COL 215 – Hardware Assignment 3

Team Members:

Vedant Talegaonkar – 2022CS11603

Sabhya Khurana – 2022CS51637

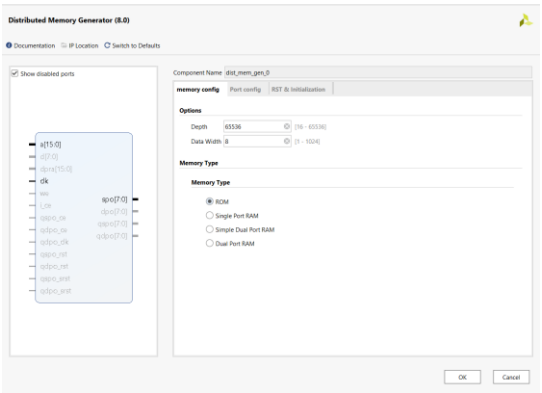
Problem Statement:

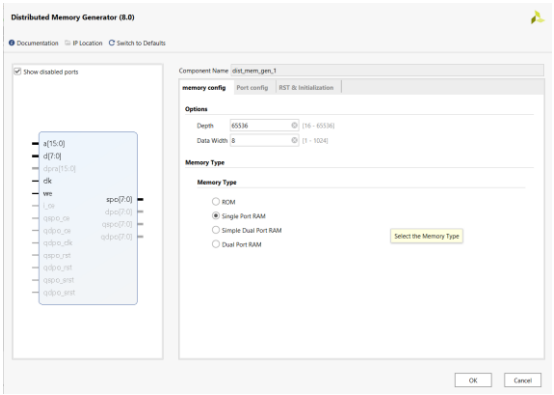
- Generate RAM/ROM using Vivado's block memory generator, populate ROM with the sample image file (.coe file) and read it from the ROM.
- Carry out a 3x3 filtering operation on the image, that is, read the image pixel values from ROM and apply the filter operation.
- Apply an image normalisation function to ensure that the output pixel values fit within 0-255, so that the output image is also stored in the 8-bit unsigned format.
- Display the memory written on RAM/ROM on a VGA display monitor.

Part 1:

We used the Block Memory Generator module of the IP catalog in Vivado to generate distributed RAM/ROM.

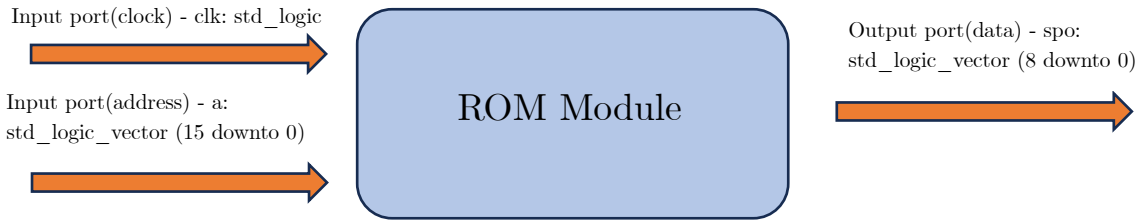
We created a registered ROM and registered RAM, as we needed the "clk" port for our operations. The image has dimensions 64x64 with each pixel taking up 8 bits, we set depth of the image data ROM as 4096 and width as 8. We set the depth and width of output block RAM as 4096 and 20 as well, the reasons for which are discussed. Since the number of values required to be stored in the kernel (filter) ROM are 9 and we can only have sizes which are perfect powers of 2, we get $2^{\lceil \log_2 9 \rceil} = 16$. Hence, the depth is 16 and the width is 8.





Description of ROM module:

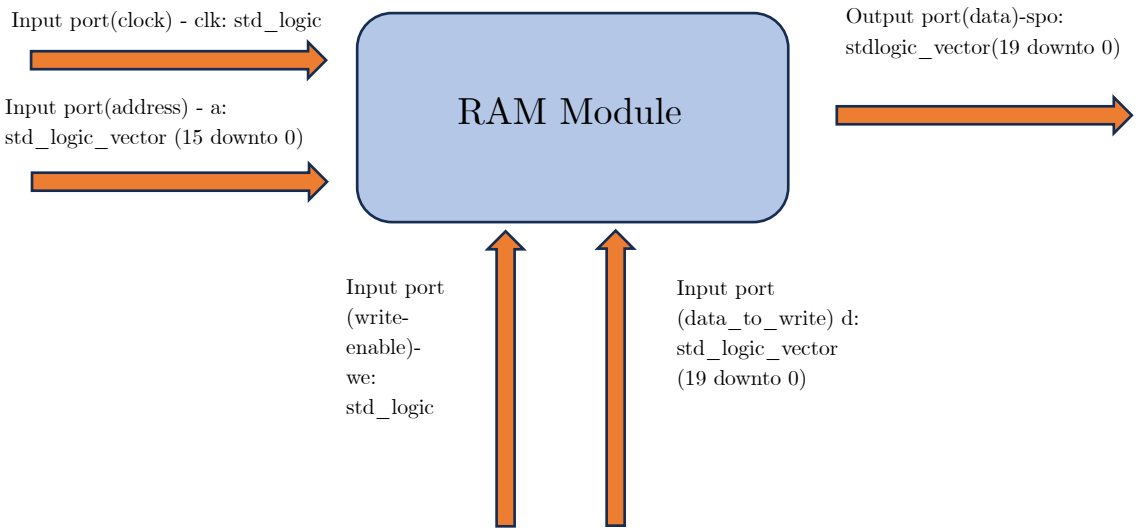
ROM (Read-Only Memory) is an entity used for storage of memory which does not needs to be changed from time to time. It is generally used for large memory sizes which need to remain constant with time. The block module for registered ROM in Vivado is shown below –



Description of RAM module:

RAM (Random Access Memory) is an entity used for storage of memory which keeps changing with time and needs to be accessed constantly. Generally, RAM is smaller in size than ROM and can be written into at any time.

The block module for registered RAM is shown below-



We have created two block ROMs here. One of them stores the image data, the other stores the kernel, also known as the filter. The block RAM stores the output.

Part 2:

Our task in part 2 was to calculate the filtered value for every 3x3 sub-grid that existed in the ROM, normalise it to fit in the 0-255 pixel limit and populate the RAM with this data. The filter is given by –

$$O(i,j) = a * I(i-1,j-1) + b * I(i-1,j) + c * I(i-1,j+1) +$$
$$d * I(i,j-1) + e * I(i,j) + f * I(i,j+1) + g * I(i+1,j-1) +$$
$$h * I(i+1,j) + i * I(i+1,j+1)$$

where I is the image stored in the ROM, and (a,b,c,d,e,f,g,h,i) correspond to the locations shown in the grid:

a	b	c
d	e	f
g	h	i

O(i, j) is the output value at cell in the i^{th} row and j^{th} column which we feed into RAM after normalising, I(i, j) is the input value at cell in the i^{th} row and j^{th} column, with both i and j going from 0 to 63. We define I(i, -1) and I(i, 64) as 0 for every i to calculate the gradient at the start and end of each row.

We have created four modules in total - FSM, memory, MAC and VGA display. All of the modules take commands from the FSM module in some form, like the control signal (for MAC) or address (for memory).

To normalise the output, we need to find the maximum and minimum output value across the 64x64 image. So, we iterate through this image twice, and we store the maximum and minimum values in the first iteration. In the second iteration, we calculate the value, normalise it according to the formula:

$$New\_I(i,j) = (I(i,j) - \min) * \frac{255 - 0}{max - min}$$

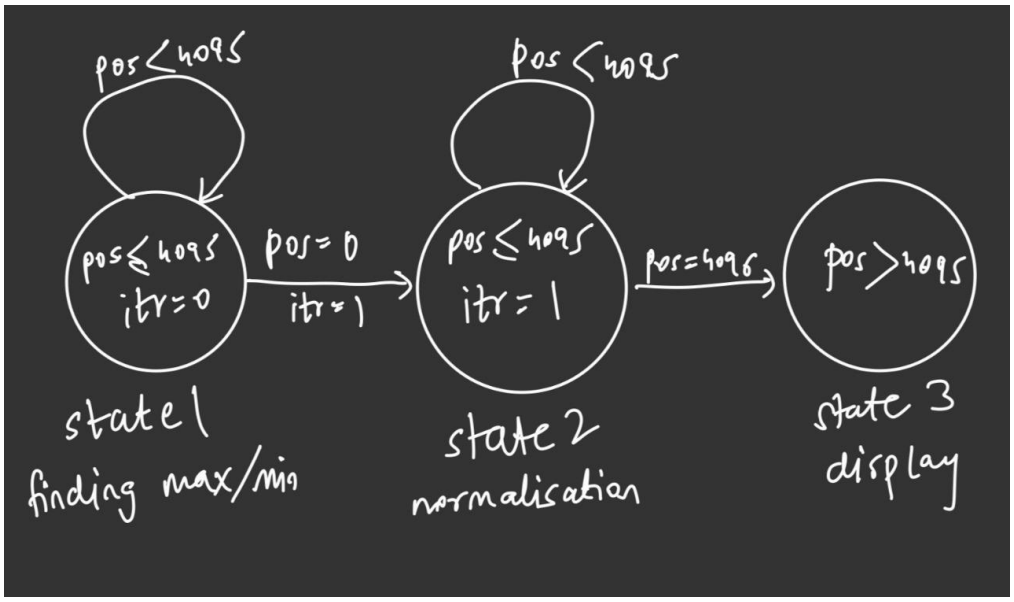
And then we write the New_I value into the RAM.

To manage the clock cycles, i.e. reading and writing data in the correct order, we have used two registers- row and ctr. These ensure that only the task designated for that unique combination of (row, ctr) will occur.

We have optimized the process of calculating the output value for each input pixel. We use the fact that while calculating the sum of product of kernel values with the neighbouring elements, 6 of the elements (2nd and 3rd columns) get overlapped in the calculation for the pixel immediately to the right. So, in that case, we simply shifted the data from column2 to column1, and column3 to column2, which reduces the no. of clock cycles used significantly.

Analysis of FSM Module:

1. There are 3 parts in our FSM process - finding max/min, normalisation and display. Using the values of signals 'pos' and 'itr', we control which part the process operates in.



2. In the first part, we calculate the value from the formula given for the kernel. Basically, we have divided this part into 10 sub-parts, and in each sub-part we calculate the value of (kernel data)*(image data) for one element from the 3x3 grid, and in the last part, we take the output from MAC and calculate max/min.
   For example, if we are doing this for the top-right corner, we first find the image data of that address, then we find the kernel data of that address, and we pass these two inputs to the MAC.
3. For the *cntrl* value of 2, MAC multiplies these two inputs and stores them. We then change the value of *cntrl* to 3, for which

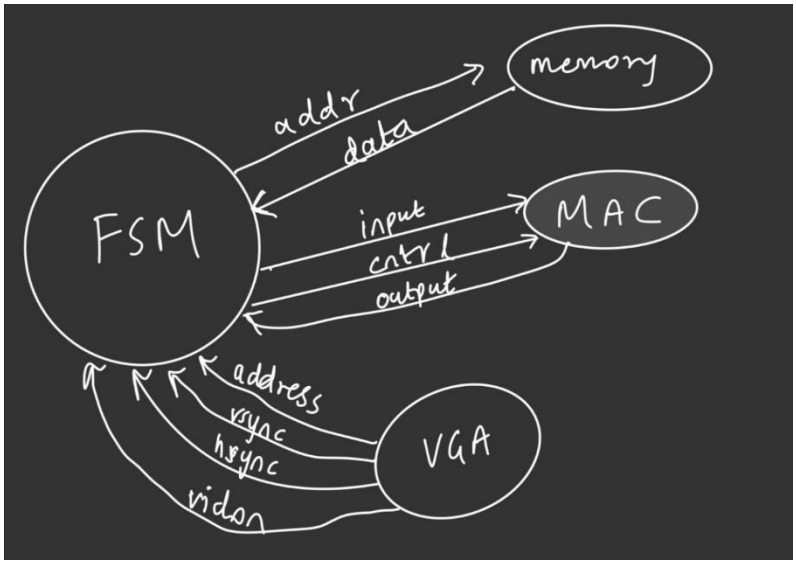
MAC adds this stored value to the accumulator register. After repeating these steps 9 times for the 3x3 grid, this value of accumulator register is sent to the output by setting the *cntrl* value to 4. The FSM then receives this data.

4. We then update the max/min value, set *write-enable* = 1 and write this data into the RAM.
5. The RAM we have created has a data width of 20 bits, and a depth of 4096. This comes from a calculation - maximum value written into RAM:

$$9 * (\max kernel) * (max * image) \leq 2^4 * 2^8 * 2^7 < 2\text{^}20.$$

Hence, we arrived at this number in order to prevent any kind of data overflow.

6. This procedure is done for all 4096 values, and the process then shifts to the next phase - normalisation. In this phase, we go at every position in the RAM, normalise the data according to the calculated max/min in the first phase, and overwrite it into the same position of RAM.
7. After this, we move to the display phase. According to the values of *hcount* and *vcount* in the VGA display, we receive a 12-bit address, which represents the pixel position on the screen for which we have to display. We set RAM address equal to this value, so that the data at that position of the RAM is displayed. Also, we set the value of write-enable to 0 to read the data.



Analysis of MAC Module:

The MAC module in this assignment differs from the one we made in part 1. The reasons for this are -

Since we didn't have to display on the VGA in part 1, we had stored the data for the MAC - the one we calculated over 9 clock cycles - in arrays of length 9. We sent this array to our MAC, and it calculated the value accordingly. This method worked because the simulation on a computer does not actually involve hardware, so even large integer calculations can be done in one clock cycle. However, for this part, that won't work.

So, we changed the MAC so that it would accumulate data gradually over a few clock cycles and give the correct output.

We had to take care of the following:

1. Normalising the value cannot be done in one clock cycle, as multiplication and division of integers takes more time on the Basys board. So, we have ensured enough delays in order to do the calculation thoroughly.

2. Writing data into RAM needs a few clock cycles to take effect (>= 5). We have left the necessary delay in order to write into the RAM properly.

Also, reading data from BRAM takes 2 clock cycles, in comparison to the 1 clock cycle needed by DRAM. We have taken this into account too.

Timing Analysis of FSM Process:

For the first phase, each of the 9 sub-part takes atleast 12 clock cycles, in the following manner:

1. Setting image address - 2 clock cycles

2. reading image data - 2 clock cycles

3. setting kernel address - 2 clock cycles

4. reading kernel data - 2 clock cycles

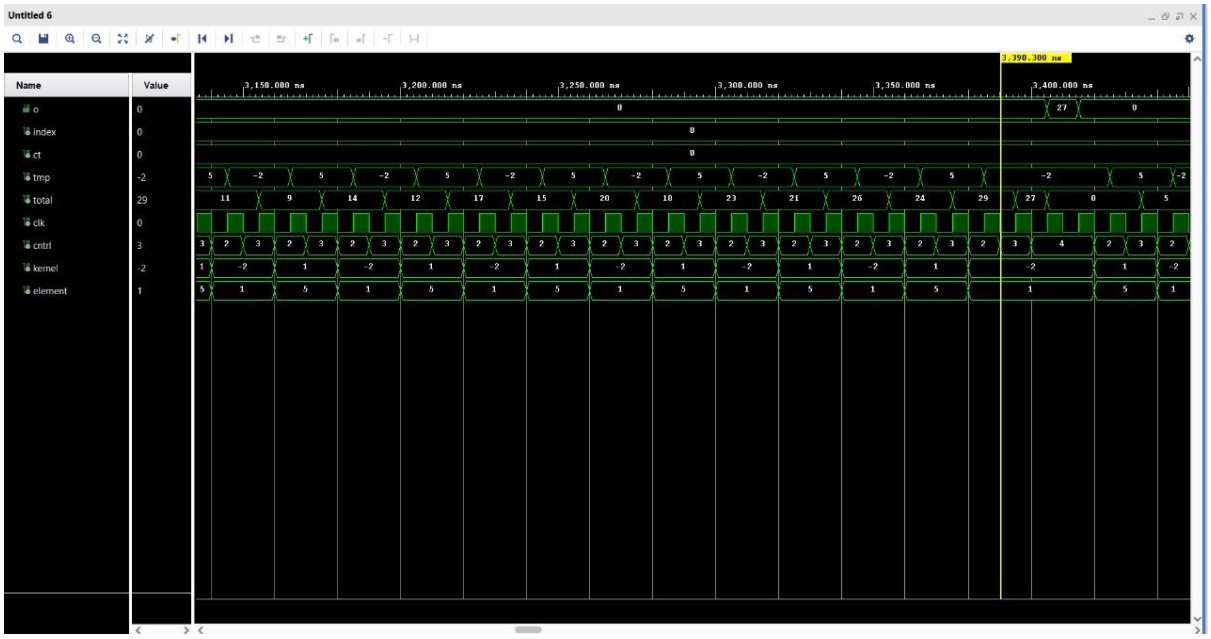
5. MAC operations - 4 clock cycles

We have kept more clock cycles in the delay as buffer, but this process can be done in 12 cycles theoretically.

This is done for each sub-part, so around 120 clock cycles are required.

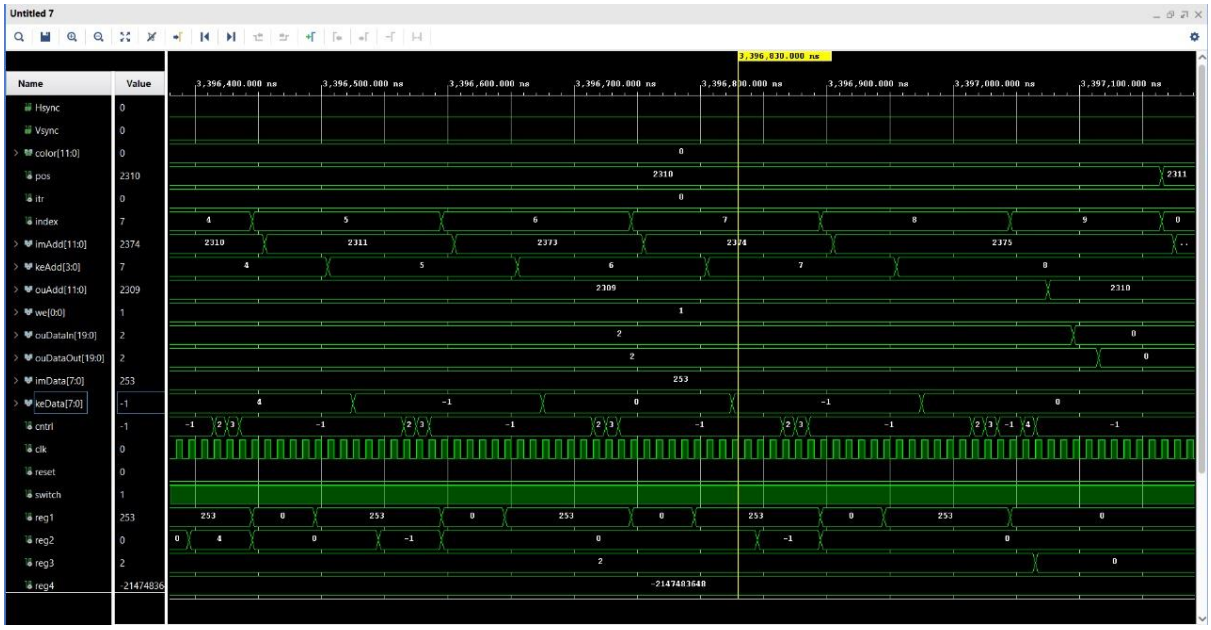
Simulation:

The simulation for the MAC module:



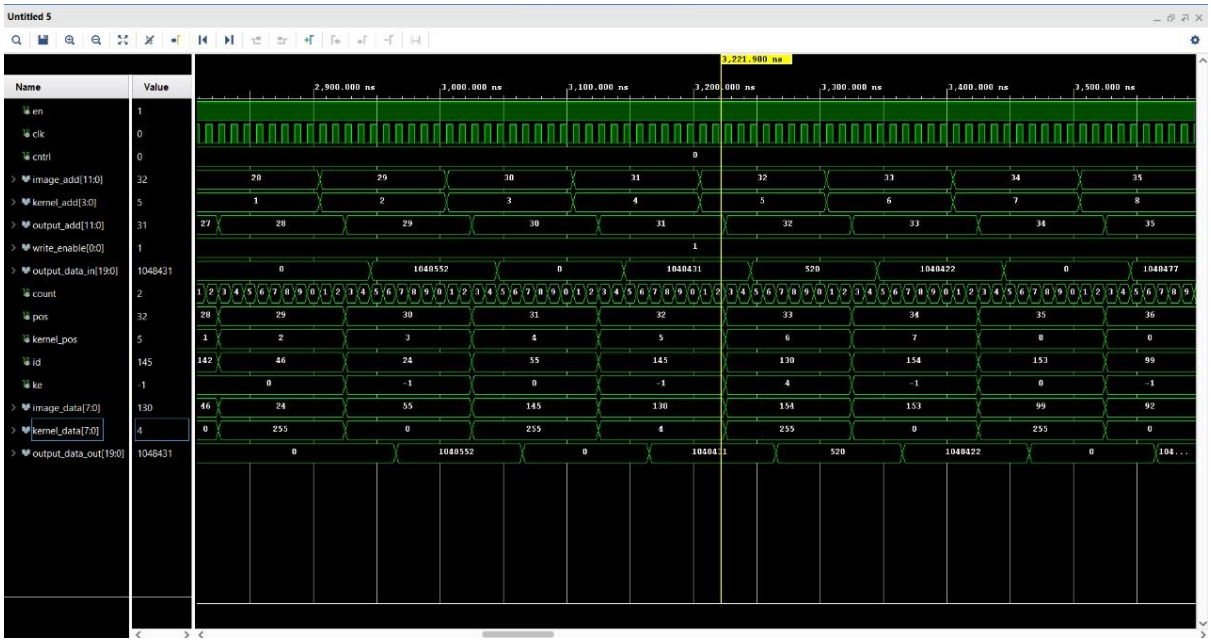
We fed random inputs to the MAC module and tested the values taken for different values of *cntrl*.

The simulation for the FSM module in its calculating phase:



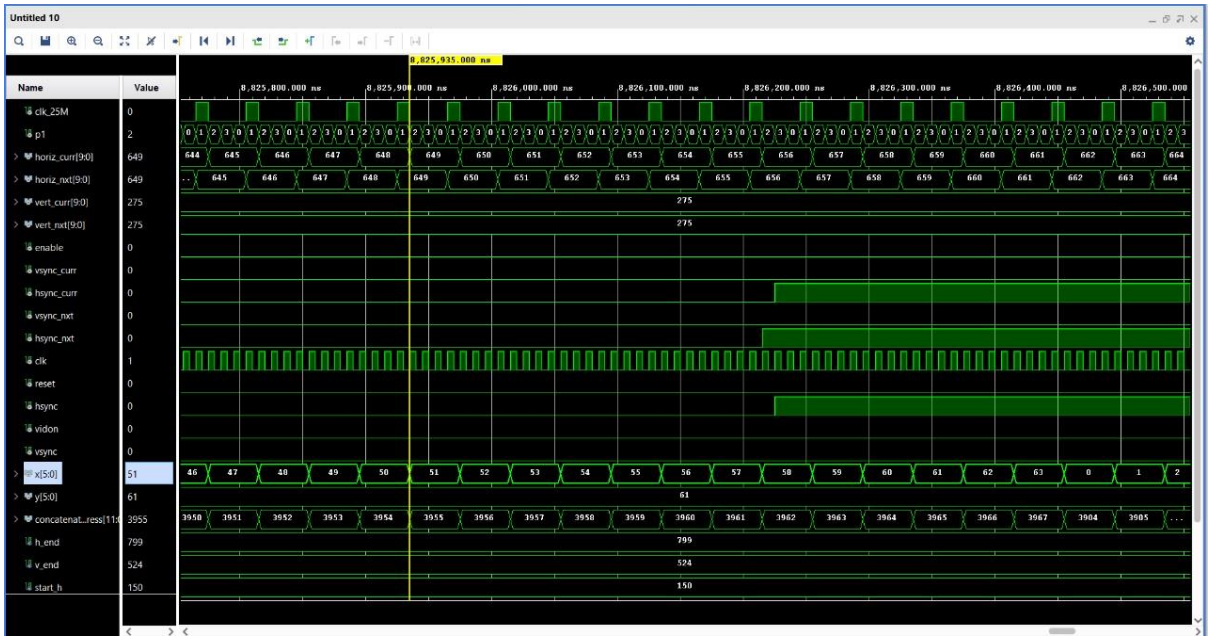
We checked if the value for maximum and minimum was being calculated properly at every step.

The simulation for the memory module:



We checked that the output_data_in matches output_data_out, and the address is being updated regularly

The simulation for the VGA module:



We checked if it was taking the values of horiz_curr and vert_curr correctly and in the entire range.

Synthesis Report:

Flip-flops and LUTs:

```
1. Slice Logic
--------------

+-------------------------+------+-------+------------+-----------+-------+
|        Site Type        | Used | Fixed | Prohibited | Available | Util% |
+-------------------------+------+-------+------------+-----------+-------+
| Slice LUTs*             | 1901 |     0 |          0 |     20800 |  9.14 |
|   LUT as Logic          | 1901 |     0 |          0 |     20800 |  9.14 |
|   LUT as Memory         |    0 |     0 |          0 |      9600 |  0.00 |
| Slice Registers         |  310 |     0 |          0 |     41600 |  0.75 |
|   Register as Flip Flop |  306 |     0 |          0 |     41600 |  0.74 |
|   Register as Latch     |    4 |     0 |          0 |     41600 | <0.01 |
| F7 Muxes                |    0 |     0 |          0 |     16300 |  0.00 |
| F8 Muxes                |    0 |     0 |          0 |      8150 |  0.00 |
+-------------------------+------+-------+------------+-----------+-------+
```

7. Primitives

Ref Name	Used	Functional Category
LUT5	1049	LUT
CARRY4	419	CarryLogic
LUT2	310	LUT
FDRE	284	Flop & Latch
LUT3	245	LUT
LUT6	208	LUT
LUT4	166	LUT
LUT1	155	LUT
FDCE	22	Flop & Latch
OBUF	14	IO
LDCE	4	Flop & Latch
IBUF	3	IO
DSP48E1	3	Block Arithmetic
BUFG	1	Clock

Memory in Synthesis Report:

2. Memory

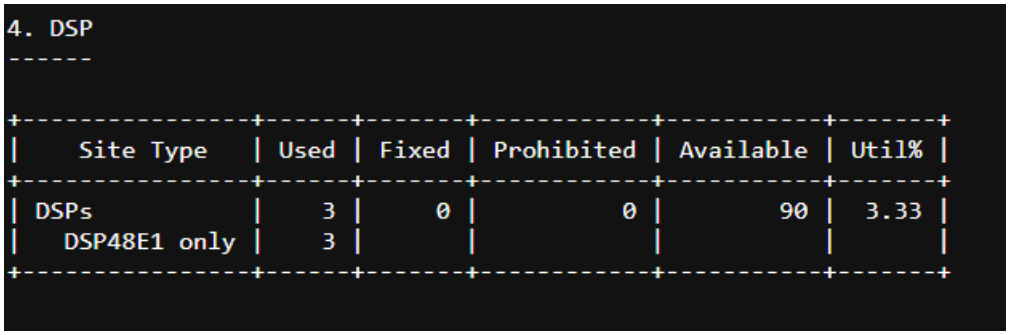
Site Type	Used	Fixed	Prohibited	Available	Util%
Block RAM Tile	0	0	0	50	0.00
RAMB36/FIFO*	0	0	0	50	0.00
RAMB18	0	0	0	100	0.00

Memory in Implementation Report:

3. Memory

Site Type	Used	Fixed	Prohibited	Available	Util%
Block RAM Tile	4	0	0	50	8.00
RAMB36/FIFO*	3	0	0	50	6.00
RAMB36E1 only	3				
RAMB18	2	0	0	100	2.00
RAMB18E1 only	2				

DSP:



```
4. DSP
------

+----------------+------+-------+------------+-----------+-------+
|    Site Type   | Used | Fixed | Prohibited | Available | Util% |
+----------------+------+-------+------------+-----------+-------+
| DSPs           |    3 |     0 |          0 |        90 |  3.33 |
|   DSP48E1 only |    3 |       |            |           |       |
+----------------+------+-------+------------+-----------+-------+
```

Other details can be found in the submitted synthesis and implementation reports.

References:

For the VGA Controller module, we referred to online resources for our understanding of VHDL and its data flow. A few of these are:

- ➔ https://stackoverflow.com/
- ➔ https://www.ece.ualberta.ca/~elliott/ee552/studentAppNotes/1998_w/Altera_UP1_Board_Map/vga.html
- ➔ https://embeddedthoughts.com/2016/07/29/driving-a-vga-monitor-using-an-fpga/