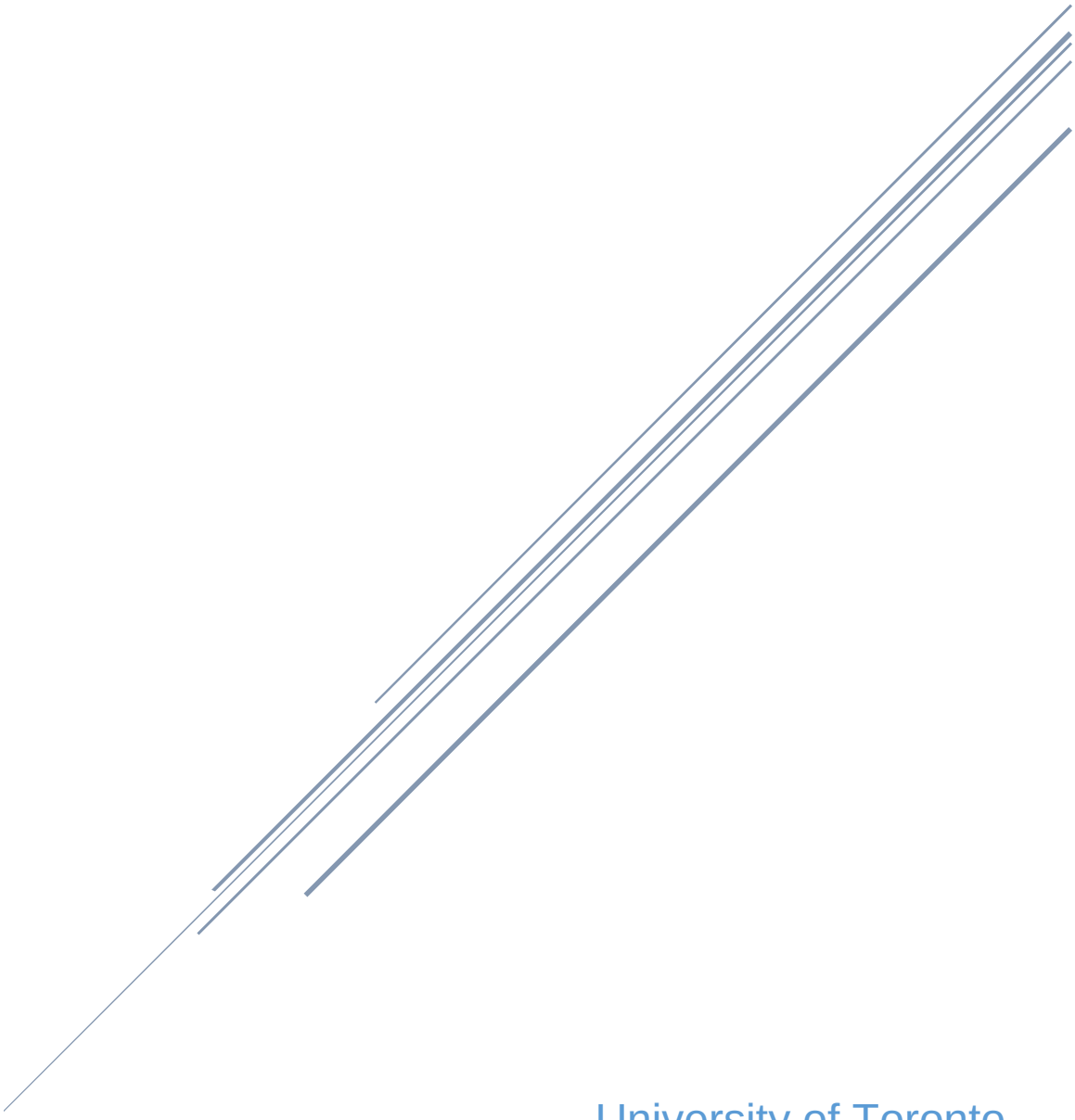
REAL-TIME DISCRETE COSINE TRANSFORM STEGANOGRAPHY

by Soon Kyu Lee, Thomas Sattolo and George Shehata



University of Toronto
ECE532 – Digital Systems Design

Table of Contents

List of Figures

List of Tables

Overview

Introduction

Steganography is the process of concealing information into images and other media. This differs from encryption in that, while encryption attempts to conceal the content of a message, steganography attempts to conceal the fact that a message exists. Two main uses of steganography are secret communication and digital watermarking.

In the context of this report, a cover image refers to the image within which a message will be embedded, and a stego image refers to the image produced by a steganographic algorithm. The quality of a steganographic algorithm is often determined by three metrics: imperceptibility, which is a measure of the algorithm's ability to hide a message's presence within a cover image; robustness, which is a measure of the retention of the hidden message upon an alteration of the stego image; and capacity, which is a measure of the size of the hidden message which can be embedded [1].

However, since we are more interested in the application of steganography as a communication medium, we are not interested in the capacity of a steganography algorithm as much as its throughput. The throughput is defined by how much of the message can be encoded into the image in a specified period of time; an increased capacity increases the throughput, but the speed at which the embedding can be achieved also affects the throughput in a significant manner. This definition of the throughput assumes images are always available to encode into in order not to interrupt the encoding process; this is always attainable assuming a video stream at high resolution such as 720p or 1080p at 30-60 fps.

The most basic steganographic algorithm is least significant bit (LSB) insertion [2], which replaces the LSBs of an image's pixels with bits of a message. Although its implementation is simple, the imperceptibility of the hidden message depends greatly on the cover image and the resulting stego images have low robustness [2]. Discrete cosine transform (DCT) steganography is a more complex algorithm which embeds the message in the cover image's frequency domain. Specifically, the DCT operation is used to convert an 8x8 block of pixels into the frequency domain, the message is embedded there, possibly by using an LSB embedding method, and finally, an inverse DCT (IDCT) operation is undertaken to convert the

image back into the spatial domain [3]. Through this manipulation of the image in the frequency domain, high robustness and imperceptibility can be achieved [2].

Most of the existing research demonstrating high-speed software steganographic implementations uses LSB insertion due to its simplicity. Consequently, these implementations suffer not only from high perceptibility and low robustness, but also from low throughput which limits their use to low bandwidth applications such as audio transmission (e.g. VoIP). A few hardware designs have been implemented [4] which have a much higher throughput (enough for 4K video at 60 fps); however, they still employ low complexity steganographic systems [4] which have limited security.

Published high-speed implementations of DCT steganography in hardware or software do not exist. Thus, a sample DCT steganography module was implemented in software for the purposes of this project and was found to operate at a limited throughput of 2280 bits per second (bps). However, many of the operations within the DCT and IDCT are parallelizable and this makes DCT steganography amenable to hardware acceleration in order to achieve a large performance gain.

Goals

The aim of this project is to implement a high-speed steganographic system of high complexity. The design will target real-time media applications such as video streaming without compromising the steganographic quality. Existing software methods, even with simple steganography schemes, have limited throughput while high-throughput hardware research has only explored low complexity steganography schemes. Additionally, by integrating encryption to the more complex steganography scheme in hardware, we will be able to quickly create images with inconspicuously embedded secret messages.

Top-Level Block Diagram

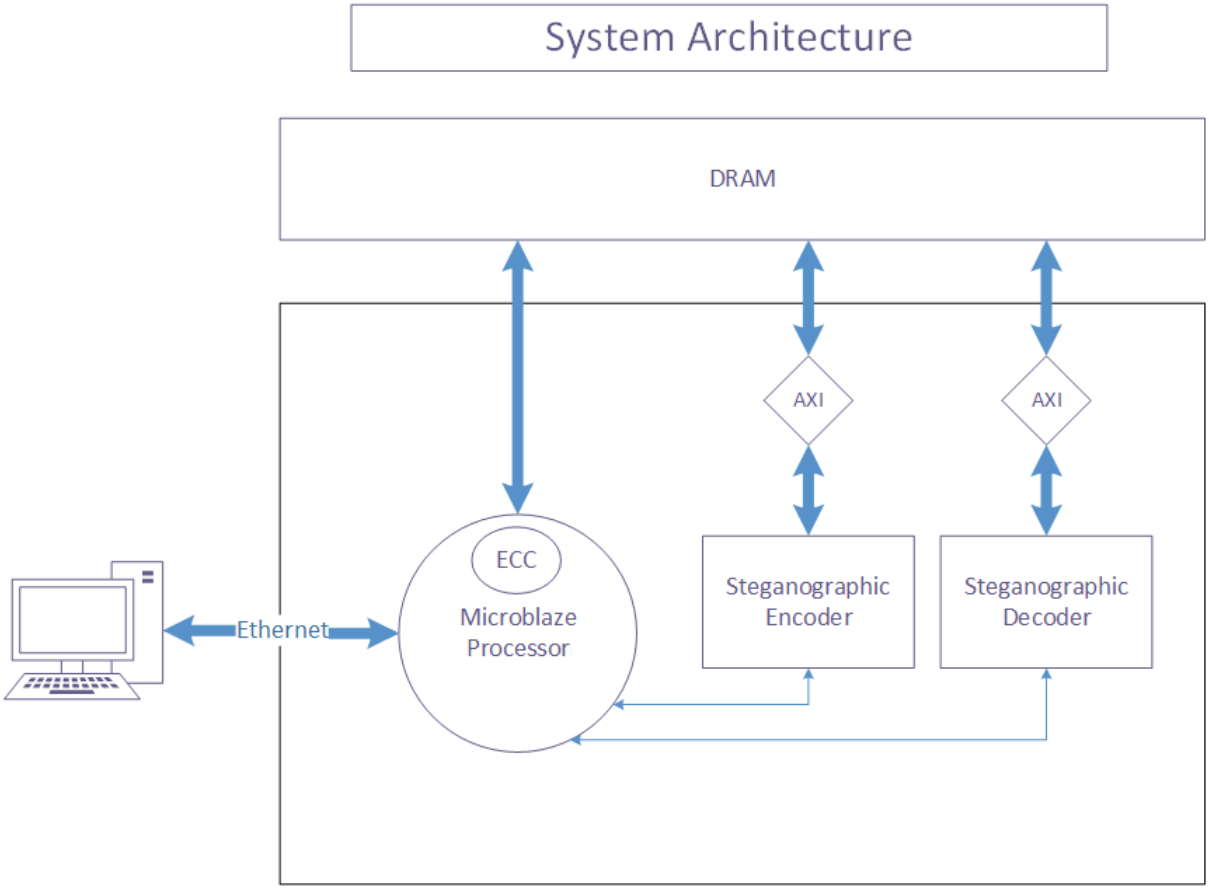


Figure 1 Project Top-Level Diagram

The top-level block diagram, Figure 1, includes three main blocks: the steganographic encoder, the steganographic decoder and the MicroBlaze processor. Both core steganography blocks are connected to DRAM through custom-made AXI communication modules. Encryption and decryption are performed in the encoder and decoder blocks respectively (not shown) to add an extra layer of security to the steganography algorithm. Error correcting codes are included as a sub-block of the processor as they were implemented in software. A PC connected via Ethernet to the product is shown because it is an integral part of the system used as the source of the message and image.

Brief Description of IP

Table 1 lists the modules in the project with short descriptions of their functions. It also indicates whether the implementation is in hardware or software, and how the code was obtained. Note that 'mostly new' indicates that some existing IP was heavily modified to fit our needs. These modules are described in great detail in **Description of the Blocks**.

Table 1 Summary of Existing IP Modules and Their Functions

Module	Description	Type	Source
DCT	Performs 2D discrete cosine transform	Hardware	Open-source
Inverse DCT	Performs 2D inverse discrete cosine transform	Hardware	New
Embedder	Replace least significant bits of frequency domain data with message	Hardware	New
Extractor	Re-create message from least significant bits of frequency domain data	Hardware	New
Encryption Module	Encrypt message with an LFSR-based stream cipher.	Hardware	New
Decryption Module	Decrypt message with an LFSR-based stream cipher.	Hardware	New
DRAM Interface	Transfer data from/to DRAM to/from encoder/decoder	Hardware	Mostly new
Ethernet Client	Send data to DRAM. Runs on a PC.	Software	New
Ethernet Server	Receive data from a PC	Software	Mostly new
Error Correction	Add error correcting codes when message is received from PC; correct error when message is sent to PC	Software	New

Outcome

Functionality

First and foremost, the project was successfully implemented in accordance with the requirements specified in the proposal. In other words, the final project includes encryption and decryption as well as steganographic encoding and decoding. The extra objectives, on the other hand, were not all achieved. For reference the extra objectives were as follows:

- Compression
- Video processing
- Streaming the messages into FPGA in real-time
- Live video capture (i.e. processing video from a camera)
- Video communication between multiple FPGAs

Compression was originally included as an objective of the project, but it was soon realised that there is no compelling rationale to implement compression in hardware. Since the message is short compared to the images, using software to implement compression is entirely compatible with real-time steganography. For this reason, implementing compression was relegated to lower priority than the other objectives.

It must be emphasized that the original requirements specified a product that could encode and decode still images rapidly, not a product that could encode and decode live video. Video processing was always an objective rather than a requirement. That said, video processing was implemented, but with significant limitations. Most notably, only the encoding side of steganography was done. The reason for this is that any testing of video decoding would necessitate having encoded video saved to a PC. This is very difficult because the only reasonable video output vector is HDMI. The directionality of HDMI ports is hardware bound, so those on a typical PC cannot be used to capture encoded video; although HDMI capture devices do exist, they are complex and expensive.

Additionally, the encoded video has some imperfections that are not present in encoded images. Firstly, video frames are sometimes skipped as the framerate of the VDMA procedure was slower than the input video framerate. The output also contain small, moving red dots that are not present in the input. It is unlikely that either of these issues is directly due to the steganographic encoder because it does not ever introduce red dots into still images and it is

more than fast enough to handle video. Presumably, some part of the surrounding interface logic is not quite right.

The remaining objectives are deeply dependent on the video processing. Live video capture and video communication between multiple FPGAs requires that video encoding and decoding be implemented. As for streaming the messages into an FPGA in real time, all the necessary hardware and software was created. However, since messages were only encoded into still images it is more convenient just to transfer the entire message into DRAM along with the image in which it is to be encoded.

Next Steps

Given that some of the project original objectives were left uncompleted, it is fairly clear what the next steps should be to improve the product. Namely, decoding of live video should be implemented and encoding quality should be improved. After these goals are achieved, video communication between two FPGAs (i.e. directly connecting an FPGA with the encoder to on with decoder) and live video should be easily within reach.

In Hindsight

If we could start over, there are several things we would have done differently. However, as our project was ultimately successful, none of our mistakes proved fatal. In fact, all of these mistakes simply led to some time being wasted. That said, in hindsight we would have:

- Investigated the available library blocks more thoroughly (and found the AXI VDMA) before attempting to directly modify the AXI master source code.
- Simulated critical open-source IP blocks (i.e. the DCT) more thoroughly before integrating them into our design.
    - The IP was not poorly designed; however, some incorrect assumptions were made about its operation.
- More thoroughly considered the likely benefit before attempting to implement extra features (e.g. trying to implement raw Ethernet communication without using TCP/IP).
- Simulated all interactions with the DRAM by replacing it with a BRAM from the very beginning.

Project Schedule

Table 2 Original and Accomplished Project Schedule

Week	Original	Accomplished
1	• Choose Steganography Algorithm. • Begin implementation of a proof of concept software design for the encoder and decoder. • Implement simple method of transferring data and image into the DRAM.	• Steganography algorithm chosen. • Software prototype of encoder and decoder begun. • Started Implementation of UART transfer of image and message to DRAM.
2	• Finish proof of concept implementation in software. • Begin implementation of test bench for encoder and decoder. • Begin implementation of steganographic encoder and decoder in hardware. • Begin Implementation of DRAM to encoder/decoder communication.	• Software implementation of encoder and decoder done. • Implementation of image and message transfer to DRAM completed through UART. • Implementation of DRAM to encoder/decoder communication begun.
3	• Finish implementation of test bench for encoder and decoder. • Integrate Encryption/Decryption Modules with MicroBlaze. • Finish Implementation of DRAM to encoder/decoder Communication.	• Test bench for encoder and decoder done. • Encryption/Decryption modules written in software. • Started Implementation of image and message transfer to DRAM through Ethernet.
4	• Finish implementation of steganographic encoder and decoder in hardware.	• Encoder block completed in hardware. • Implementation of image and message transfer to DRAM through Ethernet complete.
5	• Implement USB transfer of cover image and message to and from PC (faster method). • Begin testing the integrated system.	• Decoder block done in hardware. • Implementation of DRAM to encoder/decoder Communication done. • System Integration begun.
6	• Finish testing the integrated system. • Attempt to extend implementation to video (if time allows).	• System Integration continued. • Integrated encoder near completed.
7	• Finish implementing video encoding and implement compression/decompression blocks (if time allows).	• Integrated encoder completed. • Integrated decoder completed. • Encryption module moved to hardware. • ECC added to fix bit errors.
8	• Demo and report preparation.	• Demo and report preparation. • Design extended to video.

Table 2 presents the original and accomplished milestones. The accomplished milestones mirror the proposed milestones quite closely. However, there were some key changes made to the project and thus, to the project schedule throughout.

The first major difference appears in the implementation of the PC to DRAM communication. Originally, it was decided that a simple solution would be used such as UART, and that a faster solution (USB) would be implemented at a later time. However, it was determined that UART was extremely slow, and since image and message to DRAM transfer is a significant portion of the project and in order to speed up project debugging at a later time, a faster communication protocol would be implemented. Ethernet was selected as opposed to USB since it provides a top speed of 1 Gbps on the Nexys Video board and this exceeds the speed of the board's USB transfer.

Second, the implementation of the decoder block was delayed by about one week. This was due to difficulties encountered with the encoder. Primarily, the difficulties experienced in locating a functional IDCT block. This necessitated the creation of the block which delayed the implementation of the decoder. There was also a delay in the completion of the DRAM to encoder/decoder communication logic. The difficulty of this part was underestimated in the original milestone projection and required extra time to complete. Despite these issues, system integration was begun on schedule.

Third, system integration proceeded slower than expected due to issues experienced in the integration stage. However, this was accounted for initially since the implementation of video as well as compression/decompression were identified as optional objectives.

The fourth major difference is the movement of the encryption and decryption modules to a hardware environment. This was done to alleviate MicroBlaze use and due to the fact that fast encryption/decryption modules suited for a hardware implementation were discovered. A software ECC was added to deal with unexpected bit errors. This module was optimized to run on the MicroBlaze and so did not present reduced performance.

Finally, due to the other issues experienced, the extension to video was done in the final week. Thus, a single extra objective was achieved along with the original design.

Description of the Blocks

Encoder Block

A block diagram of the encoder is presented in Figure 2.

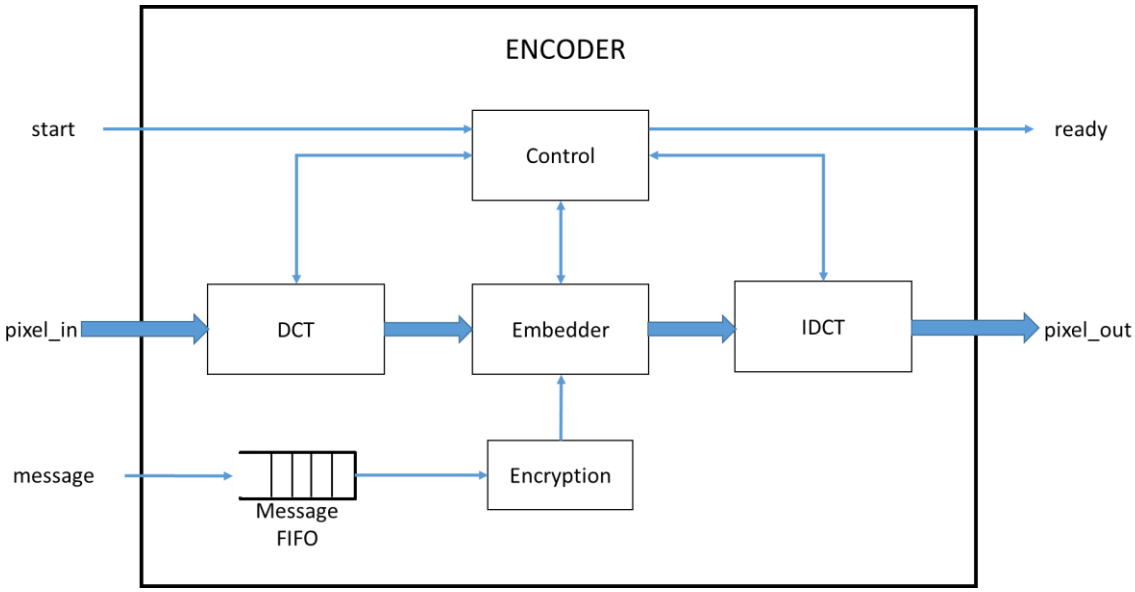


Figure 2 Encoder Top-Level Diagram

It accepts a single byte from one of the pixel channels (R, G, B, or Alpha) every clock cycle and a single message bit every 64 cycles. These inputs will be referred to as pixels from here on. One message bit is embedded for every 64 input pixels. The encoder operates on sets of 64 pixels (an 8x8 block). The DCT block computes the 2D-DCT of the input 8x8 block, accepting one element every cycle. The DCT produces a single DCT coefficient every clock cycle after filling its pipeline. The output of the DCT block is then passed to an Embedder block. The embedder block selects a particular middle frequency DCT coefficient, quantizes it, then embeds the message bit into it, and finally de-quantizes the selected coefficient and send it into the IDCT. If the DCT coefficient passed into the embedder is not the one selected for embedding, it is simply passed into the IDCT without modification. The IDCT operates on 64 DCT coefficients (8x8 block in the frequency domain) in a similar manner to the DCT, accepting a new coefficient every cycle. It then outputs a single pixel every cycle. The message is accepted and sent through a message FIFO. The message FIFO is required since the message bit is meant to be used with the DCT coefficients of the pixels being passed into the encoder. However, these coefficients will not be produced for a time period equal to the latency of the DCT, and so the

message bits must be buffered in the FIFO. Prior to being used, the message bits are encrypted using a stream cipher.

The encoder ports are described in Table 3.

Table 3 Encoder Signal Description

Signal Name	I/O	Description
start	I	Input pixel valid
pixel_in	I	Input 8-bit pixel channel (one of R, G, B or Alpha)
message	I	Message to be encoded. Must be held for 64 cycles.
ready	O	Output pixel valid
pixel_out	O	8-bit output pixel

Discrete Cosine Transform (DCT) Block

The DCT was obtained from OpenCores. The core can be found at http://opencores.org/project,mdct. It was created originally by Michal Krepa then updated by Emrah Yuce. A data sheet can be found packaged on the Git repository (https://github.com/tnsrb93/G1_RealTimeDCTSteganography).

Figure 3 presents an I/O diagram of the DCT block. The *odv1* and *dcto1* ports are debug ports and are not used in the encoder design. Table 4 lists the ports used in the DCT block (except for the clock and reset signals). In the encoder design, the *idv* input is connected to the start input of the encoder. The *odv* output is used by the control block to drive the embedder and IDCT blocks appropriately.

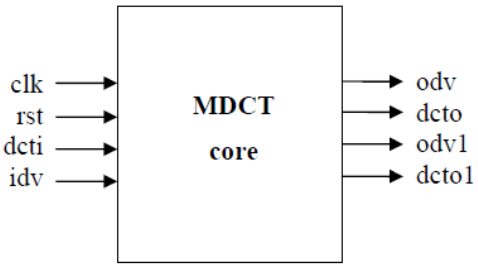


Figure 3 DCT I/O Diagram

Table 4 DCT Signal Description

Signal Name	I/O	Description
dcti	I	8-bit input data
idv	I	Input data valid
odv	O	Output data valid
dcto	O	12-bit output DCT coefficient

Embedder Block

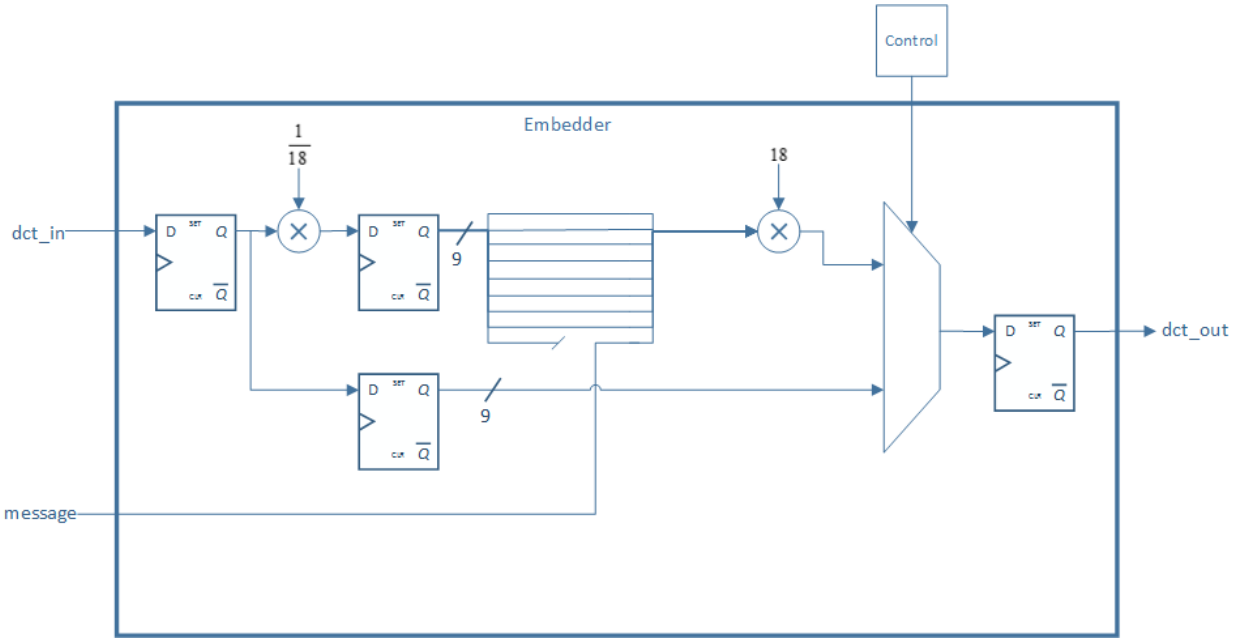


Figure 4 Embedder Block Diagram

Figure 4 presents a block diagram of the embedder. The embedder operates by first quantizing the input DCT coefficient. This is done by dividing the input by 18. In hardware, the input is multiplied by 1/18 instead. Fixed point precision is used at the same level as the IDCT (described in the next section). 18 is the JPEG quantization table coefficient for the middle frequency component chosen for embedding the message into. Quantization is done to reduce the errors that will be incurred through the noise in the DCT block as well as the noise incurred due to rounding after the IDCT stage (outputs must be integers to represent pixels). The message is then set as the LSB of the quantized coefficient. Subsequently, the coefficient is de-quantized. This is achieved by multiplying the embedded coefficient by 18. The original non-modified coefficient is also passed through the embedder stages. The control block then selects either the non-modified coefficient or the modified version of the coefficient depending on weather the coefficient presented is the correct one to embed the message into. Table 5 describes the ports used in the embedder block.

Table 5 Embedder Signal Description

Signal Name	I/O	Description
dct_in	I	12-bit input DCT data
message	I	Message bit to be embedded
dct_out	O	12-bit output DCT data

Inverse DCT Block

The 2D-DCT and 2D-IDCT equations are the following:

$$H(u,v) = \frac{2}{\sqrt{MN}} C(u)C(v) \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} h(x,y) \cos\left[\frac{(2x+1)u\pi}{2M}\right] \cos\left[\frac{(2y+1)v\pi}{2N}\right]$$

$$h(x,y) = \frac{2}{\sqrt{MN}} \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} C(u)C(v)H(u,v) \cos\left[\frac{(2x+1)u\pi}{2M}\right] \cos\left[\frac{(2y+1)v\pi}{2N}\right]$$

$$C(\gamma) = \begin{cases} \frac{1}{\sqrt{2}} for\ \gamma = 0 \\ 1 for\ \gamma > 0 \end{cases}$$

The second equation is that of the 2D-IDCT. However, the 8x8 2D-IDCT operation can be implemented differently. This can be done by implementing the 1D-IDCT on the rows of the incoming 8x8 data block. The 1D-IDCT will produce 8 outputs for each input row and so this produces an 8x8 intermediate block of coefficients. The ID-IDCT is then applied to the columns (not rows) of the intermediate block. The 1D-IDCT then produces 8 output pixels for each column, and as there are 8 columns, 64 output pixels will result. Thus, 2 1D-IDCT operations can be cascaded to implement the 2D-IDCT.

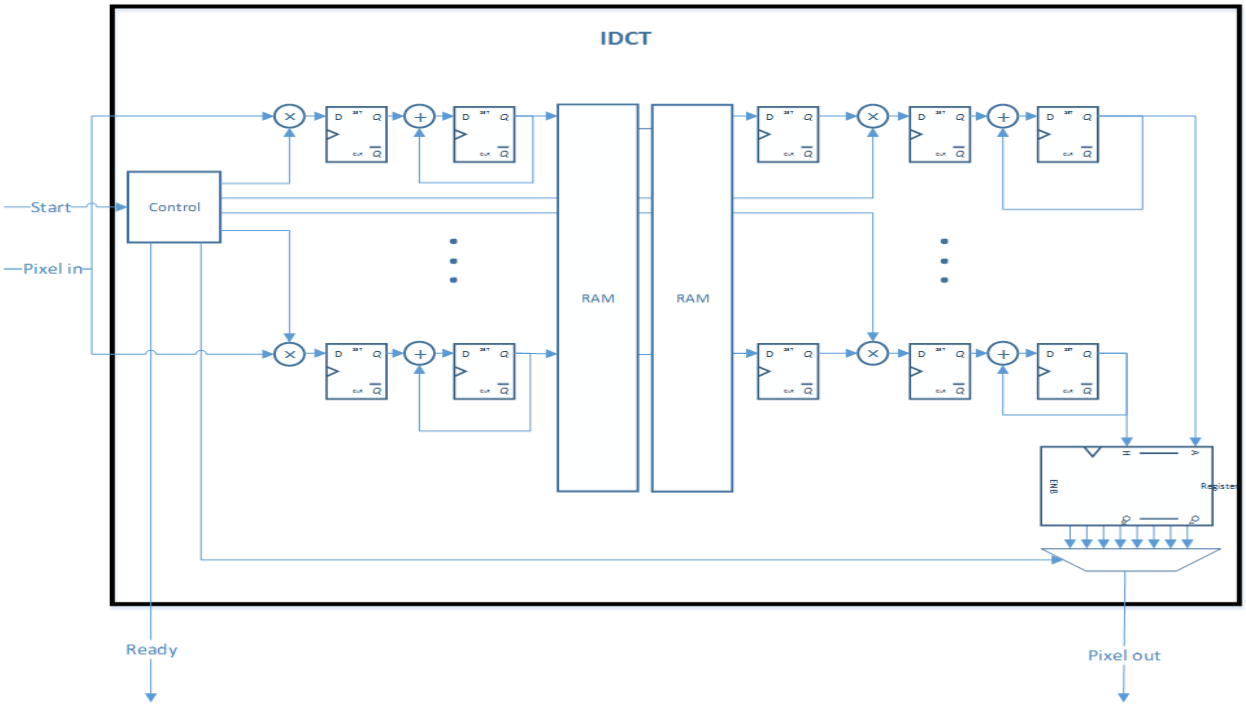


Figure 5 IDCT Block Diagram

Figure 5 presents a block diagram of the IDCT core. The core implements the 2D-IDCT. The design uses two cascaded 1D-IDCT operations on each side of the RAM blocks. There are

eight paths from the input to the first RAM block (two shown). Each path computes a single 1D-IDCT output. The paths consists of a multiplier and adder in series, as each 1D-IDCT output is produced by first multiplying the input and then adding it to the preceding sum.

The cosine and square root terms in the 2D-IDCT formula presented serve as the second inputs to the multiplier block. These are stored in memory using fixed point representation. This involves scaling the coefficients to maintain a specified precision. Multiplying the coefficients by 2^{20}, gives a precision that is a little higher than 4 decimal places, and was found not to produce errors exceeding 0.5 from the true floating point IDCT, and so this was the default precision used. The control block generates the correct multiplier coefficient at each cycle to feed each 1D-IDCT output path.

A new input DCT coefficient is accepted every cycle. After 8 coefficients are processed, the first 8 intermediate 1D-IDCT outputs are ready. These are then buffered in the first RAM block. The process is continued until the entire 8x8 block is processed. Once the 8x8 block of intermediate 1D-IDCT outputs is ready, it is transferred to the second RAM block. The 1D-IDCT is then applied again to the columns of the 8x8 intermediate block. Thus, the first RAM block is accessed in row order while the second RAM block is accessed in column order. At the output of the second 1D-IDCT stage, 8 output pixels are produced every 8 cycles. In order to be more consistent with the DCT block and maintain a true stream mode, the 8 output pixels are registered in a register file. A multiplexer then releases one output every cycle. The two 1D-IDCT stages are pipelined and operate independently from one another; thus, this design is able to produce one output pixel every clock cycle after its pipeline is full.

The multiplier blocks in the design are mapped to DSP48 blocks on the FPGA to improve performance. The maximum operating frequency of this design is 103 MHZ as indicated by Vivado's timing report.

Table 6 describes the ports used in the IDCT design.

Table 6 IDCT Signal Description

Signal Name	I/O	Description
start	I	Input valid
dct_in	I	12-bit input DCT coefficient
ready	O	Output pixel valid
pixel_out	O	8-bit output pixel

Encryption Block

Encryption is applied to the message prior to being encoded into the image pixels. Figure 6 demonstrates the encryption core used to protect the message. The Encryption core uses three LFSRs as pseudorandom generators. The output of these LFSRs are XORed together. The final output is then XORed with the message. The same module is then used in the decoder side, XORing the output of the module with the encrypted message recovers the original message bit. The LFSRs are updated after every read of the encryption output bit.

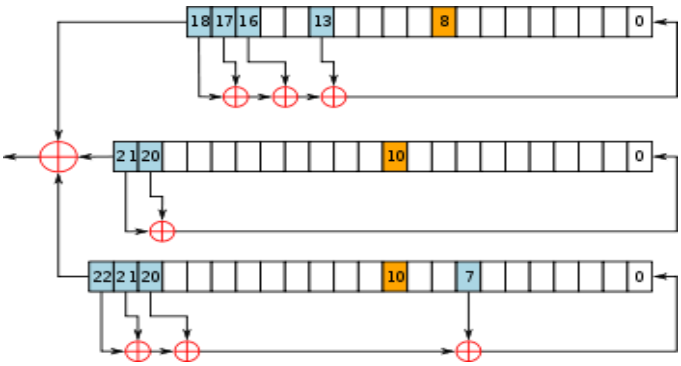


Figure 6 Encrpytion Block Diagram

Decoder Block

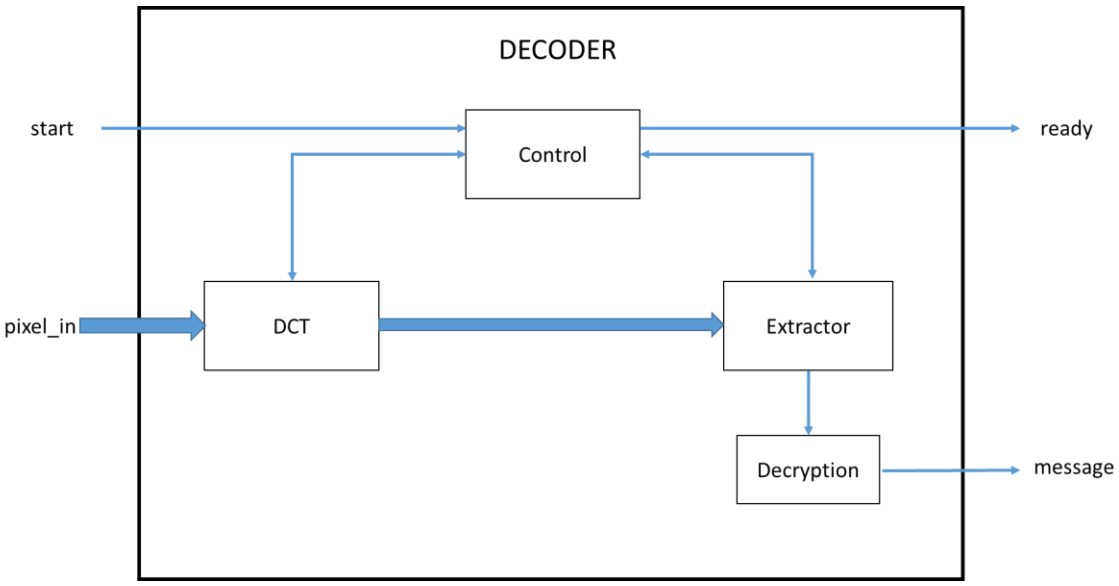


Figure 7 Decoder Top-Level Diagram

Figure 7 depicts a block diagram of the decoder. The decoder design is similar to the encoder except that the decoder lacks an IDCT since the message is recovered in the frequency domain. The input pixel is passed into the DCT block. The extractor module then recovers the message by quantizing the input coefficients, and extracting the LSB. The extracted message is then passed through the decryption module. The decryption module operates identically to the

encryption module. The *ready* signal is asserted when a valid message is observed. Table 7 describes the ports used in the decoder. Since, the decoder produces one valid message bit every 64 cycles (message is embedded into one DCT coefficient), the *ready* signal is asserted once every 64 cycles if the decoder is operated continuously.

Table 7 Decoder Signal Description

Signal Name	I/O	Description
start	I	Input valid.
pixel_in	I	Input 8-bit pixel channel (one of R, G, B or Alpha).
ready	O	Message valid.
message	O	Output message bit.

Extractor Block

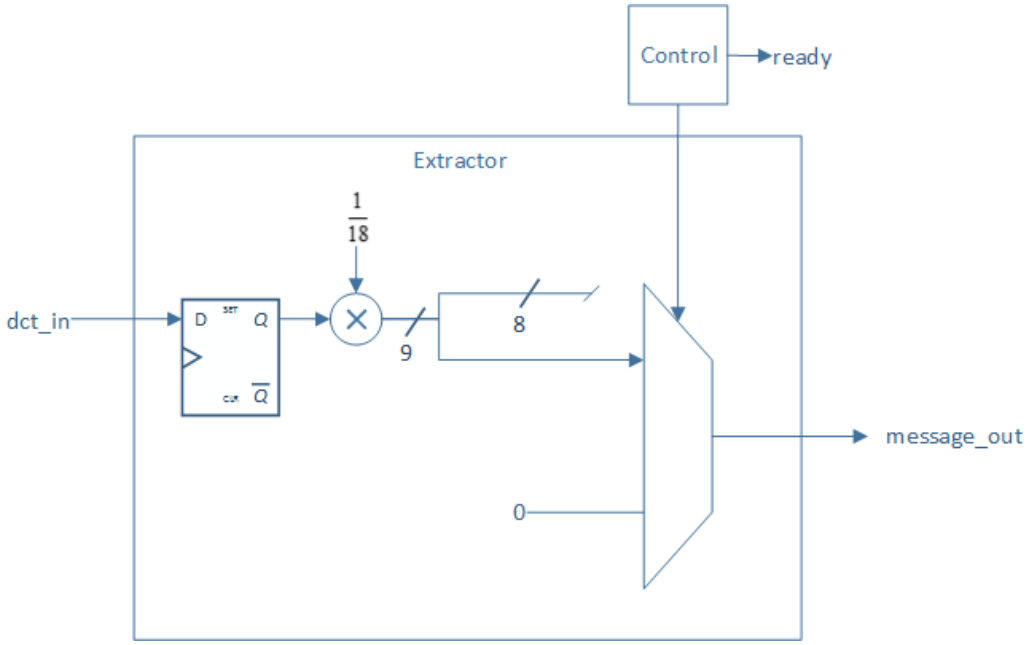


Figure 8 Extractor Block Diagram

The extractor block operates in a similar manner to the embedder block. Figure 8 presents a block diagram of the extractor. The extractor quantizes the input DCT coefficient multiplying it by 1/18. If the DCT coefficients have not changed in magnitude by more than 9 units with the noise, then this will produce the same quantized coefficient as in the embedder. The control block then chooses either the LSB of the quantized coefficient or 0 depending on weather this is the correct coefficient that the message bit is encoded into. The control block similarly asserts

the *ready* signal if the correct coefficient is observed. Table 8 describes the ports used in the extractor.

Table 8 Extractor Signal Description

Signal Name	I/O	Description
dct_in	I	12-bit input DCT data.
message	O	Message bit extracted.

Data Processing Block Interface

Shown in Figure 9 is a slightly simplified block diagram of the interface, showing only the interface to the encoder block. The decoder interface is very similar except that there is only one input FIFO (stego image FIFO) and one output FIFO (decoded message FIFO). Its message FSM will instead be on the output side of the decoder, accepting 4 bits of message data every 64 cycles and collecting them into 32-bit words.

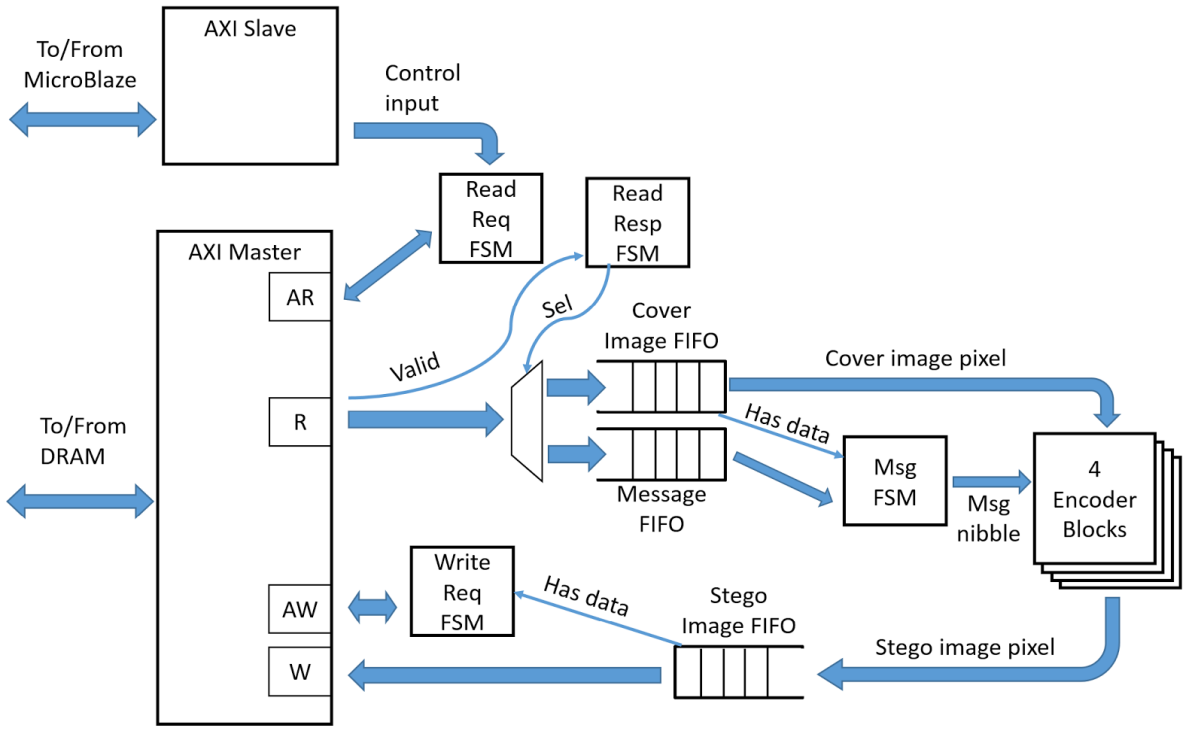


Figure 9 Top-Level Diagram of Interface between DRAM and Encoder

Here, it can be seen on the right 4 instantiations of the encoder IP, making good use of hardware parallelizability. Each encoder block is used to encode a single channel of the image (R, G, B, or Alpha). The AXI-lite slave allows the MicroBlaze to send the *begin_encoding* pulse as well as configure the cover image, stego image, and message base addresses within the DRAM. It also allows the status of the encoding process to be read. The AXI master allows communication to the DRAM, where bursts of data are read or written. There exists three FIFOs to handle incoming and outgoing data and allow flow control. Finally, there are 4 FSMs which control all components of the block once a *begin_encoding* pulse is received. Each of these components are covered in detail with respect to their exact functionality and implementation decisions.

FIFOs

The FIFOs were implemented using Xilinx's FIFO generator v12.0 IP. They were configured with the independent clock block RAM implementation, which allowed non-symmetric aspect ratio. This was extremely useful during the bring-up of the design as the incoming data was 32-bit width but only 8-bits were used at a time – only 1 encoder was instantiated initially. Also, the data count was enabled with 3-bit widths. This allowed the design to check if the fill-level is $1/8^{th}$ or $1/4^{th}$ etc. Since the FIFO depth was chosen as an even multiple of the burst size and pessimistic estimates were made for in-flight data, the reduced data width was sufficient for ensuring that the FIFO does not overflow.

Choosing the FIFO depth properly enabled the use of the fill level on the output FIFO to determine if there is enough data for one burst of write transactions. This ensured that the output FIFO did not underflow without the need to explicitly check the empty signal. This use of reduced data count (which is always pessimistic) instead of the empty signal also allowed higher performance by reducing the reliance on the availability of an exact empty signal. Similarly, by pessimistically allocating a larger FIFO depth and assuming many in-flight data bursts, the exact full signal was not required, improving the performance even further. Note that the cost to resources due to over-allocation of FIFO memory is minimal, since BRAMs on the Artix 7 FPGAs are either 18kb or 36kb, and the FIFO was configured such that this size boundary is not exceeded. Finally, the asynchronous clocks for write and read allowed the encoder block to operate at a different frequency than the AXI interface, helping to isolate and resolve issues that

may be caused by AXI interconnect contention (by slowing the encoder block resulting in lower read/write request rate).

FSMs

The simplest FSM of all, the read response FSM, simply directs the incoming data into one of the two input FIFOs. This is achieved by counting the number of bursts of data and triggering transitions based on predefined parameters. This is illustrated in the Figure 10. Initially, the FSM is in IDLE state, waiting for a *begin_encoding* pulse. Then, it transitions into WAIT_MSG state. This state asserts control flags indicating that the incoming data is message data. For reasons explained in the read request FSM section, the first two bursts are message data. Thus, the WAIT_MSG state checks if the burst is the first set of messages. Then, the FSM alternates between WAIT_MSG and WAIT_CIMG states - when a pre-specified number of cover image data bursts are received, more message data is required and thus the FSM transitions to WAIT_MSG, and once that is available, the FSM receives more image data. This process ends once the entire cover image is received, and at this point, the FSM waits for more encoding requests.

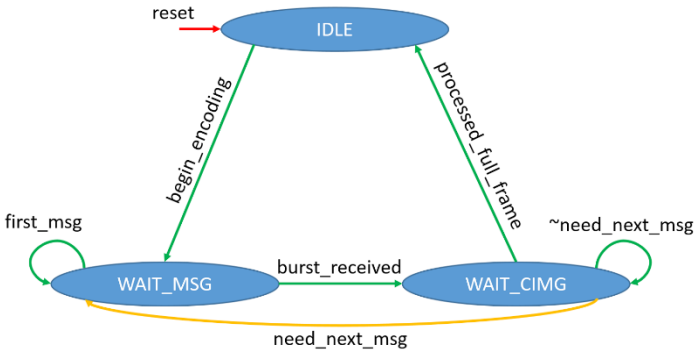


Figure 10 Read Response FSM State Transitions

The read request FSM interfaces with the AXI module – it drives the read address (*raddr*) and read address valid (*arvalid*) signals. It receives the base read address information through AXI-lite/MicroBlaze, and it also receives a *begin_encoding* pulse from the MicroBlaze. Once the begin pulse is received, it first requests two bursts of message data, ensuring that there is always message data to embed to the image. It then requests 8 bursts of 8 pixels to create the 8x8 block of pixels required for the encoder. Figure 11 consists of two essentially identical main

components - message request logic and cover image request logic. This makes sense as the control signals for requesting data from the DRAM is identical except for the read address.

The important part of the FSM is the handling of delays between consecutive bursts. First, if there is already an active burst read, the FSM must wait for its completion before a request as per the AXI4 specifications. This is achieved by the WAIT_GET_* states. On the transition to GET_*_DELAY states, the FSM asserts the *arvalid* signal along with the correct *raddr*. Additionally, either the *last_msg_read* or *last_cimg_read* registers (which feed the *raddr*) are incremented by the burst size to ensure the next read will read new data. Also, during the

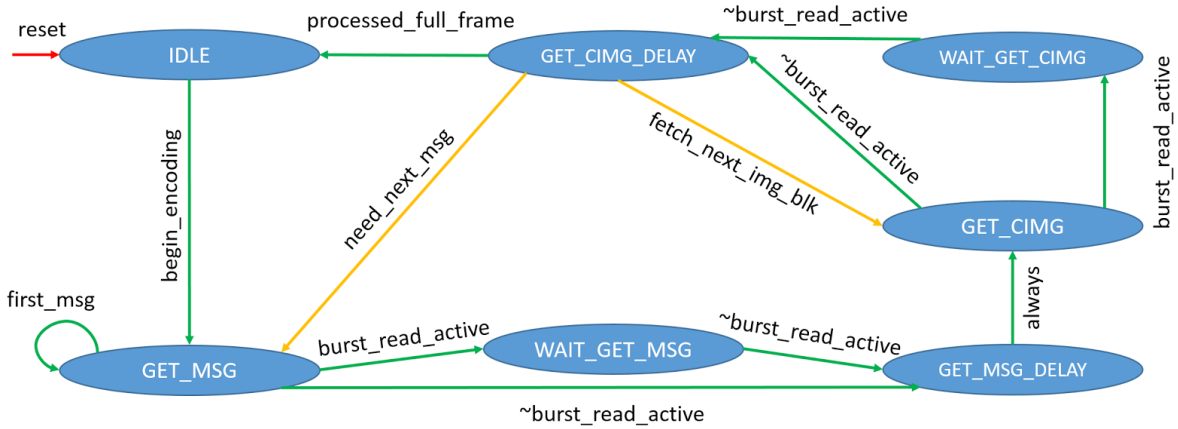


Figure 11 Read Request FSM State Transitions

cycle immediately succeeding the assertion of *arvalid*, the AXI master does not yet indicate that a new read request has come in. Thus, one cycle of delay must be added to avoid back-to-back requests.

As with the response FSM, once enough image is requested to consume *one* burst worth of message data, more message data is requested. Finally, the GET_CIMG_DELAY state determines if all of the cover image is received and decides if more are needed or if the full frame is processed.

Similarly, the write request FSM interfaces with the AXI module – it drives the write address (*waddr*), address write valid (*awvalid*), and write ready (*wready*) signals. Whenever the stego image FIFO has enough data for a burst, it initiates a write burst and sends out the data. Again, this module receives the write address from the MicroBlaze. Although the burst active logic is similar to the read request FSM, there are several differences as can be seen from Figure 12. One difference is the separation of WRITE into START_WRITE and CONT_WRITE states. The START_WRITE state asserts initial write data valid (*wvalid*) with the outgoing data. Then, the CONT_WRITE state sends the next transaction each time the *wnext* signal is asserted. Once the number of transactions sent is equal to the burst size, then the burst is complete and the FSM is ready for another burst. Another difference is the existence of the GET_BADDR state. For each *begin_encoding* pulse, the associated write address is saved internally. Since there may be long latencies between the *begin_encoding* pulse and the arrival of output data, saving the write address internally allows the FSM to handle consecutive encoding requests properly without overwriting the write addresses associated with each request.

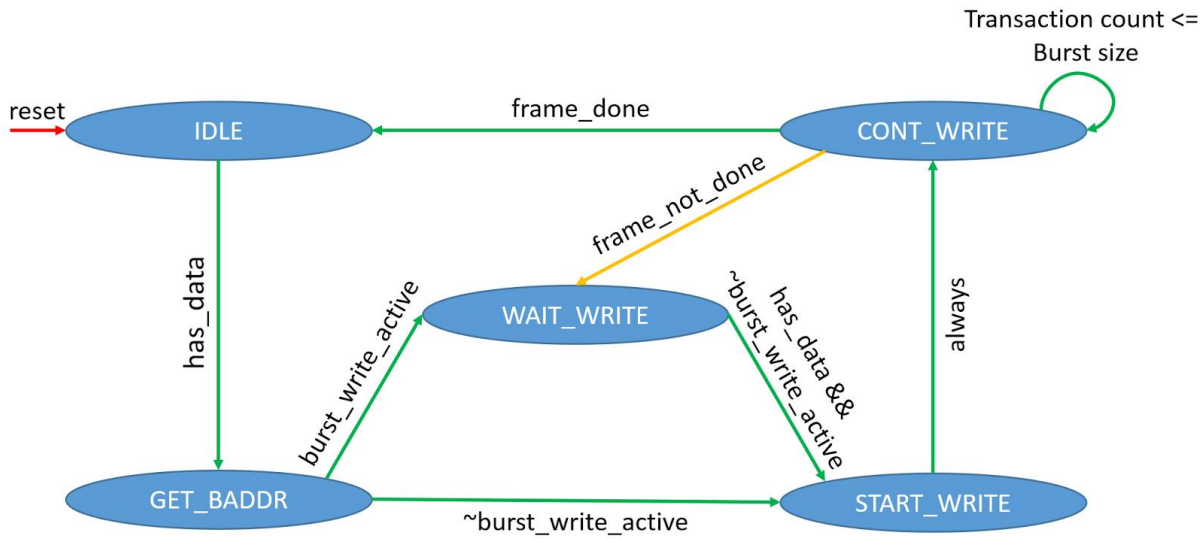


Figure 12 Write Request FSM State Transitions

The message FSM allows the proper control of the encoder block. As the encoder block accepts 1 message bit every 64 cycles, the message FSM properly handles the switching of message bit. Some complexity is added by the fact that the message from the FIFO is 32 bits

each, requiring the correct bit selection over time and switching at 32-bit boundaries of messages.

AXI-lite Slave

The slave interface is a simple modification of the example AXI-lite slave design, similar to the warm-up. Table 9 shows the encoder IP's registers and their purposes. The address registers are self-explanatory - user can configure the DRAM addresses using these registers. The *begin_encoding* register is a write-only register. Writing anything to this register generates a *begin_encoding* pulse to be sent to the encoder IP. Finally, the *done_encoding_cnt* accepts *done_encoding* pulse from the encoder IP and counts up.

Table 9 Encoder IP AXI-lite Slave Registers

Address Space Offset	Register Name	Access Type	Default Value	Description
0x00	cover_img_addr	R/W	0x0	Base address in the DRAM to read the cover image
0x04	stego_img_addr	R/W	0x0	Base address in the DRAM to write the processed stego image
0x08	msg_addr	R/W	0x0	Base address in the DRAM to read the message to embed
0x0C	begin_encoding	WO	0x0	Begin encoding register
0x10	done_encoding_cnt	RO	0x0	Counter indicating the number of times done_encoding was asserted

Table 10 shows the decoder IP's registers and their purposes. These registers are very similar to those of the encoder IP. The main difference is that the stego image address is where the decoder reads the image instead of writing it. Also, the message address is where the message extracted from the stego image is written.

Table 10 Decoder IP AXI-lite Slave Registers

Address Space Offset	Register Name	Access Type	Default Value	Description
0x00	Unused			
0x04	stego_img_addr	R/W	0x0	Base address in the DRAM to read the processed stego image
0x08	msg_addr	R/W	0x0	Base address in the DRAM to write the extracted message
0x0C	begin_decoding	WO	0x0	Begin decoding register
0x10	done_decoding_cnt	RO	0x0	Counter indicating the number of times done_decoding was asserted

AXI Master

The master interface is a modification of the example AXI master design. The data generation and checker circuits were deleted. Also, the constant parameter base read/write address definition was changed to be based on either the write address input or the read address input. Overall, the data flow of the AXI master was rerouted to allow access to incoming data and accept the outgoing data as an input. This is illustrated with the port diagrams where Figure 13 is the original AXI master and Figure 14 is the final AXI master. Note that other than the *M_AXI* bus, the input ports are shown on the left of the diagram and the output ports are shown on the right of the diagram.

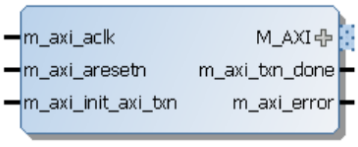


Figure 13 Original AXI Master I/O Diagram

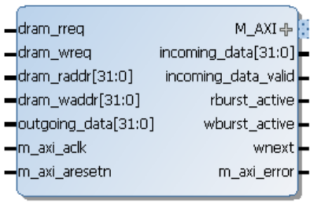


Figure 14 Final AXI Master I/O Diagram

Although slightly simplified, the *dram_raddr* connects directly to the *M_AXI_RADDR*, the *dram_waddr* connects to the *M_AXI_WADDR*, and the *outgoing_data* connects to the *M_AXI_WDATA*. On the output port side, the *incoming_data*, *incoming_data_valid*, and *wnext* signals are loopbacks of *M_AXI_RDATA*, *M_AXI_RVALID*, and *M_AXI_WREADY* inputs respectively. Finally, the *rburst_active* and *wburst_active* signals indicate the internal progress of burst transaction stage.

Encoder/Decoder IP Simulation

Initial Simulation

The IDCT block was first verified by driving it with random data. The output of each 1D-IDCT stage was then verified by comparing it with a software realization of the IP. The output was verified for two sets of different 8x8 blocks to ensure that the IDCT does not experience issues at the 8x8 boundary. Since, the IDCT operates on streams of 8x8 blocks, this process is sufficient to confirm its functionality.

The encoder was verified in a similar manner to the IDCT block. The encoder was then connected to the decoder block in simulation. A testbench was set up to verify that the input message stream is identical to the output message stream. The testbench was run for 1 ms of simulation time and was found to produce no decoding errors. It is however known, that decoding errors are produced when using real images; however, as the simulation suggests they are sparse and easily correctable.

Using BRAM for Simulation with Interface

A BRAM was instantiated using Xilinx's AXI BRAM Controller v4.0 IP. It was configured to use AXI4 protocol interface and the memory included internally. This allowed the BRAM to be used in place of DRAM during the simulation. By interfacing the custom FSM and AXI master with this the AXI slave of the BRAM, the functionality of the custom design read/write requests were verified. Specifically, the exact handshaking process with a correct IP (the BRAM) could be observed, instead of the initial naive approach of manually driving the AXI master input signals in simulation. This use of BRAM was necessary as MIG/DRAM initialization process was excessively long in simulation so that it was completely unpractical to run multiple times.

To confirm that the integration of the encoder block into the interface was successful, a sample set of steganography inputs and results were first collected from the encoder-only system. The input data was placed into the BRAM memory using .coe file initialization of the memory. This was also useful in checking that the correct data was being transferred after the handshaking occurred. Then, the data was sent to the encoder block using the block design described earlier (FIFOs, FSMs, etc) and its output was written back to the BRAM at a different location. By

confirming that the result written into the BRAM matched the encoder-only collected results, correct integration could be verified.

Later on, once the decoder block was verified, another approach of simulation was devised. The output of the encoder, written to the BRAM, was read back and processed through the decoder block. By checking that the output of the decoder is the initially embedded message, the encoder, decoder, and the interface could all be verified.

To simulate potential gaps in the input to the encoder block, the encoder clock was modified to be much faster than that of the decoder clock. This meant that arrived data would be consumed much quicker than the arrival rate.

Simple Video Implementation using VDMA

To demonstrate a real-time use of the encoder IP, it was decided that a simple video stream will be implemented. The work was done largely based off of Digilent's HDMI example project. On top of the existing VDMA, another VDMA was added. As the existing VDMA wrote the incoming video to a frame buffer in DRAM and read back another (not necessarily different) frame buffer to output video, the new VDMA allowed access to the video data in the frame buffers.

The new VDMA was configured with 3 frame buffers to match the existing VDMA. Also, it was configured with 32 burst sizes and a line buffer depth of 4096. Without doing this, the performance is not high enough due to frequent back-pressures in the data flow. The read/write channels were configured in the genlock master mode as there was no intention to cycle through the buffers, especially on the write channel.

The encoder IP was wrapped with a simple AXI stream interface, with encoder start driven by *M_AXIS_MM2S_TVALID* signal and the encoder output ready driving the *S_AXIS_S2MM_TVALID* signal. Additionally, a simple single-clock FIFO was added on the output side to save outgoing data while the master is not ready to accept *S_AXIS_S2MM_TDATA*. Figure 15 shows a block diagram of the modified 'stream' encoder IP and its connections to the VDMA.

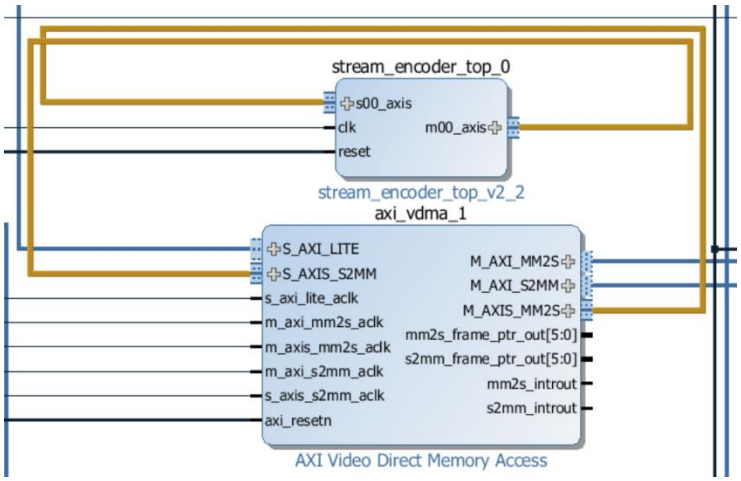


Figure 15 Stream Encoder IP and VDMA Connections

On the SDK side, the VDMA was initialized similar to how the existing VDMA was initialized. As the design was not a full-fledged video implementation, there were limitations in the communication between the stream encoder IP and the VDMA. Specifically, the vertical size of reads and writes were defined as 1, and the horizontal size and stride as 1920*3*4. This corresponds to reading or writing 4 rows of a frame of 1080p video. The software received interrupts at the end of each set of 4 rows, initiating the next read/write operation. Each time, the frame store start address was incremented to match the stride. Thus, a frame of video required 270 read/write requests - this limited the framerate of the video. For the purposes of the demo, the message data to be embedded was based on hardcoded BRAM initialization. However, in a full implementation, the AXI master and read request FSM can be added to the stream encoder to read message data to be embedded from the DRAM.

Ethernet Communication

The PC-to-DRAM transfer component of the product is realized using Ethernet. This has three main components hardware, server software and client software, as shown in Figure 16. The Ethernet software uses TCP/IP; this was the best choice because it allowed the use of some existing software and the Ethernet on the Nexys boards has limited speed anyway.

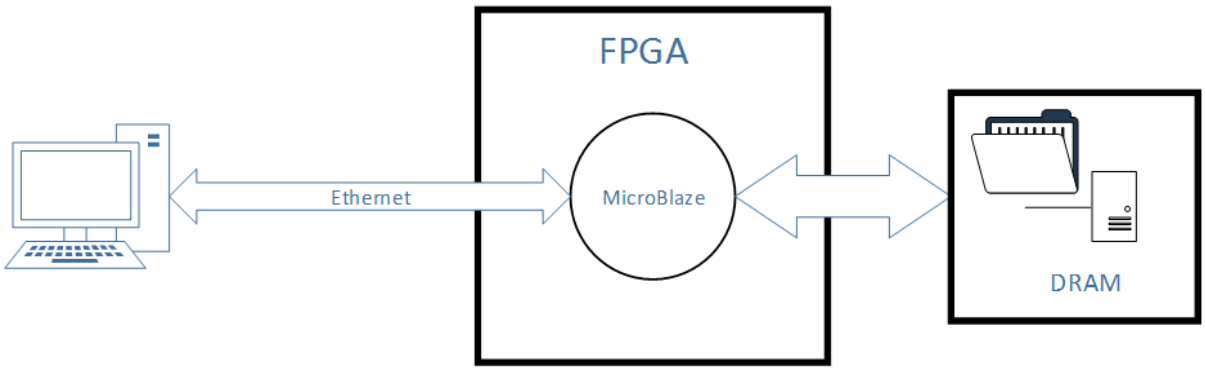


Figure 16 Ethernet Communication High-Level Diagram

Hardware

The Ethernet Hardware is made up entirely of pre-existing blocks provided by Xilinx, but it did have to be different for the Nexys Video and Nexys 4 DDR boards. For the Nexys 4 DDR, the Ethernet is primarily handed by an AXI Ethernet Lite Block; however this block has a standard MII physical Interface whereas the board has reduced MII interface. Accordingly, an Ethernet PHY MII to Reduced MII block is used to translate between these two interfaces. The Ethernet transceiver, on the Nexys Video Board has a maximum throughput of 1 Gbps as opposed to the 100 Mbps of the transceiver on the Nexys 4 DDR. The AXI Ethernet Lite block does not handle this speed so the AXI 1G/2.5G Ethernet Subsystem was used instead; no interface translation was needed.

Server Software

The Ethernet Server software— i.e. the software running on the MicroBlaze processor— is based on Xilinx's light-weight IP Echo Server, to which the functionality of a basic file system was added. This file system is stored entirely in DRAM. The files are each given a fixed size of 4 MB which is the close the size of a single 720p image with 32 bit colour depth. The images are identified with a letter of the alphabet; a limit of 26 files never presented a problem. The file

system works by checking the beginning of each Ethernet packet for a set of fixed control sequences; these were all made fairly long so that a control sequence appearing randomly in the first packet of an ordinary I file is vanishingly unlikely. If a control sequence is found the rest of the packet is discarded, otherwise the packet is written to a file. The control sequences for the file system had the following purposes:

- Open a file as an image, includes a letter to identify the file
- Open a file a message, includes a letter to identify the file
- Read

Opening a file as an image moves the cursor (a pointer) to the beginning of the specified file. The data from all subsequent packets is written to this file and the cursor is advanced to the end of each block of data. An acknowledgement is sent to the client after each packet is received (this does not significantly affect transfer speed because open file commands are infrequent). Opening a file as message has the same effect except that the data from subsequent packets will have error correcting codes added to it. Getting a Read control sequence will cause the server to send a packet with a buffer of data to the client. This buffer has a fixed size of 1400 bytes and is taken from the location of the cursor, which is then incremented by 1400. The Ethernet server also recognizes other control sequences that have nothing to do with the file system. These allow the user to read and write any of the control registers made available by the encoder and decoder IPs.

Client Software

The Ethernet client starts by opening a windows socket to connect with the server; the program proceeds to parse the command line arguments that it was given. The LodePNG open-source library is used to load PNG images. The command line arguments should each be taken from the set of commands that is elaborated below. All paths can either be relative or absolute; all location should consist of a single capital letter.

- `set_input:<path>`
    - Set the path for any PNG image file that will be sent to the server
- `set_output:<path>`
    - Set the path for any PNG image that will be received from the server

- `set_in_message:<path>`
  - Set the path for any text that will be sent to the server
- `set_out_message:<path>`
  - Set the path for any PNG image that will be received from the server
- `write_png:<location>`
  - Load a PNG image from the specified path on the local machine
  - Tell the server to open the file at location <location> as an image
  - Write the loaded PNG file to the server
- `read_png:<location>`
  - Tell the server to open the file at <location> as an image
  - Send Read commands to the server until enough data is received to fill a 720p 32-bit colour depth image
- `clear_png:<location>`
  - Tell the server to open the file at <location> as an image
  - Overwrite this entire file with zeros
- `write_txt:<location>`
  - Load a text image from the specified path on the local machine
  - Tell the server to open the file at <location> as a message
  - Write the entire text file to the server
  - Store the size of this file
- `set_msg_size:<number>`
  - Set the size of the message file
- `read_txt:<location>`
  - Tell the server to open the file at <location> as a message
  - Send Read commands to the server until the whole message is received
  - The size of the message file must be known in advance through either a previous write_txt or set_msg_size command.
- `begin_encode:<location 1><location 2><location 3>`
  - Tell the encoder block to use the file at location 1 as its message
  - Tell the encoder block to use the file at location 2 as its cover image
  - Tell the encoder block to write its stego image at location 3

    - Tell the encoder block to begin encoding
- `begin_decode:<location 1><location 2>`
    - Tell the decoder write the message it finds at location 1
    - Tell the decoder block to use the file at location 2 as its stego image
    - Tell the decoder block to begin encoding
- `check_encode`
    - Check if the encoder has finished parsing an image
- `check_decode`
    - Check if the encoder has finished parsing an image
- `wait`
    - Wait for the user to press a key before proceeding to the next argument

Error Correcting Codes

After the decoding stage was completed, it was observed that a number of bit errors were occurring in the decoded output. These bit errors were not observed in simulation as they were relatively sparse and simulation would require a large amount of time to reveal even a single error. These errors are likely due to noise emanating in the DCT block due to the use of fixed point math in combination with the need to round the ouput of the IDCT block. Bit errors due to latter were observed to be an issue with the software implementation as well but were eliminated when the decision to embed a single bit into each block was undertaken. Thus, a combination of the two issues is likely the cause. To mitigate this issue, error correcting codes were added to the design.

The error correction coding is done in a simple yet elegant manner. The message data is split into 4-bit sections and arranged in a 2x2 squares; a parity bit is computed for each row and column of this square as illustrated below.

b_0	b_1	p_0
b_2	b_3	p_1
p_2	p_3	

This allows correction of up to one bit error in every eight. If a single bit of the original message is flipped, then both its parity bits will be incorrect. Since each message bit is uniquely identified by two parity it can be corrected. If only one of the parity bits is wrong, this means that a bit flip occurred for the parity bit itself and no correction should be undertaken. Thus, this code can correct 1 bit of error in every 8 bits transmitted. Since, the errors are very sparse, this is enough to correct all bit errors in the decoding output.

Since the message is short relative to the images, it was decided that the error correcting codes would be done in software. Even so, when first integrated into the design, the error correction coding was unacceptably slow. It is likely that this occurred mostly because the functions implementing the error correcting codes involved a lot of bit shifting and were being run on a MicroBlaze processor without a barrel shifter. Regardless, the issue was solved by precomputing the 2-byte code for every possible byte and storing in in an array (as well as the byte for every possible 2-byte code). In this way, each error correction coding operation required only a memory look-up.

Design Resource Utilization

Table 11 shows the resource utilization of the design on the Nexys 4 DDR board. Note that the BRAM count is the number of 36kbit BRAMs, so 0.5 BRAM is equivalent to 1 18kbit BRAM. The heaviest resource usage (percentage wise) was on LUTs at 63% of the available resources. This indicates that the design can be improved further without resource issues.

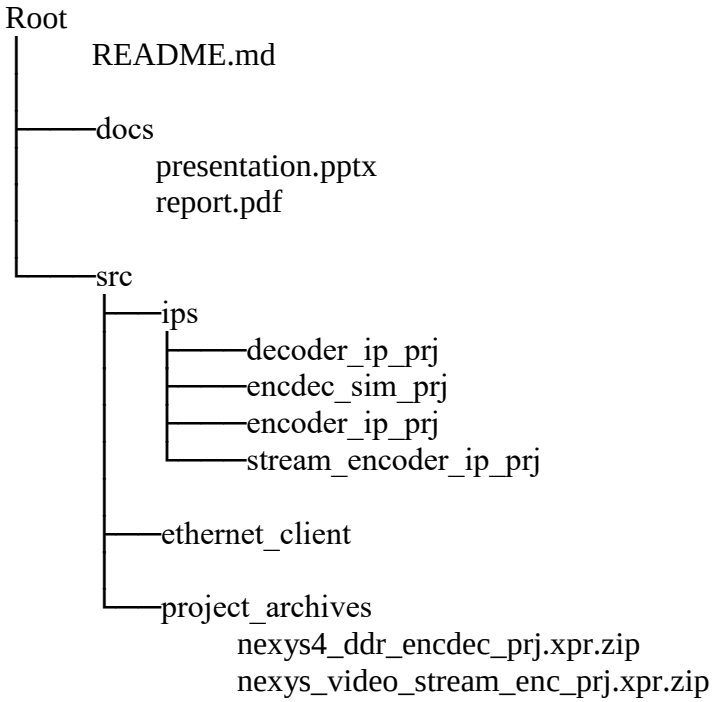
Table 11 Resource Utilization Summary

Resource	Utilization
FF	54505
LUT	40047
Memory LUT	2002
BRAM	17.5
DSP48	104

Description of Design Tree

The following is a simplified directory structure of the design repository:

Figure 17 Git Repository Directory Structure



The top-level file **README.md** describes in detail how to use the repository. For example, it gives a detailed instruction set on how to bring up the project from these archives so that the hardware functionality may be tested.

There are two files under **/docs -** this report and a set of presentation slides. The presentation gives a high-level description of the project while the report provides a much more in-depth explanation.

Under **/src**, there are three subdirectories **/ips**, **/ethernet_client**, and **/project_archives**. The project archives provide the full projects in archived form. Nexys 4 DDR archive is the main project with the ethernet client and 4 instantiations of encoders and decoders. The Nexys Video archive is the project with the stream encoder IP to demonstrate simple integration of the encoder into a video stream using two VDMAs.

The **/ethernet_client** directory contains the source code for the PC-side command-line client. The source code has already been made into the executable file also contained in the same

directory. This executable client can be run manually with command-line options or using the included .bat file. A simple example usage of the client is given in a **cmd_line.txt** file.

The **/ips** folder contains all the custom-created IPs in the project form so that they may be exported easily if modifications are made. The **/decoder_ip_prj**, **/encoder_ip_prj**, and **/stream_encoder_ip_prj** contain the respective IP projects. The **/encdec_sim_prj** contains a more thorough testbench incorporating both the encoder and decoder IPs, simulating the embedding of a message word and decoding it back from the stego data.

Conclusion

DCT Steganography was implemented in hardware. The throughput of the hardware design after accounting for the speed of the encoder and decoder IP was found to be 3.125 Mbps as opposed to the throughput of the software implementation which was found to be 2.28 kbps. The software implementation can be made to improve via various optimizations but clearly, a significant performance improvement is achieved by implementing steganography in hardware.

References

[1] A. Altaay, S. Sahib and M. Zamani, "An Introduction to Image Steganography Techniques", *2012 International Conference on Advanced Computer Science Applications and Technologies (ACSAT)*, pp. 123-124, 2012.

[2] T. Morkel, J. H. Eloff and M. S. Olivier, "An overview of image steganography", *ISSA 2005*, pp. 1-11, 2005.

[3] R. Biswas, S. Mukherjee and S. K. Bandyopadhyay, "DCT Domain Encryption in LSB Steganography," *Computational Intelligence and Communication Networks (CICN), 2013 5th International Conference on*, Mathura, 2013, pp. 405-408.

[4] W. Laces and J. Garcia-Hernandez, "FPGA implementation of a low complexity steganographic system for digital images", *2015 IEEE/ACIS 14th International Conference on Computer and Information Science (ICIS)*, 2015.