1]} ;
end
                eightBit_single_640x480: begin read_addr_out <= {Coord_X[9:1],Coord_Y[8:0]
/*Coord_Y[9:1]*/} ;end
                eightBit_double_320x240: begin read_addr_out <= {Coord_X[9:1],Coord_Y[9:1]} ;end
                fourBit_double_640x480: begin read_addr_out <= {Coord_X[9:1],Coord_Y[8:0]
/*Coord_Y[9:1]*/} ;end
            endcase
            writeMode <= waitForRequest;//loadData;
            //isFinished <= 1'b1;
            we_out <= 1'b1;
        end
    end
end




endmodule
```