Senior Design Final Report

E50: Parallelization of the Viola-Jones Face Detection Algorithm on an FPGA

Team:

Peter Irgens
Curtis Bader
Theresa Le
Devansh Saxena

Advisor:

Dr. Cristinel Ababei

May 6, 2016

Table of Contents

1) Abstract 2
2) Customer Needs Specification 3
3) FPGA Design 4
3.1) Introduction 4
3.2) Physical Overview 5
3.2.1) Camera 5
3.2.2) FPGA Development Kit 5
3.2.3) Display 7
3.2.4) Setup/ Connections 7
3.3) Top Level Design 8
3.3.1) Camera Driver 8
3.3.2) VGA Driver 8
3.3.3) Memory Architecture 10
3.3.4) Integral Image Generation 12
3.3.5) Subwindow Kernel Parallelization 14
3.3.6) Facebox 17
3.3.7) Top Level Control 19
4) Software Design 21
5) Experimental Verification 22
5.1) Software 22
5.1.1) Methodology 22
5.1.2) Analysis 25
5.2) FPGA 26
5.2.1) Methodology 26
5.2.2) Analysis 28
6) Economic Analysis 29
6.1) Component Cost 29
6.2) Development Cost 29
6.3) Final Estimated Product Cost 29
7) Design Risk Analysis 30
8) Customer Needs Analysis 30
9) Project Legacy 31
9.1) FPGA 31
9.2) Software 31
References 33
Appendix 34

1) Abstract

Human face detection is the process of determining whether there are faces present in an image. While humans excel at pattern detection like this, electronic detection requires a large amount of data processing resulting in poor performance especially on conventional single-threaded software implementations. This constricts real time face detection and thus limits the available applications it can be utilized for.

The focus of this project is to create a parallelized hardware face detection implementation using the original Viola-Jones (VJ) face detection algorithm on a Field Programmable Gate Array (FPGA) using VHSIC Hardware Description Language (VHDL). It is also important to create single and multithreaded versions of the software implementation of the original VJ algorithm for performance comparisons. The hardware system must be fully implemented on an FPGA, capture live video of frame size 320x240, determine whether there are faces present, and then output a video stream with boxes indicating the locations of detected faces. The software implementations analyze static images with a single face, 6 faces, and 53 faces at images sizes of 320x240, 640x480, and 1280x960 then produce an augmented image with face indicators for each. These benchmarks allow for comparable performance/cost analysis on 320x240 images with a single face and allow for extrapolative predictions of the performance of FPGAs at higher image sizes with more resources available.

The main goal is to compare the performance, in terms of frames processed per second (FPS) of a low-cost hardware system compared to a much more expensive general purpose CPU software system. The FPGA system was able to achieve detections at 4.4 FPS for a 320x240 image. In comparison, the single-threaded software implementation achieved an effective 7.7, 2.6, 0.7 FPS and the multithreaded implementation achieved an effective 12.9, 4.4, 1.4 FPS for one face at 320x240, 640x480, and 1280x960 image sizes respectively. We expect that the benefit of the FPGA based implementation will increase as the image sizes increase. This is because performance on a high density FPGA implementation can scale relatively linearly with the number of hardware accelerators used while software implementations are limited to costly multi-core processors whose performance does not linearly increase with thread count.

2) Customer Needs Specification

We present an efficient and cost effective FPGA based implementation of the VJ algorithm. This is an academic research project, and our customer is Dr. Cristinel Ababei. His primary objective for the project is to study the achievable performance with a low-end FPGA chip based implementation compared to sequential and parallel implementations of the VJ algorithm executed on a general purpose CPU. To be able to achieve our primary objective, the project was divided into the following subtasks:

1. Study the VJ algorithm and identify the major computational bottlenecks that could be parallelized using VHDL
2. Implement parallelized hardware acceleration for face detection using VHDL
3. Write a multi-threaded C++ implementation that runs on a general purpose CPU
4. Implement the multi-threaded C++ implementation to work with a video stream
5. Implement video stream processing on FPGA implementation for real-time face detection
6. Compare the multi-threaded C++ implementation with the VHDL implementation and the original implementation
7. Publish findings in a journal paper

VHDL code can be ported to almost any FPGA and our implementation is a complete system level hardware design that can be tested on various qualities of FPGA chips. We will be publishing all of our source code for the VHDL and software implementations as well as our journal paper. This will allow hobbyists and researchers to reproduce and compare our results with other implementations and under different conditions.

Based upon related work, it is possible to speed up the detection performance to 16 FPS versus the serial software implementation of 0.31 FPS for VGA resolutions, as described in [5]. Our aim is to reach an FPS that is close if not better than the FPS achieved by related works. Our objective is to achieve real time face detection with a video stream on the FPGA. By comparing the data from the two implementations we will be able to make speculations about performance versus cost of the FPGA system compared to the software implementations. The scope of this project is currently only limited to Dr. Ababei's research however it could be scaled to meet a wide variety of industrial applications such as security and monitoring.

3) FPGA Design

3.1) Introduction

The Design of the FPGA is made up of 3 main components: the OV7670 camera, the *Altera Cyclone IV FPGA DE2-115 development board*, and a generic 640x480 VGA monitor as seen in figure 3.1-1 below. The FPGA implementation must be able to capture frames from the camera, process the captured image for faces, and display the video stream output with a red box around detected faces. The processing step includes frame capture, integral image calculation, parallel subwindow processing of a pyramid of scaled images, and augmenting captured frames with face detection indicators.

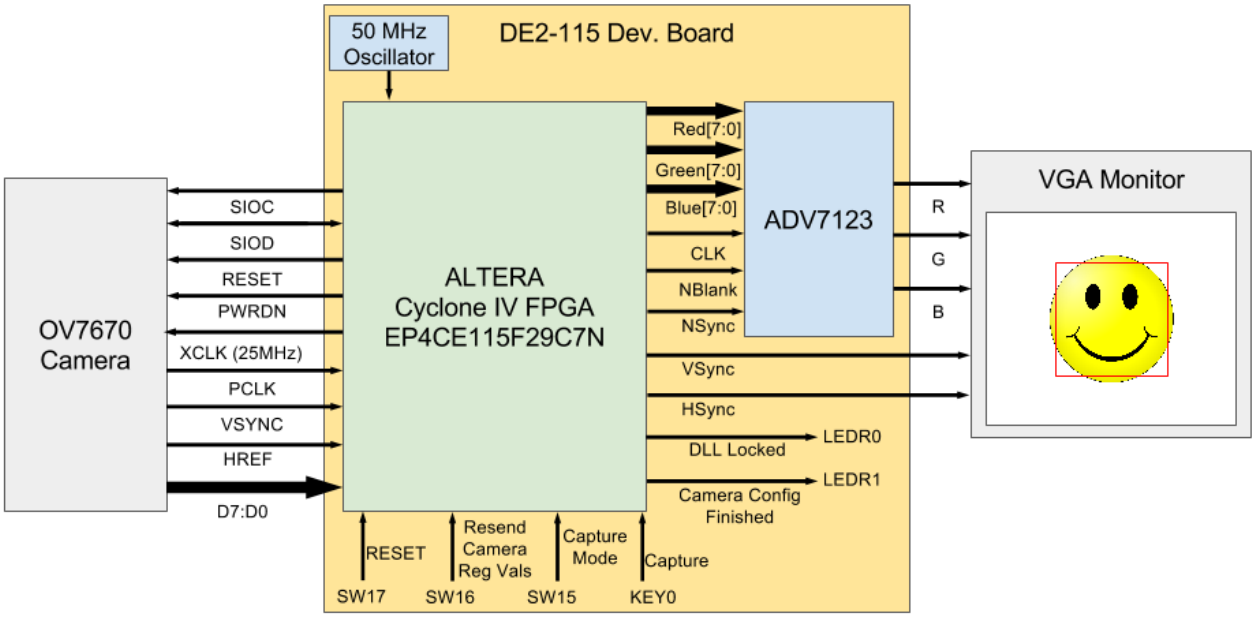


Figure 3.1-1 - System Block Diagram

3.2) Physical Overview

3.2.1) Camera

The OV7670 image sensor and DSP module is a low cost CMOS device that is capable of processing VGA resolution images (640x480) at 30 frames per second [8]. A large number of registers must be configured via the Serial Camera Control Interface (SCCB) to select the processed image resolution, output data format and other parameters related to the image quality. This SCCB interface is compatible with the industry standard I2C protocol. Due to the vast complexity and lack of documentation about register functions, it is difficult to derive the correct register settings without utilizing third party source code. The electronic hobbyist community has worked towards reverse engineering the register set for improved image quality at various resolutions. Source code for the camera configuration and image capture were liberated from Mike Field's VHDL design [7] to interface the OV7670 camera module with the FPGA. In this implementation, the camera is configured to process 640x480 resolution images, and format the output data as RGB565 format at 30 frames per second. Connections between the camera and FPGA are depicted in figure 3.1-1 and detailed in table A.2.

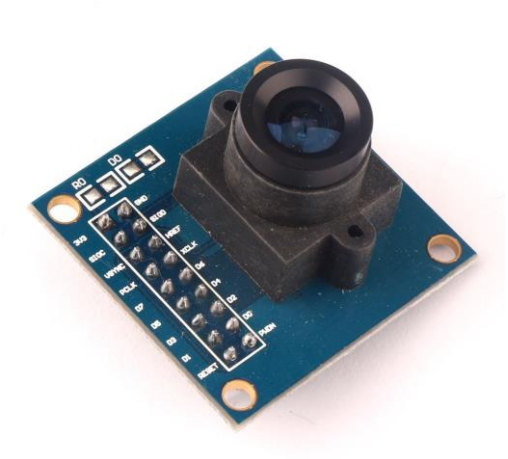


Figure 3.2-1 - OV7670 Camera Module [6]

3.2.2) FPGA Development Kit

The Terasic DE2-115 Development and Education Board [12] was chosen for our development needs since it is a cheap education/development platform that was readily available from our project advisor. Figure 3.1-1 shows a hardware overview of the development board with its peripheral connections and components. This kit implements an Altera Cyclone IV FPGA (EP4CE115F29C7N) with 3,888 Kbits of embedded memory, 114,480 logic elements and 266 embedded 18x18 multipliers. [1] In our application, the FPGA is the heart of the embedded system that captures image frames from the camera, processes the captured image for locations of faces in the image, augments the captured image to highlight the detected faces and generates VGA display signals to view the resultant face detections on a monitor. The FPGA interfaces with the camera via GPIO pins on connector JP5 while the VGA display

output is routed to a standard 15-pin D-SUB connector for monitor display. More details about the internal logic will be discussed in following sections.

A number of user controls are implemented using the development board's general purpose switches, push buttons and LEDs. As shown in figure 3.1-1 switches 15 to 17 facilitate capture mode select, resend camera register values, and reset functions. The capture mode select switch puts the system into a video processing mode if set to logic '0' or single frame capture mode if set to logic '1'. In this single frame capture mode, the KEY0 push button is used to flag a frame capture when pressed down and then store/process the single frame when the button is released. This enables us to display single frame face detection on the monitor for ease of debugging and photographing purposes. The resend camera register value switch flags the camera controller to resend OV7670 register values since there is a possibility for partial register configuration on device startup.

It should also be noted that particular debug signals and performance measurement signals were routed to GPIO pins for development and experimental verification purposes. These are listed in table A.1.

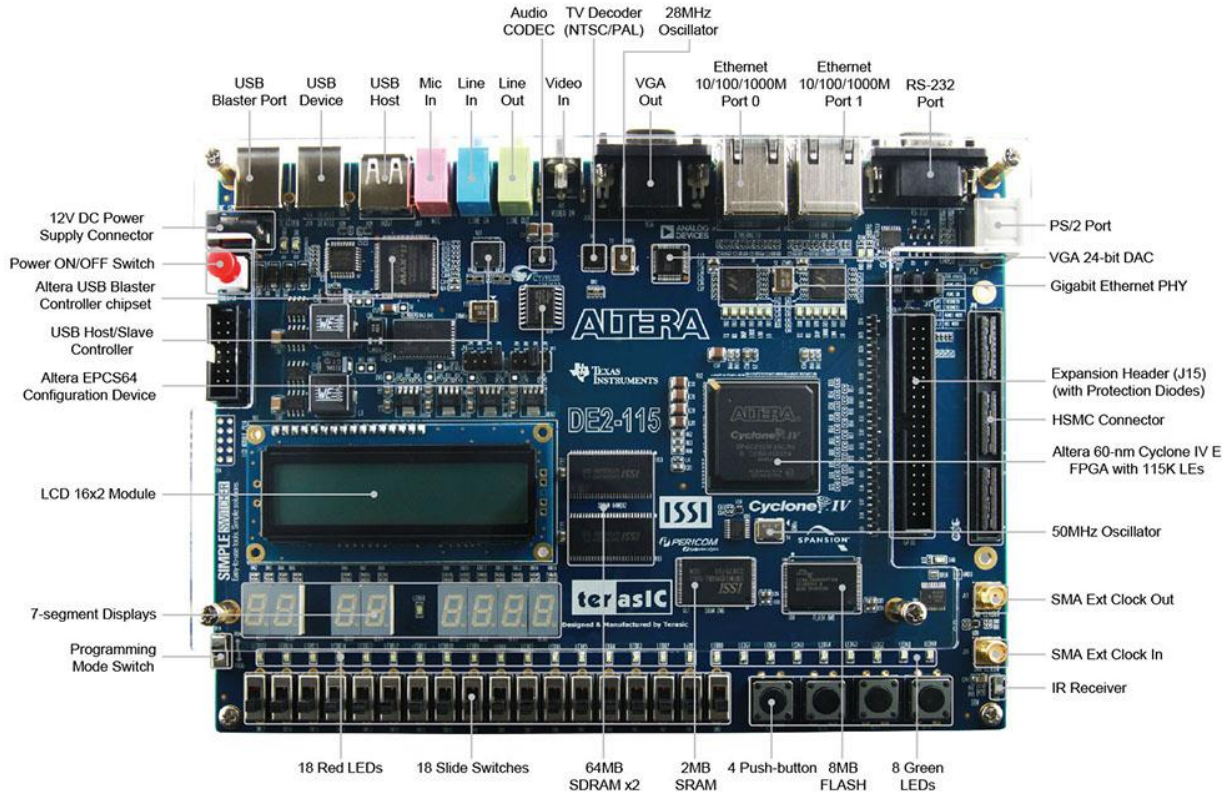


Figure 3.2-2 - Terasic DE2-115 Development Board [13]

3.2.3) Display

As previously stated, a VGA display interface is used to connect a peripheral monitor to the FPGA. Since standard VGA signals contain analog red, green and blue components, an onboard DAC (ADV7123) converts 24 bit RGB formatted data to the analog RGB. Figure 3.1-1 illustrates the signals that interface with the DAC and the signals that form the VGA interface. This implementation displays 320x240 images on 640x480, 60-Hz VGA monitors.

3.2.4) Setup/ Connections

Setup and operation of the FPGA face detection system is rather simple. Assuming the DE2-115 development kit is used, a 12V 2A power source is required. The VHDL code needs to be synthesized using Quartus Prime 15.1 software [4] and programed onto the FPGA. A VGA cable connects the board to a generic VGA monitor port and the camera is connected to the necessary GPIO pins. Tables A.1, A.2 and A.3 from the appendix contain the pin connection information. The setup should look similar to figure 3.2-3. Any display issues can be fixed using the reset switches (SW[17] and SW[16]).

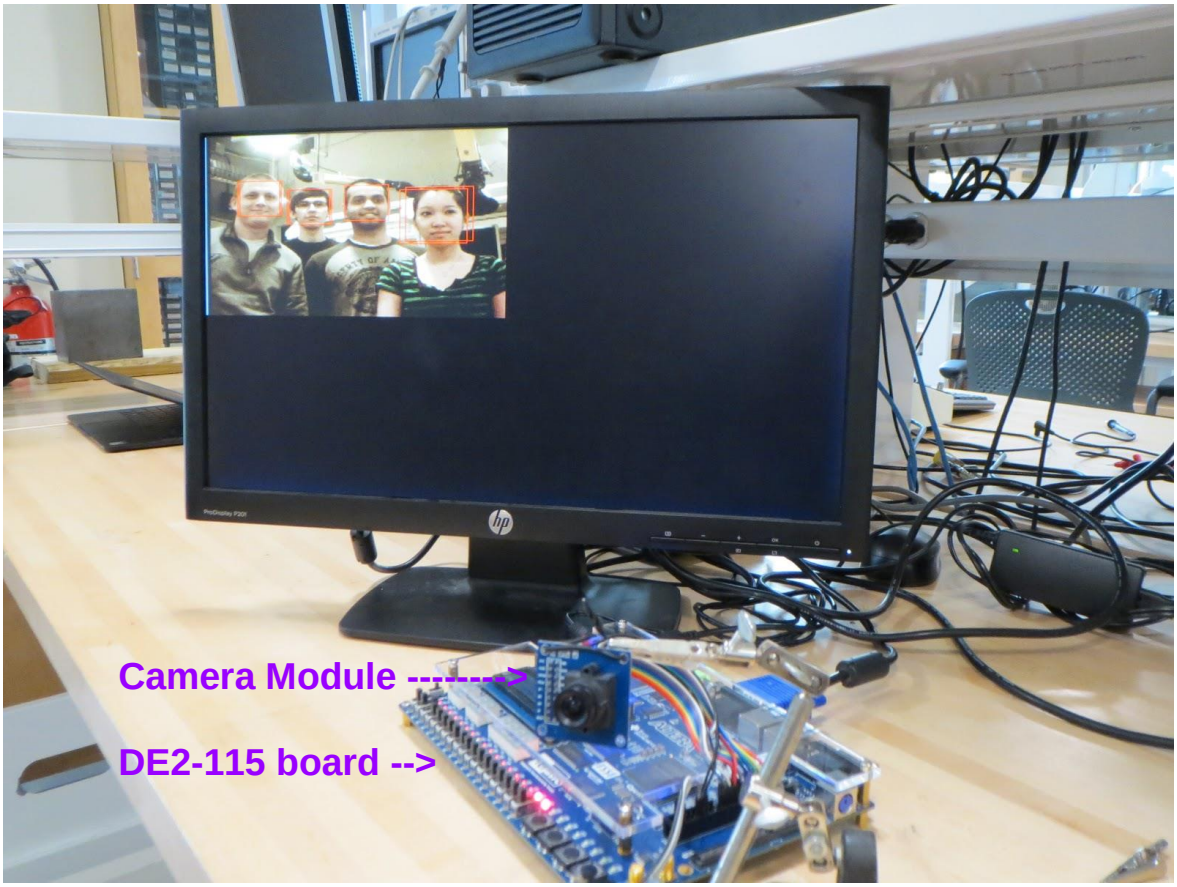


Figure 3.2-3 - Photograph of FPGA Face Detection System

3.3) Top Level Design

The top level of the design consists of a number of components. These components facilitate image frame capture, processing and display. All of these top level components are synchronized and controlled from a top level control unit. A phase lock loop (PLL) generates clock signals that are distributed to each component for synchronous processing of the subsystems. Figure 3.3-1 illustrates some of the top level interconnections between each component. Details on each top level component are discussed in the following sections.

3.3.1) Camera Driver

The camera interface consists of two components. This includes the OV7670 Controller module and the OV7670 Capture module. The OV7670 Controller configures registers on the camera via I2C, writing the configuration data sequentially to the camera.

The OV7670 Capture module effectively captures and formats data that is received from the camera. 320x240 images are captured at 30Hz into the image frame buffer with a data format of 12 bit color (RGB 4:4:4). Source code for these modules were liberated from [7] so the majority of our rapid development could be centered on the VHDL description for face detection image processing. It should also be noted that the register configuration for the OV7670 image sensor is poorly documented, so it was extremely beneficial to use working register values from [7].

3.3.2) VGA Driver

The VGA driver enables captured frames to be displayed on a peripheral 640x480, 60Hz VGA monitor. This driver continuously displays the contents of the image frame buffer, displaying both newly captured frames along with frames augmented with face detection indicators. Since this system processes only 320x240 images, the driver displays these images in the top left quadrant of the monitor. The rest of the display is blanked with null/black pixels. This module also converts the 12 bit formatted color data contained within the image frame buffer into a 24 bit color format to drive the peripheral ADV7123 digital to analog converter. VHDL code for this entity was provided from our advisor so the majority of our rapid development could be centered on the VHDL description for face detection image processing.

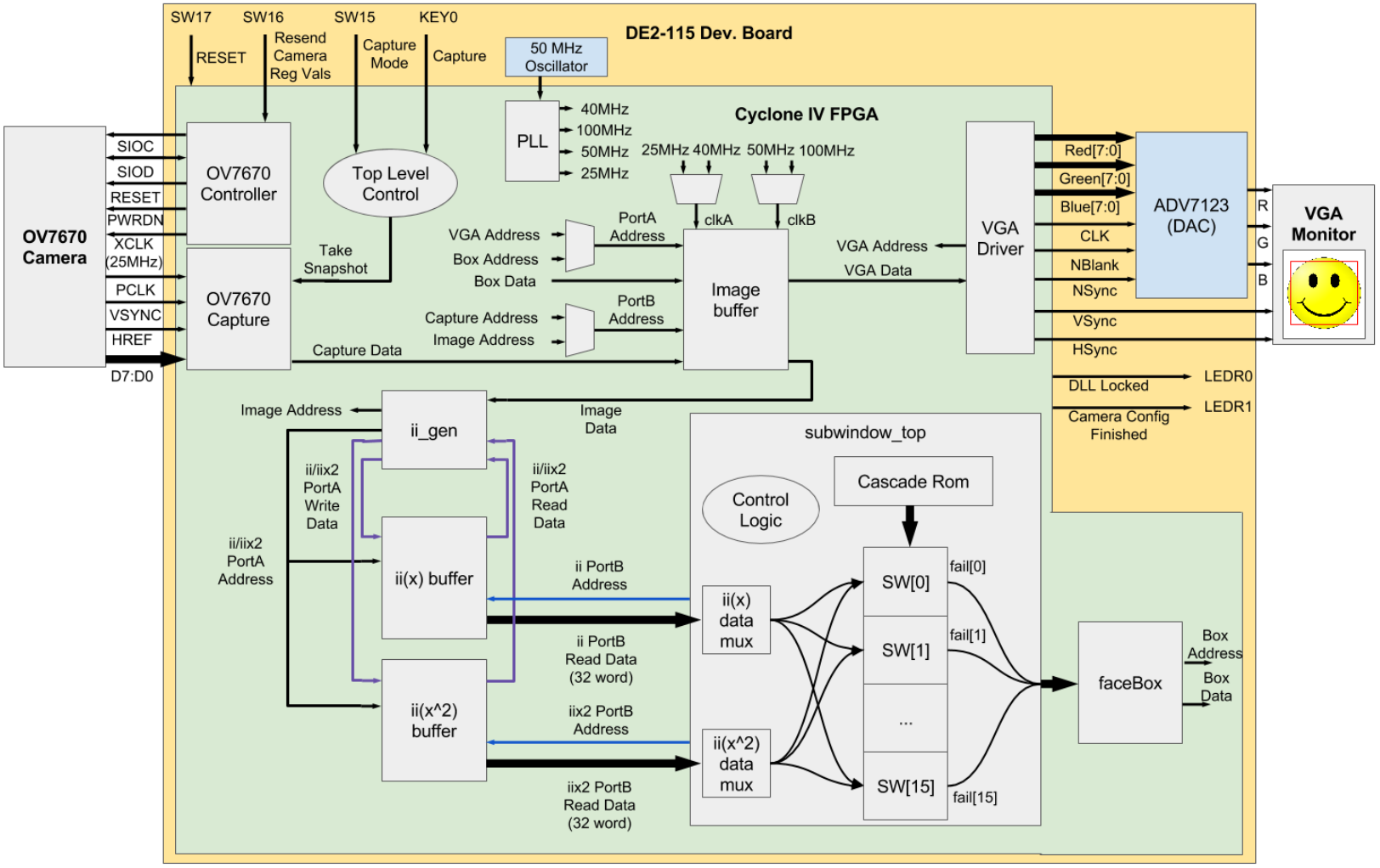


Figure 3.3-1 - Top Level Block Diagram

3.3.3) Memory Architecture

A single image frame buffer is utilized for storing captured frames from the camera and displaying the detected faces onto the VGA monitor. This buffer has 76,800 addressable memory locations to store a 12 bit color image with a resolution of 320x240. The stored image is converted into an integral image format by the integral image generator, *ii_gen*, and is augmented by the *faceBox* entity to display detected faces. Since multiple processes, operating at multiple clock frequencies, write to or read from this buffer, a number of multiplexers switch the port addresses along with the read/write clocks. Table 3.3-1 shows which clocks and addresses are multiplexed to the buffer ports during each process. Figure 3.3-1 highlights top level interconnections for this image buffer.

PortA Function	PortB Function	PortA Clock	PortB Clock	PortA address	PortB address
VGA Display	Camera Capture	25 MHz	50 MHz	VGA address (read)	Capture address (write)
FaceBox Process	II_Gen Process	40 MHz	100 MHz	Box address (write)	Image address (read)

Table 3.3-1 - Image Buffer Read and Write Addressing and Clock Multiplexing

Four buffers were implemented in the design to enable high bandwidth memory reads from both the integral image (ii) and integral image square (iix2). Each integral image consists of two buffers, containing exactly the same data, where one addresses the base(lower) chunk of data and the other addresses the next(upper) chunk in the address space. Since 16 subwindows are currently implemented, each buffer's data output width is 16*wordSize; where wordSize is 21 bit for ii and 29 bit for iix2. These chunks are adjacent in memory and allow a large mux like entity to route any 16*wordSize data set from these two addressed chunks. *Subwindow_top* is the entity that addresses these chunks based on relative position of the first subwindow kernel (*subwindow[0]*) in the scaled integral image. Each subwindow instance is offset by 1 pixel in the X direction, relative to the integral image. *Subwindow_top* generates a 13 bit address where the upper 9 bits are used to address the integral image buffers and the lower 4 bits are used as select signals for the large data mux routing. This type of memory architecture eliminates the need to access the memory twice to access data for the 16 subwindows, but comes at the cost of doubling the amount of memory needed for integral image buffering. Figure 3.3-2 shows the topology of the integral image buffers in the top level of the design and important datapath elements.

An example data access:

If *subwindow[0]* requires the data stored at pixel location x=3,y=0, then *subwindow[1]* will receive the data from x=4,y=0 and *subwindow[15]* will receive the data from x=18,y=0. So the upper 9 bits of the generated *ii_rdaddress*, within *subwindow_top*, will be zero to address the first 16 words in memory via the lower chunk buffer and the following 16 words via the upper chunk buffer. The lower 4 bits of the generated *ii_rdaddress* are used to select which 16 words are routed to the subwindows. Since the addressed pixel location was x=3,y=0, word[3]:word[18] are routed to *subwindow[0]*:*subwindow[15]* respectively.

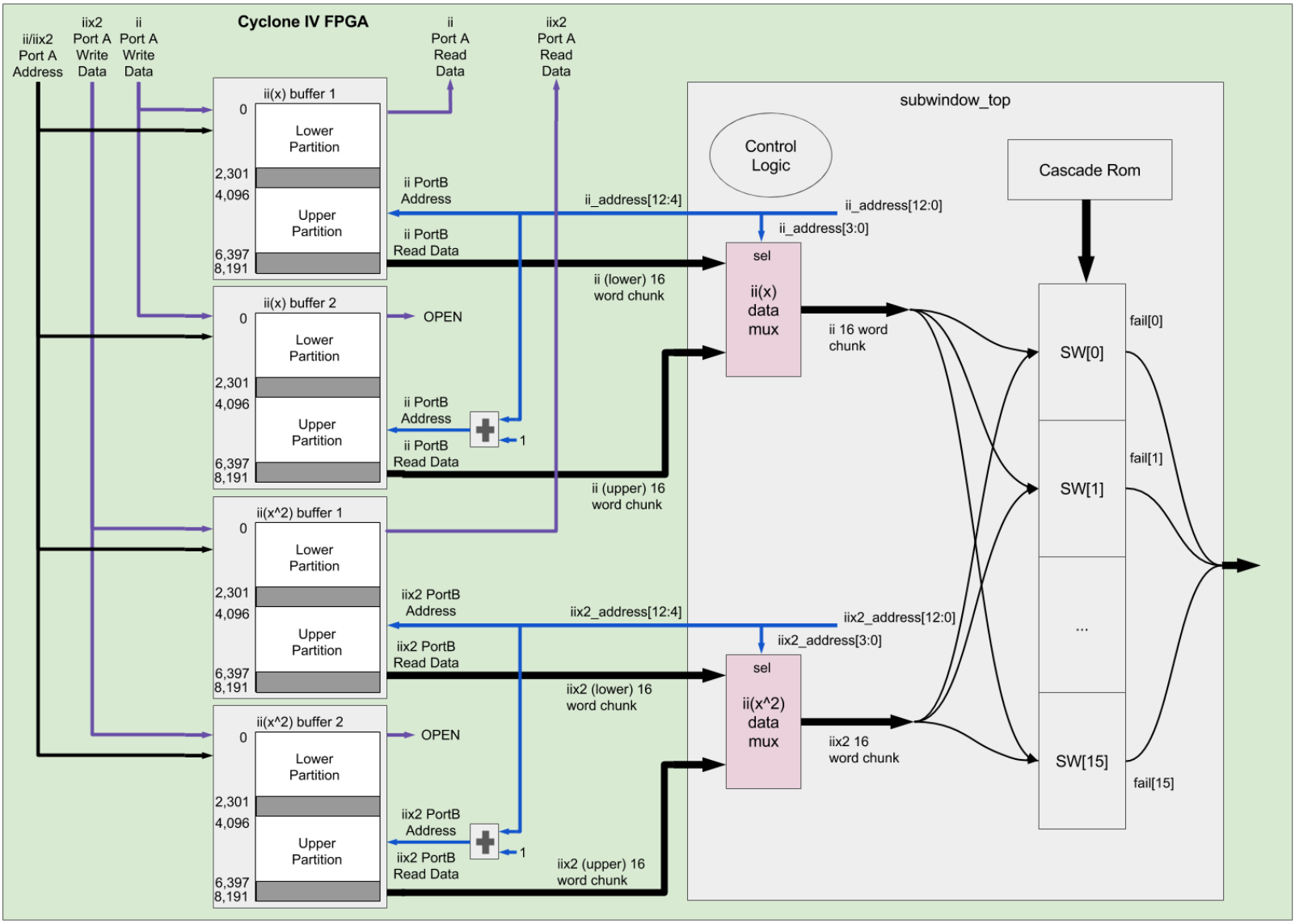


Figure 3.3-2 - Expanded Integral Image Memory Topology

It should also be noted that each integral image buffer is divided into two partitions. These are upper and lower partitions within the same addressable RAM entity. The division of each integral image buffer allows one portion to be written to by the integral image generator process while the other can be read from by the subwindow process. In this implementation, a buffer contains two 39*59 pixel integral images in the same buffer. Each buffer contains 8,192 addressable memory locations and the most significant bit of the 13 bit address determines if the lower or upper memory partitions is being operated on. So effectively 0(dec) is the base address of the lower memory partitions and 4,096(dec) is the base address of the upper memory partitions. In the top level, the *mem_state* signal determines which partitions of the integral image memory space is being written to by *ii_gen* and read from *subwindow_top*. Table 3.3-2 indicates how the integral image buffer is being written to and read from depending on the current memory state.

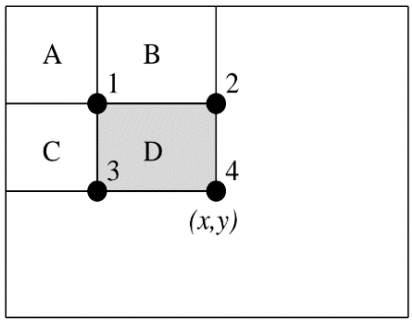
mem_state='0'	lower memory partition is being written to by ii_gen.vhd while the upper is being read to subwindow_top.vhd
mem_state='1'	upper memory partition is being written to by ii_gen.vhd while the lower is being read to subwindow_top.vhd

Table 3.3.-2 - Integral Image Memory State Table

But how does this memory architecture impact the system performance? This architecture allows the current subwindow classifier cascade to execute while the next integral image is being generated. The *ii_gen* process is always completed before the parallel subwindows can scan through the 36 possible scan locations within the current integral image. So performance is effectively the same as a memory architecture that stores the integral image of the entire scaled image.

3.3.4) Integral Image Generation

The integral image generator (*ii_gen*) converts a portion of the source image to an integral image and integral image square format. The integral image at location (x,y) contains the sum of grayscale pixels above and to the left of the of (x,y). Also, the integral image square at location (x,y) contains the sum of squared grayscale pixels above and to the left of (x,y). From this, the cumulative sum of grayscale/ grayscale squared pixels within a rectangular region can be evaluated by four array references from the respective integral image. Figure 3.3-3 illustrates that the sum of pixels inside rectangle D can be computed with four array references on values of the integral image: (4+1)-(2+3).



Grayscale= Y = 0.299*R + 0.587*G + 0.114*B Eq.1

Grayscale= 0.25*R + 0.5*G + 0.125*B Eq.2

Grayscale= R>>2 + G>>1 + B>>4 Eq.3

Grayscale= (R[3:0]&R[3:0])>>2 + (G[3:0]&G[3:0])>>1 + (B[3:0]&B[3:0])>>4 Eq.4

Figure 3.3-3 - Integral Image Calculation Reference

To generate the integral images, *ii_gen* must convert the 12 bit (RGB 4:4:4) color source data to 8 bit grayscale. From the YUV colorspace, grayscale representations of a RGB pixel can be determined by the luminance(Y) component. [3] This luminance component is the sum of individually scaled R, G and B component as expressed in Eq.1. To eliminate the need for floating point multipliers, the scalars were implemented by bit shifting each RGB component. (Eq.2 and Eq.3) Also, to increase the contrast of the grayscale data, each RGB component was concatenated to itself before the shifting. The resultant grayscale conversion of the 12 bit (RGB 4:4:4) color data is expressed in Eq.4

The integral image generator utilizes accumulators and recursive computation to create the resultant integral image. An accumulator is used to build the sum of grayscale pixels within the current row of (x,y). Adding this value to the previous rows (x,y-1) integral image value results in the integral

image value at (x,y). In the case that the generator is operating on the first row, the resultant integral image value at (x,y=0) is only the accumulated sum of grayscale pixel values up to (x,y=0). It should be noted that the datapath of the integral image generator was pipelined to increase performance. Figure 3.3-4 illustrates the data path between each buffer and indicates locations of the pipeline's registers.

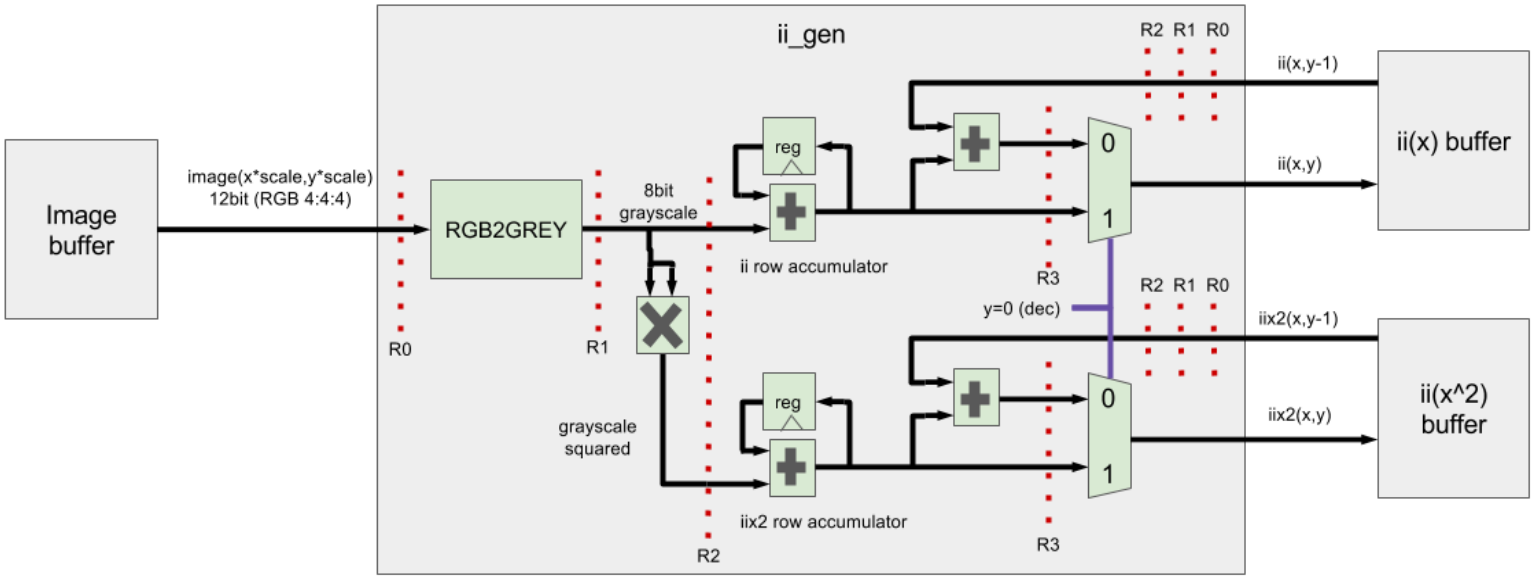


Figure 3.3-4 - Integral Image Generator Block Diagram with Pipeline Indicators

Another feature of this generator is that it enables linear scaling of the source image. It is important to process multiple scaled image sizes such that faces of difference size are scaled down to the fixed 24x24 pixel subwindow area for subwindow kernel feature evaluation. This scaling is achieved by manipulating the image buffer read address during the integral image generation process. Linear scaling effectively down-samples an image by omitting pixel data in the resultant image. The data that persists in the resultant scaled image is selected by scaling the coordinates of the source image relative to the integral image space. Figure 3.3-5 illustrates linear scaling where (x*scale,y*scale) is the location of the source data relative to the location of the resultant integral image (x,y).

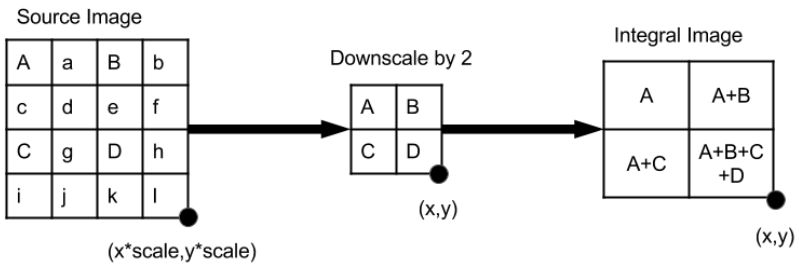


Figure 3.3-5 - Example of linear scaling by factor of 2

3.3.5) Subwindow Kernel Parallelization

A classifier cascade is a trained chain of face feature evaluations that filters out non-faces throughout its execution. Multiple feature evaluations occur throughout the 25 strong stages of the cascade, accumulating values based on feature calculations in a fixed 24x24 pixel subwindow area. This particular classifier cascade implements features comprised of three rectangular areas. The sum of grayscale values contained within these rectangular areas are used to determine differences between dark and light spots on human faces. The accumulated values from feature evaluations are compared to strong thresholds at the end of each strong classification stage. If these thresholds are not exceeded, then non-face is detected in the current subwindow. If at the end of a strong stage, a non-face is detected then the currently evaluated subwindow is rejected and another subwindow begins its evaluation for faces. If the classifier cascade completes the last classification stage without detecting a non-face, then the subwindow region is determined to contain a face. Figure 3.3-6(a) illustrates the sequential nature of the classifier cascade while figure 3.3-6(b) highlights an example rectangular feature from one cascade evaluation node.

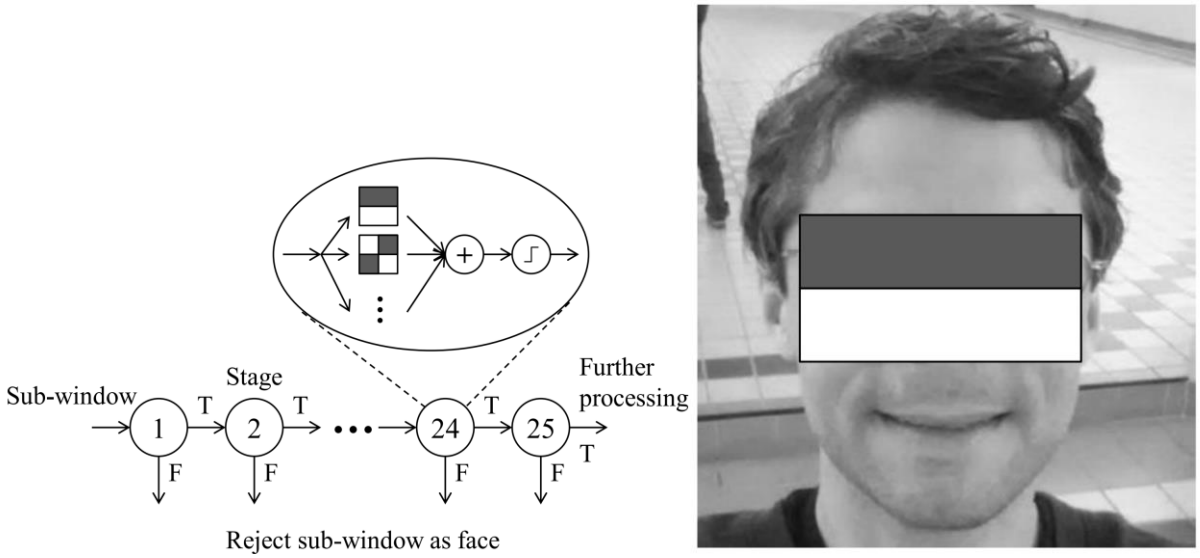


Figure 3.3-6 - (a) Classifier Cascade (b) Example Rectangular Feature

Figure 3.3-7 is a generalized representation of the subwindow kernel that facilitates feature calculations of the classifier cascade. The kernel implements two data paths: one for variance normalization of the subwindow and the other for feature calculation.

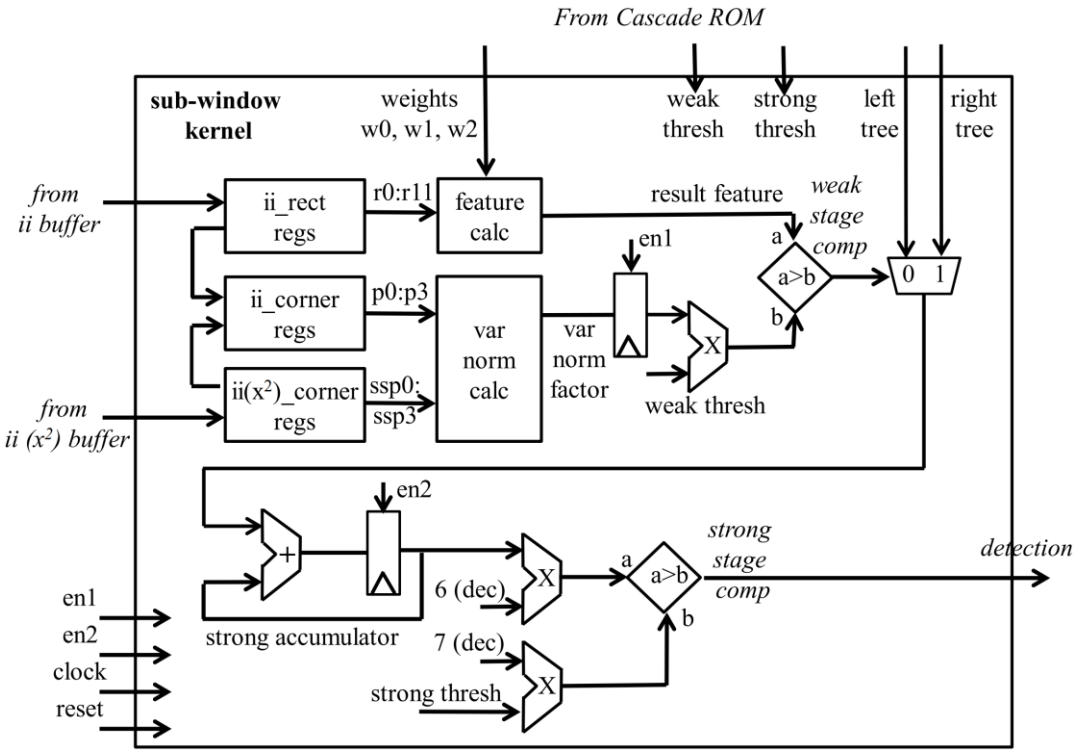


Figure 3.3-7 - Subwindow Kernel

Variance normalization aims to normalize processed subwindows to light levels of the images used in training of the classifier cascade. Similar to the Viola-Jones suggestion [14], feature values are normalized in a post calculation step. The weak threshold is scaled based upon the relative variance normalization calculation of each subwindow. Calculations for the variance normalization factor are expressed in Eq.5 and Eq.6. The mean (m) can be evaluated from the integral image (p0:p3) while the sum of squared pixel values is evaluated from the integral image of squared pixels (ssp0:ssp3). Note the the number of pixels per subwindow (N) is nominally 576 for a 24x24 subwindow area but is approximated to 512 for division via bit shifting.

$$Var\ Norm\ Factor = \sqrt{m^2 - \frac{1}{N}\Sigma(x^2)} \qquad \text{Eq.5}$$

$$Var\ Norm\ Factor\ = \sqrt{([p0 + p3] - [p1 + p2])^2 - \frac{1}{512}([ssp0 + ssp3] - [ssp1 + ssp2])} \qquad \text{Eq.6}$$

The second data path facilitates feature calculation of the classifier cascade. Integral image values representing three rectangles, or one feature, are calculated and weighted at the same time to produce the resultant feature value. Values from the integral image (r0:r11) enable quick summations of the pixels within a rectangular region.

$$Feature = weight0 * \Sigma pixels_{rect0} + weight1 * \Sigma pixels_{rect1} + weight2 * \Sigma pixels_{rect2} \qquad \text{Eq.7}$$

$$Feature = w0 * ([r0 + r3] - [r1 + r2]) + w1 * ([r4 + r7] - [p5 + r6]) + w2 * ([r8 + p11] - [r9 + r10]) \qquad \text{Eq.8}$$

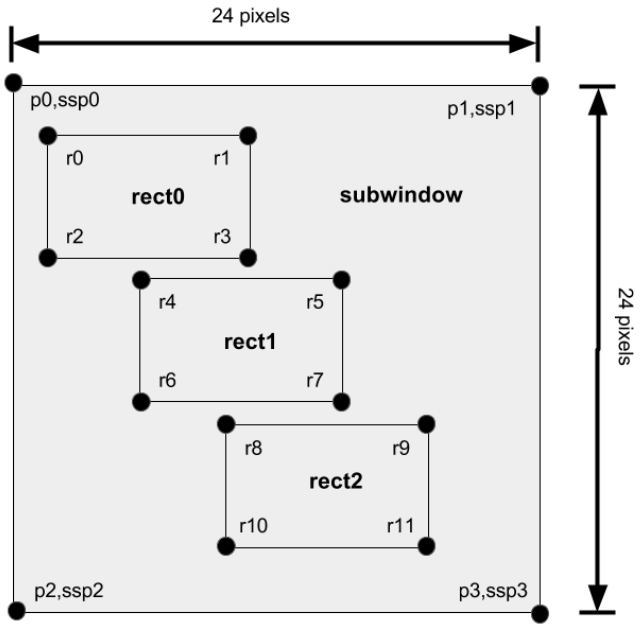


Figure 3.3-8 - Relative orientation of rectangle references in subwindow area. Locations and dimensions of rectangles are dependent on classifier cascade parameters.

Based upon the normalized weak node threshold and the resultant feature value, a *left_tree* and *right_tree* value is accumulated into a register for strong threshold comparison at the end of a classification stage. Detection results are monitored in the *Subwindow_top* level.

To achieve performance gains through parallelization, 16 subwindow kernels are implemented in parallel within the Subwindow_top level of the design. These kernels evaluate 16 individual subwindow regions that are offset by one pixel horizontally (relative to the scaled image/integral image) while sharing the same control and cascade parameters. The theoretical maximum performance gain over a single kernel would be 16x. But since the kernels operate in a single instruction multiple data manner, a single face in one of the subwindows will lengthen the effective execution time for all kernels. So an estimate performance gain over a single kernel is between 10x-12x, depending on the number and density of faces in the image.

3.3.6) Facebox

FaceBox is an entity that serves as an intermediate memory element to store detection results and draw indicators (red boxes) onto the original captured frame after all detection processing is complete. The buffer contained within this entity serves as FIFO memory that stores detections from the 16 subwindow kernels along with positions and image scales relative to the processed subwindow. After subwindow processing is complete for all scales of the image, the faceBox process writes data to the image buffer in the top level, patterning red boxes around each detected face. Figure 3.3-9 shows a generic representation of the components contained in this entity and figure 3.3-10 expresses how boxes are drawn onto captured image.

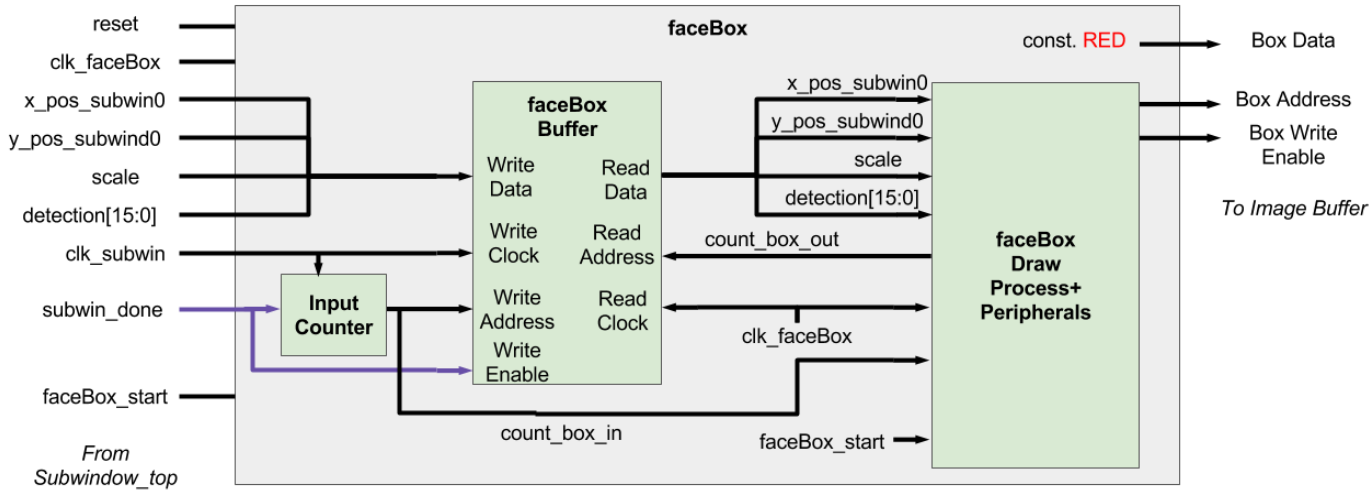


Figure 3.3-9 - faceBox Entity

Algorithm: faceBox draw process

```
1:  Input: scale, subwind0_xpos, subwin0_ypos from FIFO buffer, addressed to count_box_out
2:  Output: red pixel data from drawLine functions
3:  wait for faceBox start flag
4:  while count_box_out < count_box_in do
5:      for i=0:15 do // 16 subwindows logged in parallel
6:          if detection=true for subwindow[i] then
7:              scale=current subwindow scale
8:              box_dim=24*scale
9:              x_base=(subwin0.x_pos + i)*scale
10:             y_base=(subwindow0,y_pos)*scale
11:             drawLineHorizontal(length=box_dim, x_base, y_base, scale) // top line
12:             x_base=(subwin0.x_pos + i + 24)*scale
12:             y_base=(subwindow0,y_pos)*scale
14:             drawLineVertical(length=box_dim, x_base, y_base, scale) //  right line
15:             x_base=(subwin0.x_pos + i)*scale
16:             y_base=(subwindow0,y_pos)*scale
17:             drawLineVertical(length=box_dim, x_base, y_base, scale) // left line
18:             x_base=(subwin0.x_pos + i)*scale
19:             y_base=(subwindow0,y_pos + 24)*scale
20:             drawLineHorizontal(length=box_dim, x_base, y_base, scale) // bottom line
21:         end if
22:     end for
23: count_box_out++
24: end while
25: flag draw process is done
```

Figure 3.3-10 - faceBox Draw Process

3.3.7) Top Level Control

A top state machine was designed to coordinate the camera capture, image processing and writing detection indicators onto the captured frame. This state machine first enables new image frames to be captured into the image buffer and then proceeds to process the image. Image processing consists of integral image generation and subwindow evaluation. An integral image, of size 39x59 pixel, is initially generated. Following this, the next integral image in the scaled image is generated while the 16 parallel subwindow kernels scan through and evaluate the previous integral image for faces. By the end of the subwindow scanning, the next integral image has been generated such that the subwindows proceed to evaluate this new integral image without any downtime. The concurrent subwindow evaluation of the current integral image and generation of the next integral image continues in a scanning pattern throughout the current scaled image. Once the last subwindow scan in a scaled image is complete, the image scale increments and the system proceeds to evaluate/scan the image at the new image scale. The system evaluates four scaled images and then updates the original captured image with red box indicators around the detected faces. This augmented frame is displayed on the VGA monitor for a few frames before the system captures a new frame to process. The effective scanning of the subwindow kernels and integral images is depicted by figure 3.3-11. A high level overview of the top level state machine process is expressed in figure 3.3-12.

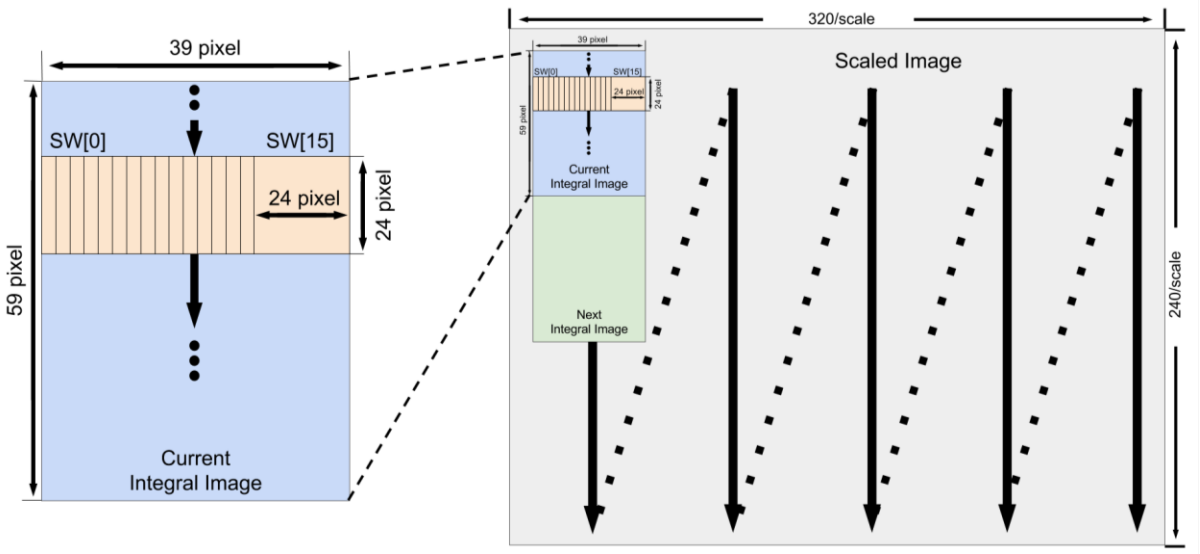


Figure 3.3-11 - Integral Image and Subwindow Kernel Scanning

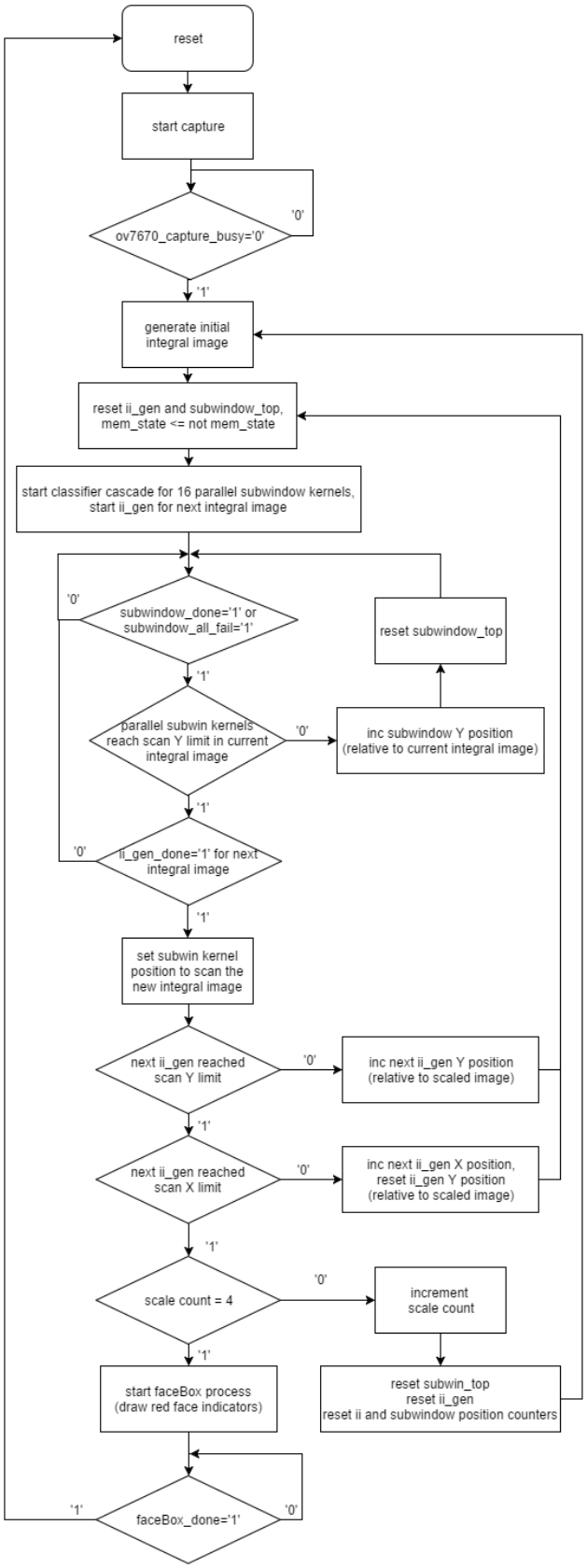


Figure 3.3-12 - Top Level Process Flow Chart

4) Software Design

The objective of the software design is to parallelize the VJ algorithm using pthreads in order to achieve a speed boost compared to the sequential (original) version [11]. For reference, the pseudocode of the algorithm is presented in Figure 4.1-1 below.

Algorithm: Viola-Jones Face Detection Algorithm

```
1:  Input: original test image
2:  Output: image with face indicators as rectangles
3:  for i ← 1 to num of scales in pyramid of images do
4:      Downsample image to create image_i
5:      Compute integral image, image_i
6:      for j ← 1 to num of shift steps of sub-window do
7:          for k ← 1 to num of stages in cascade classier do
8:              for l ← 1 to num of filters of stage k do
9:                  Filter detection sub-window
10:                 Accumulate filter outputs
11:             end for
12:             if accumulation fails per-stage threshold then
13:                 Reject sub-window as face
14:                 Break this k for loop
15:             end if
16:         end for
17:         if sub-window passed all per-stage checks then
18:             Accept this sub-window as a face
19:         end if
20:     end for
21: end for
```

Figure 4.1-1. A pseudocode variation of the Viola Jones Algorithm.

In the multithreaded implementation, the outermost for-loop in line 3 was optimized with one pthread for each scaled image in the image pyramid. The algorithm is computed over every scaled image in the image pyramid sequentially and this makes the image pyramid a very good candidate for parallelization such that multiple scaled images can be computed concurrently. After a pthread is created for every scaled image, all the pthreads are executed simultaneously.

For the pthread implementation to work correctly, the threads must be able to read from the cascade classifier and then write their results into a single data structure. To avoid any race conditions when the threads are trying to read from the cascade classifier, we created local copies of this object for each of the threads so that the threads no longer needed to compete for this object. We used this approach

instead of implementing a mutex lock because the sub-window sweeping heavily relies on the data stored in the cascade classifier and the threads must be able to access it quickly for every sub-window shift. Making the threads compete for this resource would tremendously slow down the computation. Once the threads have completed their task, they have to be able to store the results (location of detected faces) into a single data structure which is then used to draw the rectangles around faces. The threads need to access this data structure only at the end once all the computation is completed and store the results. Because the threads need access to this data structure only once, we protected it with a mutex lock instead of creating a data structure for each pthread and compiling them together into a single data structure in the end. The latter approach would have been more computationally intensive and increased the execution time.

5) Experimental Verification

5.1) Software

To be able to detect faces of different sizes, the algorithm works with a pyramid of scaled images. This effectively allows sweeping using the same set of Haar-like patterns on different scaled versions of the initial image. Thus, sliding sub-windows will sweep each of the images from the pyramid as illustrated in Figure 5.1-1. The image pyramid was parallelized using pthreads such that a thread is assigned to each of the images in the image pyramid and then the algorithm is computed over each of these images simultaneously.

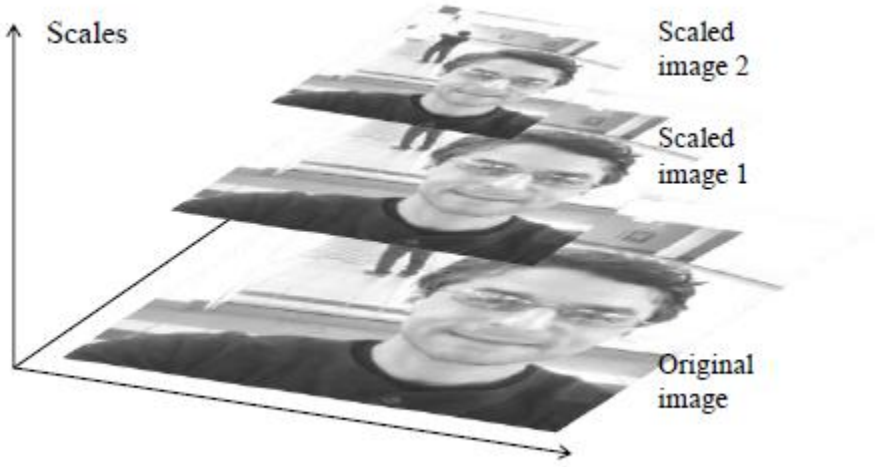


Figure 5.1-1 - Image Pyramid

5.1.1) Methodology

Testing verification involved running both the sequential and parallelized algorithms over a set of test images and recording two things: the number of faces detected and the execution time. The set of test images includes a photo with one face (Figure 5.1-2), six faces (Figure 5.1-3), and fifty-three faces

(Figure 5.1-4). All the test images were resized to three different sizes: 320x240 pixels, 640x480 pixels, and 1280x960 pixels.

Figure 5.1-2 - The 240x320 pixel image for one face (Dr. Cristinel Ababei).

Figure 5.1-3 - The 320x240 pixel image for six faces.

Figure 5.1.4 The 320x240 pixel image for fifty-three faces.

We predict that as the number of faces increases and the size of the photo increases, the execution time will also increase. This is because for a smaller image, it may not need to be scaled and resized eight times in order to detect a face whereas for a larger image, the algorithm may go through the entire set of scaled images in the image pyramid to find a face. With a larger number of faces, the algorithm needs to stop at each potential face to determine if a face is actually there. This process would happen less frequently if there are fewer faces in a photo. The majority of the sub-windows in a picture are non-faces and are rejected very fast since the algorithm does not need to go through more stages of the cascade classifier. More faces implies that the algorithm runs through all the stages of the cascade classifier for these faces and hence, requires more execution time. We also predict that the parallelized implementation will be faster than the sequential implementation in every case. Results of our tests are recorded in the table 5.1-1 and presented visually in Figure 5.1-5.

Image	Total number of faces	Size	Original -- Execution Time (ms)	Original -- Number of faces detected	Original -- FPS	Multithreaded -- Execution Time (ms)	Multithreaded -- Number of faces detected	Multithreaded -- FPS	Speedup
Cris	1	Small	128.785	1	7.765	77.381	1	12.923	1.664
		Medium	383.008	1	2.611	227.059	1	4.404	1.687
		Large	1422.079	1	0.703	732.453	0	1.365	1.942
Stock Photo	6	Small	155.474	6	6.432	93.24	5	10.725	1.667
		Medium	440.47	6	2.270	228.539	6	4.376	1.927
		Large	1640.132	6	0.610	838.548	6	1.193	1.956
Original Test Photo	53	Small	157.924	0	6.332	95.339	0	10.489	1.656
		Medium	521.372	46	1.918	288.512	33	3.466	1.807
		Large	1748.578	43	0.572	895.484	42	1.117	1.953
								Average:	1.807

Table 5.1-1 - Test results of the original and parallelized implementation of the VJ Algorithm.

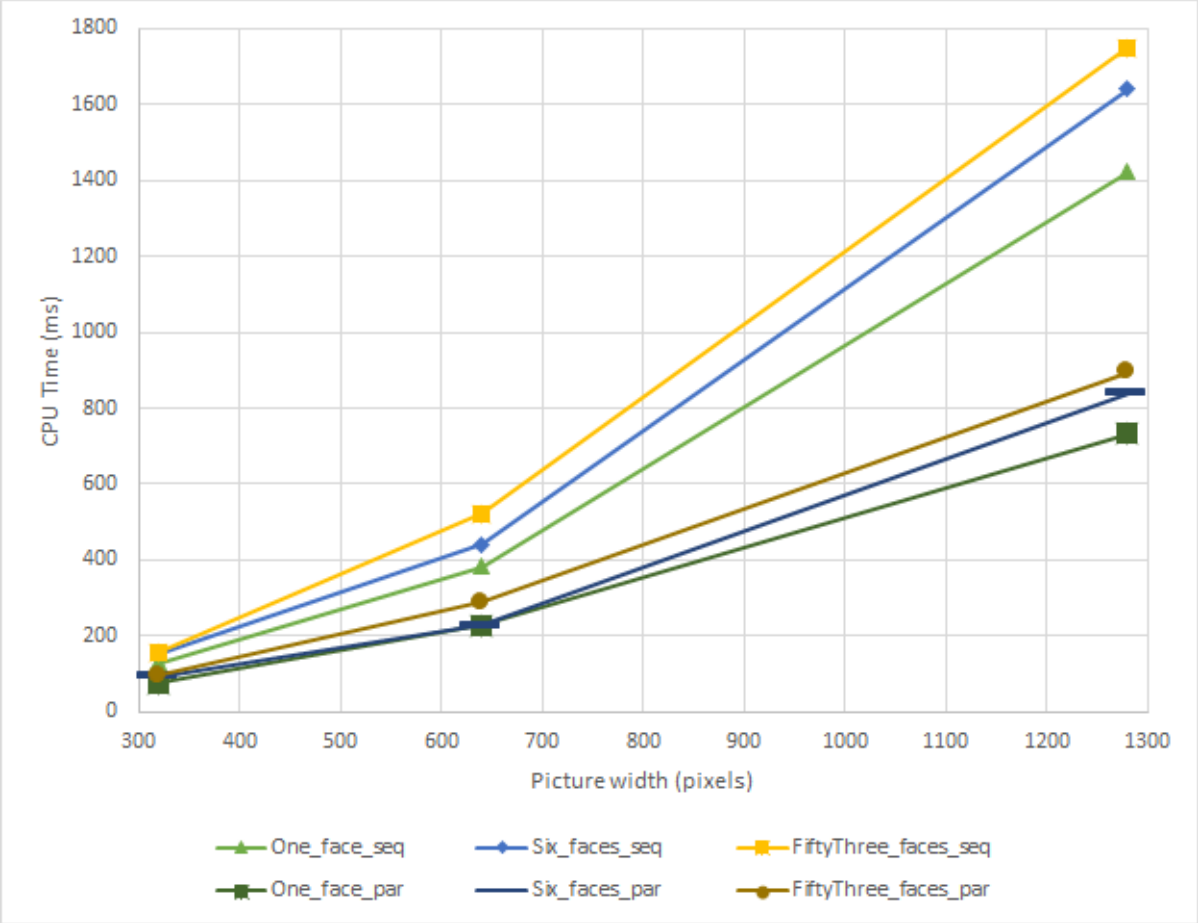


Figure 5.1-5 - A plot of execution times versus picture size and number of faces. The abbreviation "seq" stands for sequential (original) and "par" stands for parallelized (ours).

5.1.2) Analysis

As predicted, the execution time increased as the number of faces and the size of the photos increased. This makes sense because one would expect that the algorithm would need to stop more frequently at each possible face candidate when there are more faces in the photo. The algorithm scales more images in the image pyramid as the image size increased. Also, the parallelized implementation of the algorithm was consistently faster than the sequential implementation. This was also expected because some of the work was divided and executed simultaneously thereby making the overall execution time faster.

It is also important to note that there are some instances where our implementation did not detect some faces that the original implementation did. There are a few reasons why this may have occurred. One possible reason is because we constrained the number of images in the pyramid to eight so that we could assign each scaled image to a thread to execute concurrently with the other threads. Another is that we constrained the scale factor to be a whole number to be more in line with the FPGA implementation. The original VJ Algorithm did not use a whole number for the scale factor and there was not a limit to the number of images in the image pyramid. Therefore, the original implementation was able to create more images in the image pyramid with some of the sizes being in between the scaled images we generated.

5.2) FPGA

To quantify the performance of the FPGA based face detection system, the number of processed frames per second (FPS) must be measured. This is accomplished by measuring the time elapsed from capturing an image to displaying the detection results. More specifically this is the time elapsed from the start of a frame capture to the end of drawing face boxes onto the captured image. An experiment was conducted to gather the minimum, maximum and average FPS of our FPGA face detection system.

5.2.1) Methodology

To measure the FPS of the FPGA system, a register was implemented such that it toggles every time the face box draw process completes. This register output was routed to a GPIO of the FPGA such that periodic toggling of the register can be measured. An oscilloscope was used to measure the frequency of the generated signal. It should be noted that the generated signal frequency is the measure of two full image processing cycles per second. Since the actual FPS of the system is relative to one full image processing cycle, the FPS is double the observed signal frequency.

Ten measurements were conducted to determine the minimum, maximum and average FPS. Tests were performed for a single face centered in the frame, medium lighting and with a white background. Data samples were collected with the subject's face at a variety of distances from the camera. It should be noted that this system processes 320x240 pixel images with 4 scales implemented. Figures 5.2-1 and 5.2-2 show a block diagram of the setup and the actual setup we used respectively.

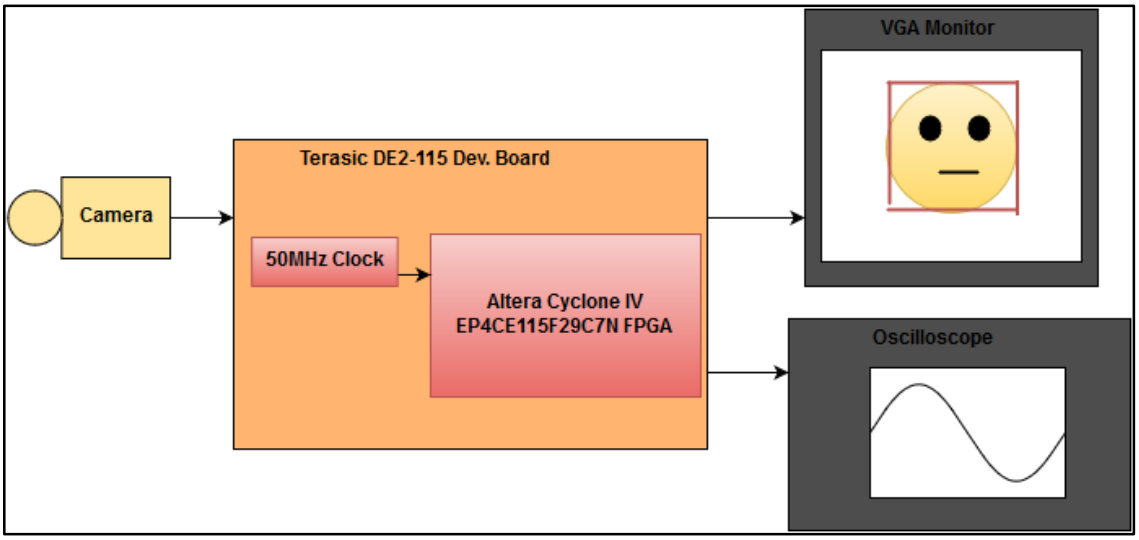


Figure 5.2-1 - General Test Setup Block Diagram

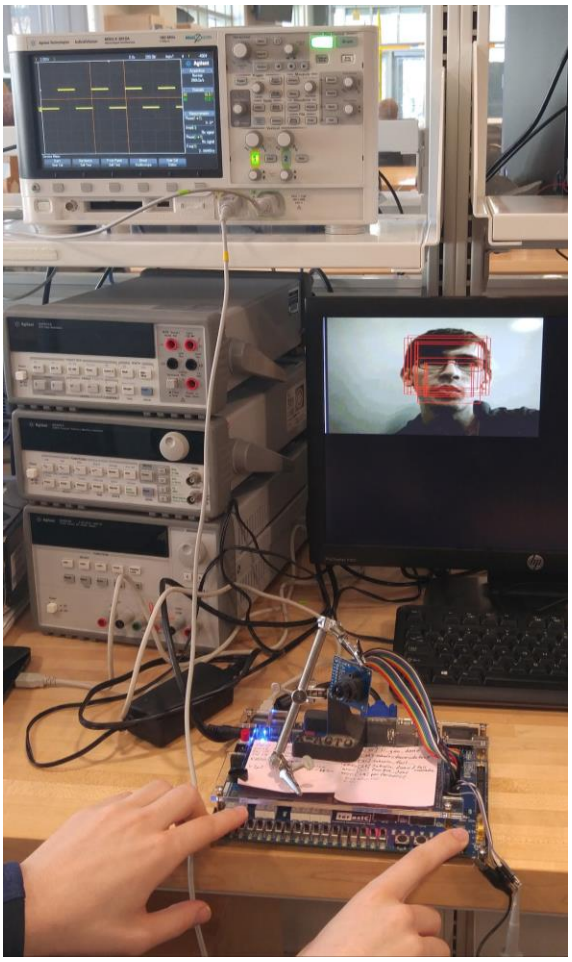


Figure 5.2-2 - Photo of Test Environment

Device	Description
Camera	ov7670 without fifo
Development Kit / FPGA	Terasic DE2-115 w/ Altera Cyclone IV EP4CE115F29C7N FPGA
Monitor	Generic 640x480 VGA, 60Hz

Table 5.2-1 - System components

Test Equipment	Serial#	Notes
Agilent MSO-X 2012A 100MHz Oscilloscope	MY53280162	Located in DLLe, no calibration history

Table 5.2-2 - Test equipment

Detection Rate		
Data Sample	**Measured frequency [Hz]**	**Frames Per Second**
1	1.97	3.94
2	2.23	4.46
3	2.2	4.4
4	2.08	4.16
5	2.4	4.8
6	2.23	4.46
7	2.41	4.82
8	2.4	4.8
9	2.08	4.16
10	1.97	3.94
Single Face Min	1.97	3.94
Single Face Max	2.41	4.82
Single Face Avg	2.197	4.394

Table 5.2-3 - FPGA test results

5.2.2) Analysis

From the results in table 5.2-3 it is clear that the FPGA performs at an average 4.4 FPS, ranging from 3.94 to 4.82 FPS. This is a respectable detection speed for this image size but could be improved by further optimizing internal routing and sequential logic. It should be noted that lighting conditions play a large role this system's performance since other tests in a darker lit room with a desk lamp lighting the target face results in detection rates between 4-7 FPS. See the appendix for details related to design metric such as resource utilization, power consumption, and timing analysis.

In comparison to the pthread software implementation, for a single face in 320x240 pixel image, the pthread effective performance for the Cris image (figure 5.1-2) was 13.5 FPS compared to the FPGA's average 4.4 FPS. This small image comparison favors the pthread implementation for detection rates.

Performance gains, in favor of the FPGA implementation, occur at larger image sizes. The pthread implementation resulted in a detection speed as low as 0.987 FPS while the performance of an FPGA based system could remain consistent at 4.4 FPS by increasing the number of subwindow kernels. Although we do not have experimental results to verify this prediction, an FPGA with more resource could effectively implement multiple instances of our 16 kernel design and process different locations of the same image with minimal performance impact. It should also be noted that the pthread results do not include the execution time for capturing an image while the FPGA dedicates 33ms to a single frame capture.

6) Economic Analysis

Product cost is not relevant to this project because it is purely an academic research project. Therefore there is not a preliminary bill of materials, an estimated sales forecast, an expected production process, or an estimation of manufacturing labor costs.

6.1) Component Cost

The team has been using an FPGA chip and DE2-115 board as well as a camera to work on this project. The FPGA chip and DE2-115 board have been given to the team by Dr. Ababei. The cost of these components are detailed in Table 6.1-1.

Component	**Cost**
DE2-115 FPGA Development Board	$595 [1]
OV7670 Camera Module (B) 640X480 CMOS CameraChip Image Sensor Development Board	$11.99 [2]
Total:	$606.99 * 4 team members = **$2427.96**

Table 6.1-1. A table listing the components we plan to use and their cost.

The software used to program the FPGA is Quartus Prime 15.1 and is available for free on the Altera website.

6.2) Development Cost

Concerning the development costs, each team member worked an estimated 160 hours to complete the project. Assuming an $50 hourly rate for each team member, the total development cost is defined in Eq.9 as:

*$50/hr * 160 hours * 4 team members = $32,000* Eq. 9

6.3) Final Estimated Product Cost

The recurring cost includes maintenance of the VHDL code. Based upon Glass's findings in his article "*Frequently forgotten fundamental facts about software engineering*" [10], maintenance accounts for 60% of the total cost of the entire software life cycle. Therefore, the total cost is defined in Eq.10 as:

*(Development cost + component costs) * 100 / 40 = ($2427.96 + 32000) * 100 / 40 = ~**$86,070*** Eq.10

[1] http://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&No=502
[2] http://www.ebay.com/itm/like/251698097943?ul_noapp=true&chn=ps&lpid=82

7) Design Risk Analysis

Concerning the design risk, the application of our project was not used for any safety applications, meaning that our project has no potential to be used in a situation that may harm human beings. Because of this reason, the design for the VHDL code was done without regard to any standards that may be audited at any point. There are also no regulatory issues or expected pathway for clearance to market. Lastly, there are no legal issues with the design concerning patents or state or federal laws.

8) Customer Needs Analysis

The primary objective of this project was to study the achievable performance with a low-end FPGA based implementation of the VJ algorithm compared to sequential and parallel implementations of the algorithm executed on a general purpose CPU. To be able to achieve this objective, the project was divided into seven subtasks. The subtasks along with their status are listed in Figure 8.1-1.

Customer Need	**Project Result**
Study Viola-Jones algorithm and identify computational bottlenecks	Computational bottlenecks were identified and parallelized using hardware and software
Implement parallelized hardware acceleration of face detection using VHDL	Image capture, display, and image processing components are described in VHDL and implemented on an FPGA. Subwindow kernels achieve parallelized execution of classifier cascade
Create single and multi-threaded software implementations for comparison benchmarks	Both versions created and tested with multiple static images of various sizes and number of faces
Implement multi-threaded implementation to work with a video stream	This task was not accomplished in the given time frame and requires more time for research and testing
Implement face detection video stream	Real time FPGA implementation achieved a detection rate of 4.4 FPS on average.
Compare the single and multithreaded CPU implementation with the FPGA implementation	Performance for the single and multithreaded CPU implementation was compared with the performance of the FPGA Implementation. Our findings are expressed in section 5 of this document.
Submit journal paper and release all source code to public	Journal paper submitted to Springer Publishing and awaiting approval. All source code and papers have been made available to Dr. Cristinel Ababei

Figure 8.1-1 - Customer Needs Analysis

We were able to achieve the primary objective for this project and study the performance of a low-end FPGA chip based implementation of the VJ algorithm compared to sequential and parallel CPU implementations of the algorithm. Even though we did not get the multithreaded CPU implementation to work with a video stream, we were able to calculate the effective frames per second for our CPU implementation which we then compared to the FPGA implementation. One of the goals of this project is to provide our source code for the VHDL and software implementations so that other researchers can reproduce our results and compare them to other implementations.

9) Project Legacy

9.1) FPGA

With respect to the FPGA design, what was learned includes elements from clean VHDL description techniques. From this project, VHDL descriptions of embedded RAM/ROM entities was a fundamental learning experience. Designing memory entities for the classifier cascade ROM emphasized the structure of Altera's embedded memory, specifically the structure of the registered inputs which enabled us to derive a state machine that properly sets up control/data signals and latches information that is read. It was also a good experience in determining how to initialize the contents of multiple ROM components that are declared from the same design entity through the use of generic mapping and file I/O libraries.

An equally important takeaway is development of design skills for complex hardware designs that are dependent on execution performance. Specifically, using the Altera TimeQuest Time Analyzer to determine approximate combinational propagation delays between registered elements enabled the integral image generator to be pipelined effectively. Pipelining enabled the integral image generator to be clocked at 100MHz while pipelined rather than 40MHz without. This performance increase, coupled with the toggling partitioned integral image memory, enables the face detection system to continuously feed the subwindow kernels while minimizing the embedded memory resource utilization for the design. It should be noted that the FPGA that was used in this design would have not been able to facilitate integral image sizes of 320x240 due to embedded resource limitations, so this memory optimization is necessary. Also, this optimization serves as platform for resource savings in future designs that clone multiple instances of our top level design on larger FPGAs.

Future design improvements for the FPGA should be focused around the integral image to subwindow kernel interface. It was determined that the timing most critical data path exists between the registered integral image address input and the integral image registers contained within each subwindow kernel. Here, the system spends most of its execution time loading integral image values into the subwindow configuration registers for variance and feature evaluations. So since the current design is limited to a 40 MHz clock speed at this interface, it would behoove future designs to optimize the addressing/ datapath delays by revisions to combinational logic or implementation of pipelining.

9.2) Software

The following are a few lessons were learned while working on the multi-threaded software implementation of the VJ algorithm.

1. For concurrent programming, the developers must be able to properly visualize how threads interacts with specific resources and remember that these threads are executing concurrently. Drawing out the threads with the resources can really help find and predict race conditions.
2. Reading the pthread documentation thoroughly helped us realize that race conditions can occur not only when the threads are trying to write to a certain resource but also when they are trying to read from it.
3. Preventing race conditions from occurring due to multiple threads. Forgetting to set up these protections led to many segmentation faults as multiple threads were unable to gain access to the

data it needed. To resolve this issue, local copies were made of the data structure that each thread needed to read from and a mutex lock was used to protect the data structure that all the threads had to write to after terminating.

4. To simplify your work, start with an image the size of the sub-window with a single face and running on a single thread. Once this implementation works, increase the image size by a pixel so that now you would have two sub-windows. Now, test your implementation with two threads. Eliminating all the unnecessary variables when debugging pthreads help find the actual source of the problem.

One improvement for software is to pick a different part of the algorithm to parallelize. The outermost for-loop (the image pyramid) was selected to parallelize however parallelizing the sub-windows may dramatically decrease execution time as the sub-window calculation is where the algorithm spends most of its time. So, instead of parallelizing the image pyramid, the focus could be shifted to parallelizing the sub-window sweeping such that there are multiple sub-windows sweeping the image to detect faces instead of a single sub-window.

References

[1] Altera. (2016, March). "Cyclone IV FPGA Device Family Overview" [online]. Available: https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/hb/cyclone-iv/cyiv-51001.pdf [May 3, 2016]

[2] Analog Devices. (2010, July) "ADV7123" [online]. Available: http://www.analog.com/media/en/technical-documentation/data-sheets/ADV7123.pdf [May 3, 2016].

[3] C. Wright, "YUV Colorspace", *Softpixel.com*, 2004. [Online]. Available: http://softpixel.com/~cwright/programming/colorspace/yuv/. [May 3, 2016].

[4] "Design Software - Overview", *Altera.com*, 2016. [online]. Available: https://www.altera.com/products/design-software/overview.tablet.html. [May 3, 2016].

[5] D. Hefenbrock, J. Oberg, N. Thanh, R. Kastner and S. Baden, 'Accelerating Viola-Jones Face Detection to FPGA-Level Using GPUs', *2010 18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines*, 2010.

[6] "DROK® VGA OV7670 640X480 Camera Sensor Module Lens CMOS SCCB Interface Compatible With I2C Interface : Electronics", *Amazon.com*, 2016. [Online]. Available: http://www.amazon.com/sunkee-OV7670-640X480-Compatible-Interface/dp/B00AZWVZKW/ref=sr_1_1?s=electronics&ie=UTF8&qid=1379557438&sr=1-1&keywords=CMOS+OV7670+Camera. [May 3 2016].

[7] M. Field, "OV7670 camera", *Hamsterworks.co.nz*, 2016. [online]. Available: http://hamsterworks.co.nz/mediawiki/index.php/OV7670_camera. [May 3, 2016].

[8] Omnivision. (2005, July). "OV7670/OV7171 CMOS VGA (640x480 CameraChip with OmniPixel Technology" [online]. Available: http://www.voti.nl/docs/OV7670.pdf [May 3, 2016].

[9] Omnivision. (2005, Sept.). "OV7670/OV7171 CMOS VGA(640x480) CameraChip Implementation Guide" [online]. Available: http://www.haoyuelectronics.com/Attachment/OV7670%20+%20AL422B%28FIFO%29%20Camera%20Module%28V2.0%29/OV7670%20Implementation%20Guide%20%28V1.0%29.pdf [May 3, 2016].

[10] R. Glass, 'Frequently forgotten fundamental facts about software engineering', IEEE Softw., vol. 18, no. 3, pp. 112-111, 2001.

[11] Sites.google.com, 'Viola-Jones Face Detection - 5KK73 GPU Assignment 2012', 2015. [Online]. Available: https://sites.google.com/site/5kk73gpu2012/assignment/viola-jones-face-detection. [Accessed: 27- Oct- 2015].

[12] Terasic Technologies, "DE2-115 User Manual", Altera.com, 2010. [online] Available:ftp://ftp.altera.com/up/pub/Altera_Material/Boards/DE2-115/DE2_115_User_Manual.pdf [May 3, 2016].

[13] Terasic Technologies, "Terasic - DE Main Boards - Cyclone - Altera DE2-115 Development and Education Board", *Terasic.com.tw*, 2016. [online]. Available: http://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&No=502. [May 3, 2016].

[14] Viola, P.A., Jones, M.J.: Rapid Object Detection using a Boosted Cascade of Simple Features. IEEE Computer Society Conf. on Computer Vision and Pattern Recognition (CVPR), (2001)

Appendix

FPGA Pinout

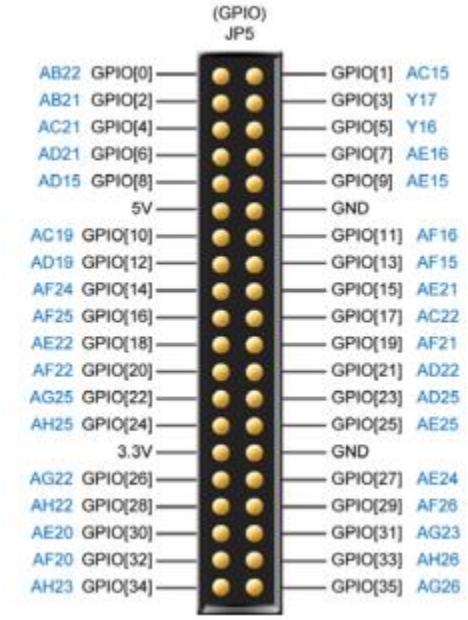


Figure A.1 - Dev Board GPIO Interface [12]

Port Name	FPGA Pin	Dev Board Interface	Notes
clk_50	PIN_Y2	CLOCK_50	50MHz external oscillator
slide_sw_RESET	PIN_Y23	SW[17]	active high
slide_sw_resend_reg_values	PIN_Y24	SW[16]	active high
slide_sw_capture_mode	PIN_AA22	SW[15]	active high
btn_capture	PIN_M23	KEY[0]	active low
LED_config_finished	PIN_F19	LEDR[1]	camera configured
LED_dll_locked	PIN_G19	LEDR[0]	PLL is locked
ii_gen_done	PIN_AG26	GPIO[35] (JP5)	debug
subwin_done	PIN_AF26	GPIO[29] (JP5)	debug
faceBox_done	PIN_AE24	GPIO[27] (JP5)	debug
measure_performance	PIN_AH23	GPIO[34] (JP5)	toggle after every faceBox_done

Table A.1 - FPGA Clock, User Interface, Debug and Performance Measurement Connections

Port Name	FPGA Pin	Dev Board Interface	Camera Pin Name	Notes
na	nc	3.3V (JP5)	3V3	
na	nc	GND (JP5)	GND	
ov7670_pclk	PIN_AC19	GPIO[10] (JP5)	PCLK	
ov7670_xclk	PIN_AF16	GPIO[11] (JP5)	XCLK	
ov7670_vsync	PIN_AF25	GPIO[16] (JP5)	VSYNC	
ov7670_href	PIN_AC22	GPIO[17] (JP5)	HREF	
ov7670_data[7]	PIN_AE16	GPIO[7] (JP5)	D7	
ov7670_data[6]	PIN_AD21	GPIO[6] (JP5)	D6	
ov7670_data[5]	PIN_Y16	GPIO[5] (JP5)	D5	
ov7670_data[4]	PIN_AC21	GPIO[4] (JP5)	D4	
ov7670_data[3]	PIN_Y17	GPIO[3] (JP5)	D3	
ov7670_data[2]	PIN_AB21	GPIO[2] (JP5)	D2	
ov7670_data[1]	PIN_AC15	GPIO[1] (JP5)	D1	
ov7670_data[0]	PIN_AB22	GPIO[0] (JP5)	D0	
ov7670_sioc	PIN_AF24	GPIO[14] (JP5)	SIOC	
ov7670_siod	PIN_AE21	GPIO[15] (JP5)	SIOD	
ov7670_pwdn	PIN_AD19	GPIO[12] (JP5)	PWDN	active high
ov7670_reset	PIN_AF15	GPIO[13] (JP5)	RESET	active low

Table A.2 - FPGA to OV7670 Camera Interface Connections

Port Name	FPGA Pin	Dev Board Interface	Notes
na	nc	VGA_R (J13)	analog out from DAC (U7)
na	nc	VGA_G (J13)	analog out from DAC (U7)
na	nc	VGA_B (J13)	analog out from DAC (U7)
vga_hsync	PIN_G13	VGA_VS (J13)	
vga_vsync	PIN_C13	VGA_HS (J13)	
vga_r[7]	PIN_H10	DAC (U7)	
vga_r[6]	PIN_H8	DAC (U7)	
vga_r[5]	PIN_J12	DAC (U7)	
vga_r[4]	PIN_G10	DAC (U7)	
vga_r[3]	PIN_F12	DAC (U7)	
vga_r[2]	PIN_D10	DAC (U7)	
vga_r[1]	PIN_E11	DAC (U7)	
vga_r[0]	PIN_E12	DAC (U7)	
vga_g[7]	PIN_C9	DAC (U7)	
vga_g[6]	PIN_F10	DAC (U7)	
vga_g[5]	PIN_B8	DAC (U7)	
vga_g[4]	PIN_C8	DAC (U7)	
vga_g[3]	PIN_H12	DAC (U7)	
vga_g[2]	PIN_F8	DAC (U7)	
vga_g[1]	PIN_G11	DAC (U7)	
vga_g[0]	PIN_G8	DAC (U7)	
vga_b[7]	PIN_D12	DAC (U7)	
vga_b[6]	PIN_D11	DAC (U7)	
vga_b[5]	PIN_C12	DAC (U7)	
vga_b[4]	PIN_A11	DAC (U7)	
vga_b[3]	PIN_B11	DAC (U7)	
vga_b[2]	PIN_C11	DAC (U7)	
vga_b[1]	PIN_A10	DAC (U7)	
vga_b[0]	PIN_B10	DAC (U7)	
vga_blank_N	PIN_F11	DAC (U7)	
vga_sync_N	PIN_C10	DAC (U7)	
vga_CLK	PIN_A12	DAC (U7)	

Table A.3 - FPGA to VGA Display Interface Connections

FPGA Design Metrics

It is important to detail particular metrics from FPGA based design such as resource utilization, power consumption estimation, maximum clock frequencies and longest path delay. These metrics are quantitative benchmarks that can be compared between other similar designs or other FPGA hardware. This information was included for completeness, even though the original customer needs specification did include a requirement for this information,

Resource Utilization

Table A.5 was reported after full design compilation in the Flow Summary of Quartus II.

Resource Utilization	
Flow Status	Successful - Fri Apr 08 09:07:51 2016
Quartus Prime Version	15.1.0 Build 185 10/21/2015 SJ Lite Edition
Revision Name	faceDetectSystem
Top-level Entity Name	top
Family	Cyclone IV E
Device	EP4CE115F29C7
Timing Models	Final
Total logic elements	33,327 / 114,480 (29 %)
Total combinational functions	29,079 / 114,480 (25 %)
Dedicated logic registers	16,035 / 114,480 (14 %)
Total registers	16035
Total pins	56 / 529 (11 %)
Total virtual pins	0
Total memory bits	2,175,501 / 3,981,312 (55 %)
Embedded Multiplier 9-bit elements	369 / 532 (69 %)
Total PLLs	1 / 4 (25 %)

Table A.5 - FPGA resources used

Power Consumption

Table A.6 was reported after full design compilation by the PowerPlay Power Analyzer tool within Quartus II. By default board, junction and ambient temperatures are 25°C while “input I/O signal toggle rate” and the “remaining signals toggle rate” were set to 50%.

Power Estimation	
PowerPlay Power Analyzer Status	Successful - Fri Apr 08 09:07:51 2016
Quartus Prime Version	15.1.0 Build 185 10/21/2015 SJ Lite Edition
Revision Name	faceDetectSystem
Top-level Entity Name	top
Family	Cyclone IV E
Device	EP4CE115F29C7
Power Models	Final
Total Thermal Power Dissipation	2497.39 mW
Core Dynamic Thermal Power Dissipation	2300.88 mW
Core Static Thermal Power Dissipation	113.59 mW
I/O Thermal Power Dissipation	82.92 mW
Power Estimation Confidence	Low: user provided insufficient toggle rate data

Table A.6 - FPGA power usage

Timing Analysis

Table 8 was reported by TimeQuest Timing Analyzer in Quartus II. Fmax is the maximum frequency of the top level control and subwindow processes. The longest delay was determined to be the longest data delay with respect to this clock.

Timing Analysis - Slow 1200mV 85C Model	
Fmax	42.17 MHz
Longest Data Delay	23.799 ns

Table A.7 - Subwindow timing analysis