EE 552 2000f 2000-11-24

EE 552 Final Report

# Image Indexing Processor

**INSTRUCTOR:**

Dr. Elliott

**GROUP MEMEBERS:**

*Yuxin Wang, yxwang@ee.ualberta.ca*

*Yunan Xiang, ynxiang@ee.ualberta.ca*

*Julien Lamoureux, lamoureu@ee.ualberta.ca*

*Chuping Liu, chopin@ee.ualberta.ca*

# TABLE OF CONTENTS

## Declaration of Original Content

The design elements of this project and report are entirely the original work of the authors except as follows:

- VGA display module refers to reference [11]
- SRAM module will be modified from reference [5,6]
- Parallel port interface refers to reference [12]

Members' signatures:

Chuping, Julien, Yunan, Yuxin

## Abstract

This document describes the design, implementation, and verification of an image-indexing processor implemented on an Altera Flex10k FPGA , which is mounted on a UP1 board. A histogram-indexing algorithm is used to compare a query image to a database of images with the goal of finding the images that most closely match the query image. The images are sent to the FPGA from a PC through a parallel port, where they are compared, and then ranked in order of decreasing similarity. After comparing all the images, the index of the most similar images are displayed on LEDs, which are also mounted on the UP1 board.

## Achievements

1. The top level processor – worked (not implemented in UP1 board yet)
2. Top control unit (TCU) – worked in simulation
3. Indexing Engine – worked in simulaltion
4. Parallel port under Windows NT – worked with real connection from PC to the board, the transmission rate can reach 734Kbits/sec.
5. Communication between PC and FPGA – worked
6. VGA display – can display 64 color with acceptable flicking at a higher scanning rate than normal.
7. SRAM – not done yet

## Algorithm Introduction

### *Principal of Image Indexing*

The objective of image indexing is to retrieve similar images from an image database for a given query image (i.e., a pattern image). Each image has its unique feature. Hence image indexing can be implemented by comparing their features, which are extracted from the images (see Fig.1). The criterion of similarity among images may be based on the features such as color, intensity, shape, location and texture, etc.

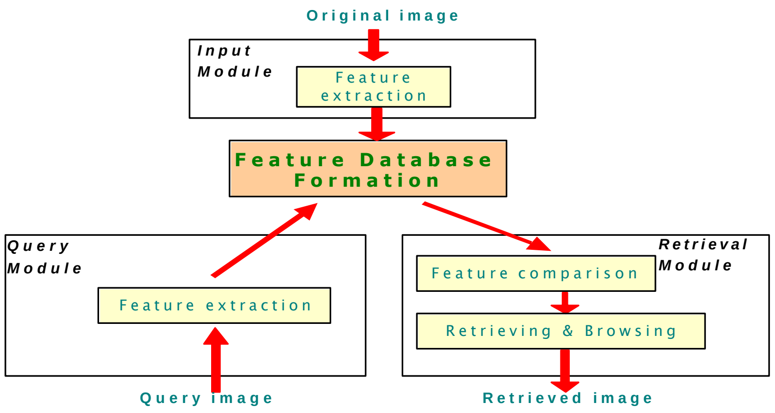


**Figure 1 Block diagram of principal of image indexing**

## Histogram

A histogram is a method used to describe the frequency distribution of a digital signal. Histograms are composed of multiple bins, with each bin corresponding to a range of values. The value of each bin is obtained by summing the occurrences of digital signal samples whose values fall in the domain of that corresponding bin. For example: An 8-bit gray-level picture is composed of N total pixels whose intensity values can range from 0 to 255. Using a 16-bin intensity histogram, with even bin intervals of 16 (here, the bin interval may be variable from 1 to 256), the image's intensity frequency distribution can be plotted as shown below (see Fig.2). The corresponding value of each bin $b_k$, ($0 \le k \le 15$), is the number of of pixels $n_k$, whose intensity value fall in the range of bin $b_k$, i.e.:

$$b_k = n_k, \qquad (0 \le k \le 15)$$
$$(n_0 + n_1 + \cdots + n_k + \cdots + n_{15} = N)$$

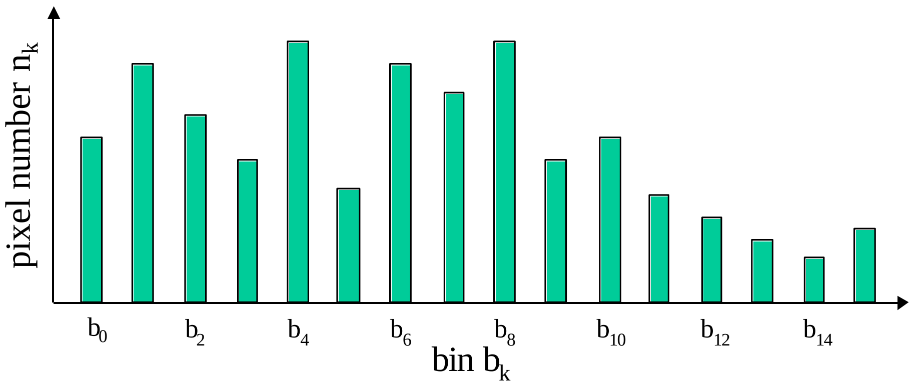


**Figure 2 A histogram example**

## Vector Quantization

Vector quantization (VQ) is an efficient compression technique used in digital signal processing. A digital signal is a sequence of bit vectors. The idea behind VQ is to map the vectors of a digital signal to a set of vectors, which already exist in the form of a look-up table (i.e. codebook). The vectors in the codebook are called codewords. Vectors are mapped by choosing the codeword that most closely matches the original vector. The criterion for choosing the codeword may change depending on the nature of the signal in question. By limiting the number of codewords in the codebook, the original signal is compressed while keeping most of its original characteristics.

Moreover, if both the sender and receiver of the digital signal have the same codebook, only the codeword labels of a signal need to be sent through a channel. The receiver can then reconstruct the signal by matching the received labels to their corresponding codewords. In this way, VQ allows for efficient transmission and storing of a signal.

### *Uniform Quantization*

Vector quantization is a very efficient compression technique, however, fully implementing VQ would place demands for FLEX10K chip. Uniform quantization (UQ) is a more basic alternative to VQ and is more suitable to this project.

Similar to VQ, UQ uses a codebook, which is also pre-generated. In UQ, the range of a digital signal is divided into *m* even intervals. If an input signal $v_{il}$ falls into the $l^{th}$ interval ($1 \leq l \leq m$), the value of the output signal $v_{ol}$ is assigned the median value of its interval, to which a label is assigned in the meanwhile.

When processing the input signals, instead of using the original signal value, the corresponding codeword label is employed. For example: A digital image whose pixel values range from 0 to 255 is quantized into 16 intervals. The corresponding output and label assignment of each interval is shown in table 1.

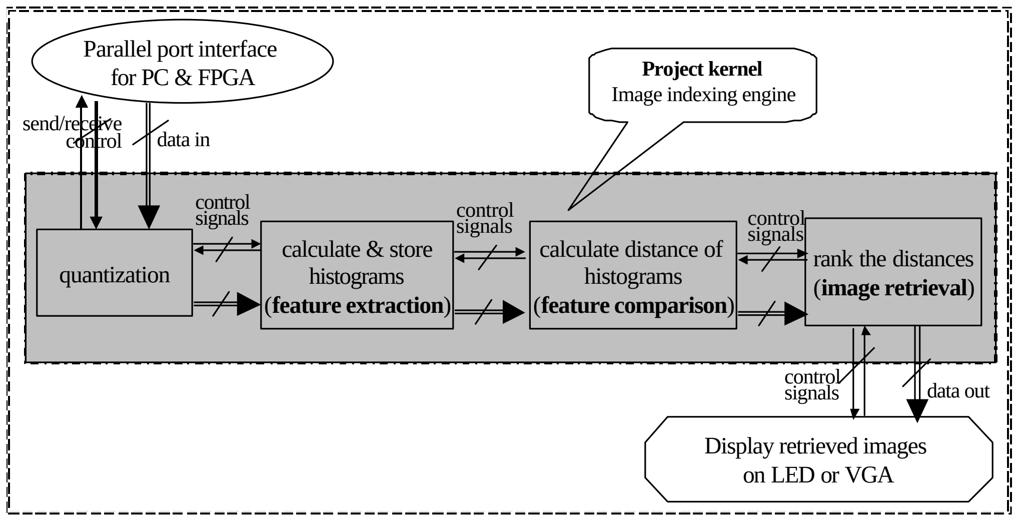
Although UQ cannot achieve the high compression performance as VQ, it can still save storage space to some degree without much distortion. As the example illustrated, a 256-level input that required 8 bits, required only 4 bits after even quantization was employed.

**Table 1 A color component distribution range and its quantization outputs and labels**

| Range | Label | Output | Range | Label | Output |
|---|---|---|---|---|---|
| 0 – 15 | 0 | 8 | 128 – 143 | 8 | 136 |
| 16 – 31 | 1 | 24 | 144 – 159 | 9 | 152 |
| 32 – 47 | 2 | 40 | 160 – 175 | 10 | 168 |
| 48 – 63 | 3 | 56 | 176 – 191 | 11 | 184 |
| 64 – 79 | 4 | 72 | 192 – 207 | 12 | 200 |
| 80 – 95 | 5 | 88 | 208 – 223 | 13 | 216 |
| 96 – 111 | 6 | 104 | 224 – 239 | 14 | 232 |
| 112 – 127 | 7 | 120 | 240 - 255 | 15 | 248 |

## Description of Operation

This project consists of an image-indexing algorithm implemented in hardware, which is interfaced with a personal computer through a parallel port. More Specifically, the algorithm is implemented on a Flex10k FPGA mounted on a UP1 board. Images are sent from the personal computer to the FPGA, first a query image, then the candidate images that make the image database. The candidate images are compared to the query image using the histogram-indexing algorithm. After processing each image,i.e., extracting the distance of each candidate image, the ranks of the images that match the orginal query image can be displayed on LEDs on the UP1 board. Finally, the computer or probably the VGA controlled by FPGA will be used to display the images in the order in which they were ranked. Figure 3 illustrates the overall system.



Parallel port interface for PC & FPGA
Project kernel
Image indexing engine
send/receive control
data in
control signals
control signals
control signals
quantization
calculate & store histograms (feature extraction)
calculate distance of histograms (feature comparison)
rank the distances (image retrieval)
control signals
data out
Display retrieved images on LED or VGA

**Figure 3 High Level Image Indexing Block Diagram**

The image indexing algorithm and hardware interfacing is implemented with three major components. These components are the Top Control Unit, the Interface Unit, and the Image-indexing Engine.

## ***Components***

### Top Control Unit

The top control unit (TCU) controls the two other major components of the image-indexing processor, i.e., the Interface Unit (IU) and the Image-Indexing Engine (IIE). The TCU is implemented with a large finite state machine, which has 21 states to guide the IU and IIE through its various processing stages. See Appendix C.2 for the state diagram.

### Image-Indexing Engine

The image-indexing engine is the heart of the image-indexing processor and it is implemented in an engine controller together with 5 sub-module blocks. Each block, apart from the EAB ram, is employed using mini-controller to control the block's behavioral data-path. The engine controller controls the states of the whole index engine and it activate and disable the sub-modules in the appropriate time. Although there are also some small controllers in the sub-modules, we still call them parts of data path due to the relationship between them and the engine controller.

#### *Control Path*

The control path for the image-indexing engine is a finite state machine (Moore machine) that changes states based on the primary inputs of the image-indexing engine, which come from the top control unit and the sub-modules that it controls. The state machine has 12 states that are used to navigate the sub-modules through its various processing stages, which are described below. See Appendix A.4 for an illustration of the state diagram.

## Data Path

The image-indexing engine data path does all of the work including the histogram building, distance calculation, distance sorting and ranking of all input images. The data path is composed of seven components: memory initialization (whole EAB rams initialization) module, the query and candidate image histogram builder (combination of uniform quantization and histogram building), the candidate image histogram ram initializaion module, the gray-level converter, the distance calculator,the image ranker, and the ram manager. See Appendix A.5 for an illustration of the data path.

### *Whole EAB rams initialization module*

This block is used to initialize the FPGA's internal memory (EAB cells) before building any histograms. Each of the memory locations, which are used to store histograms, are initialized to '0'.

The memory initializer is composed of a control path and a datapath. The control path has three state, namely "idle", "init", and "ready". In the "init" state, a counter is used the increment an address from 0 to 47, which is used to write "0" into each memory location occupied by the query histogram. See Appendix A.6 for the control path.

### *Candidate image histogram initialization*

Similarly, this component is used to clear the candidate image histogram memory, whenever we finish the distance calculation for each candidate image. The control path is implemented in a similar fashion also.

### *Gray level converter*

The gray-level converter converts color image data into a gray-level image equivalent. This component is only used when the database contains both color and gray-level images. In this case, all color images are converted to gray-level images to allow for a comparison. This option can be disabled if not desired.

In our final design, due to the limited number of logic cells of the chip, we cut this part out during the final integration of all the components.

### *Uniform quantization & Histogram calculation*

The histogram builder builds an image's histogram one pixel at a time using uniform quantization (static codebook has not been adopted due to the time constaint). Because the histogram is stored in memory, the histogram builder must use registers to load, add, and then store information, and is therefore not purely combinational as would be expected by a conventional data-path component. See Appendix A.7 for datapath and control path.

*Distance calculation*

The distance calculator sums the total absolute difference between each of the 16 bin values of the query image and the current image being processed. Please see the Appendix A.8 for the datapath and control path.

*Sorting rank*

After obtaining a candidate image's distance, it is compared to the current list of smallest distances (insert sorting) and stored in its correct position in the list by the image ranker. Please see Appendix A.9.

*EAB ram manager & Connection*

The EAB ram manager is used to control the connection between the functional modules and the EAB rams. We implemented it by a pure combinational logic, i.e., for each pieces of ram block (query histogram ram and current candidate histogram ram), using several 4 to 1 mux to select the module to be connected to the rams. The 2 bit select signals is actually decoded from 3 control signals come from the engine manager.

Since there are 12kbits internal ram in FLEX10k, each data width, which we need, is 16 bits and the total ram disk space is no more than 2.5 kbits, 8 pieces of internal ram had originally been designed. But when we began to instantiate the first piece of ram for histogram of query image, the actual 768bits ram (3 pieces) with 3 sets of input and output pins used up all 6 blocks of EABs. That means we couldn't instantiate all pieces of ram in our design. We found that not only the capacity of ram will affect the usage percentage of EABs (as we usually think), but also the number of input and output IO pins, which means the wider the address and data width, or the more pieces of rams instances you use, the more EABs will be occupied.

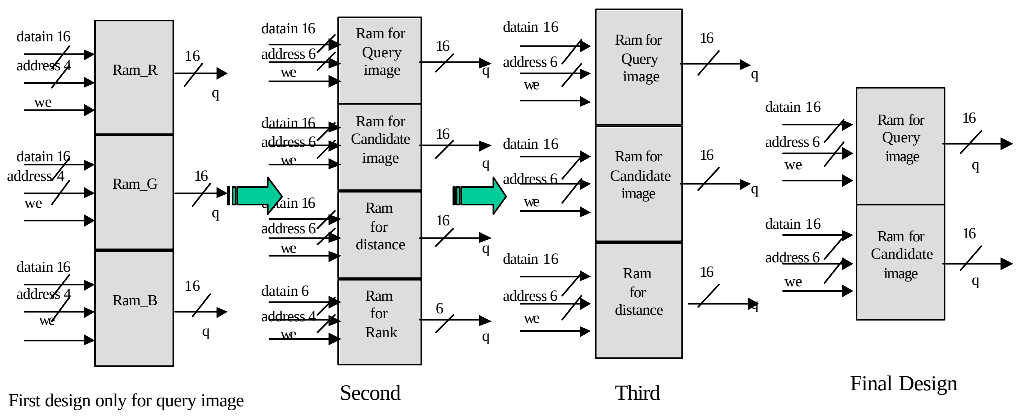
Because no solution was found for this problem, the histograms for the R, G and B components of one image were combined together by using only one piece of ram instead of 3 pieces for R,G, B separately. Therefore, we can only use one set of input and output pins to read and write the ram block sequentially for R, G, B instead of reading/writing the R,G,B in parallel. Since R,G, B has 16 bins, we will use a 4 bit address bus (16 words) if we use three ram instances. By combining the three pieces together, we have to use a 6 bit address bus to represent 48 words. This way,the total number of pieces of ram was reduced to 4, one for query image histogram, one for current candidate image histogram, one for distance, and one for rank, but they still didn't fit.

Finally, by storing the images' updated ranks in registers instead of in the internal ram, enough memory was made available for storing the histograms and distances. The 3 ram instances take all 6 full EAB blocks when only 48(words) $\times$ 16(bits) $\times$ 3 = 2304 bits are required (Figure 3 and EAB_rams.vhd).

However, in our final design, we end up by using only 2 pieces of internal ram for query image and candidate image and cut the distance ram off. The distance is calculated after we get the histogram of each new candidate image and it is directly used to calculate and update the ranks so that we do not need to store the distances of each candidate image any more.

Although Altera recommends Cycle-Cycle Dual-Port RAM (csdram) to instantiate the internal ram for Flex10k family (from Altera website), we still got the foregoing problems when we use csdram modules. Also, we find that even the percentage of used EABs decreases a little bit, but we pay the price of using some of the logic cells, which is not desirable. Another disadvantage of using csdram is that it is dual port ram, which makes the interface more complicated. We finally decided to use lpm_ram_dq instead of csdram.

The following figure simply illustrates how our design for EABs carried on.



This block is an important part in our indexing engine in that there are three functional blocks (two memory initialization blocks and the histogram calculater) need to write data from it and two block needs to read data (histogram builder and distance calculater).

We have to take care about the connections among these blocks in order to avoid contension of the data bus and address bus for query image and candidate image, respectively. Same with the write enable signal "we".

Please see Appendix A.10 for illustration of the multiplexors employed to deal with the forgoing problem.

## Indexing counter

This component is a simple counter which assigns a number to each candidate image being processed. This number is the images' label, and this label is displayed on LEDs when the most similar images are being declared after all the images have been compared.

**<u>Summary</u>**: When we integrated all of components together and simulated the top level VHDL file for engine, we found the engine took so much percent of logic cells for Flex10k240RC-4 chip. The following table demonstates the comparasion among the different pixels of image, data width for histogram and distance and the number of display for rank (**without gray level converter**).

| case | # of pixel per image | # of first top rank display | Data width for histogram and distance(bits) | Logic cells (total:1152) | Registered performance(clock period:ns) |
|---|---|---|---|---|---|
| 1 | 128x96 or 120x90 | 4 | 16 | 1028 (89%) | 72.1 |
| 2 | 64x48 | 4 | 14 | 950(82%) | 66.6 |
| 3 | 64x48 | 4 | 12 | 870(75%) | 60.4 |
| 4 | 64x48 | 2 | 12 | 764(66%) | 58.5 |

Since there are still about more than 200 logic cells to be added for final processor VHDL file, we take the case 4 to simulate our engine. One should note that in the following analysis, we still base on the 16 bits data width and 4 rank display.

## Interface Unit

The interface unit is used to allow communication between the personal computer and the UP1 board. The interface requires control logic on at both ends. On the PC side, a driver is used to control the parallel port. On the FPGA side, a controller will take over the management of the interface.

### *PC Side*

Because information is sent from the PC to the FPGA uni-directionally, the parallel port is implemented using the SPP mode. To program the parallel port in SPP mode, three registers are used. "The port base address (I/O Base) is used to write to the data lines. The parallel port status register address is at the port base address plus one and is used to read the status of the parallel port. This will allow for reading the ACK signal coming from the UP1 board. The parallel port control register is at the port base address plus two and is used to control the parallel port. This will allow setting the TRANSFER and STROBE signals" [12]. The parallel port SPP mode register definitions are as follows:

| Label | Bit 7 | Bit 6 | Bit 5 | Bit 4 | Bit 3 | Bit 2 | Bit 1 | Bit 0 |
|---|---|---|---|---|---|---|---|---|
| Data | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
| Status | | /Ack | | | /RESET | | | |
| Control | | | | | /TRANSFER | IS_HEAD | IMGEND | /Strobe |

### *FPGA Side*

To interface a PC parallel port to the FPGA, 14 pins of the UP1 FPGA are used here. These pins include 4 input control signals from PC and 2 output status signal back to

PC. In the 4 control signals, one for strobe, one for enable, one for header_data, one for anImgEnd. The 2 status pin are for Ack and Reset, respectively. The left 8 pins for input data lines.

The pin layout from the PC parallel port is as follows:

| Name | Pin | Type of Pin | Description |
|---|---|---|---|
| STROBE | 1 | Output | Strobe to FPGA |
| IMGEND | 14 | Output | Tells FPGA a complete image has been sent |
| IS_HEAD | 16 | Output | Tells FPGA data port is sending header information |
| TRANSFER | 17 | Output | Transfer data (Start/Stop) |
| Reset | 15 | Input | Reset signal from FPGA |
| ACK | 10 | Input | Acknowledge from FPGA |
| Data | 2 – 9 | Output | Data Bit 0 – 7 |

These pins will connect to the UP1 FPGA as follows:

| PC Parallel Port Pin Name | UP1 FPGA Pin Name | Type of Pin |
|---|---|---|
| STROBE | pc_c_b0 | Input |
| IMGEND | pc_c_b1 | Input |
| IS_HEAD | pc_c_b2 | Input |
| TRANSFER | pc_c_b3 | Input |
| RESET | pc_s_b3 | Output |
| ACK | pc_s_b6 | Output |
| Data | pc_data_in | Input |

"The 10 input lines will have to be connected to the UP1 FPGA through 2 74LS245 buffers and the 2 output line will have to be connected through another 74LS245 buffer. This will ensure that the parallel port and FPGA do not damage one another."[12]

## Datasheet

### *Features*

- Image database are stored in a computer.

- The retrieved images are those which have the smallest distance from the query image with respect to the color-histogram algorithm.
- Due to the limit of the capacity of the FLEX10K, the size of the image database is only up to 64. Also, the small number of logic cells restrict the number of retrieved images. So far, only 2 images can be retrieved.
- When indexing images, all images have to be sent from PC to FPGA through the parallel port of the PC in SPP mode.
- The query image is the first image to be sent, followed by the all candidate images ready to be matched to the query one.
- The indices of the retrieved images will be displayed on LEDs. One dip switch is used to select the different rank number.

## I/O Signals

**Table 2 - FPGA I/O Pins**

| Signal Name | Type | Number of Pins | Description |
|---|---|---|---|
| Interface to Parallel Port | | | |
| PC_data_In | In | 8 | Parallel port data input to FPGA from PC |
| PC_s_b6 | Out | 1 | Handshaking line between the PC and FPGA board. Acknowlege the "Strobe" signal from PC. |
| PC_s_b3 | Out | 1 | Send "reset" signal to PC. |
| PC_c_b0 | In | 1 | Strobe signal from PC. |
| PC_c_b1 | In | 1 | "An image end" signal from PC |
| PC_c_b2 | In | 1 | "Sending header information" signal from PC |
| PC_cb3 | In | 1 | Starting or stopping to transfer data (i.e.,enable or disable the FPGA) signal from PC |
| Interface to LEDs | | | |
| Position | In | 1 | Used to select different ranking results |
| LED0, LED1 | Out | 16 | The rank of the retrieved images that will be displayed in LEDs. |
| Control Signals | | | |
| Push_to_done | In | 1 | Push the FLEX_PB1 to end the indexing task. |
| Reset<br>Clock_Sys | In<br>In | 1<br>1 | Reset<br>System Clock |

## Design Hierarchy

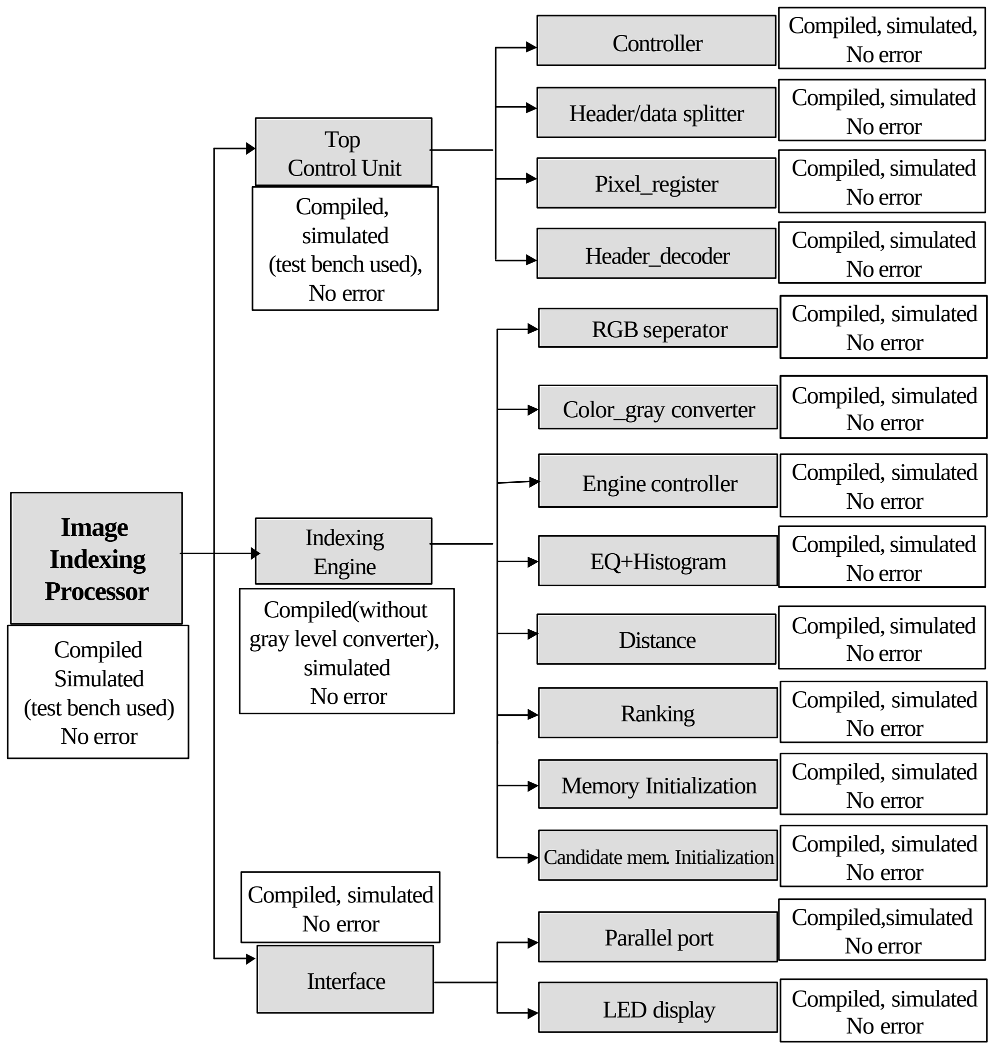


Image Indexing Processor
Compiled Simulated (test bench used) No error
Top Control Unit
Compiled, simulated (test bench used), No error
Indexing Engine
Compiled(without gray level converter), simulated No error
Interface
Compiled, simulated No error
Controller
Compiled, simulated, No error
Header/data splitter
Compiled, simulated No error
Pixel_register
Compiled, simulated No error
Header_decoder
Compiled, simulated No error
RGB seperator
Compiled, simulated No error
Color_gray converter
Compiled, simulated No error
Engine controller
Compiled, simulated No error
EQ+Histogram
Compiled, simulated No error
Distance
Compiled, simulated No error
Ranking
Compiled, simulated No error
Memory Initialization
Compiled, simulated No error
Candidate mem. Initialization
Compiled, simulated No error
Parallel port
Compiled,simulated No error
LED display
Compiled, simulated No error

**Figure 4 Diagram of design hierarchy**

## Experimental Results

### *Resource Requirements*

**Table 3 - Logic Cell Requirements and Registered Performance for Algorithm Components**

| Entity/Component | | Logic Cells Required | Registered Performance or Max. Matrix Delay (unit: ns) |
|---|---|---|---|
| Top Control Unit | Controller | 131 | 44.8 |
| | Header/Data Splitter | 33 | 13.9 |
| | Header Information Decoder | 7 | 11.1 |
| | Pixel Register | 69 | 19.9 |
| Subtotal: Top Control Unit | | 234 | 44.6 |
| Indexing Engine | RGB separate | 18 | 10 |
| | Color-gray converter | 256 | 19.8 |
| | Engine controller | 16 | 10.0 |
| | Internal ram | 0 | 8.6 |
| | Candidate memory clear | 37 | 20.3 |
| | Memory initialization | 37 | 20.3 |
| | Histogram | 224 | 25.3 |
| | Distance | 346 | 71.4 |
| | Ranking | 207 | 74.0 |
| Subtotal: Image Indexing Engine | | 1141 | 74.1 |
| Interfaces | Parallel Port Interface | 13 | 16.9 |
| | LEDs Controller | 68 | 8.2 |
| | LEDs | 32 | 30.5 |
| Clock Demultiplier | | 2 | 8 |
| Total (without the gray-level converter) | | 1054 | 106.5 |

Note: the engine part is based on the 16 bits data width and 4 rank display.

## *Speed vs. Area*

The speed of the image-indexing processor depends mostly on the design of the image-indexing engine. Most of the engine's components are directly in the processors critical path. Some of the engine's components, namely the Histogram Builder and Ranker can be implemented to increase speed in exchange for more area, or decrease area at the expense of speed.

The Histogram Builder could be implemented to build the R, G, and B histograms either concurrently or sequentially. A concurrent implementation would increase the component's throughput by a factor of three (3 color components), but would increase its area consumption by an equal factor. In the last report, the R, G, B histograms were built concurrently, however this time they are built sequentially to save area.

In the preliminary design of the image-indexing datapath, the image ranking was done after all of the images' distances had been tabulated. This involved storing both the distance and the index of each of the 64 images into memory, and then sorting them. This scheme required allot of memory, as well as considerable time at the end of the indexing process. A second, more suited solution was devised, where the sorting of the images occurs after each of their distances were calculated. This type of sorting is done concurrently when the histogram building is occuring, thus eliminating the long delay at the end of the indexing process. Also, by keeping only the N best results, it became feasible to store the distance and index of each image in registers rather than in internal memory, which is already in very low supply.

For each new distance value calculated the ranking algorithm must insert the value into the appropriate position of the list and then shift the all of the values below that position down by one. This implementation is fast because the value to be held at each position in the list is chosen concurrently. This differs from software implementations where the CPU must traverse through memory in a sequential manner to sort a list. After compiling this implementation, a problem revealed itself. The area required for each position in the list was 90 cells. This is quite high and wouldn't allow us to keep track of very many image rankings. Further analysis of the design was required.

A revised implementation, which is implemented using RTL rather than behavioral achieves the identical speed, but requires significantly less area. The savings in the area is accomplished by sharing a comparator between each stage of the ranking list. In the preliminary implementation, each stage compared both (above > new) and (current > new). Upon inspection, it can be seen that both the current and above values are compared to the new value. So by placing a single comparator between each stage, it can be used as the above comparator for the stage below it, and as the current comparator for the stage above it. This is clearly illustrated in appendix C.5.

## ***64-Color VGA Display Test***

### Introduction

A standard VGA monitor contains of a grid of pixels typically 480 rows and 640 columns. Each pixel can display various colors, depending on the state of the red, green and blue signals (R, G and B) . A VGA monitor also has an internal clock that determines when each pixel is refreshed, typically 25.175 MHz. The refresh behavior of the monitor is partially controlled by the horizontal and vertical synchronization signals. After the first pixel is refreshed, the monitor refreshes the remaining pixels in the row in the raster scan mode. When the monitor receives a horizontal synchronization signal, it retraces and then starts to refresh the second row also in raster scan mode. By repeating this process until the scan reaches the bottom of the monitor, the monitor will receive a vertical synchronization signal and retrace back to the first pixel (top left corner). Although stated above that normally the monitor only scans 640 pixels each row, actually it uses 800 clock periods of which 640 are used to scan the pixels which can be seen on the screen. The remaining 160 clock periods are used to retrace back (moving the CRT from the end of the row to the beginning) and also the synchronization. A similar process applies to the vertical synchronization, which uses 525 instead of 480 horizontal synchronization pulses. So the screen refresh frequency is typically 60 Hz.

### Problem

Fortunately the clock frequency of the UP1 board used is the same as the VGA operating frequency. So if in 800*525 clock periods, we send 640*480 signals, including RGB, i.e., a 640*480 8-color picture data, we can get a steady display of that picture. However, the picture which can be displayed is limited to 8-color only, because 3 bits – R, G, B are used to represent the color and each bit can only have two states – 0 and 1. Therefore there is a big problem: 8-color (3 bits) is too less to display a natural picture which is generally 16-bit or higher (24-bit for true color) per pixel.

### Attempt to reach 64 colors

The solution of the problem is similar to develop a VGA display driver which can display 64 colors on the screen. In theory, if we can expand the 1-bit RGB to 2-bit RGB state, we can have a display of $2^6$=64 available colors. The following scheme is the trick used to have a virtual 2 bit RGB state:

1. The picture data we use for the indexing is true color. It is easy to extract the RGB data and round them up to have 2 bits with 4 states: 11, 10, 01, 00.

2. We need to find the way to convert the 4 states of each RGB into 2 states: 0 or 1 because the screen can indeed show 8 colors pattern for every refresh screen (i.e., 1/60 second). But our eyes can be cheated by displaying same data in slightly different color patterns and acknowledge that the displayed pattern has 64 colors.

3. One example is that we can use a 2-bit counter in 3 states (00, 01 and 10), and by doing simple logic calculation, the 2-bit RGB data can be converted into 1-bit states. The counter is increased by 1 at every rising edge of the vertical

sychronization pulse. Then for a 3 continuous refresh screen clock (3*(1/60) seconds), the monitor receives the signal of the following RGB states:

| value | | R or G or B | | | |
|---|---|---|---|---|---|
| | | 00 | 01 | 10 | 11 |
| Counter | 00 | 0 | 0 | 0 | 1 |
| | 01 | 0 | 0 | 1 | 1 |
| | 10 | 0 | 1 | 1 | 1 |

4. The above way actually behaves like that we divide the refresh screen frequency by 3 into 20 Hz. So another problem appears that the screen is flicking very heavily to our eyes since only the refresh screen frequency is higher than about 45 Hz we will not perceiv e the flicking.

5. We have to accelerate the refresh screen frequency by increasing the horizontal and vertical synchronization frequency. That is, for example, we will send the horizontal synchronization signal to the monitor every 500 internal clock time instead of 800 and also the vertical synchronization signal every 300 internal clock time instead of 525.

## Test results

The following table lists the comparison of screen display performance for different parameters used:

| $f_{h\text{-}synch}$ (unit:clock period) | $f_{synch}$ (unit:horizontal synchronization pulse) | Refresh Rate | Screen Flicking |
|---|---|---|---|
| 800 | 525 | 60 | very heavily |
| 700 | 451 | 80 | heavily |
| 600 | 373 | 112 | slightly |
| 560 | 357 | 126 | acceptable |
| 500 | 271 | 186 | stable |

## Comments

(1) Obviously the trick used in this method to display more than 8-color can only be used to display patterns smaller than about 400*200 pixels, if we want to have a stable display.

(2) Because of the different properties of display monitor, some monitors can not accept refresh rate higher than a certain value. This experiment results were obtained on a Panasonic PanaSync E70i type monitor.

## VHDL Code (all are zipped in index.zip file)

### *VHDL index*

1. Top level processor:
   - **top_proc.vhd –** the top level processor
     - *compiled and simulated with no errors*
     - *The top entity of the whole design*
2. Top Control Unit:
   - **tcu.vhd –** TCU
     - *simulated, no known bugs*
     - *used to control the whole design, a unit in parallel with the indexing engine unit and interface unit.*
   - **controller.vhd** – Controller
     - *simulated, no known bugs*
     - *used as a component with a finite state machine*
   - **splitter.vhd –** Header/data splitter
     - *simulated, no known bugs*
     - *used as component to split the raw data from PC*
   - **pixel_reg.vhd –** Pixel register
     - *simulated, no known bugs*
     - *used as component to convert the serial 8-bit RGB components, which are input from PC, to parallel 24-bit pixels, which will be sent to the indexing engine.*
   - **head_dec.vhd –** Header decoder
     - *simulated, no known bugs*
     - *used to decode the header information comes from PC*
3. Indexing Engine Unit:
   - **engine_1.vhd –** top level interface for index engine
     - *compiled and simulated with no errors*

  - *top level circuit for index engine, including the engine controller and several sub- modules, data path*

- **controller_final.vhd –** Indexing engine controller
  - *compiled and simulated with no errors*
  - *controller of the index engine*
- **index_counter.vhd –** counter for index of each candidate image
  - *compiled and simulated with no errors*
  - *a counter used to keep trace of the index of each candidate image*
- **EAB_rams1.vhd –** EAB ram file
  - *compiled and simulated with no errors*
  - *used to create two pieces of ram blocks in EAB to store the histograms for query and candidate images*
- **mem_init.vhd –** ram initialization file
  - *compiled and simulated with no errors*
  - *used to initialize the internal ram (rams for storing query and candidate image histograms) each time when a query image is to be retrieved from a group of candidate images (each operation)*
- **gray_converter.vhd –** Gray level conversion
  - *compiled and simulated with no errors*
  - *used to generate a gray image histogram by using a color image*
- **color_hist1. vhd –** Color histogram calculation
  - *compiled and simulated with no errors*
  - *used to build the histogram for color (query and candidate) images*
- **mem_cand_init. vhd –** memory initialization file
  - *compiled and simulated with no errors*
  - *used to initialize the ram block storing the candidate image each time a new candidate histogram is to be built*
- **dist_ctl. vhd –** the controller for distance calculation
  - *compiled and simulated with no errors*
  - *controller for distance calculation of a candidate image*
- **dist_calculation. vhd –** distance calculation

- *compiled and simulated with no errors*
- *behavioural data path used to calculate the distance between a candidate image and the query image*

- **distance_top. vhd –** distance calculation top level circuit
  - *compiled and simulated with no errors*
  - *top level circuits for distance calculation, including the controller and the data path*
- **dist_rank.vhd –** rank calculation
  - *compiled and simulated with no errors*
  - *used to sort the images in descending order based on their rank*
- **gray_converter.vhd –** convert color to gray level
  - *compiled and simulated with no errors*
  - *used to convert input data from color to gray level*

4. Interface Unit:

- **ppi2pc.vhd –** Parallel port interface
  - *implemented*
  - *used to connect the signals between PC and board*
- **disp_led.vhd –** LEDs display controller
  - *implemented*
  - *used to display rank of image on LEDs*
- **led_hex.vhd –** Led display
  - *implemented*
  - *used as a component for Led display in testing parallel port*

5. System clock demulitplier

- **clock_scaledown.vhd –** scaledown clock frequency
  - *implemented*
  - *used to provide clock whose frequency slower than the system clock in order to meet the design requirement.*

6. Used for demo:
    - **disp_parport_test.vhd –** Parallel Port Interface VHDL code for testing
        - *implemented*
        - *used to test the parallel port in FPGA board*
7. 64-color VGA display test unit:
    - **vga_test.vhd –** the top level processor
        - *implemented*
        - *used to test multiple color display*
    - **syncgen.vhd**– synchronous signals generator
        - *implemented*
        - *used to generate the horizontal and vertivcal synchronous signals*
    - **RGB_gen.vhd –** RGB signals generator
        - *implemented*
        - *used to convert 2-bit R,G,B signals to 1-bit R,G,B signals*
    - **clock_rgb.vhd –** Accessory of RGB_gen..vhd
        - *implemented*
        - used to *convert 2-bit R,G,B signals to 1-bit R,G,B signals*

## ***Test bench Index***

### Top Level Entity Test Bench

1. **simu_top_level.vhd –** *used to simulate the top entity, and test tis performance after pack all components to a complete image indexing processor. It includes tow components: "simu_pc component", "top_proc component".*
2. **simu_pc_new.vhd –** *a component of simu_top_leve.vhd, used to simulate the PC states and generate the psuedo image data.*

### TCU Test Bench

3. **simu_proc.vhd** – *used to simulate the top entity, the image indexing processor, and test the performance of the Top Control Unit (TCU entity) before pack all components together. It includes "tcu component", "ppi2pc component" and 3 simulated components, which are "simu_pc component", "simu_index_eng component" and "disp_led component".*

4. **simu_pc_old.vhd** – *a component of simu_proc.vhd, similar to simu_pc_new.vhd.*

5. **simu_index_eng.vhd** – *a component of simu_proc.vhd, used to simulate the indexing engine states.*

6. **disp_led_old.vhd** – *a component of simu_proc.vhd, used to simulate LED display states.*

## *Design Verification*

### Comment

Actually, this part of report is only for Index engine. Because, in Mentor Graphics, the LPM_RAM_DQ module is not available, all test cases for the index engine are based on and simulated in MaxPlus II.

The most important point of this part is that we have to make sure we can retrieve and store, i.e., read/write data from/to the EAB rams without and errors. In order to verify our design, we made the following test efforts. First, we tested whether we can retrieve the data from the EAB rams initialized with *.mif files. By comparing the data read from the rams and the data in the *.mif file, the correct reading operation has been proved. Also, this way we know that the data now in the ram is initialized as in the *.mif file. Then we went further to check whether our memory initialization file works. Since we have to initiate the ram blocks to “0” each time when we try to retrieve a query image (one operation for the whole system), and each time a new candidate image come in for the candidate memory, the writing (storing) of “0” can be verified by checking the output port of the ram blocks when writing the data. The correct initialization has been proved when the output port changed to all “0”s instead of the data in the *.mif file. Finally, we tested reading/writing the data from/to EAB rams in one file, as what we build the color image histogram.

After this step, we began to integrate ram blocks together with other components in our index engine design. Because several sub-modules in the index engine all need to retrieve and write back data from the EAB rams, several 4-1 mux have been used for each set of memory *data_in, we, address, data_out* signals. And we also tested the top level engine to see whether these sub-modules can get access to the appropriate ram blocks at the right time and operate correctly.

The simulation waveforms demonstrated that everything worked fine.

### Index

1. **init_test_1.vhd** – *used to test whether we can read the data from the EAB ram blocks. It includes “EABs ram component”, two mif files for query image data and candidate image data, and a counter to generate the address from 0 to 47.*

2. **init_test.vhd** – *used to test whether we can clear the EABs when they have been initialized by *.mif files firstly. It includes “EABs ram component”, two mif files for query image data and candidate image data, “mem_init_1.vhd*

*component". After this file is executed, the all of the data in EABs should be 0 instead of the data in *.mif files.*

3. **init_cand_test.***vhd – used to test whether the cand_mem_init component can work properly. By initialized both query and candidate images with *.mif file, the query image histogram data in the query ram keep unchanged and the candidate image histogram has been cleared.*

4. **index_test_uninit.vhd –** *used to test whether we can retrieve/store histogram data from/to EABs with *.mif files in EABs firstly. It includes "EABs ram component", two mif files for query image data and candidate image data, and "color_hist1 component" to read the data out from the ram, add 1, and write the data back.*

5. **index_test.vhd** –u*sed to test whether we can retrieve/store the histograms from/to EABs after we clear the memory inistiated with *.mif files firstly. It includes "EABs ram component", two mif files for query image data and candidate image data, "mem_init_1.vhd component" and "color_hist1 component". This is to verify that the histogram can be build correctly.*

6. **distance_eab_test.vhd –** *used to test whether we can retrieve the histogram data from the EAB rams and calculate the distance between a candidate image and the query image without any errors. It includes "EABs ram component", two mif files for query image data and candidate image data and "distance_top.vhd component".*

7. **hist_dist_top_test.vhd –** *used to test whether the whole index engine can work properly. Incorporated with all the modules, i.e., "EABs ram component", two mif files for query image data and candidate image data, initialization module and cand_initializtion module, "distance_top.vhd component" and "color_hist1.vhd component", the index engine was tested to build the histograms for query and candidate images, calculated the distances for one candidate image.*

## References

1. B. Furht et al, Video and Image Processing in Multimedia System, Kluwer Academic

2. J.R. Smith, Integrated Spatial and Feature Image Systems: Retrieval, Analysis and compression, http://disney.ctr.columbia.edu/jrsthesis/thesissmall.html .

3. M.D. Marsicoi et al, Indexing pictorial documents by their content : a survey of current techniques, Image and Vision Computing 15 (1997) 119-141

4. J.R. Smith et al, Tools and Techniques for Color Image Retrieval, http://www.ctr.columbia.edu/~jrsmith/html/pubs/tatfcir/color.html

5. Tim Bensler, Eric Chan, Data Compression Co-processor Final Report, http://www.ee.ualberta.ca/~elliott/ee552/projects/1999_w/dataCompression/

6. Eric Cheung, Felicia Cheng, David Li, Tin Wai Kwan, SRAM Interfacing Basics, http://www.ee.ualberta.ca/~elliott/ee552/studentAppNotes/2000_w/interfacing/sram_basics/sram.html

7. Tim Bensler, Eric Chan, RS-232 Serial Port Application Note, http://www.ee.ualberta.ca/~elliott/ee552/studentAppNotes/1999_w/RS232/

8. John Koob, Parallel Port Interfacing, http://www.ee.ualberta.ca/~elliott/ee552/studentAppNotes/1999f/parallel_port/

9. Juan Gabriel Del Cid Portillo, Parallel Printer Port Access through Java, http://www.geocities.com/Juanga69/parport/

10. F. Idris et al, Image and video indexing using vector quantization, Machine Vision and Applications (1997) 43-50

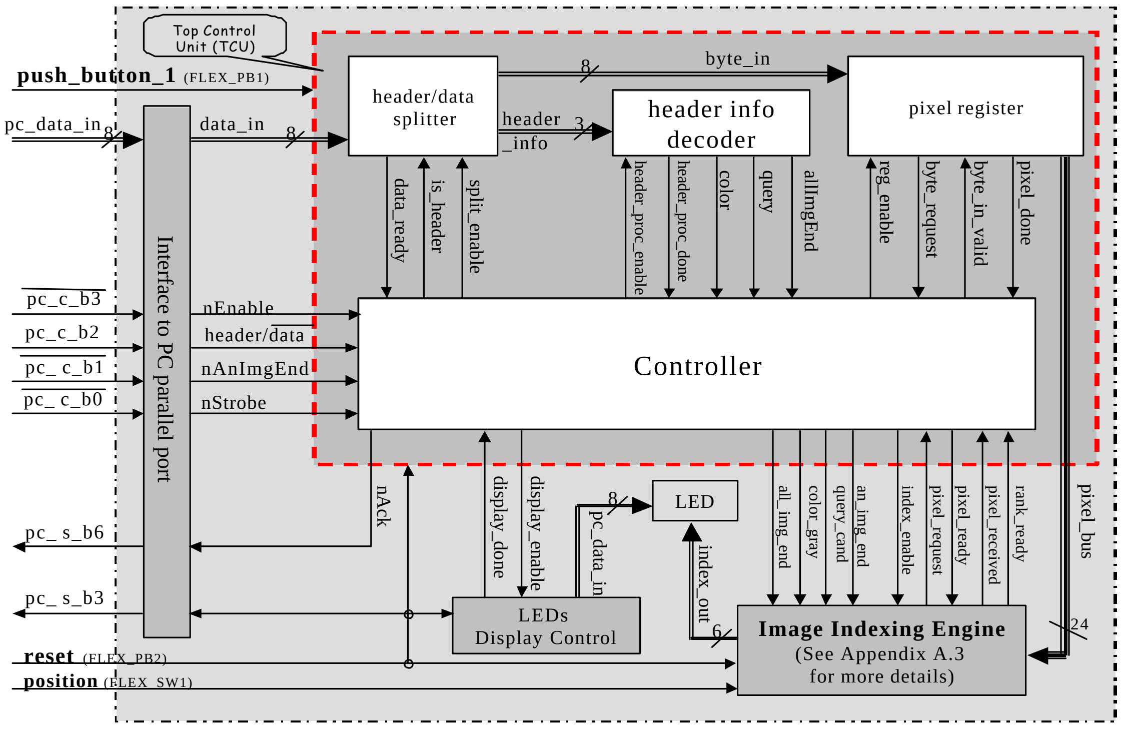
11. Hao Luan , Bo Liu and Albert Chan, EE552 APPLICATION NOTE, http://www.ee.ualberta.ca/~boliu/projects/ee552/ee552.html

12. Rabi Mahapatra, Xilinx Chapter 13 – Parallel Port I/O, http://www.cs.tamu.edu/course-info/cpsc483/spring98/rabi/xilinx/chap13/chap13

# Appendix A – Design

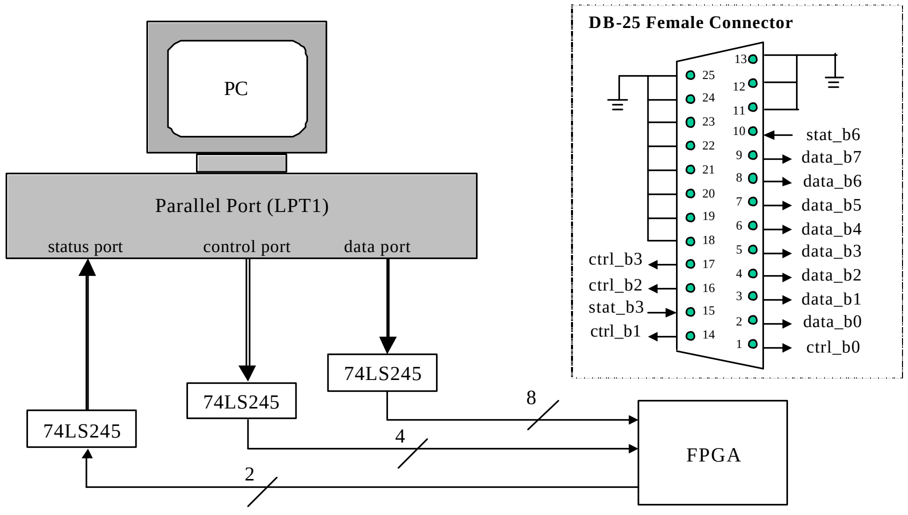
## A.1 High Level Image Indexing Processor Block Diagram

(The detailed TCU block diagram is included)



## A.2 Schematics

External circuit connecting and DB-25 femal connector



PC
Parallel Port (LPT1)
status port
control port
data port
74LS245
74LS245
74LS245
FPGA
8
4
2
DB-25 Female Connector
13
12
11
10
9
8
7
6
5
4
3
2
1
25
24
23
22
21
20
19
18
17
16
15
14
stat_b6
data_b7
data_b6
data_b5
data_b4
data_b3
data_b2
data_b1
data_b0
ctrl_b0
ctrl_b3
ctrl_b2
stat_b3
ctrl_b1