Digital Design and FPGA Implementation of a Wireless Video Surveillance System

Vivek Shah
Noel Campbell
Raymond Tong
Introductory Digital Systems Laboratory
TA: Javier Castro
5/19/2006

Abstract:
This laboratory used a Xilinx FPGA to create a video surveillance system with wireless transmission as a digital circuit. The surveillance system was fully functional and able to capture, encode, and transmit and image, as well as receive, decode, and display the image. ModelSim simulations were used to test the various modules of the surveillance system, as well as Verilog test benches. After a comprehensive suite of tests found no errors in the design, the surveillance system was programmed into a FPGA and passed physical testing as well.

Table of Contents

1. Title Page and Abstract 1
2. Table of Contents 2
3. List of Figures, Tables, and Equations 3
4. Operational Overview 4
5. Module Description and Implemention 5
    a. Capture-Encode-Transmit System 5
        i. Video Capture 5
            1. NTSC Decoder 5
            2. Store 64 6
            3. Set Address 6
            4. VGA Controller 7
            5. Delay 7
            6. YCrCb to RGB converter 7
            7. Display 7
        ii. Video Memory (Encoder) 8
        iii. Encoder 8
            1. DCT Multiply 10
                a. DCT Front 10
                b. DCT Table 10
                c. Multiplier Shift Register 10
                d. DCT Back 10
            2. Encode_memory_register 11
            3. Encode_FSM 11
        iv. Wireless Block Memory 12
        v. Wireless Transmitter 12
            1. Transmitter Control Unit 13
                a. TX Shift Register 13
                b. RS232 Sender 13
        vi. Wireless Packet Sender 14
    b. Receiver-Decode-Display System 14
        i. Wireless Receiver 14
            1. Packet Receiver 14
            2. Receiver Control Unit 14
                a. RS232 Receiver 14
                b. RX Shift Register 14
        ii. Wireless Block Memory (Decoder) 15
        iii. Decoder 15
            1. DCT Multiply Decode 16
                a. DCT Front Decode 16
                b. DCT Table 16
                c. Multiplier Shift Register 16
                d. DCT Back Decode 16
            2. Decoder FSM 17
        iv. Video Memory (Decode) 17
        v. Video Display 17
            1. Read 64 18
6. Testing and Debugging 19
7. Conclusion 22
8. Appendix 23

List of Figures

Figure 1 – System Diagram 4
Figure 2 – Block diagram of Video Capture Module. 5
Figure 3 – Set Address Finite State Machine....................6
Figure 4 – Writing to Video Memory Process....................8
Figure 5 – DCT coefficients used in the encoded block. 9
Figure 6 – Block diagram for encoder module. 9
Figure 7 – State Transition Diagram for Encoder FSM....................12
Figure 8: Transmitter Block Diagram....................13
Figure 9 - Wireless Receiver Block Diagram 14
Figure 10 – Block diagram of Decoder module....................16
Figure 11 – State Transition Diagram for Decode FSM. 17
Figure 12 – Block diagram of the Video Display Module. 18
Figure 13 – Video Capture/Display Testbench....................19
Figure 14 – Video Encoder Testbench ModelSim waveform outputs. 20
Figure 15 – Decoder Testbench ModelSim waveform output. 21
Figure 16: Logic analyzer screenshot verifying overall system data flow....................22

List of Tables

Table 1 – Blanking and Synching Signal Values....................7
Table 2 – DCT Coefficient matrix as stored in memory using fixed point notation. 10
Table 3 –Outputs for ENCODE_BLOCK state of encode FSM. 12

List of Equations

Equation 1 – Matrix novation of DCT using 8x8 matrices. 9
Equation 2 – Two dimensional Discrete Consine Transform algorithm. 9
Equation 3 – Matrix notation of IDCT using 8x8 matrices....................15
Equation 4 – Two dimensional Inverse Discrete Cosine Transform algorithm. 15

Operational Overview

The problem with conventional security systems today is that they are mostly wired, meaning the security station must be placed within a fixed distance from the camera. This usually means that the security camera must be placed in a fixed location because it is difficult to move a wire embedded in the wall or ceiling of a building. In order to provide more flexibility, this project implemented a wireless security system where image data is sent wirelessly to a receiver station and displayed on the screen. However, in a wireless system where the transfer rate is more restrictive than a wired system, the information must be condensed in some form. This particular system uses a proprietary DCT based encoding method similar to JPEG encoding to decrease the size of the image data.

There are six main components of this system. Image data must be captured, encoded, transmitted wirelessly, received wirelessly, decoded, and displayed on the monitor. A system diagram if provided in Figure 1.

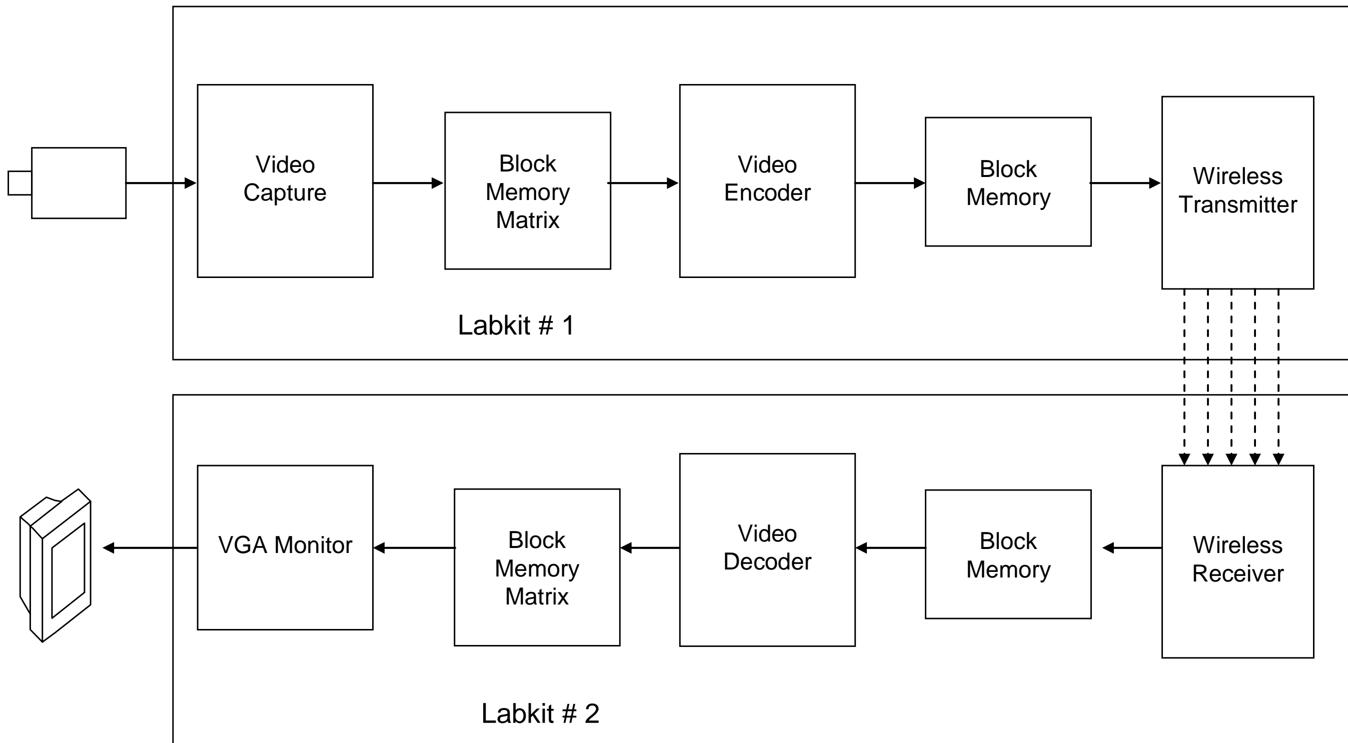

Figure 1 – System Diagram

The system was designed with modularity in mind. In order to achieve this, every subsystem is separated by a memory interface.

In normal operation, analog camera data is converted to a digital bit stream by ADV7185 chip on the labkit. Then this data is organized into a 240X240 pixel image, stripped of any chrominance (color) values, and written into memory so that it can be encoded. The encoder module processes the stored image by applying a Discrete Cosine Transform to 8x8 pixel blocks. The resulting coefficients are sent through the RS323 transmitter to the wireless transmitter. The encoded data is then divided into 10 byte packets and transmitted across the wireless channel. Once the data has been received, the wireless receiver sends the data to the receiver labkit via the RS232 interface and commits the data to memory to be decoded. Using the Inverse DCT algorithm, the encoded data is decoded into grayscale pixels. These pixels are displayed on the screen in an image by the video display unit.

A 240X240 grayscale image contains nearly half a mega bit of information. The challenge of this project was to decrease it to the smallest size possible in order to maximize the update rate (in frames/second) and still maintain a recognizable picture. Here, it was possible to take a 512 bit block of information (an 8X8 pixel block) and condense it down to 78 bits (70,200 bits for a full image). By using a variant of JPEG encoding, it is possible to retain much of the visible information in an image, while still compressing the information to just a few bytes. Our compression algorithm discarded 85% of the encoded coefficients, and was still able to transmit a coherent picture. Based on this compression and our wireless channel, we were able to achieve approximately one frame of video per second.

Module Description and Implementation

Capture-Encode-Transmit System

Video Capture (RAYMOND)

The Video Capture module has many sub-modules within it (NTSC Decoder, Store 64, Set Address, VGA Controller, Display, and Delay) whose functions will be described later. The main purpose of the Video Capture module is to connect all the sub-modules together and maintain an internal pixel counter and line counter for the data coming from the camera, which is in NTSC format. The reason why an internal pixel counter and line counter are needed is because one of the sub-modules, NTSC Decoder, only provides the start and end of lines and not the exact line number or pixel number. To compensate for this lack of information, Video Capture uses the State output from the NTSC Decoder to initialize its internal counter and line counter. Then, every time the data is valid (another output from the NTSC Decoder), it increments the internal pixel counter. To increment the internal line counter, it uses the State output from the NTSC decoder. For example, if the State output indicates the beginning of a line, and the input from the camera indicates it is in the field just before the active odd lines, then the internal line counter will reset to 0. So, when the State output indicates the beginning of a line and the camera input indicates it is now in the active odd lines, the internal line counter will increment.

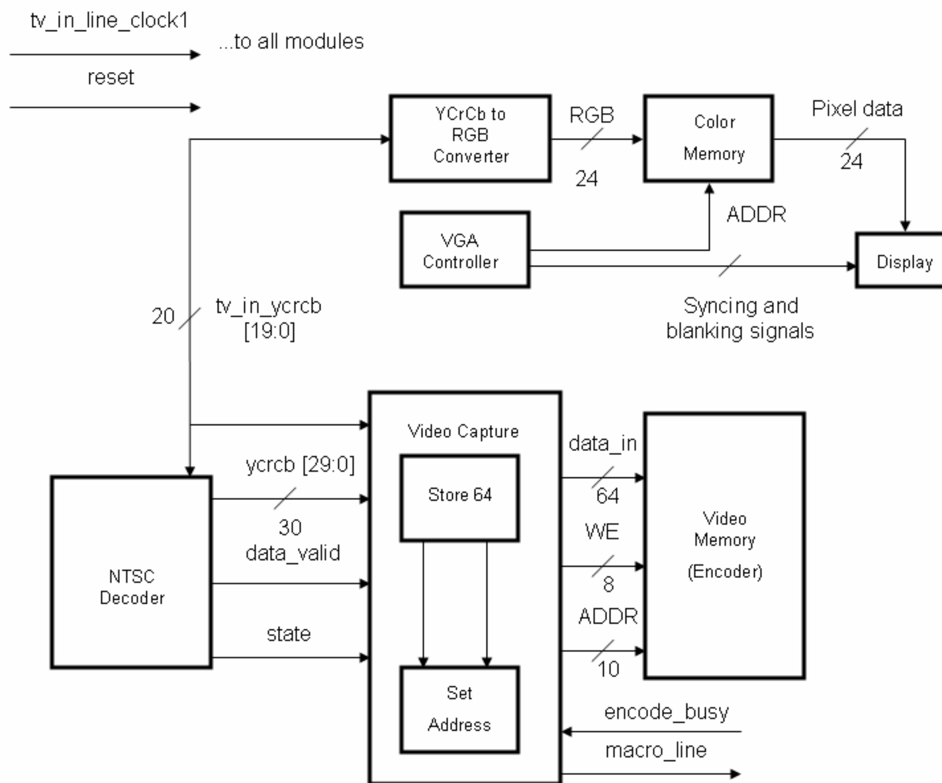

Figure 2 – Block diagram of Video Capture Module.

To reset back to pixel_count_internal = 0, Video Capture waits until the camera is at the beginning of a line. To reset back to line_count_internal = 0, Video Capture waits until the camera is at the beginning of a line and shows it is in the blanking field just before the odd active field.

Another important purpose of the Video Capture module is that it writes color data into the color memory block (24X57600). When the data is valid, 0 <= pixel_count_internal <= 239, and 0 <= line_count_internal <= 239, then it activates the write enable signal for the color memory block for one clock cycle and increments the address. To account for the delay in color conversion, the address and write enable signals are delayed for three clock cycles.

NTSC Decoder

The NTSC Decoder module takes in the 20 bit input, tv_in_ycrcb, from the ADV7185, which converts the analog video signal to a digital signal, and decodes position indicators to figure out what part of the picture is being transmitted. The NTSC Decoder does not provide an exact location in terms of which pixel or line it is on but rather the start and end of lines.

At the start or end of a line, the ADV7185 sends the same sequence of bits of information followed by a position indicator in the following form:

FF 00 00 XY
*Note: example shown in hexadecimal and only shows top 8 bits of information

where FF 00 00 indicates the start or end of a line and XY is the position indicator. The significant outputs from the NTSC decoder are State, which indicates which field it is about to enter (odd/even, active/blanking, etc), data_valid, which indicates when the YCrCb output is valid, and the YCrCb value itself.

The useful part of the NTSC decoder for this project is that it indicates when it reaches the beginning of the odd or even active lines coming from the camera and indicates when the data is valid, meaning the camera has output a chrominance and luminance value for one pixel (2 clock cycles).

Store 64

The Store 64 module is used to write grayscale data into memory used by the encoder module. It uses line_count_internal, pixel_count_internal, Y, and data_valid to gather 64 bits of data to be written to the encoder memory. It has two internal counters, data_counter and data_counter2, the first of which keeps track of when the 64 output shift register is full, the second of which keeps track of when a line (240 pixels) has passed. It also has a two bit counter called every_other. Because NTSC provides the odd lines and then the even lines, the picture of just one of these fields is twice as wide as it is tall. To compensate for this, every_other will toggle on data_valid so that the shift register will only accept every other pixel to produce a proportional image. The shift register starts and resets its internal counter every time the internal pixel counter is 0 and the internal line counter is within the active range (12 to 251). When it receives a data_valid, it checks the value of every_other. If every_other is 0, it shifts data in and changes every other to 1. If every_other is 1, it changes the value to every_other to 0 and ignores the current Y value.

Set Address

This module is responsible for setting the address the data from the Store 64 shift register is written to. Before describing how this module works, it is important to note that the data and address lines output from this module are tied to the input data line for all eight memory blocks in the encoder memory structure. This is because the data and address are always coming from the same source; the difference comes from writing the data sequentially to the eight blocks such that the encoder can utilize all the data it needs to (512 bits). A state transition diagram is provided in Figure 3. Essentially, data is shifted in until the shift register is full, at which point the data is written into the corresponding memory block for proper encoding.

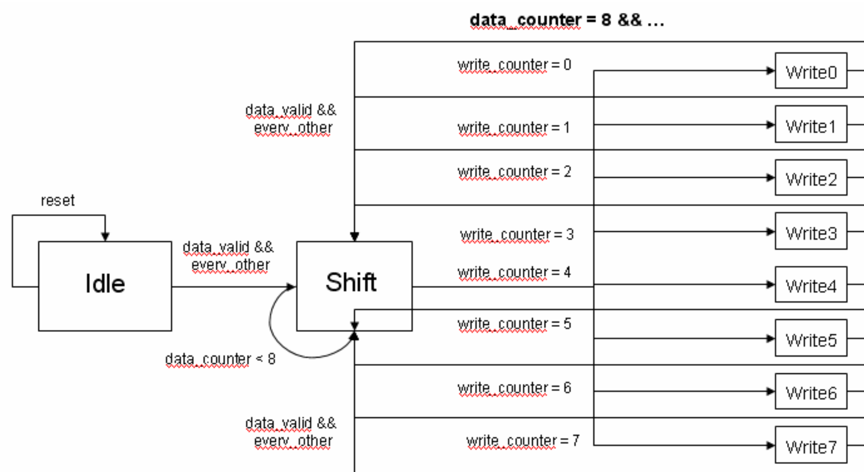

Figure 3 – Set Address Finite State Machine

Mathematically speaking, there are 240 pixels in a line or thirty 64-bit packets. Each 64 bit packet is written sequentially into one memory block until the beginning of a new line, at which point, the new-line 64 bit packets are written to the next memory block. Once eight lines have been written, the base address incremented by 30 and the process starts again with data being written to the first memory block. The Figure provided to help visualize this process.

VGA Controller

The VGA controller coordinates the signals that display images on the monitor. Every clock cycle, pixel_count increments by one and resets to zero once it hits 799. Line_count increments every time pixel_count hits 0 except when line_count is 524 and pixel_count is 799. Then both counters begin again from 0. There are two blank signals, one for the horizontal and one for the vertical directions, as well as two sync signals. The blank signal is pulled low whenever the screen should be blank. In this case, once a line has been drawn across the screen (pixel_count = 640) then hblank (horizontal blank) should be pulled low otherwise it would draw a line across the screen before it starts on the next line. The same goes for vblank only it goes low when line_count = 480. Vga_out_blank_b is the And of hblank and vblank.

The sync signals are similar only they are pulled low for a shorter period than hblank or vblank and are pulled low after hblank or vblank has been disabled. Sync signals have what are known as the front porch, sync pulse, and back porch. For the horizontal sync signal (hsync) the front porch is 16 clock cycles at the end of which the signal is pulled low, the sync pulse is 96 clock cycles (the period of time hsync remains low), and the back porch is 48 clock cycles, at the beginning of which hsync is pulled high. For vsync, the difference is the length of these intervals, 11,2, and 31 respectively, and applies as line_count increments, not pixel_count. Hsync runs off of the pixel counter and vsync runs off of the line counter. How the outputs change according to the counter values in Table 1. Using a 27 Mhz clock, this provides a refresh rate of 60 Hz.

Table 1 – Blanking and Synching Signal Values

Signal	Condition	Value
Hblank	Pixel_count <= 640	1, 0 otherwise
Hsync	656 < pixel_count <= 752	0, 1 otherwise
Vblank	Line_count <= 480	1, 0 otherwise
Vsync	491 < line_count <= 493	0, 1 otherwise

Delay

The purpose of the delay is to ensure the sync signals (hsync and vsync) are output the same time as the RGB signals are generated by the ADV7125 IC (it is necessary because the IC is pipelined). The delay will delay the sync signals by 2 clock cycles.

YCrCb to RGB Converter

This purpose of this module is to convert the YCrCb values for a given pixel into a 24-bit RGB value so it can be displayed on the screen. The conversion from Y, Cr, and Cb values into R, G, B values is given below:

$$R' = 1.164\,(Y - 16) + 1.596\,(Cr - 128)$$
$$G' = 1.164\,(Y - 16) + 0.813\,(Cr - 128) - 0.392\,(Cb - 128)$$
$$B' = 1.164\,(Y - 16) + 1.596\,(Cr - 128)$$

*Note: R', G', B' are gamma-corrected RGB values

This module is a three stage pipeline so if Y, Cr, Cb are input at t = 0, then the output will not be valid until t = 3. This is important when the display sets the address for the color memory.

Display

The Display module is responsible for drawing the screen by telling the monitor what color to output for every pixel. It does this by keeping track of where it is horizontally with pixel_count and where it is vertically with line_count. From these two counters it is able to draw a 640 X 480 resolution screen.

This module displays the screen by reading from memory, either the color memory on the encoder side or grayscale memory on the decoder side depending on which labkit it us being run on. The Display module controls which memory location is being read from the color memory block. Because the color memory block has enough memory locations for one 240 X 240 pixel frame, this module will increment every time pixel_count increments within the picture frame. For example, if the picture starts on the tenth line and the eighth pixel, then Display will reset every every time line_count = 8 and pixel_count = 797 and increments every time pixel_count increments when pixel_count is between 0 and 239. The reason why it doesn’t reset at line_count = 9 and pixel_count = 0, is because there is a delay of three clock cycles between the converted RGB values and valid YCrCb values. Thus when it is written into memory, it is lagging by three clock cycles.

The Read 64 module is responsible for reading what the decoder writes into video memory on the receiving end and outputs the correct grayscale (RGB = {Y, Y, Y}) value for a particular line_count and pixel_count combination. Therefore, the display module only outputs the RGB value from the Read 64 module when it is supposed to.

If pixel_count and line_count are neither in the color picture part of the screen or the grayscale part of the screen, then the display can output any background color (this project outputted a vga_out = 24'habcdef).

Video Memory (Encoder)

The video memory module is written by the video capture module and read by the encoder module. The video memory stores the 240x240 pixel captured image so that it can be encoded. The module uses eight instantiated 64x900 dual-port block memories in order to store the full picture. Each block memory has one dedicated write port for the video capture module, and one dedicated read port for the encoder module. This memory architecture was chosen because the original implementation required eight simultaneous memory accesses, and each memory needed to be written and read simultaneously. The eight simultaneous memory accesses were later removed, but the memory architecture remained.

The video memory module is given input data, write enable, and write address signals from the video capture module. These inputs are used to write input data into memory. The encoder block gives the video memory a macro-line, block, and inter-block row which are converted into a memory address. This memory address is used to select the specific output pixel row for encoding.

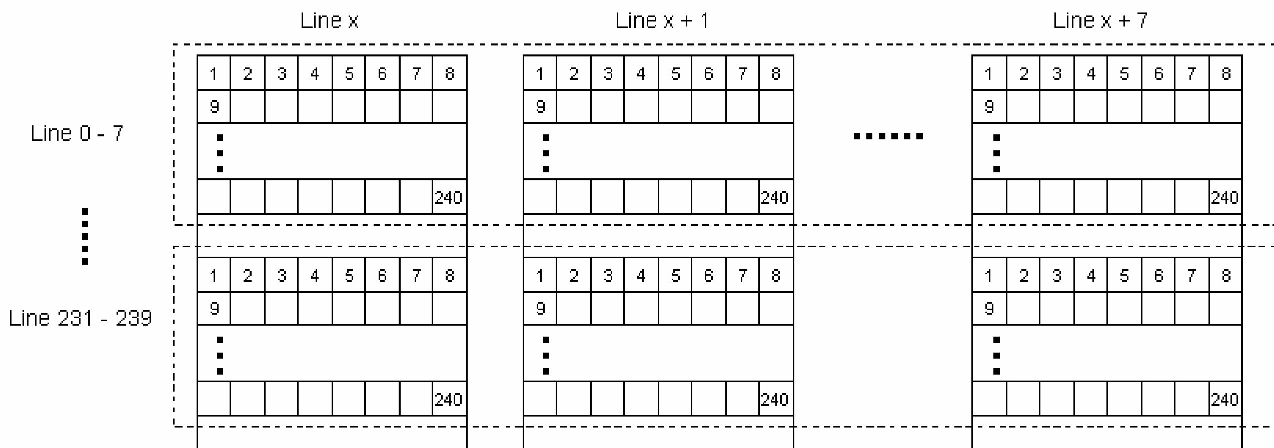

Figure 4 – Writing to Video Memory Process

Each memory address holds 8 pixels of grayscale information. Each memory holds 240 sequential pixels from the video capture module in 30 sequential memory addresses. Each 240 pixel line of video data is loaded in each memory sequentially, with the first line being written to the first memory, the second to the second memory and so forth. The ninth line of memory is then written again in the next 30 memory addresses of the first memory. The memory storage is shown in Figure 4.

Encoder (VIVEK)

The encoder module takes in grayscale information from the video memory and performs a proprietary Discrete Cosine Transform (DCT) based compression algorithm. This is based on the JPEG standard of image compressions and the visual irrelevance of high frequency noise to the human eye. The encoder converts 512 bits of data for each 8x8 pixel block into six 13-bit DCT coefficients, which is a compression of over 85%. The encoder module stores the output data in the wireless memory.

The encoder is overall a 4-stage pipeline with valid data appearing every eight clock cycles. In addition the encoder module takes in the macro-line being written from the video capture module and a transmit busy signal from the wireless transmitter. The encode module outputs an encode busy signal and the memory address of the coefficient block it is writing to the wireless memory. In addition, the encoder module outputs the macro-line, block, and inter-block row that need to be read from video memory. The wireless memory is accessed by outputting the block number as the address and a write enable signal.

Equation 1 – Matrix novation of DCT using 8x8 matrices.

$$DCT = T * IMG * T'$$

Equation 2 – Two dimensional Discrete Consine Transform algorithm.

$$F(m,n) = \frac{2}{\sqrt{MN}} C(m)C(n) \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} f(x,y) \cos\frac{(2x+1)m\pi}{2M} \cos\frac{(2y+1)n\pi}{2N}$$

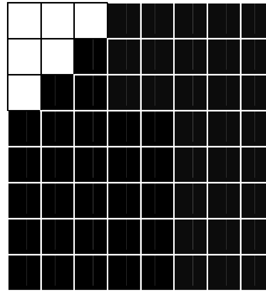

Figure 5 – DCT coefficients used in the encoded block.

The 2-dimensional DCT algorithm is shown in matrix form in Equation 1 and computation form in Equation 2. The encoding algorithm is implemented using a matrix multiplication with the DCT coefficient matrix. The DCT converts the 8x8 block into frequency data, with low frequency components in the top left corner and high frequency information in the lower right corner. The human eye is unable to discern high frequency information well, so the elimination of high frequency coefficients does not significantly affect picture quality and allows high levels of compression. The compression algorithm used eliminates over 90% of the available DCT coefficients, storing only the 6 coefficients in the top left corner in memory. The encoded coefficients are 13-bit signed numbers. A diagram of example DCT coefficients is shown in Figure 5.

The encoder module consists of three major components: the DCT multiply module, encoder FSM, and encode memory register. The top level block diagram is specified in Figure 6.

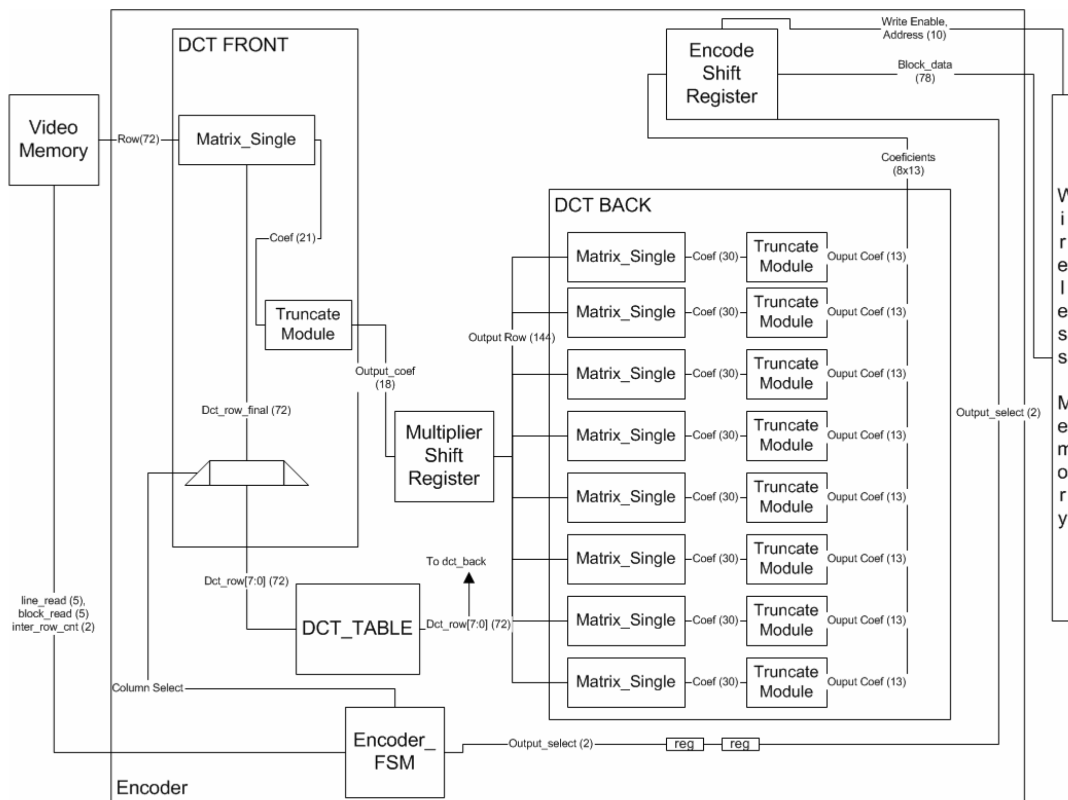

Figure 6 – Block diagram for encoder module.

DCT Multiply

The DCT multiply module takes a row of eight input pixels and performs a matrix multiplication. The module takes in eight rows of 8 pixels each and a column selector signal to multiply two 8x8 matrices together. The total matrix multiplication takes 64 clock cycles to complete, with a column of valid output coefficients appearing every 8 clock cycles. The DCT multiply module takes input data from the video memory. One caveat is that the input data from the video memory are unsigned 8-bit pixels values and the DCT multiplication requires signed numbers. Since all video capture coefficients are positive, each pixel value is sign extended before entering the DCT multiply module.

The matrix multiplication within the DCT multiply module is done using fixed point math. The numbers are multiplied together and the decimal point is noted. The output of the DCT multiply is a truncated version of the final coefficient, rounded to the nearest integer value. Fixed point math is necessary because the DCT coefficients are all numbers less than one. Fixed point was implemented instead of floating point to simplify the math functions necessary to calculate the output coefficients and to ensure that no data was lost or truncated unless explicitly done by the module.

The DCT multiply module contains four main sub-modules: DCT front, the multiplier shift register, DCT table and DCT back. The DCT multiply module takes in the column select value and selects the output row from the DCT table module to multiply against the input row. The output is the output coefficient column which is connected to the encode memory register.

DCT Front

The DCT front module takes an 8 pixel input row and 8 pixel column and performs a single row by column matrix multiplication. The input rows are eight pixels by 9 bits long which produce a single 21-bit output coefficient. The Xilinx device block multipliers have a limitation of doing 18-bit by 18-bit multiplications, so the output coefficient is truncated to an 18-bit number by removing the two MSB and one LSB. This design choice was made because MATLAB tests were run on maximum coefficient values. Based on the fixed point math, the two MSB bits could be removed without the loss of any information. The LSB removed causes the rounding of the last 1/256 of information which was deemed acceptable to retain image quality.

The DCT front module consists of two submodules: the single matrix multiplication and truncate modules. These modules are specified in the above description and therefore not described in individual sections.

DCT Table

The DCT table module is a static module that outputs the DCT coefficients for use in the DCT multiply module. The DCT coefficients were found using MATLAB and stored using fixed point notation. The coefficients are accessed as the DCT matrix (denoted as T) and also the transpose (denoted as T'). This static module is only to provide a common resource for all DCT coefficients. The values output do not change, and can be accessed by row and column, which allows any number of matrix manipulations to occur. The coefficients in the table are specified in 9-bit signed format in Table 2.

Table 2 – DCT Coefficient matrix as stored in memory using fixed point notation.

91	91	91	91	91	91	91	91
126	106	71	25	-25	-71	-106	-126
118	49	-49	-118	-118	-49	49	118
106	-25	-126	-71	71	126	25	-106
91	-91	-91	91	91	-91	-91	91
71	-126	25	106	-106	-25	126	-71
49	-118	118	-49	-49	118	-118	49
25	-71	106	-126	126	-106	71	-25

Multiplier Shift Register

The multiplier shift register is a memory accumulation shift register. The shift register takes an 18-bit coefficient and shifts it into memory every clock cycle. The shift register has a eight coefficient depth at which point it fires the done signal indicating that the data output is valid. The multiplier shift register is needed because the DCT front module only generates one coefficient per clock cycle, but the DCT back module needs a column of input data in order to compute the output coefficients. The multiplier shift register accumulates the coefficients and outputs a 144-bit vector or eight 18-bit coefficients to the DCT back module. The data is only valid once every eight cycles.

DCT Back

The DCT back module takes in a single 144 bit row (eight 18-bit coefficients) and eight 72-bit input DCT vectors (eight 9-bit coefficients). The DCT back module computes the second half of the DCT matrix multiply necessary for the matrix

transformation. The input row is multiplied in a row by column matrix multiplication eight times in parallel with eight DCT vectors. The matrix multiplication is pipelined and has latency of one cycle. The output of the matrix multiplications is a 30 bit signed number. In order to achieve acceptable compression rates and to eliminate unnecessary resolution in the output coefficients, the 30-bit signed number is truncated to a 13-bit signed coefficient. The fixed point math shows that the lowest 15 bits are less than one integer value and therefore hold very little value. The upper 15 bits are rounded to the nearest integer. Using MATLAB simulations, the maximum value for output coefficients is 2062 for an all white 8x8 block. This number requires only a 13-bit signed holder to contain the information. The two MSB are therefore removed because they do not contain any useful information. The DCT back module outputs eight 13-bit coefficients to the encode memory register.

The DCT back module contains two types of subcomponents: the row by column matrix multiplication modules and truncate blocks. The functionality of these blocks is described above and is therefore not outlined in a specific section.

Encode Memory Register

The encode memory register takes a 103-bit input (eight 13-bit coefficients) and a selector from the encode FSM. The encode memory register is a variable bit shift register which also outputs the address and write enable signals to the wireless memory. The DCT multiplication produces eight coefficients at a time, but based on the compression scheme only the six top corner coefficients should be written into memory. The memory register shifts the number of coefficients specified by the output selector from the encode FSM. Each block of the input image is encoded in six coefficients. The encode memory register stores the current number of coefficients stored. When this value reaches six, the register writes to memory and resets the stored coefficient count.

The encode memory register has a state counter to iterate the address. Upon reset the address is set to zero and the address increments by one every time the memory register fills with six coefficients. The write enable signal is set by the encode memory register to write the 78-bit output vector into the wireless memory address corresponding to the block encoded. Each encoded block is written into one memory address. When the address reaches the maximum block number of 899, the address resets to zero.

Encode Finite State Machine (FSM)

The encode FSM controls the data flow in the encoder module. The main function of the FSM is to specific the input values to the DCT multiply module and ensuring that input data is valid by addressing the video memory, as well as control output flow for the encode memory register and ensure that the valid data is stored and written into the wireless memory.

The encode FSM specifies all signals when the data is input into the DCT multiply pipeline. Because data flow is unpredictable given that the video capture module acquires useful data much slower than the encoder can encode it, the encode FSM calculates control values as the data enters the multiplier. Control signals that specify outputs are delayed to match the pipeline and therefore are in synchronization with the data as it exits the DCT multiply pipeline.

The encode FSM has three states but many state variables including line_read, block_read, column_select, and inter_row_cnt. These state variables as well as the macro-line input signal from the video capture module and the transmit busy signal from the transmitter module are the basis for state change in the encode FSM. A state transition diagram is provided in Figure 7 which summarizes the state transitions and the signals set in each state.

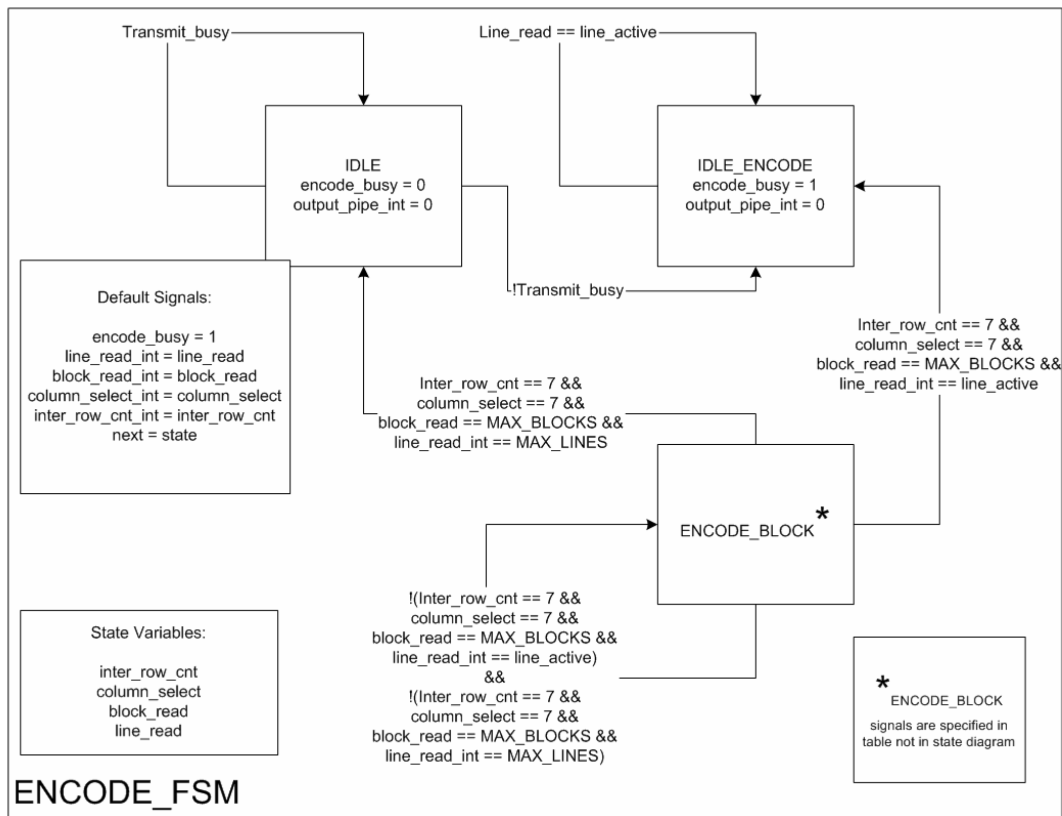

Figure 7 – State Transition Diagram for Encoder FSM

The signals set in the ENCODE_BLOCK state are specified in Table 3.

Table 3 –Outputs for ENCODE_BLOCK state of encode FSM.

Signal	Value
Inter_row_cnt	(inter_row == 7) ? 0 : inter_row + 1
Column_select_int	((inter_row == 7) && (column_select == 7)) ? 0 : column_select + 1
Block_read	((inter_row == 6) && (column_select == 7) && (block_read == MAX_BLOCKS)) ? 0 : block_read + 1
Line_read	((inter_row == 6) && (column_select == 7) && (block_read == MAX_BLOCKS) && (line_read == MAX_LINES) ? 0 : line_read + 1
Output_pipe_int	(3 – inter_row_cnt > 0) ? 3 – inter_row_cnt : 0

Wireless Block Memory (Encoder Side)

The wireless block memory module at the transmitter end provides a dual port memory interface between the encoder and wireless transmitter blocks. The memory module stores all the encoded data for a single frame. The memory module contains 900 valid address locations and each address holds a 78-bit value. The encoder block writes data to the wireless memory using the write port and the transmitter block reads data from the memory using the separate read port.

Wireless Transmitter (Noel Campbell)

The wireless transmitter block performs the task of sending encoded data from the camera end to the fixed end. The block consists of 4 main modules, the transmitter control unit, transmitter shift register, RS232 sender and the wireless packet sender. Figure 8 shows a diagram of these modules.

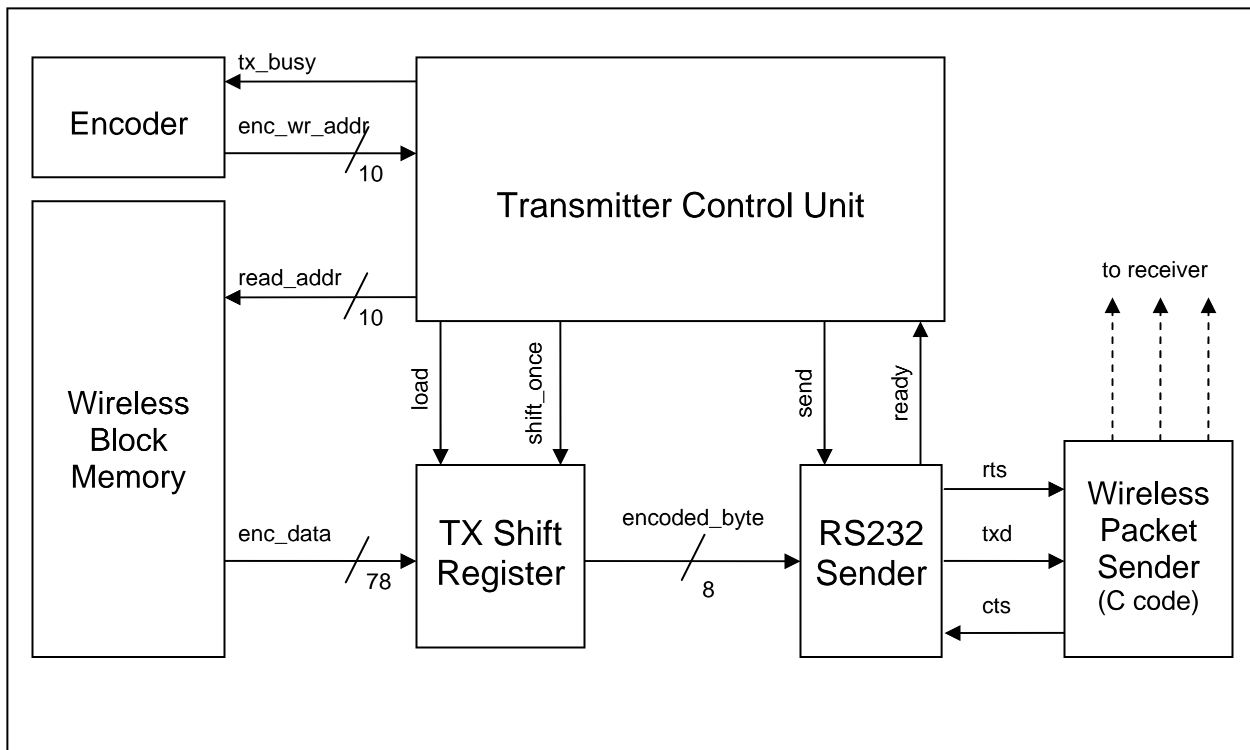

Figure 8: Transmitter Block Diagram

Transmitter Control Unit

The transmitter control unit interacts with the transmitter shift register and RS232 Sender modules to create a major-minor FSM architecture. The shift register and RS232 sender modules are instantiated within the control unit which manages their inputs. The TX control unit operates by incrementing the *read_addr* port to the wireless block memory and then enabling the *load* signal to the TX shift register module. The TX control unit then enables the *send* signal input to the RS232 sender module and waits for the *ready* signal from the RS232 sender. The control unit then enables the *shift_once* input to the shift register and tells the RS232 sender to send another byte. This process repeats until all 78 bits from the memory address have been sent. The control unit then increments the *read_addr* signal and repeats the same process. The *read_addr* is incremented repeatedly until all encoded blocks from the wireless memory have been sent serially by the RS232 Sender. The control unit then issues another *load* request and the cycle repeats continuously.

TX Shift Register

The TX shift register is a simple minor FSM that responds to a *load* request by loading 78 bits into internal registers. On each *shift_once* request, the minorFSM outputs 8 bits from the internal registers starting with the lowest 8 bits and ending with the highest 8 bits. It is the responsibility of the higher level major FSM to keep track of the number of shifts in between each load.

RS232 Sender

The RS232 sender module is a minor FSM that takes as input an 8 bit byte and serially outputs the bits on a single *txd* output line. The module operates at a 250 kbps baud rate (the maximum baud rate supported by the RS232 driver on the labkit). The module samples the 8 bit *encoded_byte* input when the major FSM issues a *send* request. The FSM then proceeds to send the input byte according to the RS232 protocol. The first bit sent is a start bit (a ‘0; bit), followed by the 8 data bits and then ending with a stop bit (a ‘1’ bit). Each bit is held for a period of 108 clock cycles in order to generate a 250 kbps baud rate (using the labkit 27 MHz clock). The process of sending a byte ends with a constant 1 (stop signal) on the *txd* line until the next *send* request is issued and another byte is sent.

The RS232 Sender uses flow control in order to deal with the difference in data rate between the wireless and serial connections. Flow control is implemented using the *rts* output and *cts* input signals. The RS232 sender issues a ‘0’ on the *rts* line so signal a request to send message to the receiver. The sender then waits for the receiver to issue a clear-to-send signal (a ‘0’) on the *cts* line which means that the receiver is ready to accept input data.

Wireless Packet Sender
The packet sender module receives serial input data from the RS232 sender module and assembles 11 byte packets that are transmitted wirelessly to the wireless packet receiver module. The code for the packet sender resides on an Atmel microcontroller on the Chipcon CC2420DBK development board. The packet sender module is quite different from the other Verilog modules in that it is coded in C (a printout of the code is available in the Appendix).

Receiver-Decoder-Display System

Wireless Receiver (Noel Campbell)
The wireless receiver block performs the tasks of receiving wirelessly transmitted encoded data from the wireless transmitter block, sending the data serially to the labkit, and then writing the data to block memory so that it can be decoded by the decoder block. Figure 9 shows a block diagram of the wireless receiver block.

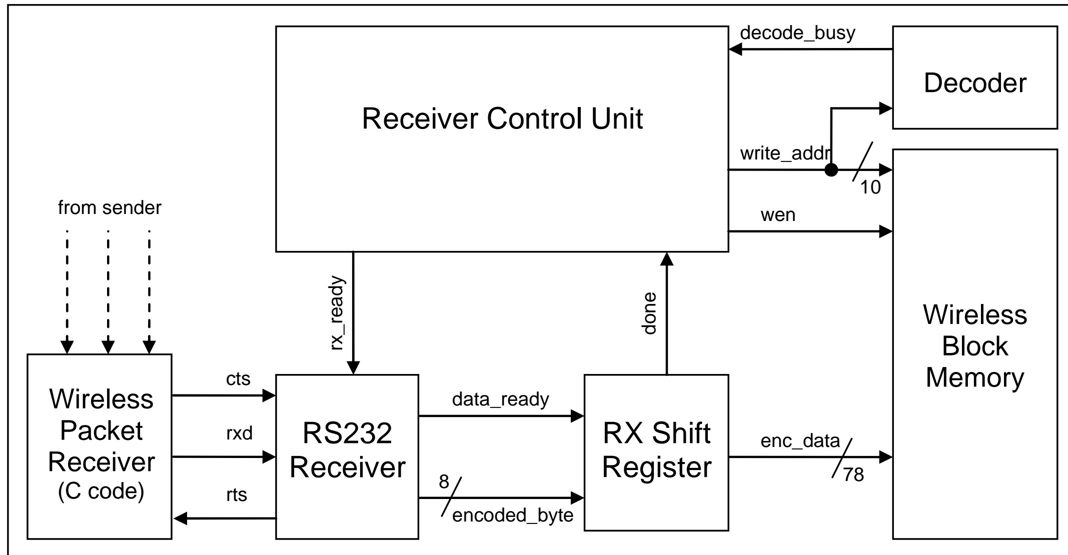

Figure 9 - Wireless Receiver Block Diagram

Wireless Packet Receiver
The packet receiver module is responsible for receiving wireless data packets from the wireless sender module, and sending the data serially from the wireless kit to the RS232 receiver module. The module is coded in C and resides on the CC2420DBK development board. The module uses transmits serial data using a baud rate of 250 kbps. In order to avoid problems caused by the slow wireless bit rate, the packet receiver uses flow control when sending RS232 data to the receiver. Flow control is implemented by using the request-to-send *rts* and clear-to-send *cts* signals to determine when it is safe to send data to the RS232 receiver. The wireless packet receiver knows it is okay to send data when it detects a low signal (logical 0) on its clear-to-send line.

Receiver Control Unit
The receiver control unit combines with the RS232 receiver and RX shift register modules to form a major-minor FSM architecture for receiving encoded data from the wireless sender and writing it to block memory where it can be accessed and decoded by the decoder block. The control unit controls the *rx_ready* and *write_addr* output signals. The control unit begins operation after reset by setting *write_addr* to memory address zero and then setting *rx_ready* to high to signal to the RS232 receiver that it should start receiving serial data. The control unit issues a *wen* write enable signal each time the *done* signal is issued by the RX shift register module (indicating that a new 78 bit block of data is ready to be written to memory. After a buffer cycle to ensure a valid write function, the *write_addr* signal is incremented and the process repeats until *write_addr* equals 899 and the address must be set back to zero.

RS232 Receiver
The RS232 receiver module is responsible for receiving serial data from the wireless packet receiver and sending out 8-bit bytes to the RX shift register. The module operates by first checking that the value of the *rx_ready* input signal is high, signaling that it is okay to receive and send data to the RX shift register. The module operates at a baud rate of 250 kbps and functions by first checking for a start bit (logical 0) at each positive edge of the clock. When the beginning of a start bit is detected, the FSM counts to a value of 53 (half the baud rate count) and samples the signal again to ensure that the bit really

is a start bit. The module then repeatedly counts up to 107 (the count value that creates a baud rate of 250 kbps) and samples the input signal on the *rxd* line at the end of each count. This counting repeats until 8 data bits are detected. The FSM then transitions to a “receive stop bit” state for the 10th bit and waits for the next start bit. During the “stop bit state,” the module also asserts the *data_ready* output signal which lets the RX shift register know that it is okay to shift in another 8 bit byte.

RX Shift Register

The RX shift register module is responsible for receiving the 8-bit byte inputs from the RS232 Receiver module and shifting out 78 bits at time each time the internal buffer fills. The module shifts 8 bits into internal registers each time the RS232 receiver module asserts the *data_ready* line. When the shift register fills with 78 bits, it asserts the *done* output signal which causes the RX control unit to write to the wireless block memory.

Wireless Block Memory (Decoder Side)

The wireless block memory module at the receiver end provides a dual port memory interface between the wireless receiver and decoder blocks. Similar to the wireless block memory on the encoder/transmitter side, the memory module stores all the encoded data for a single frame. The memory module contains 900 valid address locations and each address holds a 78-bit value. The receiver block writes data to the wireless memory using the write port and the decoder block reads data from the memory using the separate read port.

Decoder (VIVEK)

The decoder module takes in a set of DCT coefficients from the wireless memory and performs a proprietary Inverse Discrete Cosine Transform (IDCT) based decompression algorithm. The equation for the IDCT is specified in Equation 3 and Equation 4. This is based on the JPEG standard of image compression and extrapolates the lower frequency image information from a compact set of coefficients. The decoder converts six 13-bit signed coefficients into a 512-bit 8x8 pixel block. The decoder module stores the output data in the video memory.

The decoder is very similar to the encoder, and in fact shares many of the same modules. For the modules that are the same, references are made to the previous modules for details on the implementation. The decoder is a 4-stage pipeline with valid data appearing every eight clock cycles. The decoder connects to the wireless transmitter through a decoder busy signal and the transmitter sends it an address active signal. These signals prevent the decoder from decoding blocks which have not been updated with fresh data. The decode module outputs many signals to the wireless memory and to the video memory in order to control data flow.

Equation 3 – Matrix notation of IDCT using 8x8 matrices.

$$DCT = T' * IMG * T$$

Equation 4 – Two dimensional Inverse Discrete Cosine Transform algorithm.

$$f(x,y) = \frac{2}{\sqrt{MN}} \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} C(m)C(n)F(m,n) \cos\frac{(2x+1)m\pi}{2M} \cos\frac{(2y+1)n\pi}{2N}$$

The decode module has multiple subcomponents including DCT multiply and the decode FSM. The block diagram in Figure 10 shows the interconnection of the decoder’s submodules.

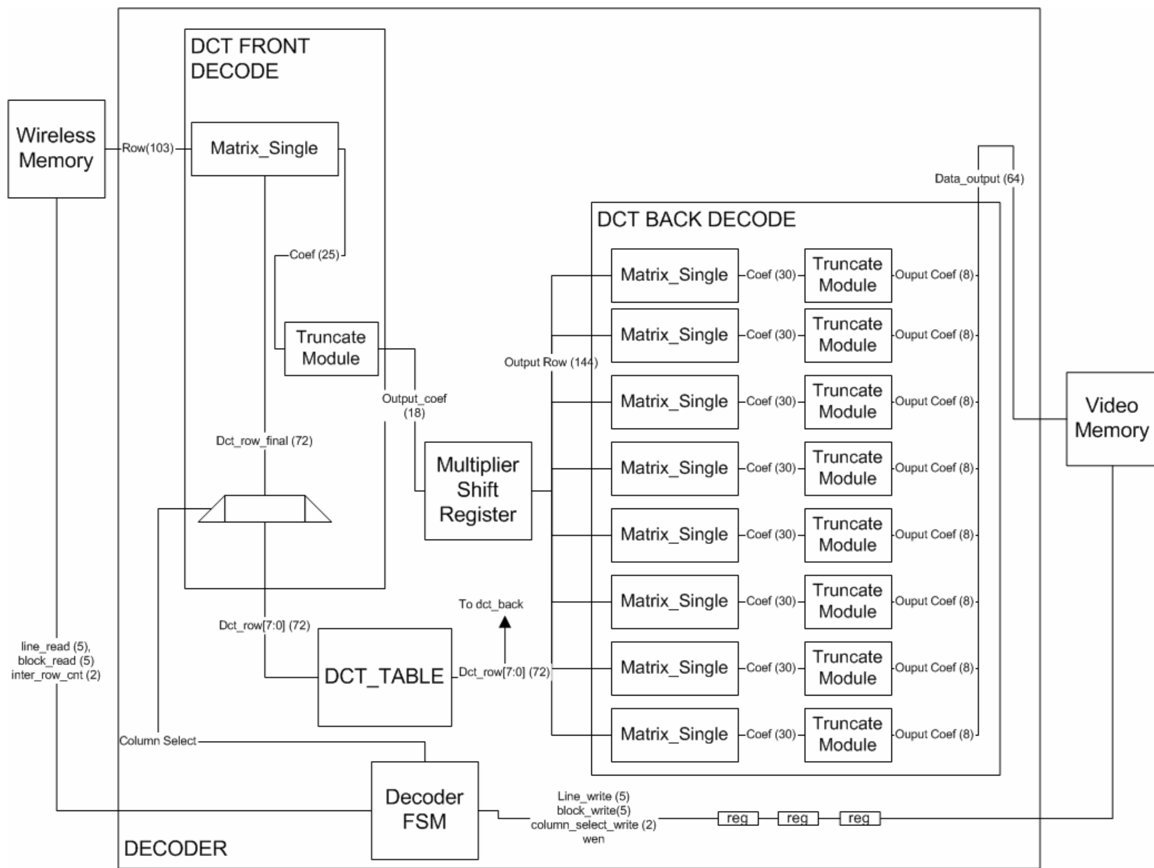

Figure 10 – Block diagram of Decoder module.

DCT Multiply Decode

The decoder DCT multiply is not significantly different than the DCT multiply in the encoder. The one critical difference is the different outputs used to multiply against the coefficients in each stage. In the encoding stage, each 8x8 pixel block was multiplied by the DCT coefficient matrix (denoted T) and then multiplied by the transpose matrix (denoted T'). In this stage, the reserve happens. The set of image post-DCT coefficients are multiplied by T' and then T. This creates the IDCT and allows the image to be decoded.

DCT Front Decode

The DCT front module is not significantly different than the DCT front in the encoder. The only difference is the bit widths coming into the first stage multiply. The coefficients are 13-bit signed numbers, and thus require different multiply modules to accommodate the larger input width. However, the output is still truncated to 18-bit signed coefficients, and the data flow path is identical.

DCT Table

This module is unchanged from the encoder. Please refer to the DCT table module in the encoder for details on how this module is implemented.

Multiplier Shift Register

This module is unchanged from the encoder. Please refer to the multiplier shift register module in the encoder for details on how this module is implemented.

DCT Back Decode

This module is unchanged from the encoder. Please refer to the DCT back module in the encoder for details on how this module is implemented. The only change is that the DCT back decode does not output to a memory register but outputs directly to the video memory.

Decoder FSM

The decoder FSM is implemented in a very similar manner to the encoder FSM. The state transitions and signals are almost identical, except that the decoder FSM is controlling data flow from the wireless memory to the video memory and the encoder is controlling the opposite. A state transition diagram is included in Figure 11.

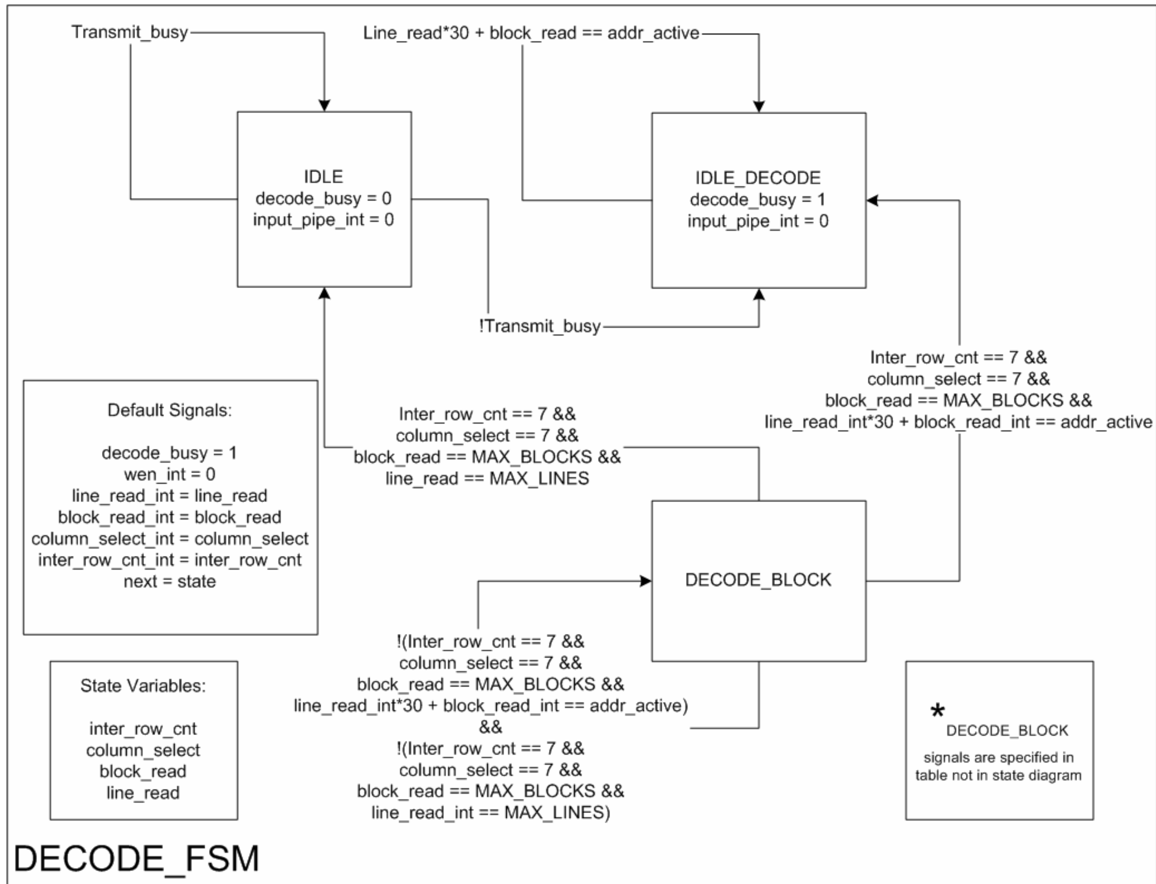

Figure 11 – State Transition Diagram for Decode FSM.

Video Memory (Decoder)

This module is very similar to the memory architecture used on the encoder side but instead of eight 64X900 dual-port block memories, it uses eight 8X7200 dual-port block memories. Each block memory has one dedicated write port for the decoder module, and one dedicated read port for the video display module. The reason why the width and depth of these memories changed is because the decoder decodes 8 pixels at a time, all of which can be written into memory immediately. There is no reason to store them in a shift register even though the Read 64 module requires 64 bits of information. The block memory on the labkit allows different port widths so the read port for the video display is 64 bits.

The video memory module is given input data, write enable, and write address signals from the decoder module. These inputs are used to write input data into memory. The decoder block gives the video memory a macro-line, block, and inter-block row which are converted into a memory address.

Each memory address holds 1 pixels of grayscale information. Each memory holds 240 sequential pixels from the video capture module in 240 sequential memory addresses. Each 240 pixel line of video data is loaded in each memory sequentially, with the first line being written to the first memory, the second to the second memory and so forth. The ninth line of memory is then written again in the next 240 memory addresses of the first memory.

Video Display (RAYMOND)

This module has fewer sub-modules than the Video Capture module. While it also has a VGA Controller, Display, and Delay module underneath, the main difference is that it has a Read 64 module which is responsible for reading from the Video Memory on the decoder side and passing the grayscale data to the Display module for visual display.

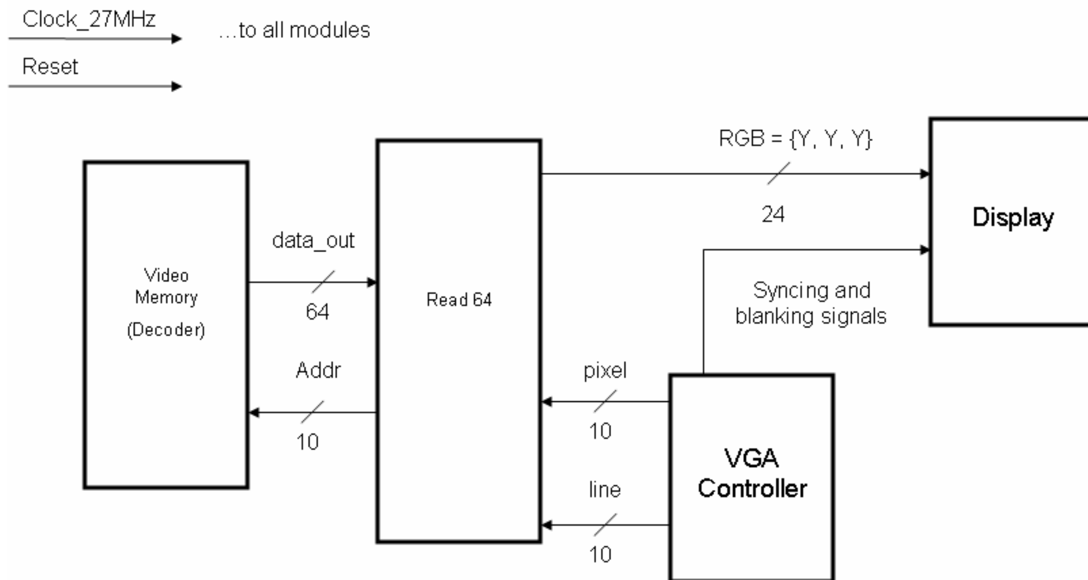

Figure 12 – Block diagram of the Video Display Module.

Read 64

The Read 64 module reads from the video memory on the decoder side. It is similar to the Store 64 module in that it has two internal counters read_counter and read_counter 2. These counters are used to know when the 64 bit shift register is empty and when the reader has reached the end of a line. Every clock cycle, the Read 64 module outputs 24 bits of RGB data that consists of a Y value repeated three times (RGB = {Y, Y, Y}) so no conversion is required here. Two shift registers are needed because when one shift register is being unloaded, the second is filled with the next 8 pixels worth of data. So, when one shift register is half way unloaded, the second shift register is loaded with the next address's data. This alternating pattern allows a continuous stream of data to be displayed on the screen. The memory architecture for the Video Memory on the decoder side is similar to the memory on the encoder side and is further described in the Video Memory (Decoder) module.

Testing and Debugging

Video Capture/Display Modules

In order to test the system, I created several test benches for each module to verify its functionality. When all the modules were complete, both Video Capture and Video Display, I created a top level module and a corresponding top level testbench to make sure all the signals were transitioning as I expected. The top level testbench waveform is given below.

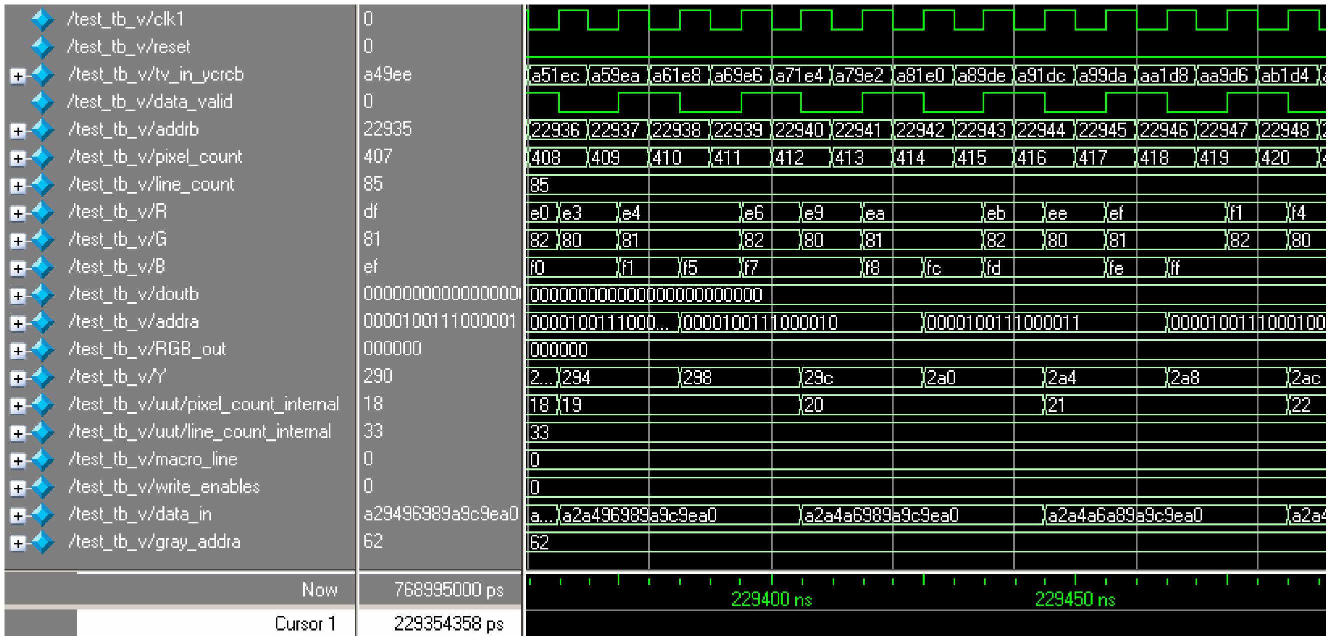

Figure 13 – Video Capture/Display Testbench

Here, I simulated camera data by beginning lines with the FF 00 00 XY sequence every 700 cycles (an approximate length of a line coming from the camera) and then incrementing the data by a fixed amount for the remaining clock cycles. Then I checked to see that the data_valid signal was transitioning as I expected, mainly that it transitions every other clock period. Then I checked to see that addrb, the address that controls where the color data is read from on the encoder side, increments every time the VGA pixel counter increments. After that, I checked to see that the RGB values were being converted from Y, Cr, Cb values. I was unable to verify the values they were being converted to except with a physical test, however this waveform shows that the R, G and B values are held for two clock cycles every other time its value changes. This makes sense because the camera outputs only one luminance (Y) and one chrominance value (Cr or Cb) per pixel. Looking at the equations for converting Y, Cr, Cb to RGB, this makes sense. Finally, I looked at the Y values coming out of my top level test bench to see if my reader was working properly. This was a test to verify that my reader was able to output sequential Y values which corresponded to the Y values that were written into memory. Here, I have verified this result because the Y value increments the same amount every clock cycle.

The entire traffic light controller was tested and debugged in a systematic method involving testbenching and FPGA simulation. Each module was tested using a test bench and writing comprehensive tests for all possible input configurations. Test benches were used to test behavioral and also post-place and route models of the design. At times, these simulations did not agree but an effort was made to have every module pass every test bench in both modes. Details of important modules and the testing are represented below.

Encoder

Every module in the encoder system was test benched and tested thoroughly in ModelSim. These comprehensive test benches are in the appendix and can be reviewed for completeness. The test results for all the submodules were successful, but are too numerous to include in this report. The top level testing and testing strategy is summarized below for the encoder module.

The simulation results shown in Figure _ show the encoder system functioning correctly with sample data input. The coefficients and the write enable indicate that the correct coefficients are being generated and they are synchronized. The block being encoded is the 8x8 matrix of all 255 (all white) values. Further tests included random matrices, with the corresponding verification with MATLAB outputs.

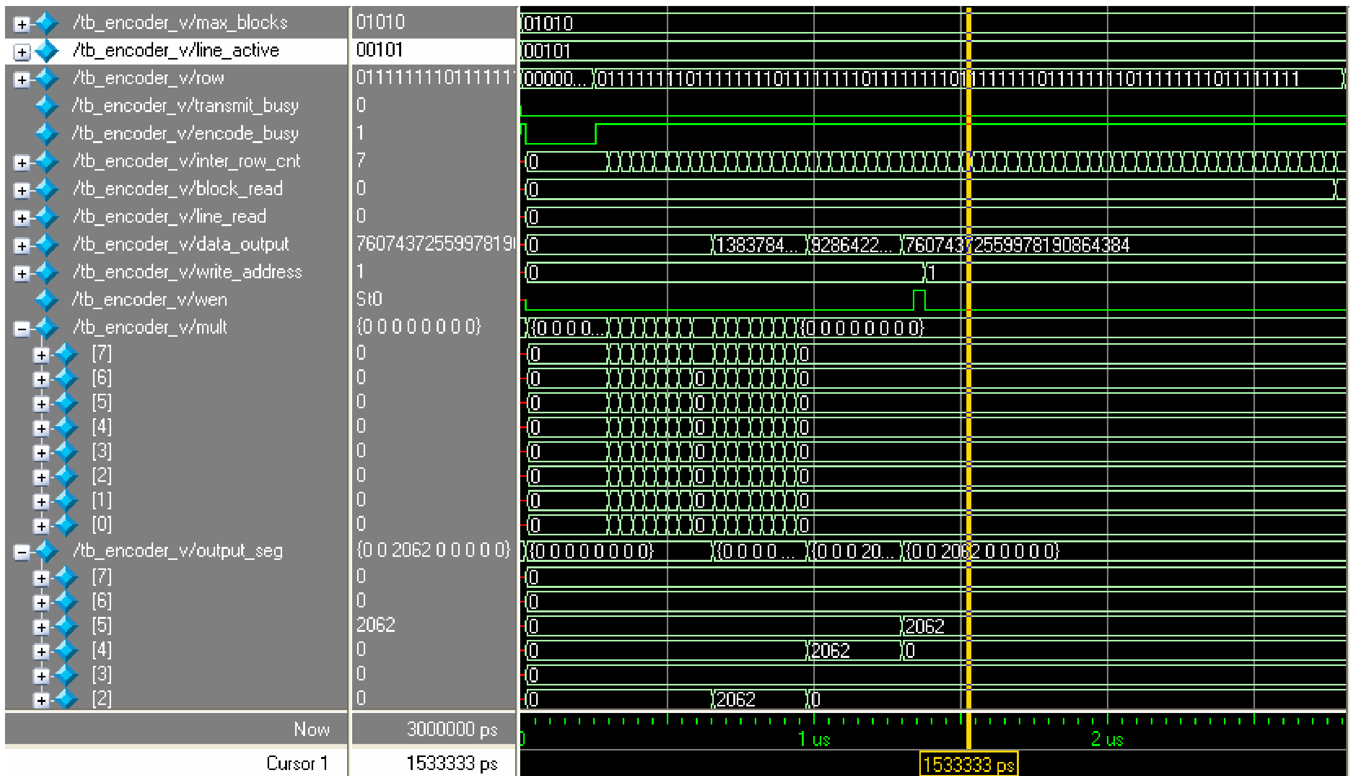

Figure 14 – Video Encoder Testbench ModelSim waveform outputs.

The test shows that the 2062 coefficient in the 6th coefficient position and zeros in all lower coefficient positions is synchronized with the written enable. The write address is also incrementing in the cycle after the write, which shows the system is moving forward into the new write cycle. The encode busy signal is high for the duration of the encoding.

Further testing on the encoder included connecting the encoder and the wireless transmission module, and placing a known COE file in the video memory and using the logic analyzer to verify that the outputs were being sent correctly. This testing was physical testing and was very useful in debugging and was proven correct in the final testing phases.

Decoder

Every module in the decoder system was test benched and tested thoroughly in ModelSim. These comprehensive test benches are in the appendix and can be reviewed for completeness. The test results for all the submodules were successful, but are too numerous to include in this report. The top level testing and testing strategy is summarized below for the encoder module.

The simulation results shown in Figure _ show the decoder system functioning correctly with sample data input. The coefficients and the write enable indicate that the correct coefficients are being generated and they are synchronized. The block being encoded is the 8x8 matrix of all 2062 which is the encoded equivalent of an all white block.. Further tests included random encoded matrices, with the corresponding verification with MATLAB outputs.

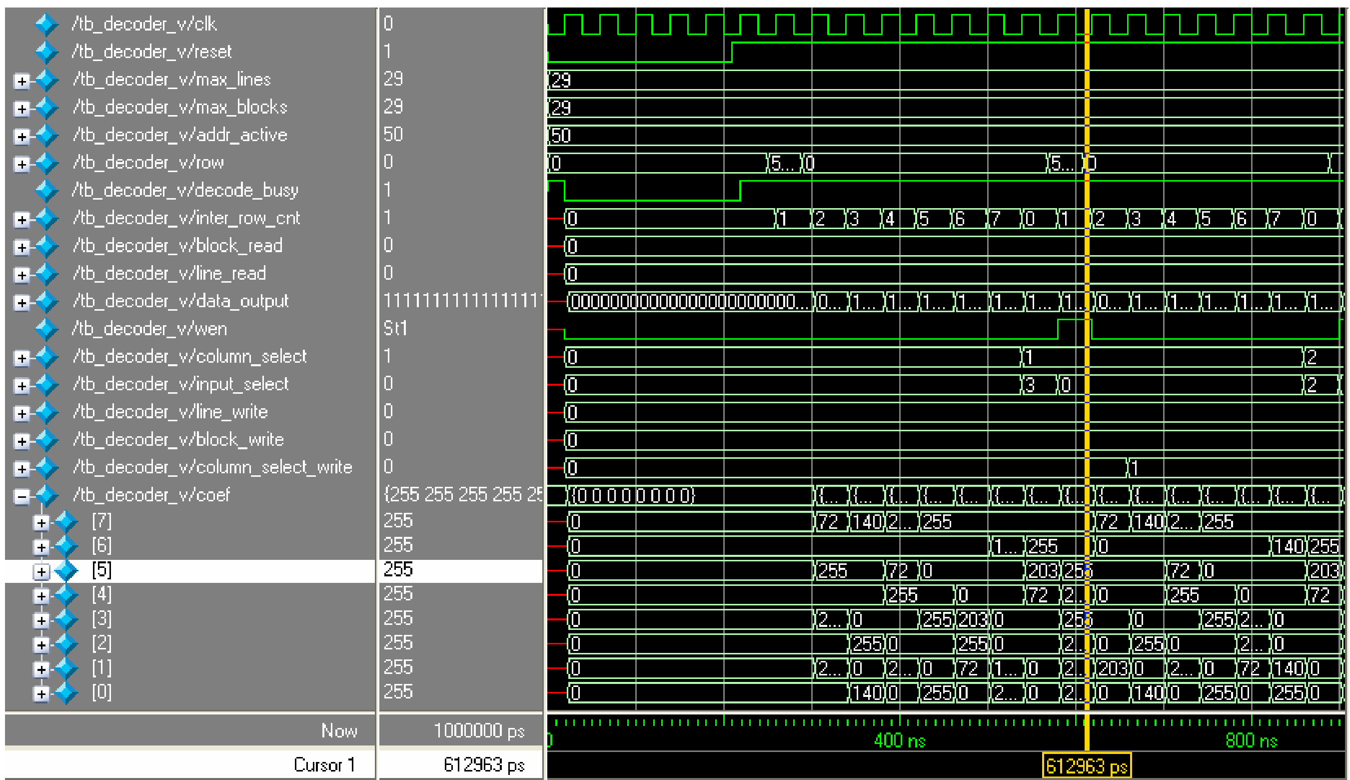

Figure 15 – Decoder Testbench ModelSim waveform output.

The test shows that the column output from the decoder is all 255 data, indicating that the coefficients were decoded correctly and recreated the all white block. The column_select_write is also incrementing in the cycle after the write, which shows the system is moving forward into the new write cycle. The decode busy signal is high for the duration of the encoding.

Further testing on the decoder included connecting the decoder and the video display unit, and placing a known COE file in the wireless memory. The COE file was for coefficients of black and white lines on the display. This testing helped to verify the encoding and video display subsystems, and also proved both systems in the final iterations.

Wireless Transmitter/Receiver Modules

Testing of the wireless modules was conducted by creating separate ModelSim testbenches for each of the modules (the testbench for each module is included in the appendix). Because of the difficulty of simulating some of the input data such as the serial input signals associated with RS232, further testing was necessary to validate the communication between the RS232 modules and the wireless kit. This was done by connecting the labkit to the wireless sender module using a serial was created that looped through the ASCII alphabet to see that the data path was functioning correctly. Further testing was conducted by connecting the receiver wireless kit to the labkit and then using a Verilog module that would send the hex value on the switches of the sender labkit and display the ASCII text on the alphanumeric display of the receiver labkit. Both of these tests functioned correctly and proved that communication was functioning properly.

When all of the modules (video capture, encoder, transmitter, receiver, decoder, display) were connected in the top level labkit file, debugging was done using the logic analyzers. The screenshot in Figure 3 displays the results of this debugging.

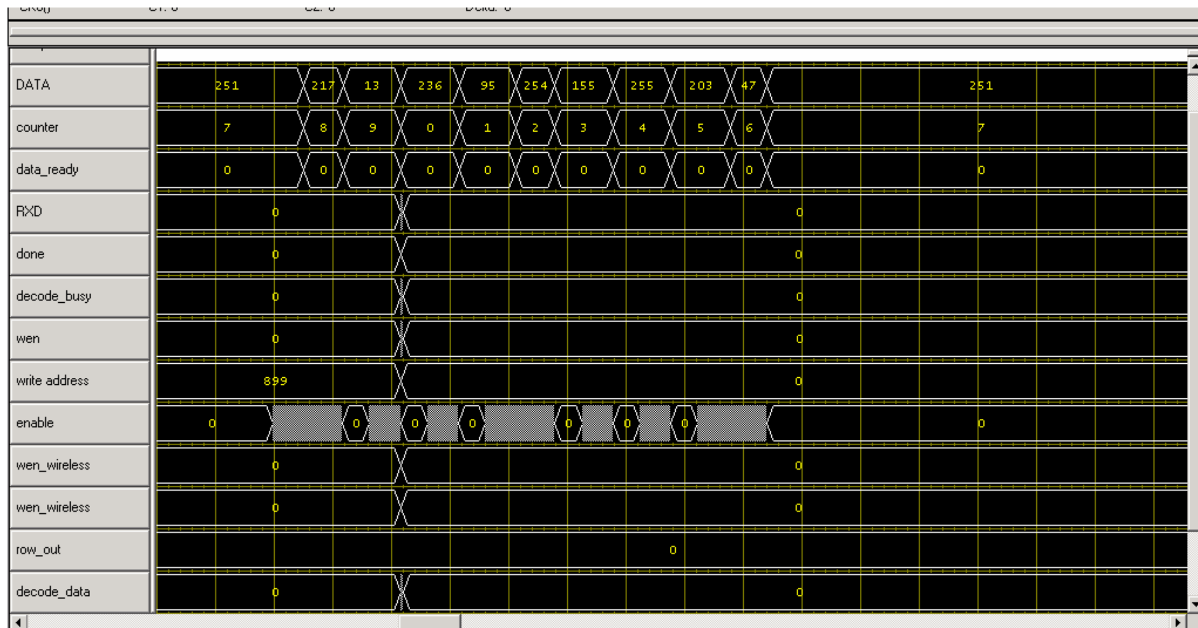

Figure 16: Logic analyzer screenshot verifying overall system data flow

The screenshot in Figure 3 shows the bytes of data (DATA) being received by the RS232 receiver module and the memory addresses (write_address) to which these values are being written. It should be noted that write actions are not occurring for every data byte because the wireless memory architecture stores 78-bit values which contain the encoded data for an entire 8x8 pixel block.

Physical Testing

The major component of the systems integration testing involved a series of physical tests using stubbed labkit implementations and known COE files in memory. These tests were performed to quickly debug only the broken parts of the system and to allow quick synthesis of only the incorrect portions of the overall project. Our testing methodology was to stub out the video capture and video encoder, and place a known COE file in the wireless memory on the encoder side. Leaving only the wireless transmitter in the labkit decreased the synthesis and generate programming file time by over 75%. Using this implementation and a known implementation on the wireless receiver side, the logic analyzer was used to debug synchronization signals in the wireless transmitter.

Using this testing methodology, we were able to see exact synchronization in the data packets received, and were able to narrow down the exact errors in the matter of a few iterations. Physical testing also included testing and viewing known COE file images, we were able to confirm the proper functionality of various subsystems of our final project.

Conclusion

The objective of this final project was to design and implement a complex digital system combining video, digital encoding, and wireless transmission of data. The analysis presented in the previous sections shows a fully functional wireless video surveillance system, along with in-depth analysis of the modular structure of the design. A comprehensive testing methodology was proposed and executed, which further validated the functionality of the system.

This final project taught integration of complex digital systems and the digital interface between many diverse analog and digital components. For designing even more complex systems, this final project demonstrates the challenges that are faced in integrating systems, even with fully functional separate parts. Moreover, the final project allows us to test our hypothesis that a high resolution image could be transmitted over wireless bandwidth to create a functional and useable surveillance system. Future improvements of this project might include developing a more robust wireless communication protocol, implementing full-color transmission, and potentially increasing the number of input video cameras.