Implementation of H.264 Decoder in Bluespec SystemVerilog

by

Chun-Chieh Lin

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

Feb 2007

Author . . . .
Department of Electrical Engineering and Computer Science
Jan 19, 2007

Certified by . . . .
Arvind
Professor
Thesis Supervisor

Accepted by . . . .
Arthur C. Smith
Professor of Electrical Engineering
Chairman, Department Committee on Graduate Students

Implementation of H.264 Decoder in Bluespec SystemVerilog

by

Chun-Chieh Lin

Submitted to the Department of Electrical Engineering and Computer Science
on Jan 19, 2007, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

In this thesis, I present a implementation of a H.264 decoder designed in Bluespec SystemVerilog, a high level hardware description language. This design is intended to serve both as a more understandable reference code, as well as a starting point for efficient hardware implementations. I illustrate this by modifying this initial design to meet a performance requirement of 720p at 60 frames per second.

Thesis Supervisor: Arvind
Title: Professor

Acknowledgments

First I would like to thank Professor Arvind for supervising my thesis project. I would like to thank Kermin Elliott Fleming and Muralidaran Vijayaraghavan for putting my decoder on an FPGA. I would like to thank Jamey Hicks, Gopal Raghavan, John Ankcorn, and the other Nokia people for their support. I would like to thank Nirav Dave, Michael Pellauer, and Chris Batten for their help over the past year and a half. I would like to thank my officemates Jae Lee and Alfred Ng, for always answering my random questions. I would like to thank Sally Lee for all the chocolate and pistachios. I would also like to thank Stephen Hou, Aekkaratt Thitimon, and all my other friends that have helped me during my college years. Finally, I would like to thank my parents for giving me the opportunity to study at MIT.

Contents

List of Figures

Chapter 1

Introduction

H.264 is currently one of the newest and most popular video coding standards. Compared to previous coding standards, it is able to deliver higher video quality for a given compression ratio, and better compression ratio for the same video quality. Because of this, variations of H.264 are used in many applications including HD-DVD, Blu-ray, iPod video, HDTV broadcasts, and various computer applications.

One of the most popular uses of H.264 is the encoding of high definition video content. Due to the high resolutions required, HD video is also the application that requires the most performance from the decoder. Common formats used for HD include 720p (1280x720) and 1080p (1920x1080) resolutions, with frame rates between 24 and 60 frames per second.

1.1 Software and Hardware Decoders

Both software and hardware decoders have been developed for H.264, and both play an important roll. Software decoders are important in providing a cheap and fast way to adapt to the new coding standard, especially for the computer users. Software decoders have the advantage of being easy to write, easy to debug, and easy to upgrade to the new versions.

However, it also takes a lot of computational power to decode a video encoded in H.264, and software decoders running on traditional processors are sometimes not fast enough for high resolution videos. This makes it worthwhile to develop hardware decoders, which can be made to have much higher performance than the software decoders. Well designed hardware decoders are also more power efficient and give off less heat. This is especially important for mobile devices, which are becoming more and more popular.

Unfortunately, designing hardware using traditional hardware description languages is much more difficult than writing software, as it requires writing more lines of code that are more bug-prone and harder to debug. It is also more difficult to try out design changes without modifying large blocks of code not directly related to the change. Compounding the issue is the fact that the designers of video coding algorithms use software to try out and test new ideas, since software is so much easier to implement and modify. This introduces the risk of algorithm designers choosing an algorithm that is unnatural or inefficient to implement in hardware.

1.2 Current Reference Codes

Both software programmers and hardware developers have to understand the algorithm to create a good implementation of H.264. To understand a complex algorithm like H.264, a good reference code to go with the specification would be very helpful. However, it is difficult to gain a clear understanding of the algorithm using the reference codes currently available.

The official reference code, which is what the algorithm designers used to test the designs, is written in C. Another popular reference source of H.264, the libavcodec implementation included in FFmpeg, is also software written in C [4]. These software implementations are imperfect reference codes for several reasons.

First of all, these software reference codes tend to assume a shared memory model. This is obvious in these C implementations, as most communication is based on pointer passing. Pointer passing is very efficient for software running on traditional computers, but it encourages the use of large data structures holding most of the parameters and data, the pointer to which is then passed around to all the major functions. This is not much better than making the contents of these data structures global variables. Someone reading the code cannot easily tell where each piece of data is supposed to be used, where it is modified, and where it stops being useful.

These software codes are also mostly written in imperative languages like C, and are often single threaded. The fixed execution order of these programs sometimes obscure the real dependencies in the algorithm itself, making parallelization very difficult. Especially with the large data structures being passed around, it is difficult to see why the execution order is the way it is, and what other orders are valid.

Optimizations to the code make these problems even worse. Since the programmers have the luxury of knowing the execution order when the optimizations are made, they can make optimizations that depend on the particular order. It can be difficult to distinguish the added dependencies, and the optimizations in general, from what is actually necessary for meeting the specification.

In the end, someone that wants to understand the algorithm by reading the code has to trace through the execution order of the program. Even after getting through all the details, the reader still would have a hard time differentiating the algorithm from the implementation choices.

1.3 Desirable Properties for Reference Codes

It is difficult to say what the perfect reference code should look like. However, there are some important properties that can make it easier for the reader to get a clear understanding of the algorithm.

The reference code needs to have well-defined boundaries between the different conceptual processes of the algorithm. The code should be physically clustered into the different processes, and there should be a clean interface between the different processes as well. On top of that, communication between these modules should be simple and easy to understand. This would give the reader an precise overall structure of the code and the algorithm, which is the first step towards understanding the code.

The code should also clearly specify where storage is needed, so the storage should be local whenever possible. Local storage and clean interfaces allow the reader to focus on a particular process of the algorithm in isolation, and learn the algorithm one step at a time.

Since the algorithm designers often use the reference code to explore different options for the algorithm itself, the code needs to be easy to modify. It would also be useful if the code could be naturally leveraged to obtain good implementations, so the impact of those changes have on performance can be accurately evaluated. The code can then be a common ground for the algorithm designers and implementers to discuss the merits of the possible options for the algorithm. Unfortunately, no current programming language can produce good hardware and software from the same source code.

1.4 Bluespec Implementation

Bluespec SystemVerilog is a new hardware description language that makes it faster and easier to create, debug, and modify efficient hardware designs [1]. It also contains many features useful for writing a good reference code. While it is currently being used to develop hardware, there are hopes that future revisions of it will be capable as software programming languages, especially for multi-core systems.

For this project, the baseline H.264 decoder is implemented using Bluespec SystemVerilog.

Chapter 2

Bluespec SystemVerilog Overview

Bluespec SystemVerilog is a high level hardware description language based on Hoe's TRAC language. Bluespec is based on the concepts of Guarded Atomic Actions or *rules*. Each rule contains a boolean predicate and an action representing a state change. The predicate specifies when the rule is valid to fire. An execution of a Bluespec program can be described as a sequence of firings of the rules in the design, meaning that each rule can be reasoned about in isolation. However, implementations are allowed to execute multiple rules concurrently provided they conform to Bluespec's one-at-a-time semantics. The current Bluespec compiler generates an efficient combinational scheduler which attempts to choose a "maximal" set of rule to fire each cycle.

It has been shown that large designs can be effectively done in Bluespec [2][3]. Also, work has shown that designs made in Bluespec are of equal quality as their RTL counterparts [1].

Bluespec has a strong type-checking system. This prevents a whole class of bugs based on representation errors commonly found in traditional hardware description languages. Bluespec also requires all state elements to be explicitly instantiated, which prevents state elements from being created unintentionally, and makes it easier to understand how the code maps down to hardware.

Like other hardware description languages, a Bluespec design can be separated into modules. However, unlike other hardware description languages, a Bluespec module's interface consists of methods, not ports. These methods are similar to methods in object oriented programming languages. Using a method-based interface helps lift the level of abstraction and prevents incorrect uses of the modules. This is especially useful for large designs, where the interacting modules are often written by different people.

Bluespec also features implicit guards on methods which specify when the method is ready to fire. For example, the enqueue method (`enq`) of a FIFO is only ready when there is space in the FIFO to accept another element. Bluespec automatically incorporates the implicit conditions of methods a rule uses into its predicate. Thus a rule which takes and element from one FIFO and enqueues an element into another will only be ready to fire when there is an element in the first FIFO, and a space in the second.

For a Bluespec design, a variety of rules schedules are possible, each with their own performance characteristics. The Bluespec compiler chooses one possibility, but gives feedback to the designer to help him guide the compiler to make better decisions. For instance, the compiler will notify when two rules in the system cannot be fired in parallel without introducing a combinational path. A designer wanting to allow this path would need to specify this fact to the compiler.

Chapter 3

H.264 Decoding 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 options over its predecessors [5].

3.1 YUV Format

H.264 uses the YUV format, which separates a pixel into a luma component and two chroma components. The luma component gives the brightness of the pixel, while the two chroma components define the color of the pixel.

The human eyes are much less sensitive to color than brightness, so it makes more sense to devote more information resources to brightness. Thus, chroma components are undersampled compared to the luma component. While there is a luma sample for every pixel, there is only one pair of chroma samples (one for each chroma component) for each 2x2 block of pixels.

3.2 Macroblocks and Further Block Division

H.264 divides each frame in the video into "macroblocks," which are blocks of 16x16 pixels. The macroblocks are decoded from left to right, and then top to bottom. Since macroblocks to the top and to the left have already been decoded, information from those macroblocks are often used in the decoding of the current macroblock to improve compression.

During the decoding process, the macroblocks are often further divided into smaller blocks, with the smallest unit being the 4x4 blocks.

3.3 Prediction Overview

Inter-prediction and intra-prediction are the primary means used by H.264 to compress video. In both cases, the decoder uses the data it has already decoded to predict the samples in the next block of the video. Instructions for how the prediction should be done is given by the encoder in the encoded video stream.

3.4 Main Decoding Processes

The main functional blocks and the data flow of H.264 are shown in figure 3-1.

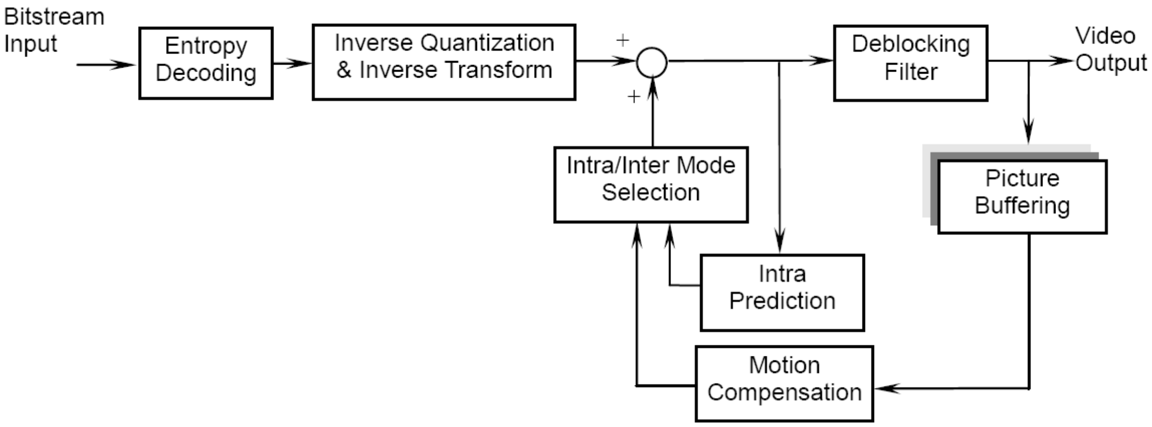


Figure 3-1: The main functional blocks and the data flow of H.264 (copied from "Overview of H.264 / MPEG-4 Part 10" by Kwon, Tamhankar, and Rao [5]).

3.4.1 NAL Units

Coded data are separated into data units called NAL (Network Abstraction Layer) units. NAL unit format serves as a wrapper for the various types of encoded data.

In the first step of the decoding process (not shown in Figure 3-1), the raw data contained in the NAL units is extracted.

3.4.2 Entropy Coding

Entropy coding refers to a type of lossless compression including 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. This reduces the overall file size.

Most of the parameters and data contained in the encoded data is encoded with some form of entropy coding, and they have to be decoded to be used by the rest of the decoder. However, the entropy coding used in H.264 is not the general entropy coding, as it is designed to exploit the known regularities in the H.264 data.

3.4.3 Residual Data, Inverse Quantization, and Inverse Transformation

Whether inter-prediction or intra-prediction is used for a block, the prediction is not perfect. The difference of the two is called residual data, and this information is also put into the encoded video after some processing.

The encoding process uses a discrete cosine transform, which transforms this residual data into the frequency domain. The result is then quantized to throw away some precision in order to achieve lower bitrate. This transformation takes advantage of spacial correlations of the data within the blocks.

3.4.4 Inter-Prediction

Inter-prediction, which is also commonly referred to as motion compensation, is the more common form of prediction used by H.264. It takes advantage of the fact that the content of a new frame in the video often has high correlation to the data in the previous frames.

For each block of the frame up to 16x16 pixels, the encoder looks for a piece (with the same size) of the previous frame that is similar to it. It then encodes that information by specifying the relative location of the block. The decoder then uses this information to reconstruct the block of the frame. This process is called motion compensation, as it compensates for the movement of the objects in the video.

In H.264, the piece of the previous frame that is used for inter-prediction is allowed to use half and quarter pixels. The previous frame is interpolated to give more options for precise motion compensation.

3.4.5 Intra-Prediction

Sometimes it is not desirable to encode a frame using inter-prediction. For example, during a scene change in a movie, the first frame of the new scene has a low correlation to the previous frame. Also, if inter-prediction is used for every frame, it would be impossible to decode a frame in the video without decoding all the previous frames. For the same reason, if the compressed video content is corrupted, inter-prediction would propagate the errors throughout the video.

This is why H.264 encodes a frame without inter-prediction once every couple of frames in the video. In these cases, intra-prediction is used to use part of the frame to predict the other parts. For every block of the frame up to 16x16 pixels, intra-prediction uses previously decoded neighbor blocks to give an estimate for the new block. There are several choices for how to do this, and the encoder picks the best one for each block, and encodes its choice in the encoded video stream.

For frames that are allowed to use inter-prediction, the encoder chooses for each macroblock whether to use intra-prediction or inter-prediction depending on which produces the better result.

3.4.6 Deblocking Filter

Since all the processing described thus far processes the frame in the video one block at a time, the imperfections of the compressed video is most visible on the boundaries between the blocks. H.264 uses a deblocking filter to blurs these edges and give better perceived quality of the decoded video. The strength of the filter used to accomplish this, and whether the filter is applied at all, are dependent on how the data is decoded and some attributes of the decoded data.

3.5 H.264 Decoder Profiles

H.264 specification includes several profiles, each of which contains different subsets of optional features. This project only contains the common features, which form the core of the H.264. None of the optional features specific to a profile are included, and they will not be discussed for the rest of this thesis report.

Chapter 4

Hardware Design Overview

The hardware diagram of the entire H.264 implementation is shown in Figure 4-1. As indicated in Figure 4-1, all the communication between the top-level modules are FIFO-based.

4.1 Module Division

The top level module is separated into modules, one for each of the major processes of H.264 decoding, as they form the natural boundaries in the decoding process. Each of the modules contains submodules where appropriate, otherwise the rules and the functions provide further separation of the different processes.

4.2 Local Memory

There is no hidden communication between the modules. In particular, there is no shared memory. All the data is simply passed through the main FIFO path, and each module saves the data it needs in its local storage or attached memory modules.

This may seem to waste storage space, but most parameters are only used by one or two modules, and any waste is more than made up for by clarity and reductions in critical paths. Furthermore, the modules may be working on different parts of the video, which may use different values for the parameters.

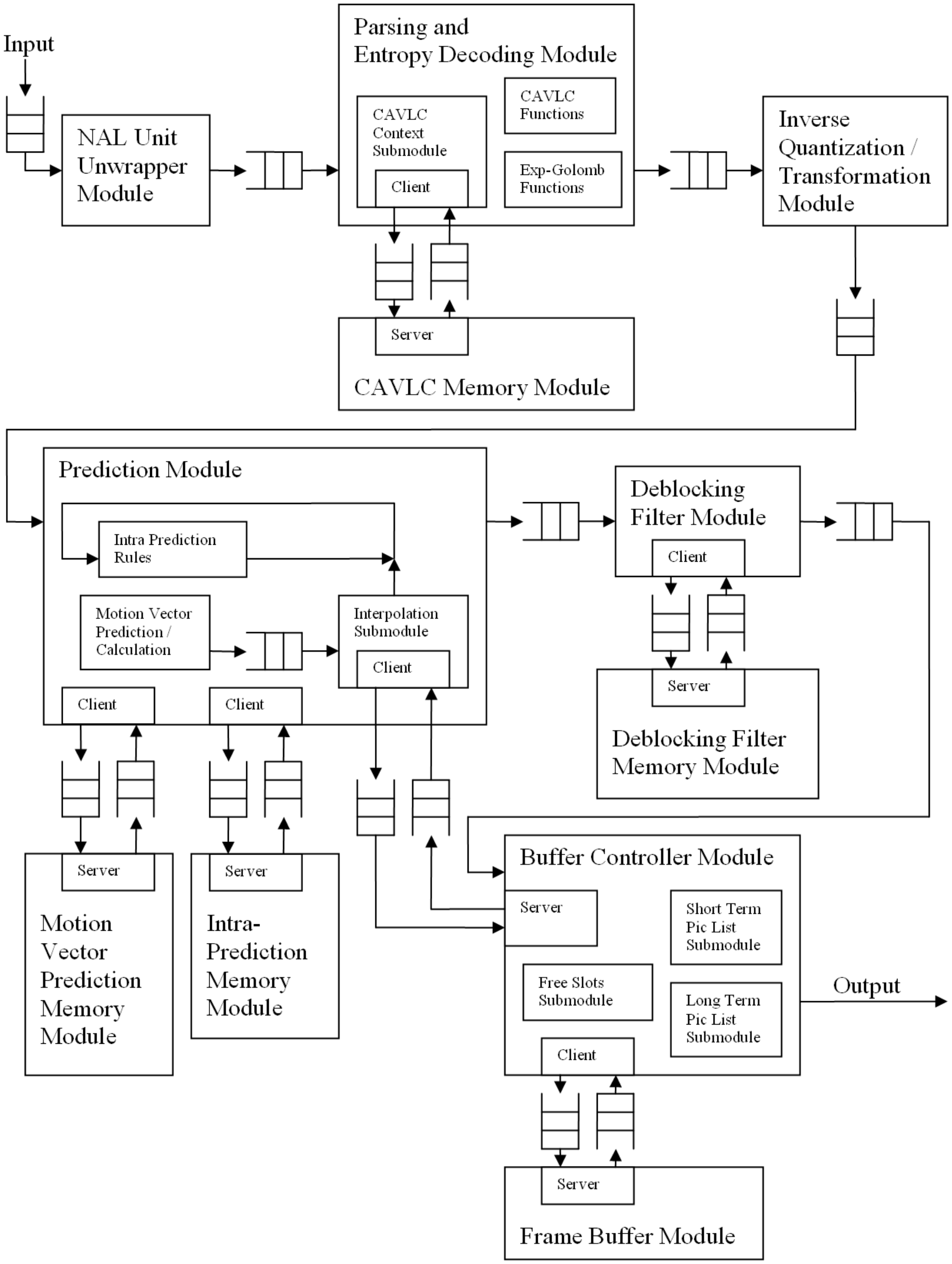


Figure 4-1: The hardware diagram for the Bluespec implementation of H.264 decoder.

4.3 FIFO Interface

Bluespec automatically stalls the rules that require data from empty FIFOs, and rules that try to enqueue into full FIFOs. Therefore, using FIFO channels between units creates an asynchronous pipeline. This greatly simplify analysis with little overhead.

Modules may need to work longer on some parts of the video stream than others. The FIFOs can be easily lengthened in those cases to help cover mismatched rates between modules.

4.4 Tagged Union Datatype

Most of the FIFOs between the modules use the "tagged union" datatype from Bluespec. As an example, the output type of the NAL unit unwrapper module is given below.

```
typedef union tagged
{
void     NewUnit;          //Identifier for the start of a new NAL unit
Bit#(8)  RbspByte;         //A byte of raw data in the NAL unit
void     EndOfFile;        //Identifier for the end of the video file
}
NalUnwrapOT deriving(Eq,Bits);
```

Each element in the output FIFO of the NAL unit unwrapper has type `NalUnwrapOT`. All values of type `NalUnwrapOT` have tags `NewUnit`, `RbspByte`, or `EndOfFile`. If the value has tag `RbspByte`, it also contains 8 bits of data. The `NalUnwrapOT` is represented by 10 bits, 2 for the tags, and 8 for the largest possible data.

Having a single FIFO path using the tagged union datatype makes it easy to keep the ordering of the parameters and data, so the parsing order of all the parameters is preserved through the whole pipeline. This makes the interface between modules very simple, which is important for non-trivial projects like this.

4.5 Memory Modules

As shown in Figure 4-1, many large memories are put inside their own modules, and each memory module communicates with the module that uses the memory through a FIFO based client-server interface. Except for the frame buffer, the memory modules all use the same parametrized implementation.

Using the FIFO client-server interface allows for more flexibility in implementation, since some implementations may not support single cycle read operations. When the memory implementation needs to be changed, the fact that the memories are located in separate modules eliminates the need to change the module that uses the memory.

While this is not shown in Figure 4-1, the memory modules are placed in the top level module, while all the processing modules are grouped into a single module called `mkH264`. This allows the processing modules to be synthesized for design exploration purposes without synthesizing the memory modules. The FIFO interface decouples the memory module from the processing modules, and makes sure that the critical path numbers given by the synthesis results are accurate.

4.6 Top Level Modules

As mentioned in the previous section, the processing modules are all included in a single module called `mkH264`. The module `mkH264` connects the processing modules together as shown below.

```
mkConnection( nalunwrap.io_out, entropydec.io_in );
mkConnection( entropydec.io_out, inversetrans.io_in );
mkConnection( inversetrans.io_out, prediction.io_in );
mkConnection( prediction.io_out, deblockfilter.io_in );
mkConnection( deblockfilter.io_out, buffercontrol.io_in );
mkConnection( prediction.mem_client_buffer, buffercontrol.inter_server );
```

Each of the `mkConnection` statements connects an output FIFO of a module to the input FIFO of another, creating a FIFO link shown in the Figure 4-1.

The `mkH264` module also exposes the interfaces for the memory connections, the compressed input stream, and the final output as shown below.

```
// Interface for input
interface io_in = nalunwrap.io_in;
// Memory interfaces
interface mem_clientED = entropydec.mem_client;
interface mem_clientP_intra = prediction.mem_client_intra;
interface mem_clientP_inter = prediction.mem_client_inter;
interface mem_clientP_buffer = prediction.mem_client_buffer;
interface mem_clientD_data = deblockfilter.mem_client_data;
interface mem_clientD_parameter = deblockfilter.mem_client_parameter;
interface buffer_client_load1 = buffercontrol.buffer_client_load1;
interface buffer_client_load2 = buffercontrol.buffer_client_load2;
interface buffer_client_store = buffercontrol.buffer_client_store;
// Interface for output
interface io_out = buffercontrol.io_out;
```

Finally, the top level module `mkTH` module then connects the `mkH264` to the input, output, and the memory modules.

```
// Input
mkConnection( inputgen.io_out, h264.io_in );
// Memory module connections mkConnection( h264.mem_clientED, memED.mem_server );
mkConnection( h264.mem_clientP_intra, memP_intra.mem_server );
mkConnection( h264.mem_clientP_inter, memP_inter.mem_server );
mkConnection( h264.mem_clientD_data, memD_data.mem_server );
mkConnection( h264.mem_clientD_parameter, memD_parameter.mem_server );
mkConnection( h264.buffer_client_load1, framebuffer.server_load1 );
mkConnection( h264.buffer_client_load2, framebuffer.server_load2 );
mkConnection( h264.buffer_client_store, framebuffer.server_store );
// Output
mkConnection( h264.io_out, finaloutput.io_in );
```

Chapter 5

H.264 and Implementation Details

5.1 NAL Unit Unwrapper

Once again, 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.

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. The start of every NAL unit is marked with this three-byte code. 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.

This H.264 decoder handles the byte-stream format only, as the byte-stream format can be used in all situations, and packet based format requires interaction with the data transfer channel.

5.1.1 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 [6]. The subsequent bytes before the next start code prefix (or the end of the video stream) contains the data for the current NAL unit.

Some bytes with value of 00000011 are inserted by the encoder to prevent the start code prefix from appearing in the data. To remove these extra bytes, the decoder scans 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.

5.1.2 NAL Unit Unwrapper Implementation

This module will check the boundaries of the NAL units, and remove the extra bytes with value 00000011 inserted by the encoder. It also removes extra bytes with value 00000000 at the end of the data (these bytes pad the data when the NAL unit is required to be bigger than the amount of data held in the module). Any 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.

State Elements

- `buffera`, `bufferb`, and `bufferc`: A three byte buffer for checking the NAL boundaries and removing the unneeded 00000011 bytes.

- `bufcount`: A counter for the number of bytes of data currently in the buffer.

- `zerocount`: 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.

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 remaining in the buffer and the remaining zero counts in the zero counter. When there is no data left, it outputs a tag indicating the end of file.

5.2 Parser and Entropy Decoder

The parser takes the raw data of the NAL units from the NAL unwrapper, and parses the data to get the individual syntax elements. Many of the syntax elements are encoded by entropy decoding, and are therefore variable length. This means that the parser cannot know the starting location of the next syntax element until the appropriate kind of entropy decoding is applied to the current one. This is why the parser and entropy decoder are deeply connected. In Figure 3-1, they are labelled as “entropy decoder.”

5.2.1 NAL Unit Types

The first step in parsing a NAL unit is to figure out what kind of data 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. Many of the parameters are encoded using Exp-Golomb code, which will be explained in the next section.

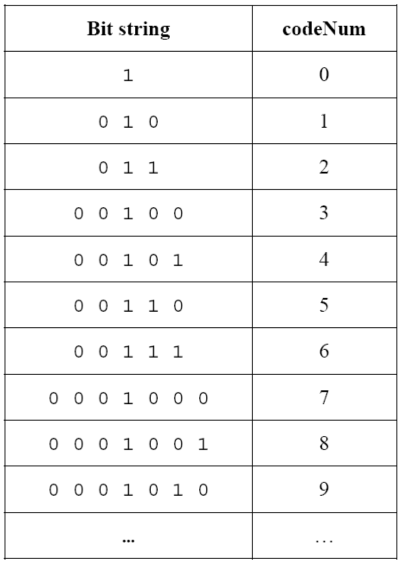
Some units contain an encoded slice, which is just a section of a frame. These units contain a header that holds some additional parameters values that are used for the slice. The rest of these units contain two kinds of data, the prediction choices made, and the residual data. The prediction choices give the kind of prediction that should be used for a block of the slice. As explained in the overview, 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.

5.2.2 Entropy Decoding

There are two types of entropy coding used in H.264, and they are the Exp-Golomb codes and CAVLC. 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.

Exp-Golomb Code Syntax

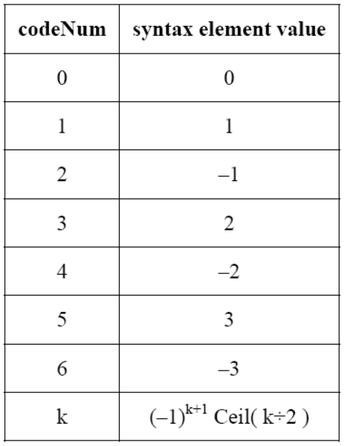
Figure 5-1 shows the mapping between the variable-length codewords and the codeNums, which are the unsigned integers they represent.



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 5-1: The codeword to codeNum table for Exp-Golomb code (copied from the H.264 specification [6]).

However, the Exp-Golomb code is used to encode many different syntax elements in H.264, and not all of them are 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 5-2. 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 5-2: The codeNum to signed integer table for Exp-Golomb code (copied from the H.264 specification [6]).

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. A typical 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 coefficients 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 the code tables are slightly different.

5.2.3 Parser and Entropy Decoder Implementation

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

State Elements

- `state`: A state register that specifies the current parsing state of the data. The following is the tagged union datatype stored in the state register.

```
typedef union tagged
{
void     Start;            //special state:  initializes the process.
void     NewUnit;          //special state:  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 CodedSlice)
Bit#(5)  MacroblockLayer;  //decodes macroblock layer (CodedSlice)
Bit#(5)  MbPrediction;     //decodes macroblock prediction (CodedSlice)
Bit#(5)  SubMbPrediction;  //decodes sub-macroblock pred.  (CodedSlice)
Bit#(5)  Residual;         //decodes residual (CodedSlice)
Bit#(5)  ResidualBlock;    //decodes residual block (CodedSlice)
}
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.

- buffer: A buffer with 64 bits. It stores the input data and is large enough to handle all variable length code. Old data is shifted out.
- bufcount: A buffer counter for the number of bits of data currently contained in the buffer.
- cavlcFIFO: 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.

Rules

- startup: This rule fires when the current state is Start. It initializes the various states for decoding a new NAL unit. If the previous NAL unit has some unneeded data or filler data left unparsed, it throws the data away. 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 consumed 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.

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 submodule 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.

5.3 Inverse Quantization & Inverse Transformation

The amount of quantization varies depending on the bitrate that the encoder wants to achieve. If the quantization process divided the data by some number Q, inverse quantization simply multiplies the quantized data by Q, but some precision is lost in the process since these are integer divisions and multiplications.

The inverse transform step reverses the effect of the discrete cosine transform, which is applied to 4x4 blocks of luma or chroma samples. In some cases, an additional "Hadamard" transformation is applied to the DC coefficients of a group of 4x4 blocks. The inverse transformation for these DC coefficients is simpler than the inverse discrete cosine transform, but both of them basically consists of matrix multiplications. The additional transformation is also the reason the CAVLC process does not always produce 16 values at once.

The regular inverse discrete cosine transform is described below.

5.3.1 Inverse Transformation Process

First, the output of CAVLC decoding is rearranged into a 4x4 input matrix for the following operations. The matrix is then multiplied element-by-element to the following matrix.

$$\begin{pmatrix} 4 & 2\sqrt{10} & 4 & 2\sqrt{10} \\ 2\sqrt{10} & 10 & 2\sqrt{10} & 10 \\ 4 & 2\sqrt{10} & 4 & 2\sqrt{10} \\ 2\sqrt{10} & 10 & 2\sqrt{10} & 10 \end{pmatrix}$$

Note that this is not a matrix multiplication. The above matrix just gives the scaling factors for the corresponding elements in the input matrix. This step is typically combined with the inverse quantization in order to save computation time.

Let us call the output of the above step X, and the following matrix H.

$$\begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & \frac{1}{2} & -\frac{1}{2} & -1 \\ 1 & -1 & -1 & 1 \\ \frac{1}{2} & -1 & 1 & -\frac{1}{2} \end{pmatrix}$$

The output of the input transformation is then $H^T X H$.

However, the above process involves non-integers. In order to make sure that the decoders are consistent with each other, more details are included in the specification. The specification also includes some optimizations for the matrix multiplications.

5.3.2 Inverse Quantization & Inverse Transformation Implementation

This module performs the inverse quantization and inverse transformation, and also the additional inverse transformation of DC coefficients when necessary. It is designed to be a finite state machine that works on only one process at a time. The throughput can be improved by pipelining, but since pipelining would also increase the area, it is only beneficial if the module is a bottleneck of the system.

State Elements

- `state` and `process`: Two state registers, one indicating the current process being done, and the other holding additional information for future state transitions.

- `workVector`: A large register holding the 4x4 block being operated on by the scaling and transform processes.

- `storeVector`: Another large register holding the result of the inverse DC transform. This is needed since these values have to be distributed to their respective 4x4 blocks, where they still have to go through the regular transform process.

- Registers holding various parameters, which dictate the amount of scaling to be done for the inverse quantization process.

Rules Each of the rules performs one of the processes that the module has to do, and only one rule fires at a time.

- `passing`: This rule takes the input from the entropy decoder module, and stores the needed parameters in the appropriate storage elements. If the inverse DC transform is needed, it changes state to initiate the `loadingDC` rule when the data arrives. Otherwise, it switches to the scaling rule instead.

- `loadingDC`, `transforming`, and `scalingDC`: These rules perform the inverse DC transform, and switch over to the `scaling` rule when they finish.

- `scaling`: This rule receives the input data from the entropy decoder, performs the inverse quantization, and stores them in the appropriate place for the inverse transform. It switches to the inverse transformation process after it goes through each 4x4 block.

- `transforming`: This rule performs the inverse transform process. After transforming a 4x4 block, it switches to the `outputing` rule. Note that the same rule is used for the inverse DC transform, but that was just an arbitrary decision.

- `outputing`: This outputs the completed residual data to the prediction module. It then switches back to the `passing` rule or the `scaling` rule, depending on the situation.

5.4 Intra-Prediction

For the luma samples, H.264 uses two types of intra-prediction. One of them operates on a 4x4 block at a time, and the other operates on the whole macroblock (16x16 block) at once.

The intra-prediction for the chroma samples is done independently, and it always works on the whole macroblock at once. However, since the chroma component is undersampled, each of the two chroma components of a macroblock is only 8x8 samples in size.

In all cases, the intra-prediction uses already-decoded samples to the left and above for the prediction. These samples used are the decoded result after the prediction and residuals are summed, but before the deblocking filter is applied.

5.4.1 4x4 Luma Intra-Prediction

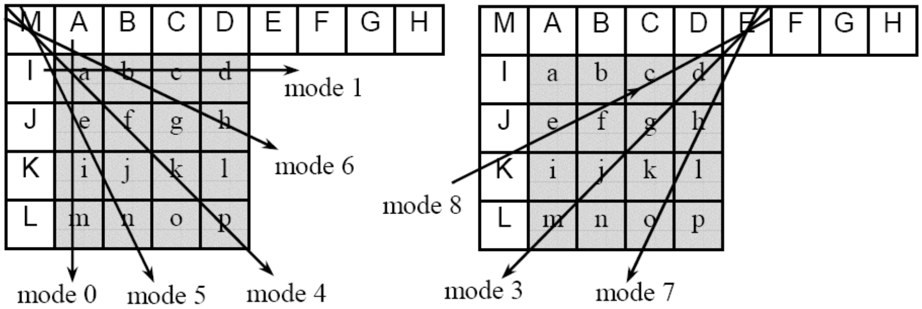


Figure 5-3: 4x4 intra-prediction modes (copied from “Overview of H.264 / MPEG-4 Part 10” by Kwon, Tamhankar, and Rao [5]).

As Figure 5-3 illustrates, there are 9 modes for the 4x4 intra-prediction. In the figure, “a” through “p” are samples to be predicted, and “A” through “M”are the previously decoded samples used for the prediction.

The prediction mode is also predicted based on the prediction mode used by the 4x4 blocks to the left and the 4x4 block to the top. Whether the predicted mode is correct, and what the mode should actually be if the prediction is not correct, is given in the encoded video stream and decoded by the entropy decoder.

5.4.2 16x16 Luma Intra-Prediction

There are only 4 modes for the 16x16 intra-prediction, “horizontal,” “vertical,” “DC,” and “plane.” All of them use the 16 samples directly to the left, the 16 samples directly above, and/or the sample to the top-left. The mode to be used is simply given in the encoded video stream.

To predict a sample, the horizontal mode uses the decoded sample to the left, while the vertical mode uses the decoded sample above the sample to be predicted. The DC mode simply averages the 32 decoded samples to the left and above the macroblock, and uses the mean as the prediction for all 256 samples. The "plane" mode is a little more complicated, as it uses a formula which uses all 33 decoded samples and the position of the sample to be predicted.

5.4.3 Chroma Intra-Prediction

The chroma intra-prediction of the two chroma components can be done independently, but the two components use the same prediction mode, which is given in the encoded video stream. The same 4 modes of the 16x16 luma intra-prediction are used here.

5.4.4 Intra-Prediction Implementation

The prediction module performs both inter-prediction and intra-prediction. While this makes the prediction module rather large and complicated, it would not be as efficient to separate the two into different modules. First of all, intra-prediction needs to use data from previously decoded macroblocks, which might have been predicted using inter-prediction. Having a single module also means that the input FIFO and the buffer for the predicted result can be shared.

Important State Elements

- Various registers that store information about the prediction modes to be used.
- `intraLeftVal`, `intraTopVal`, etc: Registers holding the necessary decoded samples to the left and above the current block. In the case of the blocks above the current macroblock, the values in the registers are loaded from the memory module.

- `intra4x4typeLeft` and `intra4x4typeTop`: Registers holding the 4x4 prediction modes of the blocks to the left and above the current block. These are used for the prediction of the 4x4 prediction mode. Once again, in the case of the blocks above the current macroblock, the values in the register are loaded from the memory module.

- `predictedfifo`: A FIFO for storing the result of the prediction.

Memory Usage The information from the blocks to the left can be stored in registers, since those blocks have just been decoded. However, the information from the blocks above the current macroblock have to be stored in memory, because those blocks were decoded many macroblocks ago.

A parametrized memory module is connected to the prediction module by a client-server interface to provide the necessary storage space. The amount of storage needed is proportional to the maximum width of the frames that the decoder needs to decode. For a maximum frame width of 2048, 4.25KB of space is needed for this memory module.

Relevant Rules

- `passing`: This rule takes the input to the prediction module, and stores them in the appropriate storage elements. When the input indicates that a new macroblock is to be decoded using intra-prediction, it activates the `intraSendReq` and `intraReceiveResp` rules. It also adds the residual calculated by the inverse transform module to the prediction, and outputs the final result. While doing this, it updates the registers and memory module in preparation for intra-prediction of later blocks.

- `intraSendReq`: This rule sends the requests to the memory to load the information from the blocks above the current macroblock.

- `intraReceiveResp`: This rule takes the responses from the memory, and stores the values inside the registers. When all the data has been received, it activates the `intraPredTypeStep` rule.

- `intraPredTypeStep`: For the 4x4 intra-prediction, this rule determines the prediction mode to be used by the current block. Either way, it activates the `intraProcessStep` rule, but only after all the necessary decoded result has been calculated by the `passing` rule.

- `intraProcessStep`: This rule iterates over the samples, and performs the intra-prediction.

5.5 Inter-Prediction

Inter-prediction consists of three main components, the reference picture list construction, motion vector calculation, and sub-sample interpolation. Each of these components will be described in a subsection below.

5.5.1 Reference Picture Lists

After the decoding of each frame, the decoded frame can be placed into either the "short-term pic list" or the "long-term pic list" so that it can be referenced for the decoding of future inter-predicted frames. Usually, the new frame is entered into the short-term pic list, replacing the oldest frame in the list if necessary. The encoded stream can contain instructions on how the frames should be placed inside the pic lists.

For each slice (a piece of a frame, usually the whole frame) to be decoded using inter-prediction, a "reference pic list" is constructed using the frames in the short-term pic list and the long-term pic list. Again, the encoded stream can contain instructions on how to reorder these frames. The more frequently used reference frames are placed in the beginning of the list (smaller index) to take advantage of the entropy coding feature of H.264.

5.5.2 Motion Vector Calculation

For each block (which can be 16x16, 16x8, 8x16, 8x8, 8x4, 4x8, 4x4 pixels in size) to be inter-predicted, the encoded stream specifies which frame in the reference pic list should be used. In addition, a "motion vector" is needed, which gives the location of the block used for motion compensation in the reference frame, relative to the current location in the frame being decoded. For example, if the motion vector is (5,2), then the predicted value for pixel location (1,0) in the current frame comes from pixel location (6,2) in the reference frame.

The motion vector for the current block is predicted using the motion vectors used for neighboring blocks that have already been decoded, taking into account whether each of the neighboring blocks uses the same reference picture or not. The difference between the predicted motion vector and final motion vector that should be used is given by the encoded stream.

5.5.3 Sub-Sample Interpolation

The motion vectors used in H.264 do not necessarily have integer components. For each of the vertical and horizontal components, the motion vector can have quarter precision. For example, For example, if the motion vector is (5.75,2.5), then the predicted value for pixel location (1,0) in the current frame comes from pixel location (6.75,2.5) in the reference frame. The values of these fractional sample locations are generated by interpolating the values of the nearby integer samples.

Luma Interpolation

To illustrate the interpolation process for the luma samples, figure 5-4 shows some integer samples (grey squares) and fractional samples (white squares), each labeled with upper case or lower case letters. For example, pixel location (6.75,2.5) corresponds to "k" in figure 5-4.

The half samples "b," "h," and "j," are calculated using a 6-tap filter. To calculate the value of sample b, the 6-tap filter is applied to the samples E, F, G, H, I, and J.

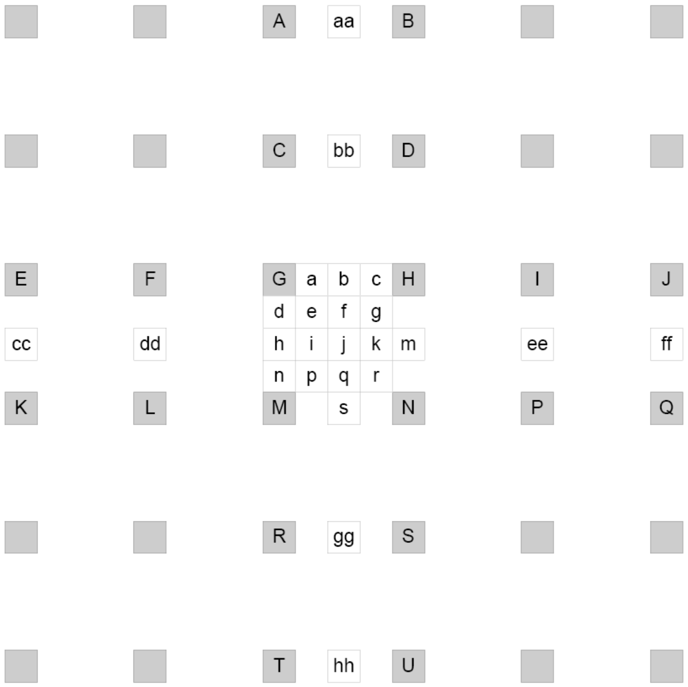


Figure 5-4: The fractional sample positions (copied from the H.264 specification [6]).

Similarly, the same 6-tap filter is applied, except vertically, to get sample h. To get the sample j, the 6-tap filter is first applied horizontally to obtain samples aa, bb, b, s, gg, and hh, then the filter is applied vertically to get the value of j. Alternatively, j can be calculated by filtering vertically then horizontally, as the result would be identical.

The quarter samples “a,” “c,” “d,” “e,” “f,” “g,” “i,” “k,” “n,” “p,” “q,” and “r,” are calculated by averaging the two closest integer samples or half samples.

Chroma Interpolation

Since there is only one chroma sample for each 2x2 block of luma samples, quarter luma positions would actually correspond to eighth chroma positions. However, the chroma interpolation does not use the 6-tap filter. Instead, a fractional sample value is calculated by using a weighted average of the 4 surrounding integer samples.

5.5.4 Inter-Prediction Implementation - Prediction Module

The prediction module performs both inter-prediction and intra-prediction. For inter-prediction, the majority of the work is done by the interpolator submodule, which will be described in the next section. The buffer control module, which constructs the reference picture lists, is also a big part of inter-prediction.

For inter-prediction, the prediction module contains rules that calculate the motion vectors, and a submodule that performs the interpolation. This section will describe the part that computes the motion vectors. The interpolator submodule will be described in the next section.

Important State Elements

- Various registers that store information about how a macroblock (16x16 block of pixels) is divided into blocks for inter-prediction, and the index of the reference picture to be used for each of those blocks.

- `interLeftVal`, `interTopLeftVal`, and `interTopVal`: Registers holding the reference indices and motion vectors used by blocks to the left and above the current macroblock. These blocks have already been decoded, and the information from them is used for motion vector prediction. In the case of the blocks above the current macroblock, the values in the registers are loaded from the memory module.

- `interMvFile`: A register file holding the completed motion vectors for the current macroblock. This is necessary since the motion vectors have to be used three times, first for the luma component, then once for each of the two chroma components.

- `predictedfifo`: A FIFO for storing the result of the prediction.

Memory Usage As mentioned before, the motion vector prediction scheme requires information from previously decoded blocks. Just like in intra-prediction, the information from the blocks to the left can be stored in registers, but the information from the blocks to the top have to be stored in memory. The same memory module implementation used by intra-prediction is used here, except with a different parameter. For a maximum frame width of 2048, 2KB of space is needed for this memory module.

Relevant Rules

- `passing`: This rule, which is also used for intra-prediction, takes the input to the prediction module, and stores them in the appropriate storage elements. When the input indicates that a new macroblock is to be decoded using inter-prediction, it activates the `interSendReq` and `interReceiveResp` rules. It also calculates and outputs the final result, which is the prediction added to the residual calculated by the inverse transform module.

- `interSendReq`: This rule sends the requests to the memory to load the information from the blocks above the current macroblock.

- `interReceiveResp`: This rule takes the responses from the memory, and stores the values inside the registers. When all the data has been received, it activates the `interProcessStep` rule.

- `interProcessStep`: This rule is the one that calculates the motion vectors for the current macroblock. It iterates over the sub-blocks, calculates the motion vectors for each of them, and stores the results in the register file.

- `interIPProcessStep`: This rule gives the interpolator submodule the reference index, the motion vector, and the current location, so that the interpolator can calculate the inter-prediction result.

- `interOutputTransfer`: This rule takes the results from the interpolator submodule, and enqueues them into the FIFO for the prediction result.

5.5.5 Inter-Prediction Implementation - Interpolator Submodule

This module takes the reference frame index, the motion vector, and the current location, from the prediction module. Using this information, it obtains the necessary samples from the reference frame, and calculates the inter-predicted result.

In the frame buffer, four neighboring samples are concatenated and stored together. This module also performs four interpolation operations in parallel to increase performance. This makes the logic more complicated, since some unnecessary samples would be loaded along with the relevant ones.

Important State Elements

- `workFile`: As mentioned before, sometimes the 6-tap filtering has to be applied twice, once horizontally and once vertically. A register file is used to hold the intermediate result between the two filtering steps.
- `resultFile`: A register file is used to hold the final result of the interpolation.
- `workVector8` and `workVector15`: Two registers of vectors are used to hold temporary values that have to accessed at the same time to perform the 6-tap filtering. It is also used to re-concatenate the relevant samples together.

Rules

- `loadLuma`: This rule takes the input from the prediction module, and sends the requests for the necessary samples to the buffer control module.
- `loadChroma`: Same as above, but as the name indicates, it is the version for the chroma. Since the interpolation process is different for the luma and chroma samples, the memory access pattern is also different.
- `workLuma`: This rule takes the response from the buffer control module, and performs the actual interpolation process. Often it has to go through two stages to finish the interpolation process.

- `workChroma`: Same as above, but as the name indicates, it is the version for the chroma interpolation process.

- `outputing`: As the register file that holds the result is being filled up, this rule takes the completed parts, and outputs it to the prediction module. It is a separate rule, because the order that the final result is generated in is not always the same as the order in which they need to be outputted.

5.5.6 Inter-Prediction Implementation - Buffer Control Module

This module handles all communication to the frame buffer. It contains submodules that keep the long-term and short-term pic lists, and it also keeps track of which sections of the frame buffer correspond to each of the reference pics, and which sections are free. Using this information, it redirects the requests from the interpolator module to the right address in the frame buffer.

This module is also in charge of writing the decoded frame into the frame buffer, and reading the frame buffer for the final output of the H.264 module. However, those functionalities will not be discussed here.

Important State Elements and Submodules

- `shortTermPicList`: A short-term pic list submodule, which contains the list of short-term reference pictures, and contains methods for updating it as required.

- `longTermPicList`: A long-term pic list submodule, which contains the list of long-term reference pictures, and contains methods for updating it as required.

- `freeSlots`: A submodule that keeps track of the free slots in the frame buffer.

- `refPicList`: A register file that is used to hold the reference picture list.

Relevant Rules

- `inputing`: This rule takes the input to the module, which includes the decoded frame data, and the information about how the reference picture list should be constructed. It calls the methods of the short-term and long-term pic list modules when it receives information on the operations that should be done to each pic list.

- `initingRefPicList`: This rule constructs the default ordering of the reference picture list from the short-term pic list and the long-term pic list when it is activated by the `inputing` rule.

- `reorderingRefPicList`: This rule reorders the reference picture list when instructed by the `inputing` rule.

- `interLumaReq`: This rule takes the requests from the interpolator module and, based on the reference picture list, redirects the requests to the right address in the frame buffer.

- `interChromaReq`: Same as above, but for the requests for the chroma samples.

- `interResp`: This rule takes the response from the frame buffer, and sends it to the interpolator.

5.6 Deblocking Filter

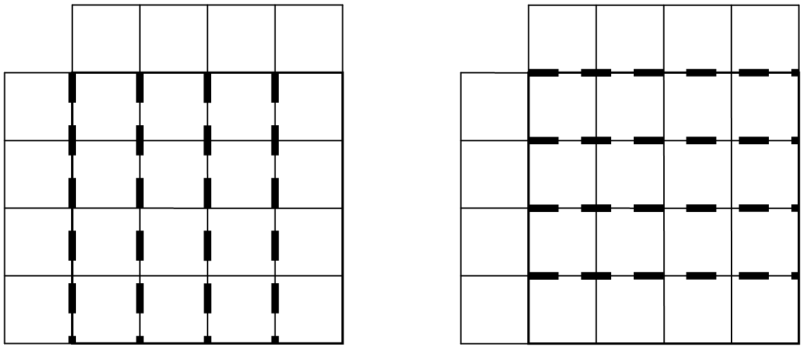


Figure 5-5: The edges for the deblocking filter.

Figure 5-5 shows a macroblock to be deblocked, 4 4x4 blocks to the left, and 4 4x4 blocks to the top.

The deblocking filter first deblocks the vertical edges between 4x4 blocks, going from left to right. These edges are marked with thick dotted line on the left half of Figure 5-5. The deblocking filter is applied to a horizontal row of 8 samples at a time, with 4 of those samples on the left side of the edge, and 4 on the right side. Therefore, each of the edges requires 16 applications of the deblocking filter, but these 16 applications can be done in parallel.

After all the vertical edges are done, the deblocking filter deblocks the horizontal edges, going from top to bottom. These edges are shown on the right half of Figure 5-5. The same process is applied, just with a different direction.

Each application of the deblocking filter includes a threshold test on the 8 samples. The actual filtering is only done if the test decides that it would not produce a harmful effect. When filtering is applied, samples on both sides of the edge get modified. Therefore, the deblocking filter process of a macroblock actually changes the sample values of the 4x4 blocks to the left and above the macroblock.

Each application of the deblocking filter depends on some parameters produced by the inverse transform and prediction process. These parameters determines the strength of the filter needed, and the threshold level for the test.

5.6.1 Deblocking Filter Implementation

The deblocking filter module performs the filtering with the parameters produced by the inverse transform and prediction modules. It currently uses many registers for storing the samples in the macroblock. If area is a concern, an alternative version using SRAM register files is possible.

Important State Elements

- Registers storing the current state and progress of the module.

- Various Registers and a register file holding the parameters.

- `workVector`, `leftVector`, and `topVector`: Registers holding the samples in the macroblock, and the necessary 4x4 blocks to the left and above the current macroblock. In the case of the blocks above the current macroblock, the values in the registers are loaded from the memory module.

Memory Usage Once again, 4x4 blocks above the current macroblock have to be stored in memory. The same memory module implementation used by intra-prediction and inter-prediction is used here, except with a different parameter. For a maximum frame width of 2048, 16KB of space is needed for this memory module.

Relevant Rules

- `passing`: This rule takes the input to the deblocking filter module, and stores them in the appropriate storage elements. When the data to be deblocked arrives, it activates the `initialize` rule.

- `initialize`: This rule initializes some state elements, and activates the `dataSendReq`, `dataReceiveResp`, and `horizontal` rules.

- `dataSendReq`: This rule sends the requests to the memory to load the 4x4 blocks above the current macroblock.

- `dataReceiveResp`: This rule takes the responses from the memory, and stores the samples inside the registers.

- `horizontal`: This rule applies the deblocking filter to the vertical edges. The name of the rule might be confusing, but it was named this way because each application of the deblocking filter works horizontally.
  When the rule is done, it activates the `vertical` rule and the `outputing` rule.

- `vertical`: This rule waits till the `dataReceiveResp` finishes, and then applies the deblocking filter to the horizontal edges.

- `outputing`: This rule outputs the results that are ready. It also stores the bottom 4x4 blocks to the memory module.

- `cleanup`: This rule deals with the special cases at the bottom of a frame, where the bottom 4x4 blocks should be outputted instead of stored to the memory module.

Chapter 6

Verification

Since video compression is a lossy process in general, the decoder cannot be verified against the uncompressed video given to the encoder. To verify the result, the output must be compared against the output of a reference code, which is assumed to be correct.

The libavcodec implementation of H.264 in the FFmpeg project, which was written in C, was used for verification purposes. It is also used to check whether the output of the individual modules in the Bluespec decoder. This is done by finding the places in the C code that correspond to the outputs of the hardware functional 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.

The Bluespec code is then verified by running a simulation of the hardware, and writing outputs of the functional blocks into a file using the `$display` statements. These values are tagged the same way that the outputs of the C code is tagged. A Perl script is then used 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.

Checking the output of each module makes it simple to localize the bugs, as it is easy to see which functional block started giving faulty outputs, and what the tags were for those outputs. The files could also be opened and compared manually for debugging, which is more efficient than using the waveforms when extra debugging outputs are added.

All the designs given in the next chapter have been verified to produce bit-perfect output for the benchmark video clips used.

Chapter 7

Result and Design Exploration

One of the advantages that Bluespec SystemVerilog has over traditional hardware programming languages is the fact that design exploration can be made easily in Bluespec. This chapter details modifications tried against the reference design explained in Chapter 5, and discusses the effects of the changes.

All of the critical path and area numbers used in this chapter are generated by Synopsys Design Compiler and Cadence Encounter using the Tower 180nm libraries. None of the design changes make any changes to the memory modules, so they are not included in the area numbers. However, the area number of the modules taken into account can be reduced by using memory generators for the large FIFOs and register files.

7.1 Initial Benchmark Result

The operation of the H.264 implementation was simulated using a video clip which contains 20 frames at 720p (1280x720 resolution). The simulation took about 56.3 million cycles. Since the synthesis result gives a clock frequency of around 80MHz, this means that the initial decoder is capable of decoding approximately 29 frames per second at 720p.

Currently, 720p is the most popular resolution used for HD video. For most videos, 30 frames per second is sufficient, but some videos are up to 60 frames per second.

The goal of the design explorations was to achieve over 60 frames per second for 720p videos. Since some videos may be harder to decode than benchmark clip used, the decoder should be able to decode the benchmark clip at more than 70 or 80 frames per second to be safe.

1080p, the highest resolution for HD video, at 30 frames per second takes roughly the same amount of computational power as 720p at 60 frames per second. Therefore, if the decoder is able to decode 720p at 60 frames per second, it should be able to decode 1080p at 30 frames per second as well.

7.2 Design Exploration - Part 1

The first group of design explorations were actually made before the H.264 decoder was completed. Therefore, these explorations were done assuming that the H.264 system includes only the NAL unit unwrapper plus the parser and entropy decoder.

7.2.1 Design Exploration 1A: Special Output for Consecutive Zeros

The transformed residual data that the CAVLC outputs often contain many consecutive zeros. In the original code, the entropy decoder just outputs each of the zeros separately. This makes the interface between the entropy decoder module and the next module a little bit simpler. However, it wastes many cycles and also wastes buffer space in the output FIFO.

We modified the entropy decoder unit to output the number of consecutive zeros instead in those cases. The output was tagged differently, so that the next module is able to differentiate between the two cases. Since the CAVLC coding also takes advantage of the fact that there are often consecutive zeros, this is actually a more natural way to write the code. This caused both an area and performance improvement in the design.

While this slightly complicates the inverse transform module, the performance of the inverse transform module is improved. This is because the inverse quantization step can be skipped for all the coefficients with value zero.

7.2.2 Design Exploration 1B: Two-Stage Exp-Golomb Function

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 take up to 65 bits in the encoded data.

The original version of the code 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 entropy decoder.

This change split the Exp-Golomb function into two versions. One version 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. It was thought that this change might increase the number of cycles needed to decode the same file, but the savings in critical path would outweigh 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.

7.2.3 Design Exploration 1C: Two Stage Buffering

The implementation of the entropy decoder module consists of a rule that fills the buffer, and the parser rule that takes the data from the buffer. These rules cannot fire in the same cycle.

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

The modification involves adding 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 size of the main buffer 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.

7.2.4 Benchmark Results

The operation of the NAL unit unwrapper and the entropy decoder was simulated using three clips taken from three different video files. One had 5 frames at 176x144 resolution, the second, 15 frames at 176x144 resolution, and the last had 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	AB added	AC added	all 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
postroute critical path (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^2)	337757	328339	281980	368960	293235

As expected, the design change 1A increases performance dramatically and also decreases area a little bit. Since it is obviously beneficial, the other changes were not tested by themselves.

Adding the design change 1B 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 1C 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. It's possible that the synthesis tools made some different decisions in area versus performance tradeoffs in these two cases.

All three of the changes made in this section are included in the following sections, as they all seem to have a positive effect overall.

7.3 Design Exploration - Part 2

At first, the FIFOs that form the main pipeline from the entropy decoder to the deblocking filter passes one sample for each FIFO element. However, since some of the parameters passed through the same FIFOs are much larger than the data samples, this is a waste of space as the FIFO must accomodate the largest possible data size. Not only that, the low sample throughput of the FIFOs was a bottleneck of the system.

This second group of design explorations changes the FIFOs to hold four samples per FIFO element, and also some related optimizations. Unlike the first group, these explorations were done after the completion of the decoder.

7.3.1 Design Exploration 2A: Four Samples for Each FIFO Element

This changes the main FIFO pipeline to hold four data samples per FIFO element. This makes more efficient use of the space in the FIFO, and also matches the 32 bits per second throughput expected from the frame buffer.

The inverse transform module's output stage is changed to output 4 sample at a time to match the FIFO element sizes, while the inter-prediction and deblocking filter already processes the data in 4-sample chunks. The intra-prediction, however, still outputs one sample at a time.

7.3.2 Design Exploration 2B: Four Samples Per Cycle for Intra-Prediction

With the change made above, the only major process that still processes one sample at a time is the intra-prediction. Since the intra-prediction wastes a relatively small number of cycles doing the preparations, this does not automatically make the intra-prediction the bottleneck of the decoder. Nevertheless, making the intra-prediction output 4 samples at a time is worth a try.

This exploration changes the `intraProcessStep` rule of the prediction module to process 4 samples at a time. The FIFO holding the predicted result and the `passing` rule have already been modified by the previous change, so the `intraProcessStep` rule is the only major modification needed.

Since this change adds parallelism to the intra-prediction process, the logic is likely to become more complicated, increasing the area of the module. Therefore, unlike the previous modification, this is not necessarily desirable.

7.3.3 Design Exploration 2C: Pipelined Inverse Transform

While output stage of the inverse transform module now takes only a quarter of the time to complete, the inverse transform module still works on only one process at a time. Since the prediction and deblocking filter modules have higher maximum throughput, the inverse transform module is still a potential bottleneck.

This optimization changes the inverse transform module to a 3-stage pipelined version. The first stage of the pipeline contains all the original processes except the regular transform and the output processes. The second stage of the pipeline is the regular transform process, and the third stage of the pipeline is the output process. While the first stage still seems to contain a lot of processes, most of them do not have to be done very often, and most of the cycles are spent on the scaling process.

Since extra storage is needed for the extra stages in the pipeline, the area of the inverse transform module is likely to go up with this change.

7.3.4 Benchmark Results

The entire decoder is simulated using the same test files used for the first group of design explorations. The simulation results and the post-route area and timing numbers are shown in the following table.

Code Version	Original	A added	AB added	AC added	all added
# cycles for the 1st clip	343712	215264	205280	213504	200000
# cycles for the 2nd clip	850944	438752	426912	439008	422720
# cycles for the 3rd clip	1237952	730144	681376	728352	663584
# cycles total	2432608	1384160	1313568	1380864	1286304
postroute critical path (ns)	11.815	14.533	12.319	14.507	11.867
total time (ms)	28.766	20.116	16.182	20.032	15.265
postroute area (um^2)	5442983	5323451	5381765	5508256	5452896

First, the critical path in all cases were in the deblocking filter module. Therefore, it is likely that the variations there was simply the result of random decisions made by the synthesis tool. The area numbers also seem to be affected by the random choices, but explorations B and C increased the area as expected.

For the performance, where the most important factor should be the total cycle count, the first design change (A) gives the biggest improvement as expected. The other changes also improves performance, but at the cost of larger area.

All of these changes are included in the next section.

7.4 Design Exploration - Part 3

Part 2A of the design explorations made the usage of the FIFOs more efficient, but some of the FIFOs still might not be large enough. If the FIFOs do not provide enough buffering, then the modules may have to wait for each other unnecessarily, increasing the number of cycles needed to process the video.

However, there are many FIFOs throughout the system, and trying to find the optimal combination of sizes for the FIFOs would take a very long time. Therefore, a reasonable set of sizes is chosen based on how the modules operate.

7.4.1 Benchmark Results

The entire decoder is again simulated using the same test files. The simulation results and the post-route area and timing numbers are shown in the following table.

Code Version	Original	Large FIFO
# cycles for the 1st clip	200000	177088
# cycles for the 2nd clip	422720	356640
# cycles for the 3rd clip	663584	571136
# cycles total	1286304	1104864
postroute critical path (ns)	11.867	11.822
total time (ms)	15.265	13.062
postroute area (um^2)	5452896	6043994

As expected, the larger FIFOs reduced the number of cycles needed while increasing the area. This change is included for the next section.

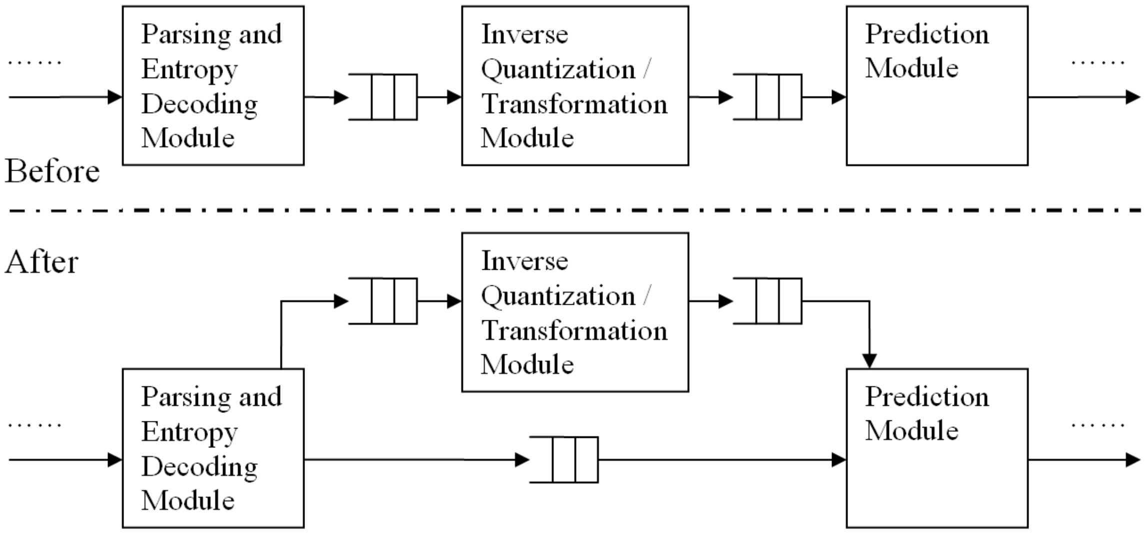
7.5 Design Exploration - Part 4

Unlike the intra-prediction process where the processing of each block may depend on the result of the previous blocks, the inter-prediction of several blocks can be done in parallel. There is still a dependency in the calculation of the motion vectors, but the motion vector calculation is not as complex as the interpolation process, so it is not a big issue. That said, unless the frame buffer can process multiple read requests per cycle, having several interpolation processes in parallel is not going to be very helpful.

At this point, the prediction module does not have any extra storage for the residual data from the inverse transform module, so the input FIFO of the prediction module acts as the buffer for the residual data. This means that the motion vector information for the next macroblock is blocked by the residual data that is still waiting for the corresponding prediction result, and the inter-prediction of a macroblock is completely finished before the processing of the next macroblock starts. Since the interpolator submodule is pipelined, this is not very efficient.

This group of design explorations deals with these issues, and increases the parallelism in the inter-prediction process.

7.5.1 Design Exploration 4A: Inter-Predicting Macroblocks in Parallel



Before
Parsing and Entropy Decoding Module
Inverse Quantization / Transformation Module
Prediction Module
After
Parsing and Entropy Decoding Module
Inverse Quantization / Transformation Module
Prediction Module

Figure 7-1: Change made to the top level module structure.

This change allows the inter-prediction process of the next macroblock to start before the current macroblock is finished. It also makes it easy to have multiple macroblocks interpolated in parallel, but as mentioned above, this would only be helpful if the frame buffer can process the read requests fast enough.

This could be done by adding extra buffering in the prediction module for the residual data, but that would not be a very clean way of doing it. We chose an alternative method and changed the module structure as shown in Figure 7-1, and that is how the code is modified.

7.5.2 Design Exploration 4B: Extra Pipeline for the Interpolator Submodule

This change to the code splits the `workLuma` and `workChroma` rules of the interpolator submodule into two pipeline stages. While this change can be made independently, the longer pipeline that results should make it more worthwhile in combination with the change described above.

Both the `workLuma` and `workChroma` rules are split into two rules, one for each pipeline stage, and extra rules are added for advancing the pipeline. While the throughput of the interpolator should increase with this change, the amount of storage needed in the module is also roughly doubled.

7.5.3 Benchmark Results

Once again, the entire decoder is simulated using the same test files. The simulation results and the post-route area and timing numbers are shown in the following table.

Code Version	Original	4A added	both added
# cycles for the 1st clip	177088	169952	158624
# cycles for the 2nd clip	356640	348896	347232
# cycles for the 3rd clip	571136	556128	536224
# cycles total	1104864	1074976	1042080
postroute critical path (ns)	11.822	11.726	13.141
total time (ms)	13.062	12.605	13.694
postroute area (um^2)	6043994	6092505	6875576

While design change 4A increases the area a little, it also increases the performance and gives the code more flexibility for future explorations.

Surprisingly, the overall effects of design change 4B is not nearly as positive. The critical path is still in the deblocking filter, so the longer critical path is likely the result of random decisions made by the synthesis tool. However, even if this is the case, the design change does not improve performance as much as expected, while it does increase the area significantly.

The trace outputs produced by the simulations revealed that there may be rule conflicts or other rule scheduling issues keeping the new design from giving the expected improvement in performance. Once the issues are fixed, the performance should be further improved.

7.6 Final Result

The same video clip used for the initial benchmark result was used to evaluate the performance of the design after incorporating all the design changes. The simulation took 21.3 million cycles. The clock frequency is still around 80MHz, so the decoder can now decode approximately 75 frames per second.

Chapter 8

Future Work

Several further optimization are possible. As mentioned in the previous chapter, the interpolation process could be further optimized by getting rid of rule scheduling issues. The output of the entropy decoder to the inverse transform module could be further compressed. The deblocking filter module could also be modified to use register files instead of registers. More importantly, large register files and FIFOs throughout the decoder can be modified to use SRAMs by using memory generators, which should reduce the area of the decoder significantly.

Bibliography

[1] Arvind, Rishiyur S. Nikhil, Daniel L. Rosenband, and Nirav Dave. High-level Synthesis: An Essential Ingredient for Designing Complex ASICs. In *Proceedings of ICCAD'04*, San Jose, CA, 2004.

[2] Nirav Dave. Designing a Reorder Buffer in Bluespec. In *Proceedings of MEMOCODE'04*, San Diego, CA, 2004.

[3] Nirav Dave, Man Cheuk Ng, and Arvind. Automatic Synthesis of Cache-Coherence Protocol Processors Using Bluespec. In *Proceedings of MEMOCODE'05*, Verona, Italy, 2005.

[4] FFmpeg Project. `http://ffmpeg.mplayerhq.hu/`.

[5] K.R. Rao Soon-kak Kwon, A. Tamhankar. Overview of H.264 / MPEG-4 Part 10. *Journal of Visual Communication and Image Representation*, 17(2):186–216, April 2006.

[6] International Telecommunication Union. Recommendation ITU-T H.264: Advanced Video Coding for Generic Audiovisual Services. ITU-T, 2003.