ECE 590: Digital Design Using HDL

Project Report

IMAGE PROCESSING USING FPGA

Submitted by:

Sumitha Ajith

Saicharan Bandarupalli

Mahesh Borgaonkar

Contents

1. Abstract *5*
2. Introduction *5*
3. Hardware and Software *6*
4. Phase 1: FPGA Implementation of VGA Display with SRAM as Video Memory *8*
4.1 VGA Controller Design *8*
4.1.1 Introduction to VGA Controller 8
4.1.2 VGA Signal Timing 9
4.1.3 HDL Implementation of the VGA Controller 11
4.1.4 Understanding the Interfacing details with FPGA 12
4.1.5 Sync Signal Generation, Testing and Results 12
4.1.6 Bitmap generation logic using Block RAM 13
4.1.7 Testing and results 16
4.1.8 Generating 16-bit VGA Color Output 16
4.2 Memory Controller *20*
4.2.1 Role of SRAM Memory Controller 20
4.2.2 Block Diagram of Memory Controller 22
4.2.3 Timing Requirement 23
4.2.4 Design 1: Safe Design – Back-to-back memory access time 60ns. 25
4.2.5 Design 2: Using Xilinx ISE DCM for Faster Access Time 26
4.2.6 Testing and Results 29
4.3 Complete Display System Design *31*
4.3.1 Experiment Results *33*
5. Image Processing *36*
5.1. Design Considerations 36
5.2. Basic Concept of Blob Analysis 37
5.3 Sequential Component Algorithm 37
5.4 Merging and Blob Parameter calculation 40
LINE FIFO Usage 41

BLOB Table Usage 42

5.5 HDL Implementation for Simulation 45

5.6 ModelSim Simulation Results.. 46

6. Reference 49

Figures

Figure 1: Spartan 3 FPGA Development Board from Digilent... 7
Figure 3: VGA Controller Block Diagram 9
Figure 4: Timing Diagram of Horizontal scan. 10
Figure 5: Timing Diagram of a Vertical scan 11
Figure 6: VGA Synch Signal generator Test Results - Screen bitmapped to unique color input read from Switches 13
Figure 7: Block RAM as a Component - Interface Diagram 14
Figure 8: Configuring Block RAM 15
Figure 9: Image from FPGA Block RAM.. 16
Figure 10: General R2R ladder circuit 17
Figure 11: Digital to analog conversion formula using the R2R ladder network......................... 17
Figure 12: Circuit board - Custom built 16bit VGA port.. 18
Figure 13: Interface between SRAM memory and FPGA.. 20
Figure 14: Role of SRAM as Memory Controller 21
Figure 15: Block diagram of a memory controller 22
Figure 16: Timing parameters of Read operation.. 23
Figure 17: Timing parameters of a write operation 24
Figure 18: ASMD chart of SRAM Controller 25
Figure 19: Block diagram of a Digital clock Manger... 26
Figure 20: Clock Synthesis Options with DCM 27
Figure 21: DCM as a component: Interface details 28
Figure 22: Improved Design with 200MHz clock satisfying timing constraints 29
Figure 23: SRAM controller test results 30
Figure 24: Display with SRAM as Video Memory... 31
Figure 25 FSM for transferring image from Block RAM to SRAM 32
Figure 26: Block diagram for Displaying image from SRAM 33
Figure 27: Image displayed from Block RAM with 3bit color resolution 34
Figure 28: Image displayed from SRAM with 16 bit resolution 34
Figure 29: Close up of the Image displayed using the 16 bit custom VGA adapter 35
Figure 30: Memory Contention 36

Figure 31: Kernel defined for labeling 38
Figure 32: Pseudo code for Sequential Component Algorithm 39
Figure 33: Example of CCL Algorithm 40
Figure 34: Blob Detection Module 40
Figure 35: LINE FIFO Usage Example 41
Figure 36: Blob detection using Connected Component Algorithm 43
Figure 37: Intermediate Labeled Pixel array 44

1. Abstract

Image processing is a computationally intensive operation and is typically done in software using CPU processing power that is readily available these days. However, even with the advances in computing technology today, software based image processing requires expensive and powerful CPUs to perform real-time image processing, making it out of reach for most robotic applications. This is where a low cost FPGA based image processing solution becomes useful. This eliminates the need for powerful CPUs and at the same time can achieve real-time processing relatively easily. This project implements such an image processing solution in hardware, using a FPGA at its core. The high level goal is to retrieve an image from an image source, detect the presence of a desired object based on color and calculate its parameters like area and centroid, while displaying the image on a VGA monitor. This report outlines the implementation of the project in two phases. First phase describes the process of displaying an image on a VGA monitor using SRAM as the video memory. The second phase describes the implementation of a blob detection algorithm based on sequential connected component labeling algorithm. A modified version of the labeling algorithm is implemented, which enables the processing of an entire image in a single pass through the image. The design enables the processing step to be inserted in the pixel data path from the image source to the video memory.

2. Introduction

We adopted a divide and conquer policy to achieve the high level goal of image processing using FPGA. Hence the work is organized into two phases. First phase consists of FPGA implementation of basic building blocks like VGA controller for display and memory controller for SRAM based video memory. The second phase consists of development of the image processing algorithm. The entire work is explained as Phase1 and Phase2 in this report.

Although Image acquisition is an important building block for the project, it requires the development of a complete camera interface unit. We decided to do it as the third phase of the project due to time constraints. This is an undergoing venture and the details are not in this current report. However, we need an image source for the rest of the project development. Hence we decided to use the memory that comes with FPGA - BLOCK RAM as the image source. Specifically the Block RAM available in Xilinx Spartan 3 boards is used for this purpose. The advantage is that we can initialize the Block RAM with an image using Xilinx ISE software. The image in Block RAM is transferred to the SRAM first and then displayed using the timing signals generated by the VGA controller module. The Spartan 3 FPGA development board has 1 MB SRAM in it and a memory controller is developed specifically for this board memory. FSMs are

developed to implement the memory controller and to communicate between video controller and the memory controller to display the image on the monitor.

Second phase involved the development of sequential connected components labeling algorithm for blob detection. The classic algorithm requires two passes through an image. We present an algorithm that allows the connected components to be analyzed in a single pass by gathering data on the regions as they are built. The scheme only requires a Line buffer and a Blob Table. This avoids the need for huge buffering requirements of the image, making it ideally suited for processing streamed images on the FPGA or other embedded system with limited memory.

Section 3 explains the hardware and software used for the project. Section 4 describes the FPGA implementation of the building blocks. First subsection explains the usage of block RAM and the default 3-bit color VGA interface available on the Spartan 3 board. Later we built a circuit board that has R2R ladder network to enhance the display with 16bit color output. Section 5 describes the sequential connected component algorithm and HDL implementation.

3. Hardware and Software

The Spartan 3 FPGA Development board from Digilent is used for the project. An additional circuit board with R2R ladder network is built to get 16bit VGA display. ModelSim is used for RTL simulation. Xilinx ISE is used for synthesis. The impact tool that comes with Xilinx ISE is used for downloading the bit stream into the Spartan 3 FPGA.

Figure 1. shows the Spartan 3 FPGA board. A few specific parts of the board that are of interest to us include dedicated VGA port, three 40-pin expansion ports, four seven segment LED displays, eight slide switches, four momentary-contact push button switches and eight LEDs. There is an on-board voltage regulation to provide 3v, 2.5v and 1.2v. The board contains 1MB of Fast Asynchronous SRAM (10 ns SRAMs in two 256K*16 configuration) and 50 MHz oscillator on the back of the board

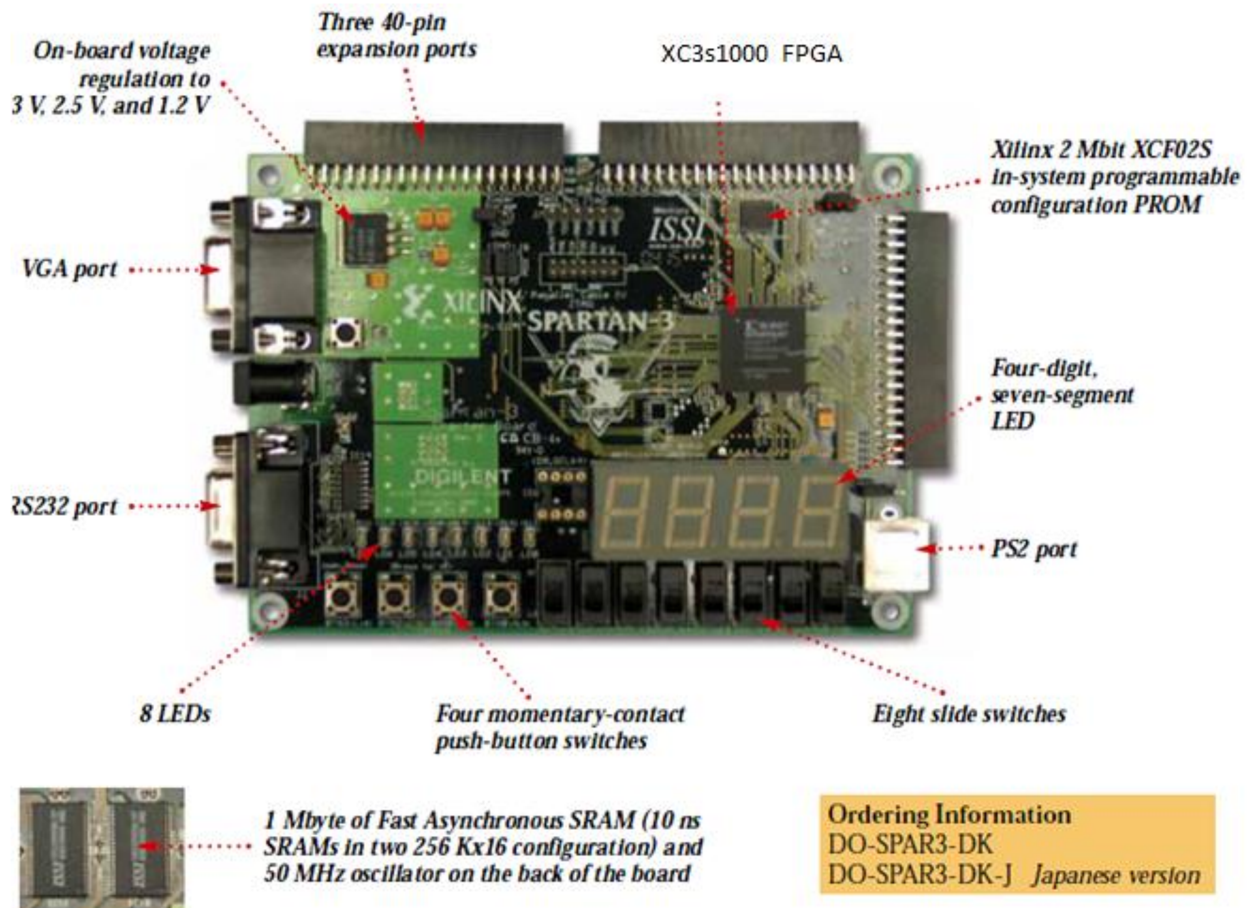


Figure 1: Spartan 3 FPGA Development Board from Digilent

4. Phase 1: FPGA Implementation of VGA Display with SRAM as Video Memory

The various development steps for this phase are listed below. Each part is explained in detail in the following sections.

- Design and implementation of VGA Controller
- Implement Bitmap generation logic using Block RAM.
- Design and Implementation of SRAM controller.
- Implementation of the Bitmap generation logic using SRAM as the video Memory.

4.1 VGA Controller Design

As our project is based on image processing, we thought to have a display mechanism first of all for efficient debugging. Our approach to the design is as follows:

1. Understanding the VGA Controller functionality.
2. Generation of VGA Synchronization signals.
3. Programming VGA Controller in Verilog.
4. Understanding the Interfacing details with FPGA.
5. Implementation, testing and results.
6. Expected additions in the Future.

4.1.1 Introduction to VGA Controller

A simple block diagram of a video controller is shown in Figure 2. A video controller generates the synchronization signals and outputs data pixels serially through the VGA port of the FPGA board. The synchronization signals generator circuit (vga_sync) generates the timing and control signals. The **hsync** and **vsync** signals control the horizontal and vertical scans. The **pixel_x** and **pixel_y** signals specify the current location of the pixel. The vga_sync circuit generates the **video_on** signal to indicate whether to enable or disable the display. The Pixel generation circuit generates the three video signals which are collectively referred to as the **rgb** signal.

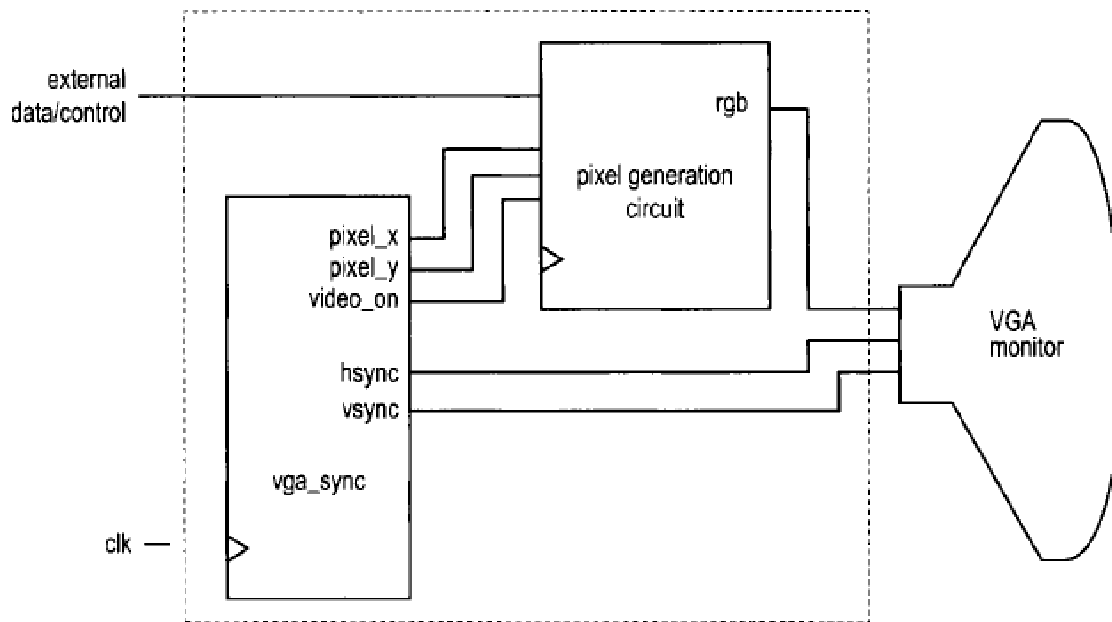


Figure 2: VGA Controller Block Diagram

4.1.2 VGA Signal Timing

The following discussions are based on a 640-by-480 VGA screen. Detailed timing diagram of one horizontal scan is shown in Figure 3 and the timing diagram of the vertical scan is shown in Figure 4. The refresh rate for the display is fixed at 60 Hz, which is the typical value used in monitors. In order to achieve this refresh rate on a 640x480 pixel screen, the pixel rate calculation is done as follows:

Pixel Rate = (Total Horizontal Pixels * Total Vertical Lines *Number of screens / second)

= 800 * 525 * 60 = 25 MHz

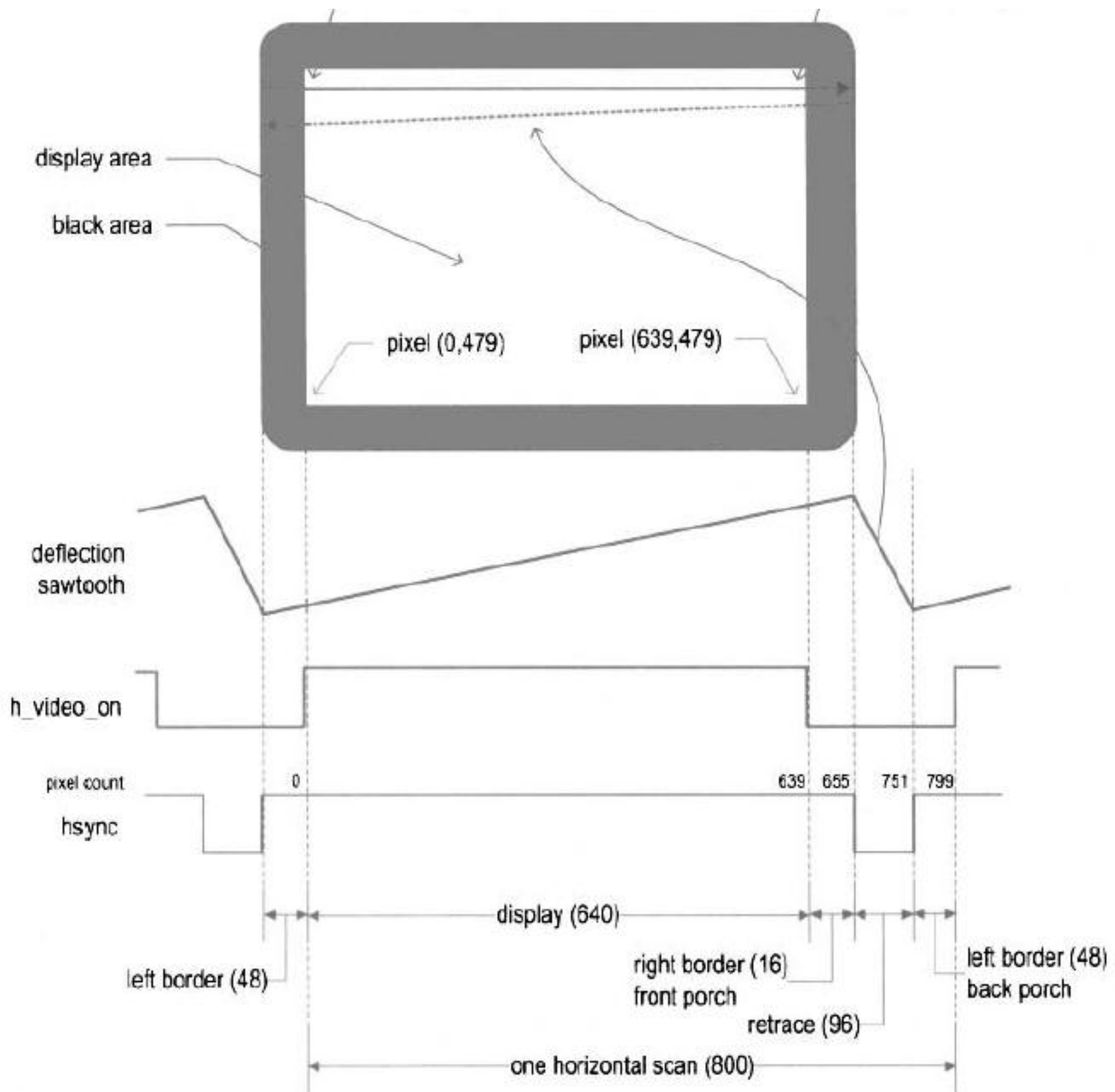


Figure 3: Timing Diagram of Horizontal scan.

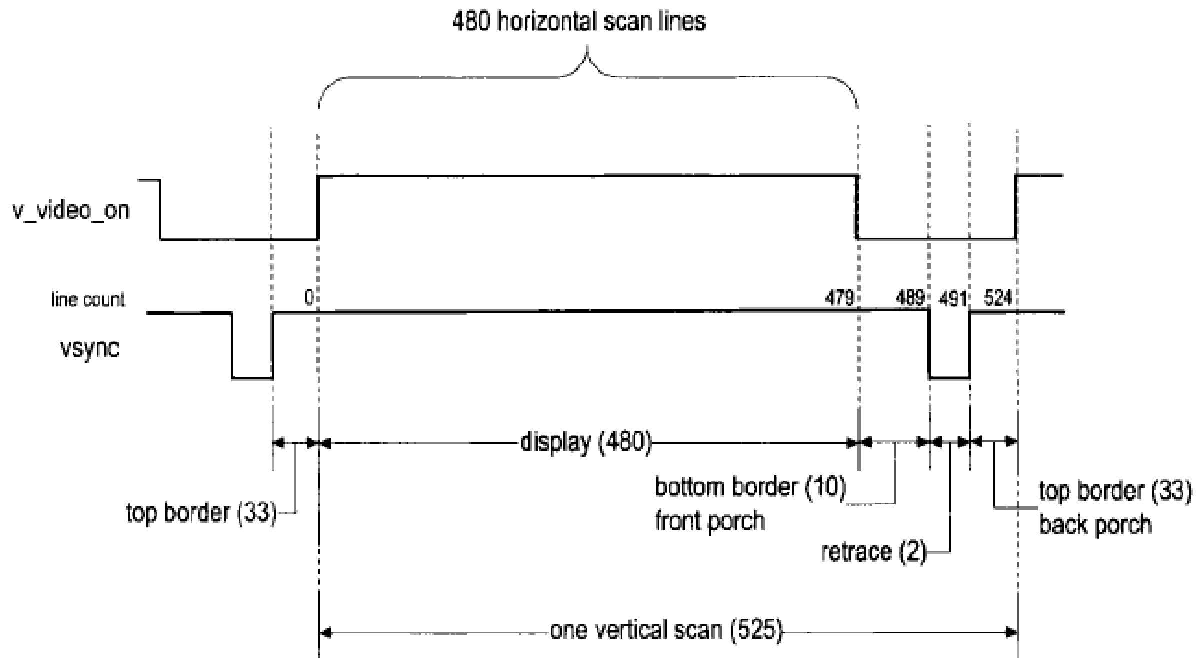


Figure 4: Timing Diagram of a Vertical scan

4.1.3 HDL Implementation of the VGA Controller

The implementation of a simple VGA controller is fairly straight forward and is described extensively as examples in various FPGA text books. The following steps summarize the implementation details:

- The sync signal can be obtained by a special mod-800 counter and a decoding circuit. We intentionally start the counting from the beginning of the display region so that the counter output can be used as the horizontal (x-axis) coordinate. This output constitutes the pixel_x signal.
- The vsync signal can be obtained by a special mod-525 counter and a decoding circuit. This output constitutes the pixel_y signal.
- video_on signal is used to indicate whether the current vertical coordinate is in the displayable region. It is asserted when the line count is smaller than 480.
- 50MHz board clock is used as the system clock.
- The pixel clock is half of the system clock. A mod-2 counter is used to generate 25MHz pixel clock signal.
- The pixel tick should be routed via port as output signal to pixel generation circuit to extract the pixel content at the same rate.

- The values of the various regions of the horizontal scans and vertical scans are defined by the constants.
- To avoid glitches, the output buffers are inserted for the hsync and vsync signals.
- The Verilog HDL code is shown in in Appendix A.

4.1.4 Understanding the Interfacing details with FPGA

The Spartan 3 FPGA board that we used for this project has a built-in VGA port with five active signals as, hsync, vsync, and video signals - red, green, blue. The video signal for VGA is an analog signal, and so a typical video controller uses a D-A converter. However, in the S3 FPGA, only 1 bit is used per color. So it does not require a D-A converter. There are three video color signals available, so we can have eight different colors which can be displayed on screen. For that we need to give proper binary input combinations to VGA port. Following table shows the different possible color combinations.

Red (R)	Green (G)	Blue (B)	Resulting Color
0	0	0	Black
0	0	1	Blue
0	1	0	Green
0	1	1	Cyan
1	0	0	Red
1	0	1	Magenta
1	1	0	Yellow
1	1	1	White

Table 1: Three Bit VGA Combinations

4.1.5 Sync Signal Generation, Testing and Results

The VGA controller module is synthesized using Xilinx Spartan3 board. The Board details are included in Appendix B Section 1. Xilinx ISE Web Pack is used for the synthesis and the Impact tool that comes with the web pack is used to download the bit stream onto the FPGA Board.

To verify the operation of the synchronization circuit, the rgb signal is connected to the three switches. The entire visible region is turned on with a single color. The eight possible combinations are verified. The HDL code is shown in Appendix A. The Figure 5 shows the experiment results.

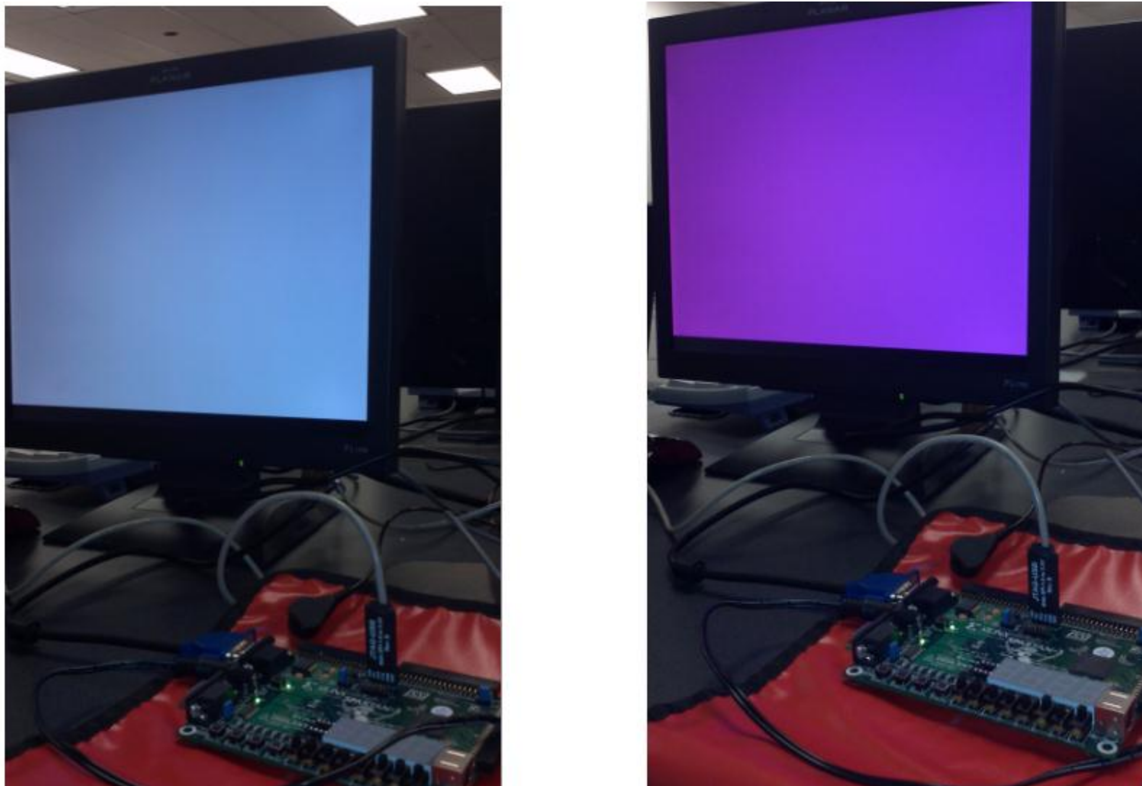


Figure 5: VGA Synch Signal generator Test Results - Screen bitmapped to unique color input read from Switches

4.1.6 Bitmap generation logic using Block RAM

The pixel generation circuit generates the 3-bit rgb signal for the VGA port. The external control and data signals specify the content of the screen, and the pixel_x and pixel_y signals from the vga_sync circuit provide the current co-ordinates of the pixel. In a bit-mapped scheme, a video memory is used to store the data to be displayed on the screen. Each pixel of the screen is mapped directly to a memory word, and the pixel_x and pixel_y signals form the address. The display processing circuit continuously reads the video memory and routes the data to the rgb signal.

There are 310k pixels in a 640-by-480 screen. This translates to 930k bits for color displays. For S3 board memory is available from external SRAM chips and FPGA's embedded block RAM. A brief explanation about the block RAM configuration is presented in the following section. Detailed explanation of the SRAM memory controller and the interfacing details are given in 0.

Using Block Ram as Video Memory to display Image on VGA Monitor

A block RAM is a special memory module embedded in FPGA device separated from regular logic cells. It can be thought of as a fast SRAM wrapped by a synchronous configurable interface. Hence no additional memory controller is required. It can be configured as either

single ported or dual ported. Each block RAM consists of 16K by 1 to 512 by 32. The Spartan 3 XC3200 device has 12 block RAMs, totaling 192K data bits.

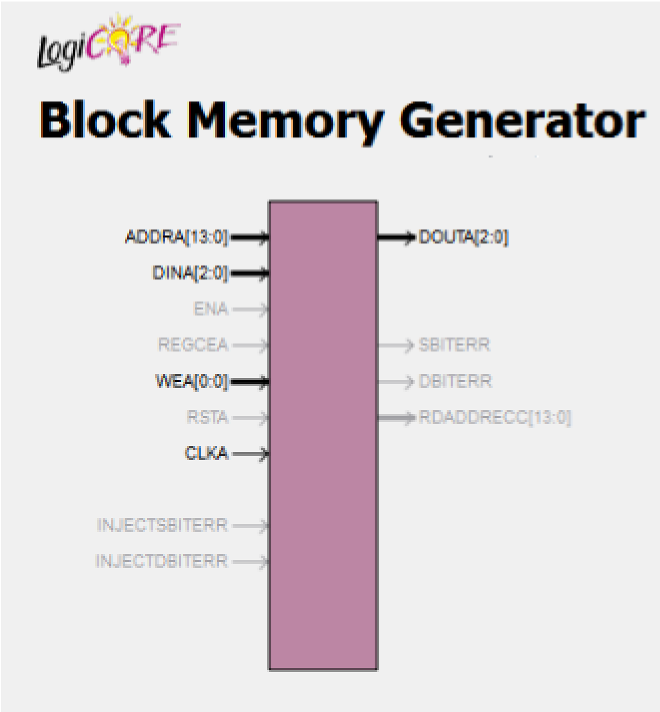


Figure 6: Block RAM as a Component - Interface Diagram

The Xilinx provided utility program, known as Core Generator (Coregen) is used to generate single ported Block RAM. In this application a small 128-by-128 area of the screen is only utilized for Bitmapped scheme. The screen has 16K pixels in this area and requires a 16K-by-3 video memory for color display which is configured using Block RAM. The image size selection is limited by the availability of the Block RAM memory.

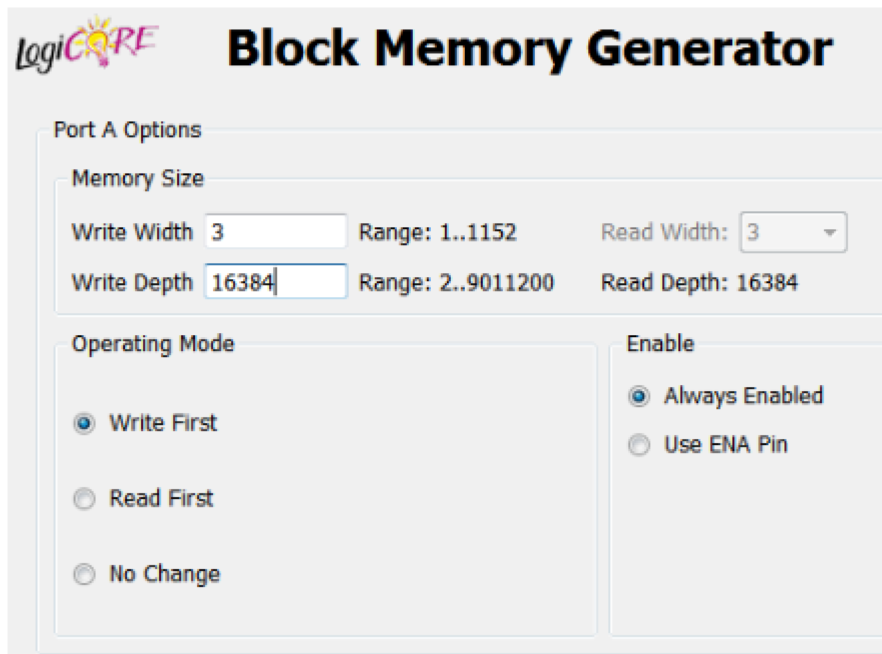


Figure 7: Configuring Block RAM

COE generation steps

A 24-bit sample picture is taken. It is converted to a 3bit format which can be used with S3 FPGA board. The code snippet is shown in Appendix A. The bit file is converted to .coe file format which can be used to initialize the block RAM.

1. Identified and downloaded a sample picture from the net.

2. Opened it in MsPaint and saved it as a 128x128 size image in 24bit RGB format. This is a standard format that has 8 bits for R, G & B.

3. Ran the application (COE generation app).

   a. The application uses the built in Bitmap class in C# to open and read each pixel of the bitmap

   b. For each pixel, look at the MSB of the R, G & B values and write them to a text file.

   c. Add appropriate commands to the text file to make it in .coe file format.

The Verilog code for the VGA sync signals generator module and the bit map generator module using Block RAM as video memory are shown in Appendix A.

4.1.7 Testing and results

The Verilog code for the testing circuit is shown in Appendix A. The vga_synch unit generates the timing and control signals. The bitmap generation circuit is written in such a way that the VGA monitor is continuously refreshed at 60 Hz with the image embedded in the Block RAM.

The resulting display is shown in Figure 8.

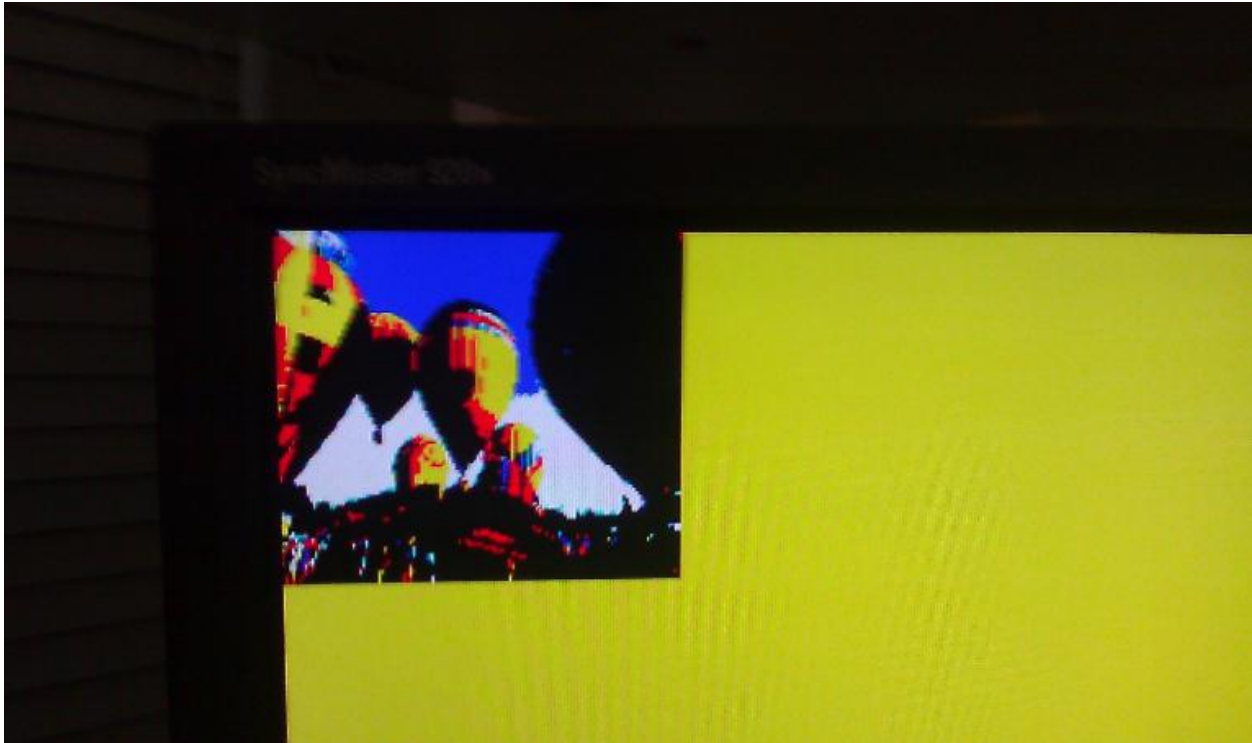


Figure 8: Image from FPGA Block RAM

4.1.8 Generating 16-bit VGA Color Output

The VGA port on the Spartan 3 board provides only 1 bit each for R, G and B color. This allows for only 8 possible color combinations which results in the displayed picture appearing very different from the original source. Hence we decided to create a custom VGA port with increased color depth. Most video controller hardware typically supports at least 16 bits of color and so we decided to keep the color depth of the custom VGA port to be 16 bits. The 16 bit color is represented in the RGB565 format, which appears to be a common image format for hardware representation. In this representation, Red is presented by 5 MSB bits, followed by 6 bits of Green and 5 LSB bits representing Blue. Hence the RGB 565 naming of the format.

VGA DAC

The RGB signal accepted by the color monitor is an analog signal, one for each color, in the range 0V to 0.7V according to the VGA spec. So the digital color signal generated by the video controller should be converted to an analog signal. A Digital-To-Analog Convertor (DAC) convertor is to be used for this purpose. In the built-in VGA port of the Spartan 3 board, there is no need for a separate DAC circuit since there is only one bit per color. The only thing to be done there is to convert the 3.3V logic signal to the voltage range acceptable to VGA. The Spartan 3 board uses a 270 ohm series resistor for each of the color signals. This resistor forms a voltage divider circuit in combination with the 75 ohm load resistance of the VGA monitor. The voltage divider circuit keeps the voltage across the 75 ohm resistance in the 0.7V range required by the VGA spec.

Increasing the color depth of each color to anything higher than 1 requires a separate DAC circuit to convert the digital color values to the analog VGA color signal. We decided to use a simple R-2R ladder circuit as the DAC. R-2R ladder was chosen as it is easy to build, cheaper and simpler to interface compared to dedicated DAC chips. A general 8 bit R-2R ladder circuit is shown in Figure 9

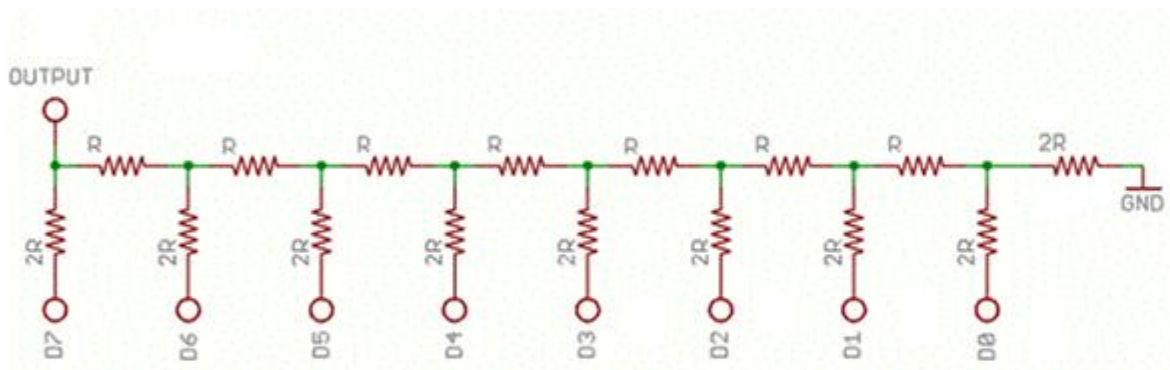


Figure 9: General R2R ladder circuit

Each bit of digital output contributes to the total output of the circuit with a weightage depending on the position of the bit. For example, the MSB will contribute half the voltage output when it is turned on, the next bit gives 1/4th of the voltage and so on. The formula for output voltage thus would be as shown in Figure 10

$$V_{OUT} = \left(\frac{D_7}{2} + \frac{D_6}{4} + \frac{D_5}{8} + \frac{D_4}{16} + \frac{D_3}{32} + \frac{D_2}{64} + \frac{D_1}{128} + \frac{D_0}{256} \right) V_{LOGIC}$$

Figure 10: Digital to analog conversion formula using the R2R ladder network

Typical R-2R ladder DAC circuits like the one shown in Figure 9 contain a voltage buffer at the output so that the output voltage is not affected by load resistance. This is done by adding an operational amplifier circuit in the output stage. However, this brings in an additional design complexity. Since the VGA circuit is operating at a 25 MHz pixel rate, the color values will

change at that rate. So any op-amp connected here should be able to switch that fast – in a matter of a few nanoseconds. General purpose op-amps available in the market have switch times in the range of a few milliseconds and so cannot be used. This calls for extremely fast amplifiers or dedicated DAC chips which complicate the design.

The need for an output amplifier in the R-2R DAC in this application can be eliminated by utilizing the fact that the load resistance of the VGA monitor/cable is known to be 75 ohms. So if we pick the resistor values of the R-2R network accordingly, the desired output voltage can be generated across the 75 ohm load resistor.

Project Implementation of 16 bit VGA output

Hardware

The color output from our new VGA controller design in the FPGA chip has 5 bits for Red, 6 bits for Green and 5 bits for Blue. These are connected to available pins in the expansion connector B1 of the Spartan 3 board by specifying the appropriate net list in the user constraints file for the project (UCF file). The Hsync and Vsync signals are also brought to this connector the same way. An external circuit assembled on a prototype PCB board, as shown in Figure 11, contains the R-2R ladder for each color – a 5 leg ladder for Red, a 6 leg ladder for Green and a 5 leg ladder for Blue. It has a 40-pin male header connector on one end and a VGA connector on the other end. This prototype board can be plugged in directly to the B1 connector using the 40-pin header as shown in Figure 11

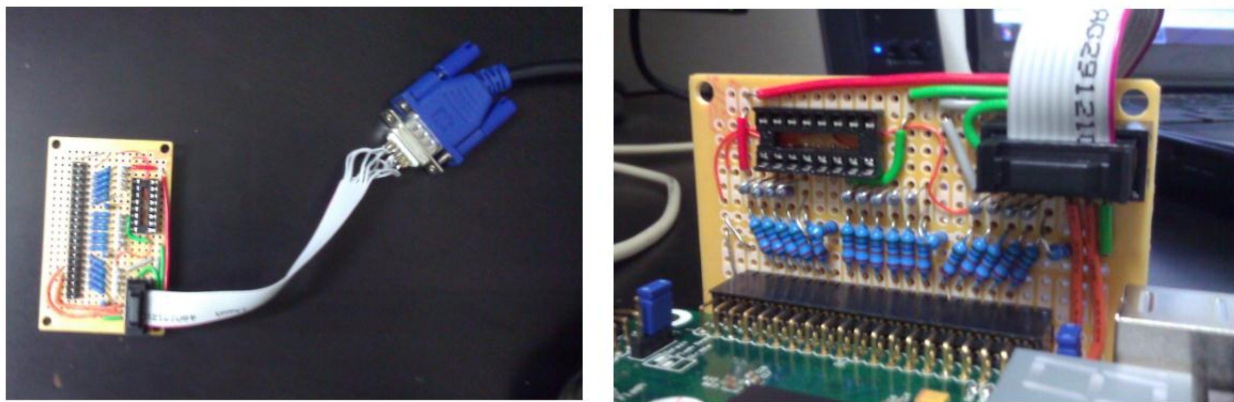


Figure 11: Circuit board - Custom built 16bit VGA port

The calculation of the resistor values for R and 2R is based off the same requirement from the VGA spec that the voltage of each color signal should be in the 0V to 0.7V range. The logic high voltage on the FPGA IO pins is 3.3V. So the maximum voltage output of the R-2R network is 3.3V when all bits are high. In order to make this 3.3V output divided down to 0.7 V across the 75 ohm resistor, the effective resistance of the R-2R network should be 270 ohms, as we saw in the 3 bit VGA case (where each bit was directly connected to the VGA port using a 270 ohm

resistor). Effective resistance of a R-2R network is R, so we get the value of R as 270 ohm and 2R as 540 ohm.

Note: It was difficult to procure 540 ohm resistor, so a 520 ohm resistor was used in its place. The measured voltage output did not make a big difference to make it a problem.

HDL Implementation changes

Switching from 3 bit VGA output to 16 bit output requires a few changes in the HDL code as well, apart from the external DAC circuit. The following are the changes:

1. A new Block RAM needs to be generated, which is 16 bits wide. Now that each pixel is 16 bits wide, we cannot accommodate a 128x128 pixel image in the block RAM available in Spartan 3. Given that the total BRAM is 192K bits, the maximum image size is now reduced to 128x96 pixels, with each pixel being 16 bits wide. So a new block RAM is generated to be of 16 bits wide and 12288 deep (128 * 96), totaling 128*96*16 = 192K bits.

2. The RGB port and the associated register variables of the pixel generation module is expanded to 16 bits wide.

3. The user constraints file is modified to connect the appropriate bits of R, G and B signals, along with Hsync and Vsync signal to the B1 expansion port of the FPGA board.

The output from circuit is shown in Figure 27 and Figure 28. This shows the output with SRAM as video memory as the SRAM controller was already developed by the time the external DAC board was built.

4.2 Memory Controller

The second major work of our project is the implementation of SRAM memory controller for the S3 FPGA board. First, the specification of the onboard memory is studied to build a memory controller. The following Figure 12 shows the interface of SRAM memory with FPGA and the interfacing signals.

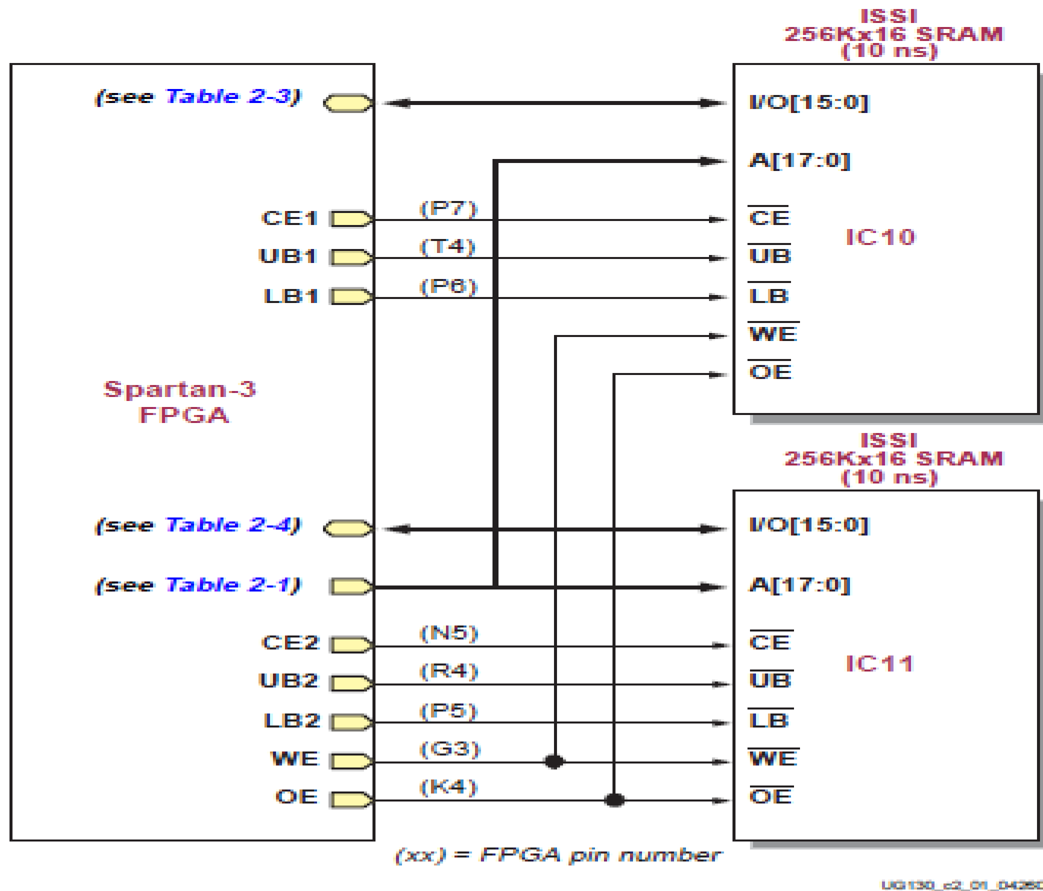


Figure 12: Interface between SRAM memory and FPGA

The Spartan S3 FPGA board has a megabyte of fast asynchronous SRAM, which is surface mounted. It has two 256K*16 SRAM devices. These devices shares common write-enable signal, output-enable signal and address signal. But each has a separate chip select. The address bus is 18 bits and I/O signal bus is 16 bits. Chip enable, output enable, write enable are active low signals.

4.2.1 Role of SRAM Memory Controller

A memory controller has been designed for the SRAM chips on the Spartan 3 board. As the timing characteristics of each RAM devices are different, the controller is applicable only to this particular device. Memory controller is used as an interface, which takes commands from main

system synchronously and then generates properly timed signals to access SRAM. The controller makes memory access appear like synchronous operation by protecting the main system from the detailed timing. Performance of a memory controller is measured by number of memory access that can be completed in a given time. Designing a simple memory controller is straightforward but achieving optimal performance is very difficult as it involves many timing issues.

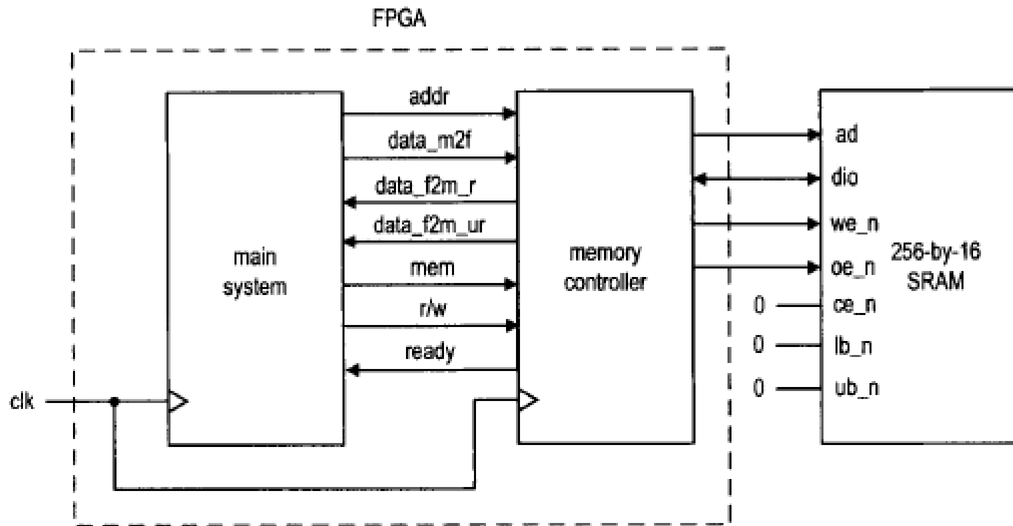


Figure 13: Role of SRAM as Memory Controller

The role of memory controller and its I/O signals are shown in Figure 13. The signals to main system side are:

- mem: is asserted to 1 to initiate a memory operation.
- rw: specifies whether the operation is a read (1) or write (0) operation.
- data_f2s: is the 16-bit data to be written to SRAM (the _f2s suffix stands for FPGA to SRAM).
- data_s2f_r: is the 16-bit registered data retrieved from SRAM (the _s2f suffix stands for SRAM to FPGA).
- data_s2f_ur: is 16-bit unregistered data retrieved from SRAM.

- ready: is a status signal indicating whether the controller is ready to accept a new command. This signal is needed since a memory operation may take more than one clock cycle.
- ce_n (chip enable): disable or enable the chip
- we_n (write enable): disable or enables write operation.
- Oe_n (output enable): disables or enables the output.
- Lb_n (lower byte enable): disable or enable lower byte of the data bus.
- Ub_n (upper byte enable): disables or enables the upper byte of the data bus.

Those signals with '_n' are active low signals.

4.2.2 Block Diagram of Memory Controller

The memory controller basically provides a synchronous wrap around the SRAM. When the main system wants to access memory, it places the address and data (for write operation) on the bus and activates the command (i.e. the mem and rw signals). At the rising edge of the clock, all signals are sampled by memory controller and the desired operation is performed accordingly. For a read operation, the data becomes available after one or two clock cycles.

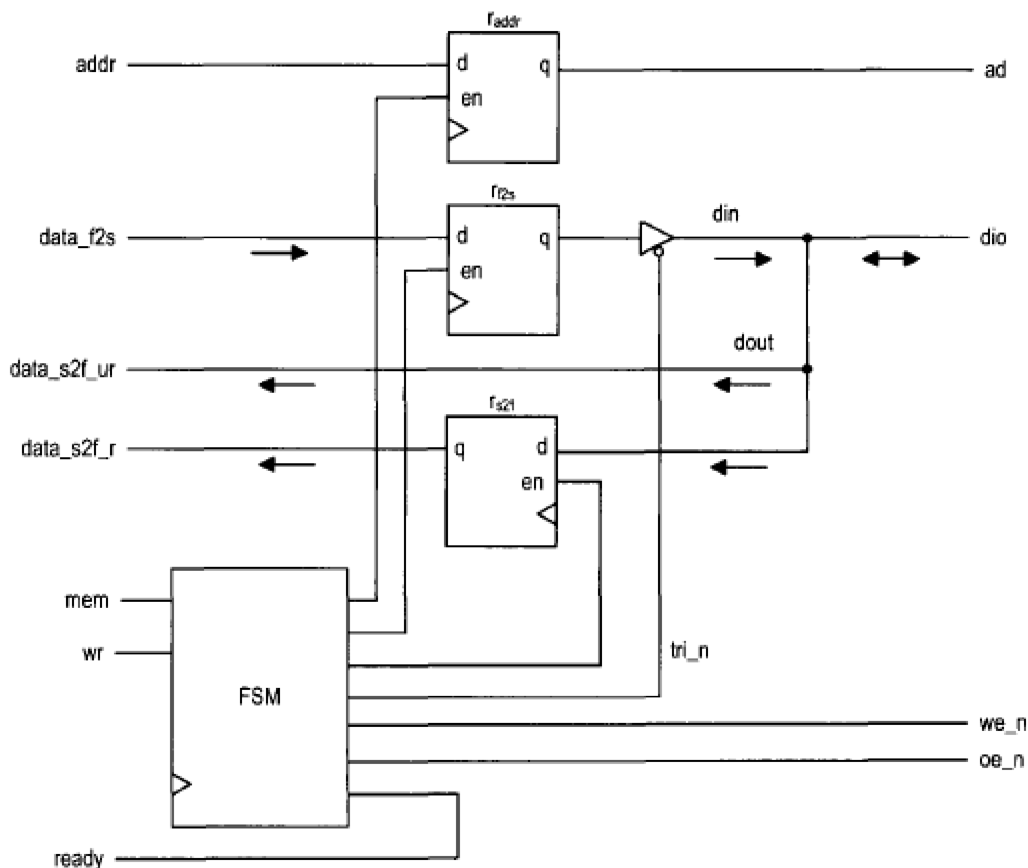
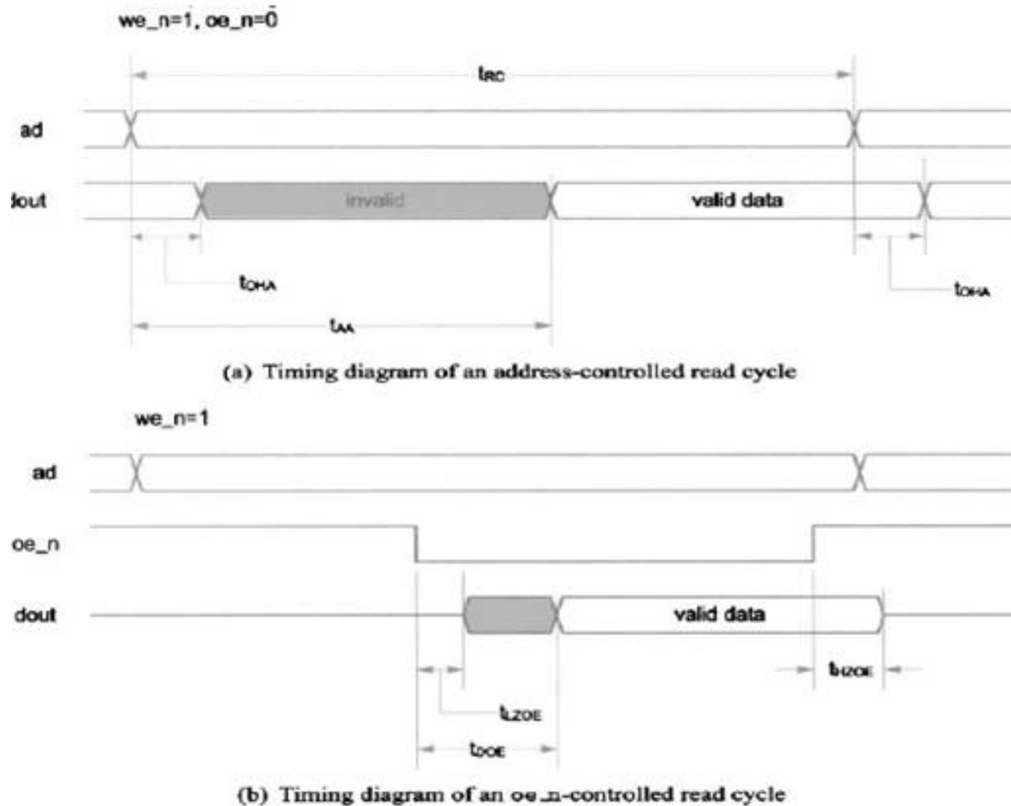


Figure 14: Block diagram of a memory controller

The block diagram of the memory controller is shown in Figure 14. Data path contains one address register, which stores the address and the two data registers, which stores the data from each direction. Since dio is a bidirectional signal, a tri-state buffer is needed. The control path is an FSM, which follows the timing diagram to generate the proper control sequence.

4.2.3 Timing Requirement

The timing diagram and parameters of a read operation and write operation are shown in Figure 15 and Figure 16 respectively.



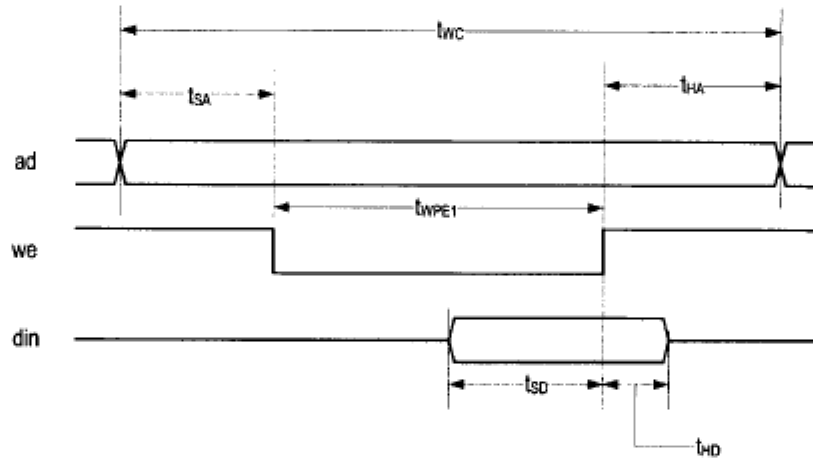
(a) Timing diagram of an address-controlled read cycle

(b) Timing diagram of an oe_n-controlled read cycle

parameter		min	max
t_{RC}	read cycle time	10	–
t_{AA}	address access time	–	10
t_{OHA}	output hold time	2	–
t_{DOE}	output enable access time	–	4
t_{HZOE}	output enable to high-Z time	–	4
t_{LZOE}	output enable to low-Z time	0	–

(c) Timing parameters (in ns)

Figure 15: Timing parameters of Read operation



(a) Timing diagram of a write cycle

parameter		min	max
t_{WC}	write cycle time	10	–
t_{SA}	address setup time	0	–
t_{HA}	address hold time	0	–
t_{PWE1}	we_n pulse width	8	–
t_{SD}	data setup time	6	–
t_{HD}	data hold time	0	–

Figure 16: Timing parameters of a write operation

The control sequences are discussed below.

Read Cycle

The we_n should be deactivated during the entire operation. Its basic operation sequence is:

1) Place the address on ad bus and activate the oe_n signal. These two signals must be stable for the entire operation
2) Wait for at least tAA. The data from SRAM becomes available after this interval.
3) Retrieve the data from dio and deactivate the oe_n signal.

Write Cycle

We use the we_n controlled write cycle in our design. The basic operation sequence is:

1) Place the address on the bus and data on the dio bus and activate the we_n signal.
2) Wait for at least tPWE1.
3) Deactivate the we_n signal. The data is latched to SRAM at the 0-1 transition edge.

4) Remove the data from dio bus.

4.2.4 Design 1: Safe Design – Back-to-back memory access time 60ns.

ASMD CHART OF SRAM CONTROLLER

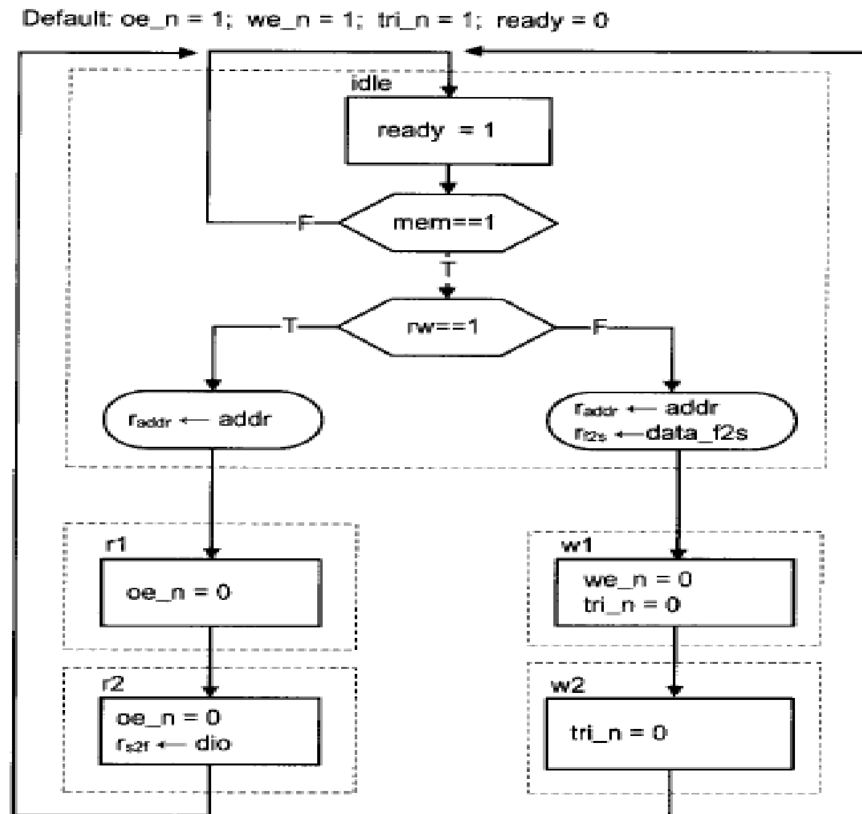


Figure 17: ASMD chart of SRAM Controller

The FSM has five states and is initially in the idle state. In this state oe_n =1, we_n = 1, tri_n = 1, ready = 0.

Read Operation

FSM moves to rd1 state. The memory address, addr is sampled and stored in the addr_register at this transition. Oe_n signal is activated in the rd1 and rd2 states. At the end of read cycle, FSM returns to the idle state. For write operation, the FSM moves to the wr1 state. The memory address addr, and data, data_f2s are sampled and stored in the addr_reg and data_f2s_reg registers at the transition. The we_n and tri_n signals are both activated in the wr1 state. The latter enables the tri-state buffer to put the data over the SRAM dio bus. When FSM moves to wr2 state, we_n is deactivated but tri_n remain asserted. This ensures that the

data is properly latched to the SRAM when we_n changes from 0 to 1. At the end of write cycle, the FSM returns to the idle state and tri_n is deactivated to remove data from dio bus. This FSM meets the above mentioned timing requirements. HDL Implementation is given in Appendix A.

As this is a safe design, back to back memory access takes 3 clocks or 60ns with 50MHz board clock which makes memory operation very slow. Another approach is used to get a fast clock to reduce memory operation timing.

4.2.5 Design 2: Using Xilinx ISE DCM for Faster Access Time

The safe design initially discussed works with 50MHz clock. Basically, an FSM cannot generate a control sequence finer than the period of its clock. An effective design with much less back to back memory access operation i.e. 20 ns can be used instead. DCM (digital clock manager) circuit of the Spartan-3 can be used as a solution to the problem. DCMs provide advanced clocking capabilities to Spartan 3 FPGA applications. There are 8 digital clock managers in a Spartan 3 FPGA device. DCM frequency synthesizer is used in our design to generate a 200 MHz clock. A basic block diagram of the DCM is shown in Figure 18. Now each clock cycle is only 5ns. This requires additional states in the read and write operations to meet the timing requirements. The new ASMD chart is given in Figure 21.

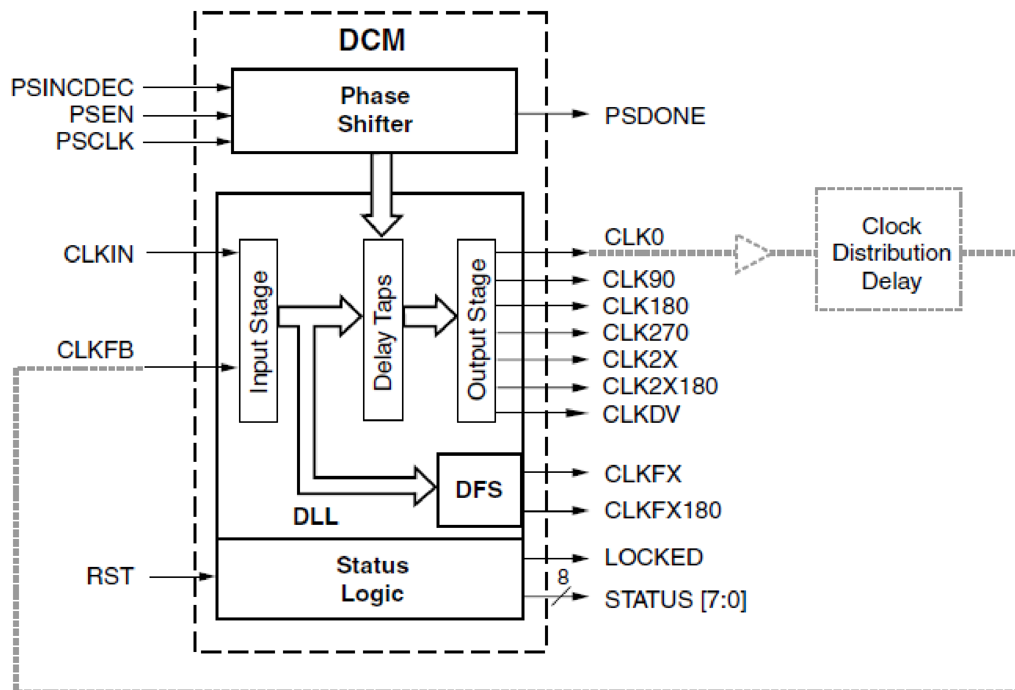


Figure 18: Block diagram of a Digital clock Manger

Spartan 3 provides 3 independent frequency synthesis functions. Among the three, a frequency synthesizer (CLKFX, CLKFX180) for generating a completely new frequency from an incoming clock frequency is used. Functional diagram of various clock synthesis options using the DCM is given in Figure 19

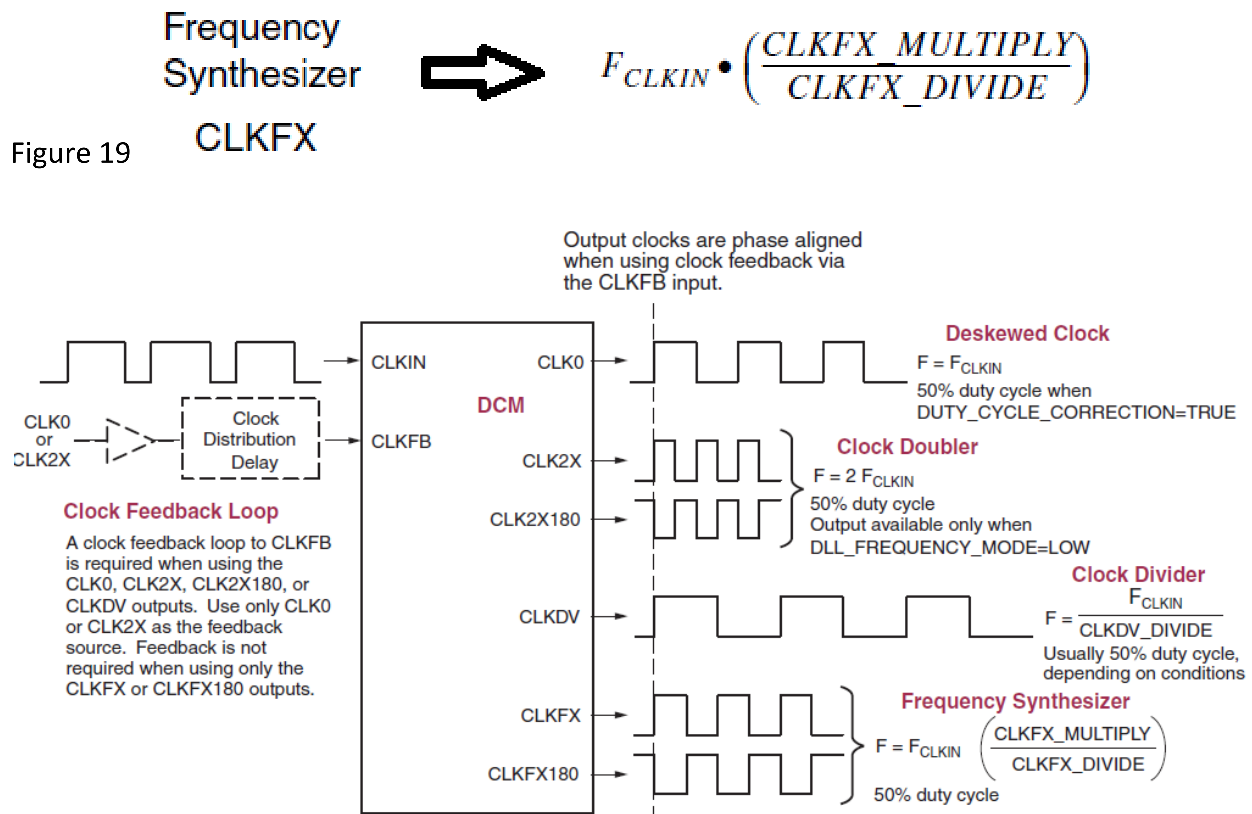


Figure 19: Clock Synthesis Options with DCM

In our application 50MHz board clock is used as input frequency. CLKFX_MULTIPLY value is set as 4 and CLKFX_DIVIDE value is set as 1 to generate 200 MHz clock.

To simplify applications using DCMs, the Xilinx ISE development software includes a software wizard that provides step-by-step instructions for configuring DCM. As shown in Figure 20 DCM wizard generates vendor specific logic synthesis file instantiating the DCM in either verilog or VHDL syntax. Similarly, DCM wizard also generates a user constraints file for the specific implementation. Finally all user specifications are saved in the Xilinx Architecture Wizard (XAW) settings file.

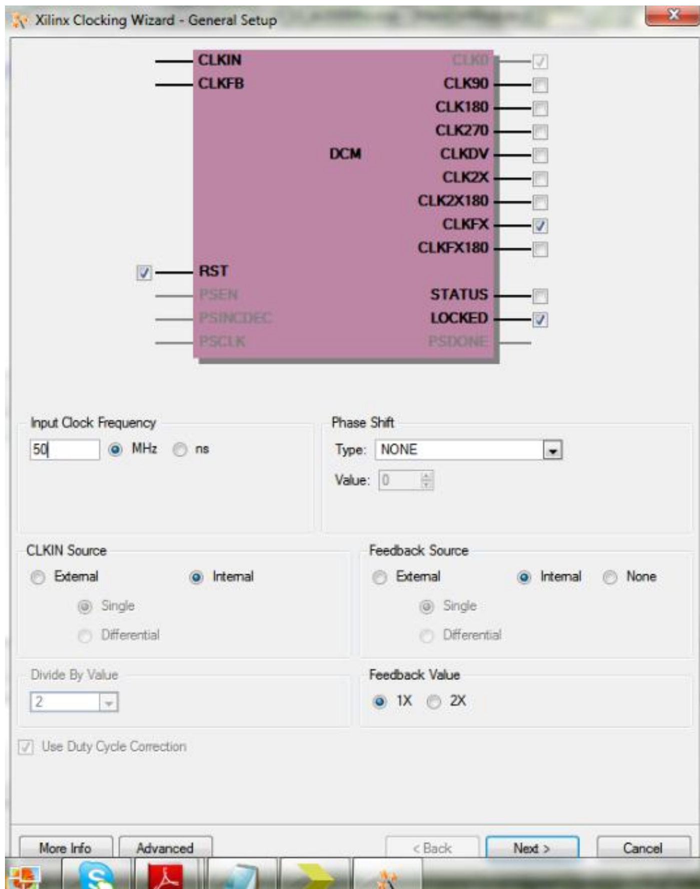


Figure 20: DCM as a component: Interface details

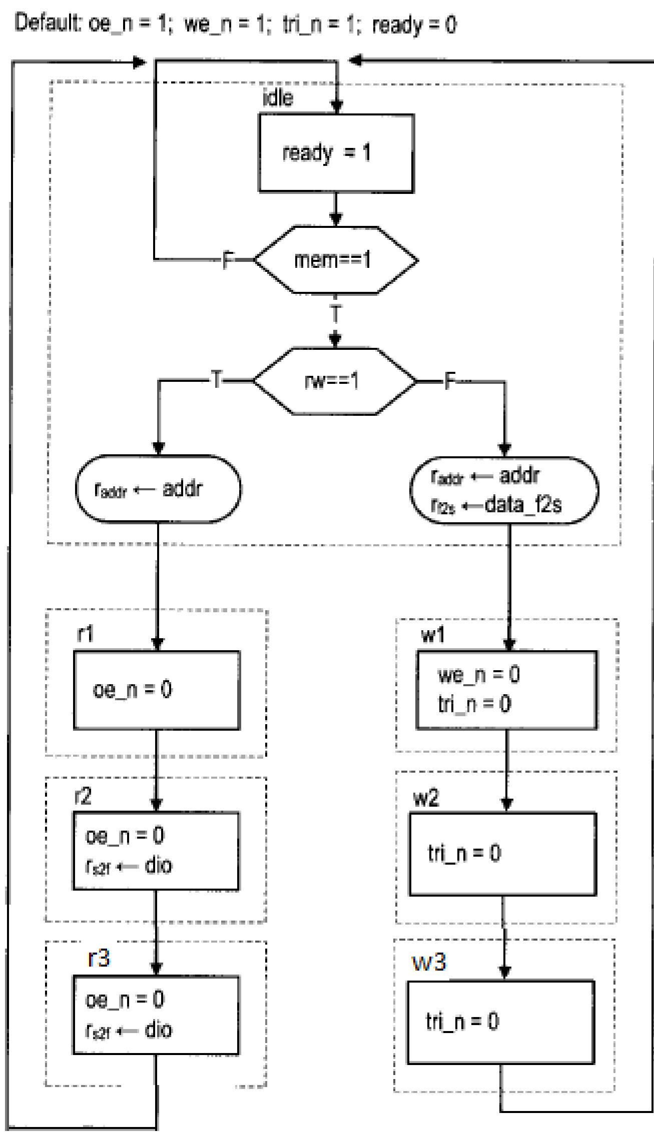


Figure 21: Improved Design with 200MHz clock satisfying timing constraints

4.2.6 Testing and Results

A basic circuit is used to test SRAM controller. It manually tests a single read and write operation. In addition to the SRAM I/O signals, the circuit has the following signals.

sw: 8bit wide and used as data.

led: 8bit wide and used to display the retrieved data.

btn[0] : when this button is pressed, the switch status is read as the data to be written onto SRAM.

btn[1] : When it is asserted, the current value of the switch is written to a hardcoded address in the memory.

btn[2] : When it is asserted, the controller uses the value of the address specified as the read address and performs a read ooperation. The read data is routed to the led signal.

The code for the basic SRAM testing circuit is given in Appendix A.

Experimental Results

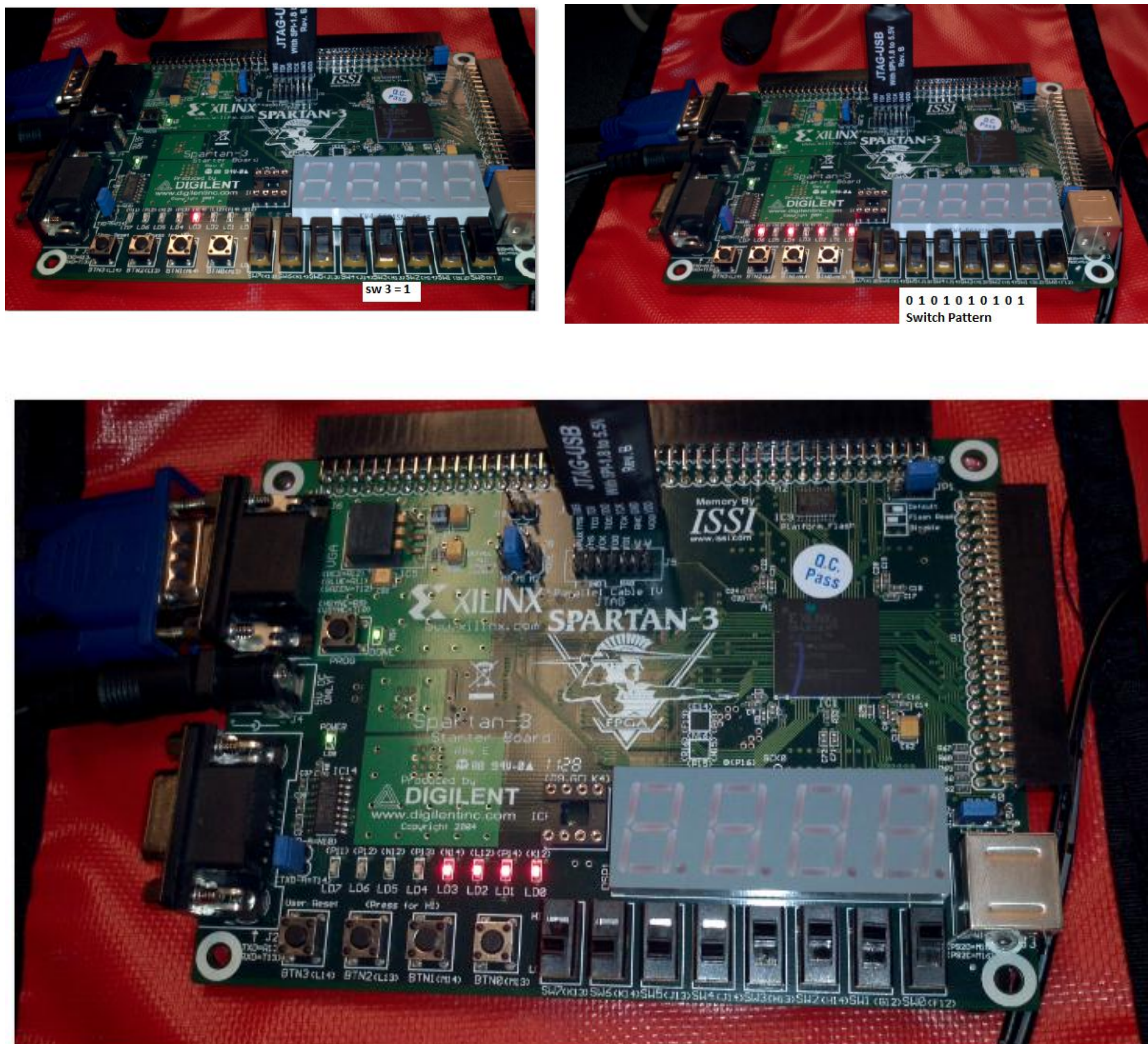


Figure 22: SRAM controller test results

The switch inputs data is read during first button press. The second button press initiates the write operation and the the third button press caused the read operation. The read data is displayed in the leds. The results are shown in Figure 26

4.3 Complete Display System Design

The individual modules are designed and tested. The next step is to combine them to build the complete system. An overview of complete display system is shown in Figure 23. The FPGA development board has the clock frequency 50MHz. The DCM takes in the board clock and generates 200MHz clock. This high speed clock is applied to Block RAM as well as Memory Access Module. This implies that the SRAM write operation from Block RAM to on board SRAM memory and SRAM read operation from SRAM to VGA happens at a high clock speed of 200 Mhz. As we discussed earlier, it is now entirely possible to have a back to back memory operation of 20ns.

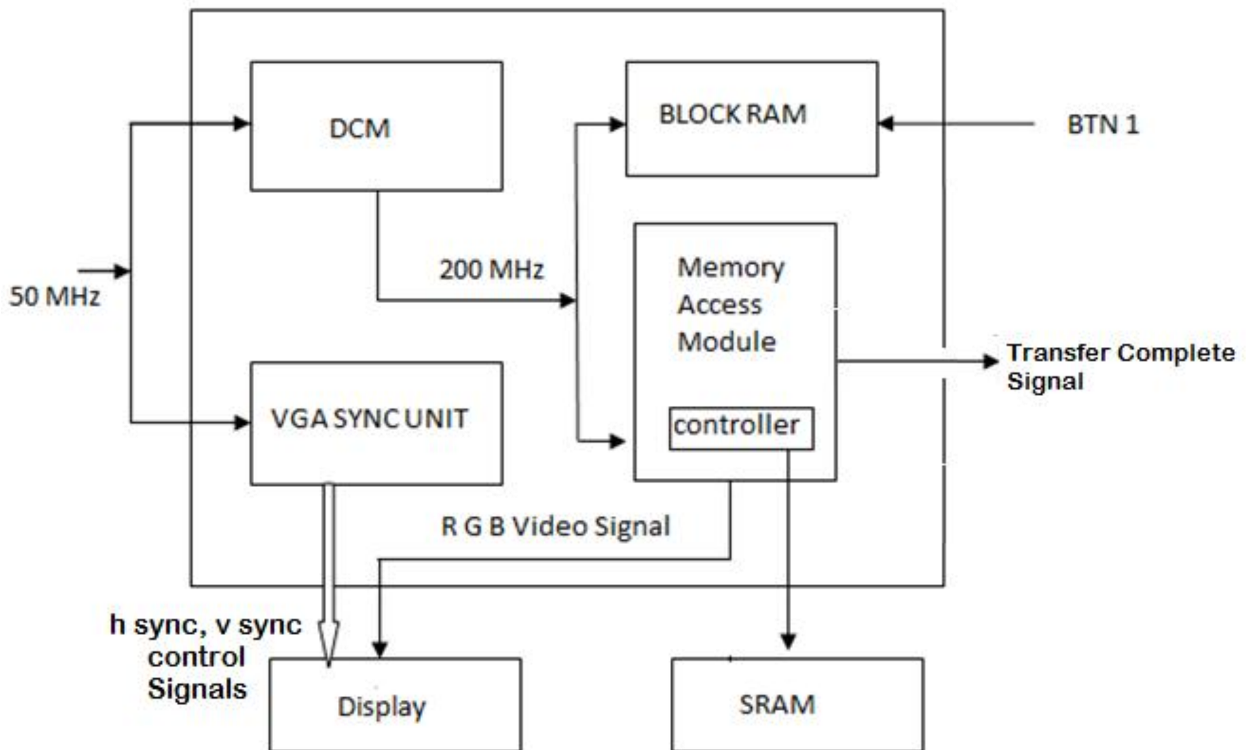


Figure 23: Display with SRAM as Video Memory

This design involves two finite state machines which are explained in detail in the following sections. The image is initialized into the Block RAM. First FSM that copies the image from block RAM to SRAM memory, waits for user input: 'BTN press' to initiate the process. When the image transfer process is complete, a flag 'TransferComplete' is raised. The second FSM that displays the image from video memory starts running as soon as it receives the

TransferComplete signal.

HDL Implementation: Transferring Image from Block RAM to SRAM

Initially the FSM is in idle state. The write cycle starts whenever the button press event happens in idle state. The FSM state transition takes place at 200 MHz frequency which translates to 5ns for each state in the finite state machine. The core unit of the memory access module is the memory controller. Write1, write2, write3, write4 states of this FSM corresponds to the four memory write cycles of the memory controller. During write 1 state, the memory write operation is initiated by setting up the SRAM memory write control signals. The control signals are reset when it reaches the write2 state. These four states all together account for a back to back memory write operation of 20ns and satisfies all SRAM write timing constraints that we discussed earlier. In write4 state all the address manipulation takes place. If image transfer is complete, FSM sets the transfer complete flag and move to idle state. If the process is incomplete, it goes back to write1 state to transfer next pixel to SRAM.

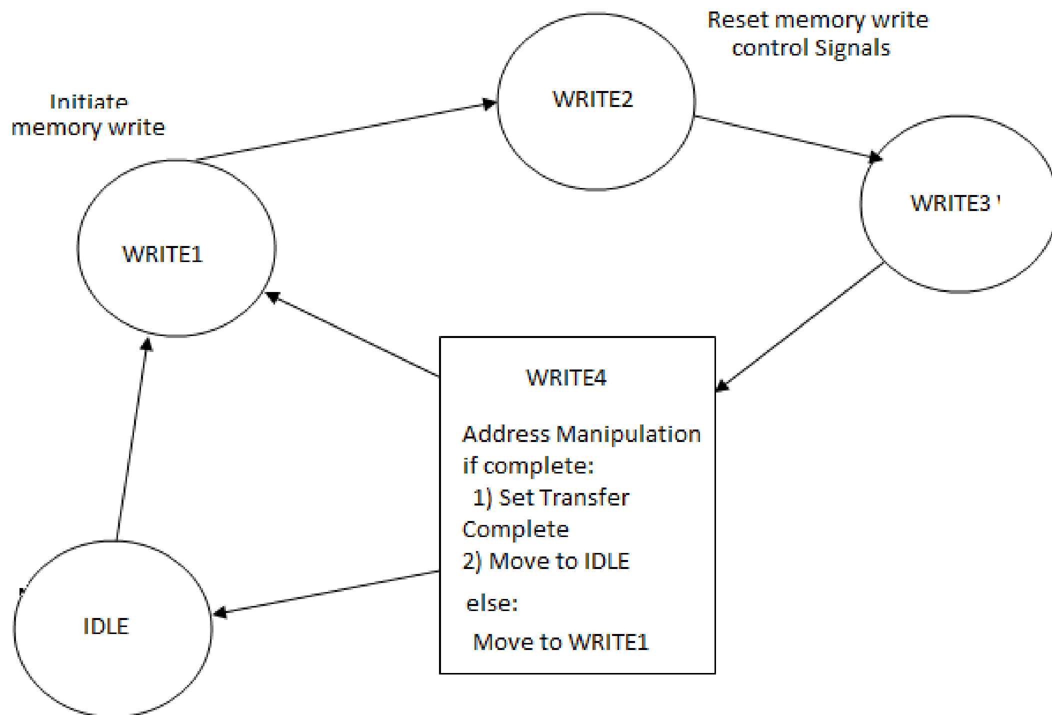


Figure 24 FSM for transferring image from Block RAM to SRAM

HDL Implementation: Displaying image from SRAM

The FSM that performs the display operation gets activated on ‘transfer complete’ signal raised by the previous FSM. If there is a valid pixel clock at that time and video_on signal from the video controller is high, the FSM moves to read state where one pixel value is read from the

video memory and displayed on monitor. Once the FSM is in READ state, it continuously feeds the display module.

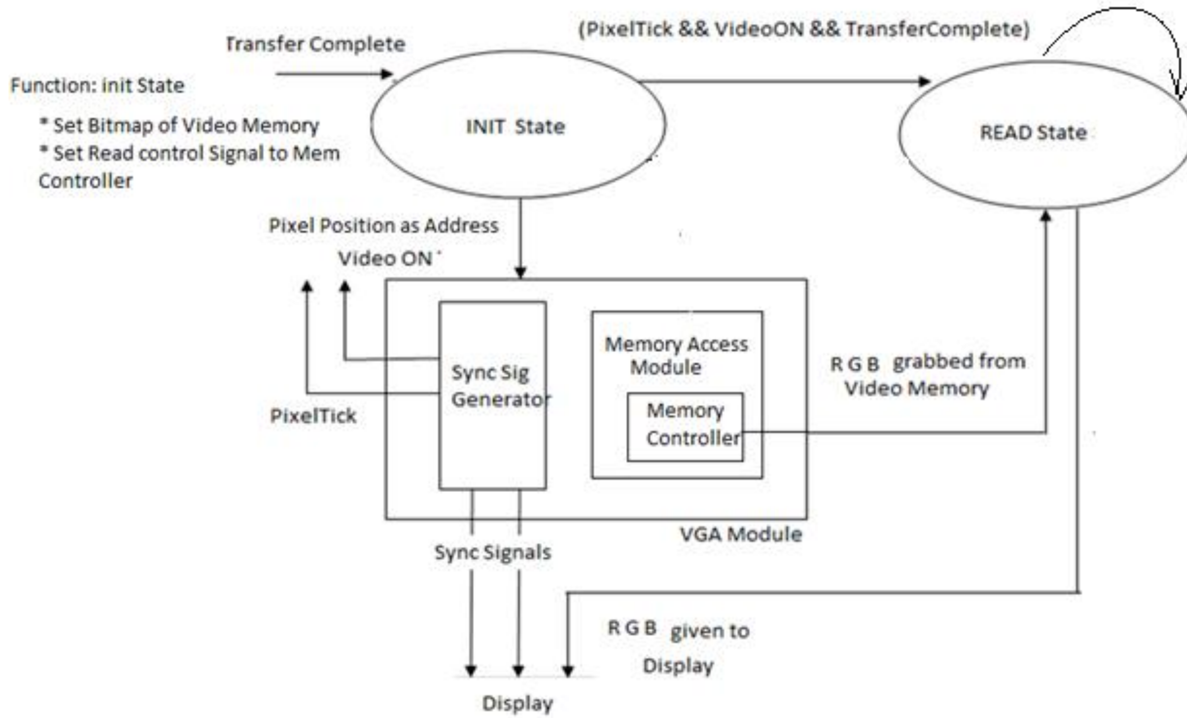


Figure 25: Block diagram for Displaying image from SRAM

4.3.1 Experiment Results

The VGA monitor is continuously refreshed at 60 Hz with the image embedded in the Block RAM. The resulting display is shown in Figure 26 and Figure 27

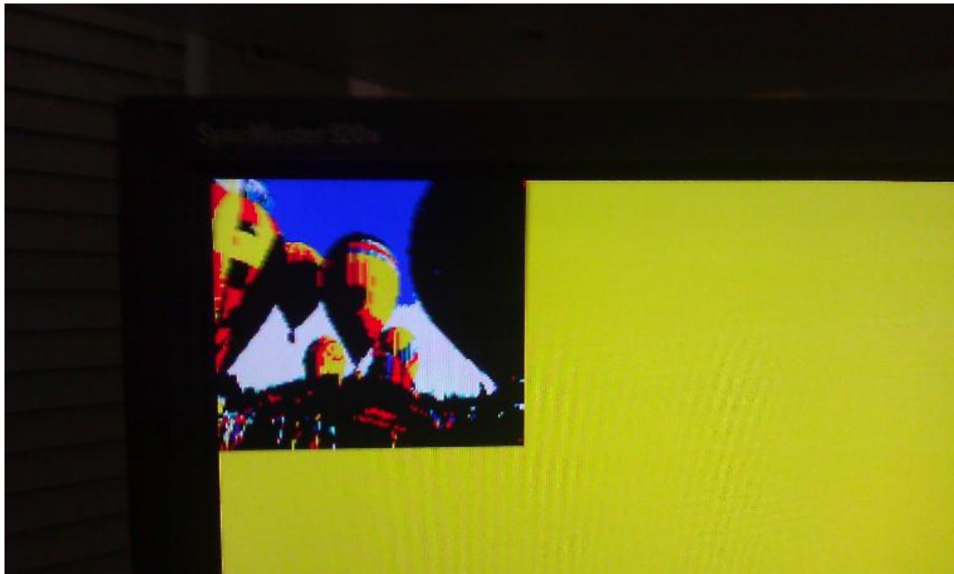


Figure 26: Image displayed from Block RAM with 3bit color resolution

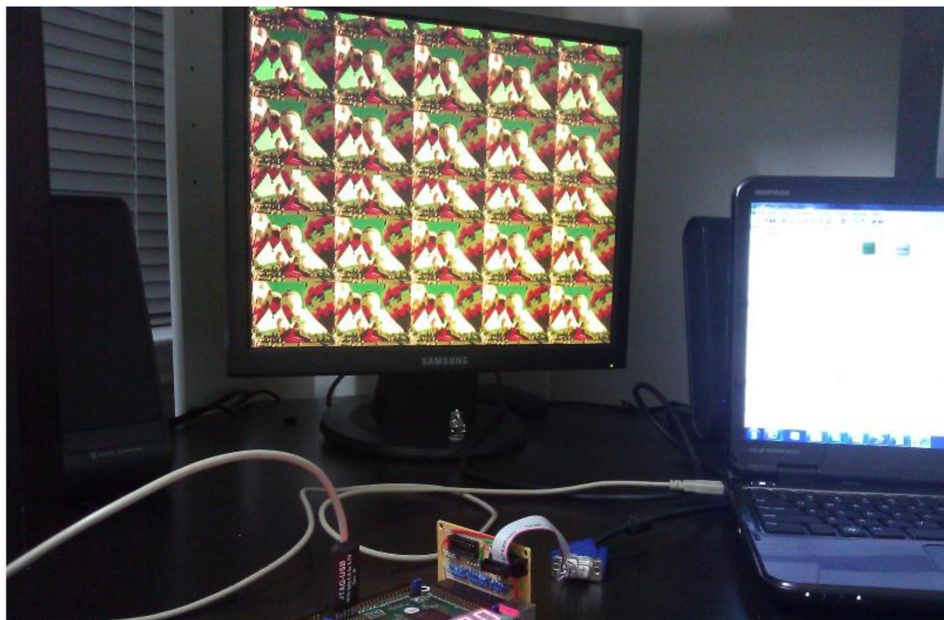


Figure 27: Image displayed from SRAM with 16 bit resolution

Figure 28: Close up of the Image displayed using the 16 bit custom VGA adapter

5. Image Processing

The main objective of this section is to develop and simulate an algorithm for Blob detection. Sequential Connected Component Algorithm was chosen for blob detection. Connected component analysis typically requires four stages in any algorithm. First, the input (color or grayscale) image is preprocessed through filtering and thresholding to segment the objects from the background. The preprocessed image is usually binary. Next, each connected group of pixels is assigned a unique label, enabling the distinct objects to be distinguished. In the third stage, each region is processed (based on the label) to extract a number of features of the object represented by the region. In the final stage, these features are used to classify each region into one of the two or more classes. The classic connected component labeling algorithm requires two passes through an image. But the algorithm implementation discussed in the following sections allows the connected components to be analyzed in a single pass.

Section 5.1 lists the challenges faced by image processing implementations that gave motivation for us to come up with an effective algorithm implementation. The basic concept of blob detection and calculation of the desired parameter is described in section 5.2. The sequential component labeling process is explained in section 5.3. After the labeling process, the blob parameter calculation with the help of Blob Parameter table is explained in section 5.4. Section 5.5 lists the simulation results.

5.1. Design Considerations

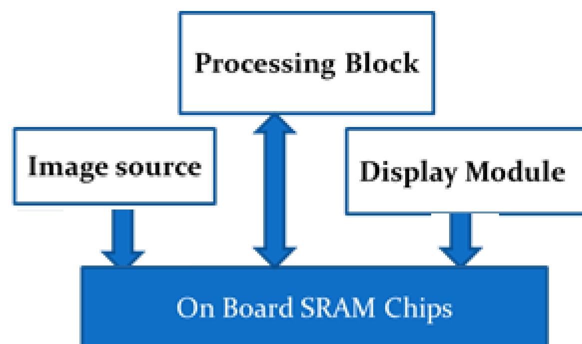


Figure 29: Memory Contention

All the traditional image processing algorithms are sequential in nature with multiple stages. This makes the real time image processing really tough. All the robotics applications require extremely fast detection processing and response. We also focused at reducing the huge image

storage demand. The traditional processing requires storage for source image, processed image and for the intermediate image. The algorithm should also work around the memory access latency. The key focus is to reduce the memory contention. In any image processing application, the image source requests memory access for storage. The processing blocks accesses the memory for each pixel processing. Simultaneously the Display module accesses the memory at very high pixel rate for displaying the processed image. We can see that memory is heavily strained and any algorithm design should be optimized to minimize the scheduling issues. Taking into account all the above mentioned design constraints, we chose to come up with an algorithm that has minimum storage demand and minimum number of passes for processing.

5.2. Basic Concept of Blob Analysis

A blob (binary large object) is an area of touching (connected) pixels with the same logical state. A picture frame might contain multiple blobs and they are said to be in foreground state. All other pixels are in background state. All pixels in the background have values equal to zero while every nonzero pixel is part of a binary object. In this project one particular color filter is used to separate blob. In the following discussion we assume that we got a preprocessed (color filtered) binary image and the algorithm is applied on this image. The objective is to calculate only two blob parameters – area and centroid. The formula for calculation is listed below.

$$\text{Area} = \sum_{i=1}^{n} \sum_{j=1}^{m} B[i,j]$$

$$\bar{x} = \frac{\sum_{i=1}^{n} \sum_{j=1}^{m} jB[i,j]}{A}$$

$$\bar{y} = \frac{\sum_{i=1}^{n} \sum_{j=1}^{m} iB[i,j]}{A}$$

After the labels merging process, total count of pixels with the same label gives the area of the blob. B[i,j] represent the non-zero valid pixel that is been labeled. In short, the algorithm is designed to identify connected blobs, calculate the area and centroid of each detected blob.

5.3 Sequential Component Algorithm

Let ‘ROWS’ be the number of rows and ‘COLUMNS’ is the number of columns of the image under consideration. Hence the overall ‘DEPTH’ of the array is ROWS * COLUMNS. The

algorithm is applied to every pixel starting from the first to last pixel in a sequential nature. In general if the pixel is zero, the pixel label is zero. If the pixel value is one, the pixel should be labeled. Hence for a current pixel four values are required for labeling

1. North Neighbor value
2. North Neighbor Label
3. West Neighbor value
4. West Neighbor Label

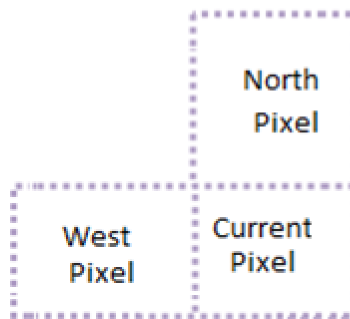


Figure 30: Kernel defined for labeling

The Pseudo code for sequential component algorithm is shown in Figure 31 and an example is illustrated in Figure 32.

```
CCL = 0
if (WestNeighbourValue == 1 && NorthNeighbourValue == 0)
  CurLabel = WestLabel;
if (WestNeighbourValue == 0 && NorthNeighbourValue == 1)
  CurLabel = NorthLabel;
if (WestNeighbourValue == 1 && NorthNeighbourValue == 1)
    if (NorthLabel < WestLabel)
        Need Update;
        CurrentLabel = NorthLabel;
        LabelToMerge  = WestLabel;
     else if (NorthLabel >  WestLabel)
        Need Update ;
        CurrentLabel = WestLabel;
        LabelToMerge  = NorthLabel;
    else
      CurrentLabel = NorthLabel;

if (WestNeighbourValue == 0 && NorthNeighbourValue == 0)
    CurLabel = CCL;
    CCL = CCL + 1;
```

Figure 31: Pseudo code for Sequential Component Algorithm

Current label is calculated based on the North, West Neighborhood values and label based on the rules define by the above listed pseudo code. According to the above logic, after each pixel is labeled, an additional flag will be raised indicating the need for merging labels. How this update is implemented makes the implementation part optimal as far as this design is concerned. The optimized design is explained in the following section

0	0	0	0	0	0	0	0
0	1	1	1	0	1	1	0
0	0	1	1	0	0	1	0
0	0	0	1	0	0	1	0
0	0	1	1	0	0	1	0
0	0	1	1	0	0	1	0
0	1	1	1	1	1	1	0
0	0	0	0	0	0	0	0

Source Image

0	0	0	0	0	0	0	0
0	1	1	1	0	2	2	0
0	0	1	1	0	0	2	0
0	0	0	1	0	0	2	0
0	0	3	1	0	0	2	0
0	0	3	1	0	0	2	0
0	4	3	1	1	1	1	0
0	0	0	0	0	0	0	0

Intermediate Labeled Image

0	0	0	0	0	0	0	0
0	1	1	1	0	1	1	0
0	0	1	1	0	0	1	0
0	0	0	1	0	0	1	0
0	0	1	1	0	0	1	0
0	0	1	1	0	0	1	0
0	1	1	1	1	1	1	0
0	0	0	0	0	0	0	0

BLOB Detected!

Figure 32: Example of CCL Algorithm

5.4 Merging and Blob Parameter calculation

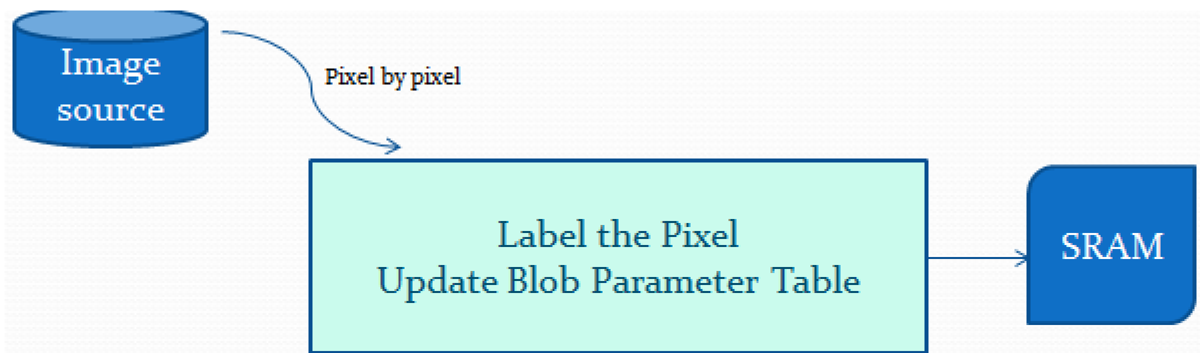


Figure 33: Blob Detection Module

The algorithm implementation only requires the following.

- Line size FIFO referred to as LINE FIFO for intermediate label storage.
- Blob Table.

The usage of the LINE FIFO and the Blob table management for blob parameter calculation makes this scheme different from the traditional schemes which are explained in detail in the following sections. It should be noted that with the availability of these extra FIFO and Table, each pixel is labeled on its way from image source and most of the blob parameters are already calculated by the time single pass is completed. The basic idea is shown in Figure 35.

LINE FIFO Usage

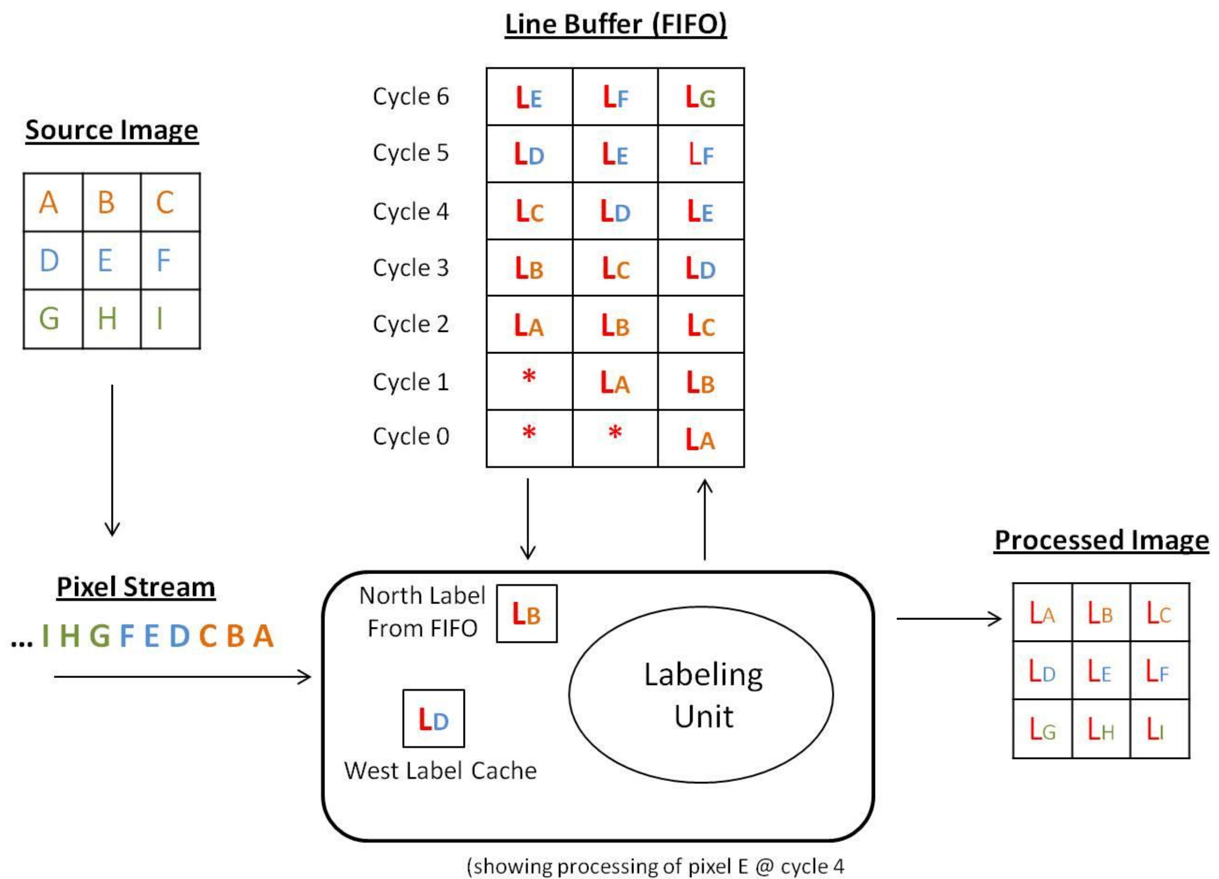


Figure 34: LINE FIFO Usage Example

The line FIFO has a circular queue based implementation. It is a register file with a write pointer and a read pointer. The write pointer points to the head of the queue and the read pointer points to the tail of the queue. The pointer advances one position for each write and read operation. The FIFO HDL code is available in the Appendix.

Consider an image with a line of 3 pixels. Hence the LINE FIFO size is also 3 bits wide. Let the incoming pixel stream be ABCDEFGHI… as shown in figure. LINE FIFO Utilization is explained below.

Defining Neighborhood

A: First Pixel of the image. Hence all neighbor values as well as labels are zeroes

B, C: Members of the first line and their North Neighbor values are zero. Hence the North neighbor labels are also zeroes. Just require west labels

D, G, J: First Pixel of the rest of the rest of the lines. West neighbor is Zero . But there exists valid North Neighbor.

E, F, H, I, K, L: Middle pixels with all valid neighbors.

Operation

Cycle 0: Pixel A arrives -> the labeling unit labels the pixel and the new Labeled pixel ‘LA’ is obtained. The new label LA is written into LINE FIFO and stored into Destination buffer. It should be noted that this acts as west label for the next arriving pixel. Hence it is saved as West Label in a temporary storage.

Cycle 1: LB obtained. Stored in LINE FIFO as well as in Destination buffer.

Cycle 2: LC obtained. Stored in LINE FIFO as well as in Destination buffer.

Cycle 3: LINE FIFO is full in this case. For pixel D, label 'LA' is taken from LINE FIFO and processed. The processed label 'LD' is again fed back to LINE FIFO as well as into Destination buffer.

Cycle 4: LINE FIFO is full in this case. For pixel E, label 'LB' is taken from LINE FIFO and processed. The processed label 'LE' is again fed back to LINE FIFO as well as into Destination buffer.

This kind of LINE FIFO utilization reduces the intermediate storage requirement.

BLOB Table Usage

Blob Table is a collection of one dimensional arrays indexed by the label value defined as follows:

1. Merge Table – described in the next section

2. Area Counter – a running counter of number of pixels in each label

3. Sigma Xbar Counter – a running counter of the sum of X coordinate of each pixel

4. Sigma Ybar Counter – a running counter of the sum of Y coordinate of each pixel

5. Centroid X (calculated by the divider).

6. Centroid Y (calculated by the divider)

A high level picture of blob table utilization is shown in Figure 35. The HDL description in section 5.4 explains the details of Blob Table update and how the table is used for blob parameter calculation.

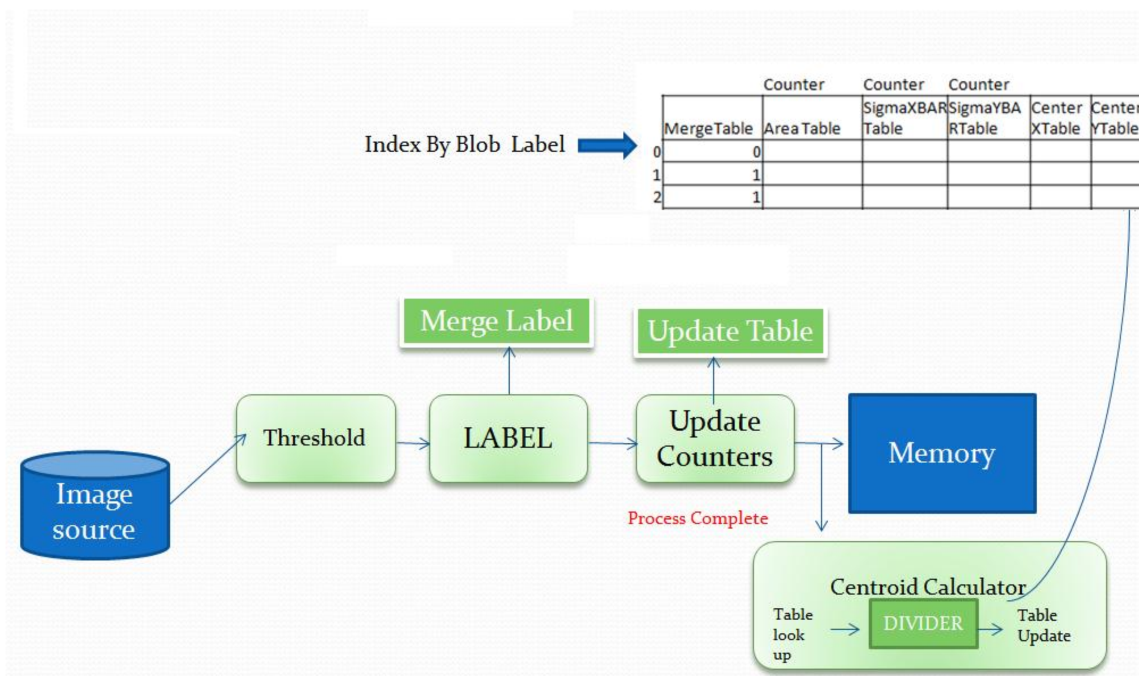


Figure 35: Blob detection using Connected Component Algorithm

Merge Table Usage

Though MergeTable is a part of the overall Blob Table described above, its design and usage require additional description. This is because the design and correct usage of the MergeTable is a key factor that enables the labeling to be done in a single pass.

Initially, the MergeTable is initialized as follows:

MergeTable*[index] = index;*

When a merge occurs as per the algorithm/pseudo code defined in Section X, the merge table is updated as follows:

MergeTable*[LabelToMerge] = CurrentLabel*

For example, consider the pixel in row 5, column 4 in the picture below. In this case, label 3 is going to be merged with label 1 as per the algorithm, since NorthLabel = 1 and WestLabel = 3. The merge table will be updated as:

MergeTable*[3] = 1*

0	0	0	0	0	0	0	0
0	1	1	1	0	2	2	0
0	0	1	1	0	0	2	0
0	0	0	1	0	0	2	0
0	0	3	1	0	0	2	0
0	0	3	1	0	0	2	0
0	4	3	1	1	1	1	0
0	0	0	0	0	0	0	0

Figure 36: Intermediate Labeled Pixel array

The running counters of Area, SigmaXbar and SigmaYbar is updated at each labeled pixel as follows, taking into account that a label could have been merged earlier:

LabelIndex = **MergeTable***[CurrentLabel];*

// The label at the current index is the "merged" label

AreaTable*[LabelIndex] =* **AreaTable***[LabelIndex] + 1*

SigmaXbarTable*[LabelIndex] =* **SigmaXbarTable** *[LabelIndex] + X*

SigmaYbarTable*[LabelIndex] =* **SigmaYbarTable** *[LabelIndex] + Y*

This logic makes sure that once a label has been merged, the running counters are updated only at the merged label index in the corresponding tables.

Lastly, the running counters are updated when a merge happens, as follows

AreaTable*[CurrentLabel] =* **AreaTable***[CurrentLabel] +* **AreaTable***[LabelToMerge]]*

AreaTable*[LabelToMerge] = 0; // needed to avoid double counting.*

Calculating the Centroid Values

The centroid calculation requires dividing the SigmaXbar and SigmaYbar values of a label with the Area associated with that label. Division is a non-trivial operation in hardware. A serial

divider module taken from OpenCores.org website is utilized for this purpose. This divider module is fully parameterizable module that generates 1 bit of quotient output per clock cycle. Since we are not interested in the remainder values, the number of clock cycles can be reduced to the number of bits for each centroid value. Two instances of the serial divider module is instantiated so that the X and Y centroid of each label can be calculated in parallel.

The centroid calculation is done after the labeling is completed for the entire image. Centroid calculation does not require access to the stored image since it only accesses the BlobTable for its calculation. The number of cycles needed for completing the centroid calculation phase is dependent upon the size of the image and the number of blobs. For example, a 8x8 image requires 4 clock cycles to calculate the centroid of each blob. Similarly, a 320x240 would require 9 clock cycles – calculated by the number of bits required to represent the max centroid value – for each blob. This requirement of additional clock cycles for centroid calculation of each blob is negligible compared to the total number of cycles it takes for labeling the image itself (example, 76800 cycle for a 320x240 image – one per pixel), and so the centroid calculation is not considered a second pass on the image.

5.5 HDL Implementation for Simulation

The heart of the implementation is the Finite state machine that sequentially runs through four main Phases – Initialization, labeling, Centroid calculation and storing the processed registers. The algorithms mainly uses the data structures – Source register, Process register, LINE FIFO and Blob table. The initialization phase, storage stage and the use of source registers are only for simulation. In real hardware implementation the image comes from the camera and only the labeling stage and LINE FIFO is required. Each incoming pixel is labeled according to the position of the pixel. States are defined for the special cases such as first pixel of the image, first line of image and not first pixel, first pixel but not the first line and any middle pixel case. The neighborhood is defined in each state and a Get Label task is called to find out the label. After the label calculation, the merging of the labels takes place if needed Merged labels are used to index into rest of the blob table array such as area and x/y bar counters and the update is performed as described in the previous section. The FSM incorporates additional wait states required for the divider to produce a valid result.

For simulation and demonstration purpose only, a second pass is performed on the processed image after all blob parameter calculation is done. In this pass, all the pixels are re-labeled according to the Merge Table values, so that correct operation of the Merge logic can be verified. The HDL description is shown in listing x Appendix.

5.6 ModelSim Simulation Results

The transcript window after the simulation of the algorithm is shown below:

```
# Reading C:/Modeltech_pe_edu_10.1a/tcl/vsim/pref.tcl
# Loading project CCA2
vsim -gui work.CCL_PROC
-gui work.CCL_PROC
# Loading work.CCL_PROC
# Loading work.fifo
# Loading work.serial_divide_uu
run
#                0Completed Initialization of FIFO
# BlobTables AFTER INITIALIZATION
# ---------------------------------
# MergeTable AreaTable SigmaXBARTable SigmaYBARTable CenterXTable CenterYTable
#
================================================================================
#       0       0           0           0          0          0
#       1       0           0           0          0          0
#       2       0           0           0          0          0
#       3       0           0           0          0          0
#       4       0           0           0          0          0
#       5       0           0           0          0          0
#       6       0           0           0          0          0
#       7       0           0           0          0          0
#
# Initialized Source Array
#
# =============================
# 0  0  0  0  0  0  0  0
# 0  1  1  1  0  1  1  0
# 0  0  1  1  0  0  1  0
# 0  0  0  1  0  0  1  0
# 0  0  1  1  0  0  1  0
# 0  0  1  1  0  0  1  0
# 0  1  1  1  1  1  1  0
# 0  0  0  0  0  0  0  0
# =======================================
#
#
#
# Initialization Complete. Start Processing Pass 1
#
run
# Processing Pass 1 complete.
```

```
#
# Calculating Centroids
#
# The Intermediate labelled ARRAY after Pass 1
#
# ================================================
# 0  0  0  0  0  0  0  0
# 0  1  1  1  0  2  2  0
# 0  0  1  1  0  0  2  0
# 0  0  0  1  0  0  2  0
# 0  0  3  1  0  0  2  0
# 0  0  3  1  0  0  2  0
# 0  4  3  1  1  1  1  0
# 0  0  0  0  0  0  0  0
#
#
# BlobTables AFTER Pass 1
# ---------------------------------
# MergeTable AreaTable SigmaXBARTable SigmaYBARTable CenterXTable CenterYTable
#
===============================================================================
#        0       0          0           0         0         0
#        1      22        102         102      0         0
#        1       0          0           0         0         0
#        1       0          0           0         0         0
#        1       0          0           0         0         0
#        5       0          0           0         0         0
#        6       0          0           0         0         0
#        7       0          0           0         0         0
#
# Centroid calculation done
#
# MergeTable AreaTable SigmaXBARTable SigmaYBARTable CenterXTable CenterYTable
#
===============================================================================
#        0       0          0           0         0         0
#        1      22        102         102      4         4
#        1       0          0           0         0         0
#        1       0          0           0         0         0
#        1       0          0           0         0         0
#        5       0          0           0         0         0
#        6       0          0           0         0         0
#        7       0          0           0         0         0
#
run
```

```
run
#
# Processing Pass 2 complete.
#
# Final Labeled array after Pass 2
# ================================
# 0  0  0  0  0  0  0  0
# 0  1  1  1  0  1  1  0
# 0  0  1  1  0  0  1  0
# 0  0  0  1  0  0  1  0
# 0  0  1  1  0  0  1  0
# 0  0  1  1  0  0  1  0
# 0  1  1  1  1  1  1  0
# 0  0  0  0  0  0  0  0
```

6. Reference

Note: The Verilog code for the VGA synch signals generator is taken from Reference [1]. The reference design and code for the SRAM memory controller and FIFO are also taken from reference [1].

FPGA Implementation Main Reference Book

1. Chu, Pong P. *FPGA prototyping by Verilog Example* Wiley Publication, New Jersey, 2008.

Xilinx Reference websites:

2. ModelSim Simulation tool
   http://www.ece.uah.edu/~milenka/cpe626-04F/doc/se_tutor.pdf

3. Xilinx ISE Synthesis tool
   http://www.xilinx.com/itp/xilinx10/books/docs/xst/xst.pdf

4. Impact reference guide
   http://homepages.cae.wisc.edu/~ece554/new_website/ToolDoc/Xilinx_Fndtn_4.2i_docs/docs/pac/pac.pdf

5. Digilent Board reference Manual
   http://www.xilinx.com/support/documentation/boards_and_kits/ug130.pdf

6. SPARTEN 3 FPGA Reference Manual
   http://www.digilentinc.com/Data/Products/S3BOARD/S3BOARD_RM.pdf

7. S3 FPGA Block RAM Usage reference Manual
   http://www.xilinx.com/support/documentation/application_notes/xapp463.pdf

8. S3 FPGA DCM Reference guide
   http://www.xilinx.com/support/documentation/application_notes/xapp462.pdf

9. Core Generator User Guide
   http://homepages.cae.wisc.edu/~ece554/website/Xilinx/Coregen_user_guide.pdf

Image Processing Reference

10. Prof. Perkowski Image Processing Slides

http://web.cecs.pdx.edu/~mperkows/CLASS_479/lectures_2012_479/2011.%200005.%20labeling-and-sequential-algorithms.Soccer.ppt

11. Seattle Robotics Society article about FPGA based vision system

    http://www.seattlerobotics.org/encoder/200601/article3/index.php

12. C. T. Johnston and D. G. Bailey, “FPGA implementation of a single Pass Connected Component Algorithm,” in 4th IEEE International Symposium on Electronic Design, Test & Appplications, 2008.