3D GPU

Final Report

Upsham Dawra (Tuesday 2.30pm)
Will McGrath (Thursday 2.30pm)
Vishal Gala (Thursday 2.30pm)

ECE 337
TA: Nitin Nitin
Submission date: 10/29/10

1) Executive Summary

Our group built a basic 3D Graphics Processing Unit. Our GPU is able to take 3D points representing lines and render an image. This represents the basic functionality of any GPU and will serve as a starting point for potential future extensions, added features and optimizations.

The GPU we created provides an easily extensible framework to develop a more advanced GPU. Creating a GPU with performance similar to any commercial product designed in the last ten years from scratch is an intractable problem. However, a simple GPU with the same basic components as a more complex GPU provides a good launching point from which to add more functionality and increase its performance.

Dedicated graphics processors got their start in the early 90‟s when computer programs became more graphically demanding. They were created to allow the CPU to offload graphical tasks to dedicated hardware, specifically designed to be able to handle highly parallel, floating point arithmetic-intensive tasks. GPUs are ASIC by nature; they must handle well-defined graphical operations as quickly as possible. A standard microcontroller would be utterly unsuitable for acting as a GPU, because of its single thread of execution.

The following sections discuss the I/O characteristics of our design, our optimizations and an overview of the structure of the design. They will also relate our success criteria and what we did to meet them. Last, we will include the layout that we produced and the layout characteristics that we found.

2) Design Specifications

2.1) Operational Characteristics

a) Table of external inputs and outputs.

Signal Name	Type IN/OUT/BIDIR	Number of bits	Description
PWR	-	-	To power the chip.
GND	-	-	The ground for the chip.
RST_N	IN	1	A asynchronous reset for the memory elements in the design.
CLK	IN	1	The CLK for the chip.
OPCODE	IN	8	The opcodes for performing instructions and the addresses of data on which the instructions should be executed.
RAM_IN_USE	IN	1	Signal to indicate the GPU not to operate on the external RAM as the processor is operating on it.
DATABUS	IN/OUT	16	The data read from the RAM or written to the RAM.
STRB_IN	IN	1	Signal to strobe the opcode into the GPU
ADDR_OUT	OUT	16	The address to load data from the external RAM.
RE_OUT	OUT	1	Signal to indicate the processor that GPU is operating on the RAM.
GPU_DONE	OUT	1	Signal to indicate when the GPU has done it's processing completely.
WE_OUT	OUT	1	Signal to set the RAM to write mode.

Table 2.1.1

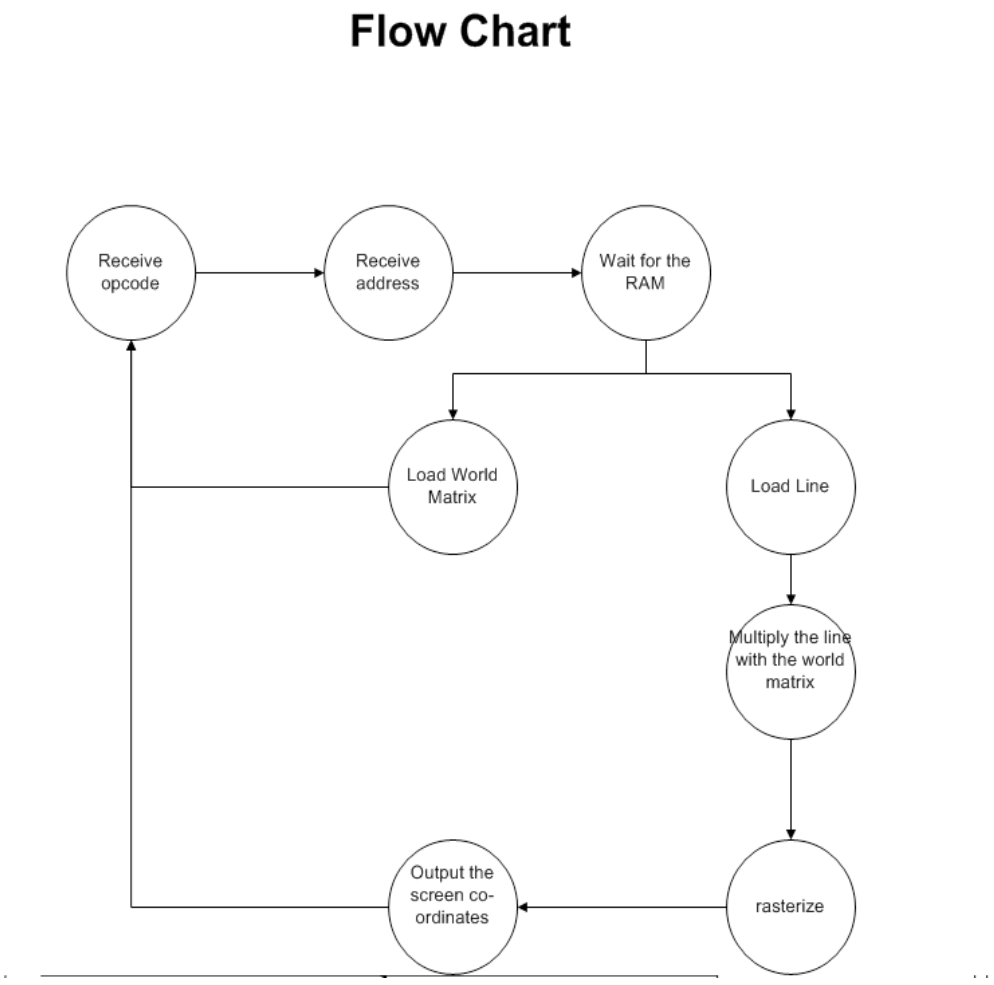


Figure 2.1.1

b) Operational Description:

In order to implement a GPU in the time available, we have made several important simplifications to the functionality of a standard commercial GPU. First, we are performing all calculations using 16 bit integers instead of floating point numbers. Support for floating point numbers would take far more area and time than we have available. However, we decided to internally represent decimal precision as (decimal numbers * 256). This gives us a range of -128 to 127 with a maximum precision of 1/256, or .003. This is effectively a fixed-point decimal. Thus, 10 in decimal would be represented as 9*256, or 2304 in decimal or 900 in hexadecimal. The number 256 was chosen because it is a power of two and dividing by it to turn the number into an integer is a simple bit shift. These numbers will be used for the input points as well as the input matrices. This

means that for a line primitive the RAM will store 12 bytes (6 bytes for each point) of data contiguously.

Our solution adopts a similar approach to OpenGL by using a “World Transform Matrix”. This is a 4x4 transform matrix that encodes information about rotation about all three axes, scaling, and translation. Our processor will be able to load this matrix from its RAM. The intended usage scenario is that the accompanying general purpose processor will place the prepared transform matrix in the RAM along with the 3D coordinates of the lines to be rendered. The processor will then send opcodes to our GPU load a transform matrix, and draw lines. The load matrix and draw line commands will reference a memory address in the RAM and the GPU will load the relevant 16 bit numbers, as discussed earlier. After loading the matrix and the two points of a line, our GPU will multiply them together to get the scaled, translated, and rotated endpoints of the line. We will use a simple orthogonal projection to represent our scene, so we will throw out the Z component of each point to get the screen coordinates of the endpoints. These endpoints will then be sent to our rasterizer. The rasterizer will use Bresenham‟s algorithm to determine the points in the frame buffer that should be shaded and write the proper information to the frame buffer.

The approach of using a world transform matrix makes loading objects defined in their own coordinate systems very convenient. If the user wanted to draw a cube that was defined with sides of length 1 in object coordinates as a cube with side length 2 at the point (1,2,3), all that is required is a transform matrix including a scale of 2 and a translation of (1,2,3). The next step is to input the lines that define a cube centered at the origin and the cube would be drawn in the desired location at the desired scale. In order to draw another cube at a different location or orientation, the user only needs to input a different world transform matrix and issue the same set of opcodes for drawing the cube.

c) Algorithms:

Our design implemented two algorithms related to graphics processing in hardware. We were able to perform matrix multiplication between a 4x4 transform matrix and a 4x1 vector representing a point in 3D space. The 4x4 matrix allows us to encode rotations, translations and scaling. Matrix-Vector multiplication involves multiplication and addition operations. We managed to get a signed multiplier and adders working. The multiplier was made combinational by making it process one matrix row and one point at a time. So, at one time, it would use this data to calculate one coordinate of the result (multiply the point and row data, and then add them) and shift that out to a buffer, and then ask for new point/row data as needed. Four points were calculated for one operation, which would be x1, y1, x2, y2 in that order. We also perform rasterization of lines using an modified implementation of Bresenham‟s algorithm. The algorithm is based on the idea of iteratively turning on the pixel that would result in the lowest error between the generated line of pixels and the „ideal‟ line. The modifications allow the algorithm to deal with integer start and end points, and any slope. Our implementation uses no signed arithmetic and uses bit shift to multiply and divide. The algorithm proceeds as follows:

1. Determine the magnitude of the difference between x2 and x1, and between y2 and y1, dx and dy respectively and set the increment direction variables iX and iY appropriately.
2. If dy>dx, the line is steep. Set steep to 1 and swap the values of iX and iY, X and Y, and dx and dy.
3. Begin at the first given endpoint. Set the total error to 256. Start at pixel x
4. Add dx to the error. Add 1 to x.
5. If the error is greater than 256, subtract dy from error and add 1 to y.
6. Turn on the pixel x,y, or if steep =1, turn on pixel y,x
7. Go to step 2 until x equals the x of the right endpoint.

When the algorithm specifies that we should turn on a point, it calculates the memory address of that point and which bit of that address"s data the pixel corresponds to. It sends this information to the controller to be written to memory. The swapping in the second step is necessary because the algorithm can at most increase x and y by one each iteration. Thus, for a line with a slope greater than 1, it will not be able to keep up. However, a line with slope of greater than 1 will by definition never increase by more than 1 in the x direction, when y increases by one, so swapping resolves the issue.

c) Format of data input and data output

The GPU is controlled by opcodes sent by the processor, with a hex value of "0001" representing "load matrix starting at <address>" and "0002" representing "draw the line found at <address>". In this case the address is a 16 bit hexadecimal number. Figure 2.1.3 shows the corresponding information in the RAM. The frame buffer is fixed from 0 to 4095, while the arguments for the opcodes can refer to data anywhere in the RAM that is not the frame buffer.

Opcode	Address
1	5000
2	5016
…	5022

Figure 2.1.2- Example opcode sequence

Ram Address	Contents	Description
0	16 Bits per address (one per pixel)	Frame Buffer
1		
…		
4094		
4095		
5000	Mat(1,1)	World Transform Matrix
5001	Mat(1,2)	
…	…	
5015	Mat(4,4)	
5016	X0	Line Primitive
5017	Y0	
5018	Z0	
5019	X1	
5020	Y1	
5021	Z1	

Figure 2.1.3- Example contents of RAM

2.2) Requirements:

Since the purpose of a GPU is to accelerate graphics calculations, our main focus for optimization was speed. We attempted to structure our design so that the internal data flows efficiently and is properly buffered, especially when performing lengthy calculations such as matrix multiplication.

An effective GPU not only can perform arithmetic swiftly, but it can also communicate with the CPU, system RAM, and its dedicated RAM quickly, in order to receive commands, output results, and cache intermediate values. Thus, memory and I/O throughput are important considerations in the design of a GPU and we tried to optimize them appropriately.

3) Final Design:

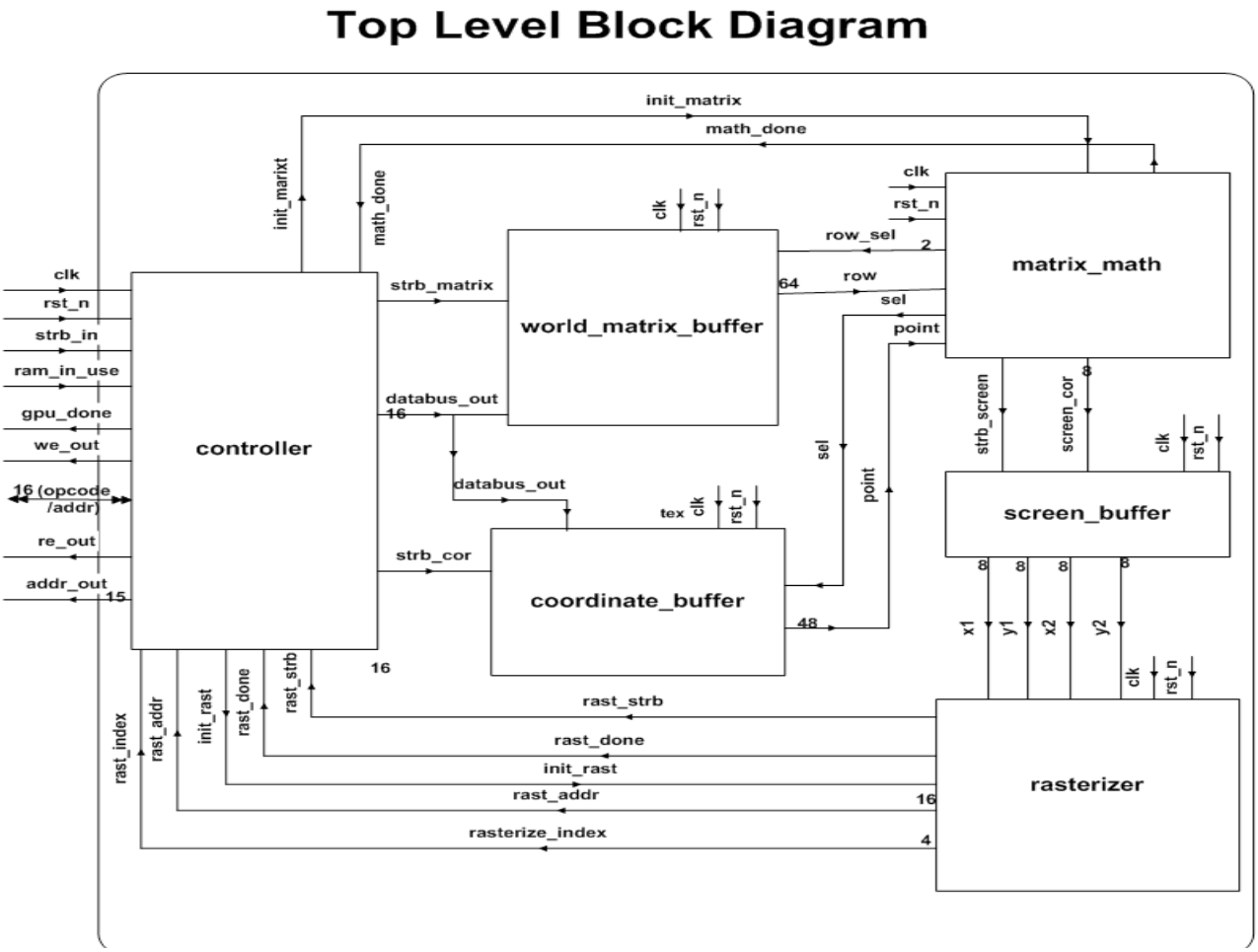


Figure 3.1.1

The top level design is shown as above. Now we shall discuss each of the above blocks, with RTL level discussions for the ones with significant complexity.

3.1) Block Diagrams

1) Controller - The controller receives the instructions to be performed from the processor/micro-controller. It also receives and sends signals to the processor/micro-controller to indicate that it is using the RAM currently. The data received from the processor is interpreted in chunks of 16 bits of data. It sends the address of data to be loaded to the external RAM. It sends signals to the World Matrix and Co-ordinate buffer to load the data from the RAM. It also sends/receives a signal to/from the Matrix Math to start the process on the data and to indicate completion of the process. It sends

signals to the Rasterizer to indicate that the data is valid and it can start rasterizing it. It receives a signal from the rasterizer to indicate the completion of its process. Overall, it controls the data flow and operation of the rest of the blocks. The controller has muxes, combinational logic and registers as shown in the Figure 3.

Controller RTL Diagram

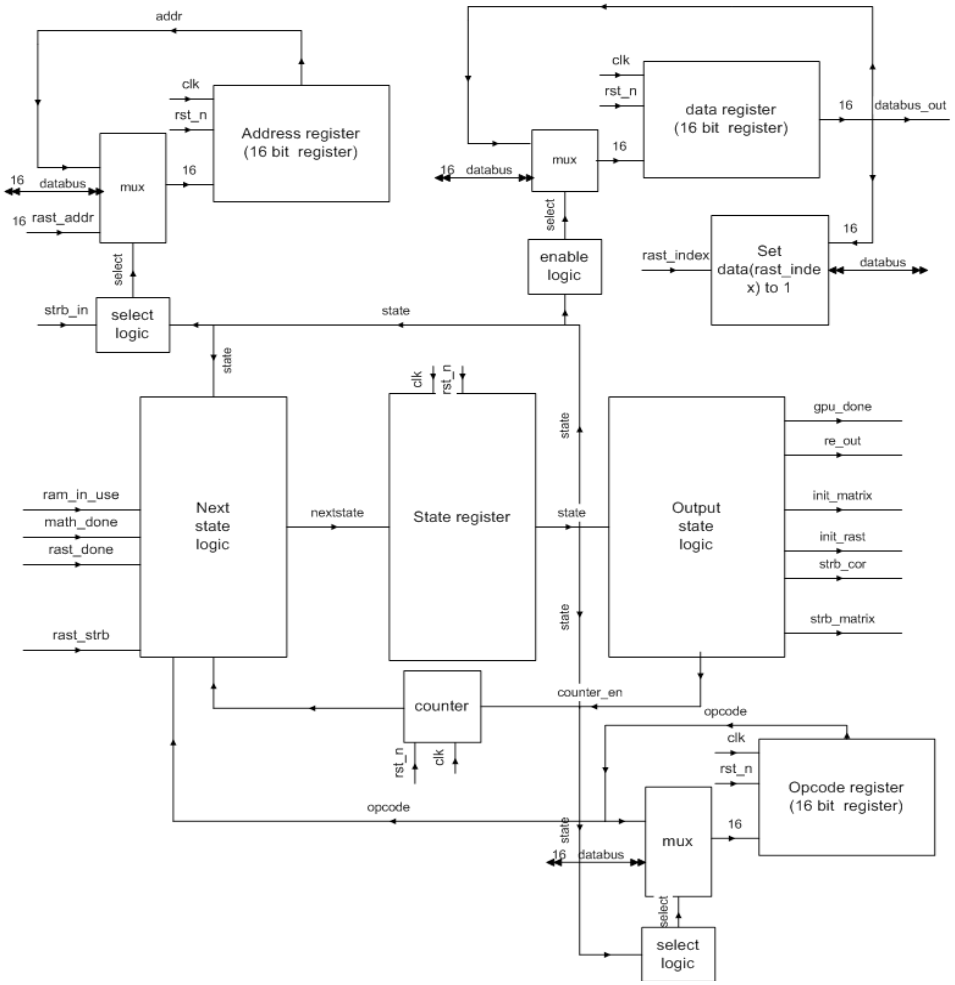


Figure 3.1.2

2) Matrix Math -It receives a signal from the Controller to multiply the matrices. It multiplies the 4x4 transformation matrices with the 4x1 input vectors to facilitate translation, rotations and scaling of the line primitive. It gets the three coordinates of a point and one row of the World Transform Matrix at a time, multiplies them and adds them to produce one transformed co-ordinate. The result is outputted to the Screen Coordinate buffer, and signals like row_sel and sel are used to ask for the next line/point. This process is repeated 4 times to get the x1, y1, x2, y2 of the line to be rendered. Finally, a done signal is asserted to the controller. At the RTL level, this block has 3 signed multipliers and

three adders implemented combinatorially. Also, there is a state machine to help choose the correct value from the combinational block (which has massive delays) and also asserts the correct signals to ask for the next point/row data as needed.

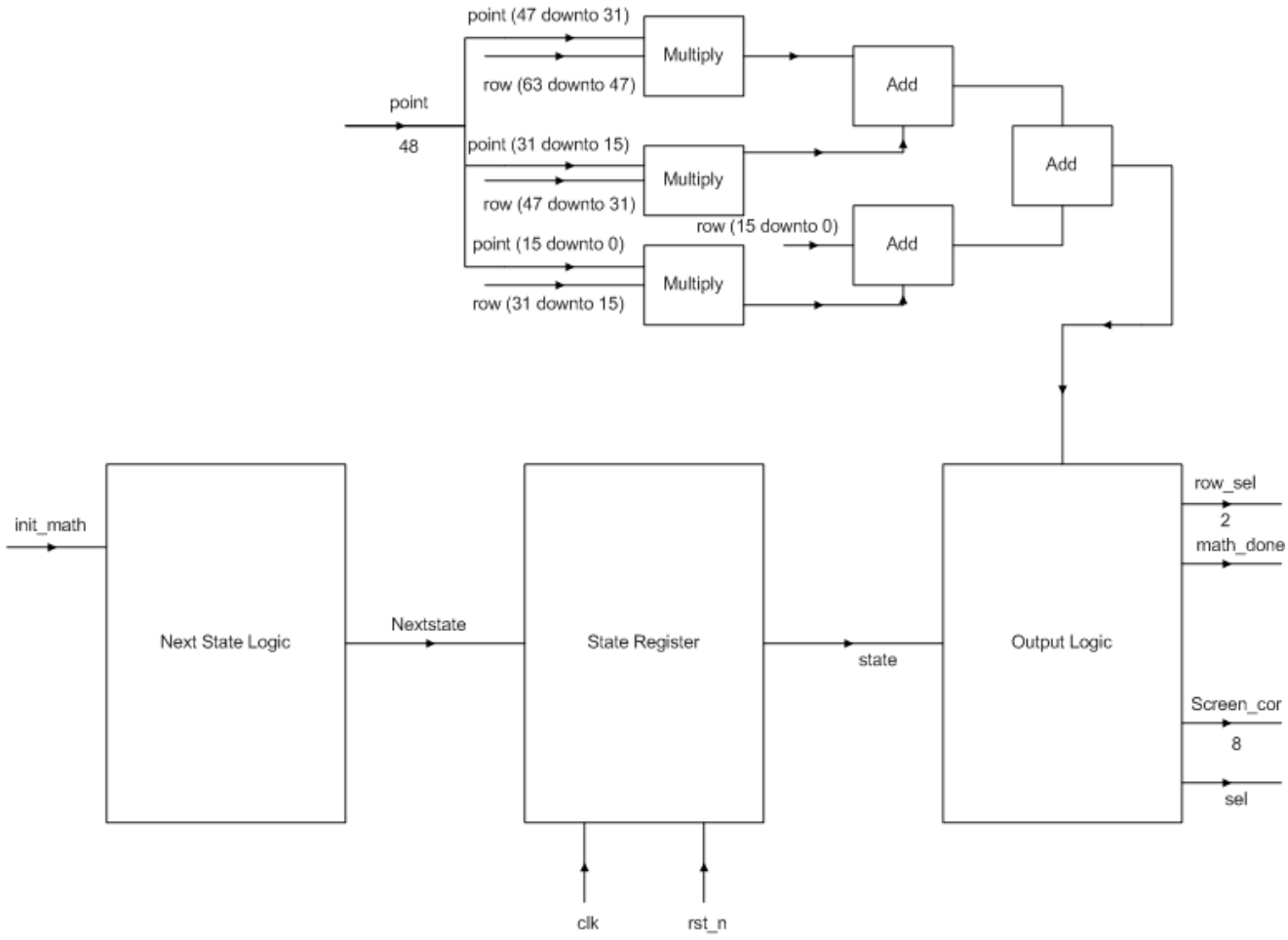


Figure 3.1.3

3) Rasterizer - The rasterizer reads the line primitive stored in the Screen Co-ordinate buffer and calculates the pixels that need to be turned on between the endpoints using Bresenham's Algorithm. As described earlier, the algorithm begins at one endpoint, classifies the slope of the line, and iteratively determines the x and y coordinates of the next point in the line. It makes this decision using a running estimation of the error between the point and the ideal „correct" line. These operations require addition, subtraction, multiplication, and division. Fortunately, an implementation of the algorithm was found online that only divided or multiplied by a factor of two. This was incredibly beneficial, as it removed the need for any slow and complex multiplier or divider blocks in the rasterizer.

The structure of the vhdl code for the rasterizer was essentially a state machine, with a different state for each step in the procedure of the algorithm. While this method was effective for coding and debugging, it is not evident how the design software managed to produce a synthesized version of the code. The output was both quick and relatively small, which saved us the time of planning out a datapath and interface logic for it. The included rtl diagram represents a realization of the algorithm based on the datapath approach, the adder, subtractor, and comparators are muxed to all of the data storage flops and counters. The software may have produced something completely different.

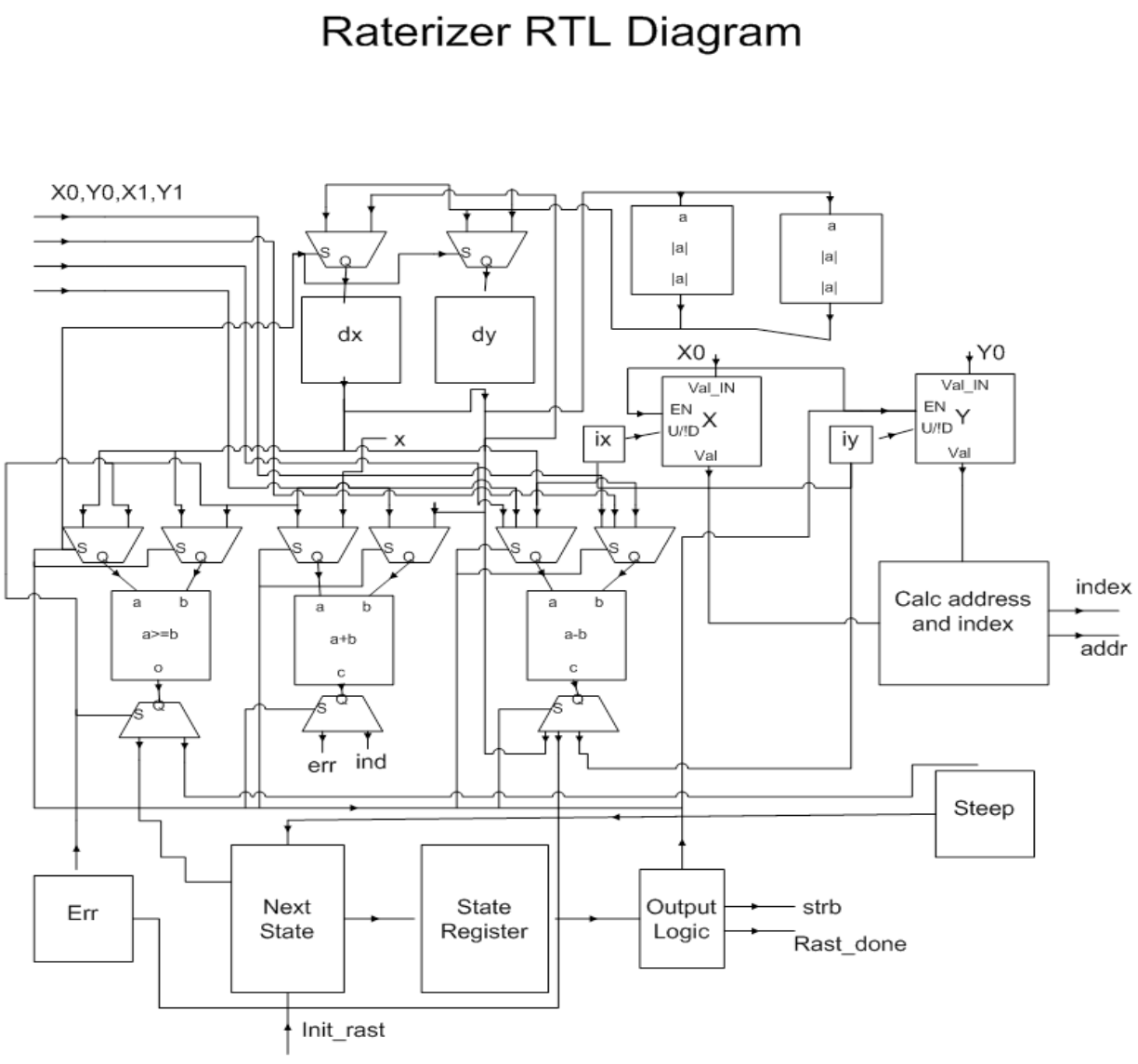


Figure 3.1.4

4) Screen Coordinate Buffer – The screen coordinate buffer stores the output of the Matrix math as two (X,Y) points, with X and Y as 16 bit fixed-decimal numbers. These points represent the coordinates of the 3D points after they have been projected onto the screen. It makes this information available to the rasterizer.

6) World Matrix Buffer – The world matrix buffer stores the transformation matrix read from the RAM and provides it as an input to the matrix math block. This matrix affects all lines read in from RAM.

7) Coordinate Buffer – The coordinate buffer stores the primitive line read from the RAM as two (X,Y,Z) points, where X,Y, and Z are all 16 bit fixed-decimal point numbers. It provides the points as inputs to the matrix math block.

3.2) External Devices

Our GPU is able to interface with an 16 bit external ram and a general purpose processor or micro-controller. The processor will first load the external RAM with the transform matrix and lines that it wants the GPU to operate on. Then the processor will send op-codes to the GPU along with a 16 bit address which are interpreted by the GPU 16 bits of data at a time. The GPU executes the instruction on the data from the RAM and outputs it to the frame buffer in RAM. There are two instructions: Load a primitive line (which consists of two coordinates) and load the world matrix (which loads a 4x4 transformation matrix). The load instructions are accompanied by a 16 bit address. The data is loaded from the RAM in chunks of 16 bits.

To ensure that the GPU can process data quickly, it is clocked at 100MHz. Both the RAM and the master processor will likely operate at a slower frequency. Thus, the controller block of our GPU is designed such that it is lenient in terms of timing requirements with external devices. We expect the RAM to have a 10 ns access time, but include a wait state, so that a marginally slower RAM will be read correctly. Additionally, for our opcode interface, we use a signal that the processor can assert when the current opcode or argument is valid, so that we do not impose an unreasonable timing constraint on the device. These features allow our GPU to operate correctly with a wide range of external RAMs and microprocessors, including the 68HCS12 from ECE 362.

In order for our GPU to coexist with an external RAM that is shared with a processor, our GPU‟s controller is intelligent enough to let the processor know when it is using the RAM and wait if the RAM is in use by the processor. It shares the addressing and data interfaces for the RAM by implementing Hi-Z outputs when the GPU is not using the RAM.

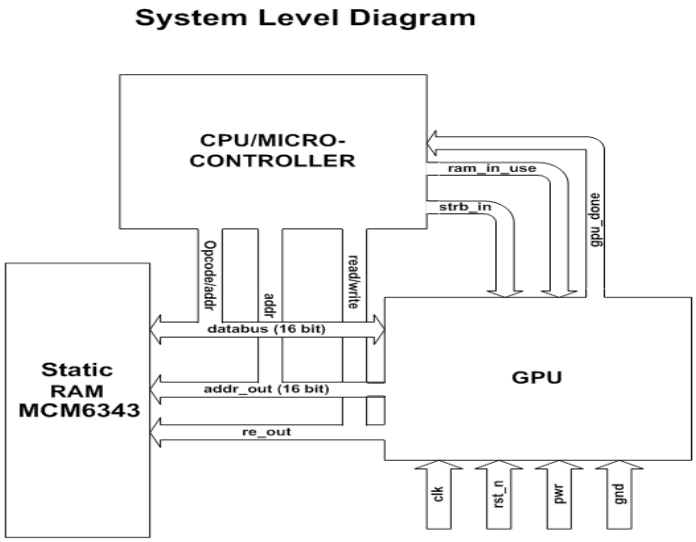


Figure 3.2.1

3.3) Timing and Area Budgets:

We were given Fixed Sucess critera to keep the area under 3mm x 3mm and the pin count under 40. Our target clock rate was set to be 10 ns. We first caluculated the area and timing budget for our design manually. As the project progressed we were able to get the timing and area budgets from the sysnthesis and after genrating the layout.

The table of the area and timing budgets from sysntehesis , SOC Encounter and Design Budgetting are given below:

	Design Budgetting	Synthesis	SOC Encounter
Core Area (mm^2)	3.82	3.03	5.28
Total Area (mm^2)	7.21	N/A	8.51
Critical Data Path	Matrix_math-World_matrix_buffer/Coordinate_buffer-Matrix_math-Screen_Buffer	Matrix_math-World_matrix_buffer-Matrix_math-Screen_Buffer	Matrix_math-World_matrix_buffer-Matrix_math-Screen_Buffer
Critical Path Delay(ns)	8.1	12.06	19.79

3.3.1 Estimating the dimensions of the I/O pad frame:

Total chip area = (1.785+(0.3 * 2*1.5)) ^2 = 7.21 mm^2

3.3.2 Comments:

During the Design Budgeting we estimated our time delays assuming that we have just three multipliers and adders in the matrix_math block. As the project progressed we decided to implement the VHDL's inbuilt signed multiplier. For signed multiplication extra logic would be required. This is one of the reasons the Critical Path Delay increased to 12.06 ns in the synthesis which is a reasonable increase since there are three 16 bit signed multipliers upgraded from the basic multiplier. The SOC encounter gave a very large increase in the Critical Path Delay . Since our design occupies a large area and there are several wide buses running across the chips, there are several long wires that cause long delays.

Even though the timing critical path delay is 19.79ns . We can still run at 10 ns as the only paths that has a delay greater that 10ns are through the matrix_math. The path through matrix_math - world_matrix_buffer - matrix_math – screen_buffer is purely combinatorial. The screen_buffer which shifts the data it gets when the matrix_math signals it to. The matrix_math also chooses the input that is going to be fed to the multipliers from the world_matrix_buffer and coordinate_buffer. Now we can get around the 19.76 ns delay by making the matrix_math shift the data into the screen_buffer only after two clock cycles and keeping the input to the multipliers stable during that period .

4) Testing

Matrix Math Block

The matrix math block was tested for different kinds of data. First, it was given the appropriate data with the the point (from the coordinate buffer) and row (from the world matrix buffer) being varied appropriately to see if the signals row_sel and sel were being asserted at the right times, and the math_done signal was asserted with the data being strobed out (strb_screen) at the right time (when it is valid). With these basic tests completed, the actual data was given. Both positive and negative data was checked. Sample positive calculation : 1 encoded in "fixed decimal" would be 1*256 = 256 which is 100 in hex. Thus "010000000000" represents 1, 0 and 0. Multiply this by the matrix row 0100000000000000 which is equivalent to saying 1*1 + 0*0 + 0*0 + 0 = 1. Now this one is added to 128 to give us 129 or 81 in hex. This is the value that should be shifted out of the math block!
The test bench had this value as well as others. After the positive check was completed, the negative values were checked in a similar fashion, where an input of -127 (x8100) to the math block with the identity matrix should just give out 01. After rigorous checking on this as well, the reset and other options were checked as well, proving the Matrix math block to be sound.
For the output waveforms, check Appendix B.

Rasterizer Block

The rasterizer is essentially a state machine with 8 modes of operation. When it is given a rast_init signal, the values of the endpoints are guaranteed to be stable. The rasterizer then compares the magnitude of each pair of values and the magnitude of their differences. It determines from these comparisons whether the line is "steep" (|slope |> 1) or not and which direction it needs to increment x and y in order to reach the end point.
To ensure that the rasterizer behaved properly for all of these cases, a test bench was created that instructed it to draw eight lines, representing all the possible combinations of slope magnitude, draw direction, and slope sign. The test bench additionally instructed it to draw a line from (0,0) to (255,255) to ensure there were no issues with long lines or the edges of the screen.
The implementation of this test procedure in a testbench was relatively straightforward, the testbench needed only to provide the proper X0, Y0, X1, and Y1 values as well as reset, rast_init, and clock. Initially, the test bench would wait for a rising edge on the rast_done pin before drawing the next line, but glitches on that pin for the mapped version required manually using a "wait for" statement for roughly how long the rasterizer took to process each line.

Gpu Controller

The controller orchestrates the operation of the whole GPU, and hence the test bench for this is quite complex, as all the blocks are involved. The controller is a moderately complicated state machine. Certain sets of states correspond to each of its duties: receiving opcodes, fetching data from RAM, instructing the matrix math and rasterizer blocks to run, and writing out the results of the rasterization. Each one of the controller‟s functions was evaluated separately.

The controller was tested to see if it was getting the data from the correct addresses from the RAM, as shown in the Appendix. We then tested to ensure that it correctly received the data and strobed the buffers at appropriate intervals. A particular challenge was balancing the time the controller took to write rasterized data to RAM and the speed at which the rasterizer works. After many collisions and misreads, we synchronized the two so that the whole process of rasterization and writeback took only 5 clock cycles to achieve. The waveforms for the tests have been included in the Appendix.

Overall

The top level test bench simply instantiated the GPU and course-provided SRAM and loaded the RAM with a specific test file corresponding to the input data for the transform matrices and lines to be drawn. The top level test bench then used Vhdl file I/O to read a file with opcodes and addresses and output them to the GPU one at a time. The primary challenge of testing the overall design was generating appropriate memory contents and opcodes to ensure everything was working properly.

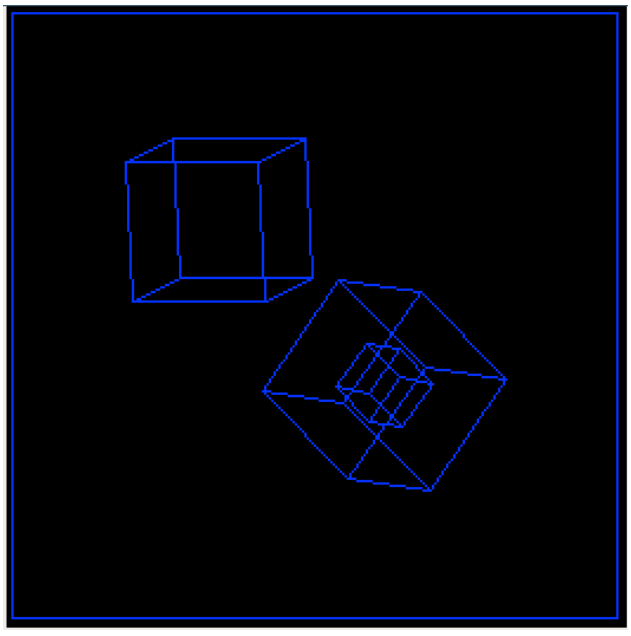


Figure 4.1.1- Output of completed program

5) Layout:

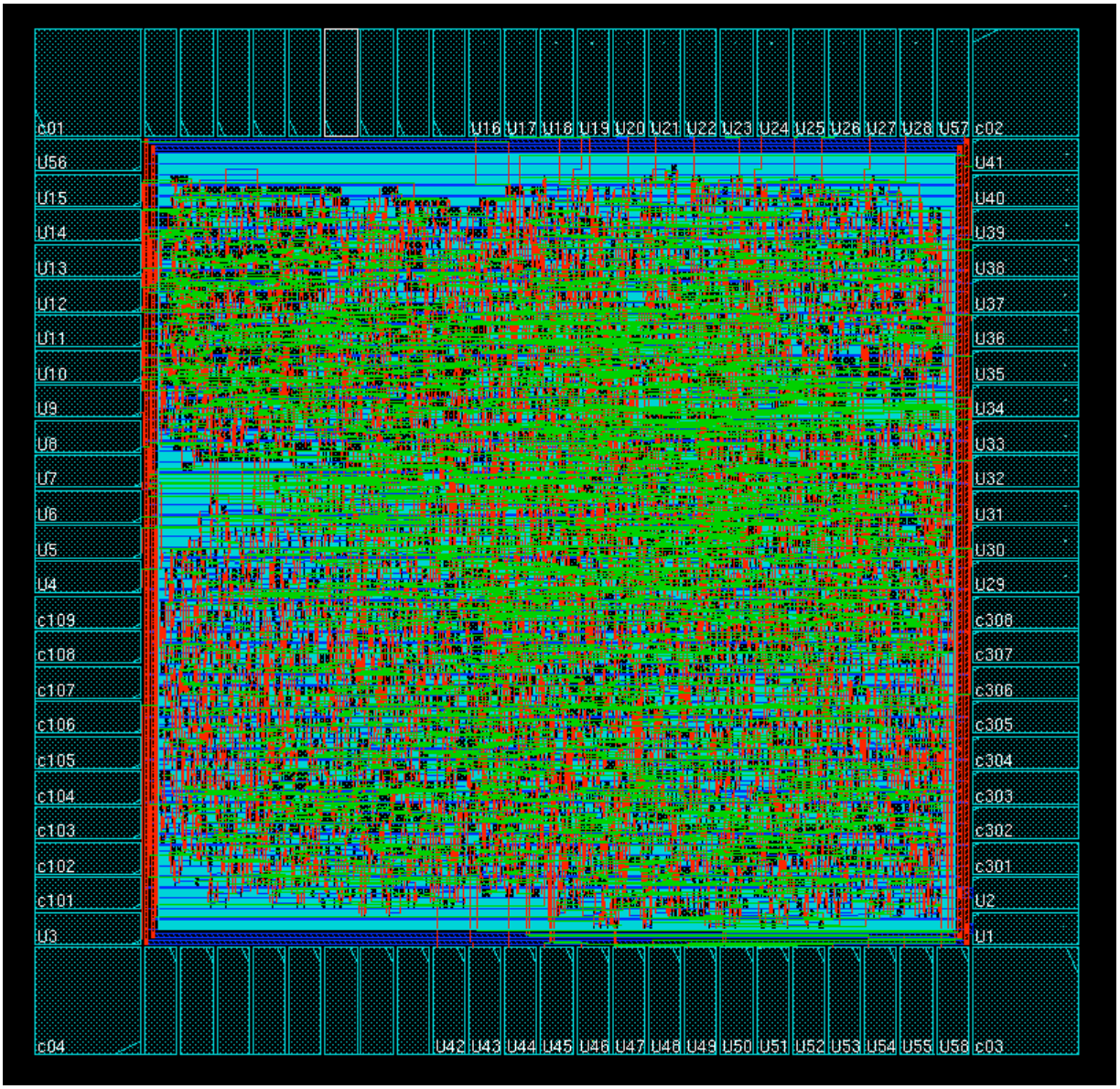


Figure 5.1.1- Final Layout

Aspect Ratio	1	Height	2.890 mm
Row Utilization	0.6	Area	8.5167 mm^2
Width	2.948 mm	Total Gates	43981

Figure 5.1.2- Layout Information

6) Results:

6.1) Fixed Criteria:

1. Test Benches exist for all top level components and the entire design. The test benches for the entire design can be demonstrated or documented to cover all of the functional requirements given in the design specific success criteria.

Status : Completed

2. Entire design synthesizes completely, without any inferred latches, timing arcs or sensitivity list warnings.

Status : Completed

3. Source and mapped versions of the complete design behave the same for all the test cases. The mapped version simulates without timing errors except at when the time is zero.

Status : Completed

4. A complete IC layout is produced. The IC layout passes all geometry and connectivity checks.

Status : Completed

5. The total area, including I/O pads, is no more than 3mm * 3mm and the required pin count (including power and ground) is no more than 40. (Half credit requires 10mm * 10mm and 100 pins)

Status : Completed

6.2) Design Specific Criteria :

1. Demonstrate by simulation of a VHDL test bench that the GPU can properly render a scene stored in RAM using either an orthographic or a perspective project.

Status : Completed

2. Demonstrate by simulation of a VHDL test bench that the Matrix Math component of the GPU performs it's calculations correctly.

Status : Completed

3. Demonstrate by simulation of a VHDL test bench that the Rasterization is working as expected.

Status : Completed

4. Demonstrate by simulation of a VHDL test bench that the interfacing with the external (no on-board) memory and the Screen Buffer is working.

Status : Completed

Appendix A

Contents:

A. Source Code
B. Test Benches
C. Timing Reports, Area Reports, Connectivity checks etc.
D. Test files and Java files
E. Datasheets

A. Source Code

Description	Location
The top level structural VHDL code that is a wrapper for all the blocks of the design.	source/overall.vhd
This is basically the control unit of the design and is also responsible for interfacing with the external RAM.	source/gpu_controller.vhd
The block that is responsible for matrix operations on the given data points.	source/matrix_math.vhd
The shift-register that stores the two points.	source/coordinate_buffer.vhd
The shift-register that stores the world transform matrix.	source/world_matrix_buffer.vhd
The shift-register that stores the points calulated by the matrix_math block.	source/screen_buffer.vhd
The block that is responsible for generating the pixel data to draw a line between the given two points.	source/Rasterizer.vhd

B. Test Benches

Description	Location
Testing the top level block .	source/tb_overall.vhd
Testing the world_matrix_buffer	source/tb_world_matrix_buffer.vhd
Testing the screen_buffer	source/tb_screen_buffer.vhd
Testing the coordinate_buffer	source/tb_coordinate_buffer.vhd
Testing the gpu_controller	source/tb_gpu_controller.vhd
Testing the rasterizer	source/tb_rasterizer.vhd
Testing the math block	source/tb_matrix_math.vhd

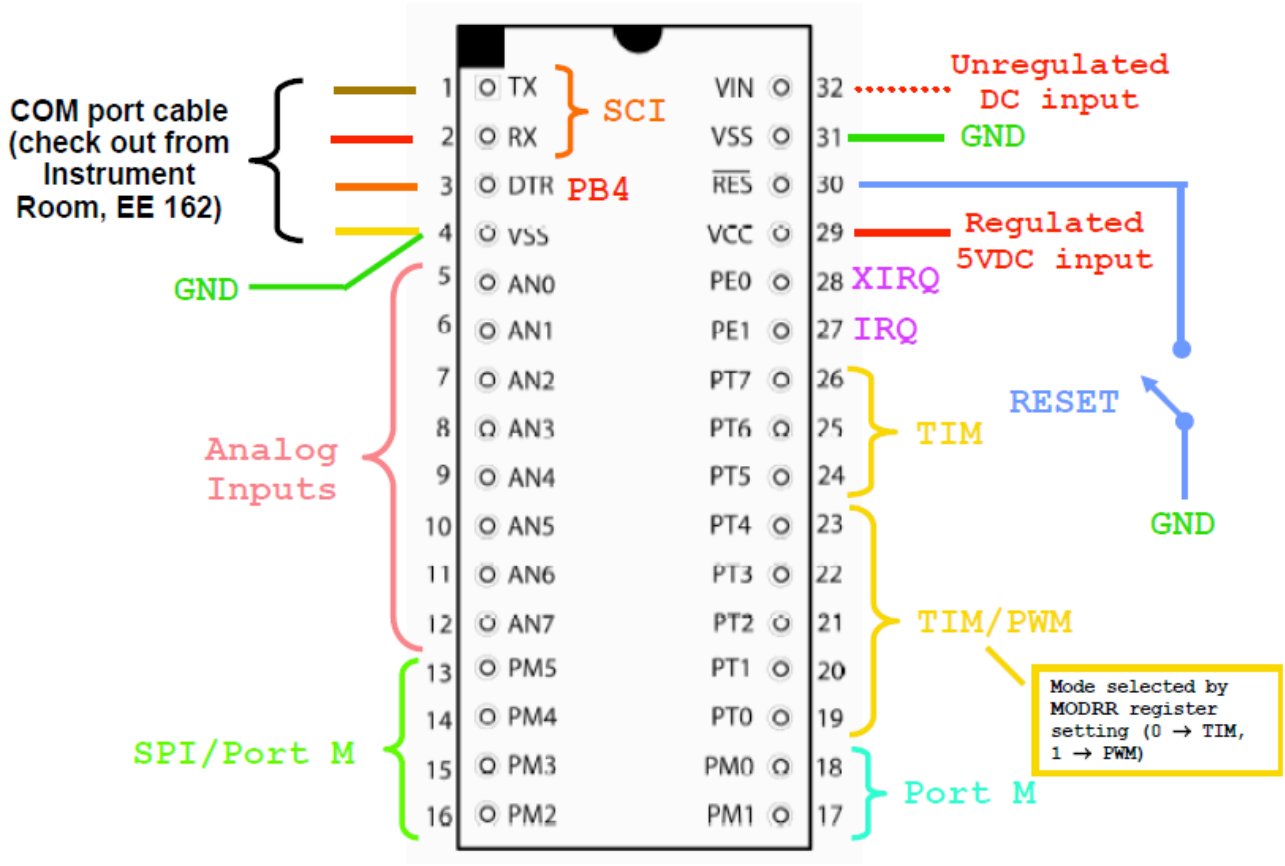
C. Timing Reports, Area Reports, Connectivity checks etc.

Description	Location
Timing Report for the overall chip.	timingReports/overall_postRoute_all.tarpt
Timing and Area Report for the Screen Buffer	reports/screen_buffer.rep
Timing and Area Report for the Coordinate Buffer	reports/coordinate_buffer.rep
Timing and Area Report for the World Matrix Buffer	reports/world_matrix_buffer.rep
Timing and Area Report for the Rasterizer	reports/Rasterizer.rep
Timing and Area Report for the Matrix math	reports/matrix_math.rep
Timing and Area Report for the GPU Controller	reports/gpu_controller.rep
Overall Area Report (mapped)	reports/overall.rep
Overall Connectivity Report after layout	encounter/overall.conn.rpt
Overall Density Report after layout	encounter/overall.density.rpt
Overall Gate Count after layout	encounter/overall.gateCount
Overall Geometry Report after layout	encounter/overall.geom.rpt

D. Test Files and Java Files

Description	Location
Inputs to draw a cube	source/memcube0.txt
Inputs to draw a diagonal line	source/memline1.txt
Inputs to draw a couple of cubes which are scaled and rotated	source/memcube1.txt
Output from the test bench that is fed to the cube0.java program to draw the cube (memcube0.txt).	source/meocube0.txt
Output from the test bench that is fed to the line1.java program to draw the diagonal line (memline1.txt).	source/meoline1.txt
Output from the test bench that is fed to the cube1.java program to draw a bunch of cubes(memcube1.txt).	source/meocube1.txt
Java program to draw the image from the outputs of the test-bench (meocube0.txt).	source/cube0.java
Java program to draw the image from the outputs of the test-bench (meocube1.txt).	source/cube1.java
Java program to draw the image from the outputs of the test-bench (meoline1.txt).	source/line1.java
A script to run the above java programs .	/demo.sh
Opcodes for drawing the object form memcube0.txt	/cube0
Opcodes for drawing the object form memcube1.txt	/cube1
Opcodes for drawing the object form memline1.txt	/memline1
The final presentation slides	Presentation/3D_gpu_final.ppt
Accepts an opcode file and outputs a memory contents file	source/MatrixGen.java

E. Datasheets



68HCS12 Microcontroller Pinout

Logic Input Timing Measurement Reference Level ..... 1.50 V
Logic Input Pulse Levels ..... 0 to 3.0 V
Input Rise/Fall Time ..... 2 ns

Output Timing Reference Level ..... 1.50 V
Output Load ..... See Figure 1

READ CYCLE TIMING (See Notes 1, 2, and 3)

Parameter	Symbol	MCM6343–10		MCM6343–11		MCM6343–12		MCM6343–15		Unit	Notes
		Min	Max	Min	Max	Min	Max	Min	Max		
Read Cycle Time	t_{AVAV}	10	—	11	—	12	—	15	—	ns	4
Address Access Time	t_{AVQV}	—	10	—	11	—	12	—	15	ns	
Enable Access Time	t_{ELQV}	—	10	—	11	—	12	—	15	ns	5
Output Enable Access Time	t_{GLQV}	—	4	—	4	—	4	—	5	ns	
Output Hold from Address Change	t_{AXQX}	3	—	3	—	3	—	3	—	ns	
Enable Low to Output Active	t_{ELQX}	3	—	3	—	3	—	3	—	ns	6, 7, 8
Output Enable Low to Output Active	t_{GLQX}	0	—	0	—	0	—	0	—	ns	6, 7, 8
Enable High to Output High–Z	t_{EHQZ}	0	5	0	6	0	6	0	7	ns	6, 7, 8
Output Enable High to Output High–Z	t_{GHQZ}	0	4	0	4	0	4	0	5	ns	6, 7, 8
Byte Enable Access Time	t_{BLQV}	—	5	—	6	—	6	—	7	ns	
Byte Enable Low to Output Active	t_{BLQX}	0	—	0	—	0	—	0	—	ns	6, 7, 8
Byte High to Output High–Z	t_{BHQZ}	0	5	0	6	0	6	0	7	ns	6, 7, 8

MCM343 External SRAM- Similar to course-provided SRAM

Appendix B

Matrix Math Test Waveforms

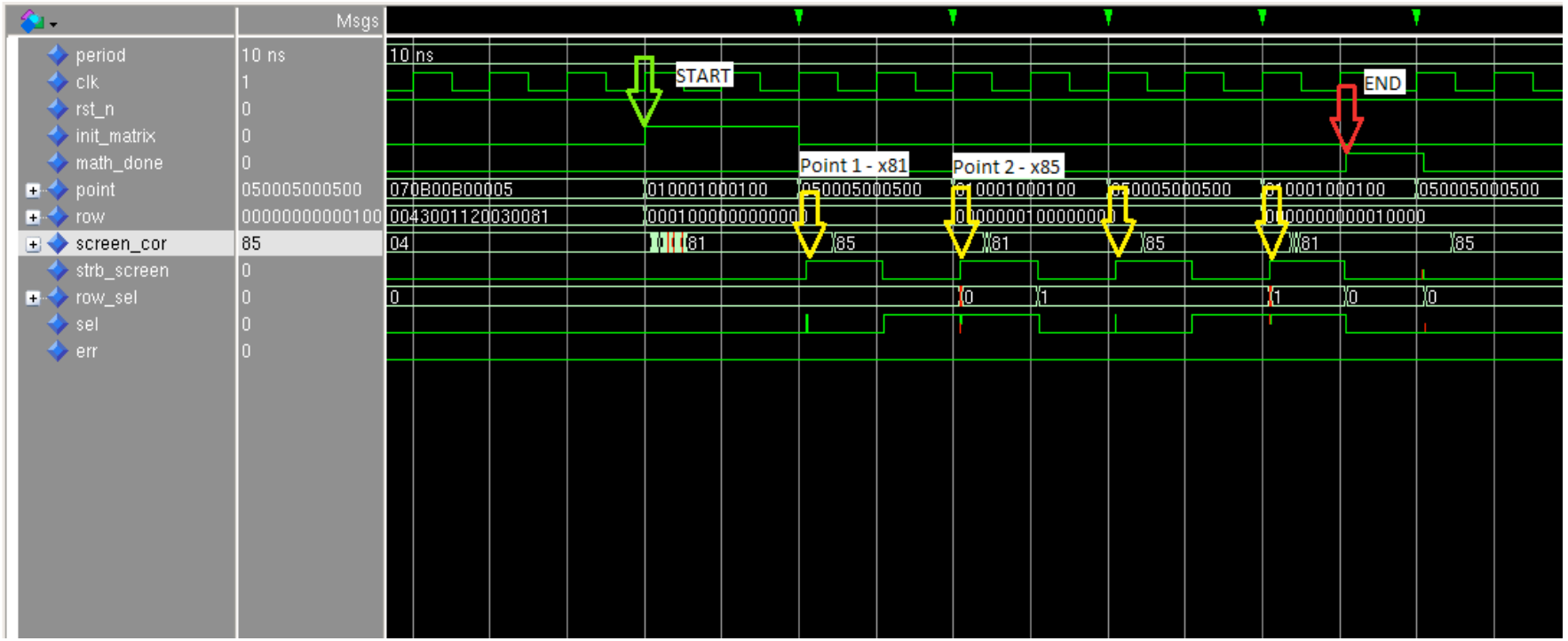


Figure A.1.1 : The waveform for the positive test data, showing the correct data and the correct number of points being shifted out.

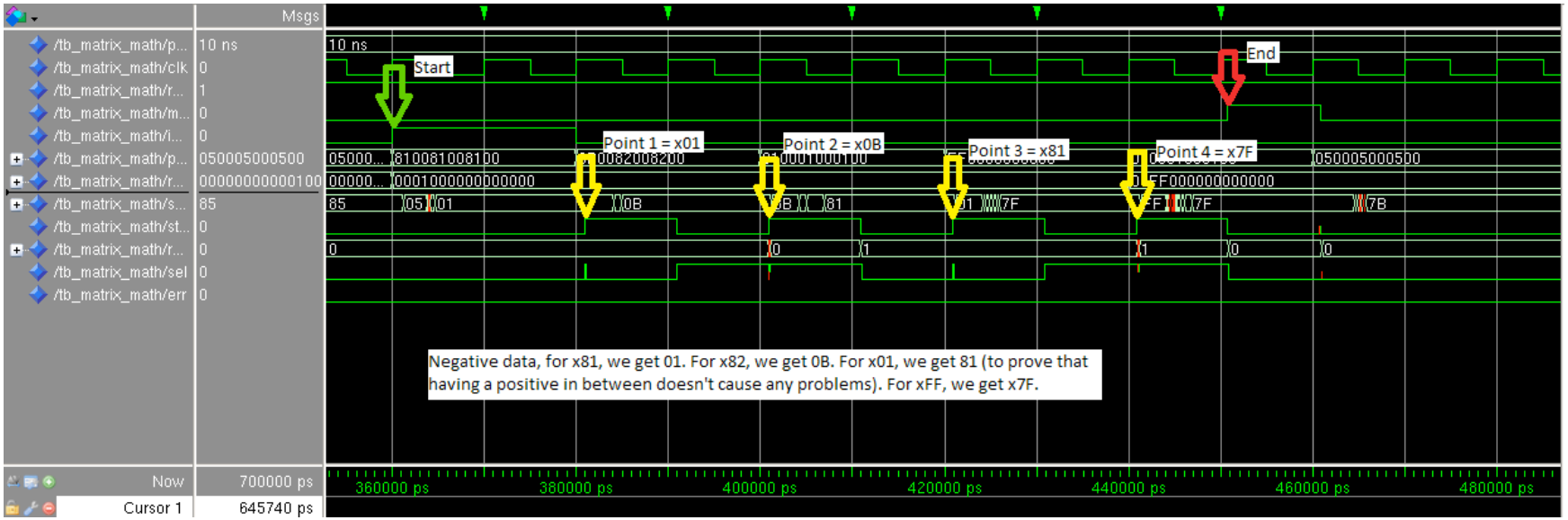


Figure A.1.2 : The waveform for the **negative** test data, showing the correct data and the correct number of points being shifted out.

Rasterizer Test Waveforms

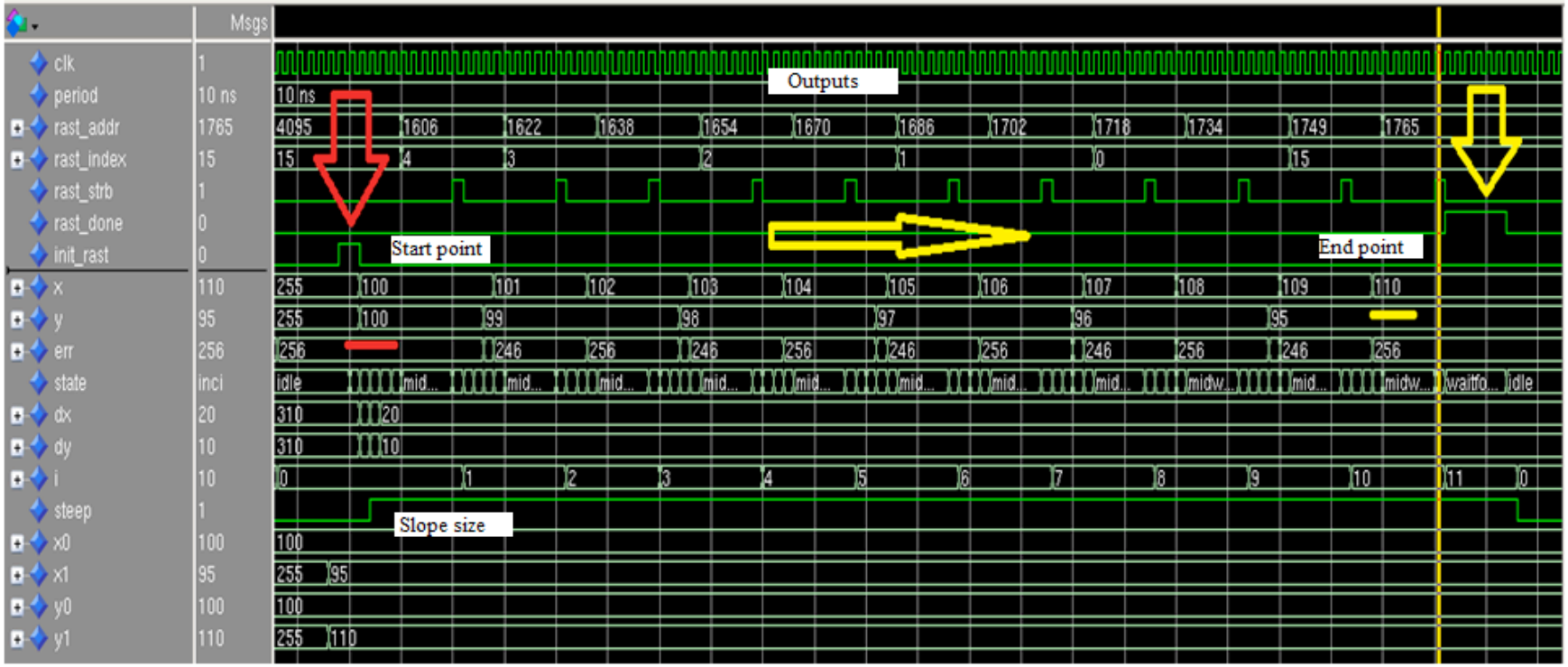


Figure A.2.1 : A standard test case for the rasterizer, drawing a line from (100,100) to (95,110).

The line in this picture falls under the categories of steep (|slope| = 10/5 > 1), right to left, and positive slope. After receiving the start signal, the rasterizer goes into a four state setup sequence where it differences the x and y values, sets the dx and dy values and calculates the slope"s magnitude. Since the slope is larger than 1, it sets steep to 1 in the next state and begins rasterizing at the right endpoint. After it calculates each point, it asserts rast_strb and moves onto the next one. When it finally reaches the left endpoint, it sends its address and announces to the controller that it is finished.

Controller Test Waveforms

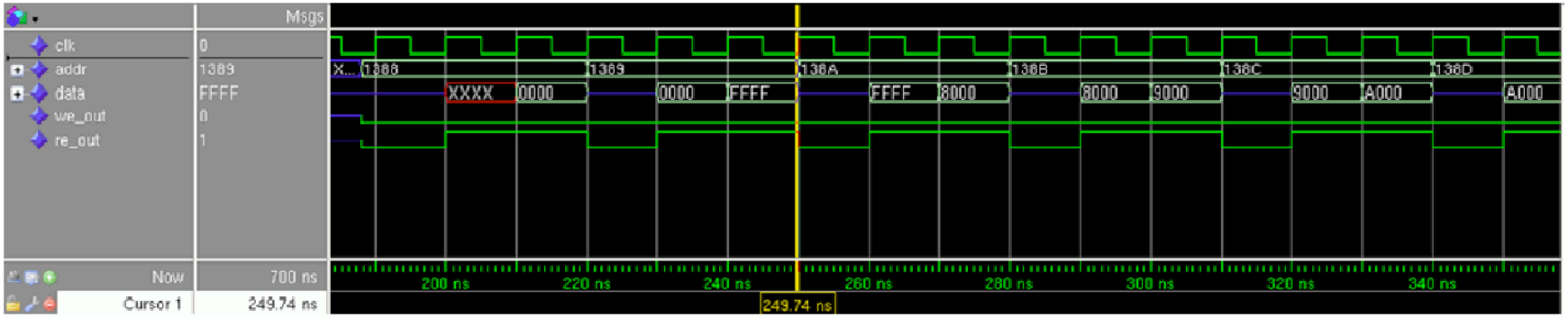


Figure A.3.1 – Ram Access

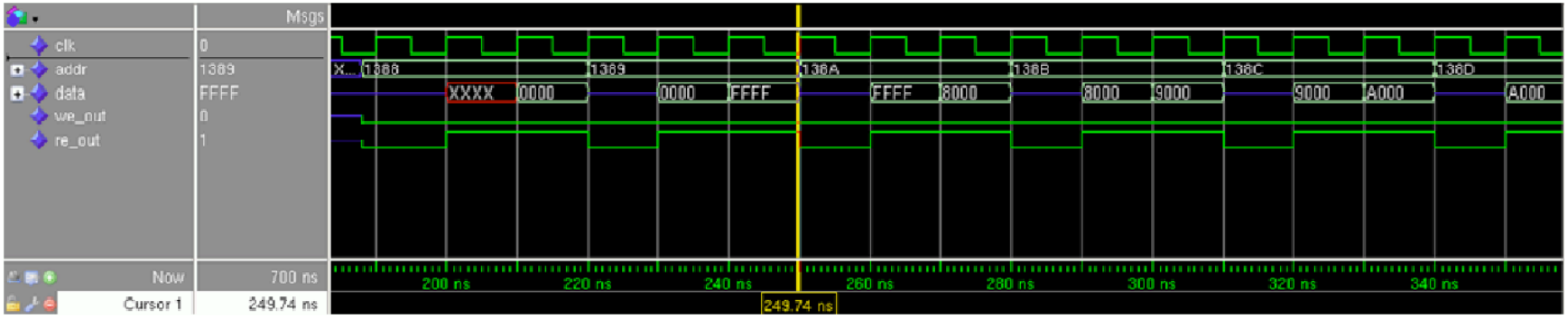
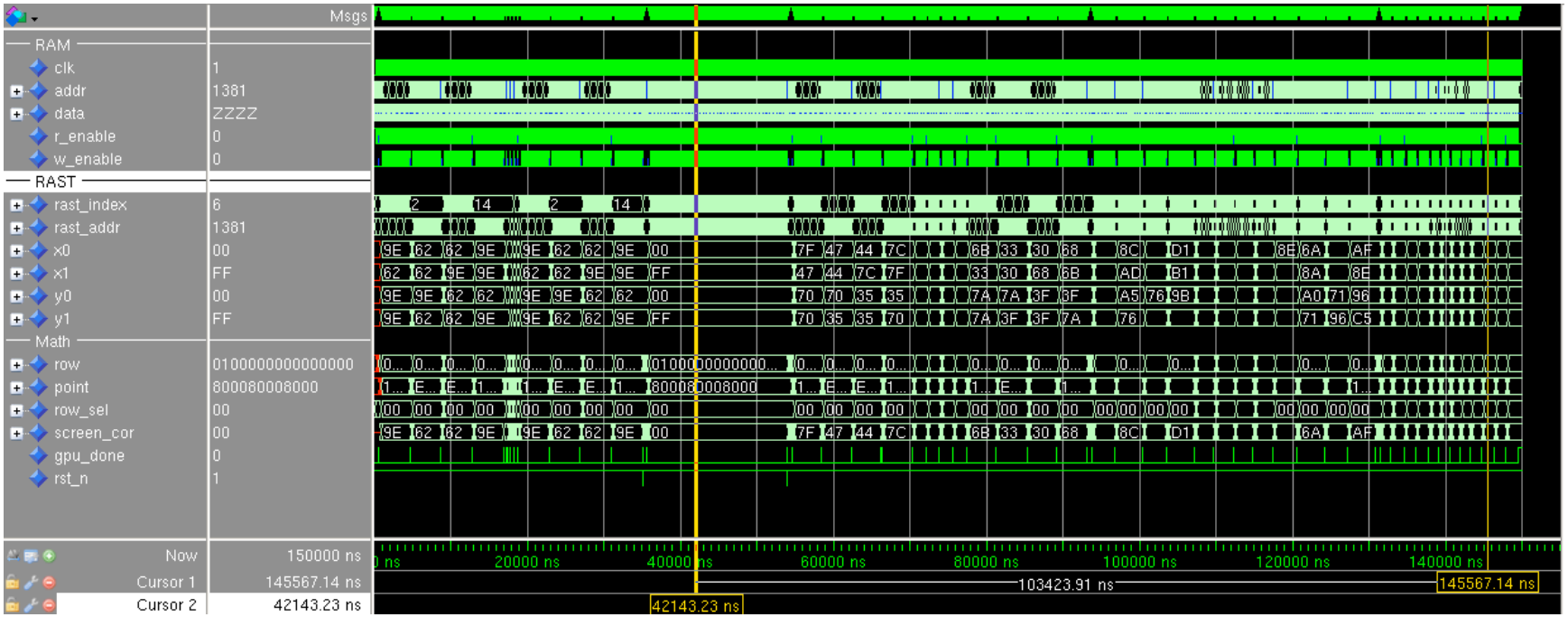


Figure A.3.2- Ram Access Zoomed out



Fgireu A.3.3- Overall Testbench output

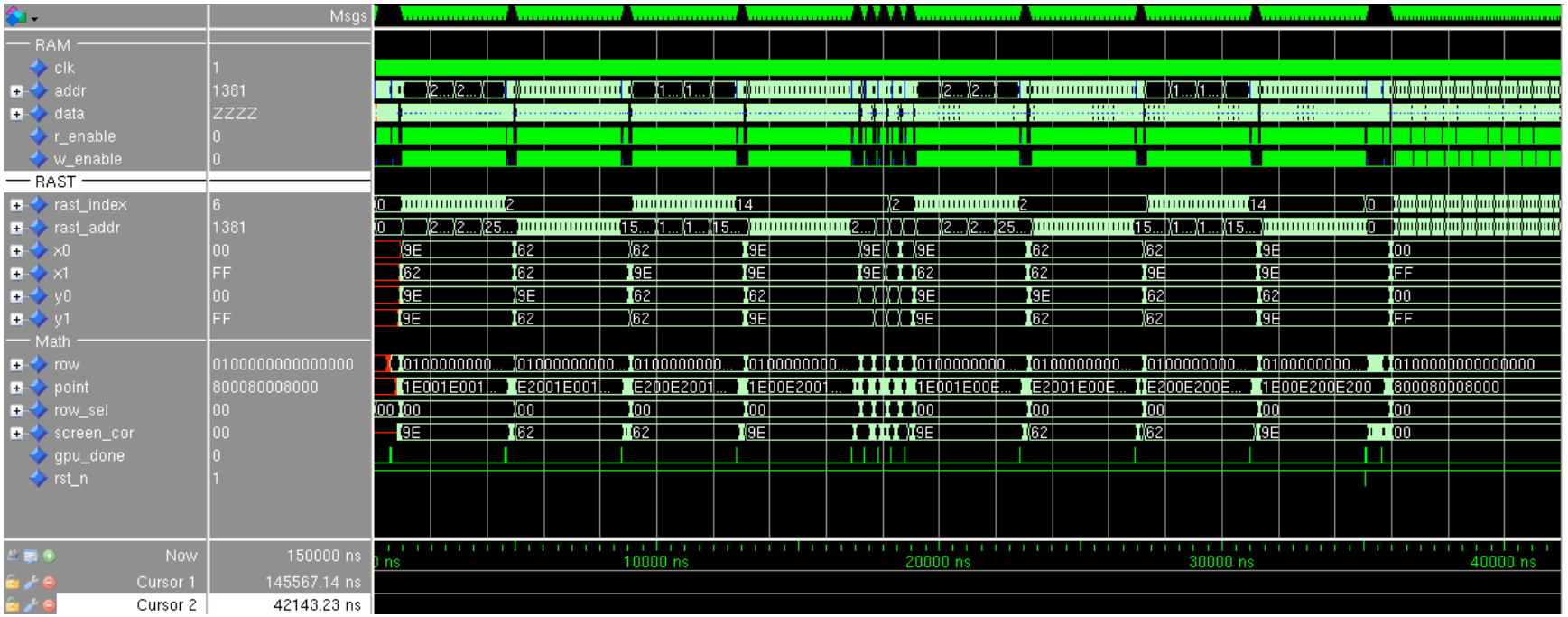


Figure A.3.4- Zoom in of First Line plotted

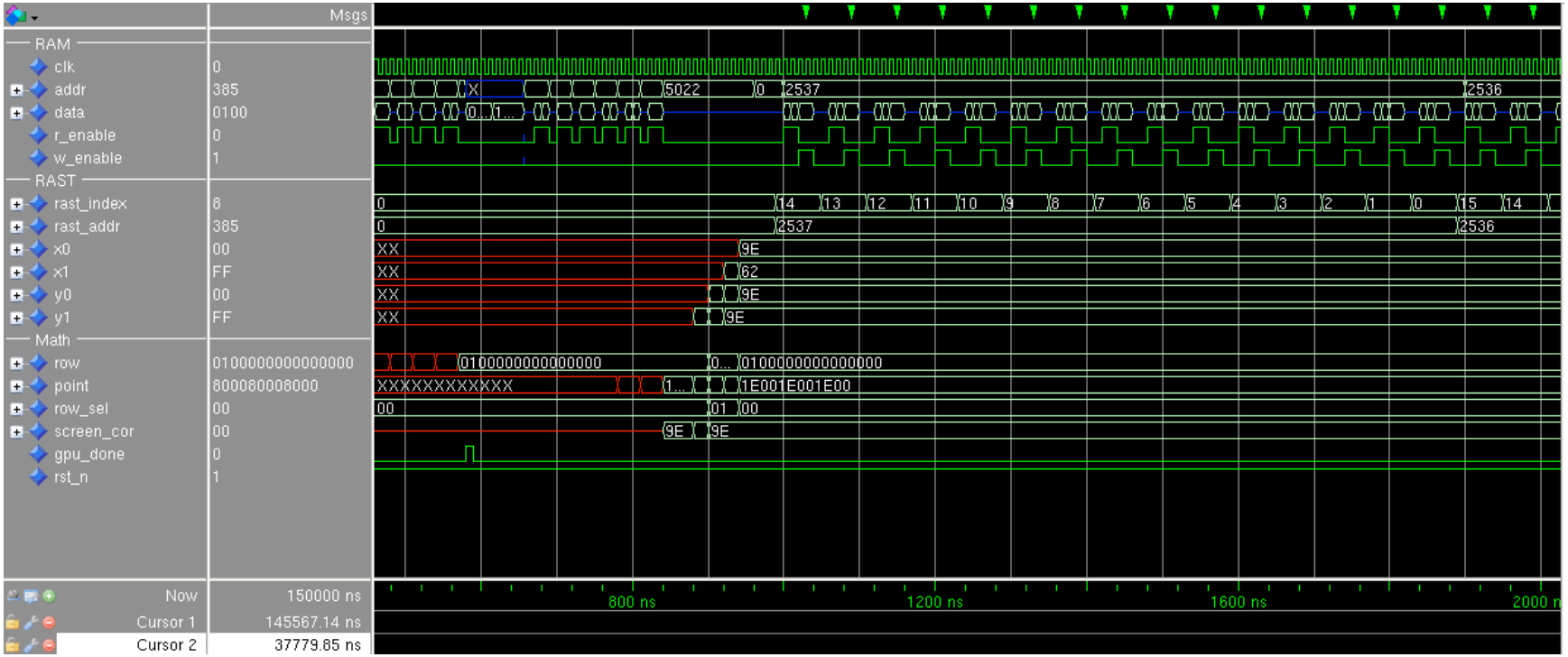


Figure A.3.5- First Point Read

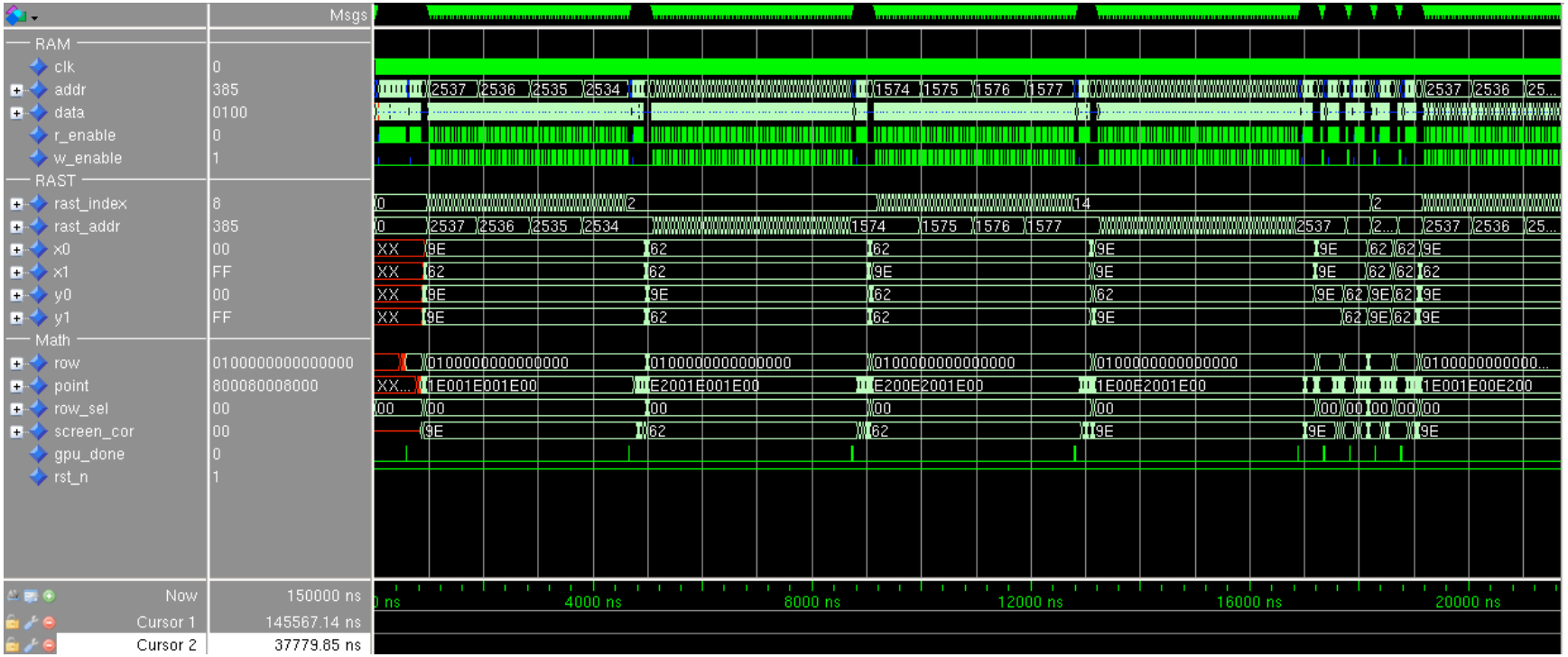


Figure A.3.2- Ram Access Zoomed out

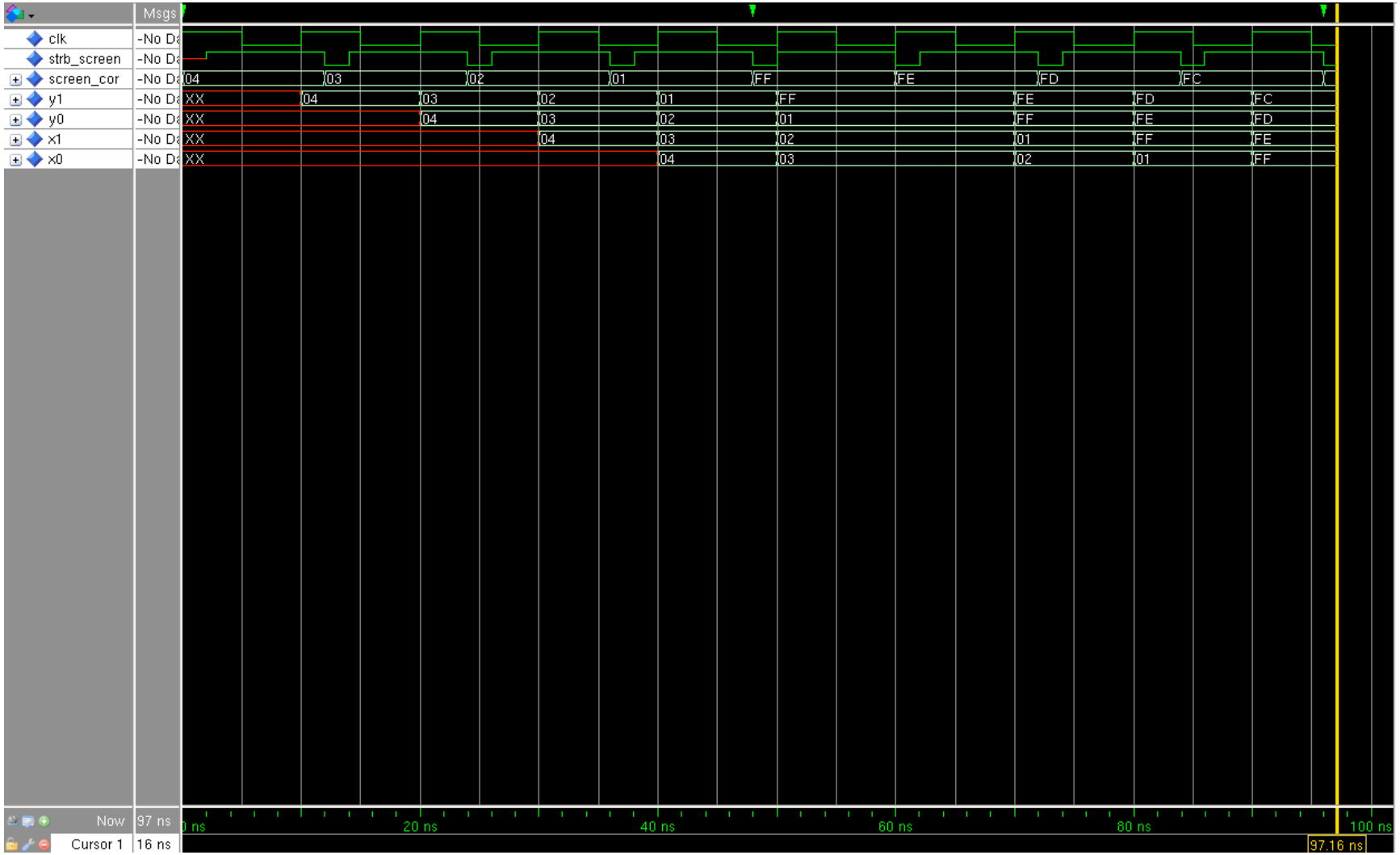


Figure A.4.1- Screen Coordinate Buffer

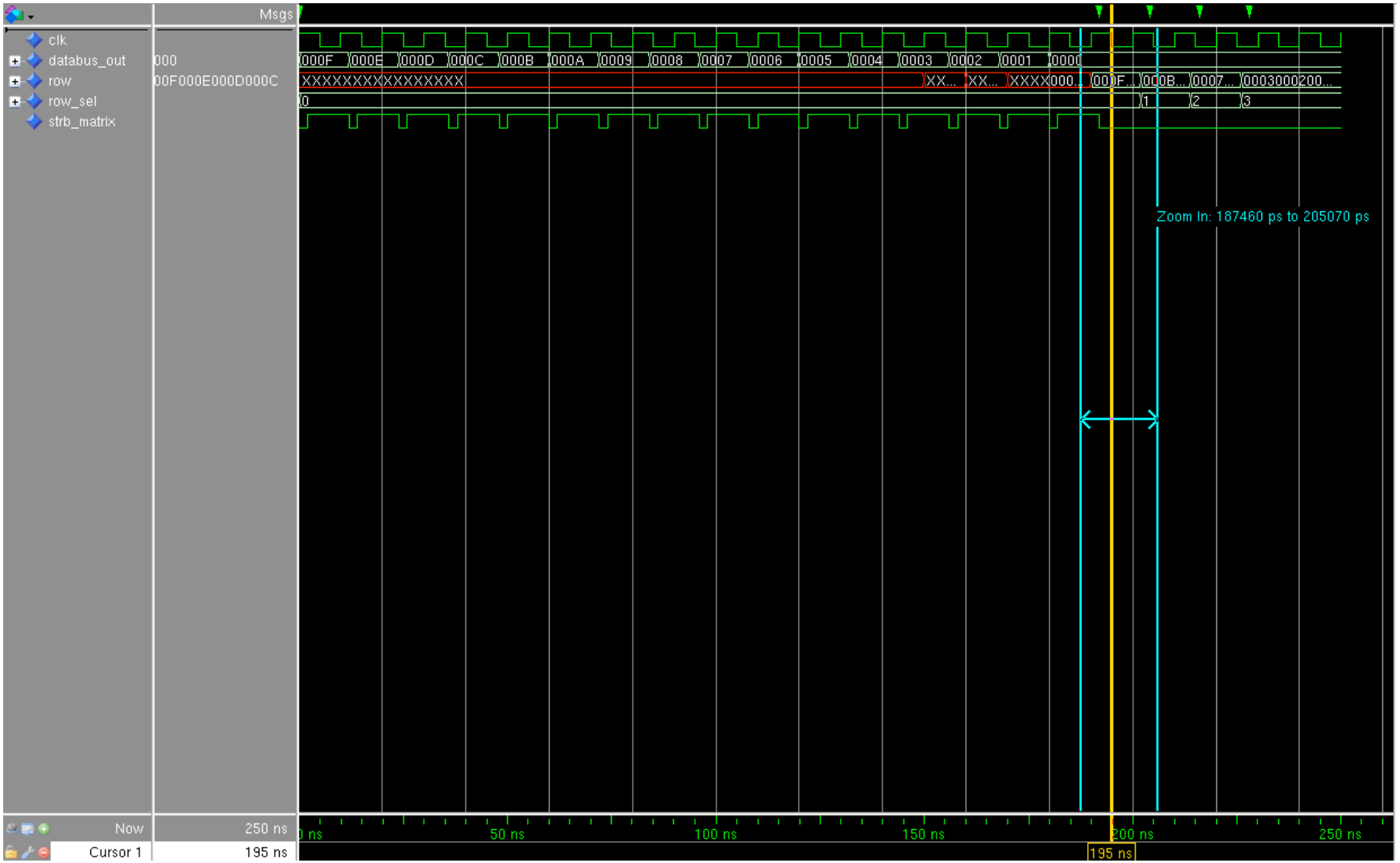


Figure A.5.1-World Matrix Buffer

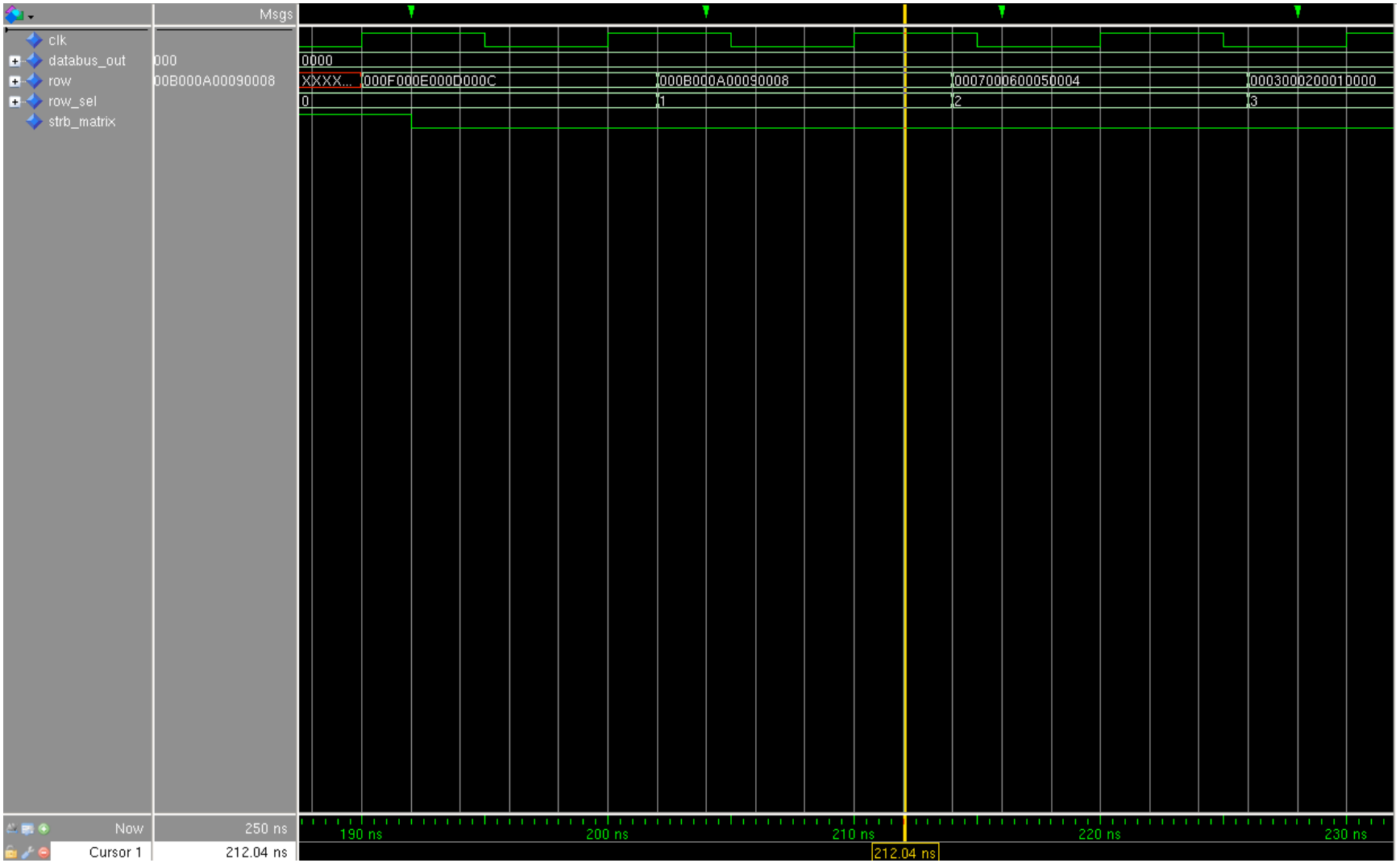


Figure A.5.1-World matrix Buffer

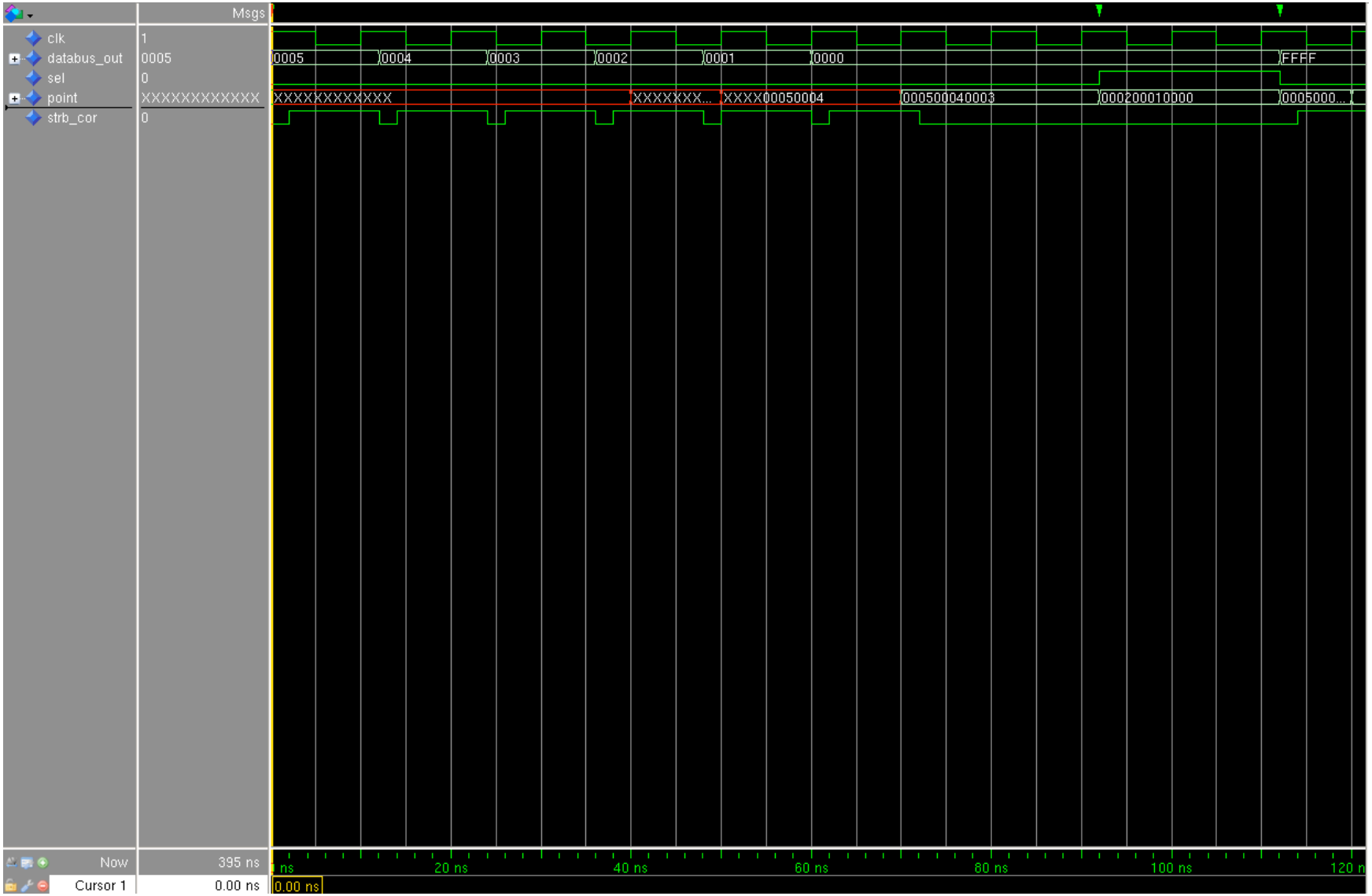


Figure A.6.1- Coordinate Buffer

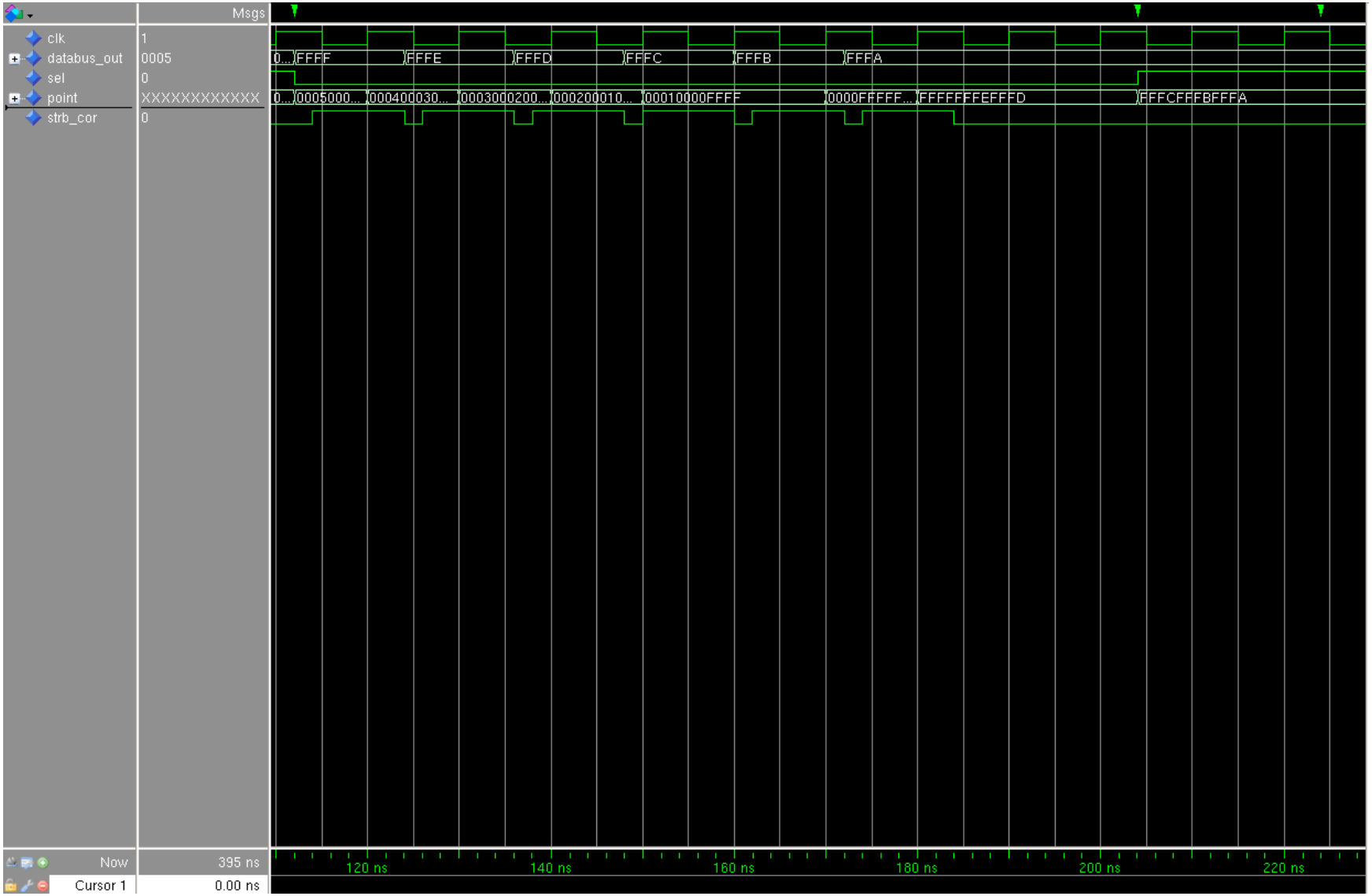


Figure A.6.1 Coordinate Buffer