Snapshot

Sanjay Jhaveri
Mike Huhs

6.111 Final Project

The goal of this final project is to implement a digital camera using a Xilinx Virtex II FPGA that is built into the 6.111 Labkit. The FPGA will interface with a video camera, and output the digital signal to a VGA monitor. The user will then be able to capture the displayed image by pressing a button on the labkit. Once the image has been captured, it will be compressed using a discrete cosine transform (DCT) in order to be stored and viewed at a later time. In addition to image compression, the user will be able to perform zoom and rotate operations as well as perform some simple image filtering.
Implementing Snapshot will involve using the onboard ZBT memory of the 6.111 Labkit to store the video signal. This project should be a natural continuation of 6.111 Labs 3 and 4, as it builds upon the concepts of VGA display and memory. The lab will be broken up into: extracting the received video signal, displaying the signal to the monitor, and storing the video signal in the onboard Labkit memory.

Table of Contents

List of Figures

Overview

1.1 Introduction

Digital Cameras use aggressive compression techniques to be able to store large image files on relatively little memory. These compression techniques significantly reduce the amount of memory required to store and image while sacrificing relatively little image quality. The purpose of this project was to implement a digital camera capable of storing multiple images on a 4MB SRAM a through discrete cosine transform (DCT) image compression. This was to be done using a NTSC video camera, a Xilinx XC2v6000 Labkit, and an XVGA monitor displaying the user interface for the camera.

1.2 User Interface

The user interface for the camera is shown in Figure 1. Basically, the user sees a live video feed from the video camera, and uses a mouse and three buttons to operate the camera. The "Shoot!" button captures and image and stores it to memory, the "View" button allows user view the pictures in memory by replacing the video feed with a stored picture, and the "Live" button takes the user back to a live video feed to shoot another picture.

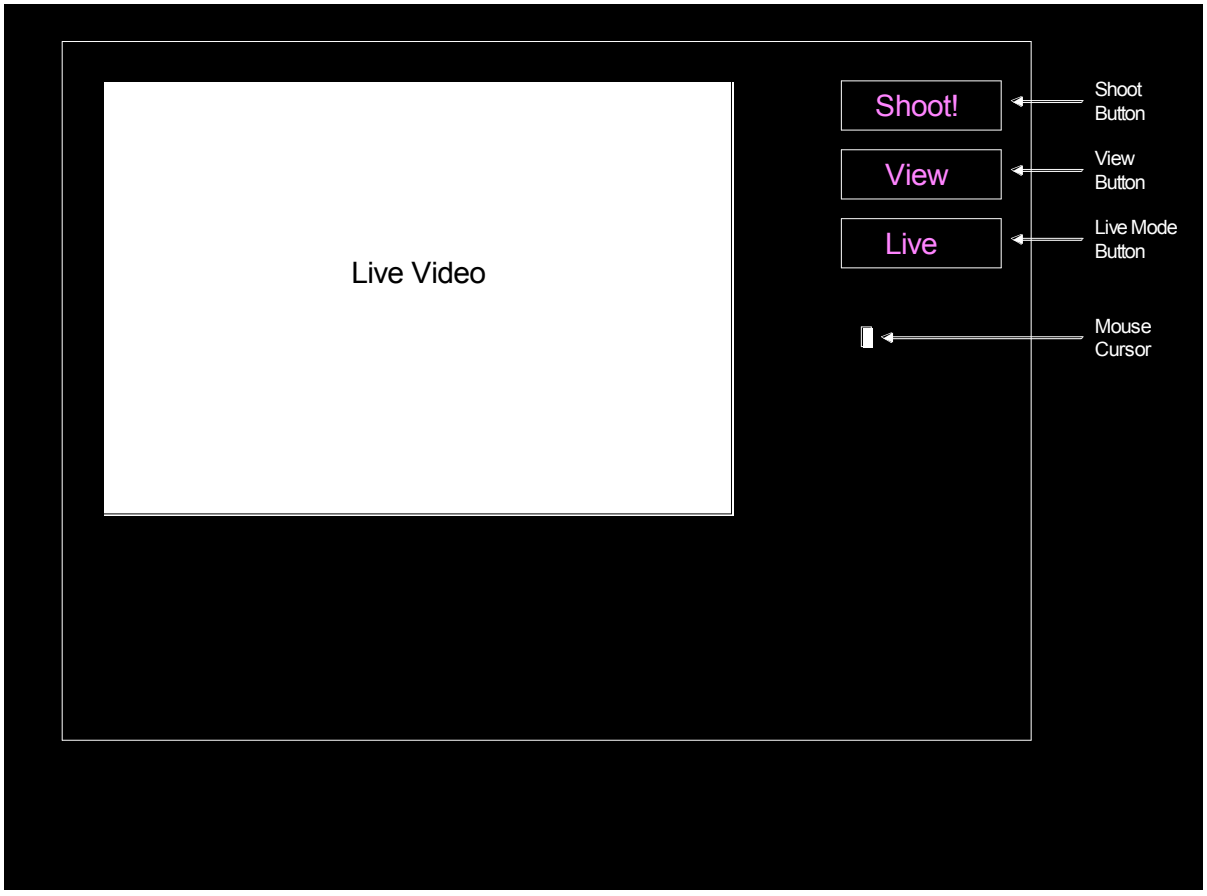


Figure 1: User Interface

1.3 Image Compression Background

Due to the limited memory resources available on the 6.111 lab kit, it was only possible to store a single uncompressed video frame to each of the two available zbt srams, thus in order to realize the full functionality of a digital camera it was necessary to compress the image prior to storage in memory. We attempted to implement a highly simplified jpeg compression scheme which relies upon the discrete cosine transform to decompose an image into its various frequency components. A 2D DCT can be applied to an image by separating the image into 8 x 8 pixel blocks and applying the transform to each of these blocks separately. The 2D DCT can be viewed as a series of matrix multiplications where the pixel matrix (M) is first multiplied by the DCT matrix (T) and then by its transpose (T') according to the equation D = TMT', where the DCT coefficients are calculated according to the figure below.

$$T_{i,j} = \begin{cases} \frac{1}{\sqrt{N}} & \text{if } i = 0 \\ \sqrt{\frac{2}{N}} \cos\left[\frac{(2j+1)i\pi}{2N}\right] & \text{if } i > 0 \end{cases} \longrightarrow T = \begin{bmatrix} .3536 & .3536 & .3536 & .3536 & .3536 & .3536 & .3536 & .3536 \\ .4904 & .4157 & .2778 & .0975 & -.0975 & -.2778 & -.4157 & -.4904 \\ .4619 & .1913 & -.1913 & -.4619 & -.4619 & -.1913 & .1913 & .4619 \\ .4157 & -.0975 & -.4904 & -.2778 & .2778 & .4904 & .0975 & -.4157 \\ .3536 & -.3536 & -.3536 & .3536 & .3536 & -.3536 & -.3536 & .3536 \\ .2778 & -.4904 & .0975 & .4157 & -.4157 & -.0975 & .4904 & -.2778 \\ .1913 & -.4619 & .4619 & -.1913 & -.1913 & .4619 & -.4619 & .1913 \\ .0975 & -.2778 & .4157 & -.4904 & .4904 & -.4157 & .2778 & -.0975 \end{bmatrix}$$

Figure 2: DCT Matrix Coefficients

The values of the resulting transformed matrix (D) represent the different frequency components of the input image. The DC value is located in the first row and column and while the higher frequency components are located in the lower right indices. With normal jpeg encoding this matrix is divided by a quantization matrix and rounded in order to force as many terms to zero as possible and the resulting matrix is then run length encoded along the diagonals. However, because the eye is less sensitive to high frequencies and because the high frequency coefficients are often much less than the lower frequency coefficients it is possible to achieve image compression by storing only the low frequency components to memory. The image can then be decompressed by replacing the discarded high frequency terms in the transformed matrix with zeros and applying the IDCT using the equation M = T'DT, which is simply the DCT in reverse.

In order to assess the effectiveness of this method we used matlab to compute the error resulting from applying this process to a sample 8 x 8 pixel matrix. When we applied the 2D-DCT to the sample matrix and stored only the first 5 rows and columns of the transformed matrix, the resulting decompressed matrix had an average pixel error of 7.25, which would be almost undetectable to the human eye considering a range of 256 possible values. However, it should be noted that the error can very greatly depending upon the frequency spectrum of a particular image block and images with lots of sharp features will be affected much more dramatically by the compression process than images with smoother color gradients. Additionally, the calculations performed by matlab can be expected to have a much higher level of precision than possible with an FPGA and so additional error due to rounding was expected in the final hardware implementation. Nonetheless, using this level of compression, and assuming an image size of 456 x 712 pixels it would be possible to store 4 images to a single zbt sram. A graphical depiction of our matlab results can be seen in the figure below.

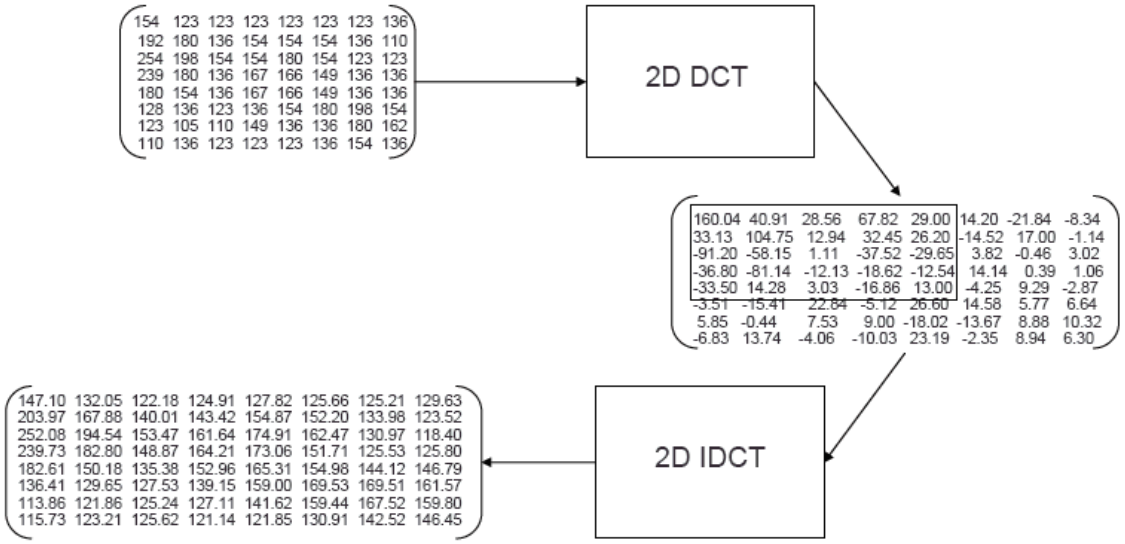


Figure 3: Matlab Results

1.4 Image Compression & Storage

Due to significant challenges encountered when trying to implement the DCT and IDCT, The image compression and storage block diagram illustrated on the following page represents the behavior of our intended system and not what was actually achieved. Upon a write request, data from a captured video frame was to be inputted into the image compression block one pixel at a time in 24 bit Y/Cr/Cb format starting at the upper left corner, reading in consecutive 8 x 8 pixel blocks. The 24 bit input was to be split into its Y/Cr/Cb components and compressed in parallel through three separate DCT's resulting in a 36 bit output which would then be stored to the a single location in the sram. Depending upon the level of compression selected by the user using the lab kit switches a variable number of transformed values would be stored. Conversely on a read request the image compression block was supposed to read the transformed data from the sram, split the 36 bit ram data into its 12 bit Y/Cr/Cb transformed components and decompress each component in parallel. Ultimately due to trouble synthesizing three separate DCT and IDCT modules the system was modified to only take in and store the Y component of the input signal, which would result in a gray scale image. However, once again this goal was not realized due to inconsistent outputs of the IDCT and DCT modules. Additionally, a memory control module was intended to keep a memory map of the starting address and compression level of each image stored in memory. This memory module was supposed to control where each image was written to and read from on a write or read request but as a result of the time spent debugging the IDCT and DCT portion of the system the memory control module was not completed.

2 High Level Block Diagram

Figure 2 shows a general block diagram of the entire system. For a more detailed view of the inputs and outputs of each module, please see the module descriptions below.

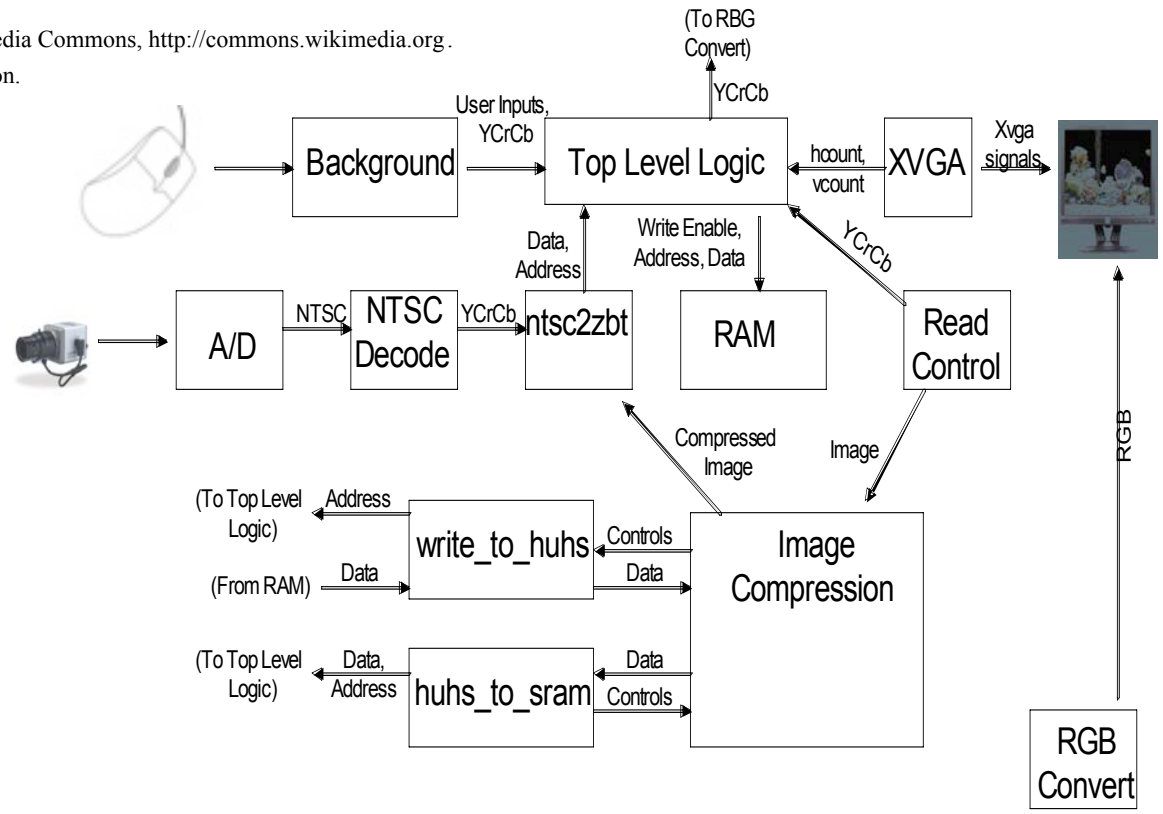


Figure 4: Digital Camera Block Diagram

3 Module Descriptions

3.1 zbt_6111_sample

Although zbt_6111_sample is a top level module, it contains code that determines the central inputs to the other modules. The DCM module is used in zbt_6111_sample to create a ~130 Mhz clock and a ~65 Mhz clock. Almost all of the modules use the 130 Mhz, the only exception being the char_string_display module.

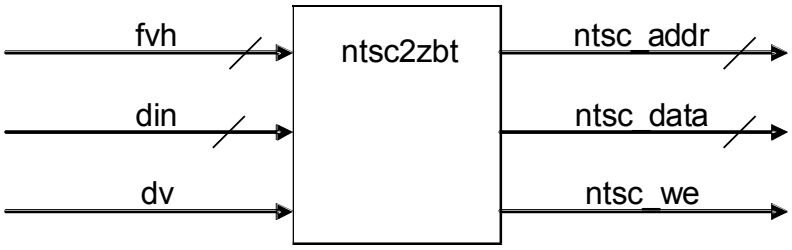
Each pixel requires at least 24 bits of color data. As the ZBT SRAMs on the labkit have 36 storage bits per address, every pixel will need its own address. If every pixel needs a separate address, the zbt will have to be able to read different addresses at least at the 65 Mhz pixel clock rate. Since the SRAM must be written to as well as read on certain clock cycles, the SRAM clocks can not be run at 65 Mhz, but must be sped up to 130 Mhz. This allows a read and a write to happen on the same 65 Mhz clock cyle. To determine whether the zbt rams should be accepting read inputs or write inputs, a parity bit that changes every positive edge of the 130 Mhz clock is created in the top level.

When the parity bit is high, the zbt_6111 module accepts an address and data from ntsc2zbt, and if parity is low, the zbt_6111 module uses the vcount and hcount as its address and reads the stored data out of the SRAM.

The state transitions from user inputs are also controlled in the top level. The camera works by starting off in live mode and writing the video data to both of the SRAMs, but reading from only one SRAM to the screen. When a user clicks "Shoot!," the SRAM currently reading the display out stops getting written to and the frame the user saw when he or she clicked "Shoot" is repeatedly read out of the SRAM. The user sees this still image until he or she clicks "Live." At this point the other SRAM that is being written to with incoming video data is read and the user sees a live video display on the screen. Keeping the last image taken in one of the SRAMs and the live video feed in the other SRAM implements a camera capable of only holding one picture.

Also instantiated in the top level is the char_string_display module. This was a module available online, and is used to display the labels on the user interface buttons. This module simply takes in an ASCI binary code for the characters to be displayed and displays the characters at a specified position on the screen.

3.2 ntsc2zbt



The ntsc2zbt module prepares the NTSC data received from the NTSC decoder and loads into the ZBT SRAM so that it can be displayed. The timing issue in this module are not trivial as video data is coming in from the NTSC at a different rate that the video data is being written to memory, and also at a different rate as the SRAM clock. To remedy this, the ntsc2zbt module uses a series of pipelines to determine when and where data should be written to the zbt SRAM.

The NTSC decoder module changes input data at 27 Mhz. The inputs it feeds to the ntsc2zbt module are a 30 bit YCrCb value, a data valid bit, and three frame information bits (F, V, and H). The ntsc2zbt module has the job of using the frame bits to link the incoming pixel data with a specific row and column on the video display. The H frame bit simply goes low high when the video data is starting on a new line. The vertical timing protocol is a bit more involved and is shown below in Table 1:

Line #	F	V	H(EAV)	H(SAV)	Notes
1-3	1	1	1	0	Blanking, Lines 1-9, 9 Lines
4-19	0	1	1	0	Blanking, Lines 10-19, 10 Lines (optional ancillary data except line 14)
20-263	0	0	1	0	Field 1 (Odd) Active Video, 244 Lines
264-265	0	1	1	0	Blanking, Lines 264-272, 9 Lines
266-282	1	1	1	0	Blanking, Lines 273-282, 10 Lines (optional ancillary data except line 277)
283-525	1	0	1	0	Field 2 (Even) Active Video, 243 Lines

Figure 5: NTSC Vertical Timing Reference

As Figure 3 shows, video data comes in when V is low. Another important aspect of the NTSC format is that the odd lines are all transmitted first, followed by all of the even lines at once.

The ntsc2zbt looks at the inputs at a 27 Mhz clock rate (video clock). If video data is not being transmitted, row stays at a starting value. When V goes low and video data is being transmitted, the column increments every clock cycle, resetting to zero every time H signals the start of a new line. The row stays at the starting value until V goes low and signals that video data is incoming. When V is low and H signals the end of the line, the row increments. Note that row resets to zero for both the start of the even and odd lines. In addition, the variable even_odd looks at F to determine whether the video data coming in is for the odd or even fields. The variable vwe is basically a one bit input that goes high whenever data valid makes a transition from 0 to 1 on a given clock cycle. The data is latched into vdata, which only registers the incoming data when dv signals that the data is valid.

The timing pipelines works by latching the row, column, data, even_odd, and dv fields through two registers at a clock rate of ~130 Mhz as shown below in Figure X. These registered fields must be used to determine when and where to write data into the zbt SRAM. Whenever we[1] is high and old_we is low, this means that data[1] consists of new unwritten data. When this is the case, mtsc2zbt outputs this data and a calculated address where this data should be stored. To calculate the 19 bit address in memory, the structure shown below is used:

This format allows the data access on a read to be relatively simply. To access the color data for the pixel corresponding to a certain vcount and hcount we can simply go to the address {vcount[8:0], hcount[9:0]}. By shifting the row bits and adding even_odd as the LSB of the row, the even and odd rows are interlaced as now row 1 of the even rows will have an adjacent memory address to row 1 of the odd rows.

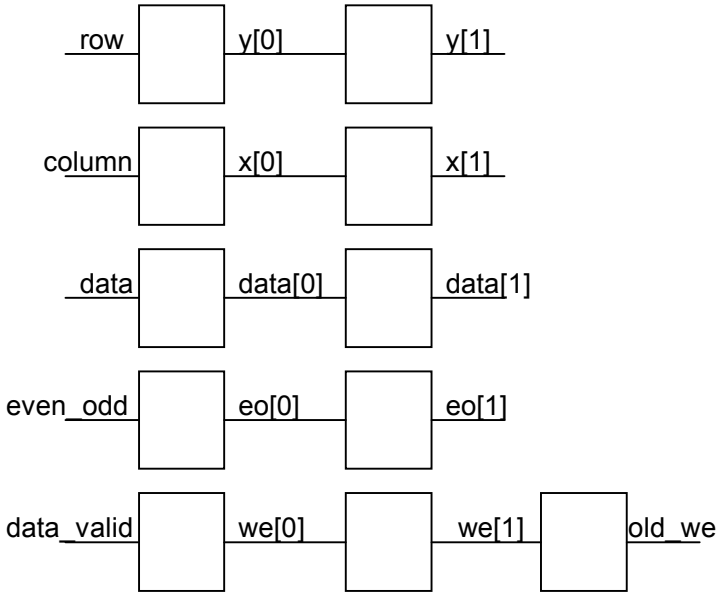
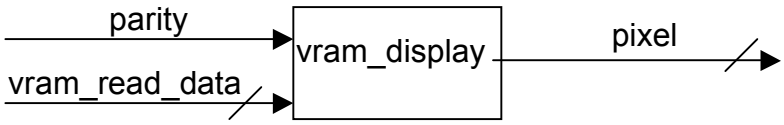


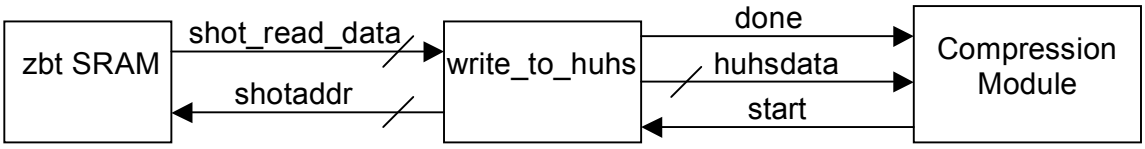
Figure 6: Timing Pipeline

3.3 vram_display



The vram_display module is responsible for reading the YCrCb pixel data from the zbt SRAM, latching the relevant 30 bits, and outputting the YCrCb value that it receives to the RGB conversion module. In addition, vram_display takes in the hcount and vcount from the vga module, and only outputs data it gets from the memory if the hcount is within the display window on the screen. If the hcount and vcount are not within the display window, vram_display outputs the 30 bit YCrCb value for black. As a read should only happen parity is low, vram_display only reads out new data when the inputted parity is low.

3.4 write_to_huhs

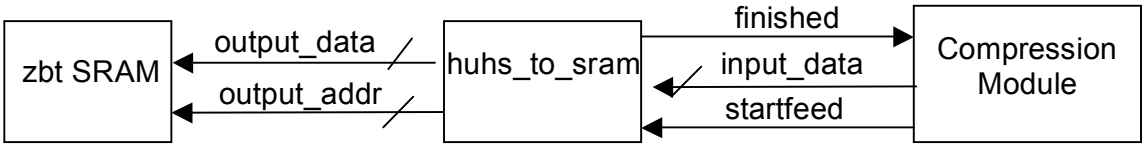


The write_to_huhs module waits for a shoot signal that indicates a picture has been taken, and then proceeds to feed the data from the zbt SRAM data into the compression module in a sequence of 8 X 8 blocks. The compression module computes the DCT of the data in these square blocks, so this module facilitates the data reaching the DCT module in a useful format. The waveform of this module is included in the Appendix. As the waveform shows, when the write_to_huhs receives a shoot signal, at a clock rate of

27Mhz, the module goes through the pixels one block at a time, outputting the address containing the pixel's data to the zbt SRAM, and reading the data output from the SRAM. The module then passes this SRAM data to the compression module.

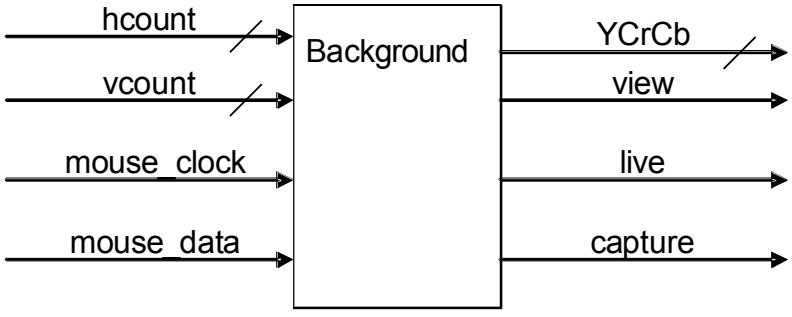
The method that write_to_huhs uses to scan through the 8 X 8 blocks in order is very similar to the technique used by the xvga module. Basically, binary variables indicate whether the pixel being fed to the compression module is at the right edge of a block, at the bottom of a block, at the very bottom edge of the bottom block, or at the right edge of the right most block. Using these binary variables, the module can scan through the pixels in the proper order.

3.5 huhs_to_sram



The huhs_to_sram module is used when the user of the digital camera would like to view a picture in memory. When this is the case, the compression module starts to feed the data out in the order that it was feed into the compression module by write_to_huhs. As this data is being fed to huhs_to_sram, the data needs to be written back into the memory in the {vcount, hcount} addressing that allows the data to be read (see ntsc2zbt module for addressing protocol) to the xvga and viewed on the monitor. As huhs_to_sram essential performs the reverse process as the write_to_huhs module, the code for these modules is very similar. A waveform showing the functionality of huhs_to_sram is show in the Appendix. As the waveform shows, when a start signal is received, the module receives the pixel data from the compression module and writes the data into the appropriate address in memory show that that the compressed image can be displayed to the screen.

3.6 Background



Background takes care of most of the user interface items appearing on the monitor. This includes the white frames surrounding the video display and buttons, and the mouse cursor. To create a border, Background simply instantiates four instances of the module

rect, which draws four rectangles at specified locations to construct a border. To produce a mouse cursor, Background instantiates the ps2_mouse_xy and blob module. The ps2_mouse_xy takes data from the PS2 mouse input on the Labkit and updates an x and y value according to movements of the mouse. These x and y values are fed into the blob module, which draw a rectangular cursor on the screen at the updated position. Background takes in hcount and vcount from the xvga module and returns a 30 bit YCrCb value. This outputted value is created by taking the "or" function of the YCrCb values from all of the instantiated rect modules and the mouse blob module.

3.7 ZBT SRAM

The ZBT SRAM module came from the 6.111 fall 2005 website. It takes in we, cen, address, and write data signals and outputs we, cen, address signals to the zbt sram. If the we input is high then the ram data inout port is assigned to the write data input and if the we input is low then the ram data inout port is assigned to the read data output.

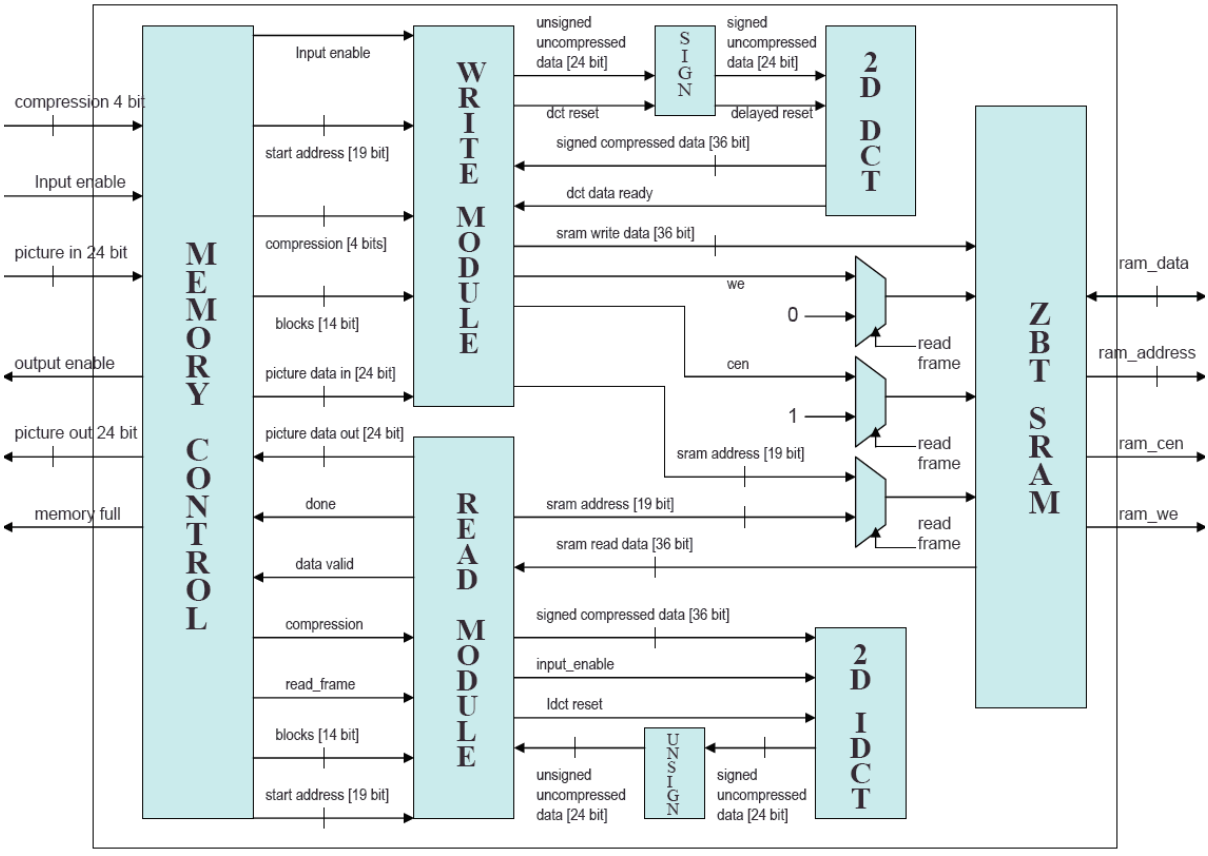


Figure 7: Image Compression & Storage Block Diagram

3.8 Write Module

When the input enable signal to the write module is high, it passes a 24 bit Y/Cr/Cb input to the DCT module. Following an initial latency period the DCT module sends a data ready signal to the write module which signals it to start writing the DCT data output to the sram. As it is writing to the sram, the write module continues to pass the image input

to the DCT module as long as the input enable signal is high. Additionally the write module keeps a count of how many image blocks it has processed and which row and column of a block it is currently writing. It uses the row and column counts along with the 4 bit compression input to determine which data to write to the sram and which data to discard. Once the module has written a specified number of image blocks corresponding to a particular image size it sends a reset signal to the DCT module and stops writing to the sram. A test bench illustrating the behavior of the write module can be found in the test bench section.

3.9 Sign/Unsign Modules

The DCT algorithm operates on input values ranging from 128 to -128, however the Y component of the input signal ranges from 0 to 256. The sign module simply subtracts 128 from the Y signal, converting it from an 8 bit unsigned value to an 8 bit signed value. Because this computation results in a clock cycle delay, the reset signal to the DCT and the Cr/Cb signals must also be delayed in order to maintain proper timing and thus the sign module simply registers these inputs.

3.10 Read Module

When a read request is asserted, the read frame signal is registered and goes high signaling the read module to begin reading from an address specified by the start address input. The compression level input specifies what compression the image it is reading was saved and the read module uses this information along with a running column and row count of where in a particular image block it is reading from sram to determine when to append zeros to the sram output which is then passed to the IDCT module. The two clock cycle latency of the zbt ram greatly complicates the behavior of this module as the address of a location in memory must be presented two clock cycles before the data from that address is needed. Additionally, the read module keeps a count of the number of inputs to the IDCT as it must begin passing the IDCT data output after a 93 clock cycle latency and stop passing the IDCT values once it has processed a specified number of image blocks.

3.11 2D-DCT/IDCT Modules

The DCT and IDCT modules were taken from the applications section of the Xilinx website yet proved to be very troublesome to interface with. The DCT module takes in an 8 bit input and produces a 12 bit output. It starts accepting data when the input reset signal goes low and expects data from a particular image block to be presented row by row from left to right. After a 76 clock cycle latency it outputs a high data ready signal and begins outputting a new transformed value each clock cycle thereafter. The IDCT module works in a similar manner except that it has a 93 clock cycle latency and does not output a data ready signal and instead takes in an additional input data ready signal. The IDCT module takes in a 12 bit input and produces an 8 bit output.

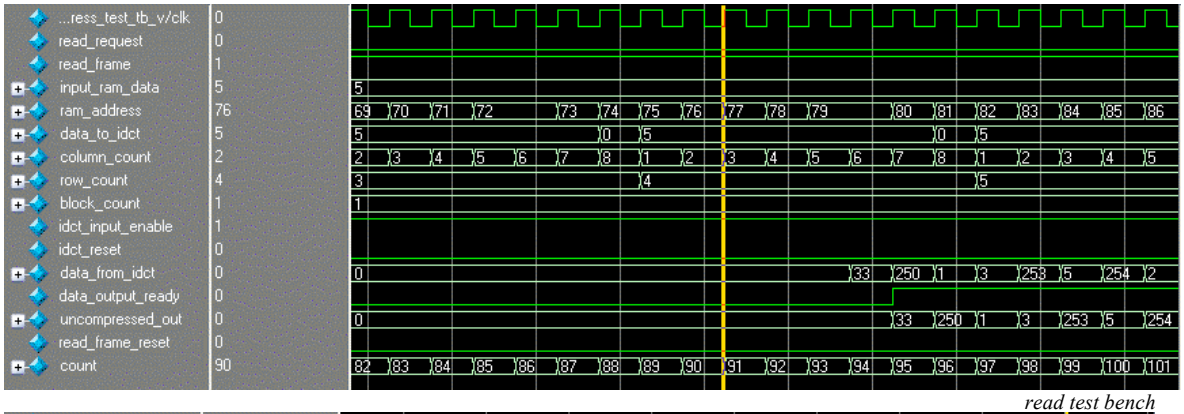
3.12 SRAM Muxes

Both the read and write module interface with the same zbt sram and therefore muxes are used to determine which module has control of the memory. The read frame signal is high for the entire duration that the read module must read from the sram and therefore it is used to select which we, cen, and address signals the sram will receive. If read frame is high, the sram receives the sram address from the read module, we is grounded and cen is set high. Otherwise, if read frame is low, the we, cen, and address inputs to the sram come from the write module.

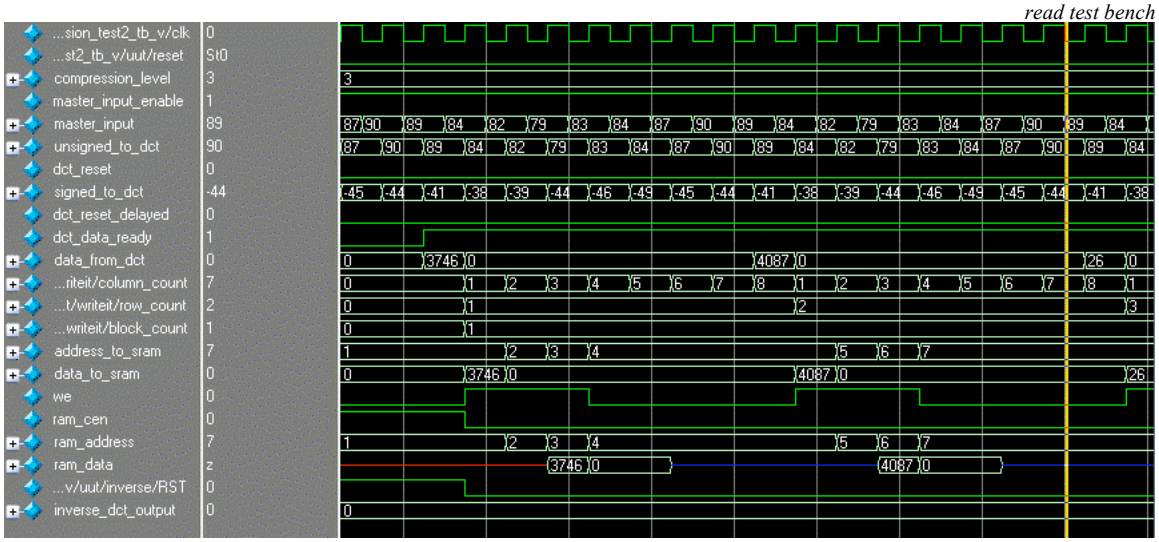
4 Testing & Debugging:

The majority of time associated with testing and debugging was spent trying to figure out the DCT/IDCT modules downloaded from the Xilinx website and if this project were to be done again I would almost certainly attempt to write my own DCT/IDCT modules. While there where many issues associated with the implementation of these modules including trouble synthesizing them, by far the biggest challenge was figuring out how these modules responded to signed inputs. In order to facilitate testing, an IDCT module was connected to the sram data out of the write module. During testing, narrow ranges of input values were used to ensure that most of the transformed coefficients would be near zero and thus the output of the IDCT would be minimally affected by the compression level. The module was first tested by passing in a string of values ranging between 239 and 245 and the output of the IDCT was almost identical to the signed input. Next values between 79 and 90 were passed in and once again the output of the IDCT was a close match to the input string (as evidenced by the second to last test bench). However, when values between 130 and 126 were passed in so that the signed input to the DCT varied between positive and negative, the output of the IDCT was not even close to the input even though the majority of the transformed DCT values were zero and thus the effects of compression were not to blame for this discrepancy.(see last test bench). Please find the DCT and IDCT waveforms on the following page

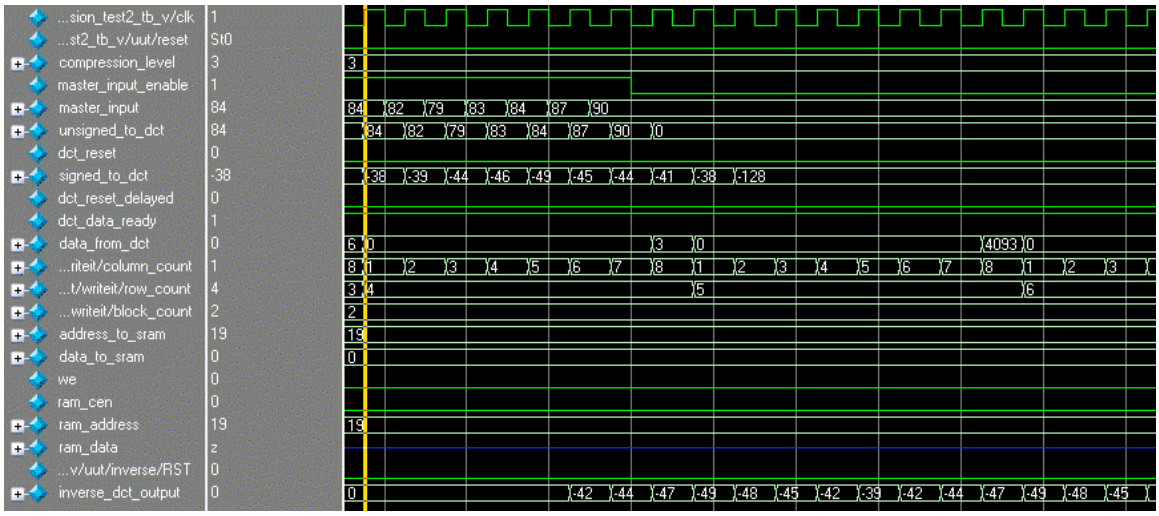
4.1 Test Benches for Read and Write Modules:



read test bench



write test bench



write test bench (idct_out = dct in)

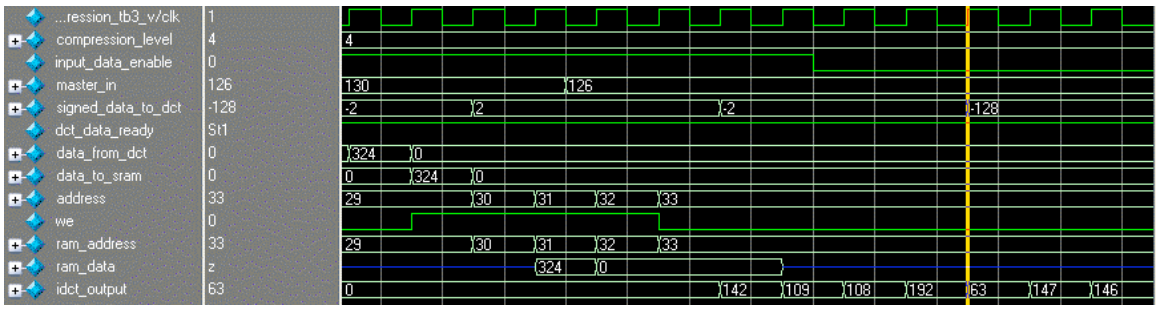


Figure 8: Test Benches

5 Conclusion

Initially, the goal of our 6.111 final project was to implement an FPGA based digital camera capable of storing and viewing multiple images. In order to reach this goal with the limited memory resources available successful data compression and decompression was essential. However, due to problems interfacing with the DCT and IDCT modules provided by Xilinx, we were not able to achieve data compression. Faced with these restrictions, the functionality of our digital camera was severely limited. In spite of these limitations, our final project was able to store an image on one of the two lab kit sram's while writing to the other and the user was able to switch between live camera and picture modes. Through the adversity we faced in the completion of our project we learned the importance of extensive test benching and we became aware of the pitfalls of using another person's code without thoroughly understanding it.