High Speed Video Compression/Decompression Pipeline

Final Report

Ariana Eisenstein, Yuan Cao

Project Objective

We have implemented a bi-directional PC to DRAM Memory pipeline that performs compression on data moving from the PC to DRAM and decompression on data moving from DRAM to PC as a proof of concept of Data Compression algorithm tests within this system. The PC transfers data to the FPGA using a SceMi connection, with an associated c++ test script to transfer the video data.

We have implemented an image compression and decompression algorithm on a frame by frame basis, as image compression is much less complex and requires less memory than video coding. For this project, we use the Discrete Wavelet Transform (DWT) based compression algorithm [1]. We want to use the DWT because of its superior performance and better compression ratio for most image data, compared to the Discrete Cosine Transform (DCT), which can give rise to blocky artifacts. The downside is that more computation power is required to perform the transform, and hence hardware acceleration is necessary to compress/decompress in real time.

In order to decrease the total number of bits to be transferred, we will be using Entropy encoding of the DWT coefficients. Entropy coding can be either lossless or lossy, depending on algorithm selection. For this application, we chose a lossless algorithm as we want to restrict our loss to thresholding on the DWT coefficients themselves. For our implementation, we have implemented the Huffman encoding algorithm.

We have a throughput of one sample per cycle. A hardware accelerated Discrete Wavelet Transform as used for compression has be explored by [2,3].

Background

Video Data can be the biggest of big data. As standard move to higher definition with larger pixel counts and real-time frame rates of 30-60 frames per second, the speed of processing video data must increase. In order to achieve the high speed data transfer of each video frame, compression algorithms are employed to decrease the amount of data needed to be transferred, allowing the high frame rates to be maintained. In compressing video data, a tradeoff is created. While less data is being transferred, a compression or decompression must be done to recreate the correct pixels. Thus, computationally complex compression and decompression need to operate at high frequency.

Increased speed of video compression and decompression is one of the focuses of the the Energy Efficient Multimedia Group (EEMS). In order to better test developed algorithms, Ariana has helped develop a system on an FPGA VC707 platform for testing of ASICs that implement energy efficient algorithms. The system currently has interfaces to a 1080p@92 FPS Camera, HDMI Display, UART Communication bus, and SD Card Reader and well as a connection to the provided DDR3. The system currently works best as a standalone test as we do not have a fast enough interface to the PC (UART is too slow, SD Card is too slow and limited in space). Currently, EEMS is experimenting with different methods of transfer to the FPGA from the PC and has developed a DRAM arbitration module that time multiplexes between three components (camera, detector, and display) with a data rate of 512 bits / cycle at 200 MHz. EEMS would like these two interfaces to communicate in order to both run the Benchmark Datasets on the physical hardware or RTL and test the implementations of these algorithms, using data gathered and processed by our system.

High-Level Design

The pipeline from PC input to DRAM is as follows: an Async Fifo, a Byte Deserializer, the Decompression modules, a Pixel Deserializer, and a FIFO. Oppositely, for the DRAM to Ethernet Pipeline: a FIFO, a Pixel Serializer, the Compression modules, and an Async Fifo. An overall block diagram of this system is shown in Figure 1. Note the separation of clock domains as shown by the dashed line.

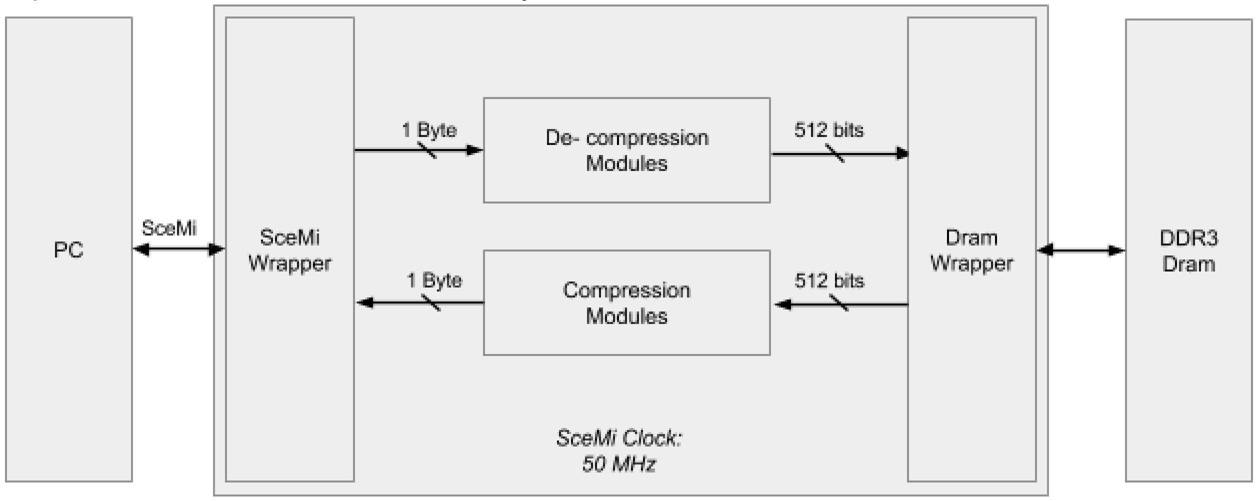


Figure 1: System Block Diagram:
This diagram shows the overall block diagram of the system. 1 Byte of data enters the system from the external SceMi modules and 512 bits exit system to DRAM.

The specific components of the Compression and Decompression modules are shown in Figure 2 and Figure 3. As shown in the figures, we will be doing three compressions / decompressions in parallel, one for each color channel RGB. These can be done in series using a super-folded architecture depending on size and timing results. The compression is done by first taking the Discrete Wavelet Transform (DWT) of each frameof pixels to get the DWT coefficients. We will then use thresholding to keep all coefficients above a desired value. These remaining coefficients will be compressed and stored using Huffman compression. The decompression works in the opposite manner, recreating the coefficients from the Huffman table and then restructuring the pixels with the Inverse DWT.

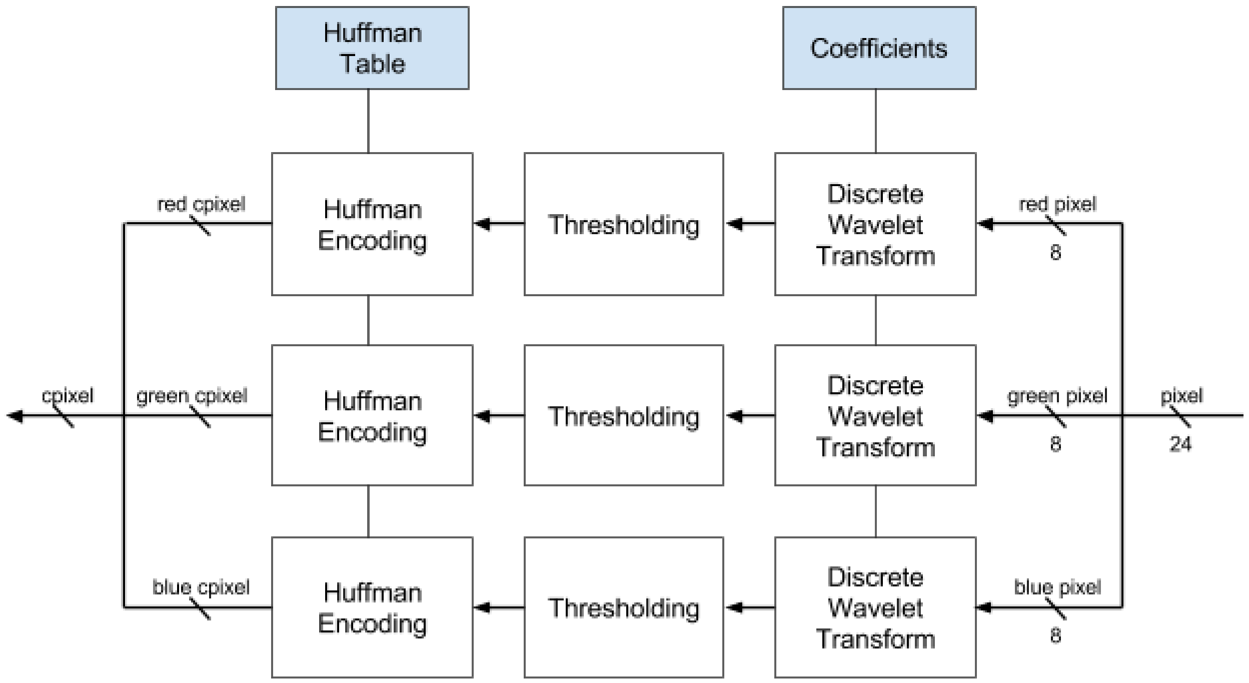


Figure 2: Compression Block Diagram:
This diagram shows a detail of the modules used in the compression. A Discrete Wavelet Transform, a Threshold, and Huffman Encoding are the modules used for compression.

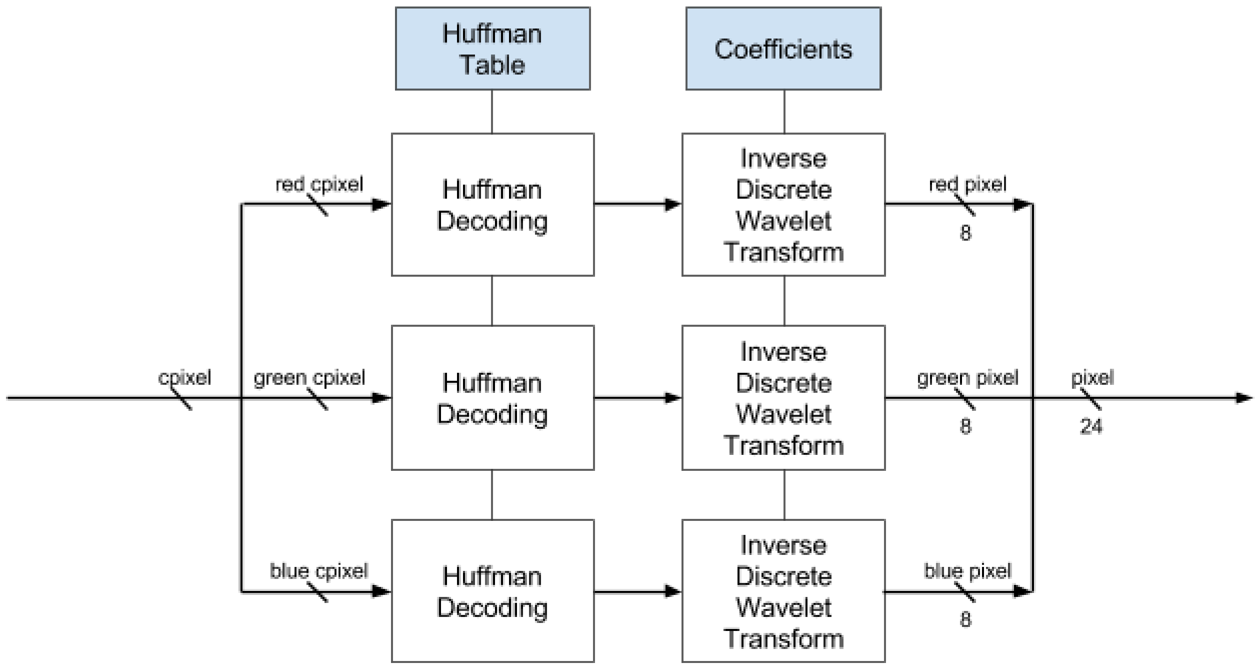


Figure 3: Decompression Block Diagram:
This diagram shows a detail of the modules used in the decompression. A Huffman Decoding and Inverse Discrete Wavelet Transform are the modules used for compression. Three decompressions will be done in parallel for each color channel, in accordance with our software model.

Test Plan

Our plan for testing the system is as follows. Each individual module will be tested using a Bluesim testbench to ensure its functionality. For the serializers and deserializers, this is simply checking that the input data is the same as the output data, either serialized or deserialized with a final test that give identical input and output when the two are connected together. For the compression and decompression, we will be using the software implementation of either the compression or decompression algorithm to generate test data for both modules. While floating to fixed point errors can cause slight differences between the two implementations, a similar result will suffice. Initially, we will test the algorithms using only gray scale data. The final test for the compression and decompression modules will be sending an image through the compression, sending the compressed image through the decompression and viewing the result. Data throughput and delay will be characterized on the FPGA platform using a SceMi interface. If the results don't meet the timing specs, we will add additional pipelining to the algorithms as well as add a second frame of delay if necessary. If the results don't meet the space specifications, we will create a more folded design that reuses the same hardware for several tasks. If the results do not meet our desired performance, we will return to tuning the algorithm parameters in software, and then modifying our hardware accordingly.

Microarchitectural Description

Discrete Wavelet Transform module

In this module, we take in a two-dimensional array of sample points (one color component) and perform DWT using a given set of parameters. We choose CDF97 filter which is standard in JPEG2000 standard and proven to give good results on a variety of images.

The transformation on a 1-D array is described by the following illustration, which is also known as “Lifting scheme”.

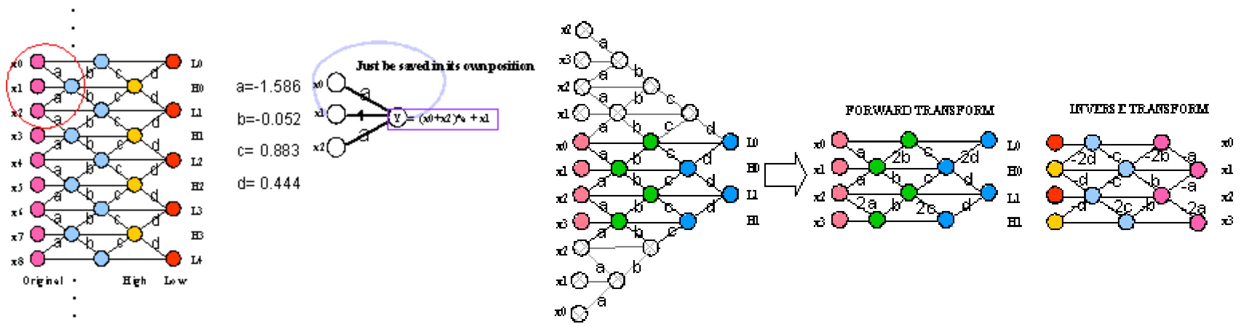


Figure 4

Left: Performing DWT on an 1-D array using lifting scheme. The given coefficients are for CDF97 biorthogonal wavelet. Right: For input of finite size, symmetric boundary extension demands that some of the coefficients must be doubled in order to be mathematically reversible.

The basic operations required in implementing this 1-D DWT algorithm is calculating the vector multiplication/addition

$$y[i]=x_0[i]+a*(x_1[i]+x_2[i]),$$

where y[i], $x_j[i]$ and a are all fixed-point numbers. In order to reach the throughput goal, we must transform multiple samples in each cycle. However, full parallelism is impossible since it implies an array of multiplier/adders with size comparable to the line width (>1000) this will put a large demand on required hardware resources. Therefore we use a serialized structure; chunks of size B is fed into the pipeline in each cycle, where B is 2~16 defines the required multiplier/adder array size.

Transformation on the second dimension is independent from the first dimension. The resources required to perform DWT on the entire image in parallel is unrealistic (~1 million multiplier), therefore we perform the second dimension on a line-by-line basis. To avoid storing the entire image in a large buffer, the so-called “Line-based lifting” technique can be used.

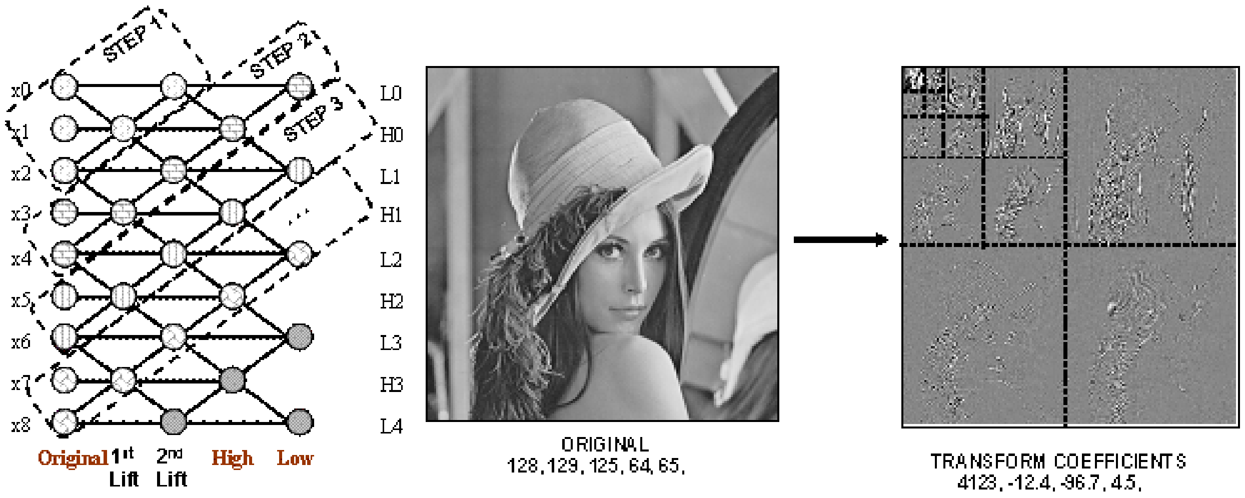


Figure 5

Left:Illustration of line-based lifting technique. Right: Example of image before DWT transformation and after performing a 4-level 2-D DWT transformation.

At initialization, three lines are read in and transformed. At this point no results are obtained. After this, every two lines are transformed, and two new lines of results from previous lines are obtained. This process is repeated until every line is transformed. In the actual implementation, we fully decouple between different stages of the pipeline using large FIFOs, so it does not need to explicitly define this behavior in the hardware.

Usually, one step of DWT transform is not enough, since the upper-left corner of the transformed image is a low-pass filtered version of the original image and the compression ratio might be still low. Therefore, 2-D DWT of smaller and smaller sizes are usually performed repeatedly on the upper-left corner of the image. In our implementation several DWT modules of exponentially smaller sizes are cascaded to provide this functionality.

The inverse transform is pretty much the same as the forward transform module, except that the order that the coefficients are applied onto the data is reversed. All of the hardware described above are simply copied and modified. In hardware the effective area is basically doubled.

Huffman Compression Module

This module takes the DWT coefficients produced by the DWT module and encodes them in less bits using a the Huffman Compression scheme. Huffman operates by taking the most prevalent coefficients and stores them in smaller bit representations, resulting in a total decrease of the bytes to store the data.

This module will access and maintain an Encoding table that will maintain the mapping from filter coefficient to encoded value. This table will maintain a static encoding based on the simulated performance of our DWT algorithm on test images.

The initial Encoding table simply encodes the integer values of the coefficients. The JPEG Standard operates using integers, so the fraction portion of the FixedPoint coefficients is thrown away. This initially compression scheme works well as the largest numbers of coefficients are primarily centered around zero and as such can be represented in few bits. Figure 7 shows a histogram of the DWT integer coefficients of a test image created using our reference code.

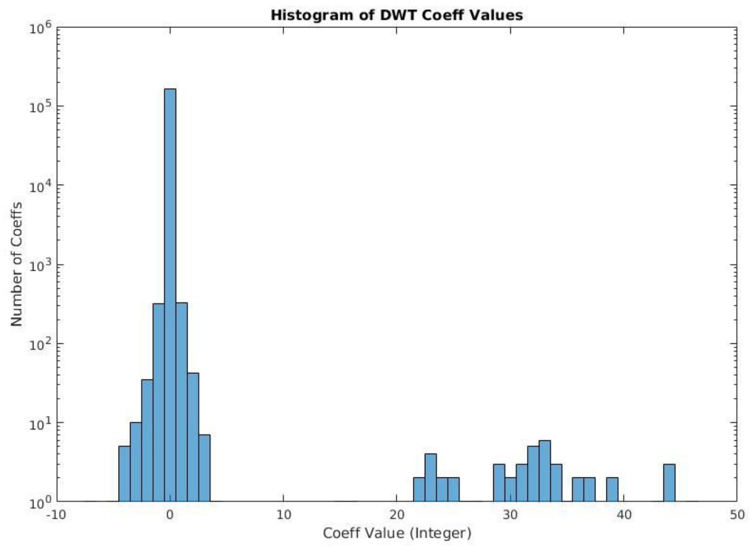


Figure 7: Histogram of DWT Integer Coefficients:
This histogram shows the distribution of DWT integer coefficients in a test image. Notice the quantities is a log scale. We have significantly more zero coefficients, our smallest encoded word.

Additionally, from this table we can see that the majority of our components are between [-4,3]. From this information, we can generate the Huffman encoding tree shown in Figure 8. All values not stored in this tree will be encoded as a concatenation of 5'b11111, and the 7 most significant bits of the coefficient, as we do not see coefficient values greater than 128.

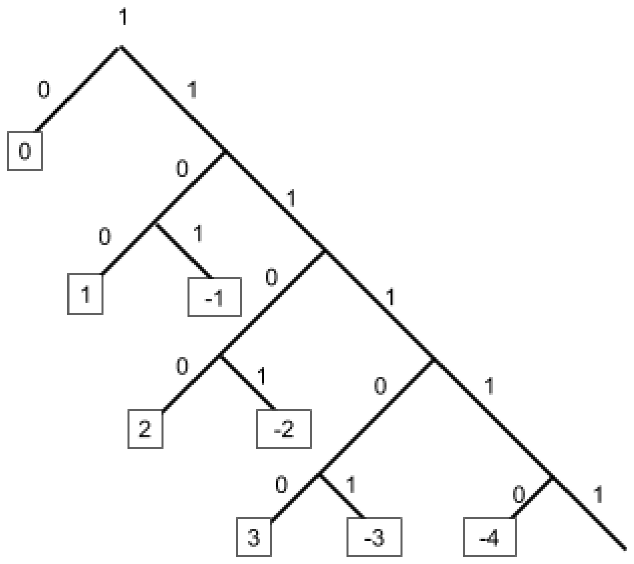
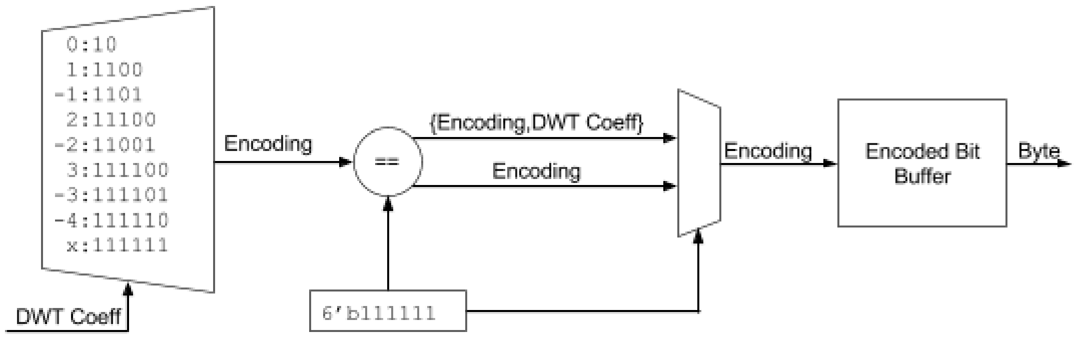


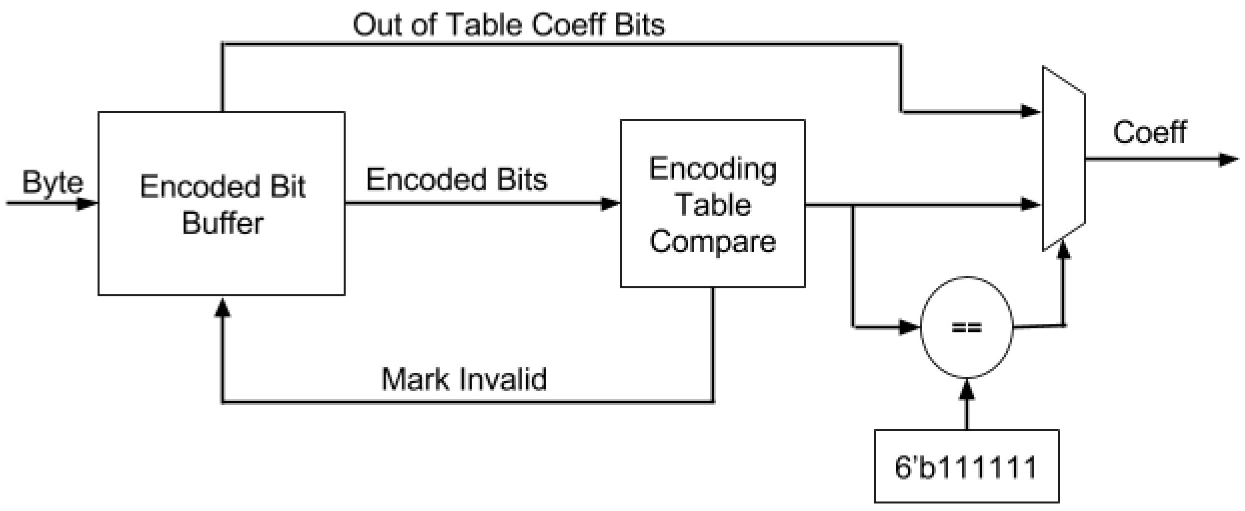
Figure 8: Base Huffman Tree:
This tree shows the basic Huffman encoding for the eight most common values for the DWT coefficients. The more frequent coefficients are represented using the shortest bit representation. The trailing node with no value associated is the position of the out of table values.

In order to build this tree in hardware, we will use the coefficient as the comparator for a lookup table. That table will output either the encoded value or in the case of an out of table value the 5'b11111 to be concatenated. A block diagram of this module is shown below:



Huffman Decompression Module

The huffman decompression module takes the compressed encoded data and transforms them to 16-bit integer coefficients. In order to accomplish this, this module must have knowledge of the same Huffman encoding tree the compression module used. This module will store incoming bits until it matches an entry in the lookup table. When an entry is matched, the stored bits are emptied. We maintain a count of the number of bits stored in order to see the stored leading zeros. A block diagram of the module is shown below. Different schemes can be explored for the table compare to reduce resources used.



Implementation Detail

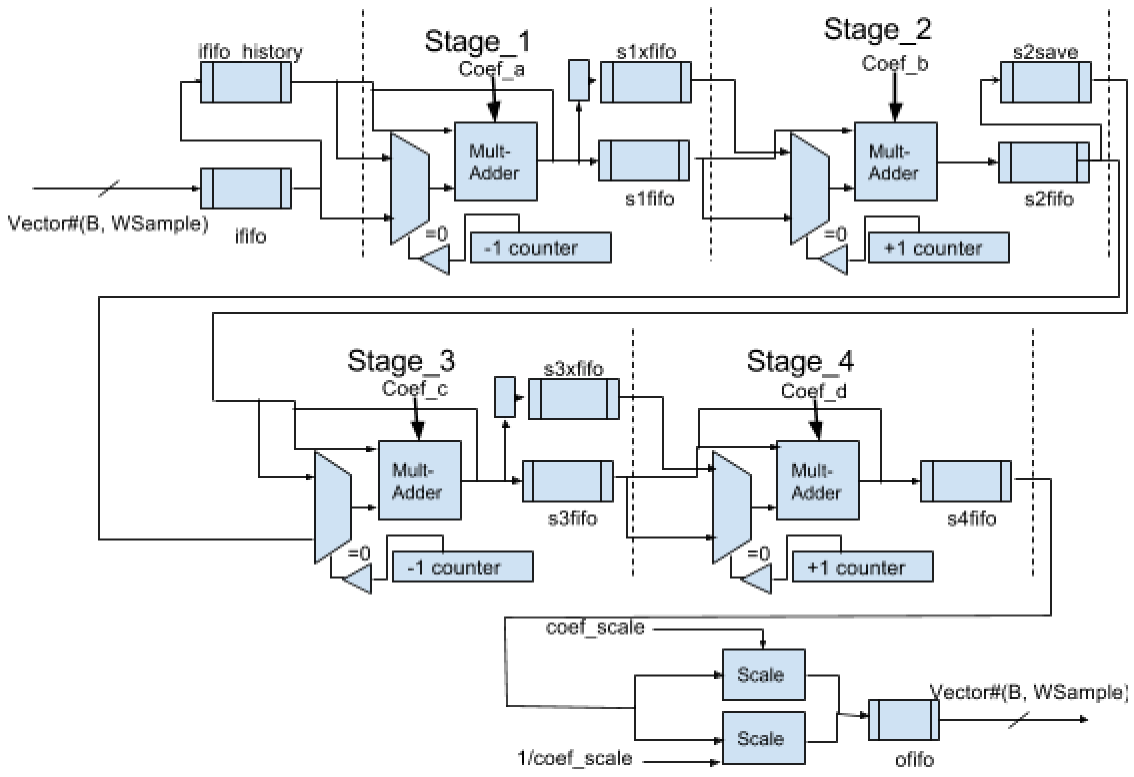
DWT module

The main consideration in implementing this module is to balance between throughput and hardware resources. Each line in the image has ~1000 samples. If one full line is fed into the pipeline in a cycle, the input vector would be as long as 32k bit and will be impossible to synthesize in FPGA. Therefore, each line is serialized into blocks of 2~16 samples and fed into the pipeline in multiple cycles.

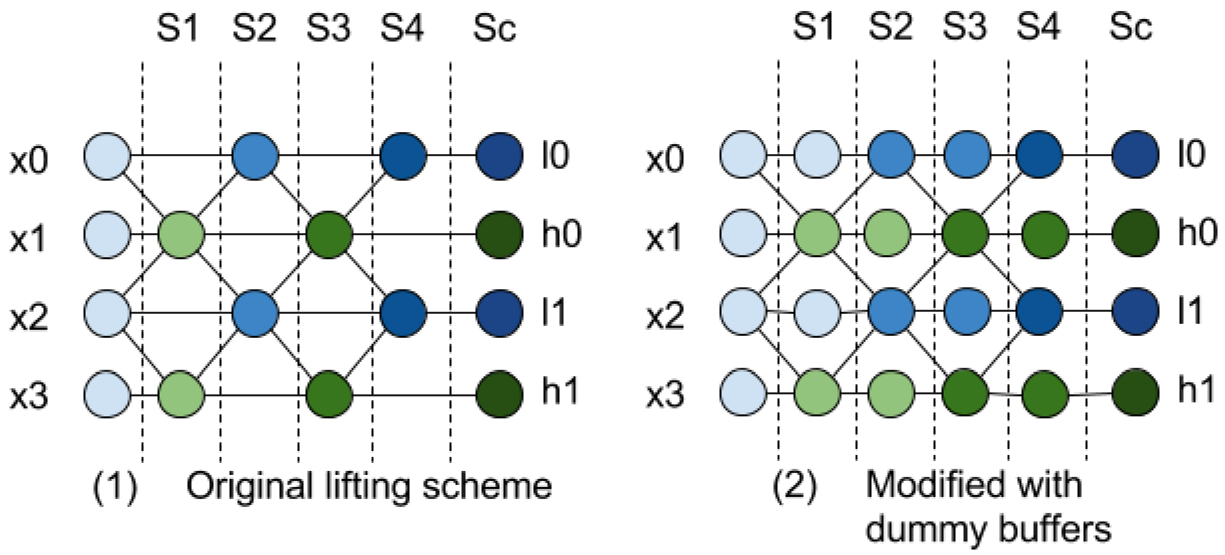
As we mentioned before, DWT is performed on both dimensions. Therefore we will first talk about transformation in the first dimension (on each line), and then the transformation in the second dimensions (on all the lines).

DWT1D module

The following block diagram describes the microarchitecture of this module. It is an elastic pipeline with 4 stages that perform multiply-add operation and one scaling stage which only performs multiplication.



Each lifting procedure consists of four multiply-add stages and one scaling stage. Because of the structure of the lifting network, each stage actually depends on previous two stages. This fact poses a difficulty in decoupling between different stages and makes the pipeline inefficient. Therefore we insert dummy FIFOs that simply pass on the values to the next stage.



Another difficulty we encountered in designing this module is that after serialization of input data into blocks, the calculation of one block is not only dependent on itself, but also on the previous or next block depending on the stage we are talking about. For example, in order to compute the first block of Stage1, we need both the first and the second block of input. This means that we cannot simply deque the input FIFO. Instead, we store the dequeued element in a “history” buffer for the calculation of the next block. The same idea is implemented in every stage except the output stage, and the details are slightly different for odd and even stages as shown in the block diagram.

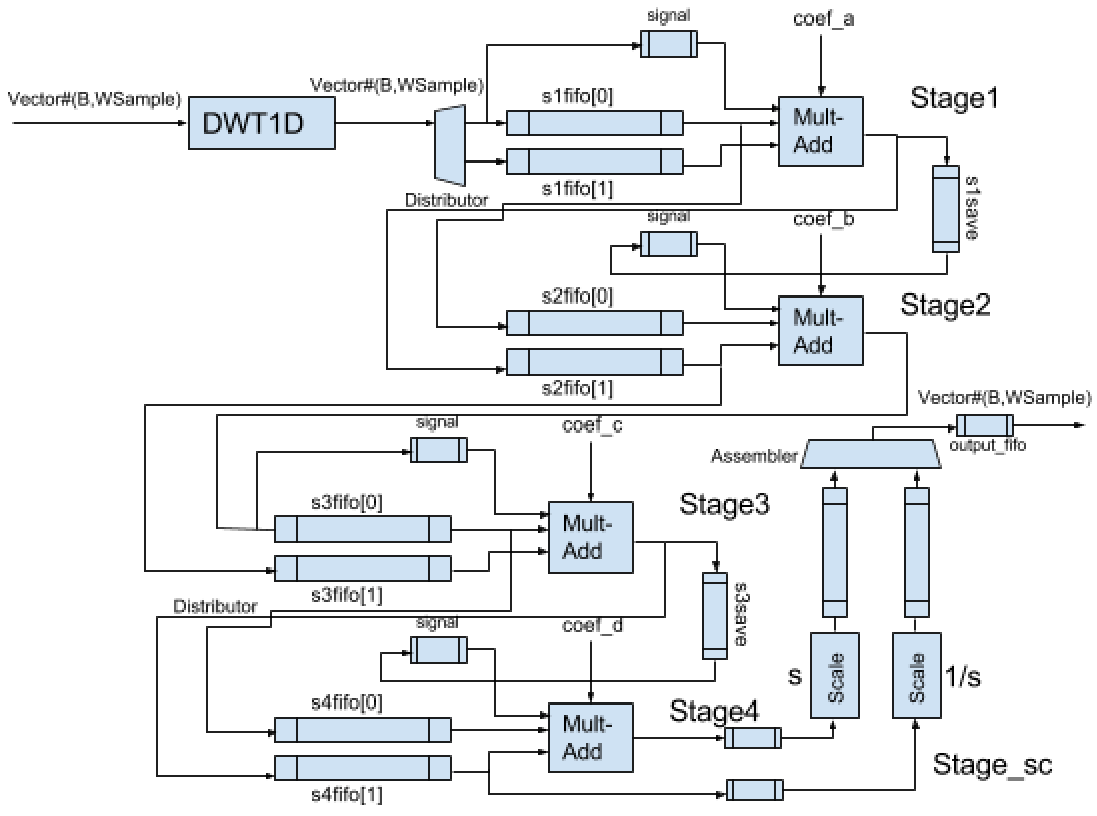
The throughput of this microarchitecture is determined by block size B. Since it is fully pipelined, the module takes in one block in each cycle and the latency from the input to the output of the first block is a few cycles at most. The throughput in ideal case is equal to B samples per cycle, which is 400MSample/s when running at 50MHz and B=8. This is more than enough for our throughput goal.

Synthesis result

We did a synthesis test on this module using B=8 samples/block, each sample is FixedPoint#(16,16), and the hierarchical utilization after place and route gives 6,387 Slice LUT and 3,880 Slice Register used. The functionality correctness has been tested on FPGA successfully using an input size of 2,048 samples.

DWT2D module

Based on the DWT1D module described above, we designed the full 2-dimensional DWT pipeline. Unlike in DWT1D where raw input is chunked into blocks, in DWT2D we process the output from DWT1D in a line-by-line basis. The block diagram is shown below.



The fundamental algorithm is the same as the lifting scheme used for 1-D transform, also with dummy buffers for decoupling between stages. As in the 1-D case, the transformation of each line is dependent on the previous line AND the next line. Therefore, it is necessary to store full lines in the FIFOs between stages. This requires a larger amount of buffer size than what can be handled by normal FIFOs, so we utilize BRAMs on the FPGA that can store up to ~Mb of data. The Bluespec module `mkSizedBRAMFIFO` is very useful for doing this for us. Since the width of RAM blocks are limited (36 bits), to synthesize a wide and shallow FIFO with BRAM will waste a lot of spaces inside the blocks. Therefore the serialization plays an important role in this module. For B=8 and FIFO depth of 512 (holds a maximum line width of 2,048 samples), each “long” FIFO in the block diagram takes 7 RAM36 blocks on FPGA.

Because the module as a whole is now fully pipelined, the throughput is in principle one block per cycle. In simulation, doing 256 consecutive transformation blocks took 293 cycle, where the extra cycles account for the initial latency and some performance loss between images where the pipeline needs to be partially flushed. The latency of a DWT2D transformation is high because the first line of output can be obtained only after the first 5 lines of input are processed.

Synthesis result

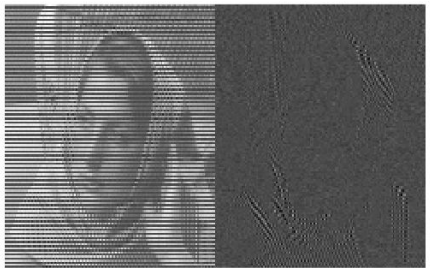
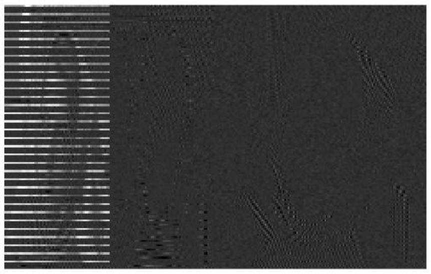
The full DWT2D module with B=8, max line width=2048 samples, was synthesized and place & routed for FPGA. The utilization report gives 20789 Slice LUTs, 9988 Slice registers, 102 RAM36 and 10 RAM18 blocks used. The module is successfully tested on actual FPGA platform with maximum 2,048x2,048 samples input.

Multi-Level DWT2D Module

In order to enhance the compression performance, usually one level of DWT transform is enough, because ¼ of the image contains the low pass filtered image and the coefficients are still relatively large. Therefore, in software implementations of the DWT-based compression algorithms such as that used in JPEG2000 standard, the LL subband (i.e. the upper-left corner that is low-pass filtered) is transformed again and again repeatedly for 4-6 levels to achieve appreciable compression ratio.

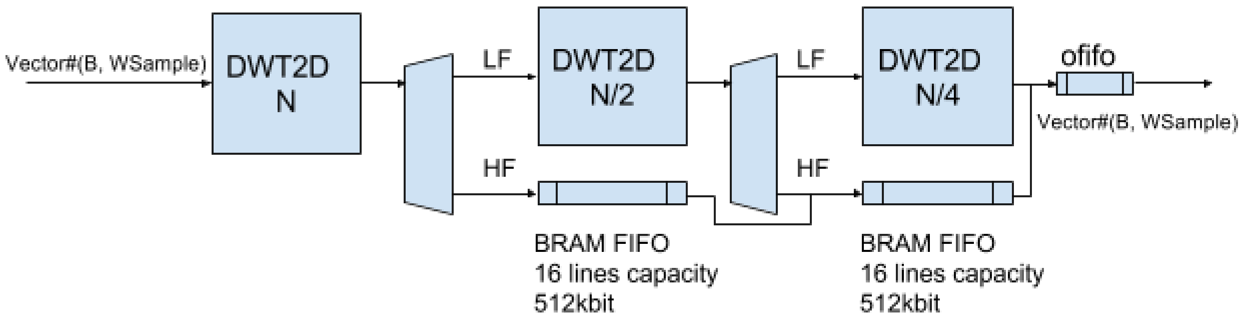
In this project since we are going to eventually synthesize the design on hardware which has limited resource, we have to limit the transformation level to 3 if we do not use a folded design which we think cannot meet the throughput requirement. This would mean that after transformation, only 1/64 of the entire image will be the low-pass filtered and downsampled original image, which contains the largest coefficients. The rest are all high-frequency components that have relatively small amplitudes.

The main challenge in implementing this part is how to avoid buffering a full frame of image, which is impossible to accomplish in the BRAM of an FPGA. As explained before, low-pass components and high-pass components always come out of the transformation module simultaneously, and in our implementation they are arranged in an interleaved manner. This behavior is illustrated below.

From left to right, top to bottom, they are the (1) original image, (2) output from the 1st level DWT2D module, (3) from the 2nd level and (4) from the 3rd level DWT2D module. This means that when feeding the low-frequency output from the first level to the second level, all high-frequency components must be buffered in the pipeline while waiting for the transformed result to come out from the second level. When passing from the second level to the third level, both HF component from the second stage and from the first stage must be stored. The required buffer grows exponentially with the number levels. This is another reason why we limit ourselves to 3 level transformation.

The block diagram of assembling DWT2D modules into a multi-level module (3 levels as shown) is shown below.



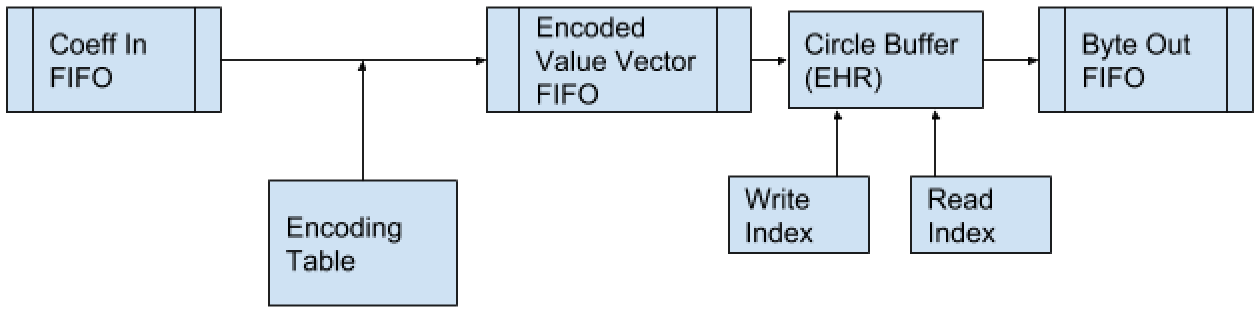
Huffman module

When building the huffman modules in hardware, the goal is create highly pipelined compression and decompression, such that the coefficients can be transferred from the DWT2D to the PC output and the encoded values can be transferred to the DWT2D quickly and be made ready for both interfaces. These modules aim to take low area on the FPGA so many instances can be created for a highly parallelized design that can compress and decompress many values at once.

We will discuss first the Huffman Encoding and then the Huffman decoding. These two schemes will use the tree generated above.

Encoder module

The following block diagram describes the microarchitecture of this module.



The vectors of coefficients (from the DWT modules) are stored in a FIFO and then compared to the table of encodings (generated in software). This table has been isolated from the implementation of the module so we can easily update the table based on need of our application. Each sample input to this table outputs a data structure containing the size of the encoded value, the encoded value, and the coeff (for use in the out of table case) to a second FIFO.

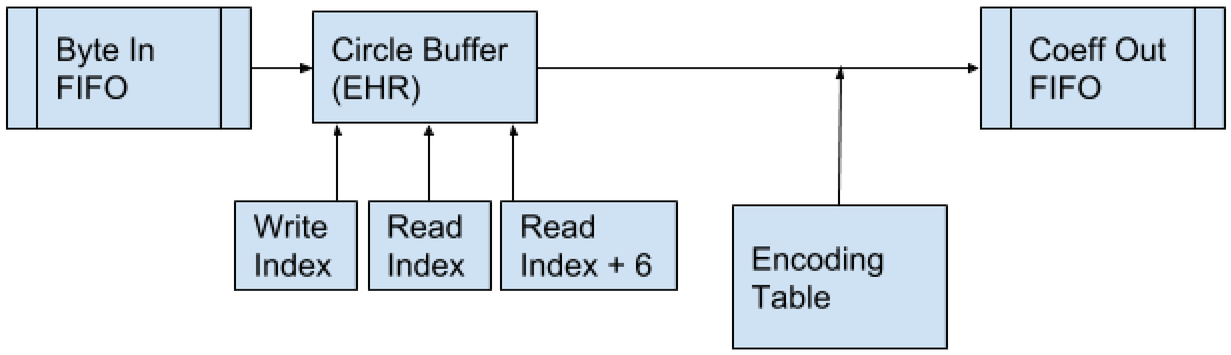
The encoded values are read from the FIFO if all bits of the encoded value can be stored in the circle buffer starting at the location pointed to by the write index. The circle buffer is a bit buffer that is made with EHRs which allow us to perform both read and write operations on the buffer in the same cycle without conflicts. We prioritize writes over reads. Each of these EHRs contains Maybe types, so we can quickly determine whether the data at each location is valid.

The circle buffer is read at a byte starting at the location pointed to by the read index each cycle given that there is at least one byte to be read. If at the end of the transform less than a byte remains in the buffer, encoder will pad the byte with zeros for a final complete byte.

We chose to optimize for speed and high throughput, thus use an architecture with higher memory and area utilization. A slower implementation with full serialization of the coefficients has been implemented and can be used to decrease area costs.

Decoder module

In order to eliminate the external serialization and deserialization, we are implementing the following architecture. The decoder takes in the stream of encoded values in byte by byte fashion. Similar to the encoder module, these bytes are stored in a circle buffer starting at the write index, provided there is space to write in the buffer. The circle buffer is constructed in the same manner as in the encoder. The decoder checks through the 6 oldest bits of the circle buffer to a match in the encoding table by checking through the read index through the read index + 6. When a match is found within these 6 bits, the bits that match are marked Invalid and the coefficient associated is sent to a deserializer to create the vector of coefficients expected by the DWT. If the match is for the out of table value, the decoder extracts the next 6+N bits, buffer locations read index + 6 to read index + 6 + N, where N is the size of coefficient, to determine the coefficient. All bit location read are marked as Invalid. A block diagram of this module is shown below:



The decoder seems to be the bottleneck of the whole pipeline, as the encoded tokens have variable length and one sample at most can be output from the pipeline in each cycle. Since the utilization of this module is pretty low, we think this problem might be able to be partially alleviated by raising the clock frequency.

Testing

We constructed a test pipeline as DWT->Encoder->Decoder->IDWT to test the functionality correctness. Since there is a quantization step between DWT and Encoder, some information is lost and the output is only similar but not exactly the same as input. We tested this pipeline in small scale (16x16 samples) with simulation and the algorithms described above are proven to work.

DRAM module

Memory management is an important factor in our design. The memory must store the decompressed data, for both use by other locations as well as for sending back to the PC. Currently, our memory implementation simply stores the pixels from the IDWT and sends them to the DWT.

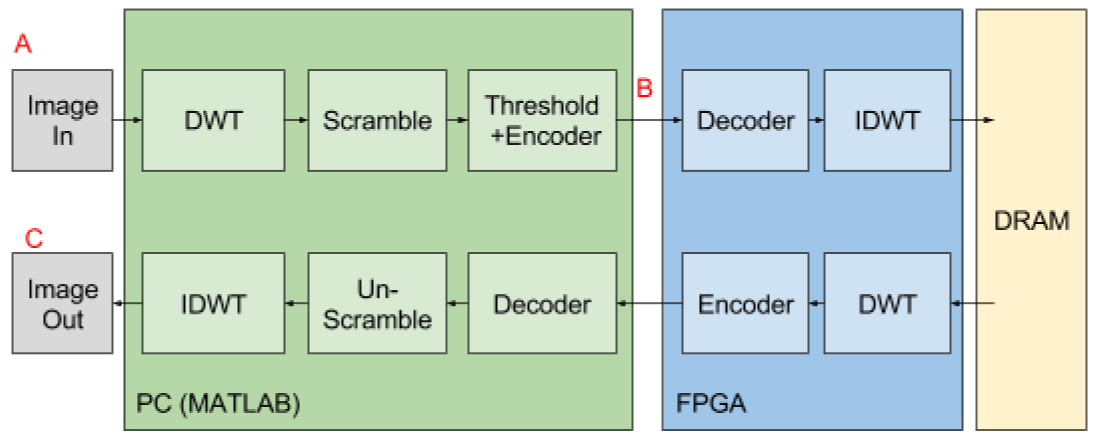
Synthesis result

We synthesized the basic SceMi to DRAM module including associated C++ test bench. Utilization is 32499 LUTs, 36881 Registers, and no on chip BRAM memory. As most of the utilization will be expanded in the inclusion of the full pipeline, this is acceptable. The modules have been implemented in FPGA and the testbench passes.

A challenge in the DRAM Implementation is that the DRAM controller dropped requests unexpectedly when interfacing with SceMi. By using the staff provided controller that adds an additional guard to the controller preventing request overflow, we were able to write and read the same data.

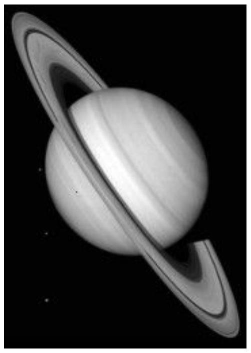
Simulation Results

We have run two simulation tests. Both tests run on the following test structure:

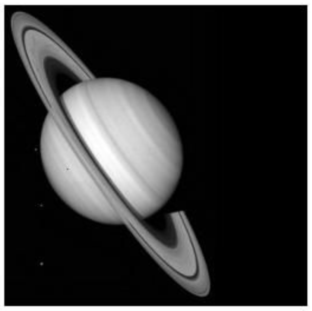


For reference, input images will be denoted as A; images compressed by the MATLAB implementation will be denoted as B; and images compressed by the FPGA implementation will be denoted as C. This is to distinguish the lossiness of the software versus hardware. Typical use case will not directly follow this path.

Our first simulation test tests with a black and white image of 256x256
Initial Image (Monochrome, padded to be 256x256) (A):

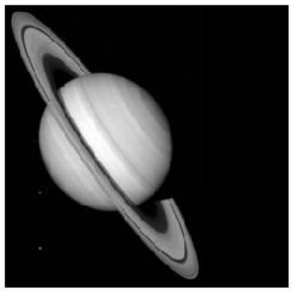


Compressed by MATLAB only (B):



Very similar to our initial image.

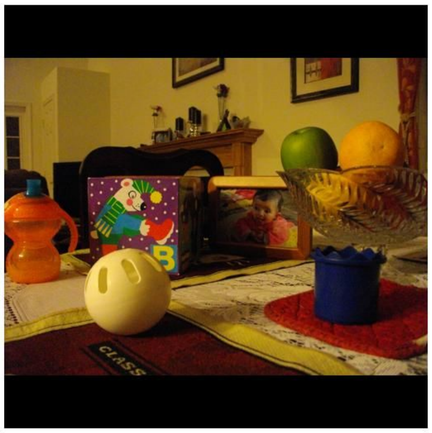
Compressed by MATLAB and FPGA (C):



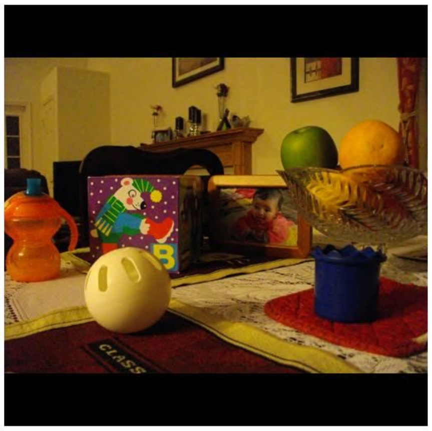
As the algorithm is not lossless, we see noise on some of the edges. The final result is within our expectation.

Our second simulation test tests with a color image of 512x512. This compression is lossy in that only 8 MSBs out of the 12-bit coefficients are transmitted, so that the compression ratio is greatly enhanced. The compression ratio for this image is 0.23 compared with raw RGB 8-bit image. Visually there are tiny differences that are visible, but overall the quality loss is very little.

Initial Image (RGB, padded to be 512x512) (A):



Compressed by MATLAB only (B):



Compressed by MATLAB and FPGA (C):

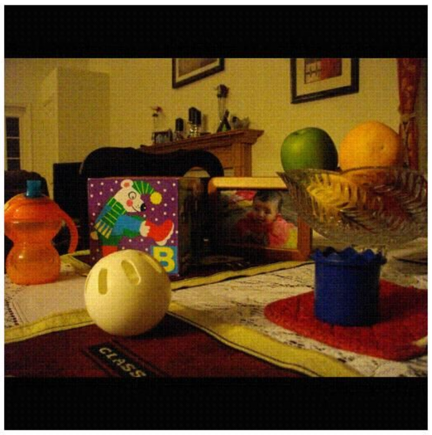


Image C gained some extra graininess because it was lossily decompressed and compressed *twice*.

Implementation Results

Utilization

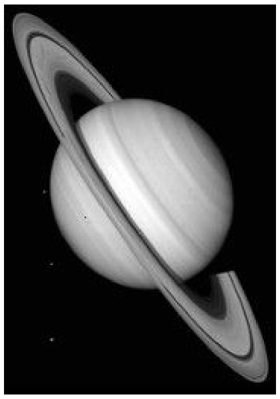
Our full pipeline for 1024x1024 black and white images utilizes 87733 LUTs of 303600 with a utilization of 28.83%, 40997 Registers of 607200 for a utilization of 6.75%, and 296 Block Ram Tiles of 1030 for a utilization of 28.73%.

Our full pipeline for 512x512 color images utilizes 87722 LUTs of 303600 with a utilization of 28.89%, 40703 Registers of 607200 for a utilization of 6.70%, and 246 Block Ram Tiles of 1030 for a utilization of 28.88%.

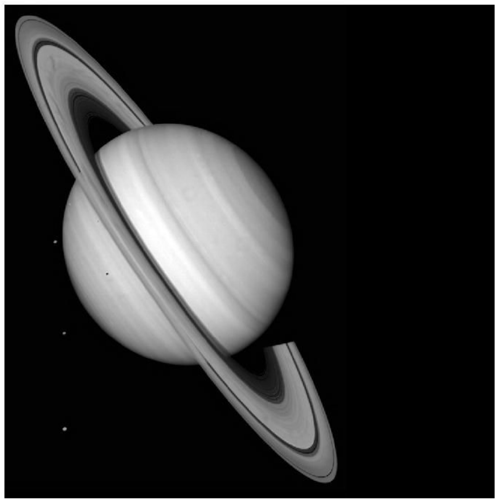
Testing Results

Using the same testing scheme described above, but now using the physical FPGA and 1024x1024 DWT, we generated the following images.

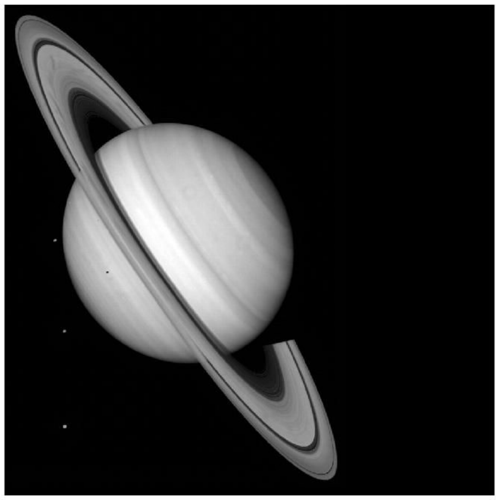
Initial Image (Monochrome, padded to be 1024x1024) (A):



Compressed by MATLAB only (B):



Compressed by MATLAB and FPGA (C):

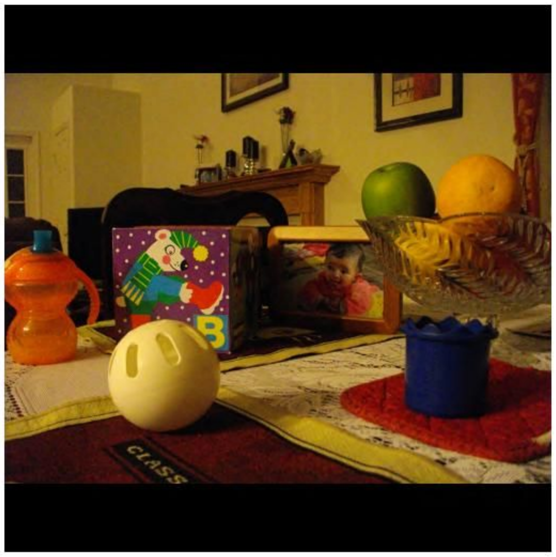


We see little noise in the image because the transformation itself is lossless and the only loss come from the quantization, which has 12-bit resolution.

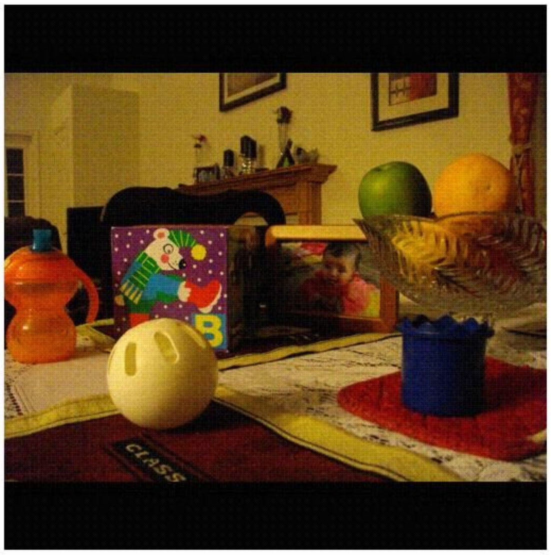
Using the same testing scheme described above, but now using the physical FPGA and 512x512 DWT and lossy transformation, we generated the following images.

Initial Image (Color, padded to be 512x512) (A):

Compressed by MATLAB only (B):



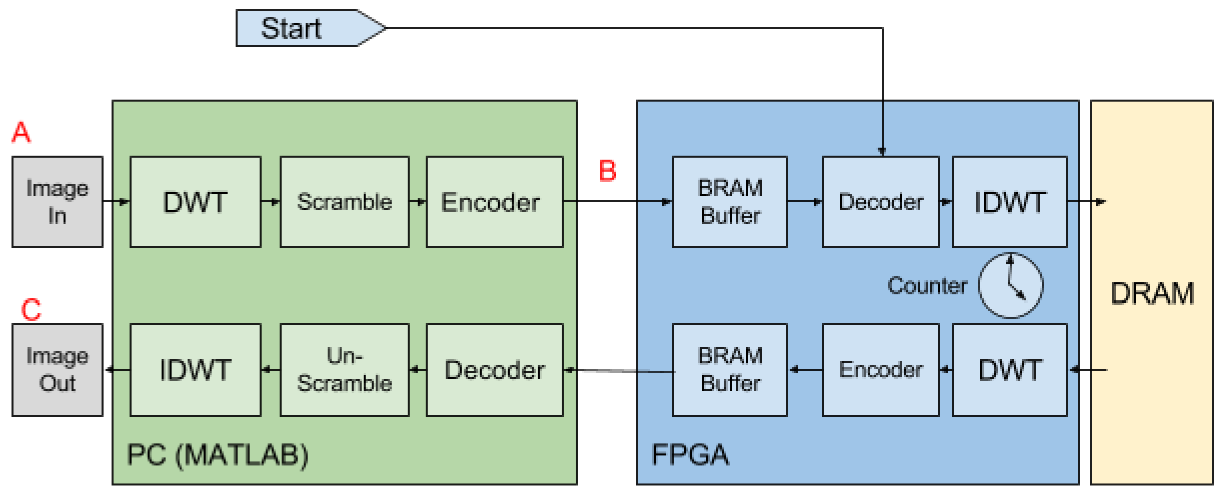
Compressed by MATLAB and FPGA (C):



Timing and Throughput

Our fully-pipelined structure should achieve a 1 Sample/cycle performance. From simulation and FPGA on-board test this is indeed the case: for 256x256 monochrome image which has 65,536 samples, the pipeline takes 88,238 cycles to complete decoding and storing into DRAM. For 512x512 RGB color image which has 786,432 samples, it takes 855,499 cycles to complete. The ratio approaches one when more samples are processed continuously. The extra cycles are mainly attributed to the delay of the multi-level DWT module and the delay is exponential with the number of levels. For 3-level as we implemented, this delay corresponds to about 40 lines of image.

Even for a 512x512 color image, the compression/decompression finishes in less than 1M cycles which takes ~0.02s @ 50MHz, in which time ~2Mb of data is fed into the pipeline. This means that our throughput will be limited by the SceMi interface if we test the pipeline in real time. Therefore we pre-store the compressed test-image in FPGA BRAM using SceMi datalink, fire a start signal to the pipeline, and count the cycles from the start signal. The modified test structure is shown below.



In this part since we need to buffer the compressed image data, we need two large FIFOs of 4Mb each, which is the largest size the FPGA would allow. This limits our test to 512x512 color image. In this part of the test the synthesis on FPGA gives a utilization of 89,048 LUTs (29.33%) and 758 BRAMs (73.59%).

Summary

In summary, we have implemented a video compression/decompression that is fully-pipelined and synthesizable on a FPGA. The performance is summarized in the following table.

Clock frequency	50 MHz
Throughput	50 MSPS
Max. Resolution	Up to 2048x2048
Utilization	~88k LUT, 41k Reg, 300 BRAM
Compression Ratio	Down to 0.23
Lines of codes	2,620 BSV 126 C++ 391 MATLAB

Challenges

DWT

Although DWT is well-described by the lifting scheme which is straight-forward to implement in hardware, it is a challenge to fully-pipeline the design to maximize the throughput, while still remains synthesizable on hardware. As a result, we used serialization to reduce the required multiplier/add on hardware, and dummy FIFOs between stages to decouple and pipeline the DWT 1-D transformation module.

In actual implementation, multiple tricks are used to tweak the performance and reduce utilization, such as use of bypass and pipeline FIFOs between stages, and using canonicalizing rules to resolve conflicts between rules. It proved to be an intellectual exercise to optimise the scheduling of different rules that can fire together.

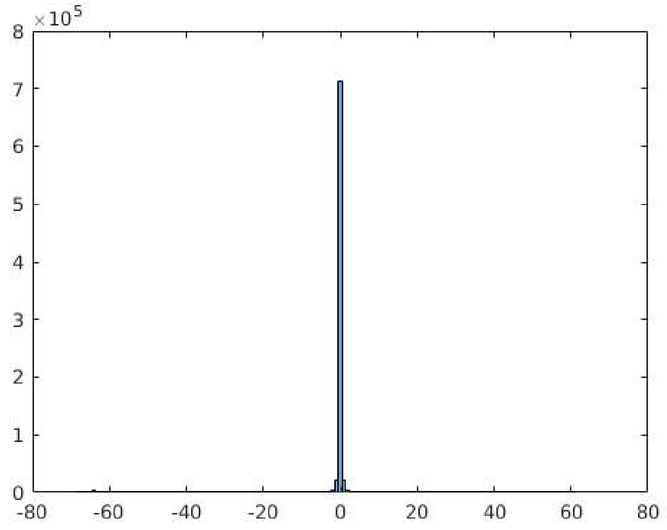
Huffman

The primary challenge with implementing huffman encoding in hardware was the variable length bit widths possible for each encoded word. The simplest solution parses the data in a bit by bit fashion. While this approach standardizes the data width, it has extremely low throughput. Our more complex solution created a buffer in which the encoded bits were stored. This allowed variable length write of encoded data, while a single byte could be read by a different rule. On the decoding side, the maximum data length could be check in a single rule, but only the matching bits were extracted. Further by making use of EHRs, both reading and writing could happen in the same cycle.

Design Exploration

As demonstrated above we have created an architecture capable of achieving a throughput of 1 sample per cycle. At these rates, we can achieve the performance of 1280x720 RGB at 18FPS. This is below the rate of 30FPS necessary for real time. Two explorations to the architecture could allow us to achieve this performance goal.

First, the data transfer rate over SceMi severely constrained our system. While exploring other methods of transfer from the PC is an option, we can also explore different compression schemes for the data. After quantizing the 3 Level DWT we had the following distribution of samples:



The samples are distributed as follows:

0	1	-1	2	-2	3	-3	-4	Other
714019	21542	21291	4233	4228	1785	1897	1171	16266

89.65% of our samples are zero. With this data distribution, Run Length Encoding (RLE), an encoding scheme that sends a data value and count of number of consecutive appearance would perform better than Huffman Compression. By further compressing our data we can transfer it to the board much faster.

Second, we did not experiment with increasing the clock frequency the architecture operates. By determining, whether our system or individual pieces of the system could operate at higher frequencies, we may be able to increase the number of frames loaded per second.

Reference

1. Li, J. (2002). Image Compression-the Mechanics of the JPEG 2000. *Microsoft Research, Signal Processing 2002*.

2. Al Muhit, A., Islam, M. S., & Othman, M. (2004, December). VLSI implementation of discrete wavelet transform (DWT) for image compression. In *Proc. International Conference on Autonomous Robots and Agents, ICARA* (Vol. 4).

3. Hasan, K. K., Ngah, U. K., & Salleh, M. F. M. (2013, November). Multilevel decomposition Discrete Wavelet Transform for hardware image compression architectures applications. In *Control System, Computing and Engineering (ICCSCE), 2013 IEEE International Conference on* (pp. 315-320). IEEE.