# H.264 Video Compression and Transmission: Final Report

1st Elena Su
*Dept. of Electrical Engineering and Computer Science*
*Massachusetts Institute of Technology*
Cambridge, Massachusetts, USA
esu@mit.edu

2nd Reinaldo Figueroa
*Dept. of Electrical Engineering and Computer Science*
*Massachusetts Institute of Technology*
Cambridge, Massachusetts, USA
reyfp@mit.edu

***Abstract*—Here we propose a low-latency implementation of the ubiquitous H.264 video compression and transmission protocol, compatible with the Nexys 4 DDR FPGA. Our system involves two FPGAs: the first will take in the raw 320 x 240 RGB video in VGA format from an input desktop computer, then proceed to encode and transmit the input video as a compressed bitstream; the second FPGA will receive and decode the bitstream in order to render a final 320 x 240 video output. Our bitstream, which is comprised of NAL units, will be transmitted via a direct Ethernet connection between the two FPGAs. Our Ethernet transmitter-receiver pair will be consistent with the official RMII specification, and builds upon the Ethernet NIC from lab.**

***Index Terms*—H.264, Video Compression, Ethernet, Digital Systems, Field Programmable Gate Arrays, Motion Compensation, Video Compression**

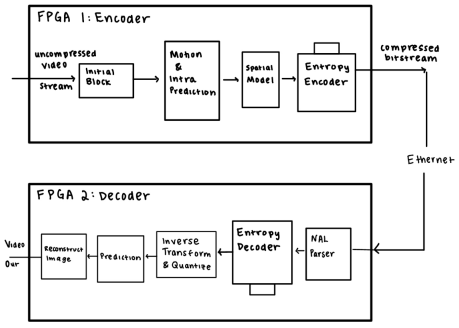
## I. Delegation

Throughout this project, Reinaldo was responsible for working with the Vivado GUI, as well as integrating the motion compensation, entropy encoding and decoding, and the deblocking filters; Elena was responsible for implementing the initial block, spatial model, intra prediction, inverse quantization and transform, NAL Parser, and Ethernet components. As delegated by all team members, Elena and Reinaldo wrote the final preliminary report, and produced most of the diagrams and visuals included in both papers. Both team members collaborated on component integration and debugging. Our code base is hosted on GitHub here, or at the following url: https://github.com/su-elena/h264.

## II. Overview

A high-level module diagram which illustrates these components and our detailed block diagrams are provided on the final page for ease of reference. At a high level, H.264 is a lossy compression standard which achieves efficient video compression and transmission via selective reduction of redundancies in the original video input. In this section, we shall commence with a high-level description of the H.264 protocol; in later sections, we will discuss implementation-specific design decisions. Under this protocol, video is commonly transmitted in YCbCr format as a series of still images, each of which is called at **frame**. First, the H.264 encoder partitions each frame into a series of contiguous 16 x 16-pixel chunks, called **macroblocks**. Within each macroblock, the luma and chroma components of each pixel are selectively sampled to discard extraneous color information. Next, each macroblock undergoes **motion-compensated prediction**, wherein a prediction model is generated using a **reference frame**, which is frequently a previously-encoded video frame. A 16 x 16-pixel block in our reference frame which most closely resembles our current macroblock is located using a search algorithm. The offset between the current macroblock and the reference block will be conveyed as a **motion vector**, which is later encoded and transmitted; additionally, the bitwise differences between the current macroblock and the reference frame, called the **residuals**, will be transmitted also.

In situations when motion prediction is inaccurate, intra prediction can be used to generate a prediction block based on previously encoded and reconstructed blocks within a frame. The prediction block is then subtracted from the current block to generate residuals prior to encoding. Prior to applying entropy encoding, each residual macroblock is sent through the **spatial model**. Within each 16 x 16 macroblock, the basic 4x4 transform is a scaled approximate Discrete Cosine Transform (DCT). The residuals and motion vectors are then quantized into a smaller set of discrete values. Finally, the outputs of the spatial model are encoded using Context-based Adaptive Variable-Length Coding (CAVLC) and Exp-Golomb encoding. All quantized coefficients are encoded using CAVLC; syntax elements are encoded using Exp-Golomb encoding. The output of the entropy encoder is a bitstream. As bytes emerge from the CAVLC encoder, they are fed into a NAL packager, which inserts headers, extraneous zeroes, and delimiters as needed. The output of the NAL packager is the final encoded bitstream. Conceptually, the decoding process is very similar; it is essentially the reverse of the encoding pipeline. Each stage in the encoding pipeline has a corresponding stage in the decoding pipeline which reverses its effects to yield the original transmitted video. Below is a simple diagram which conveys the rough steps of the H.264 encoding and decoding process. In the following sections, we will provide a thorough discussion of module-specific implementation details and design considerations.




## III. Testing and Debugging

To facilitate the testing and debugging of such a complex system, we relied not only on an extensive series of test-benches, but also on an Integrated Logic Analyzer (ILA). We enabled an ILA IP using the Vivado GUI, which was used to test and debug features such as SD reading, Ethernet transmission, and reception of downstream packages. We also relied on GTKWave outputs to ensure the accurate timing of rising and falling edges across all of our testbenched modules.

## IV. Initial Block

Our initial block consists of the RGB to YCbCr converter, frame buffer, and chroma subsampling modules. We convert each pixel from 24-bit RGB to 24-bit YCbCr format, then partition each 320 x 240 frame into 16 x 16-pixel macroblocks.

The module `rgb_to_ycbcr.sv` takes a 24-bit RGB pixel as input and outputs a 24-bit pixel in YCbCr format after a two clock-cycle delay. Conversion formulas are obtained from the Microsemi UG0639 Color Space Conversion User Guide.

Next, we read each 24-bit pixel into `frame_buffer.sv`, which contains a dual-port BRAM holding two 320 x 240 frames. We then index into that BRAM to yield 16 x 16 macroblocks; as macroblocks become available, they are sent into a chroma subsampling module. During this stage, we exploit the BRAM's low-latency reads and writes; to reduce complexity, we also designate one port exclusively for reads and one for writes only.

As the final module in the initial block, `chroma_subsample.sv` implements 4:2:0 chroma subsampling to reduce image complexity. The subsampling patterns are displayed below (Fig. 1).

To ensure the robustness and effectiveness of our initial block, we have tested the entirety of the initial block on an entire 320 x 240 frame: this test yielded the correct subsampled macroblocks, and GTKWave indicated correct `valid_in` and `busy` signals. The frame buffer incurs about 1280 clock cycles of latency; while this is many clock cycles compared to many other modules in our pipeline, this still only a matter of milliseconds in real time, which should be much faster than a modern GPU.

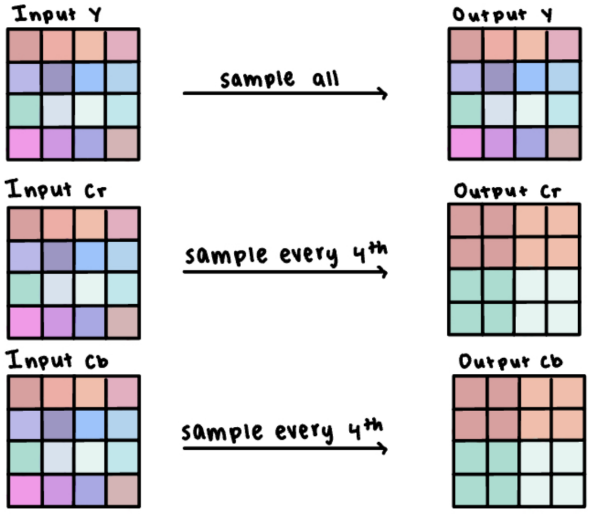



Fig. 1. The 4:2:0 chroma subsampling pattern. For each pixel, we sample the luma component in its entirety; for the red difference and blue difference components, we sample only one pixel for every four input pixels.

## V. Motion Compensation and Intra Prediction

### A. Motion Compensation

For each macroblock, we index into the BRAM containing the reference frame to find the 16 x 16 block within a reference frame which is most similar; from this reference block, we calculate the residual and motion vector, which are transmitted. At the time of this report, we have yet to implement the motion estimation and compensation modules; however, work on these modules has commenced. To find the 16 x 16 block in the reference frame which is most similar to our macroblock, we have decided to use an exhaustive search; this is conceptually the simplest algorithm, and it will simplify the pipelining process during top-level wiring.

### B. Intra Prediction

The `intra_predict.sv` module generates a prediction block based on previously encoded and reconstructed blocks within a frame. The prediction block is then subtracted from the current block prior to encoding.

We have a BRAM cache which holds our reference frame; for each macroblock in our current frame, we execute three forms of INTRA4x4 prediction: DC Mode, Vertical Mode, and Horizontal Mode. We choose the prediction mode which minimizes the sum of squared differences from the actual block, then output the predicted macroblock. The sum of absolute differences is calculated by taking the absolute difference between each pixel in the original block and the corresponding pixel in the reference block; these differences are summed across all 16 pixels to give an indicator of how similar the INTRA4x4 prediction is to the reference image. Below is a visual which demonstrates the three different modes of INTRA4x4 prediction.

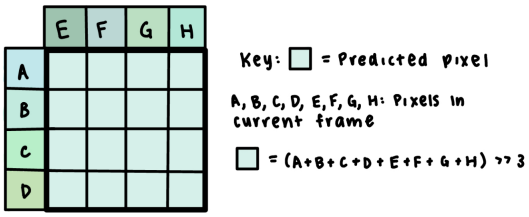



Fig. 2. In the DC mode of INTRA4x4 prediction, all pixels of a subblock are predicted from the average of the values of the pixels, which are above and on the left of the current subblock.

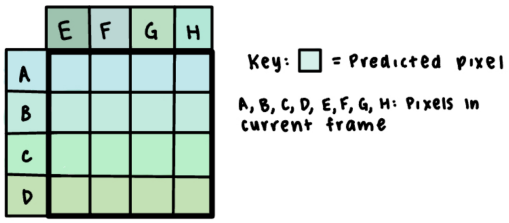



Fig. 3. In the horizontal mode of INTRA4x4 prediction, all pixels in the each row of the subblock are predicted directly from the value of the pixel in the same row, which lies immediately to the left of the block.

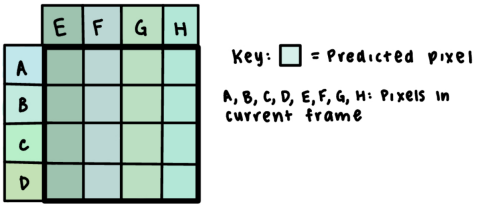



Fig. 4. In the vertical mode of INTRA4x4 prediction, all pixels in the each column of the subblock are predicted directly from the value of the pixel in the same column, which lies immediately on top of the block.

## VI. Spatial Model

Within this pipeline stage, residuals and motion vectors are transformed and quantized into a smaller set of discrete values; we obtain coefficients, which are inputs to the entropy encoder. First, a Discrete Cosine Transform will allow us to express our macroblocks as a sum of cosine functions at different frequencies. The 4 x 4 DCT matrix M is shown below:

$$M = \begin{bmatrix} \frac{1}{2} & \frac{1}{2} & \frac{1}{2} & \frac{1}{2} \\ \sqrt{\frac{1}{2}}\cos(\pi/8) & \sqrt{\frac{1}{2}}\cos(3\pi/8) & -\sqrt{\frac{1}{2}}\cos(3\pi/8) & -\sqrt{\frac{1}{2}}\cos(\pi/8) \\ \frac{1}{2} & \frac{1}{2} & \frac{1}{2} & \frac{1}{2} \\ \sqrt{\frac{1}{2}}\cos(3\pi/8) & -\sqrt{\frac{1}{2}}\cos(\pi/8) & \sqrt{\frac{1}{2}}\cos(\pi/8) & -\sqrt{\frac{1}{2}}\cos(3\pi/8) \end{bmatrix}$$

Fig. 5. 4x4 DCT matrix.

For a 4 x 4 input matrix A, we compute $MAM^T$; in this way, we will apply the DCT to the subblocks within each macroblock. Then, we scale and quantize our transformed outputs; we chose our optimal scaling and quantization parameters according to a 2008 IEEE conference paper [5].

We are performing matrix multiplication in parallel, which gives the FPGA a steep advantage over a traditional CPU. In order to enable accurate calculations using sine functions, we integrated a fixed-point math IP core generated by the Vivado GUI.

## VII. Entropy Encoder and Decoder (Reinaldo)

### A. Encoder

Here, we will use Context Adaptive Variable Length Coding (CAVLC) and Exp-Golomb encoding to encode all quantized coefficients, aided by an SD card. The encoded bitstream will be run through the NAL packager and sent to the decoder via Ethernet.

*1) Exp-Golomb Encoder:* Firstly, we implemented an Exp-Golomb entropy encoder module which you can find inside the `exp_golomb_enc_look_up.sv` file. This module encodes any numbers (or code_num) from 0-512 into its respective code_word. The Exp-Golomb coding effectively uses shorter code_words for code_nums that are more frequent. Each sector of the SD card contains 512 bytes and we use two bytes per code_word. Therefore, we use two full sectors of our SD to effectively store our full look-up table for the Exp-Golomb encoder. We decided to store our look-up tables on an SD card since it offers 1 MB of storage, which is more space than the BRAMs on our FPGA afford. Numbers between 0 and 256, inclusive will be included in Sector #1 and numbers between 257 and 512 are included on Sector #2; we choose the sector we should find our desired code_word based on this criteria. Since every code_word uses two bytes, we initiate a counter that increases every two bytes (in other words, every code_word); therefore, once our counter's value matches our target code_num, we store it, wait for the SD Card reader module to be done (i.e., for the `ready` signal to go high), and return such code_word by driving a data_valid_out signal high. Exp-Golomb encoding is a form of variable length coding (VLC), so using two full output bytes for every code_word would negate its purpose and introduce inefficiencies. Therefore, we padded a an extra 1'b1 bit before the beginning of our code_word so that we know where it starts, and filter it out later during encoding. By using the Vivado GUI, and specifically the Integrated Logic Analyzer (ILA) tool, we were able to retrieve the right code_word for a given code_num from our SD card. Exp-Golomb encoding is used for the encoding of parameters such as macroblock type, reference frame index, and motion vector difference.

*2) Context-Adaptive Variable-Length Coding (CAVLC):* CAVLC is another example of Variable-Length Coding widely used in H.264. Note that CAVLC will be necessary when encoding parameters such as the 4x4 residuals which result from the Discrete Cosine Transform (DCT) matrix multiplication. In addition, the number of non-zero coefficients in adjacent blocks are highly correlated, so our module also uses previously calculated numbers of non-zero coefficients to select which table to conduct the variable lookup. Our

module `cavlc_encoder.sv` contains all the look-up tables we need to encode these parameters: Num-VLC0, Num-VLC1, Num-VLC2, other fixed-lenght coding tables and a Chroma_DC coding table. All these are necessary and have been written inside our CAVLC module. Our module has to make a decision about which table to use based on the number of coefficients of the blocks to the left (declared as *N_l*) and above (declared as *N_u*) of the current block. By using these two parameters we compute use a new parameter *N* that later helps us select the needed table.

| Upper block ($N_U$) | Left block ($N_L$) | N |
|---|---|---|
| X | X | $(N_l+N_u)/2$ |
| X | | $N_u$ |
| | X | $N_l$ |
| | | 0 |

Fig. 6. Computing the *N* value based on upper and left blocks. *N* will be used to select encoding tables.

We then pick which table we have to use based on the value of the computed N. If no values for adjacent blocks were provided, our decision defaults to Num-VLC0. Additionally, our module uses the number of trailing ones (also known as *T1s*, each of which is $\pm 1$) and total number of non-zero coefficients (or *num_coeffs*, as we call it in our module.) Using our previously selected encoding table, our module proceeds through a set of case statements to arrive at the desired code_word. We decided not to offload our look up tables this time since the number of parameters we need to store our tables is not as big and the loop-up operating is done in a single clock cycle. These encoding modules have been tested through testbenches provided inside the sim folder.

### B. Decoder

We implemented Exp-Golomb entropy decoder which you will find in the module named `exp_golomb_decoder.sv` which takes in and exp-golomb code_word and return its respective code_num. Thankfully, this module did not require us to have look-up tables in order to find the code_nums. However, it did required some computations and understanding on how each Exp-Golomb code_word is composed. The way to decode an Exp-Golomb word of the format :

$$[Mzeros][1][log2(code\_num + 1)]$$

is the following:

1) Read all leading zeros followed by a 1, which we call *M*.
2) Read the *M*-bit sized *INFO* field.
3) Then,

$$code\_num = 2M + INFO - 1$$

Our module is capable of taking in a `code_word` of any size and will apply the aforementioned steps to find the `code_num`. It takes a single clock cycle (100 MHz) to perform this computation.

## VIII. Ethernet

We have implemented our Ethernet transmitter; downstream, we use the Ethernet NIC implementation from Lab05 to receive packages on the decoder FPGA.

### A. Ethernet Transmitter

We designed a custom Ethernet transmitter to send our entropy-encoded bitstream over the wire. Our transmitter functions as as follows: we have hardcoded our preamble, SFD, and header. We also specified a custom source address and have modified our downstream Ethernet receiver accordingly. Our input data is read into a two-port BRAM FIFO buffer; one port is used for reads, indexed by the variable `read_address`, and the other port is used exclusively for writes, indexed by the variable `write_address`. We increment `write_address` whenever a new byte is input into our BRAM, and we increment `read_address` whenever an external module requires us to output data from our buffer. Each byte we read out from the buffer was then split into four dibits and sent across the wire. To maximize buffer space and optimize the memory usage of our project, we are storing our preamble, SFD, and header in a register rather than in a BRAM. We then run CRC32-BZIP2 across the entirety of our message and send the FCS after all our data has been transmitted; additionally, all portions of the bitstream excluding the FCS are run through the `bitorder.sv` module in order to be compatible with downstream logic and to account for the Endianness of the Ethernet protocol. Similarly to the Ethernet lab, we used a clock divider to produce `eth_refclk` as an output, which is a 50MHz clock which drives the physical layer. Other outputs were `txen`, a one-bit signal that indicated a valid dibit was being transmitted, and `txd`, which held the dibit that was being transmitted.

### B. Ethernet Receiver

We use the complete Ethernet NIC implementation from Lab05. In simulation, we are able to successfully receive single packets of length up to 1500 bytes (the Ethernet MTU) which are sent via our transmitter; however, we are encountering and troubleshooting a few difficulties regarding dropped header and FCS bits when multiple packets are being sent across the wire.

## IX. NAL Parser

We have implemented the NAL packager and parser, which ensure that all information is transmitted intact between the FPGAs. Displayed below is the structure of the NAL unit.

To simplify our process, our design choice involved eliminating the SPS, PPS, and IDR. However, we decided to interject startcodes intermittently into our bitstream in order to indicate the locations of distinct frames in our bitstream. To reduce the length of our final bitstream, we replaced the official three-byte startcode with a shorter one-byte sequence (`8'b10101010`), which our decoder checks for in downstream logic. This module was debugged using testbenches; verification was conducted on GTKWave. A schematic showing the high-level design of our parser is below.

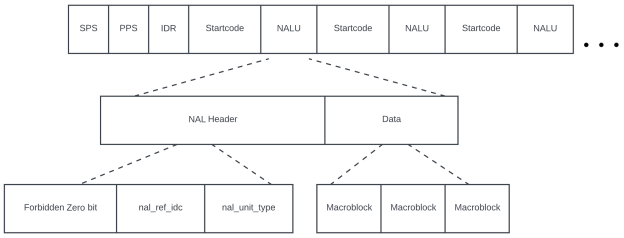
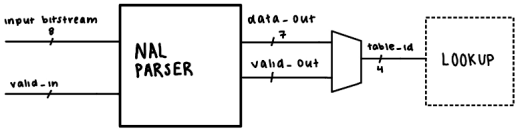



Fig. 7. The bitstream consists of three-byte startcodes, interjected with NAL units. At the beginning of the bitstream, we also would ordinarily have the Sequence Parameter Set (SPS), the Picture Parameter Set (PPS), and the IDR.




## X. Reconstruction

Our image reconstruction module comprises of two stages: first, we will use the decoded residuals and offsets to reconstruct each original frame; next, we will use a deblocking filter to smooth the edges between adjacent macroblocks, leading to a more cohesive image.

We have yet to implement our inverse motion compensation module; but it will use a previous reconstructed frame as a reference, then calculate the address of the current macroblock based on the reconstructed motion vector. To lower the latency of our implementation, and for ease of access, we will store the reference frame in a BRAM.

Our `deblocking_filter.sv` module takes a raw reconstructed image as input; our input frame, which is stored in a BRAM, consists of 16 x 16 macroblocks. We apply a Gaussian blur filter between the edges of adjacent macroblocks to yield a smoother-looking image. The simple Gaussian filters are applied concurrently, which takes 2 clock cycles. This module works in simulation, but has yet to be tested in synthesis. Finally, the output of our

## XI. Evaluation

Our project was clocked at 100 MHz. Our Exp-Golomb encoder performs SD reading which works at 25MHz which we created by using the Vivado clock wizard. As expected, reading from the SD card takes aproximately 9200 clock cycles at this frequency. As we briefly mention inside the Entropy Encoding section, we occupy two sectors of a 2Gb SD card, which means we occupy in total 1Mb to store the look-up tables. Even though our module usually finds our desired value after faster, we wait for the SD controller module to finish ready to not cause timing issues.

## XII. Future Work

If time permitted, we would have devoted more effort into top-level wiring of our modules. Although most of our components were functional in simulation, a potential focus would have been the performance of our modules in synthesis. In particular, areas of improvement include the motion compensation module. The implementation of the motion compensation module was not completed so we would have loved to spend more time finishing it up and including it into our top-level design. The CAVLC decoder module was also not fully implemented due to lack of time.

## References

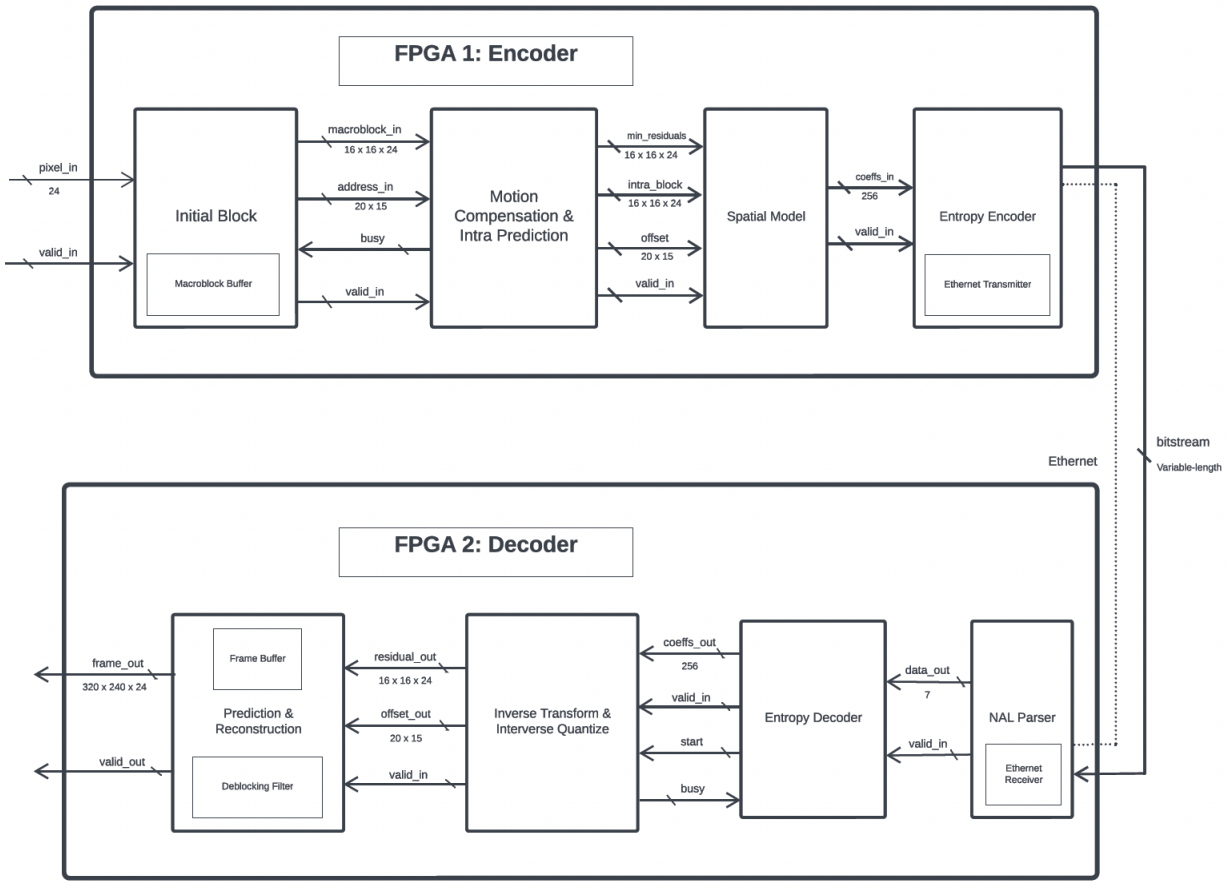
[1] JVT Document JVT-C028, Gisle Bjøntegaard and Karl Lillevold, “Context-adaptive VLC (CVLC) coding of coefficients,” Fairfax, VA, May 2002.

[2] I. E. Richardson, The H.264 Advanced Video Compression Standard. John Wiley & Sons, 2011.

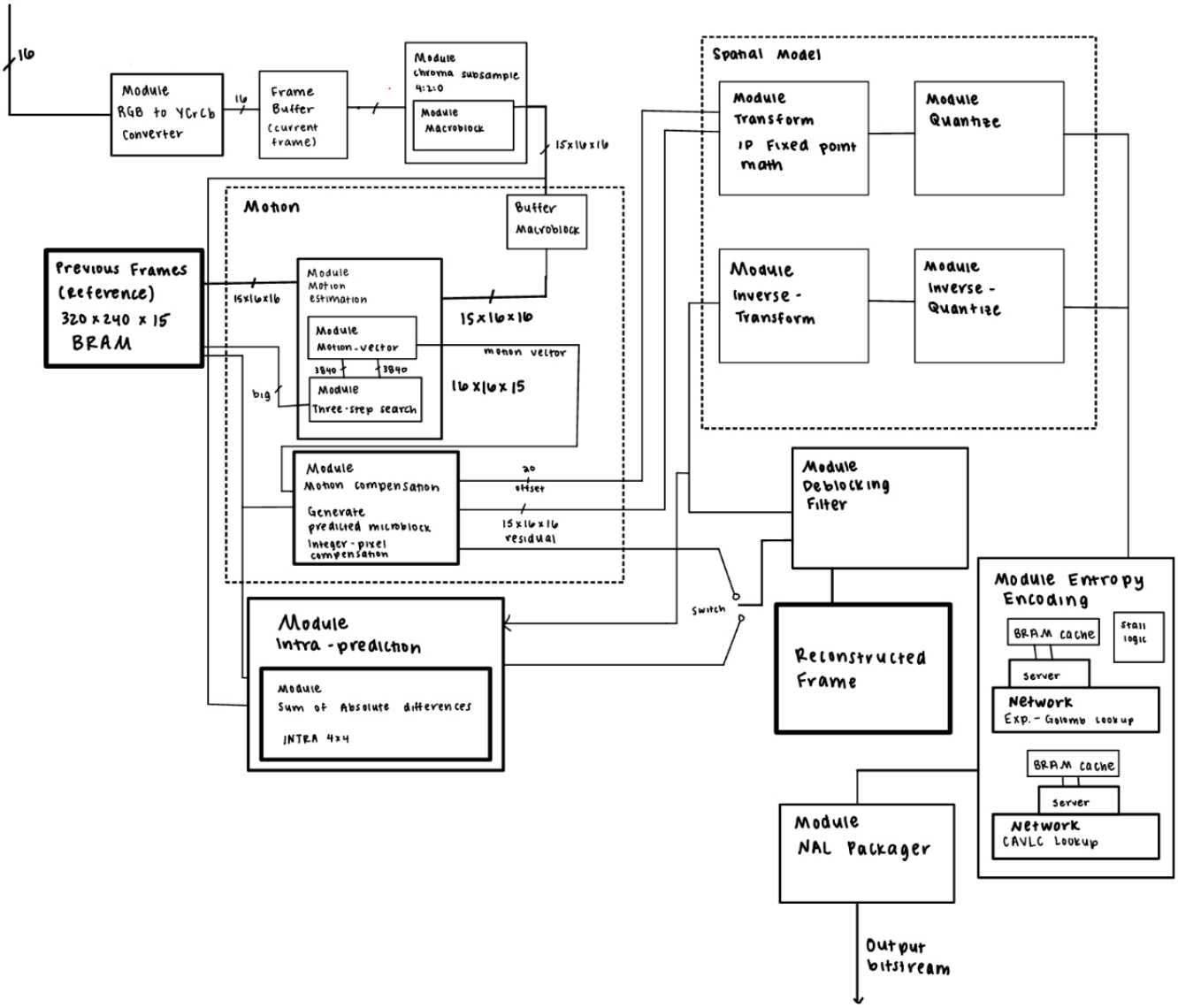
[3] “Vcodex.” Vcodex, www.vcodex.com/. Accessed 15 Dec. 2022.

[4] ”Microsemi UG0639 Color Space Conversion User Guide”, 6th ed. Microsemi Headquarters, One Enterprise, Aliso Viejo,CA 92656 USA

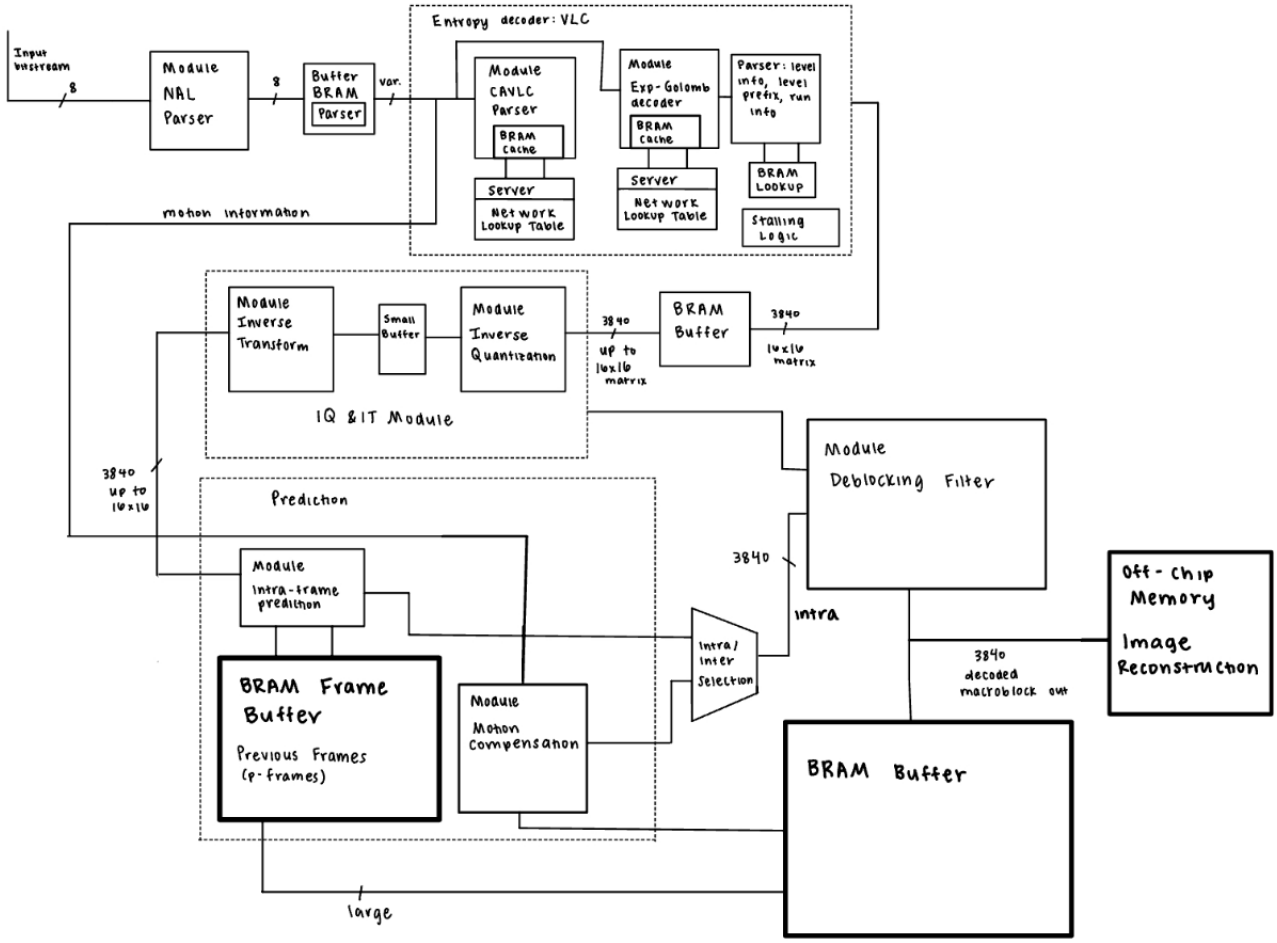
[5] Wu, Ping-Tsung, et al. “A H.264 Basic-Unit Level Rate Control Algorithm Facilitating Hardware Realization.” IEEE Xplore, 1 Mar. 2008, ieeexplore.ieee.org/abstract/document/4518077. Accessed 15 Dec. 2022.


FPGA 1: Encoder
pixel_in
24
valid_in
Initial Block
Macroblock Buffer
macroblock_in
16 x 16 x 24
address_in
20 x 15
busy
valid_in
Motion Compensation & Intra Prediction
min_residuals
16 x 16 x 24
intra_block
16 x 16 x 24
offset
20 x 15
valid_in
Spatial Model
coeffs_in
256
valid_in
Entropy Encoder
Ethernet Transmitter
Ethernet
bitstream
Variable-length
FPGA 2: Decoder
frame_out
320 x 240 x 24
valid_out
Frame Buffer
Prediction & Reconstruction
Deblocking Filter
residual_out
16 x 16 x 24
offset_out
20 x 15
valid_in
Inverse Transform & Interverse Quantize
coeffs_out
256
valid_in
start
busy
Entropy Decoder
data_out
7
valid_in
NAL Parser
Ethernet Receiver

# Encoder


16
Module RGB to YCrCb Converter
16
Frame Buffer (current frame)
Module chroma subsample 4:2:0
Module Macroblock
15x16x16
Spatial Model
Module Transform
IP Fixed point math
Module Quantize
Module Inverse - Transform
Module Inverse - Quantize
Motion
Buffer Macroblock
Previous Frames (reference) 320 x 240 x 15 BRAM
15x16x16
Module Motion estimation
Module Motion_vector
3840
3840
Module Three-step search
big
15x16x16
motion vector
16x16x15
Module Motion compensation
Generate predicted microblock
Integer-pixel compensation
20
offset
15x16x16
residual
Module Deblocking Filter
Module Entropy Encoding
BRAM cache
stall logic
Server
Network
Exp.-Golomb Lookup
BRAM cache
Server
Network
CAVLC Lookup
Switch
Module Intra-prediction
Module
Sum of Absolute differences
INTRA 4x4
Reconstructed Frame
Module NAL Packager
Output bitstream

## Decoder


Entropy decoder: VLC
Input bitstream
8
Module NAL Parser
8
Buffer BRAM
Parser
var.
Module CAVLC Parser
BRAM cache
Server
Network Lookup Table
Module Exp-Golomb decoder
BRAM cache
Server
Network Lookup Table
Parser: level info, level prefix, run info
BRAM Lookup
Stalling Logic
motion information
Module Inverse Transform
Small Buffer
Module Inverse Quantization
3840
up to 16x16 matrix
BRAM Buffer
3840
16x16 matrix
IQ &IT Module
3840
up to 16x16
Prediction
Module Intra-frame prediction
BRAM Frame Buffer
Previous Frames (P-frames)
Module Motion Compensation
Intra/ Inter Selection
Module Deblocking Filter
3840
intra
Off-chip Memory
Image Reconstruction
3840
decoded macroblock out
BRAM Buffer
large