Creation of Asteroids Game Using Verilog and Xilinx FPGA

Shield Xiao & James Verrill

6.111 Fall 2005 – Final Project

Revision Number: 17 Saved On: 14-Dec-05

Abstract

The aim of the project is to create on the labkit a version of the popular computer game 'Asteroids'. Asteroids is a game set in 2 dimensions featuring a spaceship in a field of moving asteroids. The asteroids move randomly and spin. The player controls the spaceship, moving it around the playing field and shooting at the asteroids to destroy them. When hit by the spaceship's weapon the asteroids break down into smaller asteroids and eventually are removed from the field. The game is completed when all asteroids have been destroyed. The game is lost if the spaceship collides with an asteroid.

We approached the project in a modular fashion, attempting to create modules which computed the required lines for a frame, computed the pixels on those lines, and then generated the required frames. We were successful in creating a vast array of these modules, but were unsuccessful in integrating them to create the entire project.

Contents

List of Figures

1. Overview

1.1. Introduction

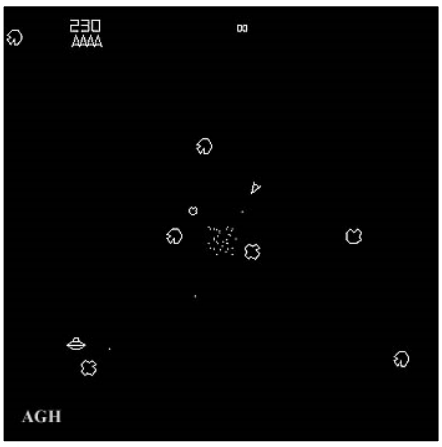


Figure 1 - Screenshot from the 1978 Arcade Version of 'Asteroids' .Created by Atari to compete with 'Space Invaders'. The screenshot shows how the asteroids are created as a collection of line vectors

Throughout the history of computing and electronics people have sought to harness technology not only to increase productivity, but also to provide a means of entertainment. Indeed, one of the main driving forces behind the increasing need for more and more powerful home computers may be considered to be the computer games industry. Every generation of game technology at its heart, consider the leap from text based games to graphic and the step up from 2D graphics to 3D graphics. Each step immersing the user more and increasing the reality factor of the game.

Asteroids was itself revolutionary and has deserved place in computer game history. The original was created in 1978 by Atari and was the company's response to the popular and famous 'Space Invaders' game. 'Space Invaders' like many popular games of the time was based upon the creation of sprites. These sprites were fixed bitmap representation that were moved around the screen to create gameplay. Often these graphics were very crude and there was a limitation to the number of possible different sprites in one game caused by the need for memory to store them in. In this area 'Asteroids' was revolutionary – instead of using sprites, 'Asteroids' was the first computer game to use vectors. By using these vectors an unlimited number of possible asteroids, which through simple mathematics, could be manipulated to provide the effects of rotation and translation, and thus simulate more degrees of freedom in an object. This same basis in mathematics is used in many modern games and with progression to the creation of facets from lines and then object from facets and complete 3D representation for an object is created.

The purpose of the project is to recreate, on a Xilinx Field Programmable Gate Array (FPGA), the classic version of 'Asteroids' by programming hardware functionality using the verilog description language.

2. Description

2.1. Design Methodology

The task of recreating the game was initially divided into 2 main sections: graphical output and computational. Each section would have several key responsibilities that would be independent of each other meaning that they could be developed entirely separately.

The computational section provides a stream of x and y co-ordinates where a pixel should be placed in the next frame. The pixels can appear randomly and anywhere on the screen at any time. The graphical section takes these pixels and draws them onto the screen appropriately.

The breakdown of each section into a series of modules is shown in Figure 2 - Block Diagram of Asteroids System..

2.2. Design Constraints

The project was implemented using a Xilinx FPGA and associated Verilog programming tools. The inputs and outputs available to be used on the machine were:

- 8 LEDs
- 9 Push button switches
- 8 Switches
- 16 Hex Outputs
- A VGA Display
- A 64 Bit Logic Analyser
- A PS2 mouse and Keyboard
- Several interface ports that had no bearing or use

When considering the implementation of the project, the following constraints were placed upon the use of these inputs:

- Output of Buttons is normally high, when pushed output is low.
- LEDs are on when the input to them is low and off when the input to them is high.
- Hex outputs were controlled using a provided Verilog module [2].

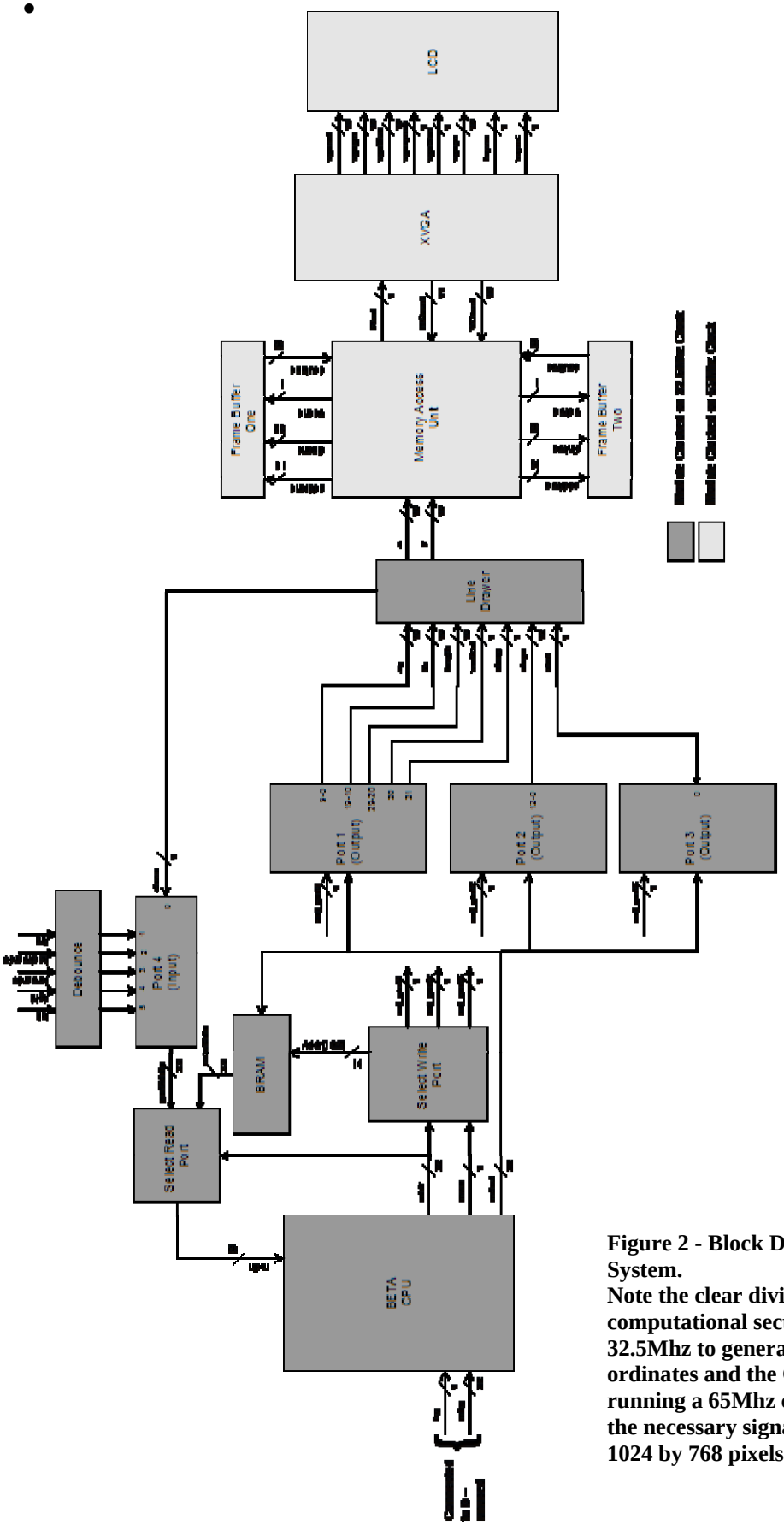


Figure 2 - Block Diagram of Asteroids System.
Note the clear divide between the computational section which runs at 32.5Mhz to generate a series of pixel co-ordinates and the Graphics systems running a 65Mhz clock to be able to create the necessary signals to create a screen of 1024 by 768 pixels.

The pixel clock frequency is dependant upon the size in pixels wide and high of the desired screen output. In our case, we choose a screen resolution of 1024 pixels wide by 768 high, requiring a 65Mhz pixel clock.

The machine was also equipped (along side the BRAMs available on the Xilinx chip) with 4MB of ZBT SRAM which was available for storage if desired.

2.3. Module Descriptions

2.3.1 Computational Subsection

2.3.1.1 Line Drawer

The computational subsection is built around the function of the line drawer unit. This unit was created to keep it as simple as possible. The most basic information possible to create a line for a human is the 2 end co-ordinates and to use a ruler to connect them. For a digital system, this is simply not possible. In this module, the line drawing is performed by knowing: one co-ordinate, which is always the leftmost; a gradient function; the length of the line.

Consider a co-ordinate space 20 pixels wide by 20 pixels high as represented by the grid below in Figure 3. If we were to draw a line of angle 45 degrees, clearly, we need to count, both in x and y, at and equal rate (shown in Figure 4). Now, if we consider drawing a line with angle of only 22.5 degrees we need to increment the y co-ordinate at half of the rate of the x co-ordinate (shown in Figure 5). This leads to the creation of the mathematical system that was used in this subsection. The value of the y co-ordinate is extended in its representation to be 20 bits wide, rather than 10. 10 of these bits are effectively after a 'decimal' point. (i.e. XXXXXXXXXX·XXXXXXXXXX). Now, the gradient is represented as an 11 bit number. Now, when drawing the line, we increment x by 1 for every pixel and we add the slope variable to the lowest 11 bits of y. We still take the y pixel to be contained in the top 10 bits.

Considering again the example of the 45 degree line (Figure 4). Here, the gradient is clearly 1. Hence, we set the 11 bit slope variable to 10000000000 and indeed, the top 10 bits of y do indeed count up 1 per pixel and give the 45 degree line. Similarly for the 22.5 degree line (Figure 5), clearly the gradient is 0.5 and so we set slope to equal 01000000000 and we see that the top 10 bits of y increment only on every 2^{nd} pixel giving the required line. In effect we see an extension to the binary system where each bit may be represented using fractions also (shown in Figure 6).

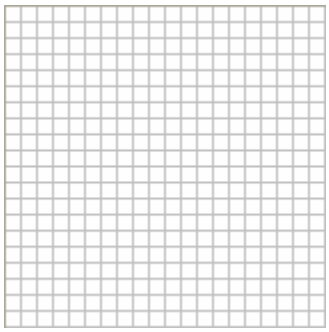


Figure 3 - 20 x 20 Grid

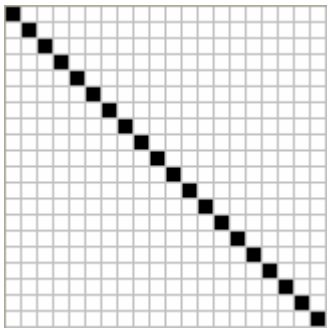


Figure 4 - 45 Degree line. Note how the x and y co-ordinate value of the pixel being drawn increment at the same rate

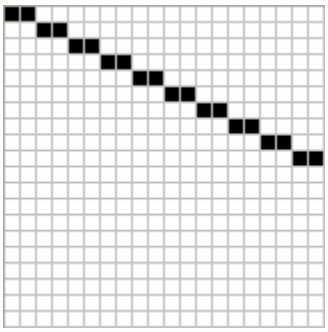


Figure 5 - 22.5 Degree line. Note how the y co-ordinate value of the pixel being drawn increments now at half the rate of the x value

Bit	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Value	512	256	128	64	32	16	8	4	2	1	0.5	0.25	0.125	0.0625	0.03125	0.015625	0.0078125	0.00390625	0.001953125	0.000976563

Figure 6 – Table showing the creation of fractional binary numbers.

Clearly, gradients could also be negative, and since don't need to do any complex arithmetic (making 2's complement unnecessary) it suffices to add a 12^{th} bit to the slope signal to designate a negative gradient.

This system is not perfect, at present we can not draw lines which increment in y faster than in x we have no way generating this. To counteract this problem, an additional control signal `steep` is used to designate that the gradient is more than 45 degrees. Now, for every pixel gradient is added to x and 1 is added to y. So, for a line of 67.5 degrees we would get the pixel output shown in Figure 7.

Now the only line we can not draw is a vertical line. We have no way of representing a gradient large enough to create a vertical line, hence, instead we add another control signal `vertical` which causes the module to increment y by 1 each pixel and leave x constant.

We finish drawing the line when it is of adequate length. This length is always calculated from the variable that is being incremented by 1, and hence may also be considered to represent the number of pixels.

To provide Major FSM/Minor FSM synchronisation between the Beta and the LineDrawer 2 control lines, `done` and `start` are provided. `done` signals high whenever the LineDrawer is not outputting any pixels, its is low at all other times. `start` is used to control when the LineDrawer starts drawing, this means that the data can be changed on the other lines without starting the LineDrawer until all inputs are valid. The LineDrawer also ignores the inputs once it has started drawing so they may be changed before the LineDrawer is done drawing. The

functionality of the LineDrawer module in simulations is shown in Figure 8. The final feature of the LineDrawer is to output the pixel (1023,1023) whilst not drawing so as to not inadvertently place a pixel on the screen.

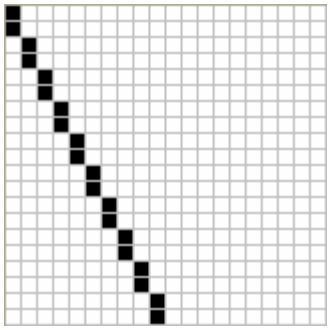


Figure 7 - 67.5 degree line.
By switching which variable (x or y) is being incremented by 1 and which is being incremented by the gradient value we are able to draw almost any line.

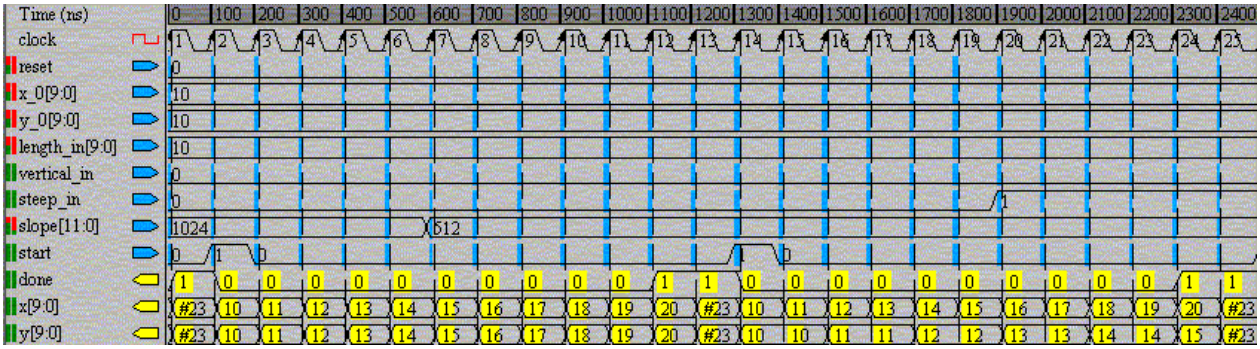


Figure 8 - Simulation Results for the LineDrawer module.
Notice how the data inputs can be changed whilst the module is drawing and how the module also only starts drawing when the start signal is taken high. The values are being displayed in decimal.

The Verilog source code for LineDrawer may be found in Appendix - D:.

2.3.1.2 Beta

A Beta is a type of microprocessor developed in the 6.004 class. It interprets instructions stored in code. A diagram showing the structure of the Beta is shown in Figure 9. Its functionality is described very well on the 6.004 course website http://web.mit.edu/6.004

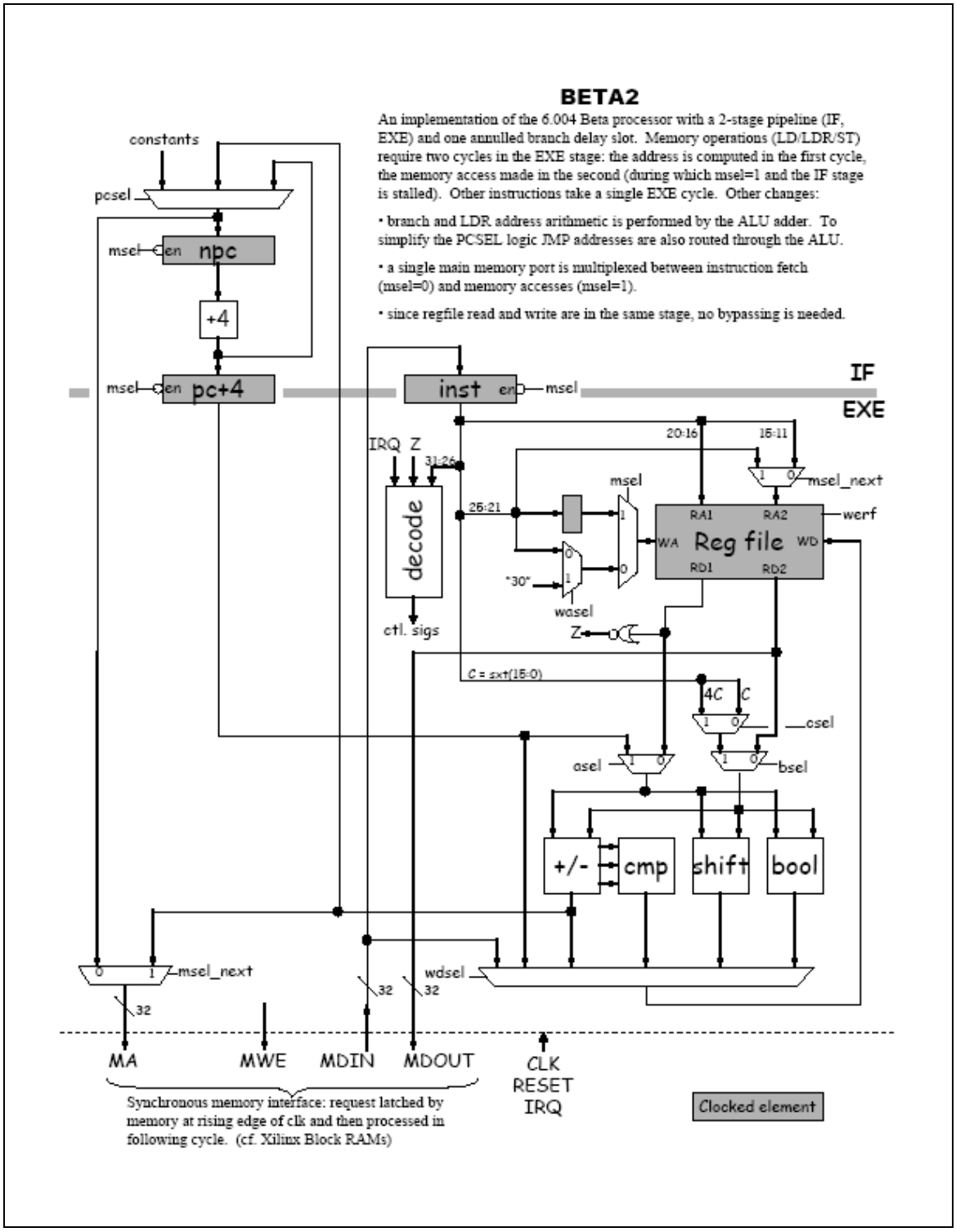


Figure 9 - Beta 2 Diagram [1]

The Beta module itself was taken from the 6.111 Course Website and was modified to include a hardware multiply function. The code for the Beta may be found in Appendix - B:

2.3.1.3 Beta Output Port

Beta output port is a set of registers for each of the 32 bits on each output port. They effectively, in combination with the 'Select Output Port' module create memory mapped ports for the beta such that it can write to hardware items. When the select line for the port is raised high, the data on the memory output of the beta is clocked through to the outputs of the Output Port and held there until the port is chosen again and new data written out. The Verilog Code may be found in Appendix - E:

2.3.1.4 Select Output Port

This module is actually simply some combinational logic substantiated between the Beta and Beta Output/Input Ports. Each Sel_Port line is taken high when the memory address of the output of the Beta is equal to the desired address of the port. It should be noted that the memory address of the port is unique and does not also exist in the BRAM memories. A sample code for Port 1 is detailed below and the code for the rest of the logic may be found in the main labkit substantiation file in Appendix - J:

2.3.1.5 Select Input Port

This was essentially substantiated as a MUX and is in the main labkit code. Appendix - J:

2.3.1.6 BRAM

The BRAM is used by the Beta as both program memory and data memory. It was created using the IP Coregren function of the Xilinx tools. It was created as a memory with 16834 locations each 32 bits wide, giving it storage for 16834 instructions or variables.

2.3.1.7 Debounce

Asynchronous inputs provide difficulties for synchronous machines because they can cause metastable states, the use of several registers in sequence on an asynchronous input reduces the probability of a metastable state being seen by the synchronous machine. This debouncer module is used for all push button inputs (including a reset function which is not shown on the block

diagrams). The module used also provides debouncing to cope with mechanical bounce of the switches. The code may be found in Appendix - A:

2.3.1.8 Clocksplitter

This module simply halves a given input clock frequency at its output. Appendix - C:

2.3.2 Beta Programming

2.3.2.1 Library of Functions

The library of functions was written using a modular approach with the idea being that the functions written would be able to be used in later programs. The modular approach was taken such that each function should not need to know if something in another module has changed, for example, the data could be changed around on ports and only the function responsible for outputting the data would need to know about it and its calling functions would not. The library functions written were (see Appendix - L:Library Functions for the code):

- WritePortX(x, y, , steep, vertical). Takes in variables to be outputted onto port X, and then writes them to the appropriate memory address having placed them onto the correct bits.
- ReadPort4(). When called ReadPort4() causes the input port to be selected and then stores the data at the input of the beta into a register before returning the value to the calling function.
- Swap(x, y). By using pointers Swap (x, y) swaps the values of x and y around.
- CalcSlope(x1, y1, x2, y2). This function implements a 20 bit divide to calculate the slope required for a given line. This program was written entirely in assembly language because of the difficulties found with creating constants and moving them around in C. A dummy C function was placed in the C programs so that the function would be created in the compiled C code and then the real assembler code was substituted in. The 20 bit divide algorithm created is an implementation of long division theory. It is implemented by the shifting of a bit, creating of masks and subtraction. The version implemented requires 20 iterations because of the limitations on the inputs (all only having values greater than 0) and only requiring the last 11 bits (magnitude values less than or equal to 1).

- Drawline(x1, y1, length, slope, steep, vertical). Takes the line data and outputs it to the correct port. It was envisaged that this module would check whether or not the line drawer was ready for the next line before starting the line drawer module. However, it was found that the line drawer always was ready due to the computational time required to calculate the slope and hence, to save on time this check was removed.

- CreateLine (x1, y1, x2, y2). This function is solely concerned with creating the correct outputs for the linedrawer to use. It considers whether a line has a gradient of more or less than 45 degrees (calling CalcSlope appropriately), whether the line is horizontal or vertical (and hence not wasting time calling CalcSlope). It then calls DrawLine to output the created line onto the line drawing hardware.

- Sin(theta) and Cosine(theta). These functions were implements in beta assembler as simple look up tables for values of theta 0 degrees to 90 degrees. It is not difficult to envisage how these could be modified to cope with all values however, to save on the computational time and memory for doing this, (as it was deemed unnecessary for the asteroids game). These functions were provided with a view to allowing rotations of the asteroids.

- Asteroid structures and functions. An asteroid structure was created defining all the data required to be able to draw and update and asteroid. Functions DrawAst(ast) and CreateAst(ast) were written to hide from the game function the actual detail behind the asteroid structure. The CreateAsteroid function is in a basic stage at the moment and always creates an asteroid centred at (100,100) that is a 20 x 20 square. It is not difficult to envisage how this could be modified to create more interesting asteroids and a random start location were a random number generator (library function that was never implemented)

Plans were made to produce additional library functions including these listed below and a set of library functions for the ship and for bullets similar to those for the asteroid, due to more urgent aspects of the project requiring attention these were never implemented:

- Rand(). Generates a random number from a random number look up table

- Rotate(ast). Rotates the asteroid points by the asteroids rotation speed.

- Move(ast). Moves the centre co-ordinates of the asteroid by its speed variables

- Destroy(ast). Sets the asteroids exists variable to be false.

2.3.2.2 Programs

Due to more urgent aspects of the project needing attention, only two programs were written to take advantage of this library of functions.

a) A program to simply draw a constant line onto the screen

b) A program to create an asteroid and then draw it constantly on to the screen.

Though an asteroids game program was never attempted, its not difficult to envisage how using the library of functions that was created that the game was not far from existence.

2.3.3 Graphical Subsection

2.3.3.1 Functionality

The Memory Access Unit (MAU) serves as an interface between the line drawer module and the XVGA module, and does so by making use of memory components on the FPGA. Conceptually, the MAU takes a set of x and y coordinates which corresponds to a new pixel that needs to be drawn on the screen, stores it in the memory, and then loads it from the memory to output this pixel to the display units.

2.3.3.2 Clock Design

From the very start, I realized that clocks, and write pulses for the RAM's are the most sensitive logic signals in designing this module. The design methodology employed by us allows most signals in the FPGA to have hazards, ringing, and even be slow, without actually making the circuit that we designed not work. The penalty for sloppiness in most signals is simply that the circuit will operate slower, rather than not at all. However, this is not the case with clock signals. If a clock has ringing on it, or a slow rise time, then our circuit may not work at all. It pays then to take special care in distributing the clock. Loading rules should be strictly obeyed.

Therefore, the first challenge that I faced was to come up with a reliable clocking strategy that ensures all of the modules are synchronized and that there's going to be enough time for computations during each clock cycle.

It is clear that synchronous signal generation requires precise timing. Labkit comes with a 27 MHz clock. The VGA interface used phase-locked-loops (PLL) and created a higher frequency clock using Xilinx FPGA's "Digital Clock Manager" (DCM), and the code is as follows:

```
DCM pixel_clock (.CLKIN(clock_27mhz), .CLKFX(pixel_clock);
// synthesis attribute CLKFX_DIVIDE of pixel_clock is 10
// synthesis attribute CLKFX_MULTIPLY of pixel_clock is 24
```

This produces a clock at 27 MHz * 24 / 10 = 64.8 MHz which was used as the frequency of the canon shooting across the screen with the cathode ray tube in the Labkit VGA Interface. Hence, it is clear that in order for the interface between the MAU and the XVGA module to work, MAU has to output pixels at a rate of 65MHz (actually 64.8MHz). Due to unavoidable delays in the memory modules, given a read request for a pixel, data is not immediately available. (And is available in the next clock cycle or the cycle after depending on the type of memory.) Therefore, my first approach was to speed up the clock by a factor of two, hence generating a 130MHz (actually 129.6MHz) from the Labkit's DCM, clocking the memory components with this clock, and using a divider (clocksplitter) module to generate the 65MHz clock from it which is then served as the clock for other modules. It took me a while to realize that this would not work. The reasoning is that clocking at a higher speed results in shorter computation time per clock cycle, and for data to be stabilized. This is especially bad when dealing with memory components: random data, address or write-enable signals all result in unpredictable behaviour and could mess up the contents of memories completely. Although the specification of the memories states that the RAM's themselves can be clocked up to 167MHz, it is quite likely that even with simple computations, setup and hold time requirements of these memories will not be met. The more mature approach would be to clock the MAU and hence the memories at the same clock frequency 65MHz, but wait for a few cycles to get the pixel when it's available. Basically this is pipelining – we introduce more latency in the circuits but increase throughput. In order for the other end to work, we had to clock the line drawer module and the Beta at half this frequency, at 32MHz. We thought this was not a problem because the Beta itself needs a long time to calculate the slope and positions of the lines and having the data stable for one more clock cycle is a good thing anyway. This approach let to another problem, being that now the maximum rate we can issue read requests (for the pixels) is at 65MHz, and that is the frequency required by the XVGA module. In other words, if we only use one memory component, we will have to read from it on EVERY clock cycle and will never get a chance to write to it. Clearly this is not going to work. As a solution, we've decided to use a double buffered memory. This means that at every instance

in time, we read from one memory and write to another part of the memory (or another physical memory), after completing a frame, we switch over. The advantage is that now we can clearly update the contents of the memory while keep the reading frequency up at 65MHz (from the other one), but the drawback is that now we need twice the size, but exactly how much? If we make the game with monographic colour, as we said we would, there's one bit per pixel.

2.3.3.3 Memory Usage

The screen resolution is 1024 * 768 pixels. Therefore, total memory needed is:

```
2 * 1024 * 768 bits = 1572864 bits = 192Kbytes
```

This is more than 50% of the Block RAM (BRAM) available from the FPGA but less than 100%. The Beta will also need some BRAM to store instruction sets and at the design stage we couldn't be sure how much memory that is going to be. The other option is to use Zero-buffer Turnaround (ZBT) SRAM (which is much bigger) for the MAU, but I thought it is easier to implement it with the BRAM's first, and then if I have time I can upgrade the module to use ZBT's.

The next thing to consider is the compromise between the size of each memory block and the depth of the memory. The best, easiest and the most natural way to do this would be to have one bit per location, and have 786432 (= 1024 * 768) locations per memory. But this will not work if we consider how we can clear the memory contents between frames. We don't want to skip frames when we clear the memory because that will make the whole idea of double buffering pointless, so somehow we have to squeeze some clock cycles for the clearing mechanism after reading at full speed. Fortunately, the video sync signals have a 'flyback' period, during which we will not be receiving any valid pixels from the line drawer and will not be able to draw any pixels on the screen. Looking at the Verilog code for Hcount and Vcount, which is the horizontal and vertical position of the current pixel respectively:

```
// assume 65 MHz pixel clock
// horizontal: 1344 pixels total
// display 1024 pixels per line
assignhblankon= (hcount== 1023); // turn on blanking
assignhsyncon= (hcount== 1047); // turn on sync pulse
assignhsyncoff= (hcount== 1183); // turn off sync pulse
assignhreset= (hcount== 1343); // end of line (reset counter)

// vertical: 806 lines total
// display 768 lines
assignvblankon= hreset& (vcount== 767); // turn on blanking
assignvsyncon= hreset& (vcount== 776); // turn on sync pulse
assignvsyncoff= hreset& (vcount== 782); // turn off sync pulse
assignvreset= hreset& (vcount== 805); // end of frame
```

We get to the 'blanking' region as soon as vcount hits 767, and that's when we can start clearing frames. We can keep clearing until the end of frame, which is when vcount = 805. Each vcount corresponds to 1343 clock cycles. Hence, the number of clock cycles we have for clearing is:

1343 * (805 – 767) = 51,034 < 786,432

Due to lack of a reset pin on the memories, we have to clear the memory locations one by one, which means we can only clear one memory address per clock cycle. Therefore, if we use one bit per memory block, we would not have time to clear the entire memory before we have to switch to the next frame, and the next time we read from it we will be drawing pixels on top of the leftover contents from the previous frame. So how many bits do we have to include in each memory location in order for us to have enough clock cycles to clear the entire memory during 'flyback' period?

786432 / 51034 = 15.4 bits

In other words, we have to make the memory blocks at least 16 bits wide. To be safe, I have decided to make the blocks 32 bits wide, resulting in 24576 (= 1024 * 768 / 32) memory addresses per memory (and there are two of them).

The screen layout in terms of 'stacking' the memories, therefore, is as follows:

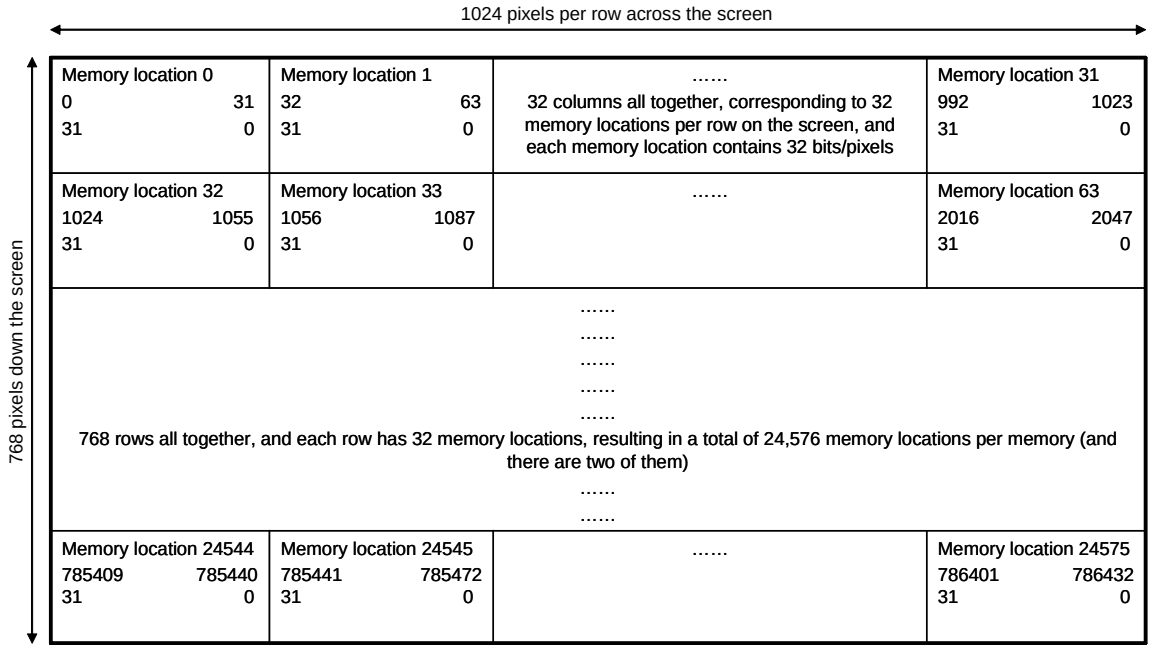


Figure 10 - layout of the screen in terms of memories and pixels
(In each memory location, first row is the pixel number, second row is the bit number for that memory location from MSB to LSB)

With this layout, x and y component of a pixel are both 10 bits. Address can be calculated by simply extracting the y component of a pixel and the top 5 bits of the x component, and concatenate them together in this order. A particular bit inside a memory location is then the bottom 5 bits of a pixel's x component, subtracted from 31.

At this stage, I thought that I was at a good position to start writing the Verilog code for this MAU module, so I've summarized its functionality, that the MAU module should be able to:

- Take in the value of the pixel from the inputs and update the corresponding memory value of the next frame to put a pixel there
- Serve the XVGA modules requests for information about a pixel on the current screen by reading the appropriate values from memory
- Read one frame from memory whilst writing the next frame to another part of memory
- Clear the frames stored in a memory after reading from it
- Switch which section of memory reading from/writing to in each frame

After completing each frame, and after clearing the previous frame, we should switch over the memories so that the memory that was updated in the previous frame can then be read and the memory that was read and erased in the previous frame can then be updated. RAMswitch is a one-bit register that was used to serve this purpose. In my implementations, it either toggles at a falling edge of vsync, or at a particular vcount and hcount value, and that I used was (hcount = 10 and vcount = 768).

It is important that the signals used to gate the write enable, and that the address input and clock enable signals arrive early enough during the clock enable cycle not to cause glitches in the RAM functioning. To help us meet the Threshold requirement of some parts, I've decided to use combinational logic for assignment of addresses, bits to be accessed and write-enable signals. I thought this was perfectly OK in terms of timing and that it could save us a clock cycle compared with using registers and hence give us more time for data to become stable.

For each memory, during the reading frame, a read request is issued by the XVGA module by supplying a set of hcount and vcount values, the specific address and bit within the address which corresponds to this pixel is found by logic in the MAU module and then we can retrieve the value of this pixel (either a 0 or a 1) from the contents of this bit from memory. Due to unavoidable latency in the memory components, this value is not immediately available on the clock cycle

when we issued the read request. Therefore, we have to employ a delay module which basically moves data along the pipeline and retrieve the pixel value a number of clock cycles later when it's become available.

During a writing frame, a write request is issued by the line drawer module by supplying a set of x and y values, and the operation is then subdivided into three stages. In the first stage, we need to find the memory address that corresponds to this set of x and y coordinates and load in the contents of the memory into a register. We then find the particular bit within this memory that we need to update and update it by using a mask. For example, if we want to update bit 31 of the memory block, we can use the following mask:

```
Updated_Memory_Data = Old_Memory_Data | 32'h8000_0000
```

In which the symbol '|' denotes an OR operation. In other words, no matter what the original contents of bit 31 in this memory location was, after masking it with the appropriate value, it becomes a 1. The third and last stage is a writing stage, where we actually write this updated value of memory back into the same memory location that corresponds to the x and y values given by the line drawer a number of clock cycles ago.

It is obvious that when dealing with memory components, timing is critical. We have to supply the correct address, data and write-enable signals at the right clock cycle in order for memory to do what we want it to. A timing diagram each for a reading frame and a writing frame is drawn very carefully by looking at the timing diagram of the memory components used from the lecture slides and is strictly followed to guide my coding of this module.

After some testing and minor debugging (described in more detail in §3.3), this module almost worked as expected within a week's time from I started coding. It puts the correct pixels on the screen for all kinds of lines. But there are two problems. The first one is that the line flickers, and the second one is that it has some glitches, resulting in random pixels flying around the screen, and near the line itself. After checking that clearing and the RAM switching mechanisms are working, I concluded that this was due to a timing error. After discussion with my lab partner we've decided to change the BRAM used in this module with ZBT SRAM's, and the reasoning is that having around two weeks until the deadline, I am in good time to complete this module by solving this timing problem and it would be nice to have a bigger memory because we might want to improve the feature of the game, say, by having more colours. The professors have already written a module that acts as an interface between the ZBT SRAM's and the FPGA, hence using the ZBT memories shouldn't be too big a problem. The other thing is that ZBT has

one more cycle of latency compared with BRAM hence the timing is different anyway. So even if I've fixed the BRAM version of MAU module I still have to change quite a bit to get it working on SRAM's. I thought that my experience with getting this module almost working with BRAM's made me in a good position to expand it and get it working with ZBT's, but was quite wrong.

3. Testing and Debugging

3.1. Process and Methodology

After the programming of a module, its functionality was tested using the "Generate Expected Results" tool. A variety of input combinations were used for each function such that every, and where necessary a redesign was made or parameters were adjusted until output performance was acceptable; leading to a lengthy process of several iterations as described in Figure 11.

Once all modules had been programmed and the interconnections between the modules programmed, the whole system was compiled and downloaded on to the FPGA for further testing on the systems operation. This testing was made simpler by outputting various items to the Hex display (e.g: current state, timer current count value, parameter values, etc). By monitoring the conditions that the system was in, bugs were made easier to find.

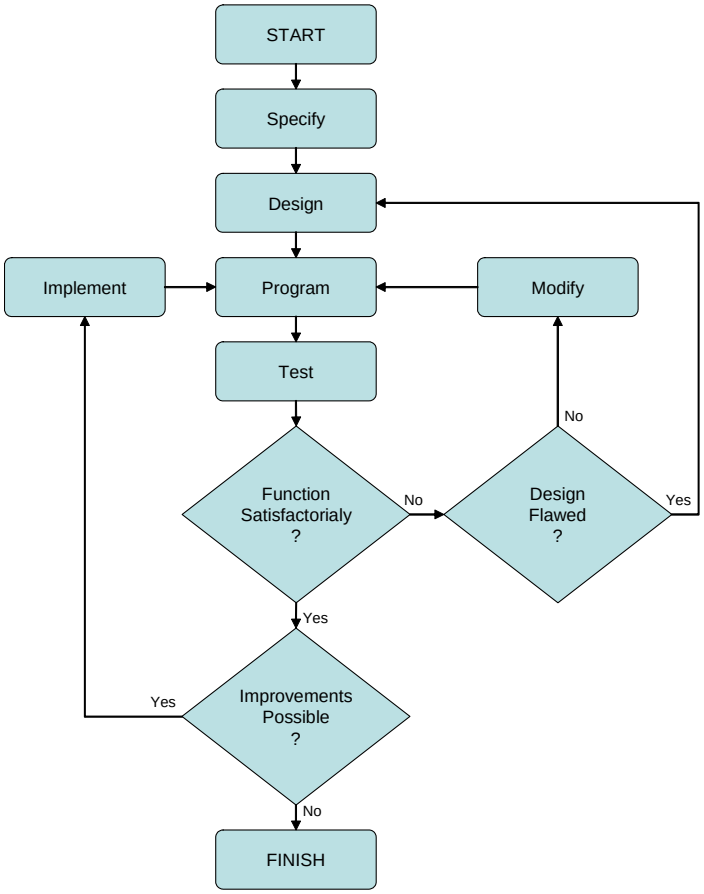


Figure 11 - Design, Testing and Debugging Process

3.2. Computational Subsection

3.2.1 Line Drawer Module

The LineDrawer module was substantiated with the X and Y outputs connected to the 16 Bit Hex Output Displays through a standard Verilog control module for the displays [2]. The inputs of the module were connected to a dummy beta module which was set to give constant values for a known line. The clock input was connected to a 1Hz clock created by a simple counter running of the 27_mhz clock. The outputs were observed for several lines and were found to be correct and match what would have been expected given the mathematical system described in §2.3.1.1. The lines were selected such that the would test all possible modes of line generation, i.e. lines with negative gradient, gradients larger and smaller in magnitude then 45 degrees and vertical lines.

3.2.2 Output Ports

The Output Ports were initially tested only on the Xilinx Simulator, this was done as they are such a simple module that it was believed that they wouldn't be the cause of a problem later on. As discussed in §3.2.3 this was found to be a valid assumption.

3.2.3 Beta

The Beta was initially substantiated exactly as found on [1]. It was not modified in any way. It came with several additional features that were unneeded in this project, notably PS2 mouse support, keyboard support and direct Beta to Hex VGA output support. However, it also came with a test program which used these features. The support for these features was provided by memory mapping (in a different way to how our memory mapped ports were created). The test program [3] was run several times and found to cause no problems.

The memory mapped ports for the additional features were then removed and a single memory mapped Output Port module was added. It was connected to the hex display module [2] so that its outputs could be monitored and a test program to write a constant value to it was written in beta assembly language and written to a memory file (.coe) and loaded into the system memory. This passed without hitch and worked successfully.

The Select Input Port module was then substantiated and a program written to test reading in data from the input port and then writing it back out to the output port. The input ports were connected to the labkit switches for this test. This test caused no problems.

An additional 2 Output Port modules were then substantiated and the Beta was modified to include the hardware multiply function. The functionality of the additional output ports was tested in the same way as the first one. The hardware multiply was tested using a program which created 2 constant variables in registers and then multiply them together and outputting the answer onto a output port connected to the Hex Displays. These tests were found to cause no problems.

3.2.4 Connection of the Beta to the LineDrawer

The LineDrawer Module and the Beta were then connected through the Output Port modules as shown in the block diagram (Figure 2). Sample code was written to output constant values to the output port to draw a line. The Beta and the LineDrawer Modules were both clock on the 1Hz clock that was used previously in §3.2.1. Several outputs were connected to the Hex displays, notably, the memory address output of the Beta, x and y. The correct line was observed to be being drawn.

3.2.5 Programming the Beta

3.2.5.1 Library Functions

The library functions created in §2.3.2.1 were testing sequentially being with the WritePort functions. A main program was created which outputted constant values to each, using the logic analyser to monitor the port outputs it was seen that the WritePort functions performed successfully One frustration was in the WritePort3() module which was designed to clear the port immediately after writing it. The C code reads

```
void WritePort3(long start)
{
      long *outputport;
      long const = 0;
      outputport = (long*) 0xffffffe0;
      *outputport = start;
      *outputport = const;
      return;
}
```

Unfortunately when complied, the ‘optimised’ assembler code neglected to output the data and instead only outputted the clearing data. This was modified at the assembler level to provide the correct functionality. This modification is highlighted in Appendix - M:- Modifications to compiled Assembler Code.

The Sin and Cosine functions were tested only on Bsim by writing a main program to call them. By observing a step wise progression through the code the were observed to function correctly.

Swap(x,y) was also tested in this fashion Bsim.

CalcSlope was tested by taking the assembled code and only taking the operational part of it and replacing the taking the calling variables out of the memory and replacing them with assigning constants to the correct registers to simulate a function call. This was then tested by progressing step wise through the code on Bsim for several possible inputs. The code created by the complier was found not to work perfectly due to the way it creates constants greater than 16 bits wide. The complier creates them in a memory address and loads them in, this was found not to work and so this was replaced at the assembler level by creating the first 16 bits, shifting them left and then adding the 2^{nd} 16 bits in to the lower order bits. This was found to function correctly. The modified assembler functional code was then substituted into the original function so that now it would deal with taking the calling variables. This modification is highlighted in Appendix - M:- Modifications to compiled Assembler Code.

Due to the simplicity of DrawLine() it was not explicitly tested, but in subsequent testing it was found that it was obvious that it was functioning correctly.

Due to time constraints the Asteroids structure and CreateAst and DrawAst functions were never tested.

3.2.5.2 Programs

The line drawing program (Appendix - N:Line Drawing Program) was tested by programming it in to the Beta and monitoring the outputs of the x and y of the line drawing module on the logic analsyer. It was found that for all lines tested that the outputs were as expected.

Due to time constraints the asteroid drawing program was never tested.

3.3. Graphical Subsection

3.3.1 Initial BRAM Combinational MAU

Initially, BRAMs are used for making the MAU module. Having followed a very carefully drawn timing diagram, this module almost worked as expected within a week's time from I started coding. It puts the correct pixels on the screen for all kinds of lines. But there are two problems. The first one is that the line flickers, and the second one is that it has some glitches, resulting in

random pixels flying around the screen, and near the line itself. After checking that clearing and the RAM switching mechanisms are working, I concluded that this was due to a timing error.

Testing, on the very top level, is done in a bottom-up manner. Each individual submodule is tested with a test bench before they are integrated in the asteroids.v module. Many problems were detected during this phase of testing because the test bench waveform that's available from the Xilinx software gave me an idea of what exactly is going on at each clock's rising edge. After simulation with test bench waveforms, both the simple form and the post place and route version of it, I started testing my module on the screen by actually programming the FPGA. At this stage, LED's and hex displays helped me acknowledging information and I could see quite clearly what was going on.

When testing on the screen, I tried putting on different patterns on the screen. For instance, using this line of code:

```
pixel <= (vcount > 300) ? 1 : 0;
```

gives me half black at the top of the screen and half white at the bottom.

```
pixel <= (hcount > 300) ? 1: 0;
```

gives me half black at the right of the screen and half white at the bottom.

```
If (vblank| (hblank& ~hreset)) pixel<= 0;
else pixel <= (hcount==0 | hcount==639 |vcount==0 | vcount==479) ? 1 : 0;
```

gives me a frame at the boundary of the visible screen and the code

```
if(vblank| (hblank& ~hreset)) pixel <= 0;
else pixel <= hcount[8:6];
```

gives me alternating vertical bars.

With these expected patterns I was able to see on the screen what I actually get and could mostly pin down where the problem was that caused this mismatch of expected and actual patterns.

One interesting thing during testing was that when I set the pixel directly to these things the display matches my expectation, which means that reading is pretty much working, but when I set the value of the data written to the memory to be these values then I get wrong patterns. For example, when I tried to put the second test on the screen, instead of getting half black half white screens, I got alternating horizontal bars of a width of about 20 pixels. After a while, I realized that the problem was that I shouldn't have set datain to be a function of hcount or vcount because they are not synchronized with x and y values. So I get different hcount and vcount values each time I do a write (although I don't care so much about it during a writing frame to this memory).

Instead, I should have set datain to be a function of some variable that is synchronized with the writing operations. I tried using the addresses of the writing and it worked as I expected.

I also ensured that RAM switching was working by writing in different patterns in different frames, and saw a switching between patterns on the screen.

I tested clearing with the logic analyzer. After running it, I find the clock cycles when the module is clearing one part of the memory (quite easy to find because write-enable is constantly 1) and then go to the very beginning of it. I looked at this memory address at the beginning and scrolls along the waveform until I reach the end. I could see that the memory location was incrementing by 1 every clock cycle and data that's being written to the memory is constantly 0. In my first try the memories didn't actually cycle all the way through. I realized that the problem was that I started clearing at a rising edge of vsync and stopped at a falling edge. This was only a short period, and was not the complete blanking period. Instead I should start clearing as soon as I could by checking vcount (and at this stage vsync is still 1) and stopping also by checking vcount. After modifying the code this way, I checked on the logic analyzer again and the addresses actually cycled all the way through almost twice, which agreed with my calculations.

After all these testing and some minor debugging, this module almost worked as expected. But there were two problems. The first one is that the line flickers, and the second one is that it has some glitches, resulting in random pixels flying around the screen, and near the line itself. After checking that clearing and the RAM switching mechanisms are working, I concluded that this was due to a timing error.

3.3.2 ZBT MAU

After discussion with my lab partner we've decided to change the BRAM used in this module with ZBT SRAM's, and the reasoning is that having around two weeks until the deadline, I am in good time to complete this module by solving this timing problem and it would be nice to have a bigger memory because we might want to improve the feature of the game, say, by having more colours. The professors have already written a module that acts as an interface between the ZBT SRAM's and the FPGA, hence using the ZBT memories shouldn't be too big a problem. The other thing is that ZBT has one more cycle of latency compared with BRAM hence the timing is different anyway. So even if I've fixed the BRAM version of MAU module I still have to change quite a bit to get it working on SRAM's. I thought that my experience with getting this module

almost working with BRAM's made me in a good position to expand it and get it working with ZBT's, but was quite wrong.

I drew completely new timing diagrams for the ZBT version of my MAU module and started expanding my module to accommodate for the timing changes. Due to the complexity of this module, the code got quite messy but the time I thought I finished. After testing on the test bench waveforms, I programmed the FPGA and tried my module, but got a random screen of pixels. A while later I realized that this was actually leftover contents of memory from the static noise and it was most certainly because I wasn't clearing the memory correctly. I looked at my clearing mechanism again and discovered that I was clearing a memory just before I read from it! This was quite a silly error I made conceptually because vsync is active-low and on a particular RAM switching cycle, it is 1(reading) before it goes to 0(clearing). I corrected the order of clearing and checked it on the logic analyzer to make sure that it worked.

I also checked that reading worked by skipping the loading in stage in a writing frame and putting in constant data during the write back stage in the writing frame. After I programmed the FPGA I saw the correct pattern on the screen.

Complex timing issues aroused in the writing frames. I spent a long time commenting out bits of codes and employed various tests to make sure that the bits I didn't comment out was working as I expected, by looking at the screen output, the hex displays and the logic analyzer. Yet, after ensuring each individual bits worked in my module, when I put them together, I still couldn't get the expected result. I tried different combinations of modifications, outputting a number of screen patterns, registering all the combination logic, inputs and outputs, delaying variable by different numbers of clock cycles, but resulted in vain.

Three days before our final presentation, after discussion with my lab partner, I thought it was about the point for me to give up on this ZBT version of my MAU module and go back to the original BRAM one, which was still the version it was two weeks ago.

Despite the flickering and glitches, we tried integration of my module with James' Beta and line drawing modules.

When having a 'dummy' beta module that outputs constant values to the line drawer, corresponding to drawing a constant line from frame to frame, the three modules seemed to have worked together without a problem, the line was flicking and there were glitches, but all the pixels that should be on the line was there.

However, when we used the actual Beta module, the output on the screen was unacceptable. There were glitches everywhere and we thought that if we'd continue from here, drawing more lines, move/rotate them from frame to frame we'd just get garbage. We figured that the main difference between the dummy beta and the actually beta modules is that the actual one has longer combinational paths, and after integration, the combinational paths might have got too long to settle before we have to use data from it. This was a problem because of the nature that in my MAU module addresses and write-enable signals were assigned combinationally. We tried pipelining all the inputs and outputs of combinational logic from module to module with registers and delay modules but still got a lot of glitches.

3.3.3 BRAM Sequential MAU

Sadly, with just under three days left to go, we've decided to rewrite my MAU module together with entirely sequential logic. After drawing another timing diagram (Appendix - R:and Appendix - S:), both of us have worked flat out to ensure that our last attempt work, because without it we will not be getting any output on the screen. Lots of considerations have been put into timing. Everything happens on a clock edge. To make sure that every input is stable when we read, we've clocked the MAU on the falling clock edge of 65MHz system clock and the memories at the rising clock edge, giving it an extra half a clock cycle to settle. The module was finished the night before our final presentation and we tested it in the morning.

The result was pretty satisfactory: The line was not flickering, and there were only few glitches. We observed the output on the logic analyzer and spotted one timing error. After correcting it, we managed to draw a perfect 45 degrees line on the screen for the first time. Then, a problem occurred: when we tried to draw a 22.5 degrees line we could only get half the pixels on the screen. The line skips every other pixel. After thorough inspection of the waveform on the logic analyzer, we realized what the problem was: with a 45 degrees line that was only one pixel that's been written to each address, but with a 22.5 degrees one two consecutive pixels are written to the same address. Due to the nature of the way I wrote my writing frame, when we were trying to read the address containing the second pixel in before it was updated, it still kept the old value because latency in the memory meant that the previous writing cycle has not yet finished. We figured that we need a bypass path and thought we knew what it should have been by looking at the memory timing diagram. We tried it but the result got worse and we had glitches. At this point we were running out of time and decided to give up on this module to save us some time to prepare for the final presentation.

As a result, I had three versions of the MAU module, that all worked to some extent, but none worked perfectly after integration. I've listed the successful and unsuccessful attributes of the modules in the following table:

Logic used	**Combinational**	**Combinational**	**Sequential**
RAM used	BRAM	ZBT SRAM	BRAM
Success	Perfect pixels for all lines	Correct pixels for dots, no flickering	Perfect pixels for some lines, no flicker
Failure	Flicker and some glitching	Lots of glitching	Is not able to draw certain lines

Table 1 - Comparison between 3 MAU modules

4. Improvements

It was a shame that in the end my MAU module wasn't working, and this really let the project down because although James was doing some exciting things with the Beta, we weren't able to put anything complex on the screen to demonstrate them.

At hindsight I thought that I made at least two major mistakes during my implementation of this module, and these are big lessons to be learnt.

Big lesson number one, I should have got my priority right. After getting the BRAM version of my module almost working, but not working completely, despite having a lot of time at hands, I shouldn't have started thinking about possible improvements and expanded my module (that wasn't working) completely from there. The transition from BRAM to SRAM was a typical case of this, where I changed timing completely and as a result got a pretty messy module with bits of codes commented out and deleted and made hard to debug. I wasted two weeks and got almost nowhere with the ZBT's, and unfortunately had to give up on it completely because we were running out of time. A small Asteroids game is better than no Asteroids game. I should have made sure my BRAM module was working completely before moving onto trying anything else. In this way, even in the worst case, we would have been able to show a small monographic version of the game running on the screen.

Big lesson number two, and perhaps more importantly, it was bad practice to write a big module in any case. During the design stage, I didn't consider carefully enough all the functionality that my module needed to have and during implementation I ended up adding bits and pieces to the module, and this was disastrous: the module became bigger and bigger, I couldn't keep track of the bits that were working and that weren't, and couldn't be sure that the bits I just added was working correctly with the rest of the module (although it was fine by itself). Moreover, within a module, all the lines of codes have side effects. In debugging, I tried commenting out bits of code and making sure that the rest work by employing various tests. (e.g. Putting patterns on the screen) Yet, after ensuring each individual bit worked on its own in my module, when I put them together, I still couldn't get the expected result. In all, I lacked a systematic testing strategy, and it was due to the nature of the way I wrote this module. I tried different combinations of modifications, outputting a number of screen patterns, registering all the combination logic, inputs and outputs, delaying variables by different numbers of clock cycles, but resulted in vain. If, however, I were to use a more modular approach, segregating the big MAU module into a

number of small submodules, perhaps one for reading, one for clearing, one for switching frames, one (or even two or three) for writing… Maybe I could also have separated assignments of addresses and write-enables into separate modules. In this way, I would be able to test each submodules individually, with all the resources that were available, be it testbench waveforms, simulations, LED's, hex displays, logic analyzer or actually programming the FPGA and looking at the screen output. In this way, I could be sure that each submodule was working perfectly before I could do integration of them and moving onto other submodules

5. Conclusions

a) We were unsuccessful in completing the project to the point of creating an asteroids game

b) Modules were successfully completed which create a Beta computer and a library of beta functions was created with a view to making an asteroids game. These functions were demonstrated to work correctly

c) A hardware linedrawer module was made to output in sequence the pixels that exist on a line

d) A start was made to creating a frame buffer module to store in memory and then recreate on the screen a set of pixels given at random to it.

e) We were able to draw successfully on the screen a given line

f) We were unable to successfully integrate the 2 subsections to allow the beta programming to draw lines on the screen

g) Using a memory to create a frame buffer is difficult and requires careful consideration of timing which is difficult to simulate and get accurate data from the logic analyser about.

References

[1] 6.111 Course Website. Beta 2 Demo Package. Beta2.pdf

[2] 6.111 Course Website. 16-Digit HEX display (display_16hex.v). Created April 27, 2004 by Nathan Ickes and updated by Ike Chuang (24-Sep-05 & 02-Nov-05)

[3] 6.004 Course Website. Lab 8 Code.

Appendices – Verilog Code

Appendix - A: Debounce Module

```
//////////////////////////////////////////////////////////////////////////////
//
// Pushbutton Debounce Module (video version)
//
//////////////////////////////////////////////////////////////////////////////

module debounce (reset, clock_65mhz, noisy, clean);
   input reset, clock_65mhz, noisy;
   output clean;

   reg [19:0] count;
   reg new, clean;

   always @(posedge clock_65mhz)
     if (reset) begin new <= noisy; clean <= noisy; count <= 0; end
     else if (noisy != new) begin new <= noisy; count <= 0; end
     else if (count == 650000) clean <= new;
     else count <= count+1;

endmodule
```

Appendix - B: Beta Module

```
//////////////////////////////////////////////////////////////////////////////
//
// Beta2.v: Creates a Beta
//
//////////////////////////////////////////////////////////////////////////////

module beta2(clk,reset,irq,xadr,ma,mdin,mdout,mwe);
  input clk,reset,irq;
  input [30:0] xadr;
  output [31:0] ma,mdout;
  input [31:0] mdin;
  output mwe;

  // beta2 registers
  reg [31:0] regfile[31:0];
  reg [31:0] npc,pc_inc;
  reg [31:0] inst;
  reg [4:0] rc_save;  // needed for second cycle on LD,LDR

  // internal buses
  wire [31:0] rd1,rd2,wd;
  wire [31:0] a,b,xb,c,addsub,cmp,shift,boole,mult;

  // control signals
  wire wasel,werf,z,asel,bsel,csel;
  wire addsub_op,cmp_lt,cmp_eq,shift_op,shift_sxt,boole_and,boole_or;
  wire wd_addsub,wd_cmp,wd_shift,wd_boole,wd_mult;
  wire msel,msel_next,branch,trap,interrupt;
```

```
// pc
wire [31:0] npc_inc,npc_next;
assign npc_inc = npc + 4;
assign npc_next = reset ? 32'h80000000 :
                  msel ? npc :
                  branch ? {npc[31] & addsub[31],addsub[30:2],2'b00} :
                  trap ? 32'h80000004 :
                  interrupt ? {1'b1,xadr} :
                  {npc[31],npc_inc[30:0]};
always @ (posedge clk) begin
  npc <= npc_next;   // logic for msel handled above
  if (!msel) pc_inc <= {npc[31],npc_inc[30:0]};
end

// instruction reg
always @ (posedge clk) if (!msel) inst <= mdin;

// control logic
decode ctl(.clk(clk),.reset(reset),.irq(irq & !npc[31]),.z(z),
           .opcode(inst[31:26]),
           .asel(asel),.bsel(bsel),.csel(csel),.wasel(wasel),
           .werf(werf),.msel(msel),.msel_next(msel_next),.mwe(mwe),
           .addsub_op(addsub_op),.cmp_lt(cmp_lt),.cmp_eq(cmp_eq),
           .shift_op(shift_op),.shift_sxt(shift_sxt),
           .boole_and(boole_and),.boole_or(boole_or),
           .wd_addsub(wd_addsub),.wd_cmp(wd_cmp),
           .wd_shift(wd_shift),.wd_boole(wd_boole),.wd_mult(wd_mult),
           .branch(branch),.trap(trap),.interrupt(interrupt));

// register file
wire [4:0] ra1,ra2,wa;
always @ (posedge clk) if (!msel) rc_save <= inst[25:21];
assign ra1 = inst[20:16];
assign ra2 = msel_next ? inst[25:21] : inst[15:11];
assign wa = msel ? rc_save : wasel ? 5'd30 : inst[25:21];
assign rd1 = (ra1 == 31) ? 0 : regfile[ra1];         // read port 1
assign rd2 = (ra2 == 31) ? 0 : regfile[ra2];         // read port 2
always @ (posedge clk) if (werf) regfile[wa] <= wd;  // write port

assign z = ~| rd1;   // used in BEQ/BNE instructions

// alu
assign a = asel ? pc_inc : rd1;
assign b = bsel ? c : rd2;
assign c = csel ? {{14{inst[15]}},inst[15:0],2'b00} :
                  {{16{inst[15]}},inst[15:0]};

wire addsub_n,addsub_v,addsub_z;
assign xb = {32{addsub_op}} ^ b;
assign addsub = a + xb + addsub_op;
assign addsub_n = addsub[31];
assign addsub_v = (addsub[31] & ~a[31] & ~xb[31]) |
                  (~addsub[31] & a[31] & xb[31]);
assign addsub_z = ~| addsub;

assign cmp[31:1] = 0;
assign cmp[0] = (cmp_lt & (addsub_n ^ addsub_v)) | (cmp_eq & addsub_z);

mul mpy(a,b,mult);
```

```
  wire [31:0] shift_right;
  // Verilog >>> operator not synthesized correctly, so do it by hand
  shift_right sr(shift_sxt,a,b[4:0],shift_right);
  assign shift = shift_op ? shift_right : a << b[4:0];

  assign boole = boole_and ? (a & b) : boole_or ? (a | b) : a ^ b;

  // result mux, listed in order of speed (slowest first)
  assign wd = msel ? mdin :
              wd_cmp ? cmp :
              wd_addsub ? addsub :
              wd_mult ? mult :
              wd_shift ? shift :
              wd_boole ? boole :
              pc_inc;

  // assume synchronous external memory
  assign ma = msel_next ? {npc[31],addsub[30:0]} : npc_next;
  assign mdout = rd2;
endmodule

////////////////////////////////////////////////////////////////////////////
//
// shift_right.v: provides SHR and SRA functions
//
////////////////////////////////////////////////////////////////////////////

module shift_right(sxt,a,b,shift_right);
  input sxt;
  input [31:0] a;
  input [4:0] b;
  output [31:0] shift_right;

  wire [31:0] w,x,y,z;
  wire sin;

  assign sin = sxt & a[31];
  assign w = b[0] ? {sin,a[31:1]} : a;
  assign x = b[1] ? {{2{sin}},w[31:2]} : w;
  assign y = b[2] ? {{4{sin}},x[31:4]} : x;
  assign z = b[3] ? {{8{sin}},y[31:8]} : y;
  assign shift_right = b[4] ? {{16{sin}},z[31:16]} : z;
endmodule

////////////////////////////////////////////////////////////////////////////
//
// mul.v: Provides hardware multiplier for the Beta
//
////////////////////////////////////////////////////////////////////////////

module mul(a,b,c);
    input [31:0] a;
    input [31:0] b;
    output [31:0] c;

    assign c[31:0] = a*b;

endmodule

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

```
//
// decode.v: Converts Beta Op Codes into the required control signals for the
// Beta Components
//
////////////////////////////////////////////////////////////////////////////


module decode(clk,reset,irq,z,opcode,
              asel,bsel,csel,wasel,werf,msel,msel_next,mwe,
              addsub_op,cmp_lt,cmp_eq,
              shift_op,shift_sxt,boole_and,boole_or,
              wd_addsub,wd_cmp,wd_shift,wd_boole,wd_mult,
             branch,trap,interrupt);
  input clk,reset,irq,z;
  input [5:0] opcode;
  output asel,bsel,csel,wasel,werf,msel,msel_next,mwe;
  output addsub_op,shift_op,shift_sxt,cmp_lt,cmp_eq,boole_and,boole_or;
  output wd_addsub,wd_cmp,wd_shift,wd_boole,wd_mult;
  output branch,trap,interrupt;

  reg asel,bsel,csel,wasel,mem_next;
  reg addsub_op,shift_op,shift_sxt,cmp_lt,cmp_eq,boole_and,boole_or;
  reg wd_addsub,wd_cmp,wd_shift,wd_boole,wd_mult;
  reg branch,trap,interrupt;

  // a little bit of state...
  reg annul,msel,mwrite;

  always @ (opcode or z or annul or msel or irq or reset)
  begin
    // initial assignments for all control signals
    asel = 1'hx;
    bsel = 1'hx;
    csel = 1'hx;
    addsub_op = 1'hx;
    shift_op = 1'hx;
    shift_sxt = 1'hx;
    cmp_lt = 1'hx;
    cmp_eq = 1'hx;
    boole_and = 1'hx;
    boole_or = 1'hx;

    wasel = 0;
    mem_next = 0;

    wd_addsub = 0;
    wd_cmp = 0;
    wd_shift = 0;
    wd_boole = 0;
    wd_mult = 0;

    branch = 0;
    trap = 0;
    interrupt = 0;

    if (irq && !reset && !annul && !msel) begin
      interrupt = 1;
      wasel = 1;
    end else casez (opcode)
      6'b011000: begin   // LD
```

```
            asel = 0; bsel = 1; csel = 0;
            addsub_op = 0;
         mem_next = 1;
          end
6'b011001: begin   // ST
            asel = 0; bsel = 1; csel = 0;
            addsub_op = 0;
         mem_next = 1;
          end
6'b011011: begin   // JMP
            asel = 0; bsel = 1; csel = 0;
            addsub_op = 0;
            branch = !annul && !msel;
          end
6'b011101: begin   // BEQ
            asel = 1; bsel = 1; csel = 1;
            addsub_op = 0;
            branch = !annul && !msel && z;
          end
6'b011110: begin   // BNE
            asel = 1; bsel = 1; csel = 1;
            addsub_op = 0;
            branch = !annul && !msel && ~z;
          end
6'b011111: begin   // LDR
            asel = 1; bsel = 1; csel = 1;
            addsub_op = 0;
         mem_next = 1;
          end
6'b1?0000: begin   // ADD, ADDC
            asel = 0; bsel = opcode[4]; csel = 0;
            addsub_op = 0;
            wd_addsub = 1;
          end
6'b1?0001: begin   // SUB, SUBC
            asel = 0; bsel = opcode[4]; csel = 0;
            addsub_op = 1;
            wd_addsub = 1;
          end
6'b1?0010: begin   // MUL, MULC
            asel = 0; bsel = opcode[4]; csel = 0;
            wd_mult = 1;
          end
6'b1?0100: begin   // CMPEQ, CMPEQC
            asel = 0; bsel = opcode[4]; csel = 0;
            addsub_op = 1;
         cmp_eq = 1; cmp_lt = 0;
            wd_cmp = 1;
          end
6'b1?0101: begin   // CMPLT, CMPLTC
            asel = 0; bsel = opcode[4]; csel = 0;
            addsub_op = 1;
         cmp_eq = 0; cmp_lt = 1;
            wd_cmp = 1;
          end
6'b1?0110: begin   // CMPLE, CMPLEC
            asel = 0; bsel = opcode[4]; csel = 0;
            addsub_op = 1;
         cmp_eq = 1; cmp_lt = 1;
            wd_cmp = 1;
```

```
                end
      6'b1?1000: begin   // AND, ANDC
                   asel = 0; bsel = opcode[4]; csel = 0;
                   boole_and = 1; boole_or = 0;
                   wd_boole = 1;
                 end
      6'b1?1001: begin   // OR, ORC
                   asel = 0; bsel = opcode[4]; csel = 0;
                   boole_and = 0; boole_or = 1;
                   wd_boole = 1;
                 end
      6'b1?1010: begin   // XOR, XORC
                   asel = 0; bsel = opcode[4]; csel = 0;
                   boole_and = 0; boole_or = 0;
                   wd_boole = 1;
                 end
      6'b1?1100: begin   // SHL, SHLC
                   asel = 0; bsel = opcode[4]; csel = 0;
                   shift_op = 0;
                   wd_shift = 1;
                 end
      6'b1?1101: begin   // SHR, SHRC
                   asel = 0; bsel = opcode[4]; csel = 0;
                   shift_op = 1; shift_sxt = 0;
                   wd_shift = 1;
                 end
      6'b1?1110: begin   // SRA, SRAC
                   asel = 0; bsel = opcode[4]; csel = 0;
                   shift_op = 1; shift_sxt = 1;
                   wd_shift = 1;
                 end
      default:   begin   // illegal opcode
                   trap = !annul && !msel; wasel = 1;
                 end
    endcase
  end

  // state
  wire msel_next = !reset && !annul && mem_next && !msel;
  wire mwrite_next = msel_next && opcode==6'b011001;

  always @ (posedge clk)
  begin
    annul <= !reset && (trap || branch || interrupt);
    msel <= msel_next;
    mwrite <= mwrite_next;
  end

  assign mwe = mwrite_next;   // assume synchronous memory
  assign werf = msel ? !mwrite : (!annul & !mem_next);
endmodule
```

Appendix - C: Clocksplitter Module

```
//////////////////////////////////////////////////////////////////////
//
// Clocksplitter.v: provides a clock output half the speed of the clock input
//
//////////////////////////////////////////////////////////////////////
```

```
module clocksplitter(clockin, clockouthalf);
	input clockin;
	output clockouthalf;

	reg clockouthalf = 0;

	always @ (posedge clockin)
	begin
		clockouthalf <= clockouthalf + 1;
	end

endmodule
```

Appendix - D: LineDrawer Module

```
//////////////////////////////////////////////////////////////////////////////
//
// linedrawer.v: Creates 1 pixel co-ordinate per clock cycle given:
// an initial co-ordinate which is always the leftmost on the screen, the
// length of the line and details about the gradient
//
//////////////////////////////////////////////////////////////////////////////

module linedrawer (clock, reset, x_0, y_0, length_in, vertical_in, steep_in,
slope, start, done, x, y);
	input clock;				// 32.5MHz clock
	input reset;				// 1 to initialize module
	input [9:0] x_0;			// 10 digit initial x co-ordinate
	input [9:0] y_0;			// 9 digit initial y co-ordinate
	input [9:0] length_in;			// 10 digit length of line (x_end-
x_0 or y_end-y_0 for vertical lines)
	input [11:0] slope;			// gradient of the line
	output [9:0] x;				// x co-ordinate of pixel to be drawn
	output [9:0] y;				// y co-ordinate of pixel to be drawn
	output done;				// signal to Beta that line is drawn
	input start;				// signal from Beta to draw the line
	input vertical_in;			// signal from Beta that the line is
vertical
	input steep_in;				// signal from Beta that the line has a
gradient of more than 45 deg

	reg [9:0] x_start;
	reg [9:0] y_start;
	reg signed [19:0] gradient;
	reg [9:0] length;
	reg drawing = 0, vertical;
	reg [19:0] y_calc;
	reg [19:0] x_calc;
	reg steep;

	always @(posedge clock)

	begin
		if(reset)
		begin
			// just stop drawing on reset... dont need to initialise
registers
			// anyway
			drawing <= 0;
		end
```

```
            else
            begin
            case(drawing)
            0:
                  begin
                        if(start)
                        begin
                              // the beta has told us that a line is ready...
                              // so... lets store in all the values so that we
can
                              // ignore it until we've drawn the line
                              x_start <= x_0;
                              y_start <= y_0;

                              length <= length_in;
                              vertical <= vertical_in;
                              //when storing the slope, sign extended it for
later use
                              gradient[11:0] <= slope;
                              gradient[12] <= slope[11];
                              gradient[13] <= slope[11];
                              gradient[14] <= slope[11];
                              gradient[15] <= slope[11];
                              gradient[16] <= slope[11];
                              gradient[17] <= slope[11];
                              gradient[18] <= slope[11];
                              gradient[19] <= slope[11];


                              steep <= steep_in;
                              // lets remember that we're drawing the line
                              drawing <= 1;
                              // lets output the first pixel of the line
                              x_calc[19:10] <= x_0;
                              x_calc[9:0] <= 10'b0;
                              y_calc[19:10] <= y_0;
                              y_calc[9:0] <= 10'b0;
                        end
                        else
                        begin
                              x_calc[19:10] <= 1023;
                              x_calc[9:0] <= 10'b0;
                              y_calc[19:10] <= 1023;
                              y_calc[9:0] <= 10'b0;
                        end
                  end
            1:
                  begin
                        case(vertical)
                        0:
                              begin
                                    case(steep)
                                    0:
                                          begin
                                                // do some drawing stuff for
none vertical lines
                                                // with gradient of less than
45 degrees
                                                if(x < (x_start + length))
                                                begin
```

```
                                                y_calc <= y_calc +
gradient;
                                                x_calc[19:10] <=
x_calc[19:10] + 1;
                                          end
                                          if((x+1) >= (x_start +
length))
                                          begin
                                                drawing <= 0;
                                          end
                                    end
                              1:
                                    begin
                                          // do some drawing stuff for
none vertical lines
                                          // with gradient of m than 45
degrees
                                          if(gradient[19])
                                          begin
                                                if(y > (y_start -
length))
                                                begin
                                                      x_calc <= x_calc -
gradient;
                                                      y_calc[19:10] <=
y_calc[19:10] - 1;
                                                end
                                                if((y-1) <= (y_start -
length))
                                                begin
                                                      drawing <= 0;
                                                end
                                          end
                                          else
                                          begin
                                                if(y < (y_start +
length))
                                                begin
                                                      x_calc <= x_calc +
gradient;
                                                      y_calc[19:10] <=
y_calc[19:10] + 1;
                                                end
                                                if((y+1) >= (y_start +
length))
                                                begin
                                                      drawing <= 0;
                                                end
                                          end
                                    end
                              endcase
                        end
                  1:
                        begin
                              // draw a vertical line...
                              if(y_calc[19:10] < (y_start + length))
                              begin
                                    y_calc[19:10] <= y_calc[19:10] + 1;
                              end
                              else
```

```
                                    begin
                                          drawing <= 0;
                                    end
                               end
                          endcase
                     end
               endcase
               end
       end
       assign x = x_calc[19:10];
       assign y = y_calc[19:10];
       assign done = ~drawing;
endmodule
```

Appendix - E: Beta Port Module

```
module beta_port(clk,reset,select,data_in,data_out);
    input clk;
    input reset;
    input select;
    input [31:0] data_in;
    output [31:0] data_out;
    reg [31:0] data_out;

    always @ (posedge clk)
    begin
    if(reset)
    begin
            data_out <= 0;
    end
    else if(select)
    begin
            data_out <= data_in;
    end
    end
endmodule
```

Appendix - F: Synchroniser Code

```
//denounces and synchronises switch/button inputs
module debounce (reset, clock, noisy, clean);

   input reset, clock, noisy;
   output clean;

   reg [18:0] count;
   reg new, clean;

   always @(posedge clock)
     if (reset)
       begin
                   count <= 0;
                new <= noisy;
                   clean <= noisy;
       end
     else if (noisy != new)
            begin
                      new <= noisy;
```

```
                    count <= 0;
          end
        else if (count == 270000)
              clean <= new;
            else
              count <= count+1;

endmodule
```

Appendix - G: Combinational MAU

```
//////////////////////////////////////////////////////////////////////////
//
// Memory Access Unit
//
//////////////////////////////////////////////////////////////////////////


module MAU (clock_65mhz, clock_32mhz, reset, x, y, vsync, hcount, vcount,
pixel, addrone, dinone, doutone, weone, addrtwo, dintwo, douttwo, wetwo,
RAMswitch);//, writeaddrone, writeaddrone_delayed, writeaddrtwo,
writeaddrtwo_delayed);  //, bitupdateone, bitupdateone_delayed, bitupdatetwo,
bitupdatetwo_delayed,);

   input clock_65mhz;             // 64.8mhz system clock (xvga clock)
   input clock_32mhz;              // 32.4mhz system clock, used as a counter
in this module
   input reset;                   // 1 to reset to initial state

   input [9:0] x;                  // horizontal index of an existing pixel
(0..1023)
   input [9:0] y;                        // vertical index of an existing
pixel (0..767)

   input vsync;                    // XVGA vertical sync signal (active low)

   input [10:0] hcount;            // horizontal index of current pixel
(0..1023)
   input [9:0] vcount;             // vertical index of current pixel
(0..767)

   output pixel;                   // Output to the xvga module

   output [14:0] addrone;          // 15-bit address in the BRAMone
   output [31:0] dinone;                 // 32-bit data to BRAMone
   input [31:0] doutone;                 // 32-bit data from BRAMone
   output weone;                   // write-enable signal from BRAMone

   output [14:0] addrtwo;          // 15-bit address in the BRAMtwo
   output [31:0] dintwo;                 // 32-bit data to BRAMtwo
   input [31:0] douttwo;                 // 32-bit data from BRAMtwo
   output wetwo;                   // write-enable signal from BRAMtwo

   output RAMswitch;

   output [14:0] writeaddrone;
   output [14:0] writeaddrone_delayed;
   output [14:0] writeaddrtwo;
   output [14:0] writeaddrtwo_delayed;*/
```

```
   // For testing with the testbench waveform
   /*
   output [14:0] readaddrone;
   output [14:0] writeaddrone;
   output [14:0] writeaddrone_delayed;
   output [31:0] doutoneprev;
   output [31:0] doutoneprev_updated;
   output [4:0] bitupdateone;
   output [4:0] bitupdateone_delayed;
   output [4:0] bitcurrentone;
   output [4:0] bitprevone;

   output [14:0] readaddrtwo;
   output [14:0] writeaddrtwo;
   output [14:0] writeaddrtwo_delayed;
   output [31:0] douttwoprev;
   output [31:0] douttwoprev_updated;
   output [4:0] bitupdatetwo;
   output [4:0] bitupdatetwo_delayed;
   output [4:0] bitcurrenttwo;
   output [4:0] bitprevtwo;

   output [14:0] countclear;
   , readaddrone, writeaddrone, writeaddrone_delayed, doutoneprev,
doutoneprev_updated, bitupdateone, bitupdateone_delayed, bitcurrentone,
bitprevone, readaddrtwo, writeaddrtwo, writeaddrtwo_delayed, douttwoprev,
douttwoprev_updated, bitupdatetwo, bitupdatetwo_delayed, bitcurrenttwo,
bitprevtwo, countclear
   */



   // Selects which RAM to read and write to
   reg RAMswitch = 0;

   // For synchronization with reading latency
   wire [10:0] hcountread = hcount + 1;

   // Parameters for BRAMone
   wire [14:0] readaddrone;
   assign readaddrone = {vcount[9:0], hcountread[9:5]};
   wire [14:0] writeaddrone;
   assign writeaddrone = {y, x[9:5]};
   reg [31:0] dinone;
   wire [14:0] writeaddrone_delayed;
   delayN dn1(clock_65mhz, writeaddrone, writeaddrone_delayed);
   //reg [31:0] doutoneprev = 0;
   reg [31:0] doutoneprev_updated;
   reg weone;
   reg [14:0] addrone;


   // Parameters for BRAMone
   wire [14:0] readaddrtwo;

   assign readaddrtwo = {vcount[9:0], hcountread[9:5]};
   wire [14:0] writeaddrtwo;
   assign writeaddrtwo = {y, x[9:5]};
   reg [31:0] dintwo;
```

```
   wire [14:0] writeaddrtwo_delayed;
   delayN dn2(clock_65mhz, writeaddrtwo, writeaddrtwo_delayed);
   //reg [31:0] douttwoprev = 0;
   reg [31:0] douttwoprev_updated;
   reg wetwo;
   reg [14:0] addrtwo;


   // Parameters to find the required bit inside one address, each address is
32 bits wide
   wire [4:0] bitupdateone;
   assign bitupdateone = 31 - x[4:0];
   wire [4:0] bitcurrentone;
   assign bitcurrentone = 31 - hcount[4:0];
   wire [4:0] bitupdatetwo;
   assign bitupdatetwo = 31 - x[4:0];
   wire [4:0] bitcurrenttwo;
   assign bitcurrenttwo = 31 - hcount[4:0];


   wire [4:0] bitupdateone_delayed;
   delayN dn3 (clock_65mhz, bitupdateone, bitupdateone_delayed);
   defparam dn3.Ndelay = 2;
   defparam dn3.width = 5;
   //assign bitupdateone_delayed = bitupdateone;

   wire [4:0] bitupdatetwo_delayed;
   delayN dn4 (clock_65mhz, bitupdatetwo, bitupdatetwo_delayed);
   defparam dn4.Ndelay = 2;
   defparam dn4.width = 5;
   //assign bitupdatetwo_delayed = bitupdatetwo;


   reg pixel = 0;

   // Count for clearing frames
   reg [14:0] countclear = 0;

   // Detect a falling edge of vsync, signalling a new frame
   reg oldvsync = 0;
   wire newframe;
   always @ (posedge clock_65mhz) oldvsync <= reset? 0 : vsync;
   assign newframe = ~vsync & oldvsync;

   wire clearcheck = (vcount >=768) && (vcount <=805);

   always @ (posedge clock_65mhz)
   begin
      if (reset)
      begin
            RAMswitch <= 0;
            countclear <= 0;
            pixel <= 0;
            dinone <= 0;
            dintwo <= 0;
      end
      else
      begin
            if (newframe) RAMswitch <= ~RAMswitch;
            else
```

```
begin
	case (RAMswitch)
	// write to one, read from two
	0:
	begin
		if (clearcheck == 0)
		// valid pixels

		begin

			// WRITE
			case (clock_32mhz)
			// load in
			1:
			begin

			end

			// update (need we high)
			0:
			begin
				//doutoneprev <= doutone;
				dinone <= doutoneprev_updated;
			end

			endcase

			// READ
			pixel <= douttwo[bitcurrenttwo];


		end

		else
		// clearing frames (need we high)
		begin

			dinone <= 0;
		end
	end

	// write to two, read from one
	1:
	begin
		if (clearcheck == 0)
		// valid pixels

		begin

			// WRITE
			case (clock_32mhz)
			// load in
			1:
			begin

			end

			// update (need we high)
			0:
			begin
```

```
                                    //douttwoprev <= douttwo;
                                    dintwo <= douttwoprev_updated;
                              end

                              endcase

                              // READ
                              pixel <= doutone[bitcurrentone];


                        end

                        else
                        // clearing frames (need we high)
                        begin
                              dintwo <= 0;
                        end
                  end

                  endcase
            end
            countclear <= (countclear == 24576) ? 0 : (countclear + 1);
      end
   end

   always @ (reset or newframe or vsync or RAMswitch or bitupdateone_delayed
or bitupdatetwo_delayed or doutone or douttwo)
   begin
      if (reset == 0 && newframe == 0 && vsync == 1)
      begin
            case (RAMswitch)
            0:
            begin
                  case (bitupdateone_delayed)
                  0 :   doutoneprev_updated = doutone |
32'b0000_0000_0000_0000_0000_0000_0000_0001;
                  1 :   doutoneprev_updated = doutone |
32'b0000_0000_0000_0000_0000_0000_0000_0010;
                  2 :  doutoneprev_updated = doutone |
32'b0000_0000_0000_0000_0000_0000_0000_0100;
                  3 :  doutoneprev_updated = doutone |
32'b0000_0000_0000_0000_0000_0000_0000_1000;
                  4 :  doutoneprev_updated = doutone |
32'b0000_0000_0000_0000_0000_0000_0001_0000;
                  5 :  doutoneprev_updated = doutone |
32'b0000_0000_0000_0000_0000_0000_0010_0000;
                  6 :  doutoneprev_updated = doutone |
32'b0000_0000_0000_0000_0000_0000_0100_0000;
                  7 :  doutoneprev_updated = doutone |
32'b0000_0000_0000_0000_0000_0000_1000_0000;
                  8 :  doutoneprev_updated = doutone |
32'b0000_0000_0000_0000_0000_0001_0000_0000;
                  9 :  doutoneprev_updated = doutone |
32'b0000_0000_0000_0000_0000_0010_0000_0000;
                  10:  doutoneprev_updated = doutone |
32'b0000_0000_0000_0000_0000_0100_0000_0000;
                  11:  doutoneprev_updated = doutone |
32'b0000_0000_0000_0000_0000_1000_0000_0000;
                  12:  doutoneprev_updated = doutone |
32'b0000_0000_0000_0000_0001_0000_0000_0000;
```

```
                  13:  doutoneprev_updated = doutone |
32'b0000_0000_0000_0000_0010_0000_0000_0000;
                  14:  doutoneprev_updated = doutone |
32'b0000_0000_0000_0000_0100_0000_0000_0000;
                  15:  doutoneprev_updated = doutone |
32'b0000_0000_0000_0000_1000_0000_0000_0000;
                  16:  doutoneprev_updated = doutone |
32'b0000_0000_0000_0001_0000_0000_0000_0000;
                  17:  doutoneprev_updated = doutone |
32'b0000_0000_0000_0010_0000_0000_0000_0000;
                  18:  doutoneprev_updated = doutone |
32'b0000_0000_0000_0100_0000_0000_0000_0000;
                  19:  doutoneprev_updated = doutone |
32'b0000_0000_0000_1000_0000_0000_0000_0000;
                  20:  doutoneprev_updated = doutone |
32'b0000_0000_0001_0000_0000_0000_0000_0000;
                  21:  doutoneprev_updated = doutone |
32'b0000_0000_0010_0000_0000_0000_0000_0000;
                  22:  doutoneprev_updated = doutone |
32'b0000_0000_0100_0000_0000_0000_0000_0000;
                  23:  doutoneprev_updated = doutone |
32'b0000_0000_1000_0000_0000_0000_0000_0000;
                  24:  doutoneprev_updated = doutone |
32'b0000_0001_0000_0000_0000_0000_0000_0000;
                  25:  doutoneprev_updated = doutone |
32'b0000_0010_0000_0000_0000_0000_0000_0000;
                  26:  doutoneprev_updated = doutone |
32'b0000_0100_0000_0000_0000_0000_0000_0000;
                  27:  doutoneprev_updated = doutone |
32'b0000_1000_0000_0000_0000_0000_0000_0000;
                  28:  doutoneprev_updated = doutone |
32'b0001_0000_0000_0000_0000_0000_0000_0000;
                  29:  doutoneprev_updated = doutone |
32'b0010_0000_0000_0000_0000_0000_0000_0000;
                  30:  doutoneprev_updated = doutone |
32'b0100_0000_0000_0000_0000_0000_0000_0000;
                  31:  doutoneprev_updated = doutone |
32'b1000_0000_0000_0000_0000_0000_0000_0000;
                  endcase
            end


            1:
            begin
                  case (bitupdatetwo_delayed)
                  0 :   douttwoprev_updated = douttwo |
32'b0000_0000_0000_0000_0000_0000_0000_0001;
                  1 :   douttwoprev_updated = douttwo |
32'b0000_0000_0000_0000_0000_0000_0000_0010;
                  2 :  douttwoprev_updated = douttwo |
32'b0000_0000_0000_0000_0000_0000_0000_0100;
                  3 :  douttwoprev_updated = douttwo |
32'b0000_0000_0000_0000_0000_0000_0000_1000;
                  4 :  douttwoprev_updated = douttwo |
32'b0000_0000_0000_0000_0000_0000_0001_0000;
                  5 :  douttwoprev_updated = douttwo |
32'b0000_0000_0000_0000_0000_0000_0010_0000;
                  6 :  douttwoprev_updated = douttwo |
32'b0000_0000_0000_0000_0000_0000_0100_0000;
```

```
                    7 :  douttwoprev_updated = douttwo |
32'b0000_0000_0000_0000_0000_0000_1000_0000;
                    8 :  douttwoprev_updated = douttwo |
32'b0000_0000_0000_0000_0000_0001_0000_0000;
                    9 :  douttwoprev_updated = douttwo |
32'b0000_0000_0000_0000_0000_0010_0000_0000;
                    10:  douttwoprev_updated = douttwo |
32'b0000_0000_0000_0000_0000_0100_0000_0000;
                    11:  douttwoprev_updated = douttwo |
32'b0000_0000_0000_0000_0000_1000_0000_0000;
                    12:  douttwoprev_updated = douttwo |
32'b0000_0000_0000_0000_0001_0000_0000_0000;
                    13:  douttwoprev_updated = douttwo |
32'b0000_0000_0000_0000_0010_0000_0000_0000;
                    14:  douttwoprev_updated = douttwo |
32'b0000_0000_0000_0000_0100_0000_0000_0000;
                    15:  douttwoprev_updated = douttwo |
32'b0000_0000_0000_0000_1000_0000_0000_0000;
                    16:  douttwoprev_updated = douttwo |
32'b0000_0000_0000_0001_0000_0000_0000_0000;
                    17:  douttwoprev_updated = douttwo |
32'b0000_0000_0000_0010_0000_0000_0000_0000;
                    18:  douttwoprev_updated = douttwo |
32'b0000_0000_0000_0100_0000_0000_0000_0000;
                    19:  douttwoprev_updated = douttwo |
32'b0000_0000_0000_1000_0000_0000_0000_0000;
                    20:  douttwoprev_updated = douttwo |
32'b0000_0000_0001_0000_0000_0000_0000_0000;
                    21:  douttwoprev_updated = douttwo |
32'b0000_0000_0010_0000_0000_0000_0000_0000;
                    22:  douttwoprev_updated = douttwo |
32'b0000_0000_0100_0000_0000_0000_0000_0000;
                    23:  douttwoprev_updated = douttwo |
32'b0000_0000_1000_0000_0000_0000_0000_0000;
                    24:  douttwoprev_updated = douttwo |
32'b0000_0001_0000_0000_0000_0000_0000_0000;
                    25:  douttwoprev_updated = douttwo |
32'b0000_0010_0000_0000_0000_0000_0000_0000;
                    26:  douttwoprev_updated = douttwo |
32'b0000_0100_0000_0000_0000_0000_0000_0000;
                    27:  douttwoprev_updated = douttwo |
32'b0000_1000_0000_0000_0000_0000_0000_0000;
                    28:  douttwoprev_updated = douttwo |
32'b0001_0000_0000_0000_0000_0000_0000_0000;
                    29:  douttwoprev_updated = douttwo |
32'b0010_0000_0000_0000_0000_0000_0000_0000;
                    30:  douttwoprev_updated = douttwo |
32'b0100_0000_0000_0000_0000_0000_0000_0000;
                    31:  douttwoprev_updated = douttwo |
32'b1000_0000_0000_0000_0000_0000_0000_0000;
                    endcase
            end
            endcase
      end
   end

  // Logic for addresses and write-enables
  wire mem_update = (reset==0) && (newframe==0);

  // Combinational assignments of addr and we
```

```
  always @ (mem_update or RAMswitch or clearcheck or clock_32mhz or countclear
or readaddrone or writeaddrone or writeaddrone_delayed or readaddrtwo or
writeaddrtwo or writeaddrtwo_delayed)
  begin
        if (mem_update)
        begin
            if (RAMswitch ==0)
            begin //ramsw=0
                  if (clearcheck)
                  begin
                        weone = 1;
                        addrone = countclear;
                        wetwo = 0;
                        addrtwo = 0;
                  end
                  else
                  begin
                        if (clock_32mhz==0)
                        begin
                              weone = 1;
                              addrone = writeaddrone_delayed;
                        end
                        else
                        begin
                              weone = 0;
                              addrone = writeaddrone;
                        end
                        wetwo = 0;
                        addrtwo = readaddrtwo;
                  end
            end
            else
            begin    //ramsw==1
                  if (clearcheck)
                  begin
                        weone = 0;
                        addrone = 0;
                        wetwo = 1;
                        addrtwo = countclear;
                  end
                  else
                  begin
                        if (clock_32mhz==0)
                        begin
                              wetwo = 1;
                              addrtwo = writeaddrtwo_delayed;
                        end
                        else
                        begin
                              wetwo = 0;
                              addrtwo = writeaddrtwo;
                        end
                        weone = 0;
                        addrone = readaddrone;
                  end
            end

        end//end if(mem_update)
  end
```

```
endmodule
```

Appendix - H: Sequential MAU

```
module MAU(clock_65mhz, clock_32mhz, reset, hcount, vcount, x, y, pixel,
              addrone, addrtwo, dinone, dintwo, doutone, douttwo, weone,
              wetwo, RAMswitch);
    input clock_65mhz;
    input clock_32mhz;
    input reset;
    input [10:0] hcount;
    input [9:0] vcount;
    input [9:0] x;
    input [9:0] y;
    output pixel;
    output [14:0] addrone;
    output [14:0] addrtwo;
    output [31:0] dinone;
    output [31:0] dintwo;
    input [31:0] doutone;
    input [31:0] douttwo;
    output weone;
    output wetwo;
    output RAMswitch;

       // Create Registers for outputting data

    reg pixel=0;
    reg [14:0] addrone=0;
    reg [14:0] addrtwo=0;
    reg [31:0] dinone=0;
    reg [31:0] dintwo=0;

    reg weone=0;
    reg wetwo=0;


       // Create Registers for pipelining data
       reg [31:0] bitupdate=0; // store the new pixel location within a memory
location
       reg [31:0] bitupdate_out=0; // store the combination of the new pixel
with the data already created
       reg [4:0] x_lowestbits=0; // stores the location of the pixel to be
written within the selected memory block
       reg [4:0] bitindex_calc=0;// calculates the location of the pixel to be
read within the selected memory block
       reg [4:0] bitindex_used=0;// delayed version of above that is then used
to read the pixel

       reg [31:0] doutoneprev=0, douttwoprev=0;
       // Create Wires and Delays
       reg [14:0] addrone_t1=0, addrone_delay=0, addrtwo_t1=0,
addrtwo_delay=0;

       // TODO: Substantiate Delay Modules - delay address from addrXXX to
addrXXX_delay by 2... i.e N=2. (and then pray)

       // Create Control Registers and Signals
```

```
    reg RAMswitch=0;  // controls which memory module is being written to/read
from
       wire clearing;  // controls when we should be clearing the previously
read buffer so that we can write to it again
       assign clearing = (vcount >=769) && (vcount <=804); // goes high when
we are off the screen and have switched rams...
       reg [14:0] clearcount=0; // used for counting to make sure that we
clear the entire memory module

       always @ (posedge clock_65mhz)
       begin
            clearcount <= (clearcount > 24576) ? 0 : clearcount + 1;
            if(reset)
            begin
                  // Reset stuff...
                  RAMswitch <= 0;
                  pixel <= 0;
               addrone <= 0;
               addrtwo <= 0;
               dinone <= 0;
               dintwo <= 0;
                  weone <= 0;
               wetwo <= 0;
                  bitupdate <= 0;
                  bitupdate_out <= 0;
                  x_lowestbits <= 0;
                  bitindex_calc <= 0;
                  bitindex_used <= 0;
                  addrone_t1 <= 0;
                  addrone_delay <= 0;
                  addrtwo_t1 <= 0;
                  addrtwo_delay <= 0;
            end
            else if(clearing)
            begin
                  case(RAMswitch)
                  0:
                  begin
                        // Writing to RAM ONE / Reading from RAM TWO
                        weone <= 1;
                        wetwo <= 0;
                        addrone <= clearcount;
                        dinone <= 0;
                  end
                  1:
                  begin
                        // Writing to RAM TWO / Reading from RAM ONE
                        weone <= 0;
                        wetwo <= 1;
                        addrtwo <= clearcount;
                        dintwo <= 0;
                  end
                  endcase
            end
            else if((hcount==10) && (vcount == 768))
            begin
                  // Have completed a frame...change over RAMs before starting
to clear
                  RAMswitch <= ~RAMswitch;
            end
```

```
else
begin
      case(RAMswitch)
      0:
            begin
            // Writing to RAM ONE / Reading from RAM TWO

                  // WRITE
                  case(clock_32mhz)
                  0:
                  begin
                        case(x_lowestbits)
                        0: bitupdate  <= 32'h80000000;

                        1: bitupdate  <= 32'h40000000;

                        2: bitupdate  <= 32'h20000000;

                        3: bitupdate  <= 32'h10000000;

                        4: bitupdate  <= 32'h08000000;

                        5: bitupdate  <= 32'h04000000;

                        6: bitupdate  <= 32'h02000000;

                        7: bitupdate  <= 32'h01000000;

                        8: bitupdate  <= 32'h00800000;

                        9: bitupdate  <= 32'h00400000;

                        10: bitupdate <= 32'h00200000;

                        11: bitupdate <= 32'h00100000;

                        12: bitupdate <= 32'h00080000;

                        13: bitupdate <= 32'h00040000;

                        14: bitupdate <= 32'h00020000;

                        15: bitupdate <= 32'h00010000;

                        16: bitupdate <= 32'h00008000;

                        17: bitupdate <= 32'h00004000;

                        18: bitupdate <= 32'h00002000;

                        19: bitupdate <= 32'h00001000;

                        20: bitupdate <= 32'h00000800;

                        21: bitupdate <= 32'h00000400;

                        22: bitupdate <= 32'h00000200;

                        23: bitupdate <= 32'h00000100;
```

```
                              24: bitupdate <= 32'h00000080;

                              25: bitupdate <= 32'h00000040;

                              26: bitupdate <= 32'h00000020;

                              27: bitupdate <= 32'h00000010;

                              28: bitupdate <= 32'h00000008;

                              29: bitupdate <= 32'h00000004;

                              30: bitupdate <= 32'h00000002;

                              31: bitupdate <= 32'h00000001;

                              endcase
                              weone <= (vcount == 805 | vcount == 0) ? 0
: 1;
                              addrone <= addrone_delay;
                        end
                        1:
                        begin
                              x_lowestbits <= x[4:0];
                              addrone <= {y,x[9:5]};
                              weone<= 0;
                              bitupdate_out <= bitupdate | doutoneprev;
                        end
                        endcase
                        dinone <= bitupdate_out;
                        //bitupdate_out <= bitupdate | 32'h0;// doutone

                        // READ
                        addrtwo <= {vcount, hcount[9:5]};
                        bitindex_calc <= hcount[4:0];
                        bitindex_used <= 31-bitindex_calc;
                        pixel <= douttwo[bitindex_used];
                        wetwo <= 0;
                  end
            1:
                  begin
                  // Writing to RAM TWO / Reading from RAM ONE

                        // WRITE
                        case(clock_32mhz)
                        0:
                        begin
                              case(x_lowestbits)
                              0: bitupdate  <= 32'h80000000;

                              1: bitupdate  <= 32'h40000000;

                              2: bitupdate  <= 32'h20000000;

                              3: bitupdate  <= 32'h10000000;

                              4: bitupdate  <= 32'h08000000;

                              5: bitupdate  <= 32'h04000000;
```

```
                        6: bitupdate  <= 32'h02000000;
                        7: bitupdate  <= 32'h01000000;
                        8: bitupdate  <= 32'h00800000;
                        9: bitupdate  <= 32'h00400000;
                        10: bitupdate <= 32'h00200000;
                        11: bitupdate <= 32'h00100000;
                        12: bitupdate <= 32'h00080000;
                        13: bitupdate <= 32'h00040000;
                        14: bitupdate <= 32'h00020000;
                        15: bitupdate <= 32'h00010000;
                        16: bitupdate <= 32'h00008000;
                        17: bitupdate <= 32'h00004000;
                        18: bitupdate <= 32'h00002000;
                        19: bitupdate <= 32'h00001000;
                        20: bitupdate <= 32'h00000800;
                        21: bitupdate <= 32'h00000400;
                        22: bitupdate <= 32'h00000200;
                        23: bitupdate <= 32'h00000100;
                        24: bitupdate <= 32'h00000080;
                        25: bitupdate <= 32'h00000040;
                        26: bitupdate <= 32'h00000020;
                        27: bitupdate <= 32'h00000010;
                        28: bitupdate <= 32'h00000008;
                        29: bitupdate <= 32'h00000004;
                        30: bitupdate <= 32'h00000002;
                        31: bitupdate <= 32'h00000001;
                        endcase
                        wetwo<= (vcount == 805 | vcount ==0) ? 0 :
1;
                        addrtwo <= addrtwo_delay;
                    end
                    1:
                    begin
                        x_lowestbits <= x[4:0];
```

```
                                    addrtwo <= {y,x[9:5]};
                                    wetwo<=0;
                                    bitupdate_out <= bitupdate | douttwoprev;
                                end
                                endcase
                                dintwo <= bitupdate_out;
                                //bitupdate_out <= bitupdate | 32'h0; //douttwo;

                                // READ
                                addrone <= {vcount, hcount[9:5]};
                                bitindex_calc <= hcount[4:0];
                                bitindex_used <= 31-bitindex_calc;
                                pixel <= doutone[bitindex_used];
                                weone <= 0;
                        end
                    endcase
                    // move stuff along in the temporary registers...
                    addrone_t1 <= addrone;
                    addrone_delay <= addrone_t1;

                    addrtwo_t1 <= addrtwo;
                    addrtwo_delay <= addrtwo_t1;

                    doutoneprev <= doutone;
                    douttwoprev <= douttwo;

            end
        end

endmodule
```

Appendix - I: ZBT Combinational MAU

```
//////////////////////////////////////////////////////////////////////////
//
// Memory Access Unit
//
//////////////////////////////////////////////////////////////////////////


module MAU (clock_65mhz, clock_32mhz, reset, x, y, vsync, hcount, vcount,
pixel, addrone, dinone, doutone, weone, addrtwo, dintwo, douttwo, wetwo,
RAMswitch);

   input clock_65mhz;              // 64.8mhz system clock (xvga clock)
   input clock_32mhz;               // 32.4mhz system clock, used as a counter
in this module
   input reset;                    // 1 to reset to initial state

   input [9:0] x;                   // horizontal index of an existing pixel
(0..1023)
   input [9:0] y;                         // vertical index of an existing
pixel (0..767)

   input vsync;                     // XVGA vertical sync signal (active low)

   input [10:0] hcount;             // horizontal index of current pixel
(0..1023)
   input [9:0] vcount;              // vertical index of current pixel
(0..767)
```

```
   output pixel;                    // Output to the xvga module

   output [14:0] addrone;           // 15-bit address in the SRAM0
   output [31:0] dinone;                  // 32-bit data to SRAM0
   input [31:0] doutone;                  // 32-bit data from SRAM0
   output weone;                    // write-enable signal from SRAM0

   output [14:0] addrtwo;           // 15-bit address in the SRAM1
   output [31:0] dintwo;                  // 32-bit data to SRAM1
   input [31:0] douttwo;                  // 32-bit data from SRAM1
   output wetwo;                    // write-enable signal from SRAM1

   output RAMswitch;


   // For testing with the testbench waveform
   /*
   output [14:0] readaddrone;
   output [14:0] writeaddrone;
   output [14:0] writeaddrone_delayed;
   output [31:0] doutoneprev;
   output [31:0] doutoneprev_updated;
   output [4:0] bitupdateone;
   output [4:0] bitupdateone_delayed;
   output [4:0] bitcurrentone;
   output [4:0] bitprevone;

   output [14:0] readaddrtwo;
   output [14:0] writeaddrtwo;
   output [14:0] writeaddrtwo_delayed;
   output [31:0] douttwoprev;
   output [31:0] douttwoprev_updated;
   output [4:0] bitupdatetwo;
   output [4:0] bitupdatetwo_delayed;
   output [4:0] bitcurrenttwo;
   output [4:0] bitprevtwo;

   output [14:0] countclear;
   , readaddrone, writeaddrone, writeaddrone_delayed, doutoneprev,
doutoneprev_updated, bitupdateone, bitupdateone_delayed, bitcurrentone,
bitprevone, readaddrtwo, writeaddrtwo, writeaddrtwo_delayed, douttwoprev,
douttwoprev_updated, bitupdatetwo, bitupdatetwo_delayed, bitcurrenttwo,
bitprevtwo, countclear
   */



   // Selects which RAM to read and write to
   reg RAMswitch = 0;

   // Parameters for the first SRAM module (1st part of memory)
   wire [14:0] readaddrone;
   assign readaddrone = {vcount[9:0], hcount[9:5]}; // for reading
   reg [31:0] doutoneprev = 0;
   wire [14:0] writeaddrone;
   assign writeaddrone = {y, x[9:5]}; // for loading when writing
   reg [31:0] doutoneprev_updated = 0;
   reg [31:0] dinone = 0;
   //wire [14:0] writeaddrone_delayed; // for updating when writing
```

```
   delayN dn4(clock_65mhz, writeaddrone, writeaddrone_delayed);
   defparam dn4.Ndelay = 4;
   defparam dn4.width = 15;

   // Parameters for the second SRAM module (2nd part of memory)
   wire [14:0] readaddrtwo;
   assign readaddrtwo = {vcount[9:0], hcount[9:5]}; // for reading
   reg [31:0] douttwoprev = 0;
   wire [14:0] writeaddrtwo;
   assign writeaddrtwo = {y, x[9:5]}; // for loading when writing
   reg [31:0] douttwoprev_updated = 0;
   reg [31:0] dintwo = 0;
   wire [14:0] writeaddrtwo_delayed; // for updating when writing
   delayN dn5(clock_65mhz, writeaddrtwo, writeaddrtwo_delayed);
   defparam dn5.Ndelay = 4;
   defparam dn5.width = 15;

   // Parameters to find the required bit inside one address, each address is
32 bits wide
   wire [4:0] bitupdateone;
   assign bitupdateone = 31 - x[4:0];
   wire [4:0] bitcurrentone;
   assign bitcurrentone = 31 - hcount[4:0];
   wire [4:0] bitupdatetwo;
   assign bitupdatetwo = 31 - x[4:0];
   wire [4:0] bitcurrenttwo;
   assign bitcurrenttwo = 31 - hcount[4:0];

   wire [4:0] bitprevone;
   delayN dn6(clock_65mhz, bitcurrentone, bitprevone);

   wire [4:0] bitupdateone_delayed;
   delayN dn7(clock_65mhz, bitupdateone, bitupdateone_delayed);

   wire [4:0] bitprevtwo;
   delayN dn8(clock_65mhz, bitcurrenttwo, bitprevtwo);

   wire [4:0] bitupdatetwo_delayed;
   delayN dn9(clock_65mhz, bitupdatetwo, bitupdatetwo_delayed);


   reg pixel = 0;

   // Count for clearing frames
   reg [14:0] countclear = 0;

   // When we are reading, we select reading memory
   // When we are writing, we select writing memory
   assign addrone = weone ? (vsync ? writeaddrone_delayed : countclear) :
((RAMswitch == 1) ? readaddrone : writeaddrone);
   assign addrtwo = wetwo ? (vsync ? writeaddrtwo_delayed : countclear) :
((RAMswitch == 0) ? readaddrtwo : writeaddrtwo);

   // Detect a falling edge of vsync, signalling a new frame
   reg oldvsync = 0;
   always @ (posedge clock_65mhz) oldvsync <= reset? 0 : vsync;
   assign newframe = ~vsync & oldvsync;

   // Logic for write-enables
```

```
   assign weone = ((reset == 0 && newframe == 0 && ((RAMswitch == 0 && vsync
== 1 && clock_32mhz == 0) || (RAMswitch == 1 && vsync == 0))) &&
(writeaddrone_delayed == 1));
   assign wetwo = (reset == 0 && newframe == 0 && ((RAMswitch == 1 && vsync ==
1 && clock_32mhz == 0) || (RAMswitch == 0 && vsync == 0)));

   always @ (posedge clock_65mhz)
   begin
      if (reset)
      begin
            doutoneprev <= 0;
            douttwoprev <= 0;
            RAMswitch <= 0;
            pixel <= 0;
            countclear <= 0;
            dinone <= 0;
            dintwo <= 0;
      end
      else
      begin
            if (newframe) RAMswitch <= ~RAMswitch;
            else
            begin
                  case (RAMswitch)
                  // write to one, read from two
                  0:
                  begin
                        if (vsync)
                        // valid pixels

                        begin

                              // WRITE
                              case (clock_32mhz)
                              // load in
                              1:
                              begin
                                    doutoneprev <= doutone;
                              end

                              // update (need we high)
                              0:
                              begin
                                    dinone <= 32'hFFFF_FFFF;
                                    //dinone <= doutoneprev_updated;
                              end

                              endcase

                              // READ

                              pixel <= douttwo[bitprevtwo];

                        end

                        else
                        // clearing frames (need we high)
                        begin
                              //dintwo <= (addrtwo[2:0] == 0) ? 32'hFFFF_FFFF
: 0;
```

```
                              dintwo <= 0;
                        end
                  end

                  // write to two, read from one
                  1:
                  begin
                        if (vsync)
                        // valid pixels
                        begin

                              // WRITE
                              case (clock_32mhz)

                              // load in
                              1:
                              begin
                                    douttwoprev <= douttwo;
                              end

                              // update (need we high)
                              0:
                              begin
                                    //dintwo <= 32'hFFFF_FFFF;
                                    //dintwo <= douttwoprev_updated;
                              end

                              endcase

                              // READ

                              pixel <= doutone[bitprevone];

                        end

                        else
                        // clearing frames (need we high)
                        begin
                              //dinone <= (addrone[2:0] == 0) ? 32'hFFFF_FFFF
: 0;
                              dinone <= 0;
                        end
                  end
                  endcase
            end
            countclear <= (countclear == 32767) ? 0 : (countclear + 1);
      end
   end

   always @ (reset or newframe or vsync or RAMswitch or bitupdateone_delayed
or bitupdatetwo_delayed or doutoneprev or douttwoprev)
   begin
      if (reset == 0 && newframe == 0 && vsync == 1)
      begin
            case (RAMswitch)
            0:
            begin
                  case (bitupdateone_delayed)
                  0 :   doutoneprev_updated = doutoneprev |
32'b00000000000000000000000000000001;
```

```
                1 :   doutoneprev_updated = doutoneprev |
32'b00000000000000000000000000000010;
                2 :  doutoneprev_updated = doutoneprev |
32'b00000000000000000000000000000100;
                3 :  doutoneprev_updated = doutoneprev |
32'b00000000000000000000000000001000;
                4 :  doutoneprev_updated = doutoneprev |
32'b00000000000000000000000000010000;
                5 :  doutoneprev_updated = doutoneprev |
32'b00000000000000000000000000100000;
                6 :  doutoneprev_updated = doutoneprev |
32'b00000000000000000000000001000000;
                7 :  doutoneprev_updated = doutoneprev |
32'b00000000000000000000000010000000;
                8 :  doutoneprev_updated = doutoneprev |
32'b00000000000000000000000100000000;
                9 :  doutoneprev_updated = doutoneprev |
32'b00000000000000000000001000000000;
                10:  doutoneprev_updated = doutoneprev |
32'b00000000000000000000010000000000;
                11:  doutoneprev_updated = doutoneprev |
32'b00000000000000000000100000000000;
                12:  doutoneprev_updated = doutoneprev |
32'b00000000000000000001000000000000;
                13:  doutoneprev_updated = doutoneprev |
32'b00000000000000000010000000000000;
                14:  doutoneprev_updated = doutoneprev |
32'b00000000000000000100000000000000;
                15:  doutoneprev_updated = doutoneprev |
32'b00000000000000001000000000000000;
                16:  doutoneprev_updated = doutoneprev |
32'b00000000000000010000000000000000;
                17:  doutoneprev_updated = doutoneprev |
32'b00000000000000100000000000000000;
                18:  doutoneprev_updated = doutoneprev |
32'b00000000000001000000000000000000;
                19:  doutoneprev_updated = doutoneprev |
32'b00000000000010000000000000000000;
                20:  doutoneprev_updated = doutoneprev |
32'b00000000000100000000000000000000;
                21:  doutoneprev_updated = doutoneprev |
32'b00000000001000000000000000000000;
                22:  doutoneprev_updated = doutoneprev |
32'b00000000010000000000000000000000;
                23:  doutoneprev_updated = doutoneprev |
32'b00000000100000000000000000000000;
                24:  doutoneprev_updated = doutoneprev |
32'b00000001000000000000000000000000;
                25:  doutoneprev_updated = doutoneprev |
32'b00000010000000000000000000000000;
                26:  doutoneprev_updated = doutoneprev |
32'b00000100000000000000000000000000;
                27:  doutoneprev_updated = doutoneprev |
32'b00001000000000000000000000000000;
                28:  doutoneprev_updated = doutoneprev |
32'b00010000000000000000000000000000;
                29:  doutoneprev_updated = doutoneprev |
32'b00100000000000000000000000000000;
                30:  doutoneprev_updated = doutoneprev |
32'b01000000000000000000000000000000;
```

```
                  31:  doutoneprev_updated = doutoneprev |
32'b10000000000000000000000000000000;
                  endcase
            end


            1:
            begin
                  case (bitupdatetwo_delayed)
                  0 :   douttwoprev_updated = douttwoprev |
32'b00000000000000000000000000000001;
                  1 :   douttwoprev_updated = douttwoprev |
32'b00000000000000000000000000000010;
                  2 :  douttwoprev_updated = douttwoprev |
32'b00000000000000000000000000000100;
                  3 :  douttwoprev_updated = douttwoprev |
32'b00000000000000000000000000001000;
                  4 :  douttwoprev_updated = douttwoprev |
32'b00000000000000000000000000010000;
                  5 :  douttwoprev_updated = douttwoprev |
32'b00000000000000000000000000100000;
                  6 :  douttwoprev_updated = douttwoprev |
32'b00000000000000000000000001000000;
                  7 :  douttwoprev_updated = douttwoprev |
32'b00000000000000000000000010000000;
                  8 :  douttwoprev_updated = douttwoprev |
32'b00000000000000000000000100000000;
                  9 :  douttwoprev_updated = douttwoprev |
32'b00000000000000000000001000000000;
                  10:  douttwoprev_updated = douttwoprev |
32'b00000000000000000000010000000000;
                  11:  douttwoprev_updated = douttwoprev |
32'b00000000000000000000100000000000;
                  12:  douttwoprev_updated = douttwoprev |
32'b00000000000000000001000000000000;
                  13:  douttwoprev_updated = douttwoprev |
32'b00000000000000000010000000000000;
                  14:  douttwoprev_updated = douttwoprev |
32'b00000000000000000100000000000000;
                  15:  douttwoprev_updated = douttwoprev |
32'b00000000000000001000000000000000;
                  16:  douttwoprev_updated = douttwoprev |
32'b00000000000000010000000000000000;
                  17:  douttwoprev_updated = douttwoprev |
32'b00000000000000100000000000000000;
                  18:  douttwoprev_updated = douttwoprev |
32'b00000000000001000000000000000000;
                  19:  douttwoprev_updated = douttwoprev |
32'b00000000000010000000000000000000;
                  20:  douttwoprev_updated = douttwoprev |
32'b00000000000100000000000000000000;
                  21:  douttwoprev_updated = douttwoprev |
32'b00000000001000000000000000000000;
                  22:  douttwoprev_updated = douttwoprev |
32'b00000000010000000000000000000000;
                  23:  douttwoprev_updated = douttwoprev |
32'b00000000100000000000000000000000;
                  24:  douttwoprev_updated = douttwoprev |
32'b00000001000000000000000000000000;
```

```
                  25:  douttwoprev_updated = douttwoprev |
32'b00000010000000000000000000000000;
                  26:  douttwoprev_updated = douttwoprev |
32'b00000100000000000000000000000000;
                  27:  douttwoprev_updated = douttwoprev |
32'b00001000000000000000000000000000;
                  28:  douttwoprev_updated = douttwoprev |
32'b00010000000000000000000000000000;
                  29:  douttwoprev_updated = douttwoprev |
32'b00100000000000000000000000000000;
                  30:  douttwoprev_updated = douttwoprev |
32'b01000000000000000000000000000000;
                  31:  douttwoprev_updated = douttwoprev |
32'b10000000000000000000000000000000;
                  endcase
            end
            endcase
      end
   end

endmodule

module testkit(clock_32mhz,reset,x,y);

   input clock_32mhz;              // 64.8mhz system clock
   input reset;

   output [9:0] x;
   output [9:0] y;

   reg [9:0] x;
   reg [9:0] y;

  // reg count = 0;

   always @ (posedge clock_32mhz)
   begin
      //if (count == 0)
      //begin
         if (reset)
         begin
            x <= 63;
            y <= 0;
         end
         /*
         else
         begin
            x <= (x == 570) ? 500 : (x + 1);
          y <= (y == 100) ? 30 : (y + 1);
        end
         */
      //end
      //count <= ~count;
   end
endmodule


//////////////////////////////////////////////////////////////////////////
//
// parameterized delay line  (off by one delay)
```

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

module delayN(clk,in,out);

   parameter Ndelay = 3;

   parameter width = 5;

   input clk;
   input [width-1:0] in;
   output [width-1:0] out;



   reg [Ndelay*width-1:0] shiftreg;

   wire [width-1:0] out = shiftreg[Ndelay*width-1:Ndelay*width-width];

   always @(posedge clk)
     shiftreg <= {shiftreg[Ndelay*width-1-width:0],in};

endmodule
```

Appendix - J: Labkit Code

```
//////////////////////////////////////////////////////////////////////////////
//
//
// 6.111 FPGA Labkit -- Template Toplevel Module
//
// For Labkit Revision 004
//
//
// Created: October 31, 2004, from revision 003 file
// Author: Nathan Ickes
//
//////////////////////////////////////////////////////////////////////////////
//
//
// CHANGES FOR BOARD REVISION 004
//
// 1) Added signals for logic analyzer pods 2-4.
// 2) Expanded "tv_in_ycrcb" to 20 bits.
// 3) Renamed "tv_out_data" to "tv_out_i2c_data" and "tv_out_sclk" to
//    "tv_out_i2c_clock".
// 4) Reversed disp_data_in and disp_data_out signals, so that "out" is an
//    output of the FPGA, and "in" is an input.
//
// CHANGES FOR BOARD REVISION 003
//
// 1) Combined flash chip enables into a single signal, flash_ce_b.
//
// CHANGES FOR BOARD REVISION 002
//
// 1) Added SRAM clock feedback path input and output
// 2) Renamed "mousedata" to "mouse_data"
// 3) Renamed some ZBT memory signals. Parity bits are now incorporated into
//    the data bus, and the byte write enables have been combined into the
```

```
//    4-bit ram#_bwe_b bus.
// 4) Removed the "systemace_clock" net, since the SystemACE clock is now
//    hardwired on the PCB to the oscillator.
//
//////////////////////////////////////////////////////////////////////////////
/
//
// Complete change history (including bug fixes)
//
// 2005-Sep-09: Added missing default assignments to "ac97_sdata_out",
//              "disp_data_out", "analyzer[2-3]_clock" and
//              "analyzer[2-3]_data".
//
// 2005-Jan-23: Reduced flash address bus to 24 bits, to match 128Mb devices
//              actually populated on the boards. (The boards support up to
//              256Mb devices, with 25 address lines.)
//
// 2004-Oct-31: Adapted to new revision 004 board.
//
// 2004-May-01: Changed "disp_data_in" to be an output, and gave it a default
//              value. (Previous versions of this file declared this port to
//              be an input.)
//
// 2004-Apr-29: Reduced SRAM address busses to 19 bits, to match 18Mb devices
//              actually populated on the boards. (The boards support up to
//              72Mb devices, with 21 address lines.)
//
// 2004-Apr-29: Change history started
//
//////////////////////////////////////////////////////////////////////////////
/

module asteroids   (beep, audio_reset_b, ac97_sdata_out, ac97_sdata_in,
ac97_synch,
             ac97_bit_clock,

             vga_out_red, vga_out_green, vga_out_blue, vga_out_sync_b,
             vga_out_blank_b, vga_out_pixel_clock, vga_out_hsync,
             vga_out_vsync,

             tv_out_ycrcb, tv_out_reset_b, tv_out_clock, tv_out_i2c_clock,
             tv_out_i2c_data, tv_out_pal_ntsc, tv_out_hsync_b,
             tv_out_vsync_b, tv_out_blank_b, tv_out_subcar_reset,

             tv_in_ycrcb, tv_in_data_valid, tv_in_line_clock1,
             tv_in_line_clock2, tv_in_aef, tv_in_hff, tv_in_aff,
             tv_in_i2c_clock, tv_in_i2c_data, tv_in_fifo_read,
             tv_in_fifo_clock, tv_in_iso, tv_in_reset_b, tv_in_clock,

             ram0_data, ram0_address, ram0_adv_ld, ram0_clk, ram0_cen_b,
             ram0_ce_b, ram0_oe_b, ram0_we_b, ram0_bwe_b,

             ram1_data, ram1_address, ram1_adv_ld, ram1_clk, ram1_cen_b,
             ram1_ce_b, ram1_oe_b, ram1_we_b, ram1_bwe_b,

             clock_feedback_out, clock_feedback_in,

             flash_data, flash_address, flash_ce_b, flash_oe_b, flash_we_b,
             flash_reset_b, flash_sts, flash_byte_b,
```

```
            rs232_txd, rs232_rxd, rs232_rts, rs232_cts,

            mouse_clock, mouse_data, keyboard_clock, keyboard_data,

            clock_27mhz, clock1, clock2,

            disp_blank, disp_data_out, disp_clock, disp_rs, disp_ce_b,
            disp_reset_b, disp_data_in,

            button0, button1, button2, button3, button_enter, button_right,
            button_left, button_down, button_up,

            switch,

            led,

            user1, user2, user3, user4,

            daughtercard,

            systemace_data, systemace_address, systemace_ce_b,
            systemace_we_b, systemace_oe_b, systemace_irq, systemace_mpbrdy,

            analyzer1_data, analyzer1_clock,
            analyzer2_data, analyzer2_clock,
            analyzer3_data, analyzer3_clock,
            analyzer4_data, analyzer4_clock);

   output beep, audio_reset_b, ac97_synch, ac97_sdata_out;
   input  ac97_bit_clock, ac97_sdata_in;

   output [7:0] vga_out_red, vga_out_green, vga_out_blue;
   output vga_out_sync_b, vga_out_blank_b, vga_out_pixel_clock,
        vga_out_hsync, vga_out_vsync;

   output [9:0] tv_out_ycrcb;
   output tv_out_reset_b, tv_out_clock, tv_out_i2c_clock, tv_out_i2c_data,
        tv_out_pal_ntsc, tv_out_hsync_b, tv_out_vsync_b, tv_out_blank_b,
        tv_out_subcar_reset;

   input  [19:0] tv_in_ycrcb;
   input  tv_in_data_valid, tv_in_line_clock1, tv_in_line_clock2, tv_in_aef,
        tv_in_hff, tv_in_aff;
   output tv_in_i2c_clock, tv_in_fifo_read, tv_in_fifo_clock, tv_in_iso,
        tv_in_reset_b, tv_in_clock;
   inout  tv_in_i2c_data;

   inout  [35:0] ram0_data;
   output [18:0] ram0_address;
   output ram0_adv_ld, ram0_clk, ram0_cen_b, ram0_ce_b, ram0_oe_b, ram0_we_b;
   output [3:0] ram0_bwe_b;

   inout  [35:0] ram1_data;
   output [18:0] ram1_address;
   output ram1_adv_ld, ram1_clk, ram1_cen_b, ram1_ce_b, ram1_oe_b, ram1_we_b;
   output [3:0] ram1_bwe_b;

   input  clock_feedback_in;
   output clock_feedback_out;
```

```
   inout  [15:0] flash_data;
   output [23:0] flash_address;
   output flash_ce_b, flash_oe_b, flash_we_b, flash_reset_b, flash_byte_b;
   input  flash_sts;

   output rs232_txd, rs232_rts;
   input  rs232_rxd, rs232_cts;

   input  mouse_clock, mouse_data, keyboard_clock, keyboard_data;

   input  clock_27mhz, clock1, clock2;

   output disp_blank, disp_clock, disp_rs, disp_ce_b, disp_reset_b;
   input  disp_data_in;
   output  disp_data_out;

   input  button0, button1, button2, button3, button_enter, button_right,
        button_left, button_down, button_up;
   input  [7:0] switch;
   output [7:0] led;

   inout [31:0] user1, user2, user3, user4;

   inout [43:0] daughtercard;

   inout  [15:0] systemace_data;
   output [6:0]  systemace_address;
   output systemace_ce_b, systemace_we_b, systemace_oe_b;
   input  systemace_irq, systemace_mpbrdy;

   output [15:0] analyzer1_data, analyzer2_data, analyzer3_data,
              analyzer4_data;
   output analyzer1_clock, analyzer2_clock, analyzer3_clock, analyzer4_clock;


////////////////////////////////////////////////////////////////////////
   //
   // I/O Assignments
   //

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

   // Audio Input and Output
   assign beep= 1'b0;
   assign audio_reset_b = 1'b0;
   assign ac97_synch = 1'b0;
   assign ac97_sdata_out = 1'b0;
   // ac97_sdata_in is an input

   // Video Output
   assign tv_out_ycrcb = 10'h0;
   assign tv_out_reset_b = 1'b0;
   assign tv_out_clock = 1'b0;
   assign tv_out_i2c_clock = 1'b0;
   assign tv_out_i2c_data = 1'b0;
   assign tv_out_pal_ntsc = 1'b0;
   assign tv_out_hsync_b = 1'b1;
   assign tv_out_vsync_b = 1'b1;
   assign tv_out_blank_b = 1'b1;
   assign tv_out_subcar_reset = 1'b0;
```

```
// Video Input
assign tv_in_i2c_clock = 1'b0;
assign tv_in_fifo_read = 1'b0;
assign tv_in_fifo_clock = 1'b0;
assign tv_in_iso = 1'b0;
assign tv_in_reset_b = 1'b0;
assign tv_in_clock = 1'b0;
assign tv_in_i2c_data = 1'bZ;
// tv_in_ycrcb, tv_in_data_valid, tv_in_line_clock1, tv_in_line_clock2,
// tv_in_aef, tv_in_hff, and tv_in_aff are inputs

// SRAMs
/*
assign ram0_data = 36'hZ;
assign ram0_address = 19'h0;
assign ram0_we_b = 1'b0;
assign ram0_clk = 1'b0;
assign ram0_cen_b = 1'b1; // clock enable
*/

// Enable RAM0 pins
assign ram0_ce_b = 1'b0;
assign ram0_oe_b = 1'b0;
assign ram0_adv_ld = 1'b0;
assign ram0_bwe_b = 4'h0;

/*
assign ram1_data = 36'hZ;
assign ram1_address = 19'h0;
assign ram1_we_b = 1'b0;
assign ram1_clk = 1'b0;
assign ram1_cen_b = 1'b1;
*/

// Enable RAM1 pins
assign ram1_ce_b = 1'b0;
assign ram1_oe_b = 1'b0;
assign ram1_adv_ld = 1'b0;
assign ram1_bwe_b = 4'h0;
assign clock_feedback_out = 1'b0;
// clock_feedback_in is an input

// Flash ROM
assign flash_data = 16'hZ;
assign flash_address = 24'h0;
assign flash_ce_b = 1'b1;
assign flash_oe_b = 1'b1;
assign flash_we_b = 1'b1;
assign flash_reset_b = 1'b0;
assign flash_byte_b = 1'b1;
// flash_sts is an input

// RS-232 Interface
assign rs232_txd = 1'b1;
assign rs232_rts = 1'b1;
// rs232_rxd and rs232_cts are inputs

// PS/2 Ports
// mouse_clock, mouse_data, keyboard_clock, and keyboard_data are inputs
```

```
/*
// LED Displays
assign disp_blank = 1'b1;
assign disp_clock = 1'b0;
assign disp_rs = 1'b0;
assign disp_ce_b = 1'b1;
assign disp_reset_b = 1'b0;
assign disp_data_out = 1'b0;
// disp_data_in is an input
*/

// Buttons, Switches, and Individual LEDs
//lab3 assign led = 8'hFF;
// button0, button1, button2, button3, button_enter, button_right,
// button_left, button_down, button_up, and switches are inputs

// User I/Os
assign user1 = 32'hZ;
assign user2 = 32'hZ;
assign user3 = 32'hZ;
assign user4 = 32'hZ;

// Daughtercard Connectors
assign daughtercard = 44'hZ;

// SystemACE Microprocessor Port
assign systemace_data = 16'hZ;
assign systemace_address = 7'h0;
assign systemace_ce_b = 1'b1;
assign systemace_we_b = 1'b1;
assign systemace_oe_b = 1'b1;
// systemace_irq and systemace_mpbrdy are inputs

// 32.4 MHz system clock
wire clock_32Mhz;

// use FPGA's digital clock manager to produce a
// 65MHz clock (actually 64.8MHz)
wire clock_65mhz_unbuf,clock_65mhztemp;
DCM vclk3(.CLKIN(clock_27mhz),.CLKFX(clock_65mhz_unbuf));
// synthesis attribute CLKFX_DIVIDE of vclk3 is 10
// synthesis attribute CLKFX_MULTIPLY of vclk3 is 24
// synthesis attribute CLK_FEEDBACK of vclk3 is NONE
// synthesis attribute CLKIN_PERIOD of vclk3 is 37
BUFG vclk4(.O(clock_65mhztemp),.I(clock_65mhz_unbuf));

wire clock_65mhz = clock_65mhztemp;

// power-on reset generation
wire power_on_reset;    // remain high for first 16 clocks
SRL16 reset_sr (.D(1'b0), .CLK(clock_65mhz), .Q(power_on_reset),
            .A0(1'b1), .A1(1'b1), .A2(1'b1), .A3(1'b1));
defparam reset_sr.INIT = 16'hFFFF;

// button0 is user reset
wire reset,user_reset;
debounce db1(power_on_reset, clock_65mhz, ~button0, user_reset);
assign reset = user_reset | power_on_reset;
```

```
   // UP and DOWN buttons for the ship
   wire up,down;
   debounce db2(reset, clock_65mhz, ~button_up, up);
   debounce db3(reset, clock_65mhz, ~button_down, down);

   // LEFT and RIGHT rotate buttons for the ship
   wire left, right;
   debounce db4(reset, clock_65mhz, ~button_left, left);
   debounce db5(reset, clock_65mhz, ~button_right, right);

   // generate basic XVGA video signals
   wire [10:0] hcount;
   wire [9:0]  vcount;
   wire hsync,vsync,blank;
   xvga xvga1(clock_65mhz,hcount,vcount,hsync,vsync,blank);

   // Wire up the SRAMs
   // Wire up the Memory Access Unit (MAU)

   wire RAMpixel;
   wire [14:0] addrone, addrtwo;
   wire [31:0] dinone, dintwo;
   wire [31:0] doutone, douttwo;
   wire weone, wetwo;
   wire RAMswitch;

   wire [9:0] x, y;

   framebuffer fb1(addrone, clock_65mhz, dinone, doutone, weone);
   framebuffer fb2(addrtwo, clock_65mhz, dintwo, douttwo, wetwo);

   MAU mau1(clock_65mhz, clock_32mhz, reset, x, y, vsync, hcount, vcount,
RAMpixel, addrone, dinone, doutone, weone, addrtwo, dintwo, douttwo, wetwo,
RAMswitch);

   reg [7:0] pixel;
   wire b, hs, vs;

   assign hs = hsync;
   assign vs = vsync;
   assign b = blank;

   always @ (posedge clock_65mhz)
   begin
      pixel[7] <= RAMpixel;
      pixel[6] <= RAMpixel;
      pixel[5] <= RAMpixel;
      pixel[4] <= RAMpixel;
      pixel[3] <= RAMpixel;
      pixel[2] <= RAMpixel;
      pixel[1] <= RAMpixel;
      pixel[0] <= RAMpixel;
   end

   // VGA Output.  In order to meet the setup and hold times of the
   // AD7125, we send it ~clock_65mhz.
   assign vga_out_red = pixel;
   assign vga_out_green = pixel;
   assign vga_out_blue = pixel;
   assign vga_out_sync_b = 1'b1;    // not used
```

```
   assign vga_out_blank_b = ~b;
   assign vga_out_pixel_clock = ~clock_65mhz;
   assign vga_out_hsync = hs;
   assign vga_out_vsync = vs;

   assign led = ~{RAMswitch,up,down,left,right,reset,switch[1:0]};

   clocksplitter cs1(clock_65Mhz, clock_32Mhz);

   wire [30:0] irq_addr = 31'b0;
   wire irq = 0;
   wire clock;

   assign clock = clock_32Mhz;

   // make a Beta!
   wire mwe;
   reg [31:0] mdin;
   wire [31:0] ma,mdout,ramout,dpyout,ps2out;
   beta2 cpu(clock,reset,irq,irq_addr,ma,mdin,mdout,mwe);

   // decode memory address
   wire highmem = &ma[31:15];    // includes supervisor bit!
   wire sel_ram = !highmem;
   wire sel_port1 = highmem && &ma[14:4] && ~ma[3] && ~ma[2]; // 0xfffffff0
   wire sel_port2 = highmem && &ma[14:4] && ~ma[3] && ma[2]; // 0xfffffff4
   wire sel_port3 = highmem && &ma[14:5] && ~ma[4] && ~ma[3] && ~ma[2]; //
0xffffffe0
   wire sel_port4 = highmem && &ma[14:5] && ~ma[4] && ~ma[3] && ma[2]; //
0xffffffe4

   // create ports
   wire [31:0] port1; // data on port 1: Registered
   wire [31:0] port2; // data on port 2: Registered
   wire [31:0] port3; // data on port 3: Unregisterd
   wire [31:0] port4; // data on port 4: Read
   beta_port bp1(clock,1'b0,sel_port1,mdout,port1);
   beta_port bp2(clock,1'b0,sel_port2,mdout,port2);
   beta_port bp3(clock,1'b0,sel_port3,mdout,port3);


   // remember what beta is trying to read
   reg mdin_sel;
   always @ (posedge clock) begin
      mdin_sel <= sel_port4 ? 1'd1 :
                  1'd0;
   end

   // select data to send back to beta
   always @ (mdin_sel or ramout or port4)
     case (mdin_sel)
       default: mdin = ramout;
       1'd1: mdin = port4;
     endcase

   // program memory: up to 16K x 32
   // (memory module generated by betamem.py)
   betaram mem(ma[15:2],clock,mdout,ramout,mwe & sel_ram);
```

```
   linedrawer ld1(clock, 1'b0, port1[9:0], port1[19:10], port1[29:20],
port1[30], port1[31], port2[11:0], port3[0], port4[0], x, y);

   /*
   // Logic Analyzer
   assign analyzer1_data = {8'h0,mdout[13:8],clock,sel_port1};
   assign analyzer1_clock = 1'b1;
   assign analyzer2_data = {8'h0,mdout[7:0]};
   assign analyzer2_clock = 1'b1;
   assign analyzer3_data = port1[15:0];
   assign analyzer3_clock = 1'b1;
   assign analyzer4_data = port1[31:16];
   assign analyzer4_clock = clock;
   */
   /*
   // Logic Analyzer
   assign analyzer1_data = {8'h0,8'h0};
   assign analyzer1_clock = 1'b1;
   assign analyzer2_data = {8'h0,8'h0};
   assign analyzer2_clock = 1'b1;
   assign analyzer3_data = {y[5:0],x[9:0]};
   assign analyzer3_clock = 1'b1;
   assign analyzer4_data = {2'h0,port4[0],port3[0],y[9:6]};
   assign analyzer4_clock = clock;
   */

   /*
   // Logic Analyzer
   assign analyzer1_data =
{8'h0,port3[0],port4[0],port2[11:0],clock,sel_port1};
   assign analyzer1_clock = 1'b1;
   assign analyzer2_data = {8'h0,port2[7:0]};
   assign analyzer2_clock = 1'b1;
   assign analyzer3_data = port1[15:0];
   assign analyzer3_clock = 1'b1;
   assign analyzer4_data = port1[31:0];
   assign analyzer4_clock = clock;
   */

   /*
   // Logic Analyzer (db)
   assign analyzer1_data = {1'b0,clk,port4[0],start,y[9:6]};
   assign analyzer1_clock = 1'b1;
   assign analyzer2_data = 16'h0;
   assign analyzer2_clock = 1'b1;
   assign analyzer3_data = {y[5:0],x};
   assign analyzer3_clock = 1'b1;
   assign analyzer4_data = 16'h0;
   assign analyzer4_clock = 1'b1;
   */

      /*
   // Digital logic analyser
   assign analyzer1_clock = 1'b1;
   assign analyzer1_data = 16'h0;

   assign analyzer2_clock = 1'b1;
   assign analyzer2_data = 16'h0;

   assign analyzer3_clock = 1'b1;
```

```
   assign analyzer3_data = 16'h0;

   assign analyzer4_clock = 1'b1;
   assign analyzer4_data = 16'h0;
   */
   /*
   // Digital logic analyser // test setup 1
   assign analyzer1_clock = 1'b1;
   assign analyzer1_data = {douttwo[12:8], vsync, x[9:0]};

   assign analyzer2_clock = 1'b1;
   assign analyzer2_data = {dintwo[12:8], clock_32mhz, y[9:0]};

   assign analyzer3_clock = 1'b1;
   assign analyzer3_data = {dintwo[7:0], douttwo[7:0]};

   assign analyzer4_clock = clock_65mhz;
   assign analyzer4_data = {addrtwo[14:0], wetwo};
   */
   /*
   assign analyzer1_clock = 1'b1;
   assign analyzer1_data = {dintwo[31:16]};

   assign analyzer2_clock = 1'b1;
   assign analyzer2_data = {dintwo[15:0]};

   assign analyzer3_clock = 1'b1;
   assign analyzer3_data = {x[9:0], clock_32mhz, wetwo, vsync, addrtwo[8:6]};

   assign analyzer4_clock = clock_65mhz;
   assign analyzer4_data = {y[9:0], addrtwo[5:0]};
   */
   /*
   assign analyzer1_clock = 1'b1;
   assign analyzer1_data = port1[31:16];

   assign analyzer2_clock = 1'b1;
   assign analyzer2_data = port1[15:0];

   assign analyzer3_clock = 1'b1;
   assign analyzer3_data = {x[9:0], clock_32mhz, wetwo, vsync, port2[11:9]};

   assign analyzer4_clock = clock_65mhz;
   assign analyzer4_data = {y[9:0], port2[8:3]};
   */
endmodule
```

Appendix - K: XVGA Module

```
//////////////////////////////////////////////////////////////////////////////
//
// xvga: Generate XVGA display signals (1024 x 768 @ 60Hz)
//
//////////////////////////////////////////////////////////////////////////////

module xvga(vclock,hcount,vcount,hsync,vsync,blank);
   input vclock;
   output [10:0] hcount;
   output [9:0] vcount;
   output   vsync;
```

```
   output   hsync;
   output   blank;

   reg        hsync,vsync,hblank,vblank,blank;
   reg [10:0]      hcount;    // pixel number on current line
   reg [9:0] vcount;     // line number

   // horizontal: 1344 pixels total
   // display 1024 pixels per line
   wire      hsyncon,hsyncoff,hreset,hblankon;
   assign    hblankon = (hcount == 1023);
   assign    hsyncon = (hcount == 1047);
   assign    hsyncoff = (hcount == 1183);
   assign    hreset = (hcount == 1343);

   // vertical: 806 lines total
   // display 768 lines
   wire      vsyncon,vsyncoff,vreset,vblankon;
   assign    vblankon = hreset & (vcount == 767);
   assign    vsyncon = hreset & (vcount == 776);
   assign    vsyncoff = hreset & (vcount == 782);
   assign    vreset = hreset & (vcount == 805);

   // sync and blanking
   wire      next_hblank,next_vblank;
   assign next_hblank = hreset ? 0 : hblankon ? 1 : hblank;
   assign next_vblank = vreset ? 0 : vblankon ? 1 : vblank;

   always @(posedge vclock) begin
      hcount <= hreset ? 0 : hcount + 1;
      hblank <= next_hblank;
      hsync <= hsyncon ? 0 : hsyncoff ? 1 : hsync;  // active low

      vcount <= hreset ? (vreset ? 0 : vcount + 1) : vcount;
      vblank <= next_vblank;
      vsync <= vsyncon ? 0 : vsyncoff ? 1 : vsync;  // active low

      blank <= next_vblank | (next_hblank & ~hreset);
   end

endmodule
```

Appendices – Beta Programs

Appendix - L: Library Functions

```
void WritePort1(long x, long y, long length, long steep, long vertical)
{
      long dataout;
      long *outputport;
      outputport = (long*) 0xfffffff0;
      steep = steep << 31;
      vertical = vertical << 30;
      length = length << 20;
      y = y << 10;

      dataout = (steep | vertical | length | x| y);
      *outputport = dataout;
      return;
}

void WritePort2(long slope)
{
      long *outputport;
      outputport = (long*) 0xfffffff4;

      *outputport = slope;
      return;
}

void WritePort3(long start)
{
      long *outputport;
      long const = 0;
      outputport = (long*) 0xffffffe0;
      *outputport = start;
      *outputport = const;
      return;
}

long ReadPort4()
{
      long *inputport;
      inputport = (long*) 0xffffffe4;

      return *inputport;
}

void createline(long x1, long y1, long x2, long y2)
{
      long steep, vertical;
      long length;
      long lengthx, lengthy;
      long slope=0;

      // Deal with pixel 1 not being the lefthand pixel

      if(x2 < x1)
      {
```

```
        swap (&x1, &x2);
        swap (&y1, &y2);
      }
      // deal with a vertical line
      if(x1 == x2)

      {
            steep = 1;
            vertical = 1;
            if(y1 > y2)
            {
                  length = y1 - y2;
                  drawline(x1, y2, length, slope, steep, vertical);
                  return;
            }
            else
            {
                  length = y2 - y1;
                  drawline(x1, y1, length, slope, steep, vertical);
                  return;
            }
      }
      // deal with a horizontal line
      if(y1 == y2)
      {
            steep = 0;
            vertical = 1;
            slope = 0;
            drawline(x1, y1, length, slope, steep, vertical);
            return;
      }
      // deal with all other lines

      else
      {
        vertical = 0;
            lengthx = x2 - x1;
            lengthy = y2 - y1;
            if(lengthx >= lengthy)
              {
                  steep = 0;
                  slope = calcslope(lengthy, lengthx);
                  drawline(x1, y1, lengthx, slope, steep, vertical);
                  return;
                  }
                  else
            {

                  steep = 1;
                  slope = calcslope(lengthx, lengthy);
                  drawline(x1, y1, lengthy, slope, steep, vertical);
                  return;
                  }
      }
      return; // should never get here... but just incase...
}

void drawline(long x1, long y1, long length, long slope, long steep, long
vertical)
{
```

```
	long done;
	long mask = 0x000000001;
	long start = 1;

	// Put the data out
	WritePort1(x1, y1, length, steep, vertical);
	WritePort2(slope);
	WritePort3(start);
	return;
}

// Exchanges 2 values
void swap(long *x, long *y)
{
	long temp;

	temp = *x;
	*x = *y;
	*y = temp;
	return;
}

long calcslope(long dividend, long divisor)
{
	// Deal with errors, our game code should never create these
situations... but if it does, dont waste time
	if(divisor == 1)
	  return 0;
	if(dividend > divisor)
	  return 0;

	long mask=0;
	long mask_generator = 0x00100000;
	long masked=0;
	long result=0;
	  int i;
	divisor = divisor << 20;
	dividend = dividend << 10;

	for(i=0; i <= 20; i++)
	{
		divisor = divisor >> 1;
		mask_generator = mask_generator >> 1;
		mask = mask | mask_generator;
		masked = mask & dividend;

		if(divisor <= masked)
		{
			result = result | mask_generator;
			dividend = dividend - divisor;
		}
	}

	return result;
}

struct Asteroid {
	long xcentre;
	long ycentre;
	  long x[8], y[8];
```

```
      long speedx, speedy, rotspeed;
      long rotdirn, exists, size; // actually booleans but doesnt seem to like
boolean...
};

createasteroid(ast)
     struct Asteroid *ast;
{
  long i;
  (ast->xcentre) = 100;
  (ast->ycentre) = 100;
  (ast->x[0]) = -10;
  (ast->y[0]) = -10;
  (ast->x[1]) = 0;
  (ast->y[1]) = -10;
  (ast->x[2]) = 10;
  (ast->y[2]) = -10;
  (ast->x[3]) = 10;
  (ast->y[3]) = 0;
  (ast->x[4]) = 10;
  (ast->y[4]) = 10;
  (ast->x[5]) = 0;
  (ast->y[5]) = 10;
  (ast->x[6]) = -10;
  (ast->y[6]) = 10;
  (ast->x[7]) = -10;
  (ast->y[7]) = 0;
}

drawasteroid(ast)
     struct Asteroid *ast;
{
  long i;
  for(i=0;i<8;i++)
    {
      if(i<7)
      {
        createline(((ast->xcentre)+(ast->x[i])),((ast->xcentre)+(ast-
>y[i])),((ast->xcentre)+(ast->x[i+1])),((ast->xcentre)+(ast->y[i+1])));
      }
      if(i=7)
      {
        createline(((ast->xcentre)+(ast->x[7])),((ast->xcentre)+(ast-
>y[7])),((ast->xcentre)+(ast->x[0])),((ast->xcentre)+(ast->y[0])));
      }
    }
}
```

Appendix - M: Modifications to compiled Assembler Code

```
calcslope:
      PUSH (LP)
      PUSH (BP)
      MOVE (SP, BP)
      PUSH (R1)
      PUSH (R2)
      PUSH (R3)
      PUSH (R4)
      PUSH (R5)
      PUSH (R6)
```

```
        PUSH (R7)
        LD (BP, -12, R5)
        LD (BP, -16, R1)
        CMPEQC (R1, 1, R0)
        MOVE (R31, R2)
        BT (R0, _LCS1)
        CMPLE (R5, R1, R0)
        MOVE (R31, R2)
        BF (R0, _LCS1)
        MOVE (R31, R4)
        CMOVE(0x4000, r3)
        SHLC(r3,6,r3)
        MOVE (R31, R6)
        SHLC (R1, 20, R1)
        SHLC (R5, 10, R5)
        CMOVE (20, R2)
        MOVE (R31, R7)
_LCS9:
        SRAC (R1, 1, R1)
        SRAC (R3, 1, R3)
        OR (R4, R3, R4)
        AND (R4, R5, R0)
        CMPLE (R1, R0, R0)
        BF (R0, _LCS6)
        OR (R6, R3, R6)
        SUB (R5, R1, R5)
_LCS6:
        SUBC (R2, 1, R2)
        CMPLT (R2, R7, R0)
        BF (R0, _LCS9)
        MOVE (R6, R2)
_LCS1:
        MOVE (R2, R0)
        POP (R7)
        POP (R6)
        POP (R5)
        POP (R4)
        POP (R3)
        POP (R2)
        POP (R1)
        POP (BP)
        POP (LP)
        JMP (LP)


WritePort3:
        PUSH (LP)
        PUSH (BP)
        MOVE (SP, BP)
        LD (BP, -12, R0)
        ST (R0, -32)
        ADDC(r31,0,r31)
        ST (R31, -32)
        POP (BP)
        POP (LP)
        JMP (LP)
```

Appendix - N: Line Drawing Program

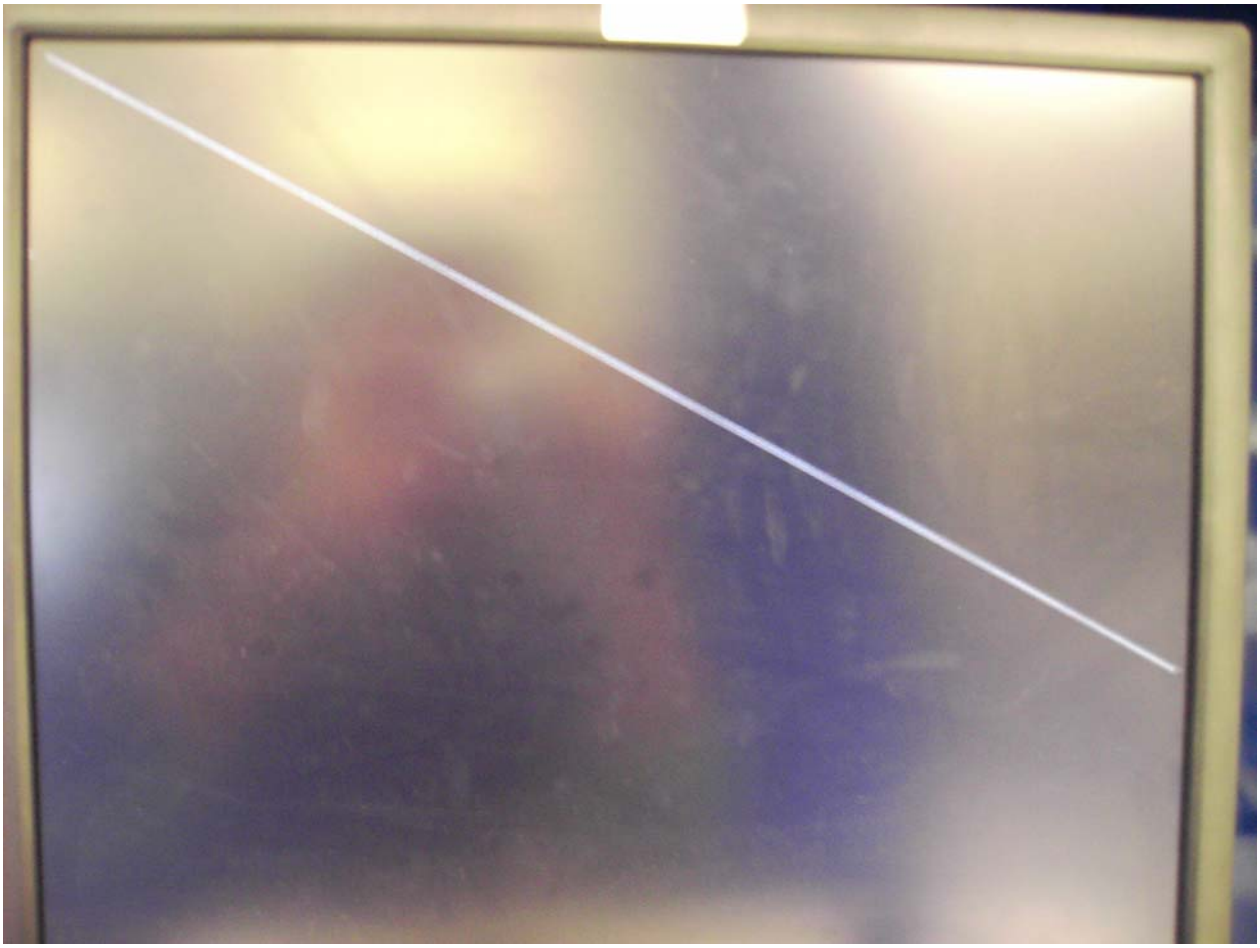
```
int main()
{
  long x1=100, y1=100, x2=200, y2=200;
  while(1)
{
createline(x1,y1,x2,y2);
}
  return 0;
}
```

Appendix - O: Asteroid Drawing Program

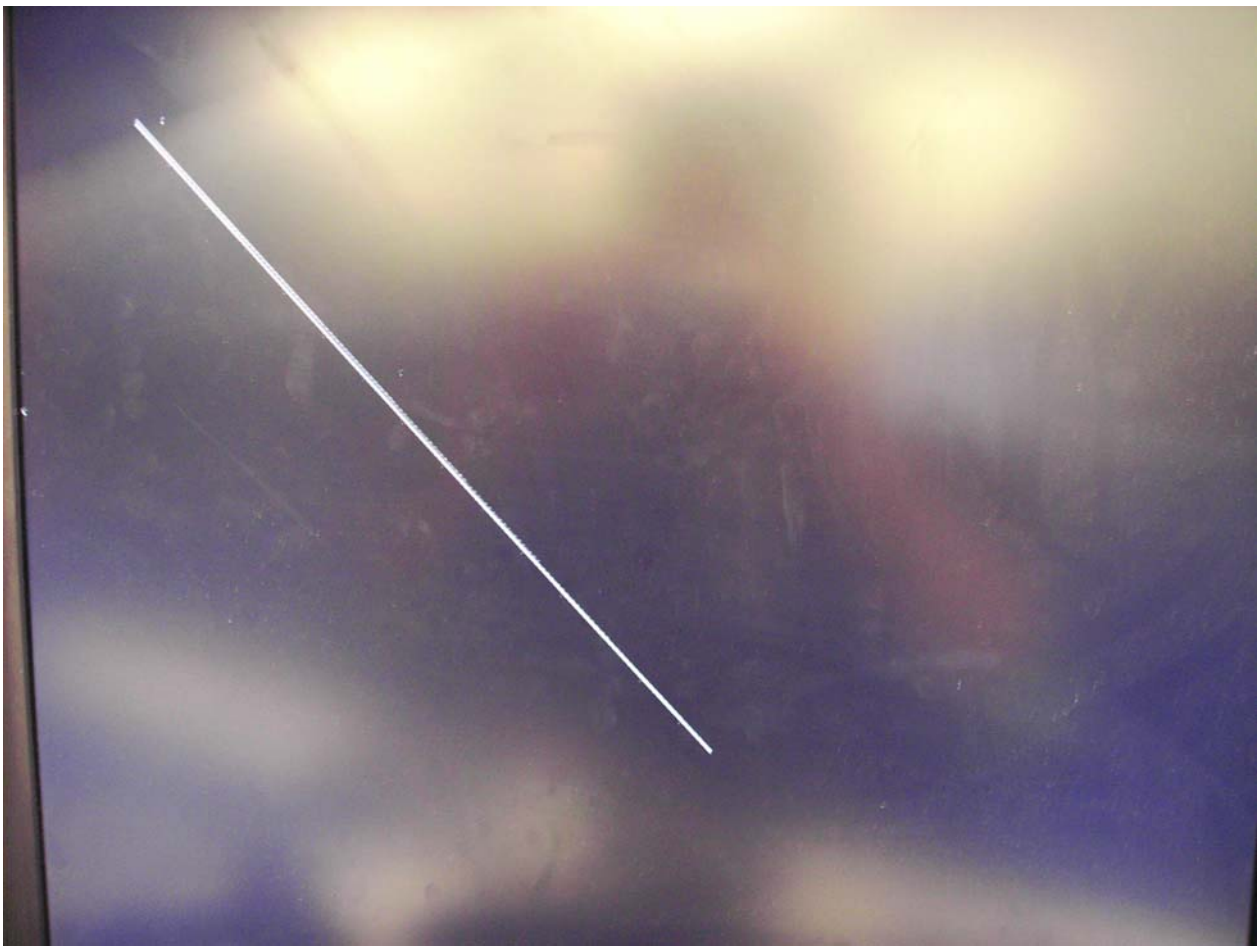
```
int main()
{

  struct Asteroid ast1;
  createasteroid(&ast1);
  while(1)
    {
      drawasteroid(&ast1);
    }
  return 0;
}
```

Appendicies – Screenshots

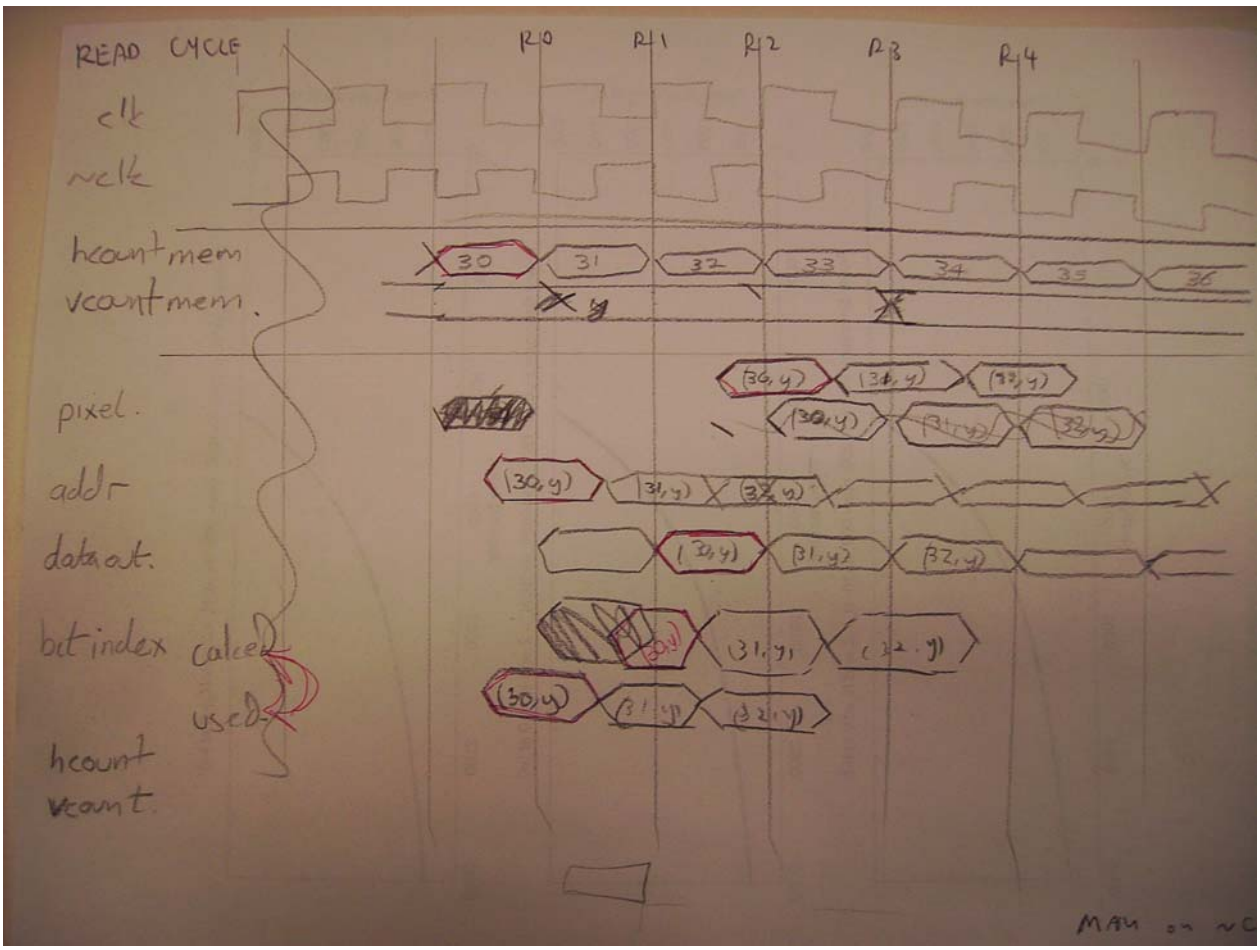
Appendix - P: Screenshot - 22.5 Degrees line from Combinational MAU



Appendix - Q: Screenshot – 45 Degree line from Combinational MAU



Appendix - R: Timing Diagram for Read Cycle for Sequential MAU



Appendix - S: Timing Diagram for Write Cycle for Sequential MAU