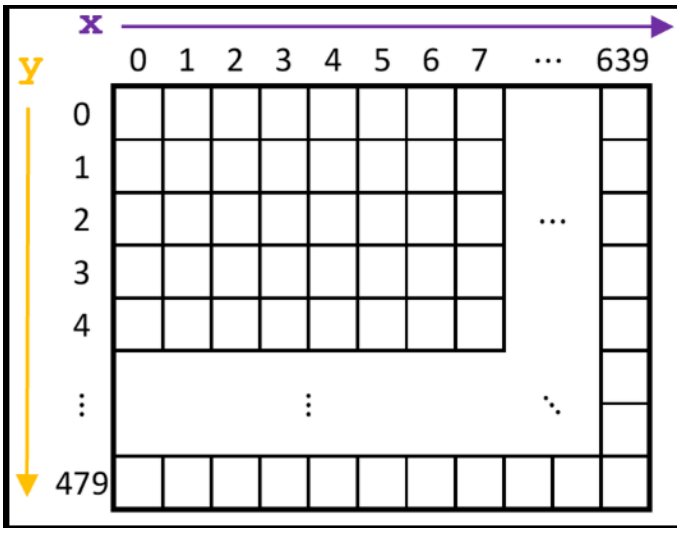
EE/CSE 371
Nov 16, 2025
Lab 5 Report

Lab 5

Design Procedure

This lab explores VGA graphics rendering on the FPGA using Bresenham's line drawing algorithm to display lines on a monitor. In Task 1 the focus is on establishing VGA connectivity and understanding the pixel buffer coordinate system. We verify that the VGA framebuffer can display images by initially showing a rainbow gradient before clearing to black, confirming proper connection between the FPGA and monitor. Task 2 implements Bresenham's line drawing algorithm to draw lines between arbitrary coordinate pairs. The algorithm avoids floating-point arithmetic by using integer error accumulation to determine which pixels to color along ideal line path. It handles all eight required line types: steep and gradual slopes in four directions (left-up, right-up, left-down, right-down). When a line is steep, coordinates are swapped to ensure iteration along the longer axis. This maintains smooth pixel transitions where x and y change by at most one pixel per a clock cycle. Task 3 integrates the line drawer into a drawing of a dog that upon assertion of the draw key (~KEY[1]), demonstrates the full algorithm functionality. Lines are shown sequentially drawn on screen using a state machine that coordinates transitions between waiting, clearing, and drawing states. A reset (~KEY[0]) function clears the screen by writing black pixels to all pixel columns, erasing the dog image. The system uses edge detection on the done signal to sequence line drawing and reset the line drawer module between lines. All components are combined in the toplevel DE1_SoC module that manages VGA output timing, connects to line drawing algorithm logic, pixel color selection, and the state based drawing control FSM.

Overall System Descriptions, Charts and Block Diagrams



Figures 1: Buffer pixel coordinate system

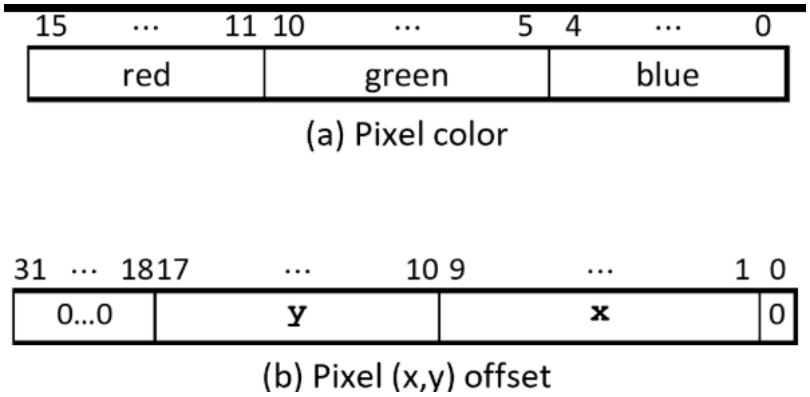


Figure 2: Data layout of pixel color values and address offsets

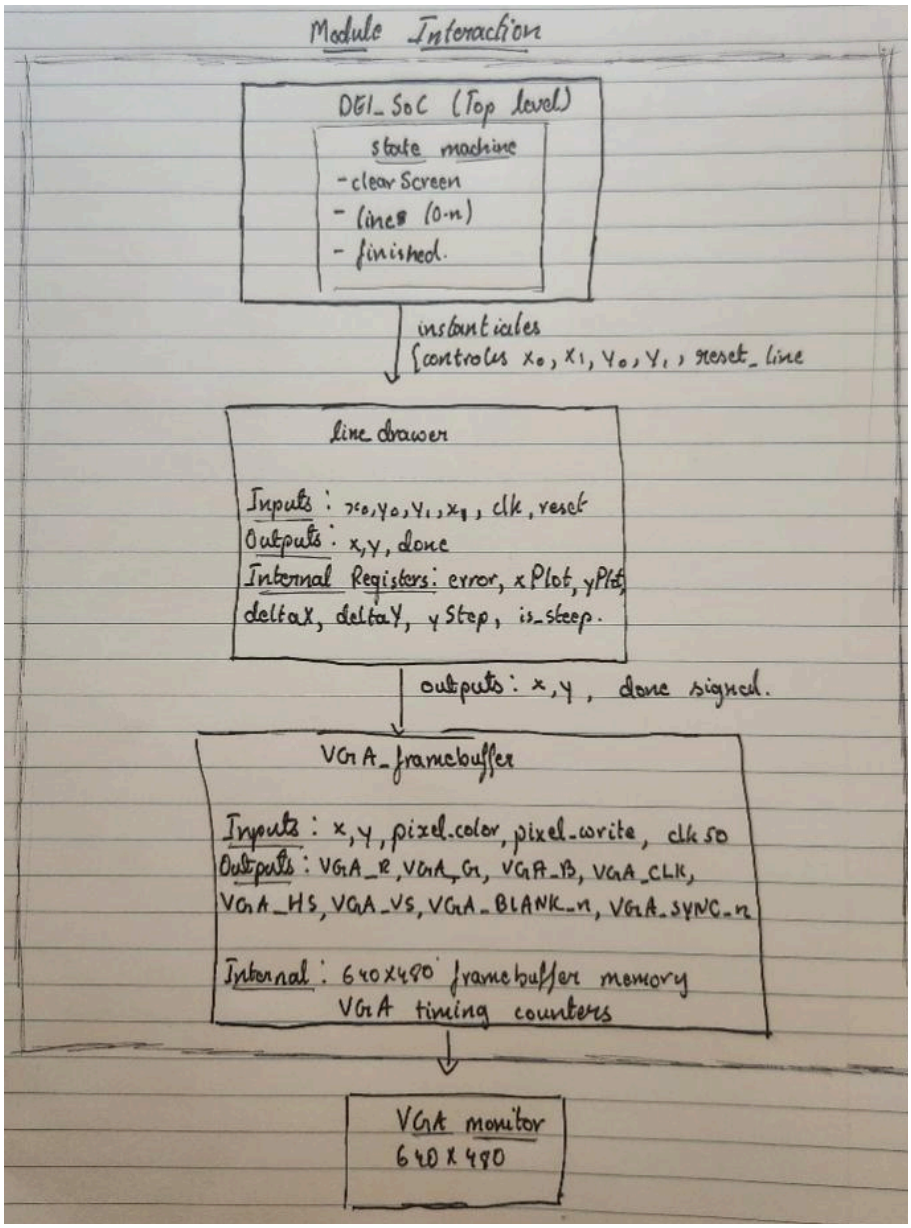


Figure 3: Module interactions

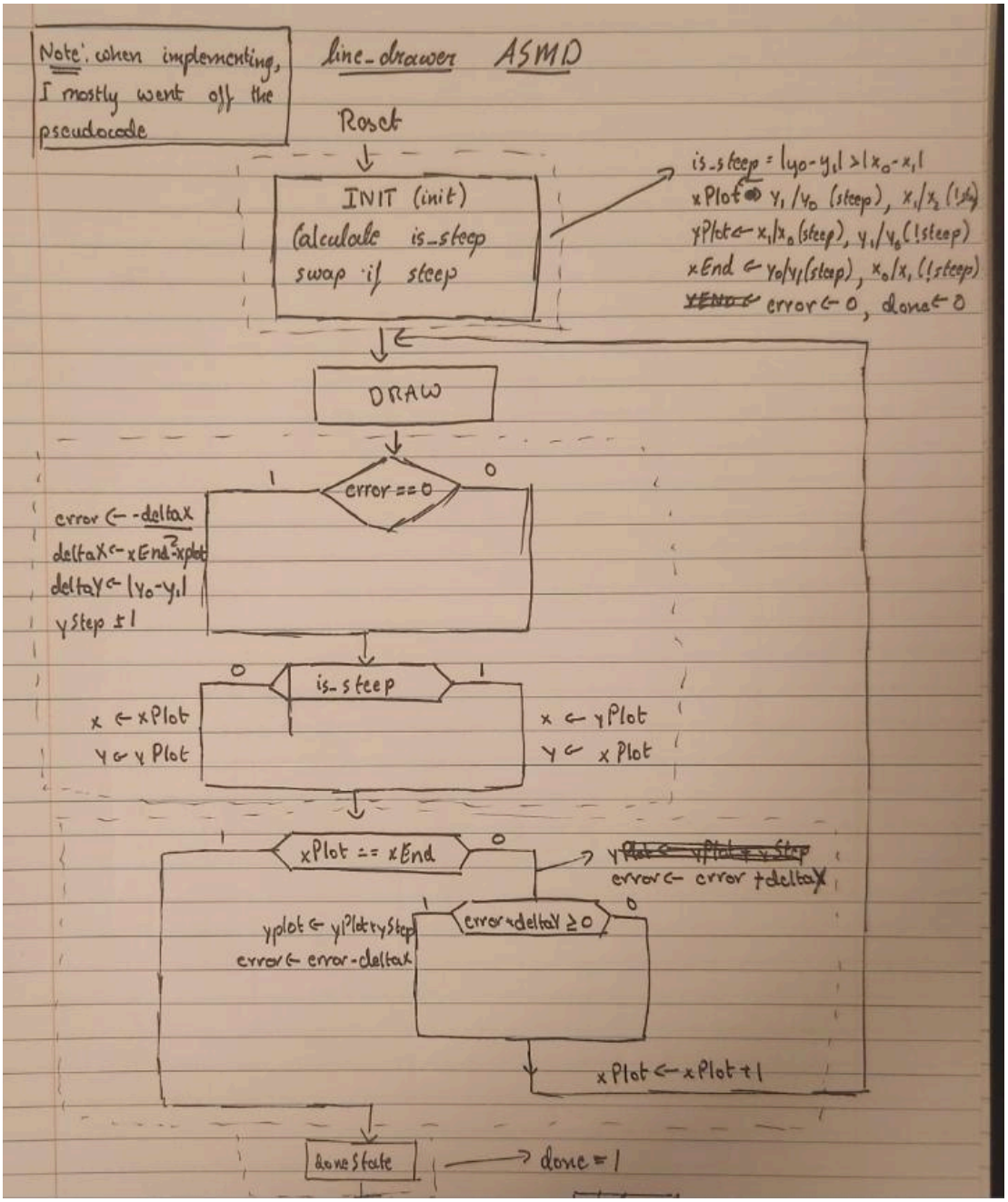


Figure 4: line drawer ASMD

Task 1

Task 1 establishes VGA connectivity and verifies pixel buffer functionality on the DE1_SoC FPGA. The starter code includes a VGA_framebuffer module that drives the VGA port and manages a 640x480 pixel buffer stored in memory. It takes a 50 MHz clock and pixel coordinate inputs (x, y) along with pixel color and write enable signals then outputs VGA signals (VGA_R, VGA_G, VGA_B, VGA_CLK, VGA_HS, VGA_VS) to the monitor. The framebuffer reads pixel data in raster scan order and sends it to display while allowing write operations to update pixel colors. The DE1_SoC top-level module instantiates the framebuffer and initially displays a rainbow gradient pattern that transitions to black confirming proper connection and pixel buffer operation. This task requires no code modifications and has no deliverables. We only needed to verify that the project compiles and produces the expected output on the VGA monitor on

LabsLand. The pixel coordinate system uses (0,0) as the top-left corner with x increasing rightward and y increasing downward.

Task 2

The line_drawer module implements Bresenham's line drawing algorithm to draw lines between 2 points in the VGA Display. It takes 2 coordinate pairs (x0,y0) and (x1,y1) as inputs and outputs pixel co-ordinates (x,y), one per clock cycle along with a done signal to show that it completed the line. The module uses internal registers to track the algorithm's state: error for error accumulation, xPlot and yPlot for current pixel position, xEnd for end point, deltaX and deltaY for the line's dimensions, yStep for the direction of y increment (positive or negative) and is_steep to indicate if coordinate swapping is needed.

On the rising edge of reset the module initializes by first determining if the line is steep by comparing absolute difference between x and y co ordinates. If the line is steep (absolute y difference greater than absolute x difference), it swaps the x and y coordinates so the algorithm always iterates along the longer axis. It then ensures the line is drawn left to right (or bottom to top for steep lines) by swapping the endpoints if necessary. This initialization also calculates deltaX and deltaY as the absolute differences between endpoints, sets yStep to either 1 or -1 depending on whether y increases or decreases and initializes error to zero.

After reset is deasserted, the first active cycle initializes the error register to the negative half of deltaX, which centers the error calculation for ideal line approximation. On each subsequent clock cycle it outputs the current pixel co-ordinates, swapping them back if is_steep is high to account for the initial co-ordinate swap. It updates the error by adding deltaY. If the error becomes non negative it increments/decrements yPlot by yStep and subtracts deltaX from error to maintain the error bound. xPlot increments by one each clock cycle advancing along the line until xPlot == xEnd when the done asserts.

Task 3

The Task 3 DE1_SoC module is the top model for the VGA drawing pipeline and acts as the control brain for a framebuffer, a Bresenham line generator module, line_drawer.sv from Task 2, and an FSM that animates the drawing of a dog figure on a virtual FPGA. Hexadecimal displays are disabled and every segment is set high. LEDR[9] is used a done signal when the drawing is complete. The design uses a 50 MHz clock, standard VGA outputs (the same as Task 1 and 2). The actual drawing on the screen happens through an instantiated VGA framebuffer module that accepts an x-y coordinate pair and a pixel color at every clock cycle. This top module uses two independent coordinate sources and provides this into the framebuffer module. The first source is a screen clearing mechanism that sweeps from pixel (0,0) to (639,479) column by column using nested counters and clears the display by writing every pixel to black. The second source is a line generator module that performs the Bresenham algorithm by stepping x,y positions for the currently selected line segment. A mux chooses what is fed to the frambuffer module between these two sources and grants priority to the clear engine until it is complete.

The drawing logic is driven by an FSM. When KEY[0] is pressed, the machine enters a clear or reset state and initializes the clearing coordinates protocol. It resets all of the drawing parameters and asserts a flag clear_active. This causes the framebuffer mux to select the clearing counters to feed to the framebuffer. On every clock tick clear_x (a coordinate) is incremented until the end of a row. It resets and then increments clear_y, eventually clearing every pixel. When the clearing finishes, the FSM transitions into a wait state where it listens for the press of KEY[1] (the start drawing key) to begin drawing. Upon this signal, the FSM loads the first part of the dog drawing, in the form of a line with endpoints which are in a case statement indexed by line_index. These coordinates feed the line_drawer module. The FSM asserts reset whenever it enters the draw state and the line drawer begins stepping from its start point toward its end point. The FSM remains in draw until the line drawer's done flag asserts and the complete line has been fed into the framebuffer.

Once a line finishes, the FSM enters a timed delay state where a counter expands time so the animation proceeds at a human visible speed, and not instantly. When the delay expires, the FSM increments line_index and gets the next line's coordinates. Then it reenters draw again. This repeats until all the defined lines are drawn. At that point the machine enters a finished state. Every line segment belonging in the dog drawing is encoded in the case statement: body, head, nose, eyes, ears, and collar and are expressed as a pair of endpoints. When the FSM loads a new index, the system routes the endpoints into the line_drawer module, which draws the pixel outputs directly into the framebuffer. Throughout this process the framebuffer_color is set high for drawing and zero for clearing. When the final line is drawn, the drawing remains visible on the VGA monitor because the framebuffer retains the written pixels. During this time the FSM is simply in the finished state having completed the animation sequence.

FSM logic summary: When the reset key is pressed, FMS immediately enters the clear state, resets coordinates, color, line index, and delay counter, and activates the clear engine to start clearing the screen pixel by pixel. While in clear, if the clear is active, it increments clear_x and clear_y until the entire screen is cleared, then moves to wait_start. In wait_start, FSM waits for start key press to begin drawing, initializing first line's coordinates, color, and resetting the line index. In draw state, it sets pixel color to active and waits for line drawer to signal done, moves to wait_delay. During wait_delay, counter runs to create visible pause between drawing lines if more lines remain, FSM updates coordinates for next line and returns to draw. Once all lines drawn, FSM enters finished state and remains there, holding the final drawn image until clear/reset is asserted via ~KEY[0].

Dog line definition summary: A coordinate table holds the line definition coordinates that make up the dog figure. Each entry stores four values. The table size is controlled by a parameter, we reasoned it shouldn't take more than 80 lines to draw our dog. The parameter tells synthesis how many entries exist and how far line_index can advance. When start is pressed, the FSM copies coordinates from first line table entry [0] into x0 y0 x1 y1. The line draw modules is called with these coordinates. Once done goes high, the FSM waits for a short delay so each stroke becomes visible. After that pause, FSM increments line_index to 1 and new index feeds new coordinates

from line_x0 line_y0 line_x1 line_y1 of index [1]. These values come arrays and a case structure tied to line_index. The FSM updates x0 y0 x1 y1 with fresh coordinates until line_index reaches eighty minus one. After the final stroke, the FSM stays in finished state and holds the final image.The low moves like this. Pick entry from table. Push coordinates into engine. Wait for done. Pause. Bump index + 1. Load next entry. Cycle eighty times. This creates full drawing sequence driven by parameter eighty.

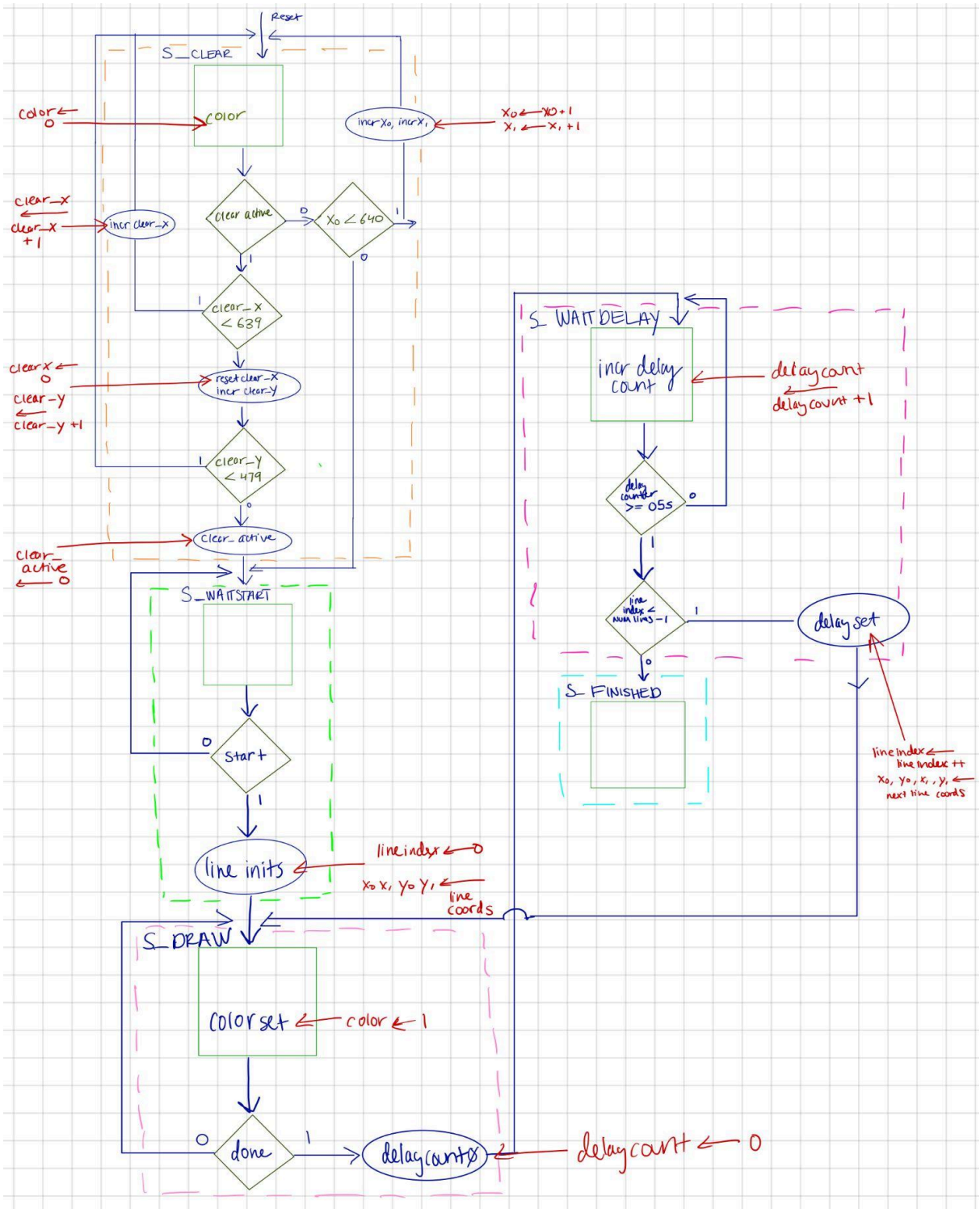


Figure 5: ASMD diagram of Task 3 Drawing FSM

Results and Testing

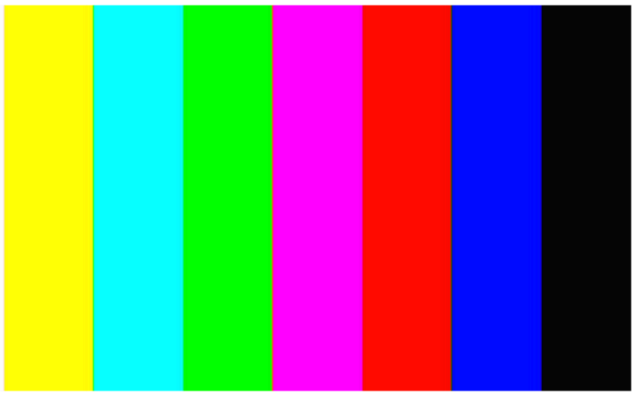


Figure 6: Task 1 output showing VGA connectivity

Task 1

No test bench was required for Task 1. Given that there were no deliverables, we just followed instructions and ensured it worked on the board (VGA initialization shown through a rainbow screen followed by a blank black screen).

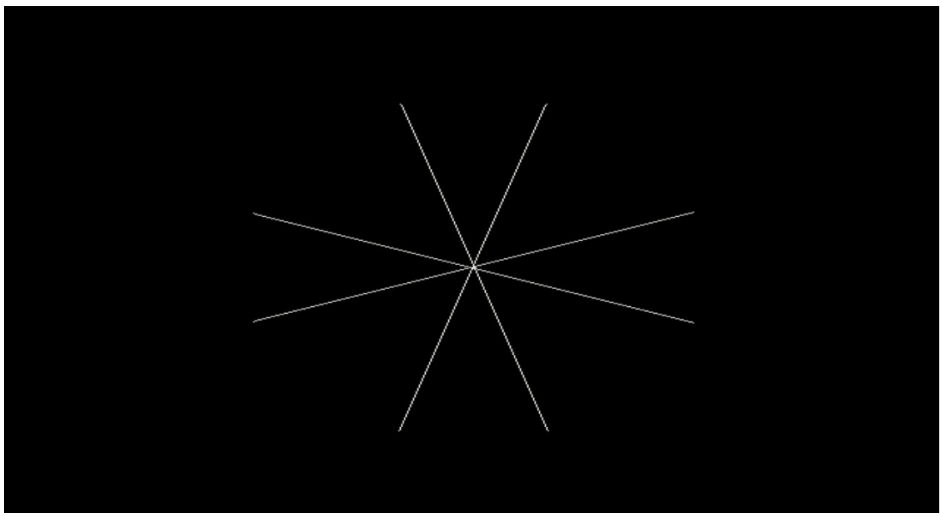


Figure 7: Screenshot showing the functionality of Task 2 drawing 8 lines of varying slopes in different directions

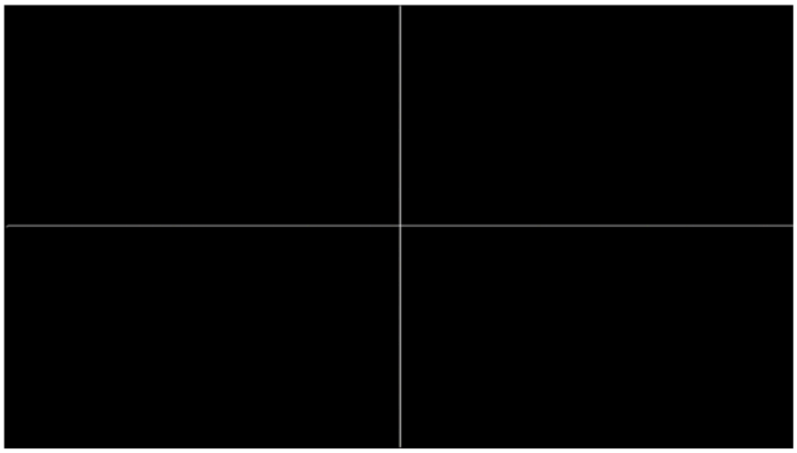


Figure 8: Screenshot showing drawing from Task 2 of horizontal and vertical line across screen

Task 2

This testbench tests the line_drawer module's implementation of the Bresenham's line drawing algorithm for all eight required line types. It verifies module correctly draws lines in all directions (left/right, up/down) and handles steep and gradual slopes. The testbench declares clk, reset, co-ordinate inputs (x0,x1,y0,y1) and co-ordinate outputs (x,y), and a done flag. The test sequence make 8 lines, each starting from the same point (320,240) and extending in the eight different directions: right-up gradual, right-up steep, left-up gradual, left-up steep, left-down gradual, left-down steep, right-down gradual, and right-down steep. For each test, reset is asserted for one clock cycle to initialize the line drawer's internal registers (error, deltaX, deltaY, yStep, is_steep), then deasserted to begin the drawing process. The testbench waits for the done signal to assert, indicating the line has been completely drawn, then pauses 100ns before starting the next test. This allows verification that x and y change by at most one pixel per clock cycle, that the algorithm correctly handles coordinate swapping for steep lines, and that error accumulation produces accurate line approximations without floating-point arithmetic.

We also made an initial DE1_SoC to test if all 8 lines were drawn, and the screenshot of that output is below. Once we verified the working, we moved on to creating a better drawing and thus editing our DE1_SoC!

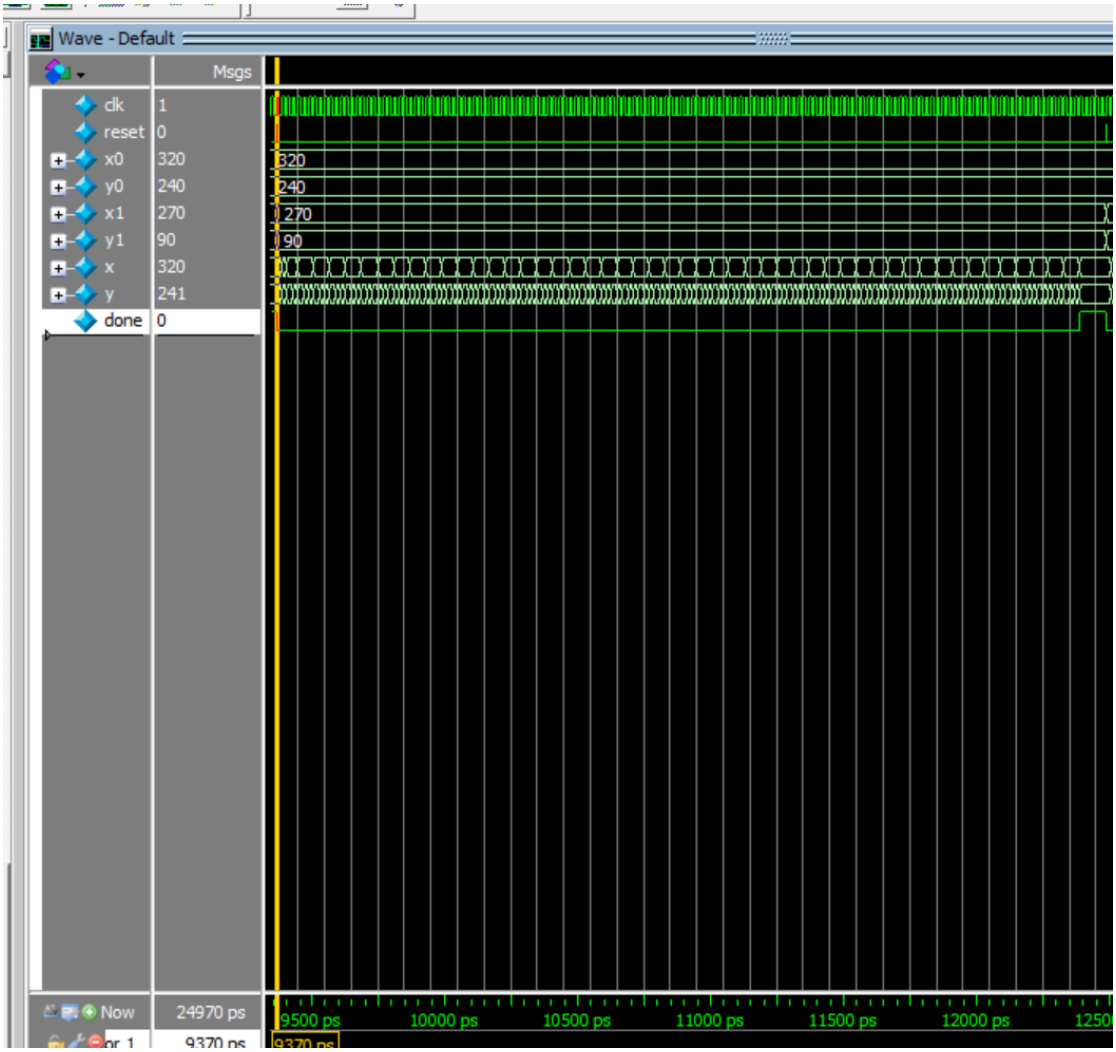


Figure 9: waveform of the task 2 line_drawer showing a full cycle of line drawing. Observe x and y showing multiple changes until done is high, showing the line drawing process.

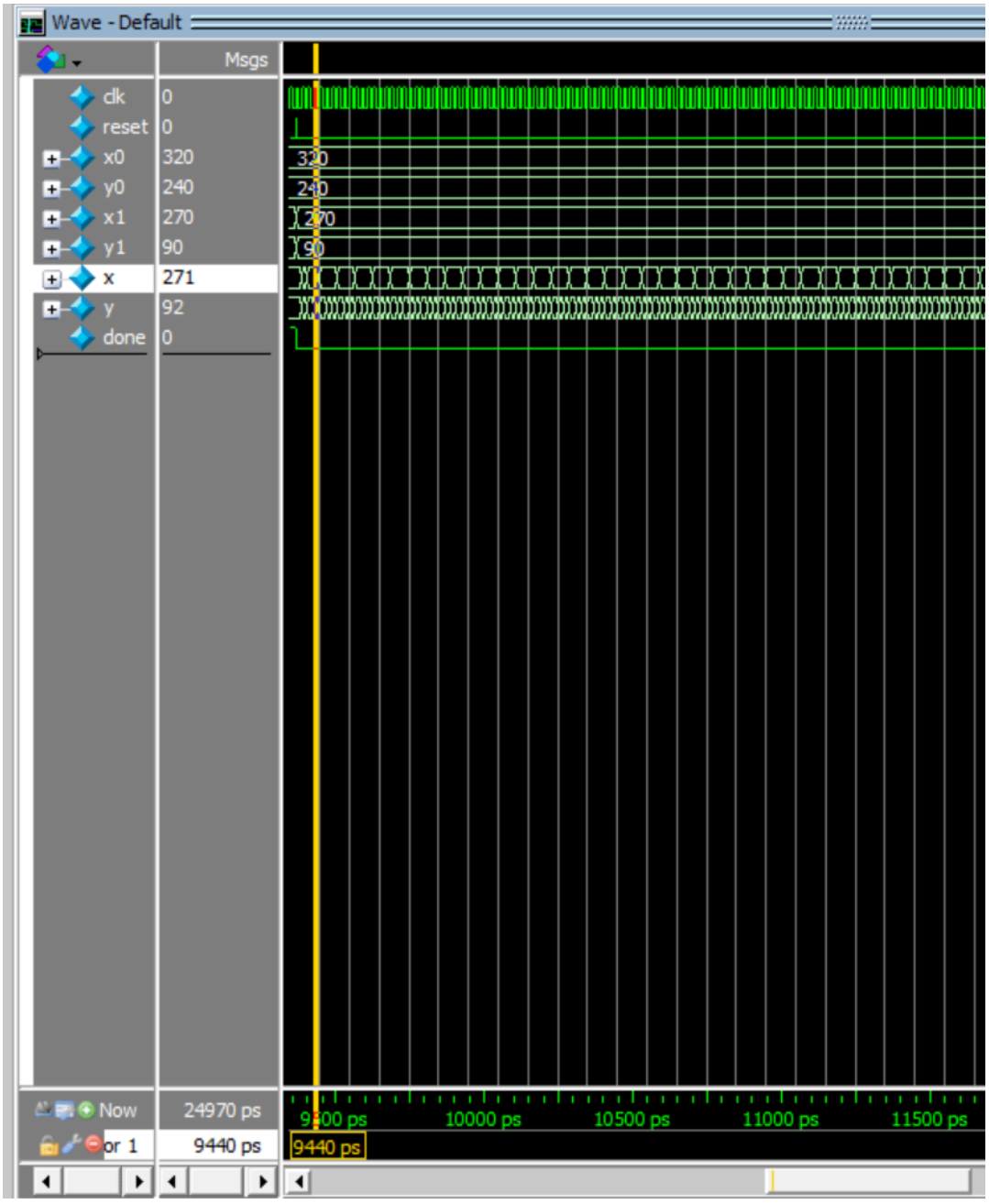


Figure 10: ModelSim Waveforms for Task 2 line_drawer showing the signal values for the same line as before, note that the value of x and y has changed, and x and y are closer to x1,y1 values, showing the swap due to a steep line.

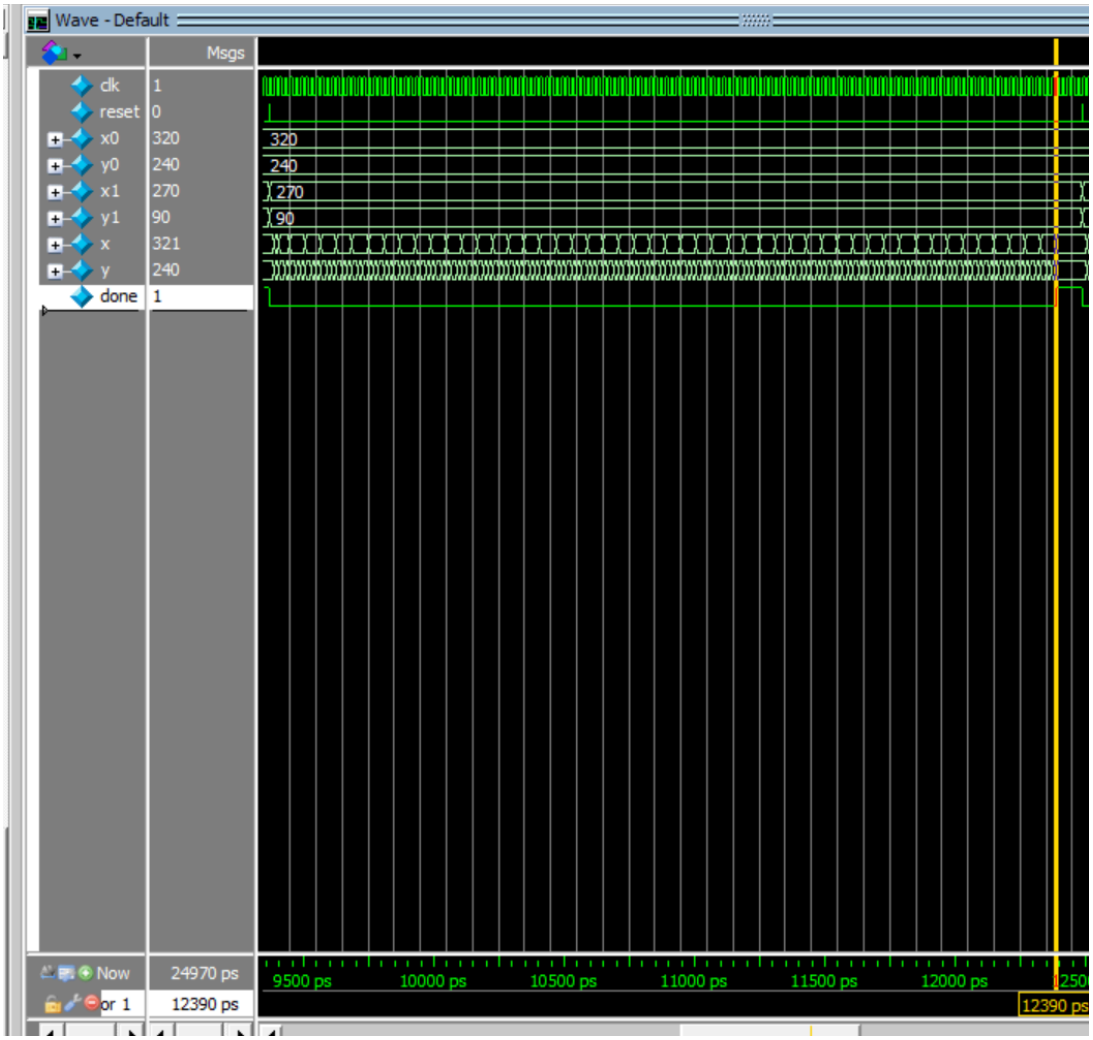


Figure 11: ModelSim waveform for the same line as before. Note that x and y are now similar to the x0 and y0 values, further corroborating the fact that a swap happened and that there was a gradual change in coordinate outputs for the line drawing. Also note that x and y stay const through the time done = 1, showing the state machine waits for reset to change the x1,y1 co-ordinates and start drawing the next line.

Now, screenshots of all 8 lines' simulations:

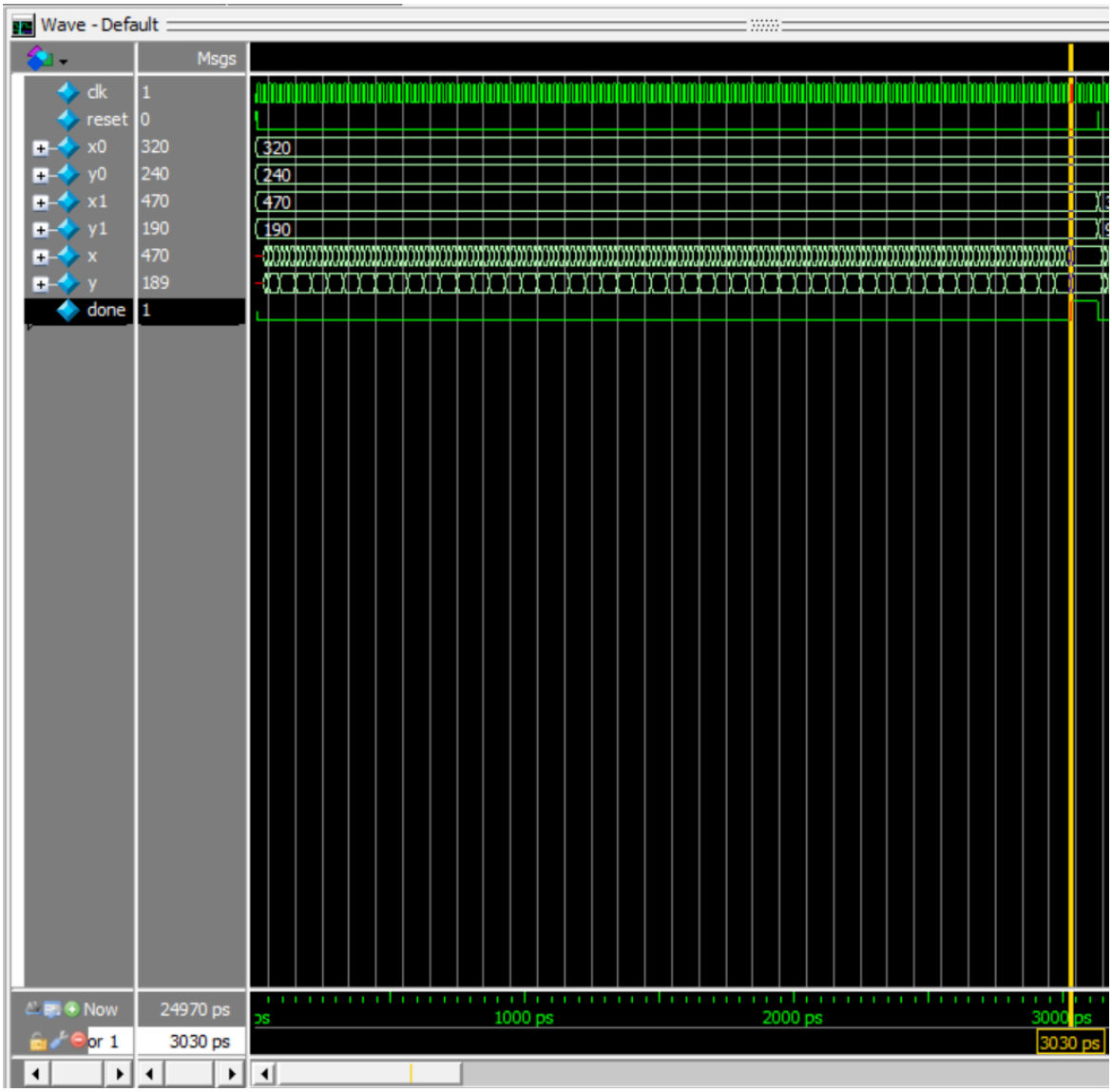


Figure 12: Line 1: Right-up gradual (320,240) to (470,190) Note that x,y are undefined in the beginning as they update after initialization of inputs.

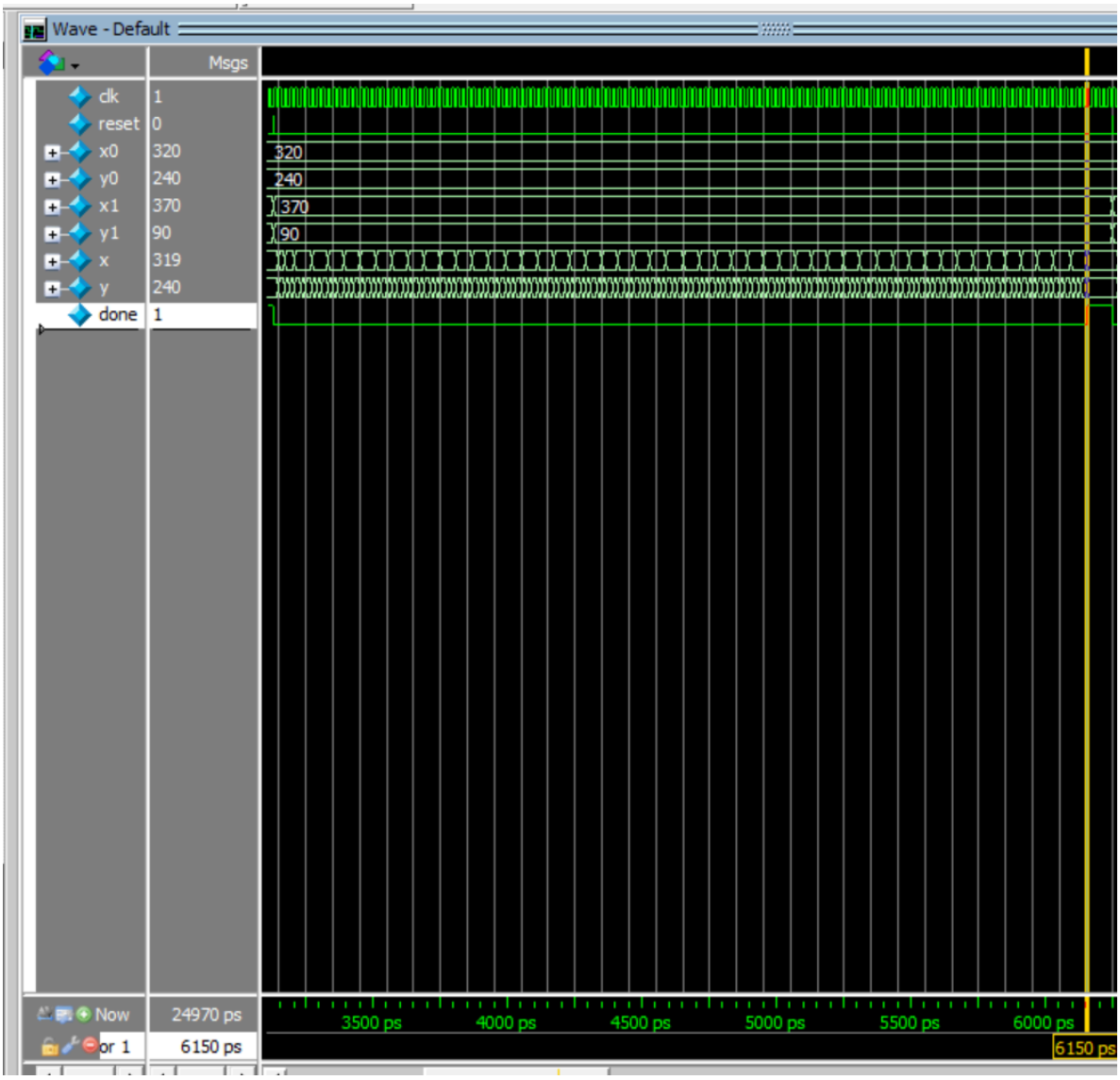


Figure 13: Line 2: Right-up steep (320,240) to (370,90)

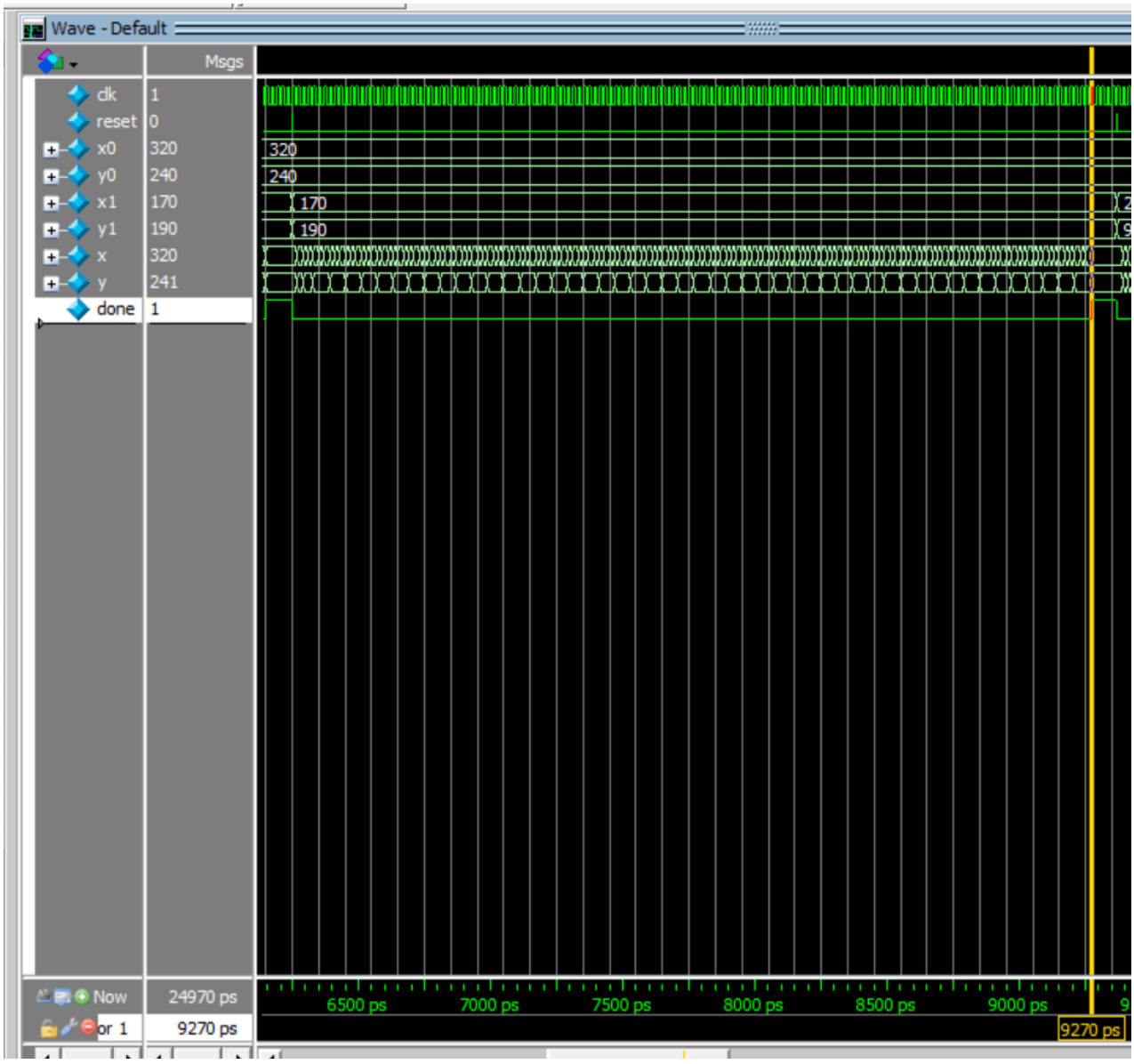


Figure 14: Line 3: Left-up gradual (320,240) to (170,190)

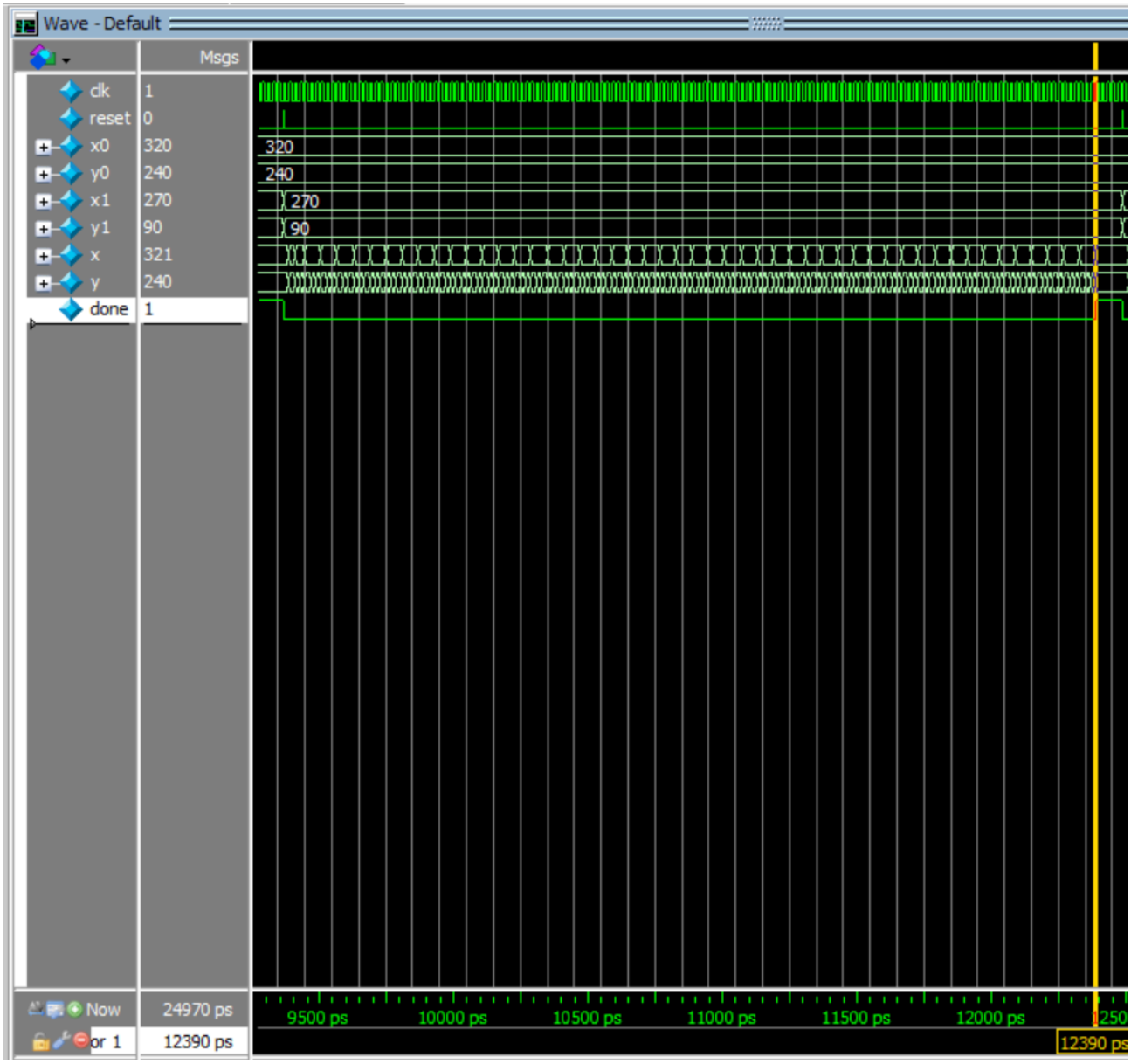


Figure 15: Line 4: Left-up steep (320,240) to (270,90)

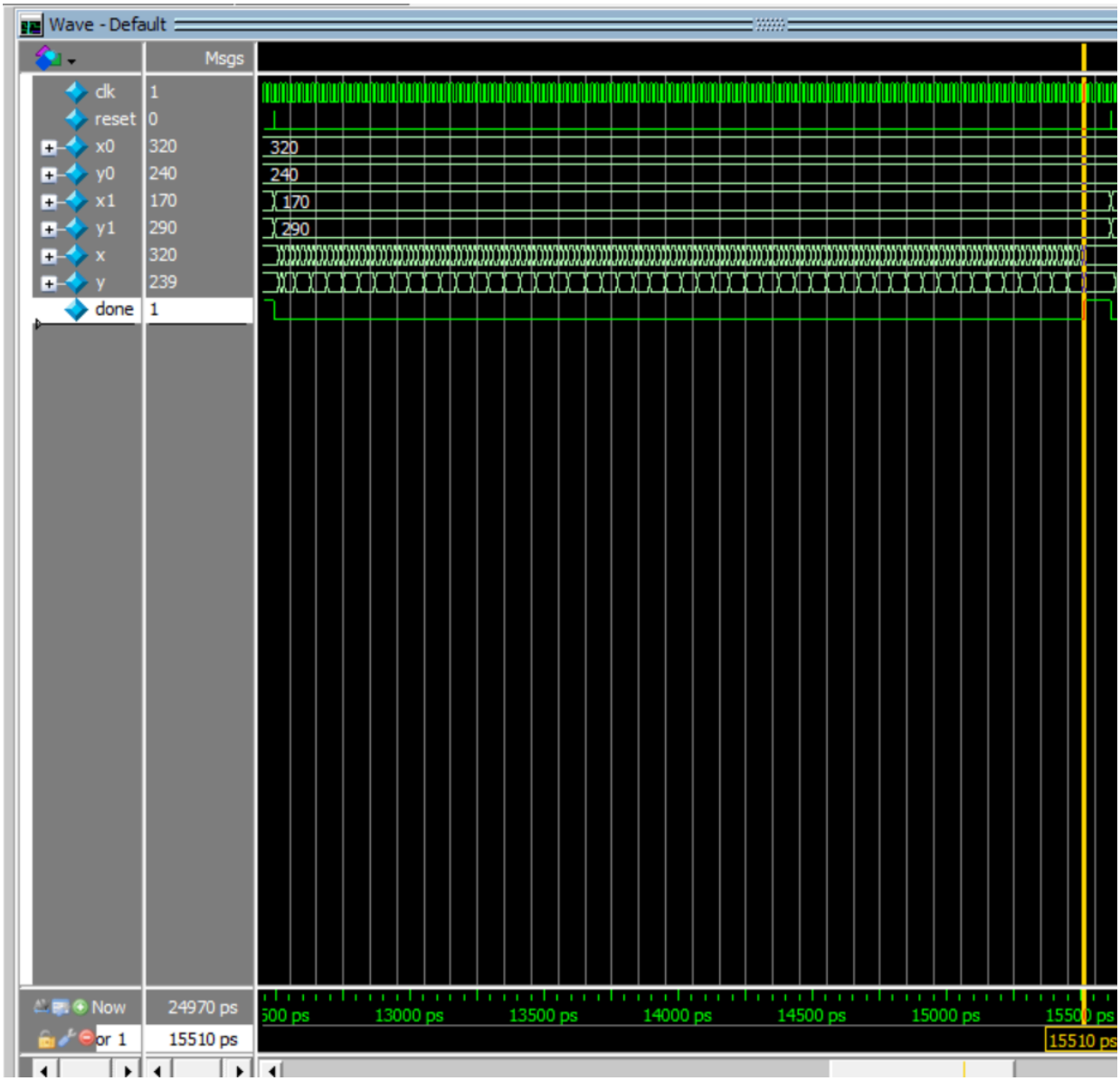


Figure 16: Line 5: Left-down gradual (320,240) to (170,290)

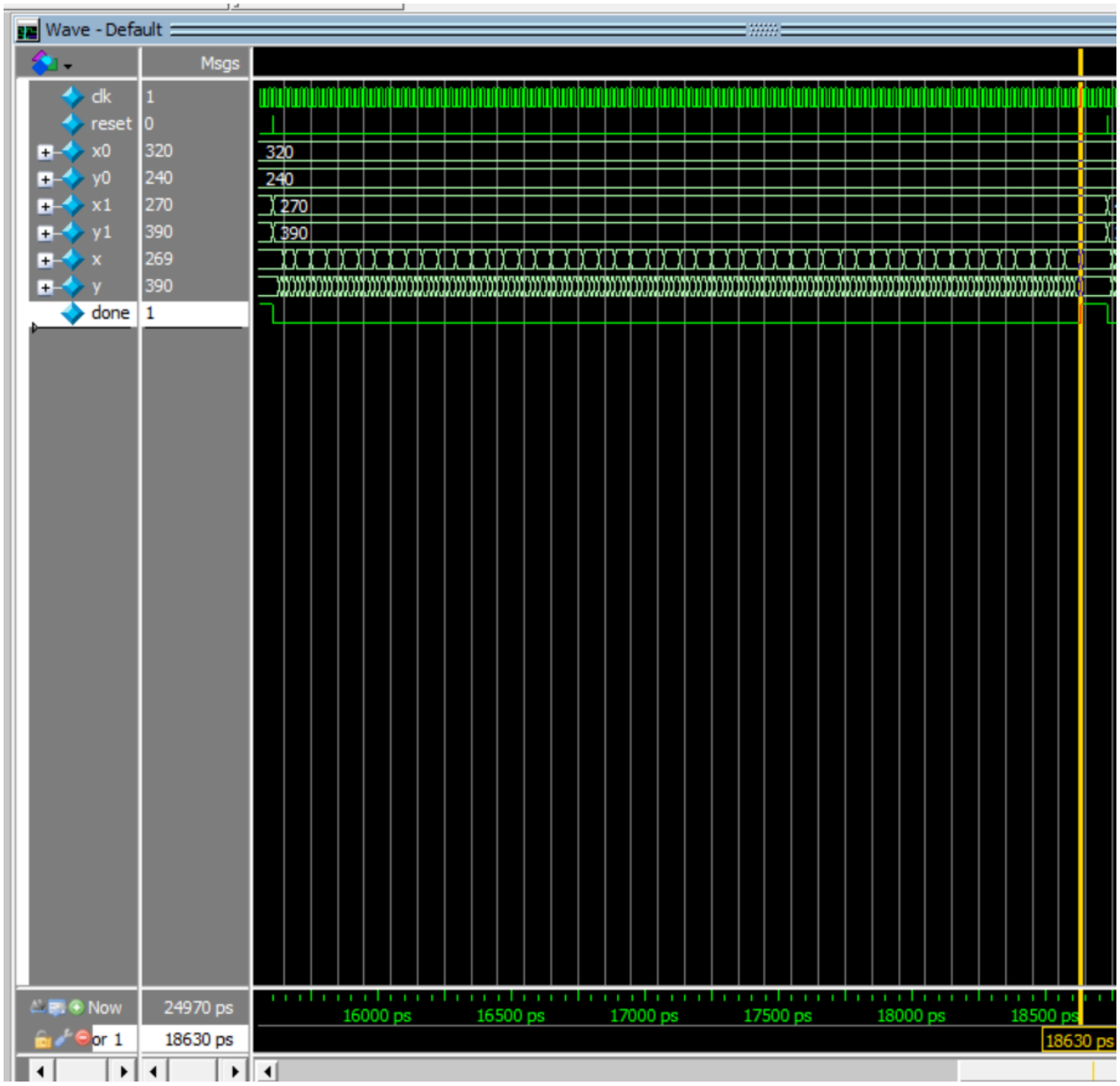


Figure 17: Line 6: Left-down steep (320,240) to (270,390)

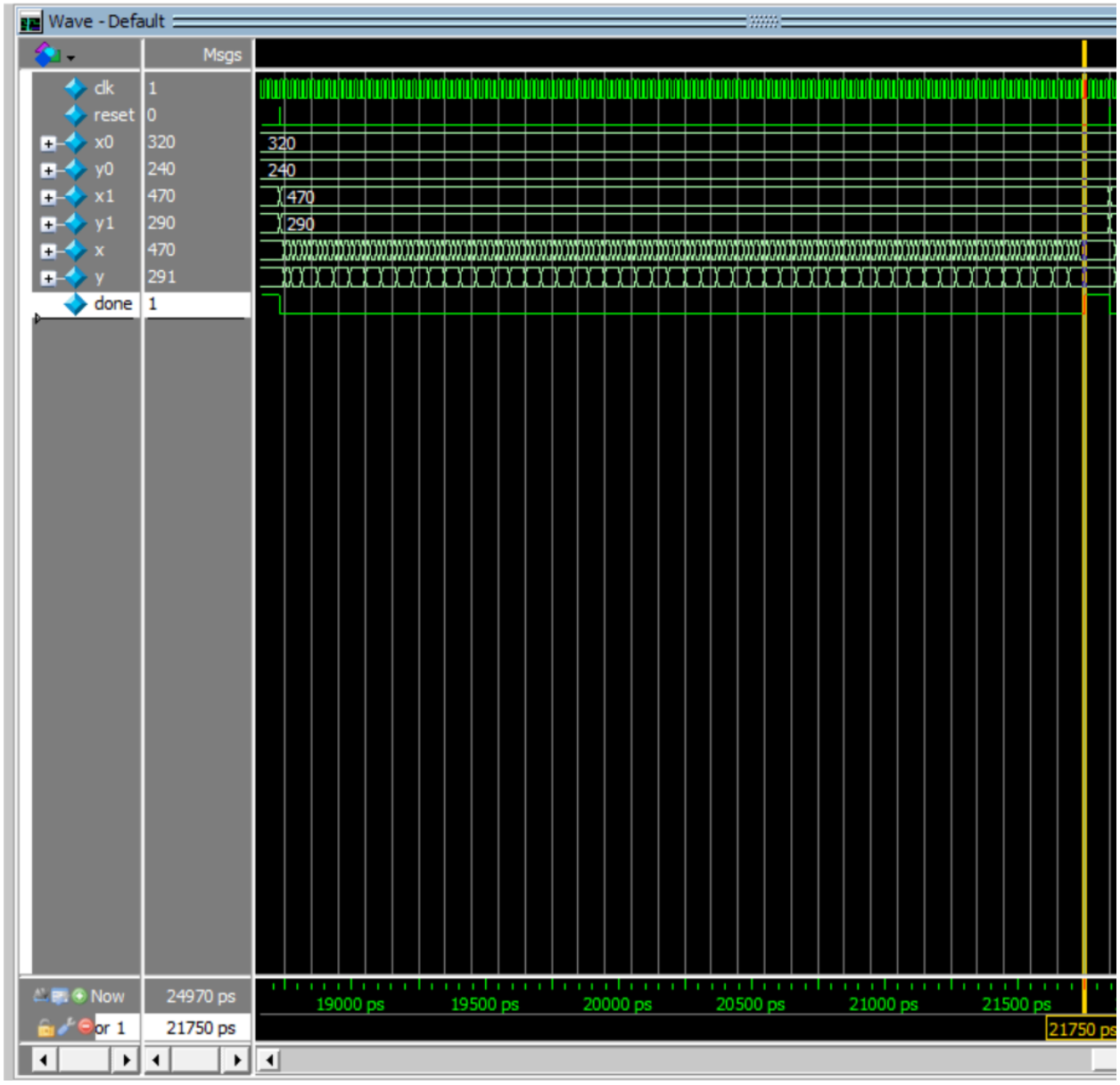


Figure 18: Line 7: Right-down gradual (320,240) to (470,290)

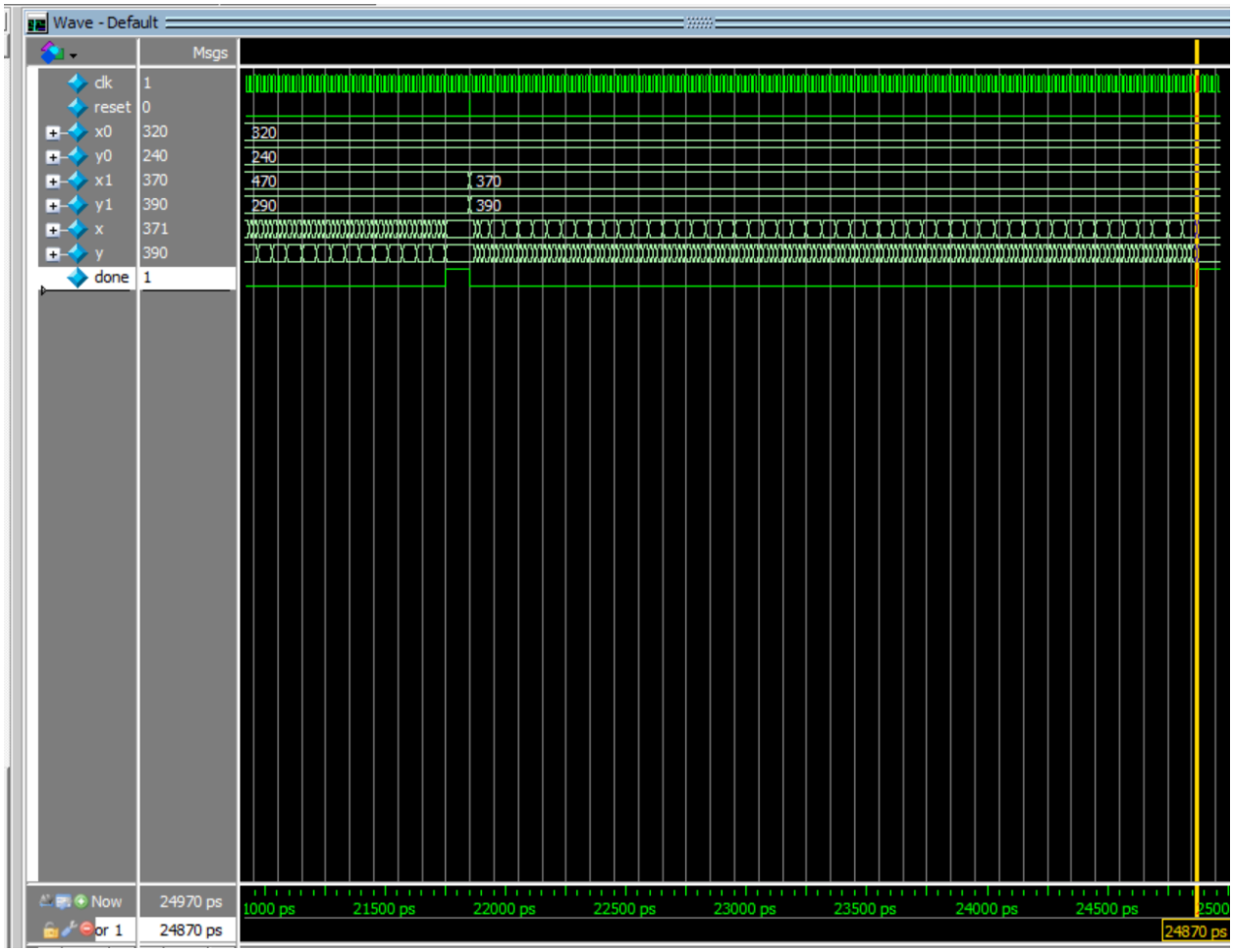


Figure 19: Line 8 : Right-down steep (320,240) to (370,390)

Task 3

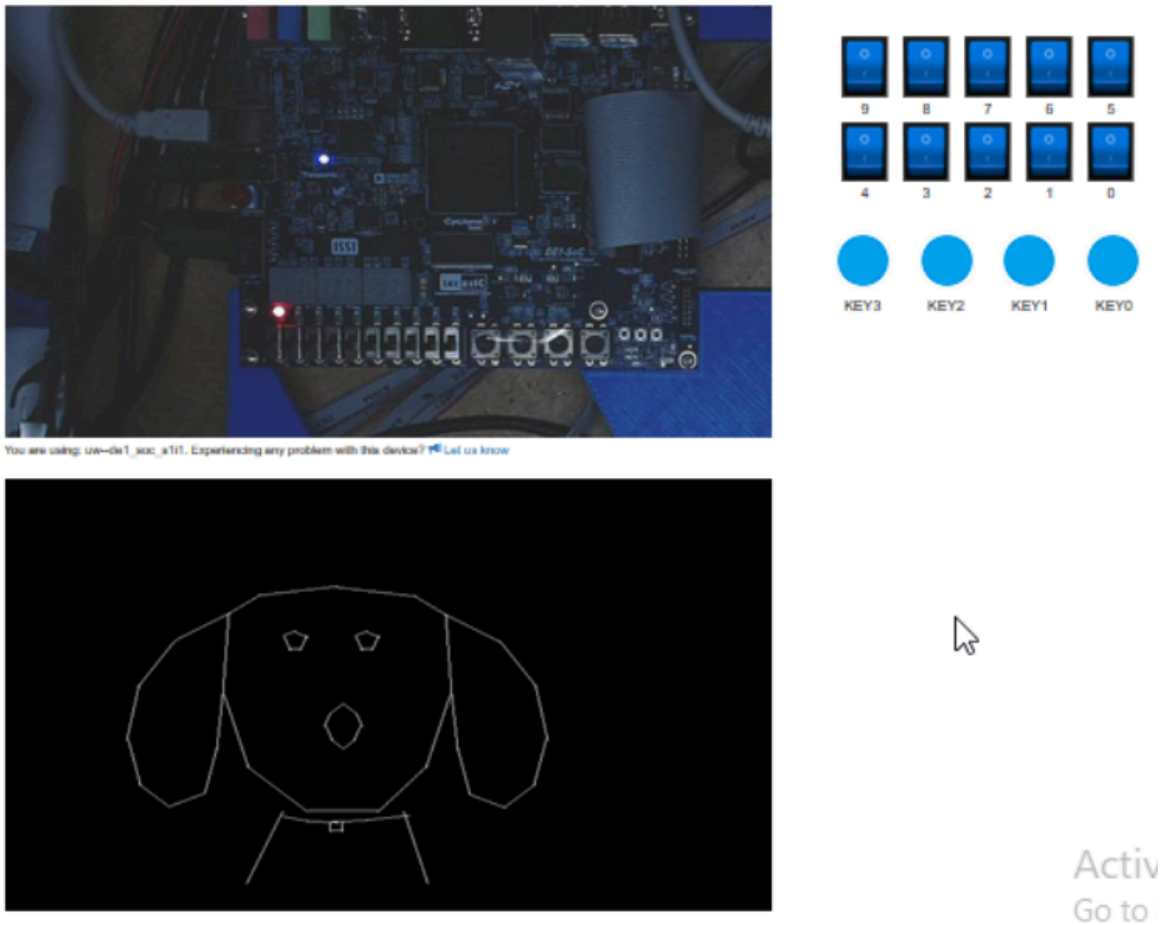


Figure 20: Showing completed drawing from Task 3, a Dog

The dog drawing started as a hand-sketch inside a simple drawing app where each feature of the dog was traced out as clean straight segments. Once the rough shape looked right every line was translated into pixel coordinates for the VGA. We choose start and end points while checking the drawing grid, converting them into x and y values in the 640x480 coordinate system, and then dropping those numbers into the case table feature by feature (starting with the dog body, head, right eye, left eye, etc). Nothing lined up intuitively on the first try. Many segments sat too high or dipped too low or the angles looked wrong. We had to adjust all coordinates, reload the design and watch the shape slowly evolve. After many rounds of trial and error, each line finally connected cleanly with its neighbors and the dog silhouette was clearly visible. .

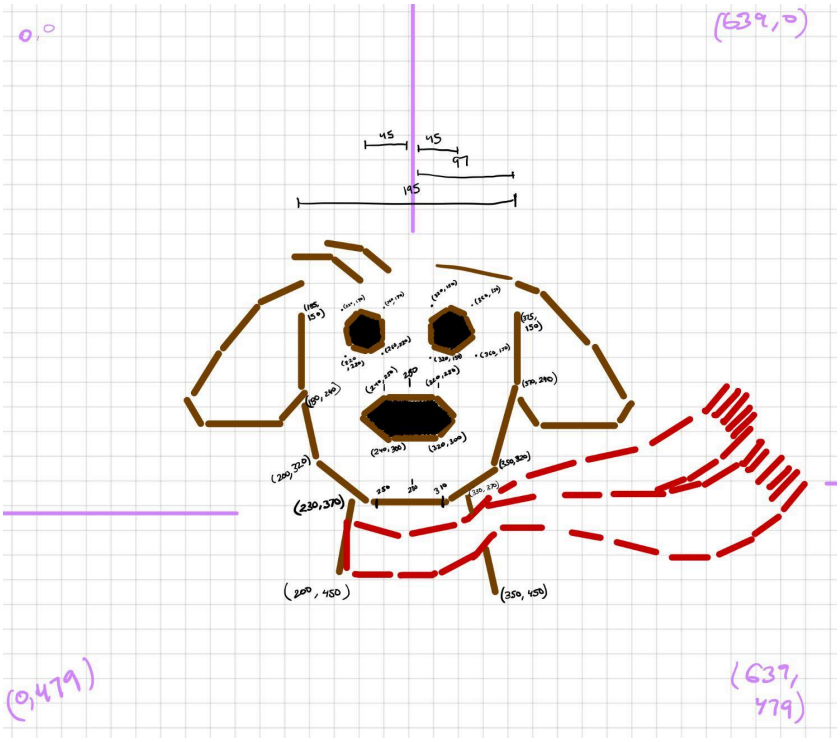


Figure 21: The starting dog sketch

The Task 3 testbench, DE1_SoC_tb_Task3, tests behavior controlled by the DE1 top module. Since we separately verified the line_drawer module in Task 2 and went through full validation, we did not need to retest the drawing algorithm. Thus we only wanted to confirm how this wrapper drives the helper modules. First, we tested reset to make sure the module properly resets. Then, the test bench checks to see whether the top level launches a draw operation at the correct moment and waits for a proper completion signal. Clear is tested to check the screen wipe feature activates when asked and returns control with a clean done pulse. Together these tests confirm control behavior.

Test 1 pulses reset to clear the screen, waits for the FSM to reach wait_start, then pulses start to begin drawing, waiting until FSM enters draw or finished. Test 2 pulses reset, waits for wait_start, starts drawing, lets the FSM run a few cycles, then pulses reset mid-draw to verify proper handling of resets during drawing. Test 3 pulses start key multiple times to ensure extra presses are ignored while drawing. At the end, testbench prints current FSM state, final line index, a completion message and it checks FSM transitions, button inputs handling, and reset behavior without simulating every pixel drawn. The reason we chose to not simulate every pixel in our testbench was because it caused crashing in ModelSim and took far too long for the simulation to perform. This is because we were waiting for the DE1_SoC FSM to complete real time drawing and the FSM uses a delay counter to slow down drawing so it can be visually observed. For example delay_1s counts tens of millions of clock cycles (26'd50_000_000 for 1 second at 50 MHz). The testbench was synchronous with the 50 MHz clock and waiting for the FSM to reach finished made the simulator to step through every single cycle of these large counters so every single draw sequence required tens of millions of simulation cycles which

slowed down the simulation and only ever made it through the first test. Thus we chose to not simulate every pixel.

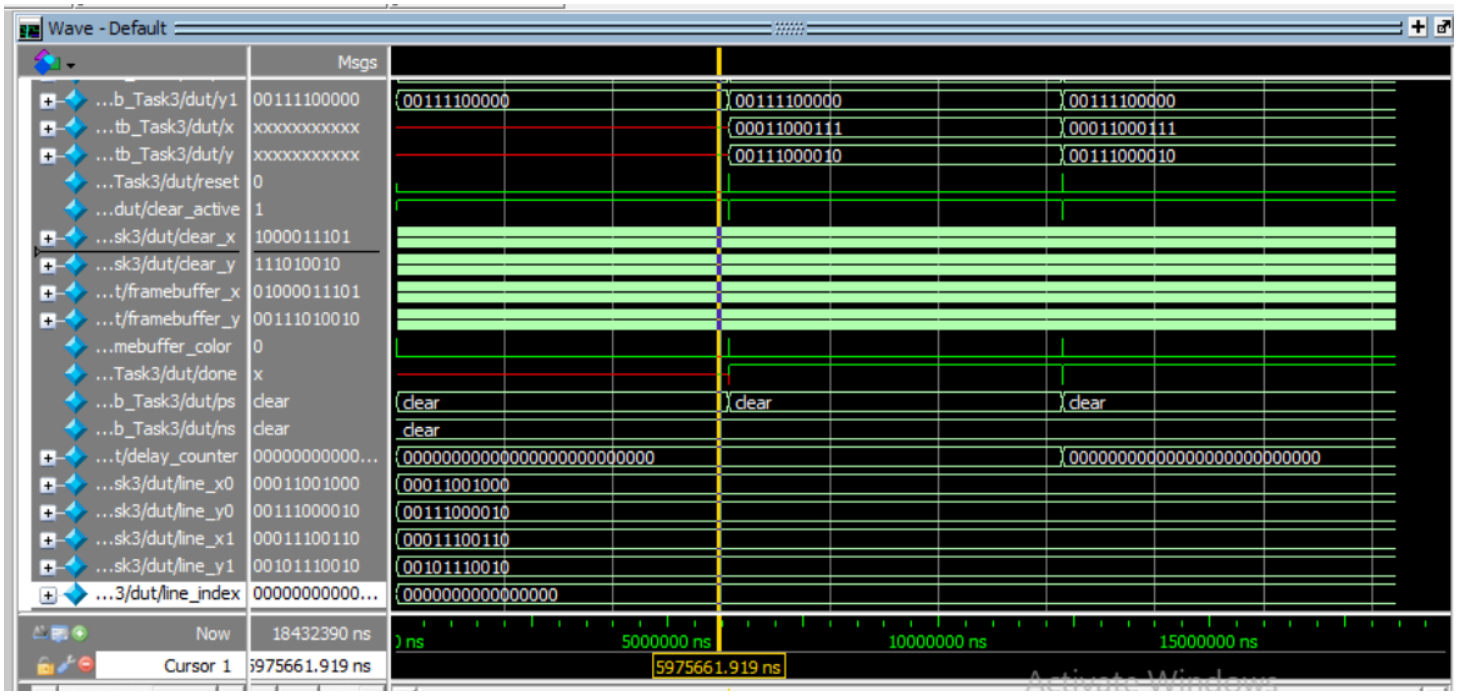


Figure 22: Waveform for simulation Test 1 showing correct behavior of draw and clear: done being asserted after drawing finishes and buffer color signal going high and then back to low upon clear. Clear state is our default state after reset.

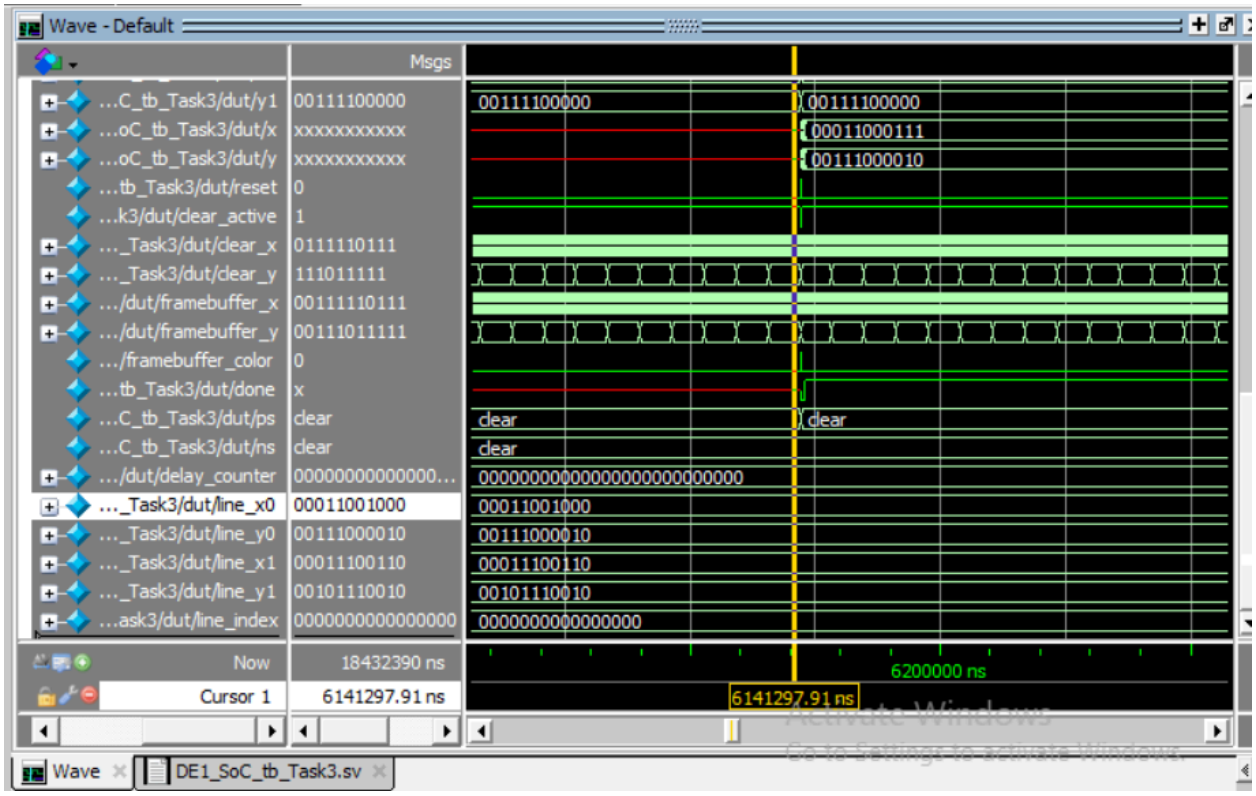


Figure 23: Waveform for simulation Test 1 showing reset and clear active asserted at the same time and present state switching back to clear. Clear state is our default state after reset.

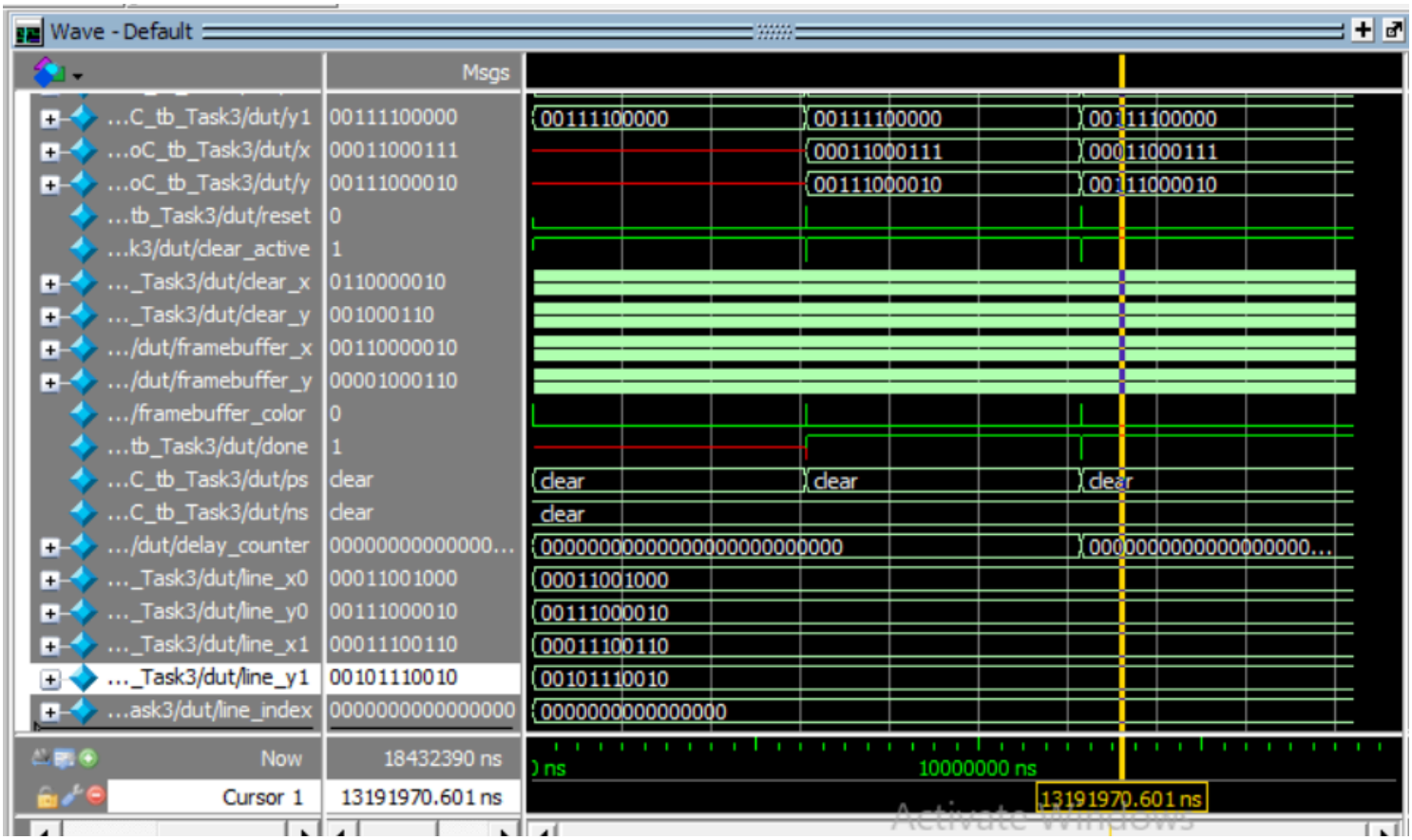


Figure 24: Waveform for simulation Test 3 showing correct behavior when start is asserted multiple times mid drawing: drawing continues and screen is not cleared. Clear state is our default state after reset.

```
# test 1: normal clear and draw
# clear complete
# drawing started or finished
#
# test 2: reset during drawing
# reset during draw successful
#
# test 3: multiple start presses
# multiple start presses done
#
# DUT current state: 3
# final line_index: 0
#
# === all tests complete ===
# ** Note: $stop    : C:/Users/aecam/
#    Time: 18432390 ns  Iteration: 1
```

Figure 25: Display statements showing testbench tests were successful

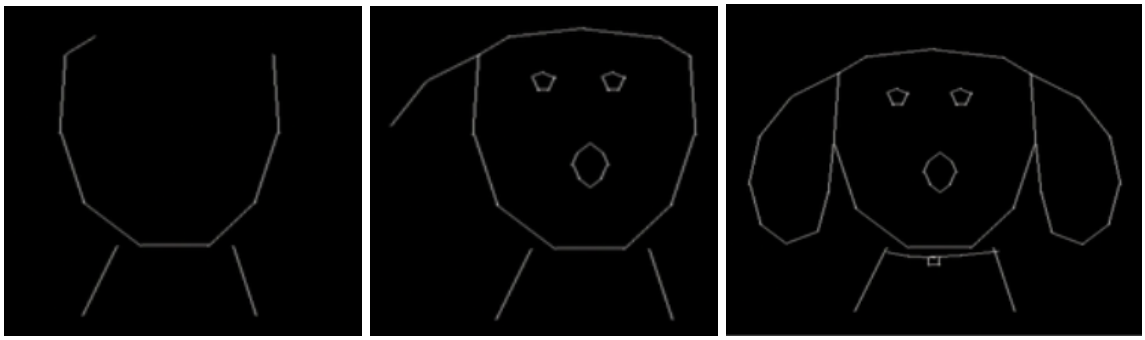


Figure 26: Stages of dog drawing animation on VGA

Flow Summary

Flow Summary

<<Filter>>

Flow Status	Successful - Sun Nov 16 11:32:17 2025
Quartus Prime Version	17.0.0 Build 595 04/25/2017 SJ Lite Editio
Revision Name	DE1_SoC
Top-level Entity Name	line_drawer
Family	Cyclone V
Device	5CSEMA5F31C6
Timing Models	Final
Logic utilization (in ALMs)	N/A
Total registers	92
Total pins	69
Total virtual pins	0
Total block memory bits	0
Total DSP Blocks	0
Total HSSI RX PCSs	0
Total HSSI PMA RX Deserializers	0
Total HSSI TX PCSs	0
Total HSSI PMA TX Serializers	0
Total PLLs	0
Total DLLs	0

Flow Summary

<<Filter>>

Flow Status	Successful - Sun Nov 16 16:06:26 2025
Quartus Prime Version	17.0.0 Build 595 04/25/2017 SJ Lite Edition
Revision Name	DE1_SoC
Top-level Entity Name	DE1_SoC
Family	Cyclone V
Device	5CGXFC7C7F23C8
Timing Models	Final
Logic utilization (in ALMs)	472 / 56,480 (< 1 %)
Total registers	255
Total pins	96 / 268 (36 %)
Total virtual pins	0
Total block memory bits	307,200 / 7,024,640 (4 %)
Total DSP Blocks	0 / 156 (0 %)
Total HSSI RX PCSs	0 / 6 (0 %)
Total HSSI PMA RX Deserializers	0 / 6 (0 %)
Total HSSI TX PCSs	0 / 6 (0 %)
Total HSSI PMA TX Serializers	0 / 6 (0 %)
Total PLLs	0 / 13 (0 %)
Total DLLs	0 / 4 (0 %)

Flow Summaries for Task 2 Line_drawer and Task 3 DE1 Top level