6.375 Final Project
Final Report

Chun-Chieh V. Lin
ragnarok@mit.edu

May 18, 2006

1 Introduction

H.264 is one of the newest and most popular video coding standards at the present time. I implemented the first part of the h.264 decoding process. More specifically, I worked on the NAL unit decoding and entropy decoding in that order.

2 H.264 Decoder Overview

H.264 is a standard for video coding, which is essential because video files tend to be very large without compression. It is one of the newest standards, and offers the capability of better compression and error resilience over its predecessors. The main functional blocks and the data flow of h.264 are shown in figure 1.

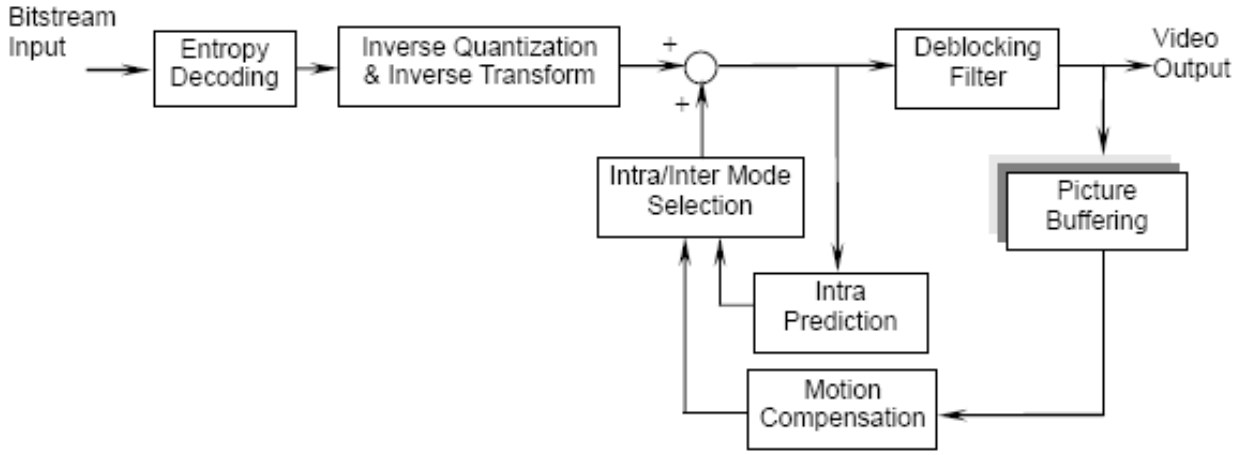

Figure 1: The main functional blocks and the data flow of h.264 (copied from "Overview of H.264 / MPEG-4 Part 10" by Kwon, Tamhankar, and Rao).

2.1 NAL Unit Unwrapping

Coded data of h.264 are separated into data units called NAL units. NAL stands for Network Abstraction Layer, and it is a wrapper of the encoded data that has two formats, byte-stream and packet-based formats. Packet-based formats can be used only when the data transfer protocol can be used as a means of specifying the boundaries of the NAL units. An example of this would be some forms of internet video broadcast. The NAL unit unwrapping step extracts the raw data from these NAL units. It is not shown in Figure 1, but it is the first step in the decoding process.

For most purposes, the byte-stream format is used, as it provides a way to locate NAL unit boundaries by simply scanning the data stream for a special three-byte code. However, this introduces complications. If the data contained in a NAL unit contains the three-byte code, then there must be a way to disambiguate it, so that it is not mistaken for a NAL unit boundary.

I will be assuming that the input is in the byte-stream format, as the byte-stream format can be used in all situations, and packet based format requires interaction with the data transfer channel.

2.1.1 NAL Unwrapping Process

The first step is to search for the "start code prefix," the three-byte code with value 00000000 00000000 00000001 which marks the start of a NAL unit. The subsequent bytes before the next start code prefix (or the end of the video stream) contains the data for the current NAL unit.

However, some bytes with value of 00000011 might have been inserted by the encoder to prevent the start code prefix from appearing in the data. To remove these, the decoder has to scan through the data once, and for every four-byte combination with value 00000000 00000000 00000011 000000xx, the third byte with value 00000011 is removed.

The above step gives a sequence of bytes, but since the data is entropy coded, the real data is actually a sequence of bits. In order to achieve byte-alignment, the encoder padded the end of the sequence of bits with a bit with value 1 followed by zero or more bits with value 0. Therefore, counting from the end of the data, the decoder has to discard all bits up to the first bit with value 1.

Sometimes the NAL units have to be bigger than the data they contain, and extra whole bytes of zeros are used to pad the end of the NAL units. Therefore, the number of zero bits that the decoder has to discard can be very large.

2.2 NAL Unit Types

After an NAL unit is unwrapped, and the data contained in the NAL unit is extracted, the next step depends on what kind of information the unit holds. Each unit contains a one byte header which specifies the type of its content. The three main categories are described below.

Some of the units are simply delimiters that give the boundaries for a single frame, a coded video sequence (a sequence of frames that could be decoded independently), or the entire video stream. For example, if the decoder needs to start decoding in the middle of a stream, it would need to search for a delimiter for a coded video sequence before it starts the decoding process.

Some units contain parameter values that correspond to a single frame or a coded video sequence. Some of the parameters are encoded using extropy coding, which will be explaned in the next section.

Some units contain an encoded slice, which is just a section of a frame. These units contain a header that holds some additional parameters values that are used for the slice. The rest of these units contain two kinds of data, the prediction choices made, and the residual data. The prediction choices give the kind of prediction that should be used for a block of the slice. The difference between the prediction and the input video data is the residual data, which accounts for most of the encoded video stream. These units go through the rest of the decoding steps described below.

2.3 Entropy Decoding

Entropy coding refers to a type of lossless compression that includes Huffman codes. Suppose a file consisting of ASCII characters is to be compressed, entropy coding takes advantage of the fact that not all characters occur with the same frequency. Shorter codewords are assigned to more frequent characters and longer codewords are assigned to less frequent characters, so the overall filesize becomes smaller.

The entropy decoding block in Figure 1 basically decodes the entropy-coded data. However, the entropy coding used in h.264 is not just the general entropy coding. Some additional operations are used to better exploit the known regularities in the h.264 data, but the details will not fit into this preliminary proposal.

Three types of entropy coding are used in h.264, and they are the Exp-Golomb codes, CAVLC, and CABAC.

The Exp-Golomb codes encode integers using a fixed codeword table, so it is fairly simple to decode. The Exp-Golomb codes are used for all entropy coded parts of the coded video stream except for the residual data.

CAVLC stands for Context-based Adaptive Variable Length Coding, and it is a kind of entropy coding where the codeword table continually changes based on the previous data seen. Accordingly, the decoding is much more complicated. CAVLC is used to encode the residual data as described in the previous section.

CABAC stands for Context-based Adaptive Binary Arithmetic Coding. Like CAVLC, it encodes based on the statistics of the previous data, but it does not use a codeword table. Instead, it encodes a large amounts of data at once to achieve compression ratio appoaching entropy. In a sense, it is similar to using codewords which do not have to be an integer number of bits. CABAC achieves better compression ratio than CAVLC, but is substantially more complex. Due to the added complexity, CABAC is only used in some applications that value compression ratio more than the complexity or performance of the coding process. When it is used, it replaces the CAVLC, and also replaces the Exp-Golomb codes for many syntax elements. I was not able to implement the CABAC in this project.

2.3.1 Exp-Golomb Code Syntax

Figure 2 shows the mapping between the variable-length codewords and the codeNums, which are the unsigned integers they represent.

Bit string	codeNum
1	0
0 1 0	1
0 1 1	2
0 0 1 0 0	3
0 0 1 0 1	4
0 0 1 1 0	5
0 0 1 1 1	6
0 0 0 1 0 0 0	7
0 0 0 1 0 0 1	8
0 0 0 1 0 1 0	9
...	...

Figure 2: The codeword to codeNum table for Exp-Golomb code (copied from "ITU Recommendation H.264").

However, the Exp-Golomb code is used to encode many different syntax elements in h.264, and not all of them are simply unsigned integers. Therefore, for many of the syntax elements, an additional table is specified for the mapping between the codeNum and the values they represent. One such example is the mapping to signed integers shown in Figure 3. For these mapping tables, the more common values of the syntax elements are mapped to smaller codeNums, so they can be coded more efficiently.

codeNum	syntax element value
0	0
1	1
2	–1
3	2
4	–2
5	3
6	–3
k	$(-1)^{k+1}$ Ceil(k÷2)

Figure 3: The codeNum to signed integer table for Exp-Golomb code (copied from "ITU Recommendation H.264").

2.3.2 CAVLC Decoding Process

The typical output of the CAVLC decoding process is an array of 16 integers, which are arranged in such a way that the numbers at the start of the array are more likely to have large absolute values. The numbers toward the end of the array are likely to have small values like 0, 1, or -1. An example of such an array would be 8, 5, -1, -2, 0, 0, 1, 0, -1, 0, 0, 0, 0, 0, 0, 0. To take advantage of these known properties, the array of integers is encoded into the following components, each of which has unique code tables.

- The first components are TotalCoeff and TrailingOnes. TotalCoeff is the number of nonzero coefficients in the array. TrailingOnes is the consecutive number of coeffients with absolute value equal to 1, counting backwards among the nonzero coefficients. In the above example, TotalCoeff would be equal to 6, and TrailingOnes would be equal to 2. The pair of these two components is coded as a single token. Depending on some parameters and decoded results of the neighboring blocks, one out of six variable length code tables is used.
- For each of the trailing ones, a single bit is used to specify whether it is positive or negative.
- The other nonzero coefficients are then coded in reverse order (starting from the end of the array). Instead of a code table, an algorithm is specified for decoding the coefficients iteratively. For each coefficient, the algorithm uses the TotalCoeff, TrailingOnes, and the previous decoded coefficient are used as parameters to decode the coefficients.
- The next component decoded is totalZeros, which is the total number of zeros in the array located before the last nonzero coefficient. In the example above, totalZeros would have value 3. Depending on the value of TotalCoeff, one of 15 variable length code tables is chosen to decode totalZeros.
- Now that the number of zeros before the last nonzero coefficient is known, the index of the last nonzero coefficient can be computed. What is left is the distribution of those zeros. First, the number of zeros between the last nonzero coefficient and the second-to-last nonzero coefficient is coded. Next, the number of zeros between the second-to-last nonzero coefficient and the previous nonzero coefficient is coded. For the example given above, these numbers would be 1 and 2, respectively, and no further information would be needed since there are no zeros left. For the encoding of each of these numbers, a variable length code table is chosen based on the maximum value possible (the number of zeros left).

Sometimes, the output of the CAVLC decoding is an array of 4, 8, or 15 integers instead of 16. In these cases, the same decoding process is used, but some code tables might be different.

2.4 Summary of the Above Steps

A more detailed flowchart for the described parts of the h.264 decoder is shown in figure 4. The flowchart shows the decoding flow of NAL units, and is not meant to be a hardware block diagram. The parts of the algorithm that are shown in the figure are the parts that were implemented in this project.

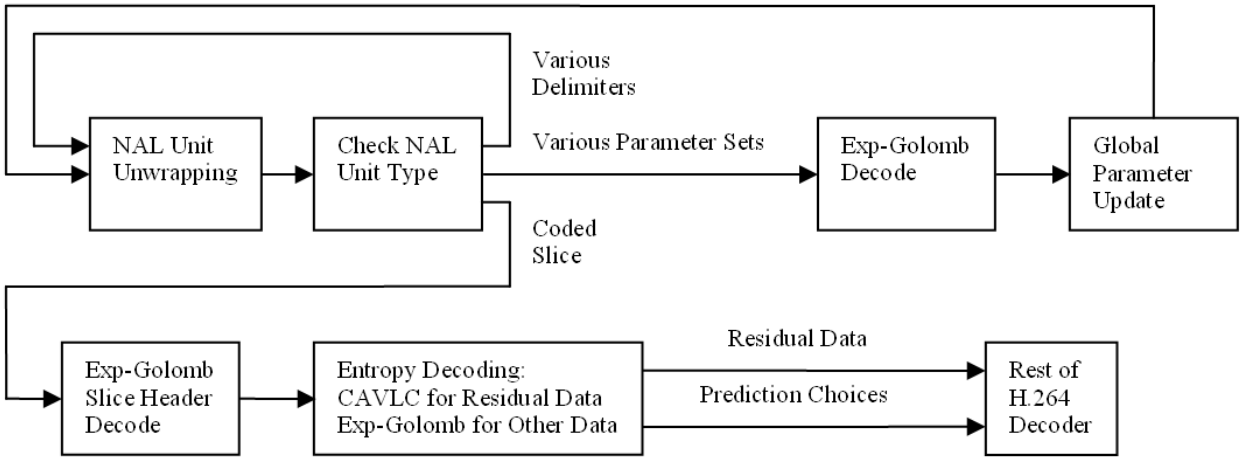

Figure 4: The flowchart for the parts of h.264 decoder that were implemented.

2.5 The Rest of H.264

As shown in Figure 1, there are several more steps to the h.264 decoding process after the steps described in the previous sections. These parts tie together into a feedback loop, and I will not be able to get to them.

3 Hardware Design

The hardware diagram of the implementation is shown in Figure 5. As shown in the diagram, there are two main modules. The first deals with the NAL unwrapping, and the second checks the unit type and performs entropy decoding. In addition, there is a module that reads a file and generates the inputs for verification purposes, and a memory module for the CAVLC context.

Except in the case of the entropy decoder module calling methods of its CAVLC context submodule, the only communication between the modules will be through FIFOs as shown in Figure 5.

3.1 NAL Unwrapping Module

This module will check the boundaries of the NAL units, and remove the extra bytes with value 00000011 inserted by the encoder. After that, it will also remove the extra bytes with value 00000000 at the end of the data (these bytes pad the data when the NAL unit has to be bigger than the amount of data held in the module for some reason). However, the bits used to achieve byte alignment are left untouched, and the entropy decoder module takes care of those. This module will contain the following components.

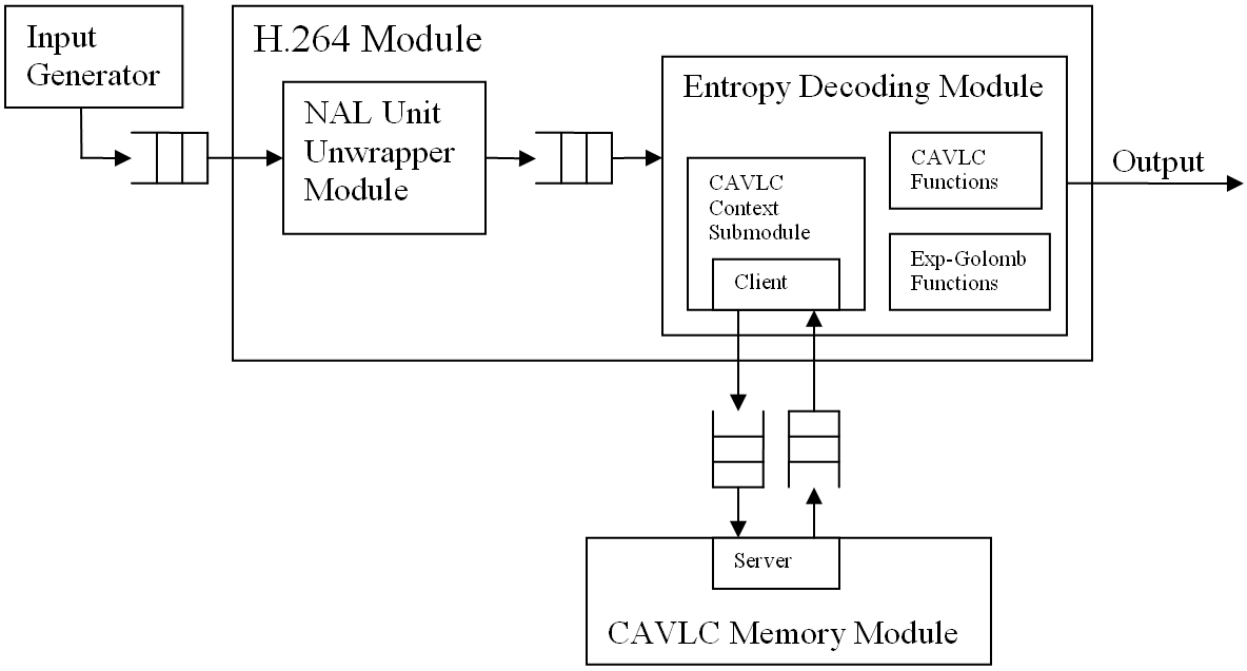

Figure 5: The hardware diagram for the Bluespec implementation.

3.1.1 State Elements

- A three byte buffer for checking the NAL boundaries and removing the unneeded 00000011 bytes.
- A counter for the number of bytes of data currently in the buffer.
- A counter for the number of consecutive 00000000 bytes in the data. This is needed since the module will not know whether these bytes should be removed or not until it sees the next nonzero byte.

3.1.2 Rules

- `fillbuffer`:
  This rule fires when the three byte buffer is not full, and the end of the file has not been reached.
  It simply adds a byte to the buffer with data from the input FIFO, dequeues the input FIFO, and increments the counter by 1.
- `newnalunit`:
  This rule fires when the buffer is full, the end of the file has not been reached, and the start code prefix (code for a new NAL unit) is found in the buffer.
  It throws away the start code prefix, sets the consecutive zero counter to 0, and puts a "new unit" tag in the output FIFO.
- `remove3byte`:
  This rule fires when the buffer is full, the end of the file has not been reached, and the buffer bytes plus the next byte from the FIFO indicates that a byte with value 3 needs to be removed.
  It increments the zero counter by 2, and subtracts 3 from the buffer counter.
- `normalop`:
  This rule fires when the predicates for the three previous rules are all false, and the end of the file has

not been reached.
If the first byte in the buffer is a zero, it increments the zero counter. Otherwise, it checks the zero counter. If the zero counter is 0 it just outputs the first byte in the buffer and decrements the buffer counter. If the zero counter is greater than zero, it outputs a 0 and decrements the zero counter.

- `endfileop`:
  This rule fires when the end of the file is reached.
  It processes the last bytes remainig in the buffer and the remaining zero counts in the zero counter. When ther is no data left, it outputs a tag indicating the end of file.

3.2 Entropy Decoding Module

This module will check the type of each NAL unit, and parse the unit accordingly. The module will contain the following elements.

3.2.1 State Elements

- A state register that specifies the current parsing state of the data. The following is the type of the data stored in the state register.

```
typedef union tagged
{
void    Start;           //special state that initializes the process.
void    NewUnit;         //special state that checks the NAL unit type.

Bit#(5) CodedSlice;      //decodes a type of NAL unit
void    SEI;             //decodes a type of NAL unit
Bit#(5) SPS;             //decodes a type of NAL unit
Bit#(5) PPS;             //decodes a type of NAL unit
void    AUD;             //decodes a type of NAL unit
void    EndSequence;     //decodes a type of NAL unit
void    EndStream;       //decodes a type of NAL unit
void    Filler;          //decodes a type of NAL unit

Bit#(5) SliceData;       //decodes slice data (part of a CodedSlice NAL unit)
Bit#(5) MacroblockLayer; //decodes macroblock layer (part of a CodedSlice NAL unit)
Bit#(5) MbPrediction;    //decodes macroblock prediction (part of a CodedSlice NAL unit)
Bit#(5) SubMbPrediction; //decodes sub-macroblock prediction (part of a CodedSlice NAL unit)
Bit#(5) Residual;        //decodes residual (part of a CodedSlice NAL unit)
Bit#(5) ResidualBlock;   //decodes residual block (part of a CodedSlice NAL unit)
}
State deriving(Eq,Bits);
```

  `Start` and `NewUnit` are special states for initializing the decoding process. The other tags each corresponds to a kind of NAL unit. Several states are used to decode the CodedSlice NAL units, since they are larger than the other types, and the decoding process is much more complicated.

- A buffer with 77 bits (72 should actually be enough). It stores the input data and is large enough to handle all variable length code. Old data is shifted out.

- A buffer counter for the number of bits of data currently contained in the buffer.

- A 16 element FIFO for holding the result or intermediate result of the residual data CAVLC decoding.

- Additionally, some decoded syntax elements are saved in registers since their values are needed for parsing other parts of the data, and some counters and other temporary registers are used.

3.2.2 Rules

- `startup`:
  This rule fires when the current state is `Start`.
  It initializes the various states for decoding a new NAL unit. Before that, it the previous NAL unit has some unneeded data or filler data left unparsed, it throws the data away. At then end, when all the NAL units are decoded, this rule outputs a tag into the output FIFO to indicate that the end of the file has been reached.

- `newunit`:
  This rule fires when the current state is `NewUnit`.
  It checks the NAL unit type contained in the first byte of the NAL unit, and it updates the state register accordingly.

- `fillbuffer`:
  This rule fires when the current state is neither `Start` nor `NewUnit`, the buffer has space for another byte of data, and there are more data bytes from the NAL unit currently being decoded.
  It simply dequeues the next data byte from the input FIFO, and inserts the data into the appropriate place in the buffer. It also adds 8 to the buffer counter.

- `parser`:
  This rule fires when the predicate for all three of the above rules is false.
  It is a finite state machine that parses a NAL unit. Depending on the current state, it decodes a syntax element using functions that parse the Exp-Golomb code or CAVLC. It outputs the results, subtracts the number of bits comsumed from the buffer counter, shifts out the used data from the buffer, and sets the next state. When it has decoded the needed information from a NAL unit, it sets the next state to `Start` so that the next NAL unit can be decoded.

3.2.3 Functions, Submodules, and Memory Usage

Several functions are used by the parser rule for decoding the different versions of Exp-Golomb codes and CAVLC.

In addition, one of the CAVLC decoding schemes requires information from previously decoded blocks. The information is used as the context for CAVLC decoding, and helps determine the decoding table used for the current block. A submodule, Calc_nC, is used to retrieve this information, and determine the appropriate decoding table for the current block. It also saves the information from the current block for future use. This submodule is the one that interfaces with, and directly uses the memory module.

As mentioned above, a memory module is used by the Calc_nC submodule of entropy decoding. This is made a separate module since the amount of memory needed is significant (about 5000 bits for 1080p resolution), and a separate module would provide more flexibility. A client-server interface is established between the Calc_nC submdule and the memory module, with FIFOs carrying the requests and responses. The memory module is not included in the main H.264 module, which allows easier swapping of different memory module implementations. It also makes design exploration easier, as the memory module can be left out of the synthesis calculations.

4 Verification

I used the C version of h.264, currently incoporated into XBS by Jae Lee, for the verification. I found the places in the C code that correspond to the outputs of the hardware funtional blocks, and inserted print function calls to output the values into a file as the C decoder runs. The values are tagged to specify the source of the values to enable easier debugging.

I verified my code by running a simulation in VCS, and writing outputs of the fuctional blocks into a file using the $display statements. These values are tagged the same way that the outputs of the C code is tagged. I then used a Perl script to compare the output against the output from the C version of h.264. Values with the same tag are compared against each other, and when necessary, the order of the values with different tags is also verified.

This allowed me to localize the bugs, as I was able to see which functional block started giving faulty outputs, and what the tags were for those outputs. I could also open the files and compare the outputs for debugging, instead of always having to look at waveforms.

5 Design Exploration

I tried out three design changes. I thought of several ideas, but these three were the most promising and interesting ones.

5.1 Design Exploration A: Special Output for Consecutive Zeros

The transformed residual data that the CAVLC outputs often contain many consecutive zeros. In fact, sometimes all sixteen output numbers are zeros.

In the original code, the entropy decoder just outputs all the zeros one by one. This makes the interface between the entropy decoder module and the next module a little bit simpler. However, I decided that it wasted too many cycles.

I changed the entropy decoder unit to output the number of consecutive zeros instead. When this happens, the output is tagged differently, so that the next midule is able to differentiate between the two. Since the CAVLC coding also takes advantage of the fact that there are often consecutive zeros, I discovered that this is actually the easier way to write the code, and both the performance and area improved for the module.

While this might make the next module slightly more complicated, I realized that this change is actually good for the performance of the next module. This is because the next module multiplies a number to the output of the entropy decoder, then stores the data into a 4x4 matrix, before performing some more operations on the data. The zeros do not need to be multiplied, and the matrix can be initialized to zero, so the zeros don't have to be inserted into the matrix either.

5.2 Design Exploration B: Two-Stage Exp-Golomb Function

While I was debugging the code, I realized that most of the Exp-Golomb coded syntax elements are at most 16 bits in length after decoding. This means that they take up at most 33 bits in the encoded data. However, 11 very infrequently used syntax elements are 32 bits after decoding, and can be up to 65 bits in the encoded data.

Originally, I only had one version of the Exp-Golomb decoding function, and it was capable of decoding even the largest syntax elements in one cycle. This contributed to the critical path of the system.

I created two versions of Exp-Golomb decoder functions. One of them can only decode syntax elements up to 16 bits in decoded length, but can do so in only one cycle. The other version can decode the largest syntax elements, but has two stages, and takes two cycles to complete the decoding. This might increase the number of cycles needed to decode the same file, but the savings in critical path should outweight that effect.

This change also has the added benefit of shortening the required length of the main buffer. In the original design, the main buffer has to be able to hold the 65-bit encoded syntax elements. In the revised design, only 33 bits of the 65-bit encoded syntax elements need to be used per cycle.

5.3 Design Exploration C: Two Stage Buffering

My implementation of the entropy decoder has a rule that filles the buffer, and the parser rule that takes the data from the buffer. The rules cannot fire in the same cycle.

The buffer filler originally inserts one byte per cycle, since the NAL unwrapper outputs a byte at the time. Therefore, the parser has to wait one cycle per byte of input.

I added an extra 32-bit buffer, a counter for it, and a rule for filling it with the output of the NAL unwrapper module. This allows the main buffer filler rule to insert 32 bits at once into the main buffer, which decrease the number of cycles that the parser rule idles.

In order to implement this change, the main buffer size has to be increased by 24 bits. This might lengthen the critical path, and the size of the module would also increase. Therefore, this change may not always be desirable.

6 Benchmark Results

I simulated the operation of the implementation using three clips taken from three different video files. One of them has 5 frames at 176x144 resolution, the second has 15 frames at 176x144 resolution, and the last one has 5 frames at 352x288 resolution. The simulation results and the post-route area and timing numbers are shown in the following table.

Code Version	Original	A added	A+B added	A+C added	A+B+C added
# cycles for the 1st clip	177762	63699	63696	58540	58518
# cycles for the 2nd clip	102448	40850	40880	37711	37713
# cycles for the 3rd clip	374080	146975	146976	134499	134481
# cycles total	654290	251524	251552	230750	230712
post-route critical path delay (ns)	6.468	6.405	5.955	6.400	6.184
total time (ms)	4.232	1.611	1.498	1.477	1.427
postroute area (um$\hat{2}$)	337757	328339	281980	368960	293235

As expected, the first design change (A) increases performance dramatically and also decreases area a little bit. It is so obviously beneficial, I did not bother testing the other changes by themselves.

Adding the second design change (B) also produced the expected benefits. However, it was surprising that it actually decreased the number of cycles used in some cases. It turns out that the syntax elements decoded using the 2-cycle Exp-Golomb decoder are used so infrequently, they did not appear in any of the

three benchmark clips. The changes in the cycle numbers actually come from the changed buffer size, which seems to randomly affect the cycle numbers.

Adding the third change (C) increases overall performance by decreasing the number of cycles needed, but also increases area significantly. Performance per area seems to remain about the same. Whether this change is desirable or not will depend on the rest of the modules not yet implemented, and the relative importance of performance and area. It is interesting that going from A to A+C results in a large increase in area and basically unchanged critical path delay, while going from A+B to A+B+C results in a moderate increase in both area and critical path delay. My guess is that the synthesis tools made some different decisions in area versus performance tradeoffs in these two cases.

7 Possible Future Work

I am planning on finishing the BSV implementation of the h.264 decoder, and using it for my MEng thesis. I hope nobody has done this yet.