# Introduction:

## Project Description

This project involved the design of a number of interfaces to multiple peripheral devices. It drives a digital-to-analog converter (DAC) using a serial peripheral interface (SPI), and it also drives a video graphics array (VGA) display monitor. This project also utilized the Xilinx Core Generator and other existing IP to complete the final design.

## Project Requirements

The project was divided into four major parts as defined below:

* Preliminary Part: Four-digit seven-segment display
* Part 1: Creating VGA Display using an existing VGA controller
* Part 2: Controlling a moving block on the VGA Display
* Part 3: Using a DAC to generator a sine wave

## Outline

This report will discuss the requirements, system design, and implementation details, and test results of the each of the aforementioned sections in detail. Followed by a conclusion that discusses the problems faced while completing this project, in addition to the solutions to these problems. Further suggestions for improvement will also be discussed.

# Four-Digit Seven-Segment Display:

This section of the project required the modification of a seven-segment display module that was created in a tutorial prior to this project. The original display had the capability to display a single digit on the seven-segment display. The modified module was required to be able to display the digits “0000” to “FFFF” on the four seven-segment displays.

## System Requirements

1. The module should have a 16-bit input bus
2. The module should display the digits between “0000” to “FFFF”
3. The module should be a stand-alone entity

## System Design

The design for this module is shown in the diagram below.

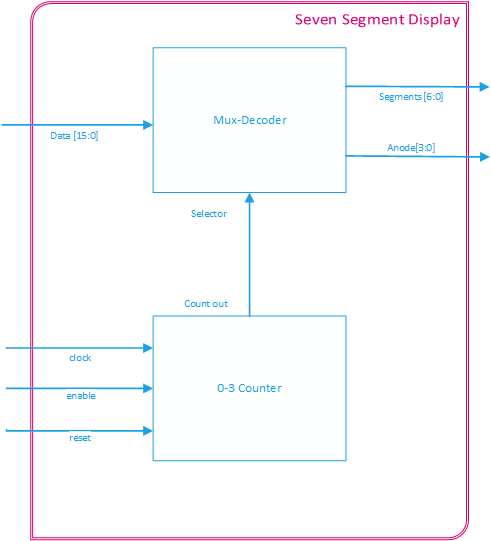


Figure : Seven Segment Display Module System Design

## Implementation Details

The module accepts a 16-bit line as an input which is sent to a decoder and multiplexer component. The decoder splits the 16-bit signal into four 4-bit sub-signals. Each of the 4-bit sub-signals is then decoded in order to represent a hexadecimal digit to be displayed on one of the seven-segment anodes, as shown in the code sample below.

-- Splits the 16-bit bus into 4 4-bit parts.

-- Each part will serve at the number representation for an anode

-- AN3 AN2 AN1 AN0

alias data\_3 : std\_logic\_vector(3 downto 0) is data (15 downto 12);

alias data\_2 : std\_logic\_vector(3 downto 0) is data (11 downto 8 );

alias data\_1 : std\_logic\_vector(3 downto 0) is data ( 7 downto 4 );

alias data\_0 : std\_logic\_vector(3 downto 0) is data ( 3 downto 0 );

Code Sample : Seven Segment input splitter

-- Decodes the data provided by the mux

process(data\_to\_seg)

begin

case data\_to\_seg is

when "0000" => segLED <= zero;

when "0001" => segLED <= one;

when "0010" => segLED <= two;

when "0011" => segLED <= three;

when "0100" => segLED <= four;

when "0101" => segLED <= five;

when "0110" => segLED <= six;

when "0111" => segLED <= seven;

when "1000" => segLED <= eight;

when "1001" => segLED <= nine;

when "1010" => segLED <= a;

when "1011" => segLED <= b;

when "1100" => segLED <= c;

when "1101" => segLED <= d;

when "1110" => segLED <= e;

when "1111" => segLED <= f;

when others => segLED <= off;

end case;

end process;

Code Sample : Seven Segment Decoder

Additionally, a 2-bit counter was connected to the select signal in order to cycle through the four anodes in the multiplexer component. Depending on the value on the select line, the corresponding anode would be enabled, and its respective digit would be displayed. The sample code for this functionality is shown below.

-- Selects the appropiate input data bus and

-- turns anode corresponding anode on

process(mux\_in, data\_3, data\_2, data\_1)

begin

case mux\_in is

when "11" =>

data\_to\_seg <= data\_3;

anode <= "0111";

when "10" =>

data\_to\_seg <= data\_2;

anode <= "1011";

when "01" =>

data\_to\_seg <= data\_1;

anode <= "1101";

when others =>

data\_to\_seg <= data\_0;

anode <= "1110";

end case;

end process;

Code Sample : Seven Segment Anode Mux

## Test Results

A test bench was created to test the functionality of the module. The following waveforms verify that the module performed as expected.



Figure : Seven Segment Test Bench Results

As shown in the diagram above the segled bus sets the bits low for the LEDs that correspond with the current anode and data signals. As an example, when the mux\_in is 1 (in decimal), anode[1] is enabled and the digit on data\_1 (which is 3 in hexadecimal) is represented on the segled bus by driving all the cathodes low except for cathode ‘e’ and cathode ‘f’.

# Creating VGA Display using an existing VGA controller:

This section of the project required the design of a module that could create a VGA display by using an existing VGA controller provided by Digilent. The display was required to be able to change colors and images depending on specific inputs from the user. Additionally, a digital clock manager (DCM) was required to be created to drive the VGA controller.

## System Requirements

VGA Display Module

1. The module should use the VGA controller provided by Digilent
2. The display should have a resolution of 640 x 480 pixels
3. The module should have inputs to display the following patterns
   1. Completely blue display
   2. Completely blue display
   3. A black screen with a white rectangle 10 pixels in from the outside edges (one pixel wide)

DCM

1. The DCM should produce a 25MHz clock signal
2. The 25MHz clock signal should drive the VGA controller

## System Design

The top-level design for this section is shown in the diagram below.

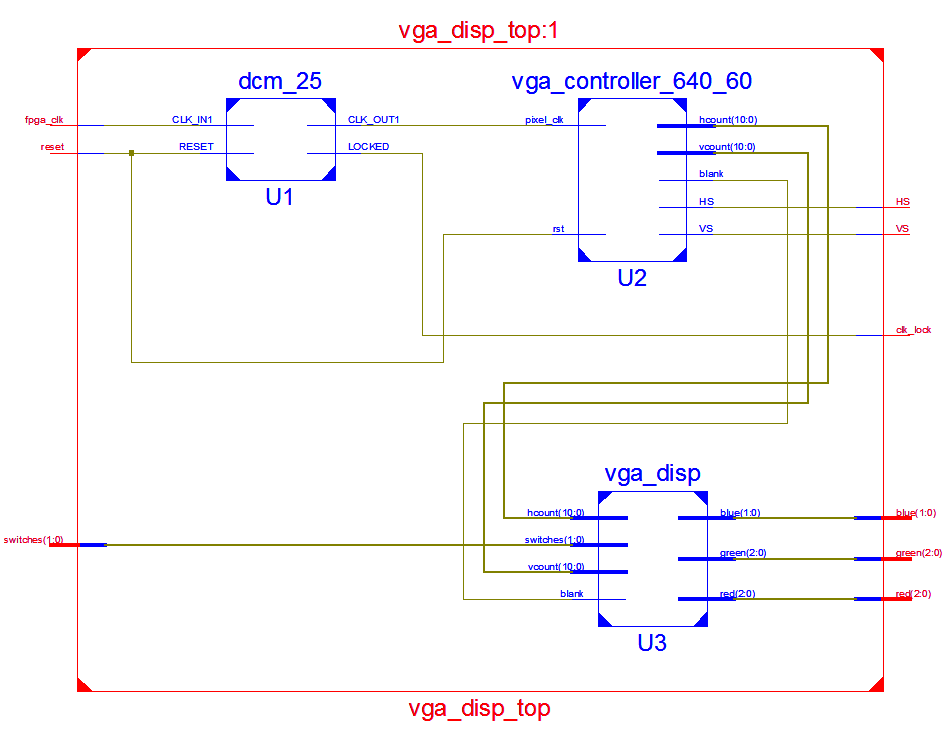


Figure : VGA Display Top-Level Design

The DCM converts the clock signal provided by the FGPA board into a 25 MHz signal that is used by the VGA controller. The VGA controller provides the horizontal sync (HS) and vertical sync (VS) outputs to an external monitor, the blank, horizontal count, and vertical count outputs are used as inputs to the VGA display module. The VGA display module outputs the corresponding red, green, and blue color signals to an external monitor based on the horizontal and vertical count inputs, and switch inputs provided.

## Implementation Details

The VGA display was the only module in this section that had to be designed, as mentioned before. The module was implemented as a purely combinational design. The switch signal has only 4 states, three of the states were used to display the colors specified in the requirements, and the unused state displayed a black screen. It is also important to note that a monitor will not display an image when the blank state is on. The code sample below shows this implementation using a case statement.

-- Changes the screen to be either blue or green

-- depending on the switch input

process (switches, blank, hcount, vcount)

begin

if blank ='0' then

case switches is

-- Make the screen blue

when BLUE\_C => BLUE; -- Make the screen green

when GREEN\_C => GREEN;

-- Draw a white rectangle on the screen

when RECT\_C => -- Omitted for clarity

-- if not blue or green, or a rectangle,

-- make the screen black.

when others => BLACK;

end case; -- End the switch-screen case

end process;

Code Sample : VGA Display Case Statement

To implement the white rectangle display, the boarders of the rectangle had to be determined, additional logic was needed to accomplish this. The following flowchart illustrates how the boarders were formed.

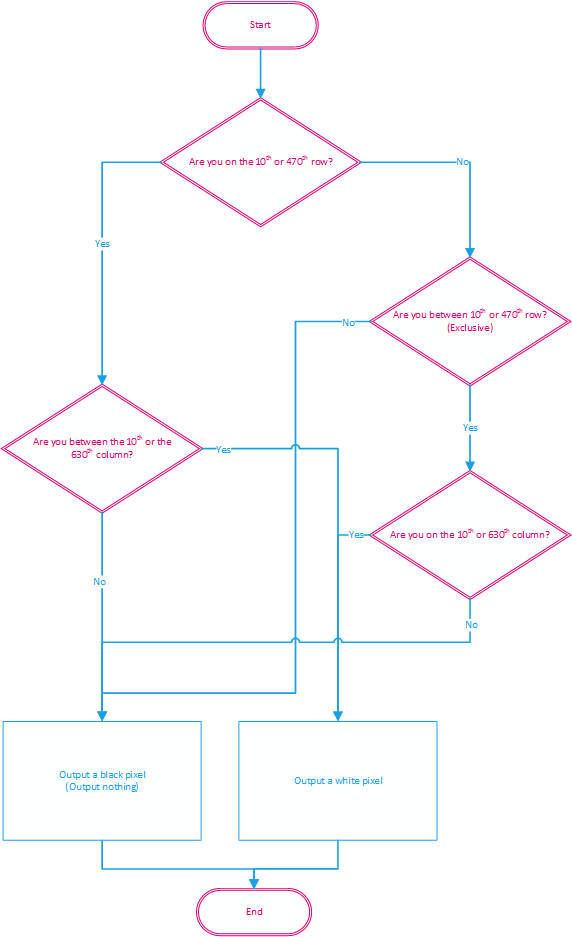


Figure : White Rectangle Decision Chart

The code used to implement this is shown below.

-- Check if vcount is on 9th row or the 470th row --

if (vcount = Y1) OR (vcount = Y2) then

-- Draw a white horizontal line starting from the 10px

if (hcount >= X1) AND (hcount <= X2) then <=

px <= WHITE;

else

px <= BLACK;

end if;

-- If vcount\_sig is between the 10th row and the 470th row

elsif (vcount >= Y1) AND (vcount < Y2) then

-- Draw white pixel only on 9th coloumn and 630th column --

if (hcount = X1) OR (hcount = X2) then

px <= WHITE;

else

px <= BLACK;

end if;

-- It's not on the perimeter of the rectangle, set it to black

else

px <= BLACK;

end if; -- END DRAW RECTANGLE IF

Code Sample : White Rectangle Drawer

## Test Results

The VGA display was tested on the FPGA board and as shown in the pictures below, the display changed depending on the input given.

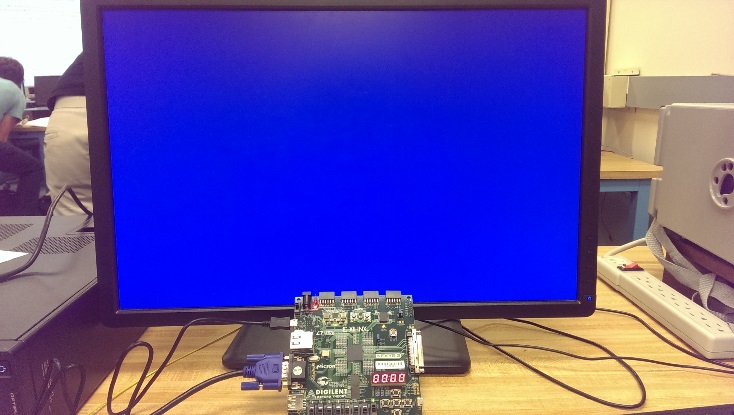


Figure : Blue VGA Display

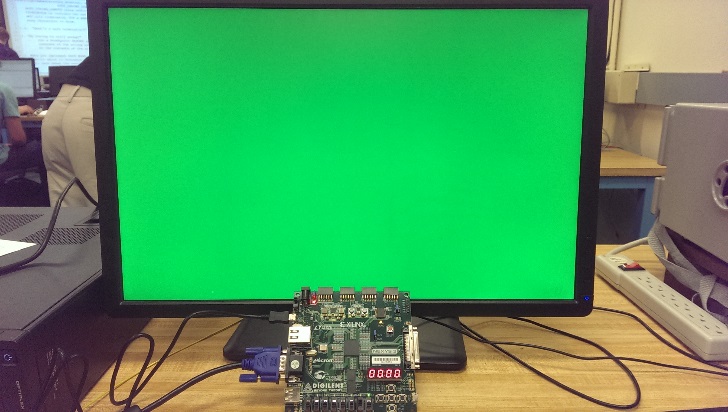


Figure : Green VGA Display

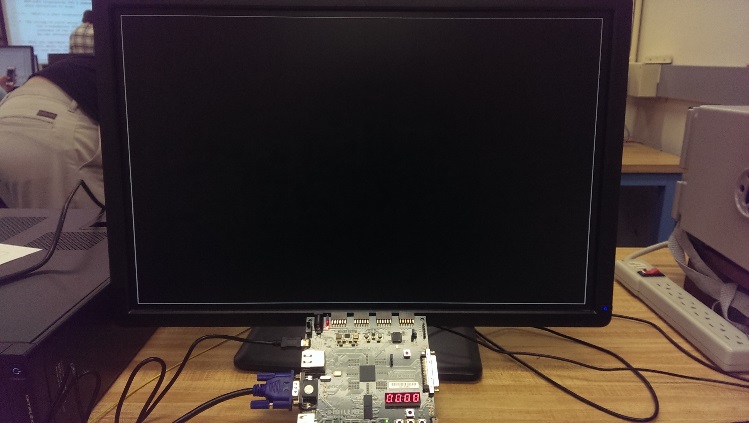


Figure : White Rectangle VGA Display

# Controlling a moving block on the VGA Display:

This section of the project required a movable block to be displayed on a monitor. The 640 x 480 display was divided into 400 segments, each 32 pixels wide by 24 pixels high. A yellow block was displayed in an initial segment, and the block was able to move between positions depending on button presses provided by a user.

## System Requirements

1. A yellow block should be displayed at a starting position determined by the value of the slider switches
   1. Three switches should be used for the x-position
   2. Three switches should be used for the y-position
2. Four separate pushbuttons should move the block either up, down, left, right at a time when pressed
3. The yellow block should not be able to move further if it hits the outside of the display
4. The current position of the block should be displayed on the seven segment display
   1. Two digits should be used for the x-position
   2. Two digits should be used for the y-position

## System Design

The top-level design for this section is shown in the diagram below.

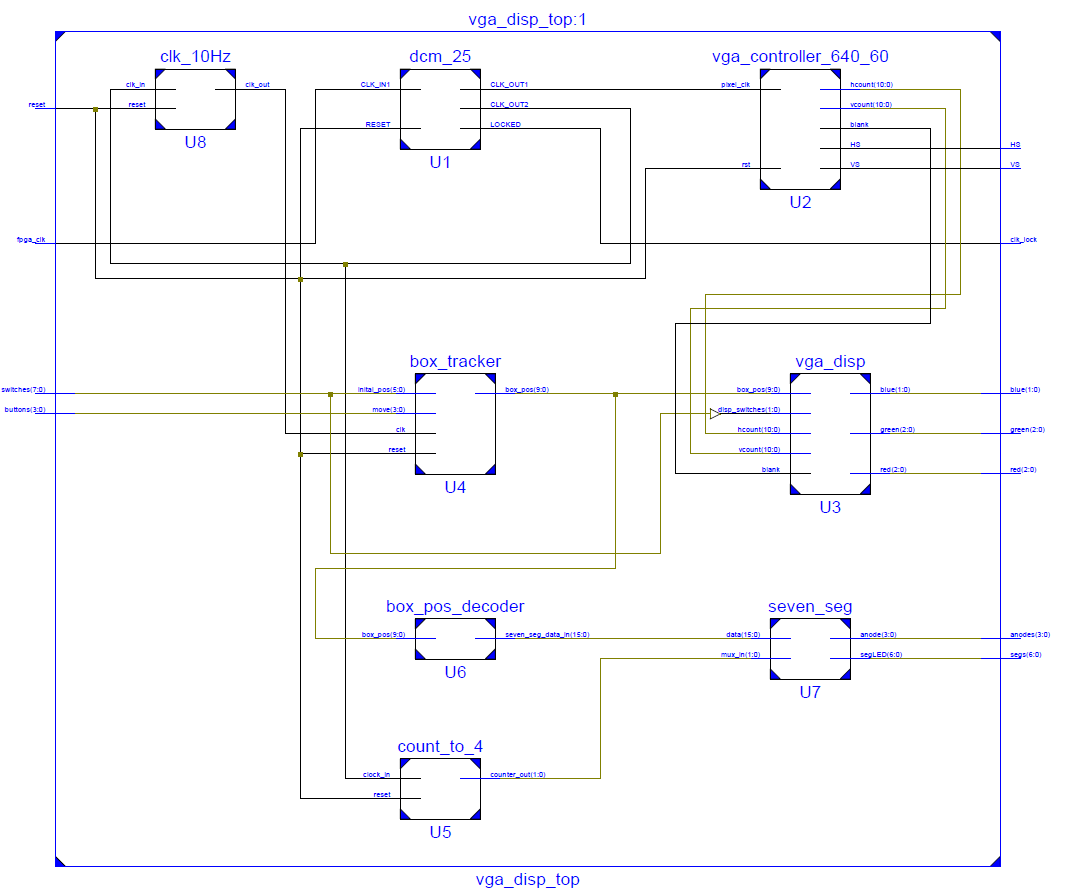


Figure : Moveable Block Top-Level Design

To accomplish the functionality for this section of the project, the box\_tracker module (U4) was created to interface with a modified version of the vga\_disp module (U3) that was discussed in Section 3, by providing the current position of the yellow block. Additionally, the box\_pos\_decoder module (U6) was created as a module that decoded the box’s current position and converted it into a 16-bit input for the seven-segment display module which was discussed in Section 2.

## Implementation Details

As mentioned above, vga\_disp module was modified. The switch inputs that determined which screen would be displayed on the monitor was given additional functionality. The unused state (the black screen) became background for the yellow block.

A 10-bit input was that contained the block’s current x and y positions was also added to the module. The positions, which are coordinates between 0-19, were multiplied by the block’s height and width in order to determine the area of the block, as shown in the code sample below.

-- Sets the position of the box

process (box\_pos)

begin

-- x-coordinate \* box width

x\_pos <= box\_pos(9 downto 5) \* BK\_X;

-- y-coordinate \* box height

y\_pos <= box\_pos(4 downto 0) \* BK\_Y;

end process;

Code Sample : Determining the block's area

Lastly, formally unused state now checked the vertical count then and horizontal count to determine where the block should be displayed. The implementation is shown in the sample code below.

-- if the vcount and hcount are more than the starting position and

-- less than (the start position + the height & width) of the box

-- then that's the area that needs to be filled.

if ( (vcount >= y\_pos) AND (vcount < (y\_pos + BK\_Y)) AND

(hcount >= x\_pos) AND (hcount < (x\_pos + BK\_X)) ) then

color <= YELLOW\_PX;

else

color <= BLACK\_PX;

end if;

Code Sample : Displaying the block's position on the screen

### Box Tracker Module

The box tracker module contained two major components: a state machine that determined which direction the yellow block should be moved, and two bi-directional counters that stored the current x and y positions of the block (as a coordinate between 0-19).

The state machine that was used is represented by the diagram below.

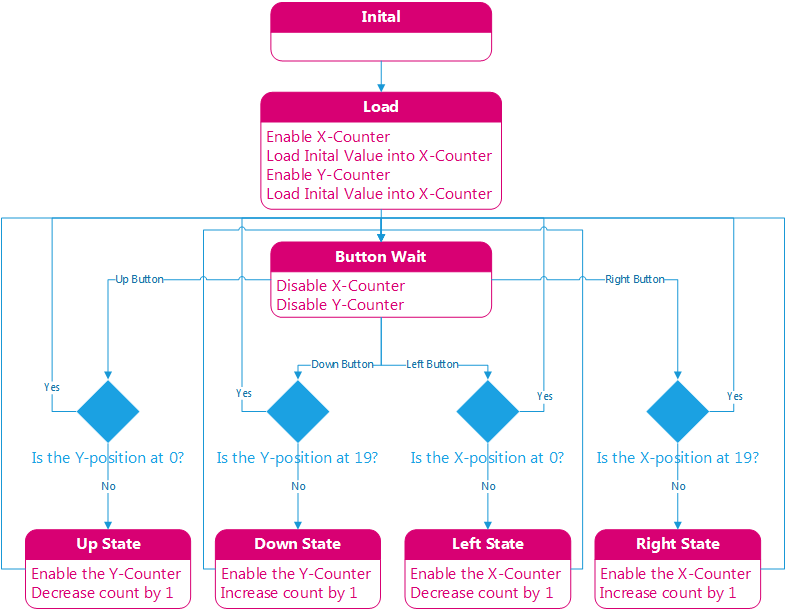


Figure : Box Tracker State Machine

The bi-directional counters were implemented as two separate modules. Depending on which state the state machine was currently in determined which directions the counter would count in. The sample code for the counters is shown below.

process(clk, reset, srt\_pos, en, pre)

begin

-- if reset, set the counter at the initial value

if reset = '1' then

count <= 0;

-- Statements for clk edge

elsif rising\_edge(clk) then

-- preset high, load the inital value

if (en = '1' and pre = '1') then

count <= conv\_integer(srt\_pos);

-- counting up

elsif (en = '1' and pre = '0' and dir = '1') then

if count = 19 then

count <= 0;

else

count <= count + 1;

end if;

-- counting down

elsif (en = '1' and pre = '0' and dir = '0') then

if count = 0 then

count <= 19;

else

count <= count - 1;

end if;

end if;

end if;

end process;

Code Sample : Bi-directional Counter

### Box Position Decoder

The box position decoder consists of three concurrent processes: one that splits the input data into two 5-bit segments, another which decodes the block’s x -position, and the last process decodes the y-position. The x and y decoder processes first determines if their respective positions are greater than 9, if this is true, then it sets the digits that correspond to the fourth and second anode as ‘1’, else it sets them as ‘0’, as shown in the code below.

-- if x\_position less than 10, then set digit\_1 to display 0

if (x\_pos < "01010") then

digit\_3 <= "0000";

-- else it's more than 9, so set digit\_1 to display 1

else

digit\_3 <= "0001";

end if;

-- if y\_position less than 10, then set digit\_1 to display 0

if (y\_pos < "01010") then

digit\_1 <= "0000";

-- else it's more than 9, so set digit\_1 to display 1

else

digit\_1 <= "0001";

end if;

Code Sample : Determines the digit for the second and fourth anodes

The x and y decoder then determines the digit to be shown on the first and third anode as shown in the code below.

-- decode to determine what digit\_0 should be

case x\_pos is

-- 0 to 9

when "00000" => digit\_2 <= "0000";

when "00001" => digit\_2 <= "0001";

when "00010" => digit\_2 <= "0010";

when "00011" => digit\_2 <= "0011";

when "00100" => digit\_2 <= "0100";

when "00101" => digit\_2 <= "0101";

when "00110" => digit\_2 <= "0110";

when "00111" => digit\_2 <= "0111";

when "01000" => digit\_2 <= "1000";

when "01001" => digit\_2 <= "1001";

-- 10 to 19

when "01010" => digit\_2 <= "0000";

when "01011" => digit\_2 <= "0001";

when "01100" => digit\_2 <= "0010";

when "01101" => digit\_2 <= "0011";

when "01110" => digit\_2 <= "0100";

when "01111" => digit\_2 <= "0101";

when "10000" => digit\_2 <= "0110";

when "10001" => digit\_2 <= "0111";

when "10010" => digit\_2 <= "1000";

when others => digit\_2 <= "1001";

end case;

Finally, the decoder combines the four 4-bit digits into a 16-bit digit that is used at the input to the 7-segment decoder, as shown below.

-- Combines the digit data into a 16-bit bus

seven\_seg\_data\_in <= digit\_3 & digit\_2 & digit\_1 & digit\_0;

## Test Results

### Box Tracker Module

A test bench was created to verify the functionality of the box tracker module. The waveform below illustrate that the box tracker performed as specified.

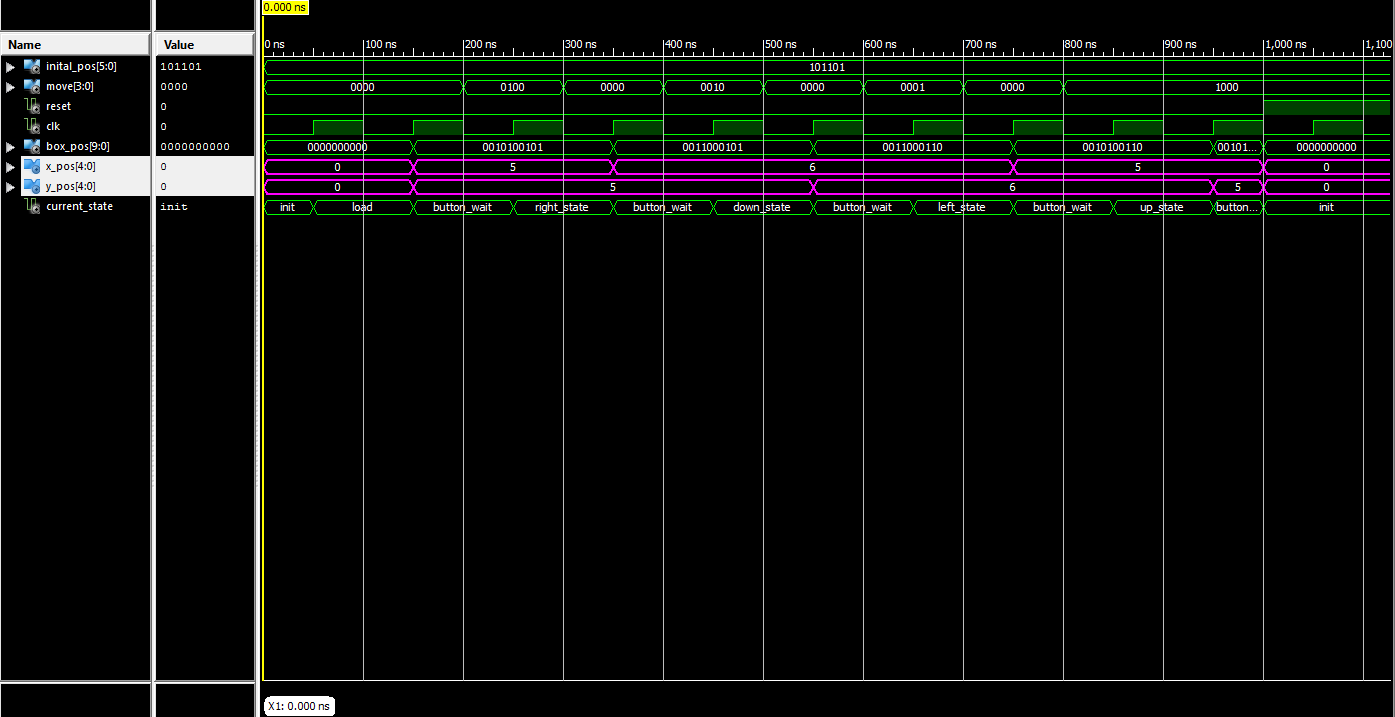


Figure : Box Tracker Waveform

As seen in the diagram above, the x and/or y position of the box changed respective to the current state of the box tracker.

Additionally, the boundary conditions were tested and verified as shown in the waveform below.

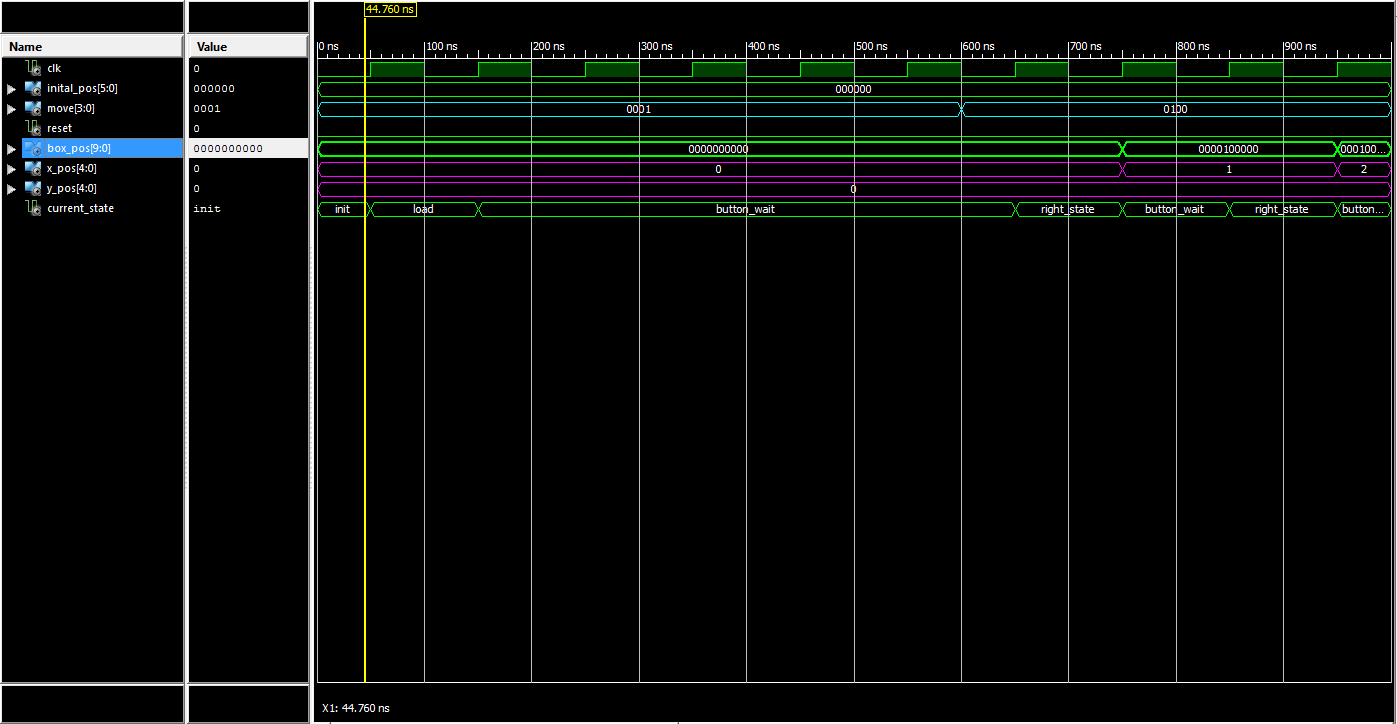


Figure : Testing Box Tracker Boundary Conditions

In the diagram above, the move button has a value of “0001” which represents left pushbutton being pressed. The button is held for six cycles, but the current x-position is at 0, the box should not be able to move left any further, hence the state machine stays in the button wait state.

### Box Position Decoder

A test bench was also created to verify that the box position decoder functioned correctly, as shown in the waveform below.

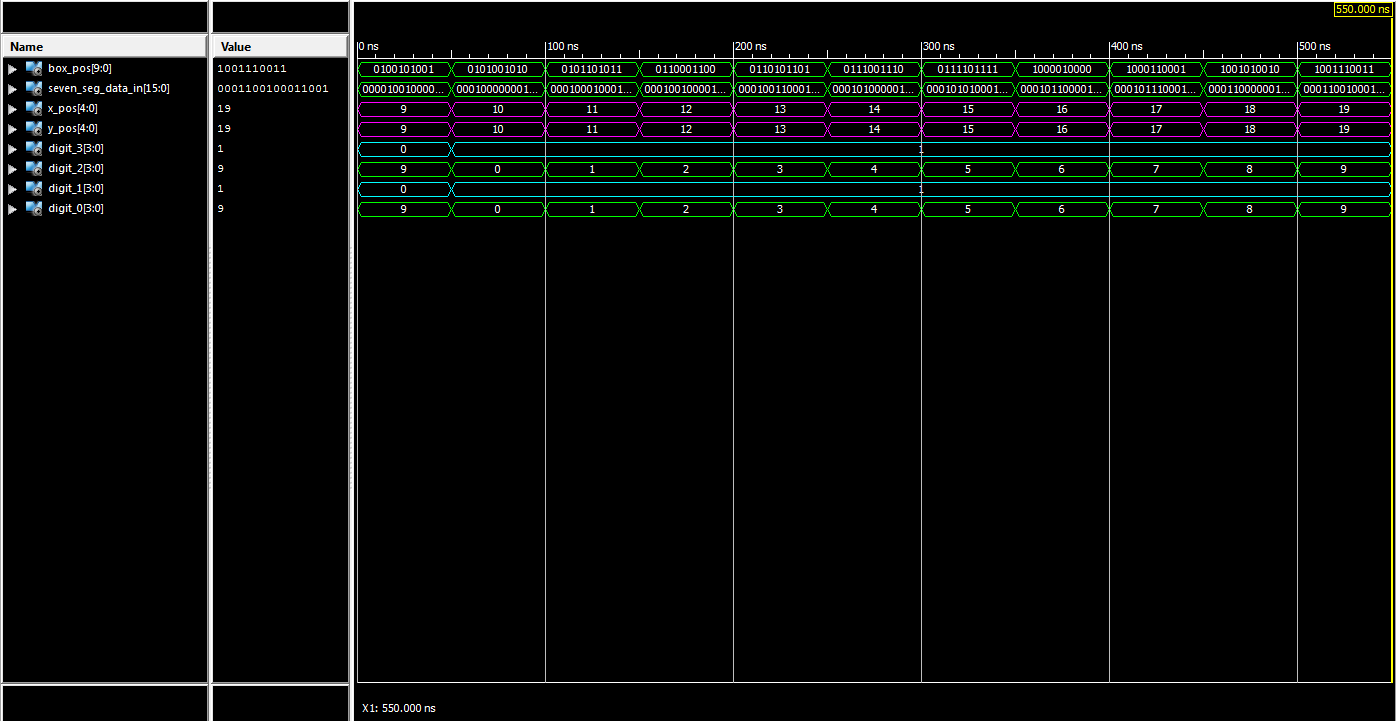


Figure : Testing the Box Position Decoder

As seen in the figure above, when the x or y position is greater than 9, digit\_3 and digit\_1 are both equal to 1, while the digit\_2 and digit\_0 contain the correct value of the block’s position. Lastly, it is also shown that seven\_seg\_data\_in correctly combines digit\_3, digit\_2, digit\_1, and digit\_0 in order to provide a 16-bit output for the seven segment display decoder.

### Combined Result

The figure below shows the combined implementation of the box tracker and box position on the FPGA board.