H.264 Encoder Design

Maxine Lee, Alex Moore - Group 3

May 17, 2006

1 Introduction

H.264 is everywhere. If you have ever downloaded videos to your iPod from the iTunes movie store or ripped a home movie to the latest and greatest MP4 video format, you have taken advantage of H.264 encoding. H.264 is a giant step forward from previous standards because it achieves greater compression, provides a full networking framework, and streamlines the algorithms to achieve similar quality with much simpler mathematics. In addition, it is an ITU-recognized standard, so it is non-proprietary and open to everyone.

At this point, storage space and network traffic conditions are still the major bottlenecks in video encoding, so there should be a market for hardware H.264 encoders that can perform compression on video without needing a full-size computer and access to a power outlet. H.264 is well-suited to implementation in hardware because unlike previous standards, which required large numbers of multiplication operations and exponential calculations, virtually all of the calculations in H.264 can be done with adders and shifters. In this project, we will begin the implementation of an H.264 encoder by writing an intraframe prediction system in Bluespec.

The major blocks in intraframe prediction are listed below:

- Discrete cosine transform (DCT)
- Quantization
- Inverse quantization
- Inverse DCT
- Prediction Mode Select
- Prediction Calculation

Of these blocks, the most challenging will probably be the prediction mode select block, which determines the optimal prediction mode to minimize the sum of absolute differences (SATD) between

the predicted block and the actual image block. We expect that the most significant blocks in terms of size and speed will be the transformation, although they may be smaller than expected because of the optimization of the standard to use integers.

Our baseline implementation only handles intraframe prediction, and we delve into simple interprediction for design exploration. Still, in H.264 terminology, intraframe prediction mode alone is all we need to predict "I-Slices," and this form of prediction (coupled with the remainder of the spec!) is sufficient to generate H.264-compliant video streams.

2 Background

H.264 is a standard for high-quality video encoding that is resilient to poor network conditions, yet high-quality enough to serve as a basis for HDTV and HD-DVD encoding. The standard defines only the decoder, but it is relatively straightforward to derive an encoder specification from the standard.

H.264 has a number of important advantages that make it the most important video standard in the technology world. First, it uses more sophisticated transforms, like quantization, Hadamard, and discrete cosine transforms to achieve top-quality compression. These operations are focused on taking advantage of what digital computation does best - integer shifts and addition. As such, even though it is computationally intensive work, the blocks will be small and can capitalize on processor multimedia features. Equally important, H.264 was designed to compensate for lossy networks. It includes its own network layer to facilitate streaming video (with lost packets) and to minimize the amount of transfer that needs to be completed. Perhaps most significantly, H.264 is an open standard, produced by the ITU. It is a loose enough standard to allow for improvement, and there have been five significant versions of the reference software and documents that have been released to capitalize on improvements since H.264's acceptance in 2003. As such, we consider working on a hardware version of the encoder a relevant and valuable project.

We chose to implement the intraframe encoding block of the H.264 encoder. The video stream begins its trip through the encoder when the video is split into single-image frames, then sliced into 16x16 pixel macroblocks. This slicing algorithm is complex and aims to minimize discrepancy within a macroblock and around its borders. These macroblocks need not be continuous, although they need to be in the same frame. These macroblocks are then passed into intra- and interframe prediction blocks. Predictions are made based on the macroblock's appearance in previous frames and on the neighboring pixels within the same frame. Once the best prediction mode has been found, the difference between the prediction mode and the image is found, transformed, quantized, and entropy encoded. At that point, the entropy-coded changes in the macroblocks are sent to the network layer, which processes and packages the data for transport over a potentially lossy network.

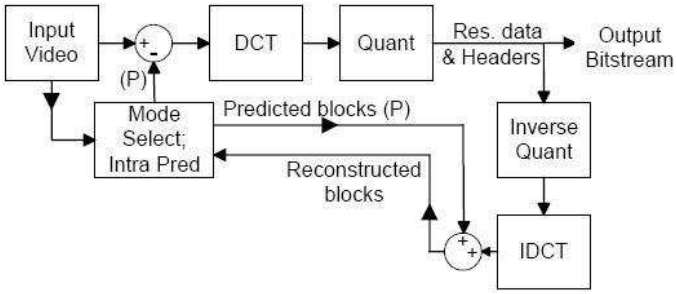

Figure 1: Encoder Block Diagram for Intra-Prediction Only [1]

We choose to implement the prediction features because we found them to be the most interesting components of the specification. Fraught with controls, the system must evaluate and make decisions about the predictions that will minimize transmission over 300 times per frame, resulting in hundreds of thousands of calculations per video clip. These blocks are mathematically intensive, and should contain most of the "heavy lifting" involved in encoding. As such, they are perfect targets to farm out to a specific, optimized chip. We also considered this slice of the H.264 encoder to be a good choice because it is a very modular block. Since the specification for H.264 is only defined in what the decoder must support, it is possible to consider anything from our baseline intraframe predictor all the way through a sophisticated weighted-frame interframe predictor sufficient to meet the standards for H.264 video.

3 High Level Design

The block diagram for the H.264 encoder using only intraprediction is shown in Figure 1. Our baseline architecture will only support intraprediction, since interprediction is considerably more complex. The three main building blocks are the DCT, Quant, and Intra-Prediction blocks, which will each be discussed below. The purpose of the inverse blocks is to perform the same steps the decoder will perform, and therefore base encoding decisions on how accurate the decoding process will be. For the most part, the inverse operation is conceptually and structurally similar to the forward operation, so further discussion on the inverse will be minimal.

3.1 Intra-Prediction

Intra-prediction utilizes spatial correlation in each frame to reduce the amount of transmission data necessary to represent the picture. H.264 performs intraprediction on two different sized blocks: 16x16 (the entire macroblock) and 4x4. 16x16 prediction is generally chosen for areas of the picture that are smooth. 4x4 prediction, on the other hand, is useful for predicting more detailed sections of the frame. The general idea is to predict a block, whether it be a 4x4 or 16x16 block, based on

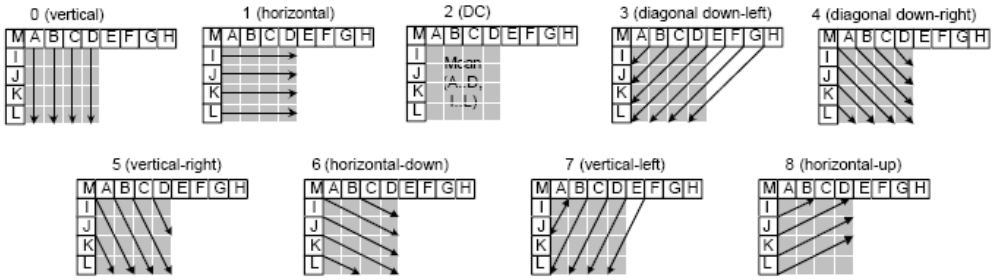

Figure 2: Prediction modes for a 4x4 pixel block [2]

$$\mathbf{H} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 2 & 1 & -1 & -2 \\ 1 & -1 & -1 & 1 \\ 1 & -2 & 2 & -1 \end{bmatrix} \quad \tilde{\mathbf{H}}_{\text{inv}} = \begin{bmatrix} 1 & 1 & 1 & 1/2 \\ 1 & 1/2 & -1 & -1 \\ 1 & -1/2 & -1 & 1 \\ 1 & -1 & 1 & -1/2 \end{bmatrix}$$

Figure 3: (a) Transform Matrix (left) and (b) its Inverse (right) [3]

surrounding pixels using a mode that results in a prediction that most closely resembles the actual pixels in that block. There are nine 4x4 prediction modes, shown in Figure 2, and four 16x16 modes. The four 16x16 modes are similar to modes 0, 1, 2, and a combination of modes 3 and 8 of the 4x4 modes.

The intra-prediction block is a computationally intensive block. Making a prediction decision for a single 4x4 block requires: 1) making a prediction for each pixel for each mode, as shown in Figure 2, 2) computing the cost of using each prediction method, which includes calculating the sum of the differences (or sum of the hadamard transformed differences) between every prediction and actual pixel value, and 3) choosing the smallest cost as the correct prediction mode for that input block. 16x16 prediction is similar.

3.2 DCT

The DCT block takes in a 4x4 prediction residual and reduces the amount of redundancy by applying a transformation. The inverse DCT block, naturally, gets the necessary information back.

The transformation used is a 4x4 integer transform that has all of the essential properties of the complex 8x8 DCT used by previous standards. The matrices used for the transformation and inverse transformation are shown in Figure 3. Since the inverse transform is also integer, this transformation has the added benefit of having no encoder/decoder mismatch.

The final output, Y, of the DCT block, given input X, is $Y = HXH^T$.

3.3 Quantization

This block first scales each transformed coefficient by a predefined value. It then quantizes each value for transmission. There are 52 different quantization levels possible, specified by the quantization parameter (QP). An increase in QP by six doubles the quantization step size, which doubles the compression. Thus, this block directly determines the compression versus quality tradeoff.

3.4 Assumptions and Limitations

Due to the short time frame of this project, it was necessary for us to limit the scope of the project. Even implementing a full-blown intra-predictor would be far too much to handle. Thus, the following assumptions and limitations are made:

- We are only worrying about luma, the grayscale component. Prediction on the chroma (color) component is a simpler rehash of luma prediction, so it adds no real insight.
- We assume that the video frames have a width and height that are multiples of 16, so that the picture can be partitioned into an integer number of macroblocks. The macroblock partitioning is fixed: the first macroblock is the 16x16 block on the top left corner of the frame. Thus, we assume it is always possible for the predictor to move along a column, and then down to the next row, until the entire frame is predicted. Also for toolflow and chip size reasons, our default frame size is 64x48 (4 macroblocks x 3 macroblocks), which is much smaller than the IPOD's 320x240 resolution.
- The standard also specifies intraframe coding for 8x8 blocks, and the reference software uses 8x8 blocks rather than 4x4. We chose 4x4 prediction, because there is far more literature on it.

4 Testing Strategy

Our Test Harness has a two-stage modular design to make it easy to add additional tests. First, we test each component with a unit test suite that ensures that the block is functioning correctly for basic conditions. This design makes it easy to change the internals of the blocks and quickly verify that they still function.

We also have an infrastructure for testing a random set of vectors to promote better coverage. Where reference code is available and easy to extract, we drive deterministic blocks with the same input and verify that the output is the same. We plan to integrate at least one qualitative, human-readable test to provide functional, demonstrative verification of our project, although we were unable to complete these tests before the project deadline.

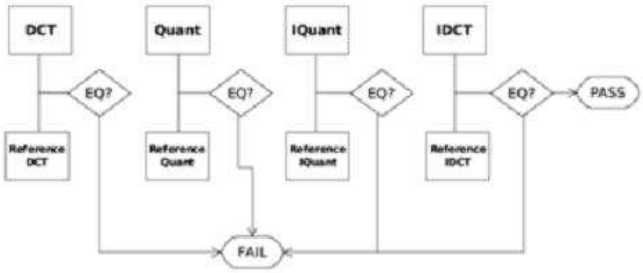

Figure 4: Test Harness Flowchart

4.1 Basic Tests

Our microarchitecture allows us to extract and test each value of a 4x4 luma matrix as it passes through each block of our test bench. Our first set of tests took sample matrices from papers that described the DCT and Quantization blocks and verified that our hardware produces the same values as the mathematical operations done by hand on this matrix. These tests provide coverage to ensure that changes in the code do not make these blocks nonfunctional. Similarly for the prediction blocks, our most primitive tests involve hand-calculations of sum-of-error and predicted matrices using each of the possible modes.

These basic tests do not provide sufficient coverage to ensure that the blocks are functional across input cases. Additionally, since the same mathematical tutorials that provided the algorithm provides the hand-calculated values of our examples, some other form of testing is needed. We have extracted pieces of reference code from both the x264 encoder and the JVT reference software from ITU, based on how modular each section appears. We then compiled that code to provide a comparison after each block to verify that our system is operating as intended.

Beyond these specific problem cases, we also have the capability to generate completely random test vectors and drive them through both our reference implementation and our Bluespec implementation of our predictor. Our test harness provides the ability to read in these vectors and compare the output of each block with the output from the reference code block. This methodology allows us to very quickly drive our system with almost complete coverage without the need to manually work through test cases. We were unable to build all the tests we wished, but with a completed test bench, the framework has the potential to be very powerful.

4.2 Additional Tests

While functional verification is good, it is difficult to develop an intuition that the system is working without tests that allow people to see the behavior of the system. To that end, we have two plans to demonstrate the functionality of our system in a more visible manner than a set of [Passed] messages

displayed.

First, we have created a series of YUV macroblock images that cater to the strengths of different prediction modes. From these, we extracted the luma component and pushed it through our predictor. While time constraints prevented us from verifying that the predictor behaved as expected, the architecture exists for quick implementation of such tests. We feel that they would be very valuable to any group interested in continuing the project.

4.3 Trace Output

In much the same vein, there is an extensive set of trace output that can be enabled at compile-time to show exactly the behavior of the system. Since time constraints prevented us from building unit tests to verify the behavior of the complex control logic in the predictor blocks, it was crucial that we could trace the behavior of the system to verify that it behaves logically.

This information is also our best indicator that the predictor is functional. The simulation data shows the prediction blocks executing and calculating costs for each mode, then passing along the best residual for eventual transformation and quantization. The trace output shows cycle times, which are a critical performance benchmark.

A sample of the trace code is shown below.

```
34946 : mbPickMB\_intra4  : START prediction on 4x4 block : Row = 3, Column = 3
34970 : mkPickMB         : Send 4x4 block to reconstruct -- for 4x4 Intra Prediction
34972 : mkDct            : Applying DCT on single 4x4-predicted block
34973 : mkQuant          : Applying Forward Quantization on single 4x4-predicted block
34976 : mkQuant          : Applying Reverse Quantization single 4x4-predicted block
34977 : mkIDct           : Applying Inverse DCT on block
34979 : mkPickMB         : Got reconstructed 4x4 block
34980 : mkPickMB\_intra4  : FINISHED Intra 4x4 prediction : cost =        2048
34981 : mkPickMB         : FINISHED predicting macroblock.  Row =  2, Col =  0
34981 : mkPickMB         : Intra 4x4 prediction selected.  Cost = 0       2048
34983 : mkDct            : Applying DCT on block of Intra 4x4 Predicted MB
34984 : mkQuant          : Applying Forward Quantization on block of Intra 4x4 Predicted MB
```

5 Microarchitecture of Baseline Design

Figure 5 depicts the top level hardware design of the encoder illustrated in Figure 1. The Picture Parsing block feeds the next macroblock to be processed into the Choose Macroblock Mode block (This block we left unimplemented, opting to let the testbench or software pass on the next macroblock). The Choose Macroblock Mode block must determine what boundary pixels are available

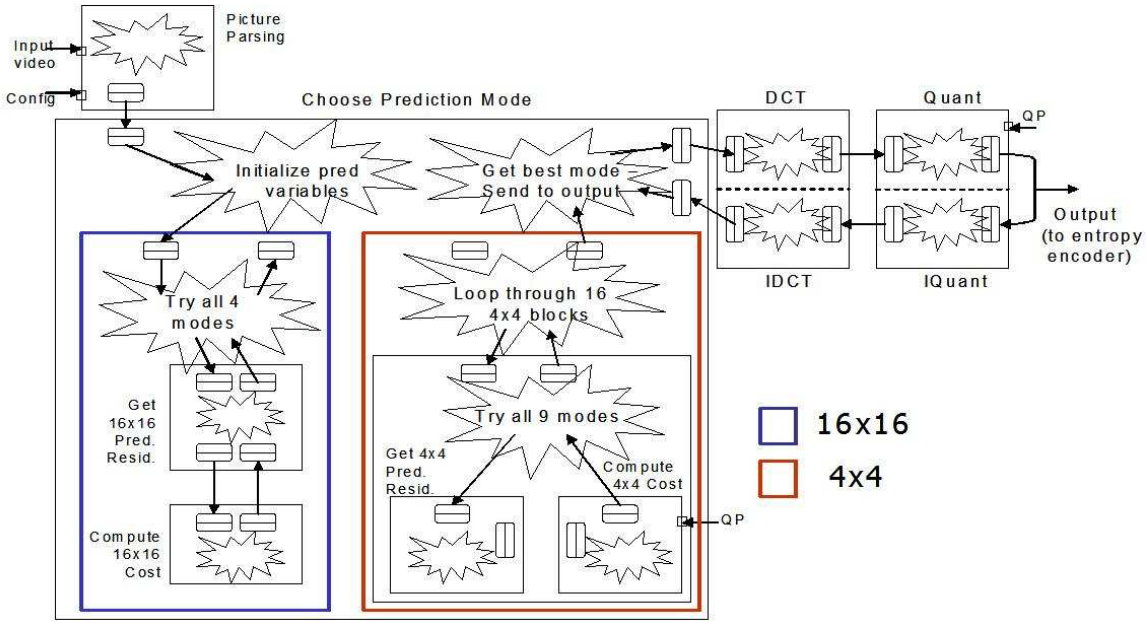

Figure 5: Overall block diagram for H.264 encoder for intra-frame prediction

based on the location of the macroblock in the frame. It then starts both 16x16 and 4x4 prediction on the current macroblock, which are completed by the two colored boxes in Figure 5.

Each of these blocks rely on the computation of their respective Get Prediction Residual and Compute Cost to generate a quantitative cost for which to base its mode selection. (The 4x4 predictor has an additional submodule to loop through each of the 16 4x4 blocks.) As can be inferred by the name of the block, Get x Prediction Residual calculates the predicted block, and then the prediction residual, based on an input mode. The Compute x Cost uses this prediction residual to calculate the cost of using that prediction mode. Finally, the 4x4 and 16x16 predictors return their best residuals to the Choose Macroblock Mode. The Choose Macroblock Mode block picks a prediction method and sends appropriate prediction residual through the DCT and Quantization blocks for eventual entropy coding (which is not in the scope of this project), and back through the inverse path to use for predicting the next macroblock.

In the following sections, we break down the discussion of Figure 5 into the same three parts we did in Section 3: DCT, Quantization, and Prediction. Also like Section 3, we will not discuss IDCT and IQuant, because of the similarity to their forward counterparts. Prediction will be further broken down into the various prediction control systems.

5.1 DCT

The hardware implementation of the matrices shown in Figure 3 is pictured in Figure 6. This is the basic building block for the DCT transformation. Each of these building blocks requires eight

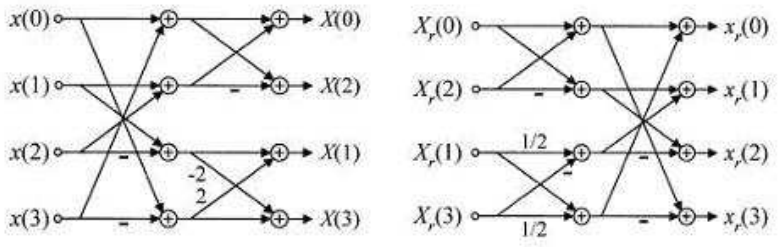

Figure 6: Implementation of a) H matrix and b) IH matrix

adders/subtractors and two shifters (a left shifter for the DCT and a right shifter for the IDCT). To complete the transformation HXH^T, we need eight such building blocks, four for the horizontal transform and four for the vertical transform, as shown and described in Figure 7.

Since the path is strictly combinational in the baseline design, the data flow is relatively straightforward. Each each transformation is completed in a single cycle.

Figure 8 shows the internal workings of the DCT block. A second transform using a hadamard matrix is required for the DC component of 16x16 prediction modes. This adds one additional clock cycle. As input, the DCT block needs a 4x4 prediction residual and the prediction mode (16x16 or 4x4) to complete the operation. Its output is similar: a 4x4 transformed residual and the current prediction mode, both of which are needed in the next step, quantization. Furthermore, its FIFOs must also carry extra information required by the entropy coder for encoding the residual, such as the most_probable_mode flag for 4x4 intra-prediction (more on this later).

As a final remark, the DCT block, as well as the remainder of the reconstruction chain (Quant, IQuant, and IDCT) must be able to distinguish between a 4x4 block that is part of a macroblock and one that is a standalone block being used be 4x4 intraprediction. The former must be sent to the entropy encoder and saved in the frame buffer/boundary buffer while the latter should just pass through the chain and go back to the 4x4 intra-prediction block.

5.2 Quantization

In hardware, the quantization process, shown in Figure 9, is most easily implemented using multiple lookup tables that are indexed using the quantization parameter (QP). This is what is done in the H.264 reference software. The three parameters are generally called MF for multiplication factor, f, and $qbits$, and are key in the basic quantization equation: $(MF * pixel + f) >> qbits$. (The equation changes slightly in some cases, but always has the same structure : multiplication, followed

Figure 7: Combinational Transform Circuit using either H or IH for forward and inverse DCT, respectively. The input 4x4 block (on the far left) is first horizontally transformed, and its output (the 4x4 block in the middle) is then vertically transformed to give the 4x4 output on the far right.

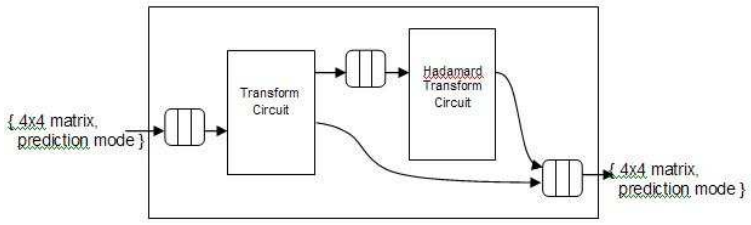

Figure 8: DCT architecture. Top path shows 16x16 prediction mode operation (which still operates on 4x4 blocks), and bottom path shows 4x4 prediction mode operation

by addition, followed by a shift) MF especially is not easily computed from scratch since the value depends not only on QP, but also on the location of each pixel in the 4x4 block. The other two parameters, $f = 2^{qbits}/3$ and $qbits = QP/6 + 15$, may also be directly implemented in hardware (i.e. without a lookup table). Our baseline implements the actual division and causes the encoder to stall for thousands of clock cycles. Thus, instead of two lookup tables, as shown in Figure 9, we actually have two looping rules that compute the division. Even though the delay is large, it may not matter, because QP is a relatively unchanging variable. In the reference software, it is set at compiletime. Thus, the direct calculation is undoubtedly the better approach in terms of area savings.

Figure 9 only shows the flow for a single bit in the 4x4 block. Since there are 16 bits in the block, 16 adders, 16 multipliers, and 16 shifters are required for this operation. Similar to the DCT process, the input to the Quantization block include the 4x4 block of pixels to be operated on and the mode that determines the specific operation. It outputs the same types, with the 4x4 block now being the quantized block. It also passes the extra information bits as discussed in the previous section.

5.3 Intra-Prediction

Most of the interesting control will fall into the category of intra-prediction. This block contains a large amount of control logic and data manipulation. The control system is in charge of iterating through all of the possible prediction methods to choose the best one. The data manipulation part

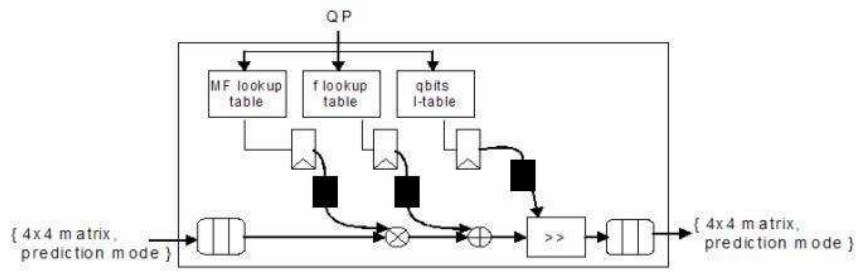

Figure 9: A simplified diagram of the quantization architecture. The black boxes show that the function parameters depend on the prediction mode.

includes creating the prediction block and determining the cost associated with that block.

In the following sections, we begin by discussing the top level predictor block. Then we continue with the 16x16 intraprediction, which is simpler and faster than the 4x4 case. In the 4x4 intraprediction section, we will place particular emphasis on its subtle control issues. Finally, we will touch on the data manipulation aspects of the intra-predictor.

5.3.1 Top Level Predictor

The main logic for the top level predictor involves keeping track of where the current macroblock is located in the frame. This is particularly important for determining what pixels are available for prediction. For example, the first macroblock in the frame (i.e. top left macroblock) has no information to do prediction, but a block in the center of the picture has all its boundary pixels available for prediction.

The input to the predictor is a single macroblock, so the top level must store all the data involved with processing it. In order to determine the location of the current macroblock, the predictor must know the width and the height of the picture, which is stored in a configuration file. It uses a row and column counter to store the current state, and increments these counters appropriately when it accepts a new input macroblock. Based on the row and column counter, it can easily determine which boundary pixels are available and valid for use in predicting the current macroblock.

In addition to keeping track of the location, the predictor must also have all of the necessary boundary pixels from reconstructed macroblocks available. The left boundary of the current macroblock is taken from the macroblock that just finished the prediction process, so its pixels can easily be saved in a 16-entry register. The boundary pixels above the current macroblock, however, were predicted several macroblocks ago. Therefore, a register file of 16-entry vectors is required to store all of the boundary pixels needed by macroblocks later in the frame. The size of the register file is directly proportional to the width of the frame, which is one of the reasons for our decision to assume a small picture.

Once the top level prediction modules takes an input macroblock, determines which boundary pixels are available, and determines which pixels to extract from the register file, it sends a request to the 16x16- and 4x4-predictor blocks via their input FIFOs and waits for their response. These blocks respond with their best prediction mode(s) and the associated cost from using that mode. After both blocks have this information ready, the predictor makes a mode decision. It chooses 4x4 prediction if $Cost_{16x16}/2 > Cost_{4x4} + 24 * \lambda(QP)$ (See 5.3.4 for explanation of cost function). From this equation, we can see that the predictor favors 16x16 prediction due to the added overhead of 4x4 prediction. The residual is then sent through the reconstruction chain for output and reconstruction.

When the reconstructed data comes back to the top level block, the pixels that will be used for future prediction are saved in the register file. At this point, the request can be dequeued,

signaling that the predictor is ready to process a new macroblock. Note that macroblocks cannot be processed in parallel, since previous macroblocks must be processed in order to process the current one. Therefore, the strategy of dequeuing the request FIFO only at the completion of the request is utilized throughout all of the blocks to reduce the number of extra registers needed.

5.3.2 Control System for 16x16 Blocks

The job of the 16x16-predictor is loop through the four prediction modes and choose the one that results in the lowest transmission cost. The 16x16-predictor has a simple counter to keep track of which mode it is currently trying to process. Based on which boundary pixels are available, the 16x16 predictor must determine whether the current mode should be attempted. If the mode is valid, it requests the prediction and cost from its submodules. When the response returns, the predictor saves the residual and boundary prediction pixels if the mode resulted in the best cost so far. (Both the residual and prediction pixels are needed to reconstruct the boundary pixels later)

This module has a very large hardware requirement, since it must have a register that saves the best macroblock residual.

5.3.3 Control System for 4x4 Blocks

The 4x4 block is significantly more complex than the 16x16 control system. This is because the 4x4-predictor functions like the top level predictor. It has to loop between all 16 4x4 blocks the same way that the top level predictor must loop through all the macroblocks in the frame. Thus, it has all the same boundary-storage requirements. Because of this complexity, the intra-predictor has a submodule (See Figure 5), similar to the 16x16 predictor, that loops through all nine 4x4 prediction modes and chooses the best one.

The predictor for 4x4 blocks is quite a bit more complicated than the top level module. The boundary pixels may be in neighboring macroblocks, they may be in the current macroblock, or they may be unavailable. The boundary availability thus depends not only on the location of the current macroblock, but also on the location of the current 4x4 block.

As will be explained in the following section, the cost of using a particular mode is partly a function of the mode selected for the 4x4 block above and to the left of the current block. What happens if we are on the top or left edge of the current macroblock? The 4x4-predictor must save both the boundary pixels and the boundary modes in order to operate properly.

5.3.4 Generating the Prediction and Cost

An important part of the intra-frame predictor is generating the predicted matrix and computing the associated cost. The general idea for obtaining the predicted matrix is pictured in Figure 10. Other than modes one and two, the remainder of the modes utilize a weighted average of two or

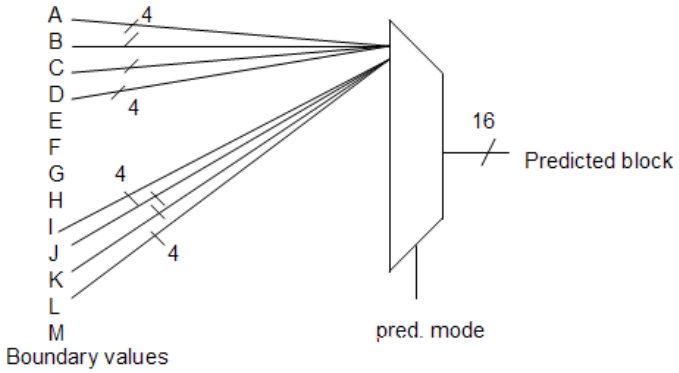

Figure 10: Generating the predicted block based on the prediction mode. A-M are the boundary pixels, and are labeled according to Figure 2

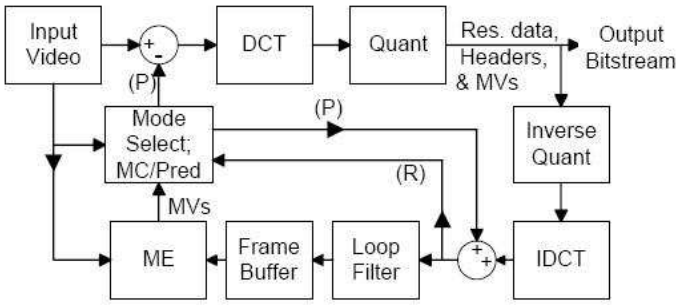

Figure 11: Encoder Block Diagram for both Inter- and Intra-Prediction [1]

three boundary pixels to form the prediction. This weighted average can easily be implemented by shifters (to give a higher weight to a pixel and to do the final division) and adders.

Once we have the predicted matrix, we must compute the cost of the prediction. This requires, in the simplest case, finding the sum of the absolute error (SAE). To get the SAD, we use 16 subtractors followed by a 16-input (or equivalent) adder. Finally, to get the cost, the simplest acceptable equation we can implement is $cost = SAD + 4 * P * \lambda(QP)$, where P is 0 if the mode is the same mode as its neighbor, else one, and lambda is some exponential function of QP. This is logical, since the effect of the 4x4 prediction overhead is greater when the compression is greater. The four in the equation comes from the four extra bits required to send the prediction mode.

6 Design Space Exploration: Interframe prediction

Inter-prediction requires adding three blocks, and modifying one, to our original block diagram (see Figure 11). Each of these blocks are described below.

The loop filter eliminates the boundaries between each 4x4 block, so the picture will be less pixilated. The loop filter is very complicated (and not very interesting), so we chose to leave it out

of our predictor. Doing so should not significantly affect the performance of the rest of the system, and under the given time constraints, we decided that our time should be better spent implementing the rest of the blocks.

The frame buffer stores previous frames to use for motion estimation. In the reference software, the frame buffer size is a parameter that is set before the encoding of a video. Our design will fix the size of the buffer to one frame, which should store only the previous frame. This design choice was made to reduce the amount of memory needed and the amount of data to search through. With our set frame size of 64x48, the frame buffer size must be at least 24 Kbits.

The ME, or motion estimation block, determines the best relationship, in terms of a motion vector, between the current picture and the ones stored in the frame buffer. The motion vector in H.264 has a resolution of 1/4 of a pixel. Values between pixels are interpolated based on six nearby pixels. Additionally, each macroblock can be split up into sub-blocks, much like in intraprediction. In our implementation, we chose to limit our resolution to single-pixel, which is the more interesting case. In the H.264 specification, a macroblock may be split into 16x16, 16x8s, 8x16s, or four 8x8. If 8x8 is chosen, this block can further be split into 8x8, 4x8s, 8x4s, or four 4x4s. Thus, if the macroblock is split into 4x4s, the entire block will have 16 independent motion vectors. Since the algorithm requires exhaustive searching to find the best mode, having this many options will be far too complicated for our design and require far too much time to complete. Thus, for consistency with intraprediction, we limited the block sizes to 16x16 or 4x4.

6.1 Microarchitecture

We implemented this section as the following architecture:

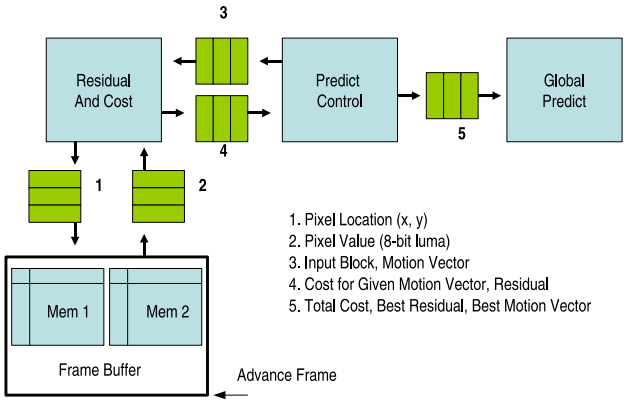

Figure 12: Microarchitecture for Interframe Prediction Block

We were able to reuse a number of the blocks from our intraframe predictor in our microarchi-

tecture. In this case, the cost function block remained identical to the one in interframe prediction, since the sum of absolute error is the same. By changing the number of bits needed to transmit control and adjusting the algorithm to generate a prediction based on a previous frame instead of boundary pixels, we were able to make small modifications to the Residual and Cost blocks. The only sections that required significant changes were the top-level interframe prediction block and the frame buffer, which needed to be written completely from scratch.

We chose to implement the frame buffer using the Tower memory generators, since the amount of memory needed to store a realistic-sized frame would be so far beyond the size of the register files we have placed and routed automatically (about a 50x size increase!) that it made sense to capitalize on external tools.

6.2 Algorithm

Since there will be over 300,000 possible motion vectors that need to be checked for a single frame, we chose to implement the motion estimation block with a more sophisticated algorithm. We chose a hexagonal algorithm that reduces the number of motion vectors that must be checked by over 90%. In exchange for the extra computational power, simulation results from the authors of the algorithm report an increase in mean residual error of about 30%. Since the mean residual error is so close to zero, however, the additional inaccuracy is not significant compared to the time savings in encoding. Implementing sub-pixel resolution will also improve this error rate.

The algorithm works by assuming that the best motion vector will be near the current block. It starts by trying a motion vector of (0,0), that is, the exact same macroblock on the previous frame will be used as the prediction for the macroblock in the current frame. It then checks all motion vectors in a hexagonal pattern, so it will try a motion vector of (2,0), (1,2), (-1,2), (-2,0), (-1, -2), and (1, -2). It will predict each pixel as the pixel in the previous frame that is shifted by the location of the motion vector, so for a pixel at (140, 140) and a motion vector of (2,0), the predicted pixel value will the the value of the pixel (142, 140) in the previous frame.

Once the best nearby motion vector has been found, the algorithm assumes it is the new best motion vector, and repeats the hexagonal search around that block. Once the best motion vector is the center vector, it is calculated to be the best, then its cost is sent to the top-level predictor block to compare with intraframe values.

6.3 Top-Level Changes

The prediction block now has more options to choose from. Not only must the predictor decide between the nine 4x4 modes and four 16x16 modes, it must determine whether inter- or intra-prediction is more suitable. The top-level prediction block sends the results from whichever prediction mode delivers the best results to the DCT block. It also packages the control bits needed to specify the

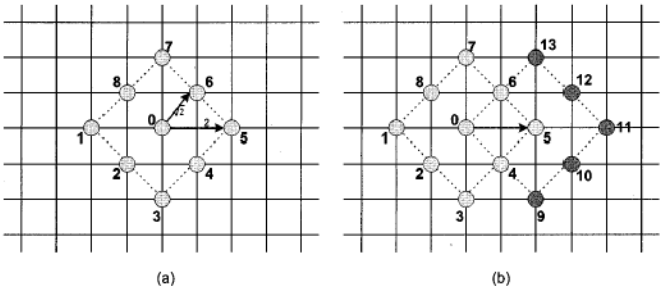

Figure 13: Hexagon-based Motion Estimation

prediction type (inter or intraframe), the most-probable mode flag, and the needed intraframe mode or interframe motion vector.

The modified top-level microarchitecture diagram is below:

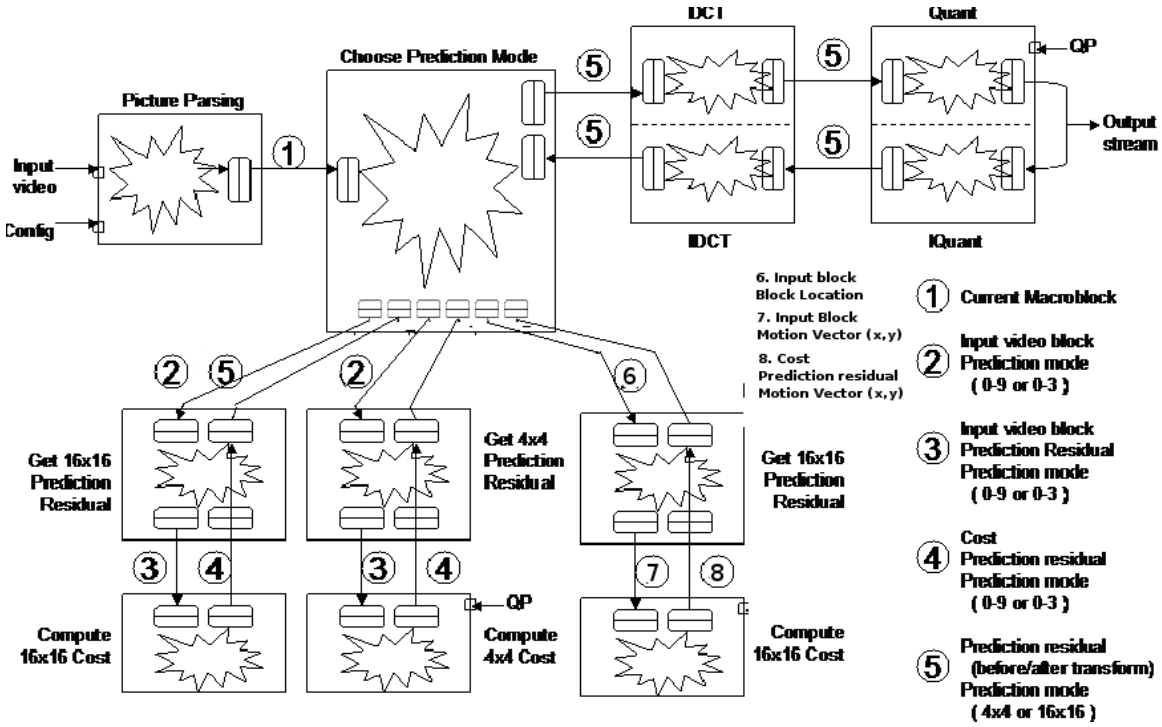

Figure 14: Top-level microarchitecture for both Inter- and Intra-Prediction

As previously stated, by extending our baseline intraframe predictor to do inter-frame prediction, we expected to trade off area and complexity for a significant improvement in the amount of data needed to transmit video. With only intra-frame prediction, H.264 can compress video only a very small amount. With interframe prediction, even on just a single frame, we would expect a dramatic difference in the amount of compression achieved. Of course, interframe prediction is far more complex than intraframe, so we expected an equally dramatic difference in the amount of circuitry required and a potentially significant impact on throughput.

We were unable to determine hard numbers for the performance of the new blocks. Due to the number of bugs and issues that remained in our baseline processor code at the time the interprediction block was forked, the full system is not yet integrated. While we have area and timing information, we were not able to produce simulation data that would allow us to compare reduction in transmitted bits and number of additional clock cycles for interframe prediction. We tested the blocks with unit tests for the frame buffer, and only by tracing output with the interframe predictor alone. It looks as if there are bugs remaining in the interframe block from the trace output, and we would suggest that any group trying to expand on the project begin by fixing the integration issues.

Were we able to get performance results, it would have been difficult to fully determine the amount of reduction in transmitted bits from inter-frame encoding, since we did not implement the entropy coding, the control system that translates encoding decisions into a minimum number of bits (i.e. which block in the previous frame to base the current prediction on), or the network layer. We should, however, be able to make reasonable guesses for entropy coding using standard compression utilities like gzip. Similarly, we can roughly estimate the amount of data needed to transmit our encoder's mode decisions at a per-transition level. For example, we know that it requires three additional bits to transmit a change in intraframe prediction mode from the mode used for neighboring blocks. If we can count the number of these transitions, we should be able to get a value that is in the ballpark.

7 Results

7.1 Area

When looking at our Area results, one fact jumps out – this is one big honkin chip with a lot of complex parts. It is so complex, in fact, that the Bluespec compiler cannot unfold it beyond module boundaries because of a lack of memory space.

More frustrating, because of the same limitations and the scale of the chip, placing and routing were not possible. Encounter requires more RAM than any normal workstation possesses to place and route a chip as large as this one. In an ideal situation, we would have plenty of time to attempt floorplanning the chip to try to constrain the area calculations enough to allow encounter to finish. As it stands, time constraints forced us to spend our efforts in other areas.

7.1.1 Baseline

The (slightly optimized for area) baseline chip weighs in approximately 20 times the size of the SMIPS processor from Lab 3. The bulk of the area comes from having to push huge amounts of data through all of our blocks. Several modules require macroblock interfaces, which each contain 128 8-bit pixels. That is equivalent to moving around two SMIPS register files! These synthesis

numbers represent a significant decrease as well, from areas near 600,000 unitless area units before removing excess registers, reducing the size of FIFOs, and other streamlining improvements.

Table 1: Area results for baseline predictor

Cycle Time	7.23ns
Total Area	406,500
Predictor	59%
DCT	7%
Quant	21%
Misc	13%

7.1.2 Interframe Prediction

We followed the same toolflow for the expanded predictor, including interframe prediction. As the baseline chip changed due to bugs and improvements, comparisons are most fairly made between the expanded predictor and the comparison predictor, from which the expanded predictor was built. The comparison predictor came in significantly smaller, though it is distinctly lacking in functionality – it makes mistakes and is prone to stalls.

Table 2: Area results for predictor including interframe mode

Cycle Time	4.80ns
Comparison Time	4.89ns
Total Area	440,485
Comparison Area	356,346
Interframe	19%
Increase in size	24%

The interframe predictor adds 24% area, 19% due to the predictor itself, and 5% due to the extra control logic needed to make comparisons between the interframe and intraframe blocks. This block is significantly smaller than intraframe prediction, which is curious, since it is more computationally intensive. Why? Because the interframe prediciton block only encompasses 16x16 prediction at this time. Additionally, the boundary pixel vectors needed for intraframe predictors are included in the area calculations, whereas the data storage for interframe prediction is not included. Finally, the cost functions are exactly the same as in intraframe prediction, so we can reuse those blocks.

These area results do not include information about the Frame Buffer. As the Frame Buffer consists mostly of two memory cells generated using the Tower Memory Generator tools, it does not appear in the synthesis area numbers. We were unable to gather place and route numbers due to the size of the chip and the limitations of our tools.

Based on placing and routing the memory alone, we would anticipate a Frame Buffer that works with 4x3 macroblock cells to require approximately 19,000 additional area units. To expand this

frame buffer to account for 320x240 resolution pictures, it would account for 475,000 area units, more than doubling the size of the chip. This constraint would probably result in an ideal system that uses an off-chip memory. Since the spacial requirements change slowly, we believe it would be possible to load the window around the predicted block from memory at runtime.

7.2 Timing

It is interesting to note the large clock period discrepancy between the two tables in the previous section. The slow clock period reported by the baseline implementation is due to the multiplication in the quantization block. The remainder of the design is implemented using mainly shifters and adders, which should fairly easily meet the timing constraint of 5ns. The faster clock period reported by the second iteration indicates that they synthesis tools managed to find a better implementation of the multiplication routine, probably giving up some area in the process.

7.3 Intraframe Prediction Latency

All in all, 6714 clock cycles are required to complete the prediction of a 64x48 frame. With a clock rate of 7.23ns, the Design Compiler reported clock period, the total time to completion is 48.5us. Extrapolating these numbers to the standard IPod frame size, we conclude that each 320x240 IPod frame takes approximately 1.2ms to make a prediction.

As shown in Figure 15, 4x4 prediction has a much higher latency than 16x16 prediction in all cases. 4x4 prediction takes over 500 cycles while 16x16 takes 152 cycles at most. This is partly because each 4x4 block must be sent through the reconstruction chain before the next 4x4 block can begin prediction. A 4x4 block cannot begin prediction until 13 clock cycles after the previous 4x4 block finished. Even though there are only four computation blocks in the chain, some cycles are used processing the block and ensuring that the modules know that this block is not part of the output. Furthermore, currently it takes the predictor submodule 21 cycles at most to predict nine modes, which could be greatly reduced with a better pipelining structure. With this change, 4x4 prediction latency could be reduced to only twice that of 16x16 prediction.

16x16 prediction is much more susceptible to changes due to what boundary pixels are available, since 16x16 prediction latency is directly proportional to the number of available prediction modes. 4x4 prediction, on the other hand, has nine blocks that are guaranteed to have all valid boundary pixels, so the location of the macroblock has little effect on the latency.

8 References

[1]Ghandi, M.M., and Ghanbari, M. The H.264 Video Coding Standard for the Next Generation Multimedia Communication. IAEEE Journal.

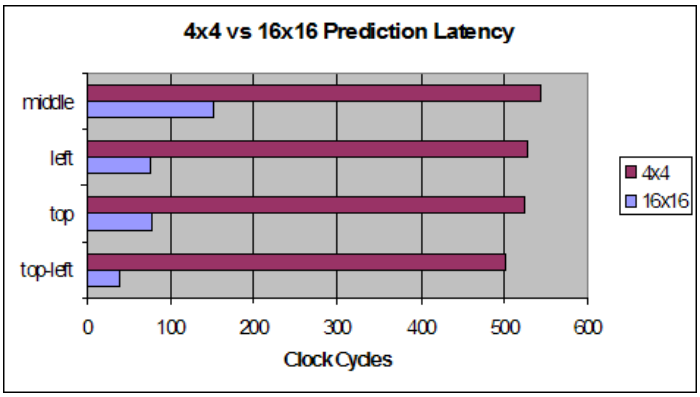

Figure 15: Number of clock cycles it takes to completely predict a macroblock based on its location in a frame.

[2]Richardson, I. H.264 / MPEG-4 Part 10 Tutorial www.vcodex.com.

[3]Malvar, H.S., Hallapuro, A., Karczewicz, M., and Kerofsky, L. Low-Complexity Transform and Quantization in H.264/AVC. IEEE Transactions on Circuits and Systems for Video Technology, July 2003.

[4]Zhu, C., Lin, X., and Chau, L. Hexagon-Based Search Pattern for Fast-Block Motion Estimation. IEEE Transactions on Circuits and Systems for Video Technology, May 2002.