# AN EFFICIENT H.264 INTRA FRAME CODER HARDWARE DESIGN

by
ESRA ŞAHİN

Submitted to the Graduate School of Engineering and Natural Sciences
in partial fulfillment of
the requirements for the degree of
Master of Science

Sabancı University
Spring 2006

AN EFFICIENT H.264 INTRA FRAME CODER HARDWARE DESIGN

APPROVED BY:

Assist. Prof. Dr. İlker Hamzaoğlu …….

(Thesis Supervisor)

Assist. Prof. Dr. Ayhan Bozkurt …….

Assist. Prof. Dr. Hasan Ateş …….

DATE OF APPROVAL: …….

AN EFFICIENT H.264 INTRA FRAME CODER HARDWARE DESIGN

Esra Şahin

EECS, Master Thesis, 2006

Thesis Supervisor: Assist. Prof. Dr. İlker Hamzaoğlu

## ABSTRACT

H.264 / MPEG-4 Part 10, a recently developed international standard for video compression, offers significantly better video compression efficiency than previous international standards. Since it is impossible to implement a real-time H.264 video coder using a state-of-the-art embedded processor alone, in this thesis, we developed an efficient FPGA-based H.264 intra frame coder hardware for real-time portable applications targeting level 2.0 of baseline profile.

We first designed a high performance and low cost hardware architecture for real-time implementation of entropy coding algorithms, context adaptive variable length coding and exp-golomb coding, used in H.264 video coding standard. The hardware is implemented in Verilog HDL and verified with RTL simulations using Mentor Graphics Modelsim. We then designed a high performance and low cost hardware architecture for real-time implementation of intra prediction algorithm used in H.264 video coding standard. This hardware is also implemented in Verilog HDL and verified with RTL simulations using Mentor Graphics Modelsim.

We then designed and implemented the top-level H.264 intra frame coder hardware. The hardware is implemented by integrating intra prediction, mode decision, transform-quant and entropy coding modules. The H.264 intra frame coder hardware is verified to be compliant with H.264 standard and it can code 35 CIF (352x288) frames per second. The hardware is first verified with RTL simulations using Mentor Graphics Modelsim. It is then verified to work at 71 MHz on a Xilinx Virtex II FPGA on an ARM Versatile Platform development board. The bitstream generated by the H.264 intra frame coder hardware for an input frame is successfully decoded by H.264 Joint Model (JM) reference software decoder and the decoded frame is displayed using a YUV Player tool for visual verification.

ETKİN BİR H.264 İNTRA ÇERÇEVE KODLAYICI DONANIM TASARIMI

Esra Şahin

EECS, Yüksek Lisans Tezi, 2006

Tez Danışmanı: Yard. Doç. Dr. İlker Hamzaoğlu

## ÖZET

Yakın tarihte geliştirilmiş uluslararası bir standart olan H.264 / MPEG4 Part 10, kendinden önceki standartlara göre belirgin şekilde daha iyi sıkıştırma verimi sunmaktadır. H.264 video kodlayıcısının son teknoloji gömülü işlemcilerle gerçek zamanlı uygulamasının imkansız olması nedeniyle bu tez çalışmasında taşınabilir uygulamalar için taban profilinin 2.0 düzeyini hedefleyen FPGA tabanlı H.264 intra çerçeve kodlayıcı donanımı geliştirilmiştir.

Öncelikle, H.264 standardında kullanılan entropy kodlamasi algoritması icin gerçek zamanlı çalışan yüksek performanslı ve düşük maliyetli bir donanım mimarisi Verilog HDL kullanılarak tasarlanmıştır. Tasarımın doğrulama işlemi Mentor Graphics Modelsim benzetim programı kullanılarak "RTL" benzetimleri ile gerçekleştirilmiştir. Daha sonra, H.264 standardında kullanılan intra tahmin algoritması için gerçek zamanlı çalışan düşük maliyetli bir donanım mimarisi tasarlanmıştır. Bu donanım da yine Verilog HDL kullanılarak gerçekleştirilmiş ve Mentor Graphics Modelsim benzetim programı kullanılarak "RTL" benzetimleri ile doğrulanmıştır.

Daha sonra, H.264 intra çerçeve kodlayıcı donanımı tasarlanmış ve gerçekleştirilmiştir. Donanım intra tahmin, moda karar verme, dönüşüm-nicemleme ve entropy kodlamasi modülleri entegre edilerek gerçekleştirilmiştir. Kodlayıcı donanımının H.264 standardıyla tamamen uyumlu olduğu doğrulanmıştır. Kodlayıcı saniyede 35 CIF (352x288) çerçevesini kodlayabilmektedir. Tasarım, Mentor Graphics Modelsim benzetim programı kullanılarak "RTL" benzetimleri ile doğrulanmıştır. Daha sonra, donanımın 71MHz hızla çalışması "ARM Versalite Platform" geliştirme ortamında doğrulanmıştır. H.264 intra çerçeve kodlayıcı donanımı ile bir girdi çerçeve için oluşturulan "bitstream" H.264 Joint Model (JM) şifre çözücü ile başarılı bir biçimde çözülmüş ve şifresi çözülen çerçeve "YUV Player" programı kullanılarak görsel anlamda doğrulanmıştır.

*To My Mother and Father,*
*and to My Sister Esin…*

## ACKNOWLEDGEMENTS

I would like to express my deep appreciation to my supervisor, Assist. Prof. Dr. Ilker Hamzaoglu, for his skills, enthusiasm, unconditional support, guidance and patience during the process of this thesis. I appreciate very much for his suggestions, detailed reviews and invaluable advices. I feel myself privileged as his student.

I also want to thank Assist. Prof. Dr.Yucel Altunbasak, who played an important role in initiating H.264 research project at Sabanci University; Assist. Prof. Dr. Hasan Ates, for his excellent help with particular issues; and Assist. Prof. Dr. Ayhan Bozkurt, who participiated in my thesis jury.

Thanks also my partners in H.264 research project, Ozgur Tasdizen, Mehmet Guney, and Sinan Yalcin, for valuable discussions and feedback throughout the project.

Special thanks due to my family. This thesis is dedicated with love and gratitude to my parents and my sister for their constant support and encouragement for going through my tough periods with me.

I am indebted to Yasin Erdogan, whose part in my own success is so vast I can not measure.

My warmest thanks go to Serkan Oktem for his friendship.

Finally, my acknowledgements go to Sabanci University for supporting our H.264 research project.

## TABLE OF CONTENTS

## LIST OF FIGURES

## LIST OF TABLES

## ABBREVIATIONS

| | |
|---|---|
| ARM | Advanced RISC Machines |
| ASIC | Application Specific Integrated Circuit |
| AVC | Advanced Video Coding |
| CABAC | Context-Adaptive Binary Arithmetic Coding |
| CAVLC | Context Adaptive Variable Length Coding |
| CIF | Common Intermediate Format |
| CODEC | Coder, Decoder Pair |
| CPU | Central Processing Unit |
| DFF | D Flip Flop |
| DSP | Digital Signal Processor |
| DVD | Digital Versatile Disc |
| FPGA | Field Programmable Gate Array |
| HDL | Hardware Description Language |
| ISDN | Integrated Services Digital Network |
| ISO/IEC | International Standards Organization, International Electrotechnical Commission |
| ITU-T | International Telecommunications Union, Telecommunications Standardization Sector |
| JVT | Joint Video Team |
| LCD | Liquid Crystal Display |
| MB | Macroblock |
| MPEG | Motion Picture Experts Group |
| NAL | Network Abstraction Layer |
| PSNR | Peak Signal Noise Ratio |
| PVT | Process Voltage Temperature |
| RAM | Random Access Memory |

| | |
|---|---|
| QCIF | Quadrature Common Intermediate Format |
| QP | Quantization Parameter |
| SAD | Sum of Absolute Difference |
| SATD | Sum of Absolute Transformed Difference |
| SRAM | Static Random Access Memory |
| UMC | United Microelectronic Corporation |
| VCL | Video Coding Layer |
| VGA | Video Graphics Array |
| VLC | Variable-Length Coding |

# CHAPTER 1

# INTRODUCTION

## 1.1 Motivation

Video compression systems are used in many commercial products, from consumer electronic devices such as digital camcorders, cellular phones to video teleconferencing systems. These applications make the video compression hardware devices an inevitable part of many commercial products. To improve the performance of the existing applications and to enable the applicability of video compression to new real-time applications, recently, a new international standard for video compression is developed. This new standard, offering significantly better video compression efficiency than previous video compression standards, is developed with the collaboration of ITU and ISO standardization organizations. Hence it is called with two different names, H.264 and MPEG4 Part 10.

H.264 video coding standard has a much higher coding efficiency potential (capable of saving up to %50 bit rate at the same level of video quality) than the previous standards. Due to its high coding efficiency and due to its flexibility and robustness to different communication environments, in the near future, H.264 is expected to be widely used in many applications such as digital TV, DVD, video transmisson in wireless networks, and video conferencing over the internet.

The video compression efficiency achieved in H.264 standard is not a result of any single feature but rather a combination of a number of encoding tools. The top-level block diagram of an H.264 Encoder is shown in Figure 1.1.

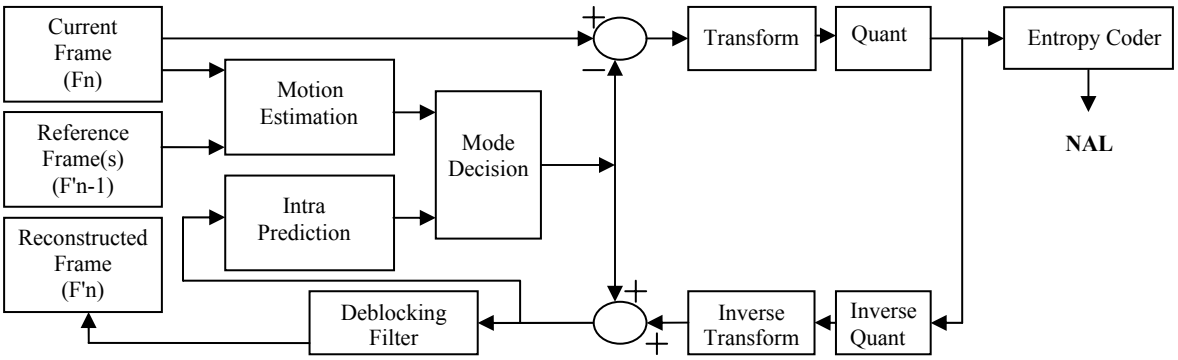



**Figure 1.1** H.264 Encoder Block Diagram

The H.264 standard includes a Video Coding Layer (VCL), which efficiently represents the video content, and a Network Abstraction Layer (NAL), which formats the VCL representation of the video and provides header information in a manner suitable for transportation by particular transport layers or storage media [1].

As shown in Figure 1.1, an H.264 encoder has a forward path and a reconstruction path. The forward path is used to encode a video frame by using intra and inter predictions and to create the bit stream. The reconstruction path is used to decode the encoded frame and to reconstruct the decoded frame. Since a decoder never gets original images, but rather works on the decoded frames, reconstruction path in the encoder ensures that both encoder and decoder use identical reference frames for intra and inter prediction. This avoids possible encoder – decoder mismatches [1, 3, 4].

Forward path starts with partitioning the input frame into MBs. Each MB is encoded in intra or inter mode depending on the mode decision. In both intra and inter modes, the current MB is predicted from the reconstructed frame. Intra mode generates the predicted MB based on spatial redundancy, whereas inter mode, generates the predicted MB based on temporal redundancy. Mode decision compares the required amount of bits to encode a MB and the quality of the decoded MB for both of these modes and chooses the mode with better quality and bit-rate performance. In either case, intra or inter mode, the predicted MB is subtracted from the current MB to generate the residual MB. Residual MB is transformed using 4x4 and 2x2 integer transforms. Transformed residual data is quantized and quantized transform coefficients are re-ordered in a zig-zag scan order. The reordered quantized transform coefficients are entropy encoded. The entropy-encoded coefficients together with

header information, such as MB prediction mode and quantization step size, form the compressed bit stream. The compressed bit stream is passed to network abstraction layer (NAL) for storage or transmission [1, 3, 4].

Reconstruction path begins with inverse quantization and inverse transform operations. The quantized transform coefficients are inverse quantized and inverse transformed to generate the reconstructed residual data. Since quantization is a lossy process, inverse quantized and inverse transformed coefficients are not identical to the original residual data. The reconstructed residual data are added to the predicted pixels in order to create the reconstructed frame. A deblocking filter is applied to reduce the effects of blocking artifacts in the reconstructed frame [1, 3, 4].

Since it is impossible to implement a real-time H.264 video coder using a state-of-the-art embedded processor alone, in this thesis, we developed an efficient FPGA-based H.264 intra frame coder hardware for real-time portable applications targeting level 2.0 of baseline profile.

H.264 intra frame coder is a competitive alternative to JPEG2000 for still image compression, in terms of both coding efficiency and encoder/decoder complexity [2, 11, 12, 13]. The rate-distortion performance of H.264 intra frame coder using CABAC and R-D optimized mode decision is about the same as that of JPEG2000 with default optimized settings. The use of CAVLC for entropy coding results in about 0.5 dB loss in coding efficiency, and with low-complexity mode decision (R-D optimization turned off) there is an additional loss which is less than 0.5 dB. Without R-D optimized mode decision, the computational complexity of the H.264 encoder is similar to that of JPEG2000 encoder; but the decoding complexity of H.264 is much lower than that of JPEG2000. Moreover, H.264 coding algorithm is easier to implement in hardware, due to its use of block-based processing, which reduces the memory requirements and allows for a pipelined approach.

Another application area for H.264 intra frame coder is in motion picture production, editing and archiving, where video frames are coded as I-frames only to allow for random access to each individual picture. For such applications, H.264 is shown to be superior to Motion-JPEG2000, especially at lower resolutions [2, 11, 12, 13].

The block diagram of the proposed H.264 intra frame coder hardware is shown in Figure 1.2.

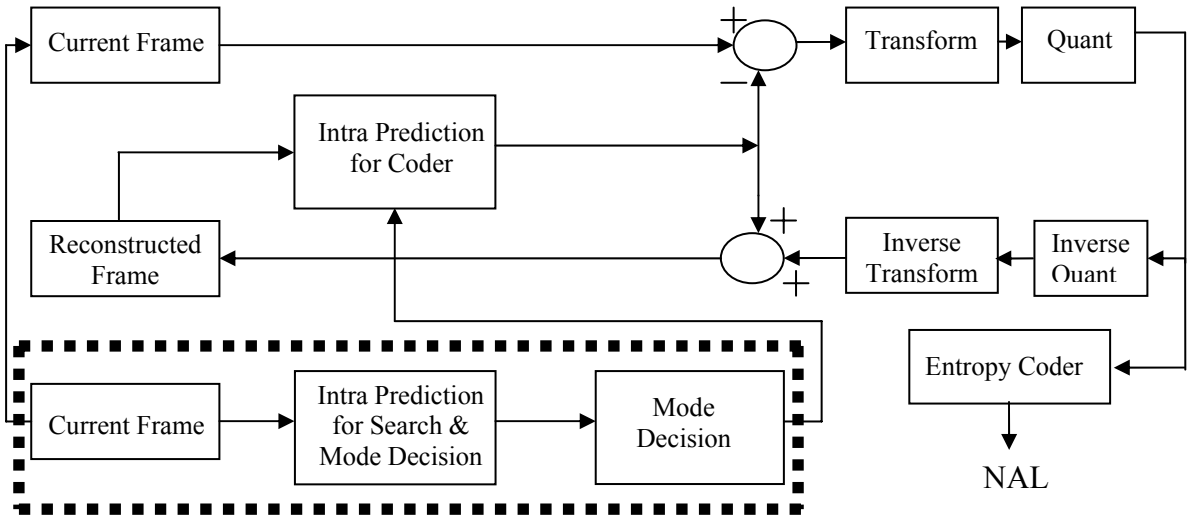



**Figure 1.2** H.264 Intra Frame Coder Block Diagram

The intra prediction algorithm used in H.264 reduces spatial redundancies by exploiting the spatial correlation between adjacent blocks in a given picture. Each picture is divided into 16×16 pixel MBs and each MB is composed of luma and chroma components. Intra prediction module predicts the pixels in a MB using the pixels in the available neighboring blocks. For the luma component of a MB, a 16x16 predicted luma block is formed either by performing intra predictions for each 4x4 luma block in the MB or by performing intra prediction for the 16x16 MB. There are nine prediction modes for each 4x4 luma block and four prediction modes for a 16x16 luma block. For the chroma components of a MB, a predicted 8x8 chroma block is formed for each 8x8 chroma component by performing intra prediction for the MB. There are four prediction modes for each chroma component. Additional information for intra prediction algorithm and the proposed hardware for intra prediction are given in Chapter 2.

Context Adaptive Variable Length Coding (CAVLC) algorithm encodes transformed and quantized residual luminance and chrominance data. CAVLC uses multiple tables for a syntax element. It adapts to the current context by selecting one of these tables for a given syntax element based on the already transmitted syntax elements. Information other than the residual data is coded using Exp-Golomb code words [3, 4, 5]. Additional information

for entropy coding algorithm and the proposed hardware for entropy coding are given in Chapter 3.

Transform algorithm is based on a 4x4 integer transform. The algorithm does not include any floating point operations; it only uses integer addition and binary shift operations. In this way, a possible drift between encoder and decoder is avoided. H.264 is the first standard to attain exact equality of decoded video content from all decoders [3, 4, 5]. Detailed information about transform algorithm and the proposed hardware for transform are given in [6].

The quantization algorithm uses a non-uniform quantizer. Quantization parameter can take a value between 0-51. The quantization step size doubles for an increment of 6 in quantization parameter. That means an increment of 1 in quantization parameter results in 12.2% increment in quantization step size. The quantization algorithm requires an integer multiplication [8, 9, 10]. Detailed information about quantization algorithm and the proposed hardware for quantization are given in [6].

The mode decision algorithm compares the 4x4 and 16x16 predictions and selects the best luma prediction mode for the MB. 4x4 prediction modes are generally selected for highly textured regions while 16x16 prediction modes are selected for flat regions. Each 8x8 chroma component of an intra coded MB is predicted from previously encoded and reconstructed chroma samples above and/or to the left and both chroma components always use the same prediction mode. The mode decision algorithm compares the 8x8 predictions and selects the best chroma prediction mode for the MB. Mode decision algorithm implemented in the proposed mode decision hardware is the same as the algorithm implemented in the JM Software when there is no Rate-Distortion optimization [4]. Additional information for mode decision algorithm and the proposed hardware for mode decision are given in Chapter 4.

## 1.2 Thesis Organization

The rest of the thesis is organized as follows.

Chapter 2 explains intra prediction hardware designed as part of H.264 intra frame coder hardware. First, it introduces intra prediction algorithm used in H.264 / MPEG4 Part 10 video coding standard. Then it describes the designed hardware in detail and the implementation results are given.

Chapter 3 explains entropy coding hardware designed as part of H.264 intra frame coder hardware. First, it introduces entropy coding algorithm used in H.264 / MPEG4 Part 10 video coding standard. Then it describes the designed hardware in detail and the implementation results are given.

Chapter 4 explains the top-level intra frame coder hardware. The modules used in the intra frame coder hardware and their scheduling are explained.

Chapter 5 presents the conclusions and the future work.

# CHAPTER 2

# HARDWARE ARCHITECURES FOR H.264 INTRA PREDICTION ALGORITHM

The video compression efficiency achieved in H.264 standard is not a result of any single feature but rather a combination of a number of encoding tools. As it is shown in the top-level block diagram of an H.264 encoder in Figure 1.1, one of these tools is the intra prediction algorithm used in the baseline profile of H.264 standard [3, 4, 5]. Intra prediction algorithm generates a prediction for a MB based on spatial redundancy. H.264 intra prediction algorithm achieves better coding results than the intra prediction algorithms used in the previous video compression standards. However, this coding gain comes with an increase in encoding complexity which makes it is an exciting challenge to have a real-time implementation of intra prediction for H.264 video coding.

## 2.1 H.264 Intra Prediction Algorithm Overview

Intra prediction algorithm predicts the pixels in a MB using the pixels in the available neighboring blocks. For the luma component of a MB, a 16x16 predicted luma block is formed by performing intra predictions for each 4x4 luma block in the MB and by performing intra prediction for the 16x16 MB. There are nine prediction modes for each 4x4 luma block and four prediction modes for a 16x16 luma block. A mode decision algorithm is then used to compare the 4x4 and 16x16 predictions and select the best luma

prediction mode for the MB. 4x4 prediction modes are generally selected for highly textured regions while 16x16 prediction modes are selected for flat regions.

There are nine 4x4 luma prediction modes designed in a directional manner. A 4x4 luma block consisting of the pixels a to p is shown in Figure 2.1. The pixels A to M belong to the neighboring blocks and are assumed to be already encoded and reconstructed and are therefore available in the encoder and decoder to generate a prediction for the current MB. Each 4x4 luma prediction mode generates 16 predicted pixel values using some or all of the neighboring pixels A to M as shown in Figure 2.2. The examples of each 4x4 luma prediction mode for real images are given in Figure 2.3. The arrows indicate the direction of prediction in each mode. The predicted pixels are calculated by a weighted average of the neighboring pixels A-M for each mode except Vertical and Horizontal modes.

The prediction equations used in each 4x4 luma prediction mode are shown in Figure 2.4 where [y,x] denotes the position of the pixel in a 4x4 block (the top left, top right, bottom left, and bottom right positions of a 4x4 block are denoted as [0, 0], [0, 3], [3, 0], and [3, 3], respectively) and pred[y,x] is the prediction for the pixel in the position [y,x].

| M | A | B | C | D | E | F | G | H |
|---|---|---|---|---|---|---|---|---|
| I | a | b | c | d | | | | |
| J | e | f | g | h | | | | |
| K | i | j | k | l | | | | |
| L | m | n | o | p | | | | |

**Figure 2.1** A 4x4 Luma Block and Neighboring Pixels

DC mode is always used regardless of the availability of the neighboring pixels. However, it is adopted based on which neighboring pixels A-M are available. If pixels E, F, G and H have not yet been encoded and reconstructed, the value of pixel D is copied to these positions and they are marked as available for DC mode. The other prediction modes can only be used if all of the required neighboring pixels are available [4, 5]. Available 4x4 luma prediction modes for a 4x4 luma block depending on the availability of the neighboring 4x4 luma blocks are given in Table 2.1.

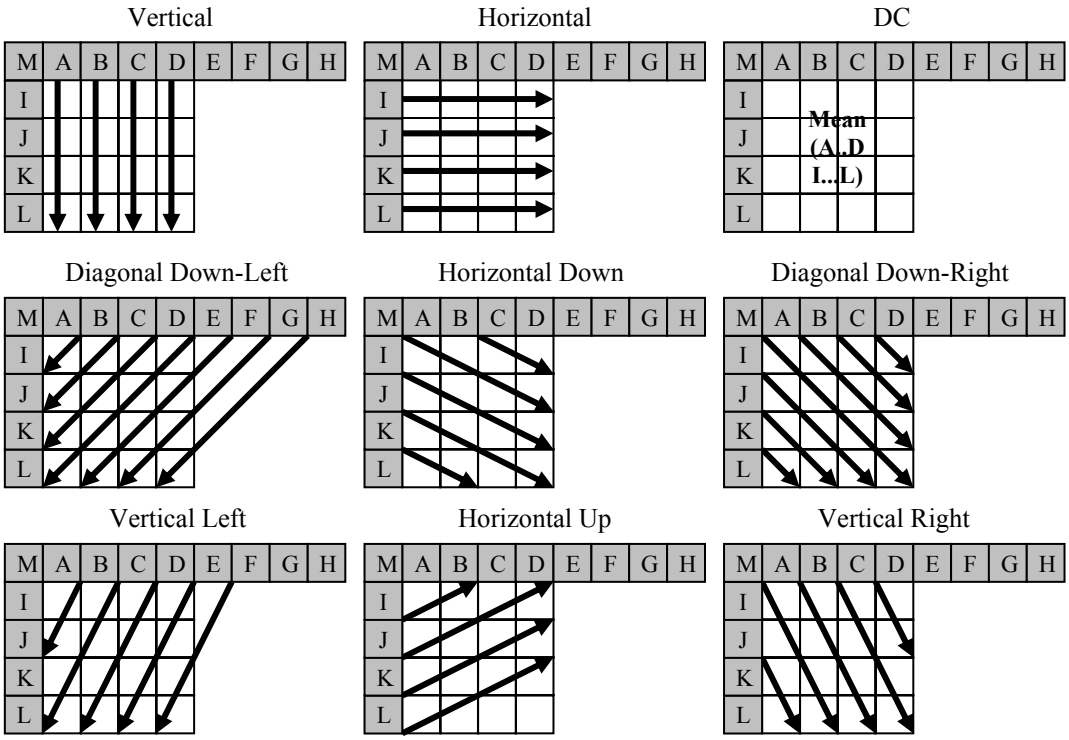



**Figure 2.2** 4x4 Luma Prediction Modes

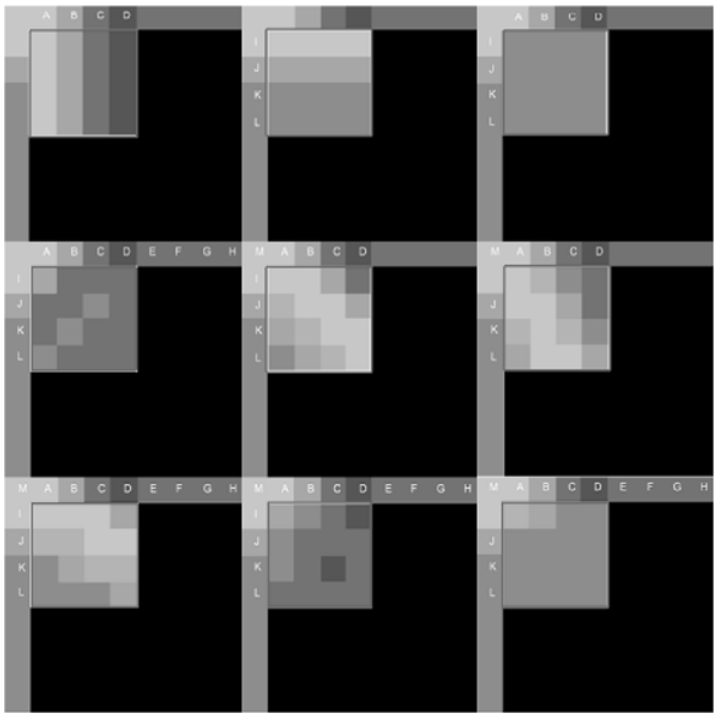


**Figure 2.3** Examples of Real Images for 4x4 Luma Prediction Modes

pred[0, 0] = A
pred[0, 1] = B
pred[0, 2] = C
pred[0, 3] = D
pred[1, 0] = A
pred[1, 1] = B
pred[1, 2] = C
pred[1, 3] = D
pred[2, 0] = A
pred[2, 1] = B
pred[2, 2] = C
pred[2, 3] = D
pred[3, 0] = A
pred[3, 1] = B
pred[3, 2] = C
pred[3, 3] = D

(a) 4x4 Vertical Mode

pred[0, 0] = I
pred[0, 1] = I
pred[0, 2] = I
pred[0, 3] = I
pred[1, 0] = J
pred[1, 1] = J
pred[1, 2] = J
pred[1, 3] = J
pred[2, 0] = K
pred[2, 1] = K
pred[2, 2] = K
pred[2, 3] = K
pred[3, 0] = L
pred[3, 1] = L
pred[3, 2] = L
pred[3, 3] = L

(b) 4x4 Horizontal Mode

pred[y,x] = (A + B + C + D + I + J + K + L + 4) >> 3
(If the left and the top neighboring pixels are available)

pred[y,x] = (I + J + K + L + 2) >> 2
(Else If only the left neighboring pixels are available)

pred[y,x] = (A + B + C + D + 2) >> 2
(Else If only the top neighboring pixels are available)

pred[y,x] = 128
(Else //If the left and the upper neighboring pixels are not available)

(c) 4x4 DC Mode

```
pred[0, 0] = A + 2B + C + 2 >> 2
pred[0, 1] = B + 2C + D + 2 >> 2
pred[0, 2] = C + 2D + E + 2 >> 2
pred[0, 3] = D + 2E + F + 2 >> 2
pred[1, 0] = B + 2C + D + 2 >> 2
pred[1, 1] = C + 2D + E + 2 >> 2
pred[1, 2] = D + 2E + F + 2 >> 2
pred[1, 3] = E + 2F + G + 2 >> 2
pred[2, 0] = C + 2D + E + 2 >> 2
pred[2, 1] = D + 2E + F + 2 >> 2
pred[2, 2] = E + 2F + G + 2 >> 2
pred[2, 3] = F + 2G + H + 2 >> 2
pred[3, 0] = D + 2E + F + 2 >> 2
pred[3, 1] = E + 2F + G + 2 >> 2
pred[3, 2] = F + 2G + H + 2 >> 2
pred[3, 3] = G + 3H + 2 >> 2
```

(d) 4x4 Diagonal Down Left Mode

```
pred[0, 0] = A + 2M + I  + 2 >> 2
pred[0, 1] = M + 2A + B + 2 >> 2
pred[0, 2] = A + 2B + C + 2 >> 2
pred[0, 3] = B + 2C + D + 2 >> 2
pred[1, 0] = M + 2I + J + 2 >> 2
pred[1, 1] = A + 2M + I + 2 >> 2
pred[1, 2] = M + 2A + B + 2 >> 2
pred[1, 3] = A + 2B + C + 2 >> 2
pred[2, 0] = I + 2J + K + 2 >> 2
pred[2, 1] = M + 2I + J + 2 >> 2
pred[2, 2] = A + 2M + I + 2 >> 2
pred[2, 3] = M + 2A + B + 2 >> 2
pred[3, 0] = J + 2K + L + 2 >> 2
pred[3, 1] = I + 2J + K + 2 >> 2
pred[3, 2] = M + 2I + J + 2 >> 2
pred[3, 3] = A + 2M + I + 2 >> 2
```

(e) 4x4 Diagonal Down Right Mode

```
pred[0, 0] = M + A + 1 >> 1
pred[0, 1] = A + B + 1 >> 1
pred[0, 2] = B + C + 1 >> 1
pred[0, 3] = C + D + 1 >> 1
pred[1, 0] = I + 2M + A + 2 >> 2
pred[1, 1] = M + 2A + B  + 2 >> 2
pred[1, 2] = A + 2B + C + 2 >> 2
pred[1, 3] = B + 2C + D + 2 >> 2
pred[2, 0] = M + 2I + J + 2 >> 2
pred[2, 1] = M + A + 1 >> 1
pred[2, 2] = A + B + 1 >> 1
pred[2, 3] = B + C + 1 >> 1
pred[3, 0] = I + 2J + K + 2 >> 2
pred[3, 1] = I + 2M + A + 2 >> 2
pred[3, 2] = M + 2A + B + 2 >> 2
pred[3, 3] = A + 2B + C + 2 >> 2
```

(f) 4x4 Vertical Right Mode

```
pred[0, 0] = M + I + 1 >> 1
pred[0, 1] = I + 2M + A + 2  >> 2
pred[0, 2] = B + 2A + M + 2 >> 2
pred[0, 3] = C + 2B + A + 2 >> 2
pred[1, 0] = I + J + 1 >> 1
pred[1, 1] = M + 2I + J + 2 >> 2
pred[1, 2] = M + I + 1 >> 1
pred[1, 3] = I + 2M + A + 2 >> 2
pred[2, 0] = J + K + 1 >> 1
pred[2, 1] = I + 2J + K + 2 >> 2
pred[2, 2] = I + J + 1 >> 1
pred[2, 3] = M + 2I + J + 2 >> 2
pred[3, 0] = K + L + 1 >> 1
pred[3, 1] = J + 2K + L + 2 >> 2
pred[3, 2] = J + K + 1 >> 1
pred[3, 3] = I + 2J + K + 2 >> 2
```

(g) 4x4 Horizontal Down Mode

pred[0, 0] = A + B + 1 >> 1
pred[0, 1] = B + C + 1 >> 1
pred[0, 2] = C + D + 1 >> 1
pred[0, 3] = D + E + 1 >> 1
pred[1, 0] = A + 2B + C + 2 >> 2
pred[1, 1] = B + 2C + D + 2 >> 2
pred[1, 2] = C + 2D + E + 2 >> 2
pred[1, 3] = D + 2E + F + 2 >> 2
pred[2, 0] = B + C + 1 >> 1
pred[2, 1] = C + D + 1 >> 1
pred[2, 2] = D + E + 1 >> 1
pred[2, 3] = E + F + 1 >> 1
pred[3, 0] = B + 2C + D + 2 >> 2
pred[3, 1] = C + 2D + E + 2 >> 2
pred[3, 2] = D + 2E + F + 2 >> 2
pred[3, 3] = E + 2F + G + 2 >> 2

(h) 4x4 Vertical Left Mode

pred[0, 0] = I + J + 1 >> 1
pred[0, 1] = I + 2J + K + 2 >> 2
pred[0, 2] = J + K+ 1 >> 1
pred[0, 3] = J + 2K + L + 2 >> 2
pred[1, 0] = J + K+ 1 >> 1
pred[1, 1] = J + 2K + L + 2 >> 2
pred[1, 2] = K + L + 1 >> 1
pred[1, 3] = K + 3L + 2 >> 2
pred[2, 0] = K + L + 1 >> 1
pred[2, 1] = K + 3L + 2 >> 2
pred[2, 2] = L
pred[2, 3] = L
pred[3, 0] = L
pred[3, 1] = L
pred[3, 2] = L
pred[3, 3] = L

(i) 4x4 Horizontal Up Mode

**Figure 2.4** Prediction Equations for 4x4 Luma Prediction Modes

**Table 2.1** Availability of 4x4 Luma Prediction Modes

| **Availability of Neighboring 4x4 Luma Blocks** | **Available 4x4 Luma Prediction Modes** |
|---|---|
| None available | DC |
| Left available, Top not available | Horizontal, DC, Horizontal Up |
| Top available, Left not available | Vertical Right, DC, Vertical Left, Diagonal Down-Left |
| Both available | All Modes |

There are four 16x16 luma prediction modes designed in a directional manner. Each 16x16 luma prediction mode generates 256 predicted pixel values using some or all of the upper (H) and left-hand (V) neighboring pixels as shown in Figure 2.5. Vertical, Horizontal and DC modes are similar to 4x4 luma prediction modes. Plane mode is an approximation of bilinear transform with only integer arithmetic. The examples of each 16x16 luma prediction mode for real images are given in Figure 2.6. The prediction equations used in 16x16 luma prediction modes are shown in Figure 2.7 where [y,x] denotes the position of the pixel in a MB (the top left, top right, bottom left, and bottom right positions of a MB are

denoted as [0,0], [0,15], [15,0], and [15,15], respectively), *p* represents the neighboring pixel values and Clip1 is to clip the result into [0-255] range.

DC mode is always used regardless of the availability of the neighboring pixels. However, it is adopted based on which neighboring pixels are available. The other prediction modes can only be used if all of the required neighboring pixels are available [2, 3]. Available 16x16 luma prediction modes for a MB depending on the availability of the neighboring MBs are given in Table 2.2.

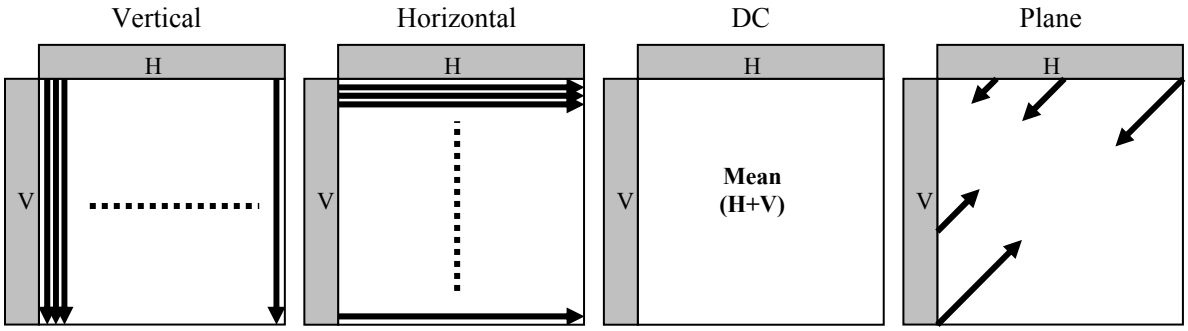



**Figure 2.5** 16x16 Luma Prediction Modes

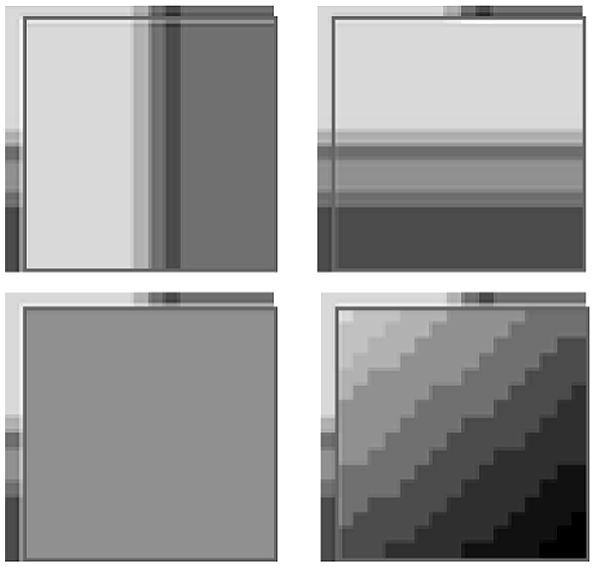


**Figure 2.6** Examples of Real Images for 16x16 Luma Prediction Modes

**Table 2.2** Availability of 16x16 Luma Prediction Modes

| **Availability of Neighboring 16x16 Luma Blocks** | **Available 16x16 Luma Prediction Modes** |
|---|---|
| None available | DC |
| Left available, Top not available | Horizontal, DC |
| Top available, Left not available | Vertical, DC |
| Both available | All Modes |

pred[y, 0] = *p*[-1, 0]
pred[y, 1] = *p*[-1, 1]
pred[y, 2] = *p*[-1, 2]
pred[y, 3] = *p*[-1, 3]
pred[y, 4] = *p*[-1, 4]
pred[y, 5] = *p*[-1, 5]
pred[y, 6] = *p*[-1, 6]
pred[y, 7] = *p*[-1, 7]
pred[y, 8] = *p*[-1, 8]
pred[y, 9] = *p*[-1, 9]
pred[y, 10] = *p*[-1, 10]
pred[y, 11] = *p*[-1, 11]
pred[y, 12] = *p*[-1, 12]
pred[y, 13] = *p*[-1, 13]
pred[y, 14] = *p*[-1, 14]
pred[y, 15] = *p*[-1, 15]

(0 <= y < = 15)

(a) 16x16 Vertical Mode

pred[0, x] = *p*[0, -1]
pred[1, x] = *p*[1, -1]
pred[2, x] = *p*[2, -1]
pred[3, x] = *p*[3, -1]
pred[4, x] = *p*[4, -1]
pred[5, x] = *p*[5, -1]
pred[6, x] = *p*[6, -1]
pred[7, x] = *p*[7, -1]
pred[8, x] = *p*[8, -1]
pred[9, x] = *p*[9, -1]
pred[10, x] = *p*[10, -1]
pred[11, x] = *p*[11, -1]
pred[12, x] = *p*[12, -1]
pred[13, x] = *p*[13, -1]
pred[14, x] = *p*[14, -1]
pred[15, x] = *p*[15, -1]

(0 <= x < = 15)

(b) 16x16 Horizontal Mode

$$pred[y,x] = p[-1, x'] + p[y', -1] + 16 \gg 5$$

(x' = 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15)
(y' = 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15)

(If the left and the upper neighboring pixels are available)

$$pred[y,x] = p[y', -1] + 8 \gg 4$$

(y' = 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15)

(Else If the left neighboring pixels are available)

$$pred[y,x] = p[-1, x'] + 8 >> 4$$

(x' = 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15)

(Else If the top neighboring pixels are available)

$$pred[y,x] = 128$$

(Else //If the left and the upper neighboring pixels are not available)

(c) 16x16 DC Mode

$$pred[y,x] = Clip1 [((a + b * (x - 3) + c * (y - 3) + 16) >> 5]$$

$$a = 16 *(p[-1,15] + p[15,-1])$$
$$b = (5 * H + 32) >> 6$$
$$c = (5 * V + 32) >> 6$$

$$H = \sum (x'+1)*( p[-1,8 + x'] + p[-1, 6- x'])$$
(x' = 0, 1, 2, 3, 4, 5, 6, 7)

$$V = \sum (y'+1)*( p[8 + y',-1] + p[ 6- y',-1])$$
(y' = 0, 1, 2, 3, 4, 5, 6, 7)

(d) 16x16 Plane Mode

**Figure 2.7** Prediction Equations for 16x16 Luma Prediction Modes

For the chroma components of a MB, a predicted 8x8 chroma block is formed for each 8x8 chroma component by performing intra prediction for the MB. There are four 8x8 chroma prediction modes which are similar to 16x16 luma prediction modes. A mode decision algorithm is used to compare the 8x8 predictions and select the best chroma prediction mode for each chroma component of the MB. Both chroma components of a MB always use the same prediction mode. The prediction equations used in 8x8 chroma prediction modes are shown in Figure 2.8 where [y,x] denotes the position of the pixel in a MB (the top left, top right, bottom left, and bottom right positions of a MB are denoted as [0,0], [0,7], [7,0], and [7,7], respectively), *p* represents the neighboring pixel values and Clip1 is to clip the result into [0-255] range.

DC mode is always used regardless of the availability of the neighboring pixels. However, it is adopted based on which neighboring pixels are available. The other prediction modes can only be used if all of the required neighboring pixels are available [4, 5]. Available 8x8 chroma prediction modes for a MB depending on the availability of the neighboring MBs are given in Table 2.3.

**Table 2.3** Availability of 8x8 Chroma Prediction Modes

| **Availability of Neighboring 8x8 Chroma Blocks** | **Available 8x8 Luma Prediction Modes** |
|---|---|
| None available | DC |
| Left available, Top not available | Horizontal, DC |
| Top available, Left not available | Vertical, DC |
| Both available | All Modes |

pred[y, 0] = $p$[-1, 0]
pred[y, 1] = $p$[-1, 1]
pred[y, 2] = $p$[-1, 2]
pred[y, 3] = $p$[-1, 3]
pred[y, 4] = $p$[-1, 4]
pred[y, 5] = $p$[-1, 5]
pred[y, 6] = $p$[-1, 6]
pred[y, 7] = $p$[-1, 7]

(0 <= y < = 7)

(a) 8x8 Vertical Mode

pred[0, x] = $p$[0, -1]
pred[1, x] = $p$[1, -1]
pred[2, x] = $p$[2, -1]
pred[3, x] = $p$[3, -1]
pred[4, x] = $p$[4, -1]
pred[5, x] = $p$[5, -1]
pred[6, x] = $p$[6, -1]
pred[7, x] = $p$[7, -1]

(0 <= x < = 7)

(b) 8x8 Horizontal Mode

$$\text{pred}[y, x] = (\sum p[x', -1] + \sum p[-1, y'] + 4) >> 3$$

(If p[x', –1] with x' = 0..3, and p[–1, y'] and y' = 0..3 are available)

$$\text{pred}[y, x] = (\sum p[x', -1] + 2) >> 2$$

(Else If p[x', –1] with x' = 0..3 are available and p[–1, y'] and y' = 0..3 are not available)

$$\text{pred}[y, x] = (\sum p[-1, y'] + 2) >> 2$$

(Else If p[–1, y'] and y' = 0..3 are available and p[x', –1] with x' = 0..3 are not available)

pred[y, x] = 128
(Else //If p[x', –1] with x' = 0..3, and p[–1, y'] and y' = 0..3 are not available)

(0 <= x < = 3)
(0 <= y < = 3)

(c-1) 8x8 DC Mode

pred[y, x] = (∑ *p*[x', -1] + 2) >> 2
(If p[x', –1] with x' = 4..7 are available)

pred[y, x] = (∑ *p*[-1, y'] + 2) >> 2
(Else If p[–1, y'] and y' = 0..3 are available)
pred[y, x] = 128
(Else)

(4 <= x < = 7)
(0 <= y < = 3)

(c-2) 8x8 DC Mode

pred[y, x] = (∑ *p*[-1, y'] + 2) >> 2
(If p[–1, y'] and y' = 4..7 are available)

pred[y, x] = (∑ *p*[x', -1] + 2) >> 2
(Else If p[x', –1] with x' = 0..3 are available)
pred[y, x] = 128
(Else)

(0 <= x < = 3)
(4 <= y < = 7)

(c-3) 8x8 DC Mode

pred[y, x] = (∑ *p*[x', -1] + ∑ *p*[-1, y'] + 4) >> 3
(If p[x', –1] with x' = 4..7, and p[–1, y'] and y' = 4..7 are available)

pred[y, x] = (∑ *p*[x', -1] + 2) >> 2
(Else If p[x', –1] with x' = 4..7 are available and p[–1, y'] and y' = 4..7 are not available)

pred[y, x] = (∑ *p*[-1, y'] + 2) >> 2
(Else If p[–1, y'] and y' = 4..7 are available and p[x', –1] with x' = 4..7 are not available)

pred[y, x] = 128
(Else //If p[x', –1] with x' = 4..7, and p[–1, y'] and y' = 4..7 are not available)

(4 <= x < = 7)
(4 <= y < = 7)

(c-4) 8x8 DC Mode

$$pred[y,x] = Clip1\ [((a + b * (x - 3) + c * (y - 3) + 16) >> 5]$$

$$a = 16 *(p[-1,7] + p[7,-1])$$
$$b = (5 * H + 32) >> 6$$
$$c = (5 * V + 32) >> 6$$

$$H = \sum (x'+1)*( p[-1,4 + x'] + p[-1, 2- x'])$$
$$(x' = 0, 1, 2, 3)$$

$$V = \sum (y'+1)*( p[4 + y',-1] + p[ 2- y',-1])$$
$$(y' = 0, 1, 2, 3)$$

(d) 8x8 Plane Mode

**Figure 2.8** Prediction Equations for 8x8 Chroma Prediction Modes

## 2.2 Proposed Hardware Architecture

In the previous video coding standards, prediction part and coding part (DCT/Q/IQ/IDCT/VLC) can be clearly separated. After prediction part finishes processing a MB, coding part starts coding this MB and prediction part starts processing the next MB. However, because of the intra prediction algorithm used in H.264 standard, after prediction of a MB, the next MB can not be predicted before corresponding reconstructed pixels at the output of the Transform/Quant/Inverse Quant/Inverse Transform are produced. The situation is even worse for 4x4 luma prediction modes. Prediction and mode decision of a 4x4 block cannot be performed until the previous 4x4 block is reconstructed. Therefore, the

prediction part must wait the coding part which makes the MB pipelining impossible and the design of a real-time intra frame coder hardware very costly.

We, however, divided our proposed H.264 intra frame coder hardware into two main parts; the search & mode decision part and coder part. The search & mode decision hardware and the coder hardware work in a pipelined manner. After the first MB of the input frame is loaded to the input register file, search & mode decision hardware starts to work on determining the best mode for coding this MB. After search & mode decision hardware determines the best mode for the first MB, coder hardware starts to code the first MB using the selected best mode and search & mode decision hardware starts to work on the second MB. The entire frame is processed MB by MB in this order.

This is achieved by performing intra prediction in the search & mode decision hardware using the pixels in the current frame rather than the pixels in the reconstructed frame. However, intra prediction in the coder hardware is performed using the pixels in the reconstructed frame in order to be compliant with H.264 standard. Therefore, in this thesis, two different low-cost hardware architectures are designed for H.264 intra prediction algorithm, one for search & mode decision hardware and one for coder hardware. This makes the MB pipelining and therefore the implementation of a low-cost H.264 intra frame coder hardware possible at the expense of a small PSNR loss in the video quality.

### 2.2.1 Intra Prediction Hardware for Search & Mode Decision

The block diagram of the proposed intra prediction hardware architecture for the search & mode decision part of the H.264 intra frame coder is shown in Figure 2.9. The proposed hardware generates the predicted pixels using available 16x16 and 4x4 luma prediction modes for luma and 8x8 chroma prediction modes for chroma components of a MB with different configurations. In the proposed hardware, there are two parts operating in parallel in order to perform intra prediction faster.

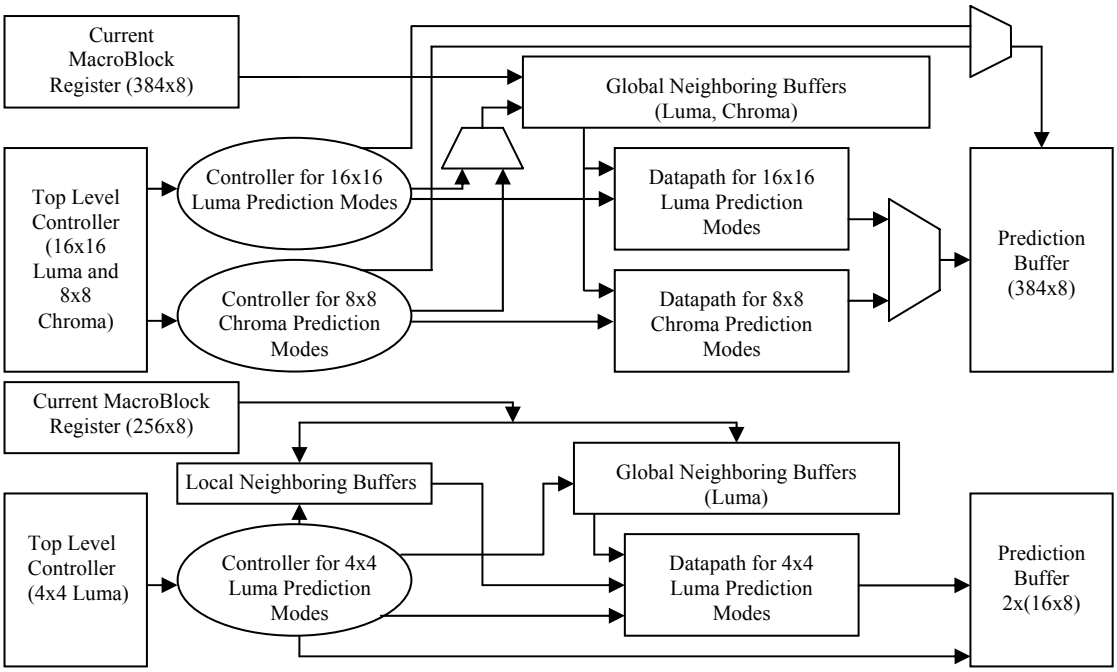



**Figure 2.9** Intra Prediction Hardware for Search & Mode Decision

The upper part is used for generating the predicted pixels for the luma component of a MB using available 16x16 luma prediction modes and for generating the predicted pixels for the chroma components of a MB using available 8x8 chroma prediction modes. The size of register files that are used for the current MB and the prediction buffer is 384x8, because they are used for storing both luma and chroma components of the current and predicted MB respectively.

The lower part is used for generating the predicted pixels for each 4x4 block in the luma component of a MB using available 4x4 luma prediction modes. The lower part is more computationally demanding and it is the bottleneck in the intra prediction hardware for search & mode decision. The size of the current MB register file is 256x8, because it is used for storing only luma components of the current MB. The size of the prediction buffer is 16x8 since it is used for storing the predicted pixels for a 4x4 luma block.

Two local neighboring buffers, local vertical register file and local horizontal register file, are used to store the neighboring pixels in the previously coded and reconstructed neighboring 4x4 luma blocks in the current MB. After a 4x4 luma block in the current MB is coded and reconstructed, the neighboring pixels in this block are stored in the corresponding local register files.

Local vertical register file is used to store the neighboring pixels d, h, l, and p in the left-hand previously coded and reconstructed neighboring 4x4 luma blocks in the current MB. Local horizontal register file is used to store the neighboring pixels m, n, o, and p in the upper previously coded and reconstructed 4x4 luma blocks in the current MB. The proposed hardware uses this data to determine the neighboring pixels in the left-hand and upper previously coded neighboring 4x4 luma blocks in the current MB.

Six global neighboring buffers, three global vertical neighboring buffers and three global horizontal neighboring buffers, are used to store the neighboring pixels in the previously coded and reconstructed neighboring MBs of the current MB.

Global luma vertical register file is used to store the neighboring pixels d, h, l, and p in the 4x4 luma blocks 5, 7, 13 and 15 of the previously coded MB. The proposed hardware uses this data to determine the neighboring pixels in the left-hand previously coded neighboring MB of the 4x4 luma blocks 0, 2, 8, and 10 in the current MB. Global Cb vertical register file and global Cr vertical register file are used for the chroma Cb and chroma Cr components of the MBs.

Global luma horizontal register file is used to store the neighboring pixels m, n, o, and p in the luma blocks 10, 11, 14, and 15 of the previously coded MBs in the previously coded MB row of the frame. The proposed hardware uses this data to determine the neighboring pixels in the upper previously coded neighboring MB of the 4x4 luma blocks 0, 1, 4, and 5 in the current MB. Global Cb horizontal register file and global Cr horizontal register file are used for the chroma Cb and chroma Cr components of the MBs.

Instead of using one large external SRAM, we have used 8 internal register files to store the neighboring reconstructed pixels in order to reduce power consumption. The power consumption is reduced by accessing a small register file for storing and reading a reconstructed pixel instead of accessing a large external SRAM. In addition, we have disabled the register files when they are not accessed in order to reduce power consumption.

#### 2.2.1.1 Proposed Hardware for 4x4 Luma Prediction Modes

After a careful analysis of the equations used in 4x4 luma prediction modes, it is observed that there are common parts in the equations and some of the equations are identical. The intra prediction equations are organized for exploiting these observations to reduce both the number of memory accesses and computation time required for generating the predicted pixels. The organized prediction equations for 4x4 luma prediction modes are shown in Figure 2.10. As it can be seen from the figure, (A + B), (B + C), (C + D), (D + E), (E + F), (F + G), (G + H), (J + K), (I + J), (M + I) and (M + A) are common in two or more equations, and some of the prediction equations (e.g. [(A + B) + (B + C) + 2] >> 2) are identical.

Pred[0, 0] = Pred[1, 0] = Pred[2, 0] = Pred[3, 0] = A

Pred[0, 1] = Pred[1, 1] = Pred[2, 1] = Pred[3, 1] = B

Pred[0, 2] = Pred[1, 2] = Pred[2, 2] = Pred[3, 2] = C

Pred[0, 3] = Pred[1, 3] = Pred[2, 3] = Pred[3, 3] = D

(a) 4x4 Vertical Prediction Mode

Pred[0, 0] = Pred[0, 1] = Pred[0, 2] = Pred[0, 3] = I

Pred[1, 0] = Pred[1, 1] = Pred[1, 2] = Pred[1, 3] = J

Pred[2, 0] = Pred[2, 1] = Pred[2, 2] = Pred[2, 3] = K

Pred[3, 0] = Pred[3, 1] = Pred[3, 2] = Pred[3, 3] = L

(b) 4x4 Horizontal Prediction Mode

pred[y,x] = [(A + B) + (C + D) + (I + J) + (K + L) + 4] >> 3
(If the left and the top neighboring pixels are available)

Pred[y, x] = [(I + J) + (K + L) + 2] >> 2
(Else If only the left neighboring pixels are available)

pred[y, x] = [(A + B) + (C + D) + 2] >> 2
(Else If only the top neighboring pixels are available)

pred[y,x] = 128
(Else //If the left and the top neighboring pixels are not available)

(c) 4x4 DC Prediction Mode

| | |
|---|---|
| Pred[0, 0] | = [(A + B) + (B + C) + 2] >> 2 |
| Pred[0, 1] = Pred[1, 0] | = [(C + D) + (B + C) + 2] >> 2 |
| Pred[0, 2] = Pred[1, 1] = Pred[2, 0] | = [(C + D) + (D + E) + 2] >> 2 |
| Pred[0, 3] = Pred[1, 2] = Pred[2, 1] | = [(E + F) + (D + E) + 2] >> 2 |
| Pred[3, 0] | = [(E + F) + (D + E) + 2] >> 2 |
| Pred[1, 3] = Pred[2, 2] = Pred[3, 1] | = [(E + F) + (F + G) + 2] >> 2 |
| Pred[2, 3] = Pred[3, 2] | = [(G + H) + (F + G) + 2] >> 2 |
| Pred[3, 3] | = [(G + H) + (H +H) + 2] >> 2 |

(d) 4x4 Diagonal Down-Left Mode

| | |
|---|---|
| Pred[0, 2] = Pred[1, 3] | = [(A + B) + (B + C) + 2] >> 2 |
| Pred[0, 3] | = [(C + D) + (B + C) + 2] >> 2 |
| Pred[3, 0] | = [(J + K) + (K+ L) + 2] >> 2 |
| Pred[2, 0] = Pred[3, 1] | = [(J + K) + (I + J) + 2] >> 2 |
| Pred[1, 0] = Pred[2, 1] = Pred[3, 2] | = [(M + I) + (I + J) + 2] >> 2 |
| Pred[0, 0] = Pred[1, 1] = Pred[2, 2] = Pred[3, 3] | = [(M + I) + (M + A) + 2] >> 2 |
| Pred[0, 1] = Pred[1, 2] = Pred[2, 3] | = [(A + B) + (M + A) + 2] >> 2 |

(e) 4x4 Diagonal Down-Right Mode

Pred[3, 0] = [(I + J) + (J + K) + 2] >> 2
Pred[2, 0] = [(I + J) + (M + I) + 2] >> 2
Pred[1, 0] = Pred[3, 1] = [(M + A) + (M + I) + 2] >> 2
Pred[1, 1] = Pred[3, 2] = [(M + A) + (A + B) + 2] >> 2
Pred[1, 2] = Pred[3, 3] = [(B + C) + (A + B) + 2] >> 2
Pred[1, 3] = [(B + C) + (C + D) + 2] >> 2
Pred[0, 1] = Pred[2, 1] = [(A + B) + 1] >> 1
Pred[0, 3] = [(C + D) + 1] >> 1
Pred[0, 0] = Pred[2, 1] = [(M + A) + 1] >> 1
Pred[0, 2] = Pred[2, 3] = [(B + C) + 1] >> 1

(f) 4x4 Vertical Right Mode

Pred[3, 1] = [(K + L) + (J + K) + 2)] >> 2
Pred[2, 1] = Pred[3, 3] = [(I + J) + (J + K) + 2] >> 2
Pred[1, 1] = Pred[2, 3] = [(I + J) + (M + I) + 2] >> 2
Pred[0, 1] = Pred[1, 3] = [(M + A) + (M + I) + 2] >> 2
Pred[0, 2] = [(M + A) + (A + B) + 2] >> 2
Pred[0, 3] = [(B + C) + (A + B) + 2] >> 2
Pred[0, 0] = Pred[1, 2] = [(M + I) + 1] >> 1
Pred[1, 0] = Pred[2, 2] = [(I + J) + 1] >> 1
Pred[2, 0] = Pred[3, 2] = [(J + K) + 1] >> 1
Pred[3, 0] = [(K + L) + 1] >> 1

(g) 4x4 Horizontal Down Mode

Pred[1, 0] = [(A + B) + (B + C) + 2] >> 2
Pred[1, 1] = Pred[3, 0] = [(C + D) + (B + C) + 2] >> 2
Pred[1, 2] = Pred[3, 1] = [(C + D) + (D + E) + 2] >> 2
Pred[1, 3] = Pred[3, 2] = [(E + F) + (D + E) + 2] >> 2
Pred[3, 3] = [(E + F) + (F + G) + 2] >> 2
Pred[0, 0] = [(A + B) + 1] >> 1
Pred[0, 1] = Pred[2, 0] = [(B + C) + 1] >> 1
Pred[0, 2] = Pred[2, 1] = [(C + D) + 1] >> 1
Pred[0, 3] = Pred[2, 2] = [(D + E) + 1] >> 1
Pred[2, 3] = [(E + F) + 1] >> 1

(h) 4x4 Vertical Left Mode

Pred[0, 1] = [(I + J) + (J + K) + 2] >> 2
Pred[0, 3] = Pred[1, 1] = [(J + K) + (K + L) + 2)] >> 2
Pred[1, 3] = Pred[2, 1] = [(L + L) + (K + L) + 2)] >> 2
Pred[1, 2] = Pred[2, 0] = [(K+ L) + 1] >> 1
Pred[0, 0] = [(I + J) + 1] >> 1
Pred[0, 2] = Pred[1, 0] = [(J + K) + 1] >> 1
Pred[2, 2] = Pred[2, 3] = Pred[3, 0] = Pred[3, 1]
= Pred[3, 2] = Pred[3, 3] = L

(i) 4x4 Horizontal Up Mode

**Figure 2.10** Organized Prediction Equations for 4x4 Luma Prediction Modes

The proposed hardware first calculates the results of the common parts in all the 4x4 luma prediction modes and stores them in temporary registers. It, then, calculates the results of the prediction equations using the values stored in these temporary registers. If both the left and top neighboring blocks of a 4x4 luma block are available, 12 common parts are calculated in the preprocessing step and this takes 8 clock cycles. The neighboring buffers are only accessed during this preprocessing. Therefore, they are disabled after the preprocessing for reducing power consumption.

The proposed hardware calculates the results of the identical prediction equations only once and stores them in temporary registers. It, then, determines the results of identical prediction equations by reading the values stored in these temporary registers, instead of calculating the same equations again.

The proposed datapath for generating predicted pixels for a 4x4 luma block using all 4x4 luma prediction modes is shown in Figure 2.11. Level0 (L0) registers are used to store the results of the common parts in the equations of all the 4x4 luma prediction modes. Level1 (L1) registers are used to store the results of the identical prediction equations used in all the 4x4 luma prediction modes. If both the left and top neighboring blocks of a 4x4 luma block are available, it takes 165 clock cycles to generate the predicted pixels for that 4x4 block using available 4x4 luma prediction modes. Clock cycles required for preprocessing and performing available 4x4 luma predictions based on the availability of neighboring 4x4 luma blocks for a 4x4 luma block are given in Table 2.4.

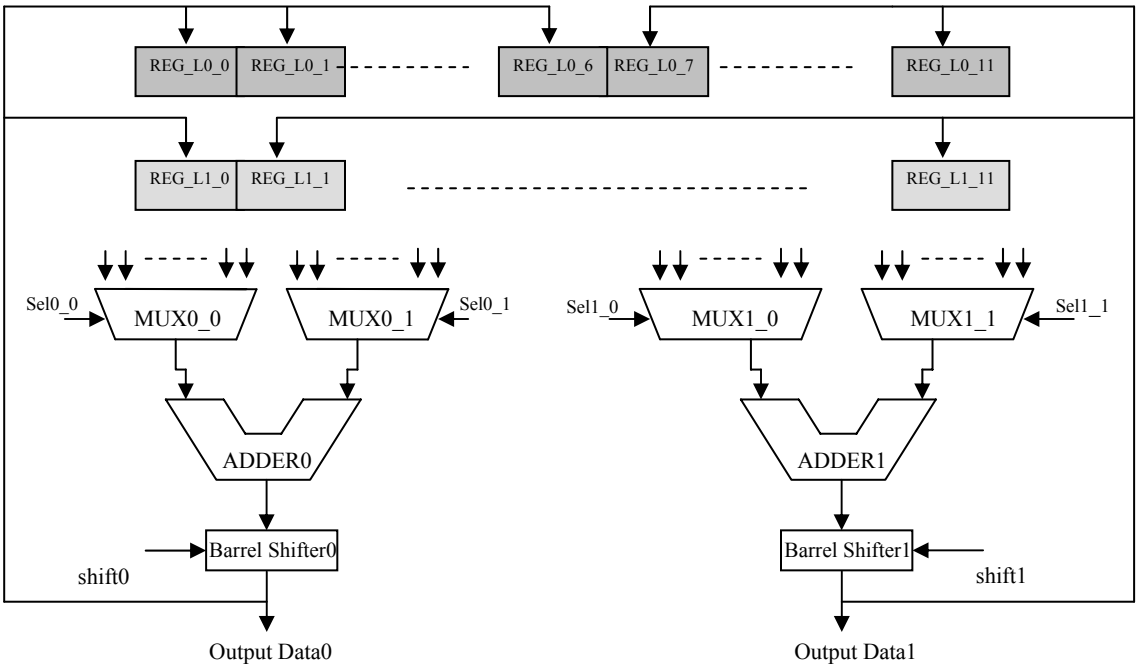



**Figure 2.11** Datapath for 4x4 Luma Prediction Modes

**Table 2.4** Clock Cycles Required for Performing Available 4x4 Luma Prediction Modes

| **Preprocessing & Available Modes** | **Clock Cycles / 4x4 Luma Block** |
|---|---|
| Preprocessing | 8 |
| Vertical | 17 |
| Horizontal | 17 |
| DC | 19 |
| Diagonal Down-Left Mode | 18 |
| Diagonal Down-Right Mode | 18 |
| Vertical Right | 17 |
| Horizontal Down | 17 |
| Vertical Left | 17 |
| Horizontal Up | 17 |

(a) Top and Left Neighboring 4x4 Luma Blocks are available

| **Preprocessing & Available Modes** | **Clock Cycles / 4x4 Luma Block** |
|---|---|
| Preprocessing | 8 |
| Vertical | 17 |
| DC | 18 |
| Diagonal Down-Left Mode | 18 |
| Vertical Left | 17 |

(b)Top Neighboring 4x4 Luma Block is available

| **Preprocessing & Available Modes** | **Clock Cycles/4x4 Luma Block** |
|---|---|
| Preprocessing | 5 |
| Horizontal | 17 |
| DC | 18 |
| Horizontal Up | 18 |

(c) Left Neigboring 4x4 Luma Block is available

| **Preprocessing & Available Modes** | **Clock Cycles/4x4 Luma Block** |
|---|---|
| Preprocessing | - |
| DC | 17 |

(d) Top and Left Neighboring 4x4 Luma Blocks are not available

Since the order of the equations used in a 4x4 luma prediction mode is not important for functional correctness, the equations are ordered to keep the inputs of the adders the same for as many consecutive clock cycles as possible. This avoids unnecessary switching activity and reduces the power consumption.

#### 2.2.1.2 Proposed Hardware for 16x16 Luma Prediction Modes

After a careful analysis of the equations used in 16x16 luma prediction modes, it is observed that Vertical, Horizontal and DC mode equations can directly be implemented using adders and shifters, however the equations used in Plane mode can be organized to avoid using a multiplier and to reduce computation time required for generating the predicted pixels. The organized prediction equations for Plane mode are shown in Figure 2.12. A similar organization for the Plane mode prediction equations is given in [7, 8]. However, our hardware design is different than their design and it is a more cost-effective solution for portable applications.

$$a = (p[-1,15] + p[15,-1]) << 4,\ b = [(H << 2) + (H + 32)] >> 6$$
$$c = [(V << 2) + (V + 32)] >> 6$$

$$\mathbf{C0 = [a - (7 * b) - (7 * c) + 16]}$$

pred[0, 0] = Clip1 [(C0) >> 5]
pred[0, 1] = Clip1 [(C0 + b) >> 5]
pred[0, 2] = Clip1 [(C0 + 2b) >> 5]
pred[0, 3] = Clip1 [(C0 + 3b) >> 5]

pred[1, 0] = Clip1 [(C0 + c) >> 5]
pred[1, 1] = Clip1 [((C0 + c) + b) >> 5]
pred[1, 2] = Clip1 [((C0 + c) + 2b) >> 5]
pred[1, 3] = Clip1 [((C0 + c) + 3b) >> 5]

pred[2, 0] = Clip1 [(C0 + 2c) >> 5]
pred[2, 1] = Clip1 [((C0 + 2c) + b) >> 5]
pred[2, 2] = Clip1 [((C0 + 2c) + 2b) >> 5]
pred[2, 3] = Clip1 [((C0 + 2c) + 3b) >> 5]

pred[3, 0] = Clip1 [(C0 + 3c)>> 5]
pred[3, 1] = Clip1 [((C0 + 3c) + b) >> 5]
pred[3, 2] = Clip1 [((C0 + 3c) + 2b) >> 5]
pred[3, 3] = Clip1 [((C0 + 3c) + 3b) >> 5]

**C1 = [a – (3 * b) - (7 * c) + 16] = C0 + 4b**

pred[0, 4] = Clip1 [(C1) >> 5]
pred[0, 5] = Clip1 [(C1 + b) >> 5]
pred[0, 6] = Clip1 [(C1 + 2b) >> 5]
pred[0, 7] = Clip1 [(C1 + 3b) >> 5]

pred[1, 4] = Clip1 [(C1 + c) >> 5]
pred[1, 5] = Clip1 [((C1 + c) + b) >> 5]
pred[1, 6] = Clip1 [((C1 + c) + 2b) >> 5]
pred[1, 7] = Clip1 [((C1 + c) + 3b) >> 5]

pred[2, 4] = Clip1 [(C1 + 2c) >> 5]
pred[2, 5] = Clip1 [((C1 + 2c) + b) >> 5]
pred[2, 6] = Clip1 [((C1 + 2c) + 2b) >> 5]
pred[2, 7] = Clip1 [((C1 + 2c) + 3b) >> 5]

pred[3, 4] = Clip1 [(C1 + 3c) >> 5]
pred[3, 5] = Clip1 [((C1 + 3c) + b) >> 5]
pred[3, 6] = Clip1 [((C1 + 3c) + 2b) >> 5]
pred[3, 7] = Clip1 [((C1 + 3c) + 3b) >> 5]

**C2 = [a – (7 * b) - (3 * c) + 16] = C0 + 4c**

pred[4, 0] = Clip1 [(C2) >> 5]
pred[4, 1] = Clip1 [(C2 + b) >> 5]
pred[4, 2] = Clip1 [(C2 + 2b) >> 5]
pred[4, 3] = Clip1 [(C2 + 3b) >> 5]

pred[5, 0] = Clip1 [(C2 + c) >> 5]
pred[5, 1] = Clip1 [((C2 + c) + b) >> 5]
pred[5, 2] = Clip1 [((C2 + c) + 2b) >> 5]
pred[5, 3] = Clip1 [((C2 + c) + 3b) >> 5]

pred[6, 0] = Clip1 [(C2 + 2c) >> 5]
pred[6, 1] = Clip1 [((C2 + 2c) + b) >> 5]
pred[6, 2] = Clip1 [((C2 + 2c) + 2b) >> 5]
pred[6, 3] = Clip1 [((C2 + 2c) + 3b) >> 5]

pred[7, 0] = Clip1 [(C2 + 3c) >> 5]
pred[7, 1] = Clip1 [((C2 + 3c) + b) >> 5]
pred[7, 2] = Clip1 [((C2 + 3c) + 2b) >> 5]
pred[7, 3] = Clip1 [((C2 + 3c) + 3b) >> 5]

**C3 = [a – (3 * b) - (3 * c) + 16] = C0 + 4c + 4b**

pred[4, 4] = Clip1 [(C3) >> 5]
pred[4, 5] = Clip1 [(C3 + b) >> 5]
pred[4, 6] = Clip1 [(C3 + 2b) >> 5]
pred[4, 7] = Clip1 [(C3 + 3b) >> 5]

pred[5, 4] = Clip1 [(C3 + c) >> 5]
pred[5, 5] = Clip1 [((C3 + c) + b) >> 5]
pred[5, 6] = Clip1 [((C3 + c) + 2b) >> 5]
pred[5, 7] = Clip1 [((C3 + c) + 3b) >> 5]

pred[6, 4] = Clip1 [(C3 + 2c)>> 5]
pred[6, 5] = Clip1 [((C3 + 2c) + b) >> 5]
pred[6, 6] = Clip1 [((C3 + 2c) + 2b) >> 5]
pred[6, 7] = Clip1 [((C3 + 2c) + 3b) >> 5]

pred[7, 4] = Clip1 [(C3 + 3c)>> 5]
pred[7, 5] = Clip1 [((C3 + 3c) + b) >> 5]
pred[7, 6] = Clip1 [((C3 + 3c) + 2b) >> 5]
pred[7, 7] = Clip1 [((C3 + 3c) + 3b) >> 5]

**C4 = [a + (1 * b) - (7 * c) + 16] = C0 + 8b**

pred[0, 8] = Clip1 [(C4) >> 5]
pred[0, 9] = Clip1 [(C4 + b) >> 5]
pred[0, 10] = Clip1 [(C4 + 2b) >> 5]
pred[0, 11] = Clip1 [(C4 + 3b) >> 5]

pred[1, 8] = Clip1 [(C4 + c) >> 5]
pred[1, 9] = Clip1 [((C4 + c) + b) >> 5]
pred[1, 10] = Clip1 [((C4 + c) + 2b) >> 5]
pred[1, 11] = Clip1 [((C4 + c) + 3b) >> 5]

pred[2, 8] = Clip1 [(C4 + 2c)>> 5]
pred[2, 9] = Clip1 [((C4 + 2c) + b) >> 5]
pred[2, 10] = Clip1 [((C4 + 2c) + 2b) >> 5]
pred[2, 11] = Clip1 [((C4 + 2c) + 3b) >> 5]

pred[3, 8] = Clip1 [(C4 + 3c)>> 5]
pred[3, 9] = Clip1 [((C4 + 3c) + b) >> 5]
pred[3, 10] = Clip1 [((C4 + 3c) + 2b) >> 5]
pred[3, 11] = Clip1 [((C4 + 3c) + 3b) >> 5]

**C5 = [a + (5 * b) - (7 * c) + 16] = C0 + 12b**

pred[0, 12] = Clip1 [(C5)>> 5]
pred[0, 13] = Clip1 [(C5 + b)>> 5]
pred[0, 14] = Clip1 [(C5 + 2b)>> 5]
pred[0, 15] = Clip1 [(C5 + 3b)>> 5]

pred[1, 12] = Clip1 [(C5 + c)>> 5]
pred[1, 13] = Clip1 [((C5 + c) + b) >> 5]
pred[1, 10] = Clip1 [((C5 + c) + 2b) >> 5]
pred[1, 11] = Clip1 [((C5 + c) + 3b) >> 5]

pred[2, 12] = Clip1 [(C5 + 2c)>> 5]
pred[2, 13] = Clip1 [((C5 + 2c) + b) >> 5]
pred[2, 14] = Clip1 [((C5 + 2c) + 2b) >> 5]
pred[2, 15] = Clip1 [((C5 + 2c) + 3b) >> 5]

pred[3, 12] = Clip1 [(C5 + 3c)>> 5]
pred[3, 13] = Clip1 [((C5 + 3c) + b) >> 5]
pred[3, 14] = Clip1 [((C5 + 3c) + 2b) >> 5]
pred[3, 15] = Clip1 [((C5 + 3c) + 3b) >> 5]

**C6 = [a + (1 * b) - (3 * c) + 16] = C0 + 8b + 4c**

pred[4, 8] = Clip1 [(C6)>> 5]
pred[4, 9] = Clip1 [(C6 + b) >> 5]
pred[4, 10] = Clip1 [(C6 + 2b) >> 5]
pred[4, 11] = Clip1 [(C6 + 3b) >> 5]

pred[5, 8] = Clip1 [(C6 + c)>> 5]
pred[5, 9] = Clip1 [((C6 + c) + b) >> 5]
pred[5, 10] = Clip1 [((C6 + c) + 2b) >> 5]
pred[5, 11] = Clip1 [((C6 + c) + 3b) >> 5]

pred[6, 8] = Clip1 [(C6 + 2c)>> 5]
pred[6, 9] = Clip1 [((C6 + 2c) + b) >> 5]
pred[6, 10] = Clip1 [((C6 + 2c) + 2b) >> 5]
pred[6, 11] = Clip1 [((C6 + 2c) + 3b) >> 5]

pred[7, 8] = Clip1 [(C6 + 3c)>> 5]
pred[7, 9] = Clip1 [((C6 + 3c) + b) >> 5]
pred[7, 10] = Clip1 [((C6 + 3c) + 2b) >> 5]
pred[7, 11] = Clip1 [((C6 + 3c) + 3b) >> 5]

**C7 = [a + (5 * b) - (3 * c) + 16] = C0 + 12b + 4c**

pred[4, 12] = Clip1 [(C7) >> 5]
pred[4, 13] = Clip1 [(C7 + b) >> 5]
pred[4, 14] = Clip1 [(C7 + 2b) >> 5]
pred[4, 15] = Clip1 [(C7 + 3b) >> 5]

pred[5, 12] = Clip1 [(C7 + c)>> 5]
pred[5, 13] = Clip1 [((C7 + c) + b) >> 5]
pred[5, 10] = Clip1 [((C7 + c) + 2b) >> 5]
pred[5, 11] = Clip1 [((C7 + c) + 3b) >> 5]

pred[6, 12] = Clip1 [(C7 + 2c) >> 5]
pred[6, 13] = Clip1 [((C7 + 2c) + b) >> 5]
pred[6, 14] = Clip1 [((C7 + 2c) + 2b) >> 5]
pred[6, 15] = Clip1 [((C7 + 2c) + 3b) >> 5]

pred[7, 12] = Clip1 [(C7 + 3c) >> 5]
pred[7, 13] = Clip1 [((C7 + 3c) + b) >> 5]
pred[7, 14] = Clip1 [((C7 + 3c) + 2b) >> 5]
pred[7, 15] = Clip1 [((C7 + 3c) + 3b) >> 5]

**C8 = [a – (7 * b) + (1 * c) + 16] = C0 + 8c**

pred[8, 0] = Clip1 [(C8) >> 5]
pred[8, 1] = Clip1 [(C8 + b) >> 5]
pred[8, 2] = Clip1 [(C8 + 2b) >> 5]
pred[8, 3] = Clip1 [(C8 + 3b) >> 5]

pred[9, 0] = Clip1 [(C8 + c) >> 5]
pred[9, 1] = Clip1 [((C8 + c) + b) >> 5]
pred[9, 2] = Clip1 [((C8 + c) + 2b) >> 5]
pred[9, 3] = Clip1 [((C8 + c) + 3b) >> 5]

pred[10, 0] = Clip1 [(C8 + 2c) >> 5]
pred[10, 1] = Clip1 [((C8 + 2c) + b) >> 5]
pred[10, 2] = Clip1 [((C8 + 2c) + 2b) >> 5]
pred[10, 3] = Clip1 [((C8 + 2c) + 3b) >> 5]

pred[11, 0] = Clip1 [(C8 + 3c) >> 5]
pred[11, 1] = Clip1 [((C8 + 3c) + b) >> 5]
pred[11, 2] = Clip1 [((C8 + 3c) + 2b) >> 5]
pred[11, 3] = Clip1 [((C8 + 3c) + 3b) >> 5]

**C9 = [a – (3 * b) + (1 * c) + 16] = C0 + 4b + 8c**

pred[8, 4] = Clip1 [(C9) >> 5]
pred[8, 5] = Clip1 [(C9 + b) >> 5]
pred[8, 6] = Clip1 [(C9 + 2b) >> 5]
pred[8, 7] = Clip1 [(C9 + 3b) >> 5]

pred[9, 4] = Clip1 [(C9 + c) >> 5]
pred[9, 5] = Clip1 [((C9 + c) + b) >> 5]
pred[9, 6] = Clip1 [((C9 + c) + 2b) >> 5]
pred[9, 7] = Clip1 [((C9 + c) + 3b) >> 5]

pred[10, 4] = Clip1 [(C9 + 2c) >> 5]
pred[10, 5] = Clip1 [((C9 + 2c) + b) >> 5]
pred[10, 6] = Clip1 [((C9 + 2c) + 2b) >> 5]
pred[10, 7] = Clip1 [((C9 + 2c) + 3b) >> 5]

pred[11, 4] = Clip1 [(C9 + 3c)>> 5]
pred[11, 5] = Clip1 [((C9 + 3c) + b) >> 5]
pred[11, 6] = Clip1 [((C9 + 3c) + 2b) >> 5]
pred[11, 7] = Clip1 [((C9 + 3c) + 3b) >> 5]

**C10 = [a – (7 * b) + (5 * c) + 16] = C0 + 12c**

pred[12, 0] = Clip1 [(C10) >> 5]
pred[12, 1] = Clip1 [(C10 + b) >> 5]
pred[12, 2] = Clip1 [(C10 + 2b) >> 5]
pred[12, 3] = Clip1 [(C10 + 3b) >> 5]

pred[13, 0] = Clip1 [(C10 + c) >> 5]
pred[13, 1] = Clip1 [((C10 + c) + b) >> 5]
pred[13, 2] = Clip1 [((C10 + c) + 2b) >> 5]
pred[13, 3] = Clip1 [((C10 + c) + 3b) >> 5]

pred[14, 0] = Clip1 [(C10 + 2c) >> 5]
pred[14, 1] = Clip1 [((C10 + 2c) + b) >> 5]
pred[14, 2] = Clip1 [((C10 + 2c) + 2b) >> 5]
pred[14, 3] = Clip1 [((C10 + 2c) + 3b) >> 5]

pred[15, 0] = Clip1 [(C10 + 3c) >> 5]
pred[15, 1] = Clip1 [((C10 + 3c) + b) >> 5]
pred[15, 2] = Clip1 [((C10 + 3c) + 2b) >> 5]
pred[15, 3] = Clip1 [((C10 + 3c) + 3b) >> 5]

**C11 = [a – (3 * b) + (5 * c) + 16] = C0 + 4b + 12c**

pred[12, 4] = Clip1 [(C11) >> 5]
pred[12, 5] = Clip1 [(C11 + b) >> 5]
pred[12, 6] = Clip1 [(C11 + 2b) >> 5]
pred[12, 7] = Clip1 [(C11 + 3b) >> 5]

pred[13, 4] = Clip1 [(C11 + c) >> 5]
pred[13, 5] = Clip1 [((C11 + c) + b) >> 5]
pred[13, 6] = Clip1 [((C11 + c) + 2b) >> 5]
pred[13, 7] = Clip1 [((C11 + c) + 3b) >> 5]

pred[14, 4] = Clip1 [(C11 + 2c) >> 5]
pred[14, 5] = Clip1 [((C11 + 2c) + b) >> 5]
pred[14, 6] = Clip1 [((C11 + 2c) + 2b) >> 5]
pred[14, 7] = Clip1 [((C11 + 2c) + 3b) >> 5]

pred[15, 4] = Clip1 [(C11 + 3c)>> 5]
pred[15, 5] = Clip1 [((C11 + 3c) + b) >> 5]
pred[15, 6] = Clip1 [((C11 + 3c) + 2b) >> 5]
pred[15, 7] = Clip1 [((C11 + 3c) + 3b) >> 5]

**C12 = [a + (1 * b) + (1 * c) + 16] = C0 + 8b + 8c**

pred[8, 8] = Clip1 [(C12)>> 5]
pred[8, 9] = Clip1 [(C12 + b) >> 5]
pred[8, 10] = Clip1 [(C12 + 2b) >> 5]
pred[8, 11] = Clip1 [(C12 + 3b) >> 5]

pred[9, 8] = Clip1 [(C12 + c)>> 5]
pred[9, 9] = Clip1 [((C12 + c) + b) >> 5]
pred[9, 10] = Clip1 [((C12 + c) + 2b) >> 5]
pred[9, 11] = Clip1 [((C12 + c) + 3b) >> 5]

pred[10, 8] = Clip1 [(C12 + 2c)>> 5]
pred[10, 9] = Clip1 [((C12 + 2c) + b) >> 5]
pred[10, 10] = Clip1 [((C12 + 2c) + 2b) >> 5]
pred[10, 11] = Clip1 [((C12 + 2c) + 3b) >> 5]

pred[11, 8] = Clip1 [(C12 + 3c)>> 5]
pred[11, 9] = Clip1 [((C12 + 3c) + b) >> 5]
pred[11, 10] = Clip1 [((C12 + 3c) + 2b) >> 5]
pred[11, 11] = Clip1 [((C12 + 3c) + 3b) >> 5]

**C13 = [a + (5 * b) + (1 * c) + 16] = C0 + 12b + 8c**

pred[8, 12] = Clip1 [(C13) >> 5]
pred[8, 13] = Clip1 [(C13 + b) >> 5]
pred[8, 14] = Clip1 [(C13 + 2b) >> 5]
pred[8, 15] = Clip1 [(C13 + 3b) >> 5]

pred[9, 12] = Clip1 [(C13 + c)>> 5]
pred[9, 13] = Clip1 [((C13 + c) + b) >> 5]
pred[9, 10] = Clip1 [((C13 + c) + 2b) >> 5]
pred[9, 11] = Clip1 [((C13 + c) + 3b) >> 5]

pred[10, 12] = Clip1 [(C13 + 2c) >> 5]
pred[10, 13] = Clip1 [((C13 + 2c) + b) >> 5]
pred[10, 14] = Clip1 [((C13 + 2c) + 2b) >> 5]
pred[10, 15] = Clip1 [((C13 + 2c) + 3b) >> 5]

pred[11, 12] = Clip1 [(C13 + 3c) >> 5]
pred[11, 13] = Clip1 [((C13 + 3c) + b) >> 5]
pred[11, 14] = Clip1 [((C13 + 3c) + 2b) >> 5]
pred[11, 15] = Clip1 [((C13 + 3c) + 3b) >> 5]

**C14 = [a + (1 * b) + (5 * c) + 16] = C0 + 8b + 12c**

pred[12, 8] = Clip1 [(C14)>> 5]
pred[12, 9] = Clip1 [(C14 + b) >> 5]
pred[12, 10] = Clip1 [(C14 + 2b) >> 5]
pred[12, 11] = Clip1 [(C14 + 3b) >> 5]

pred[13, 8] = Clip1 [(C14 + c)>> 5]
pred[13, 9] = Clip1 [((C14 + c) + b) >> 5]
pred[13, 10] = Clip1 [((C14 + c) + 2b) >> 5]
pred[13, 11] = Clip1 [((C14 + c) + 3b) >> 5]

pred[14, 8] = Clip1 [(C14 + 2c)>> 5]
pred[14, 9] = Clip1 [((C14 + 2c) + b) >> 5]
pred[14, 10] = Clip1 [((C14 + 2c) + 2b) >> 5]
pred[14, 11] = Clip1 [((C14 + 2c) + 3b) >> 5]

pred[15, 8] = Clip1 [(C14 + 3c)>> 5]
pred[15, 9] = Clip1 [((C14 + 3c) + b) >> 5]
pred[15, 10] = Clip1 [((C14 + 3c) + 2b) >> 5]
pred[15, 11] = Clip1 [((C14 + 3c) + 3b) >> 5]

**C15 = [a + (5 * b) + (5 * c) + 16] = C0 + 12b + 12c**

pred[12, 12] = Clip1 [(C15) >> 5]
pred[12, 13] = Clip1 [(C15 + b) >> 5]
pred[12, 14] = Clip1 [(C15 + 2b) >> 5]
pred[12, 15] = Clip1 [(C15 + 3b) >> 5]

pred[13, 12] = Clip1 [(C15 + c)>> 5]
pred[13, 13] = Clip1 [((C15 + c) + b) >> 5]
pred[13, 10] = Clip1 [((C15 + c) + 2b) >> 5]
pred[13, 11] = Clip1 [((C15 + c) + 3b) >> 5]

pred[14, 12] = Clip1 [(C15 + 2c) >> 5]
pred[14, 13] = Clip1 [((C15 + 2c) + b) >> 5]
pred[14, 14] = Clip1 [((C15 + 2c) + 2b) >> 5]
pred[14, 15] = Clip1 [((C15 + 2c) + 3b) >> 5]

pred[15, 12] = Clip1 [(C15 + 3c) >> 5]
pred[15, 13] = Clip1 [((C15 + 3c) + b) >> 5]
pred[15, 14] = Clip1 [((C15 + 3c) + 2b) >> 5]
pred[15, 15] = Clip1 [((C15 + 3c) + 3b) >> 5]

**Figure 2.12** Organized Prediction Equations for 16x16 Luma Plane Mode

The proposed hardware first calculates the common parts C0, (C0 + b), (C0 + 2b), and (C0 + 3b) and stores them in temporary registers. It, then, generates the predicted pixels in the first row by using the values stored in these temporary registers. The proposed hardware, then, adds c to the values stored in the temporary registers and stores the resulting values in the same temporary registers. It, then, generates the predicted pixels in the second row by using the values stored in these temporary registers. The proposed hardware repeats this process until all the predicted pixels for the current MB are generated.

The proposed datapath for generating predicted pixels for a 16x16 luma block using all 16x16 luma prediction modes is shown in Figure 2.13. REG0 - REG7 registers are used to store the results of the common parts in the equations. The neighboring reconstructed pixels stored in the neighboring buffers are given as inputs to the datapath. If both the left and top neighboring MBs of a 16x16 luma block are available, it takes 1127 clock cycles to generate the predicted pixels for that 16x16 luma block using available 16x16 luma

prediction modes. Clock cycles required for performing available 16x16 luma predictions based on the availability of neighboring 16x16 luma blocks for a 16x16 luma block are given in Table 2.5.

Plane mode is the most computationally demanding 16x16 luma prediction mode. Therefore, using two parallel adders and shifters in the proposed datapath is especially important for Plane mode. The predicted pixels for a 16x16 luma block are generated in 340 clock cycles using Plane mode.

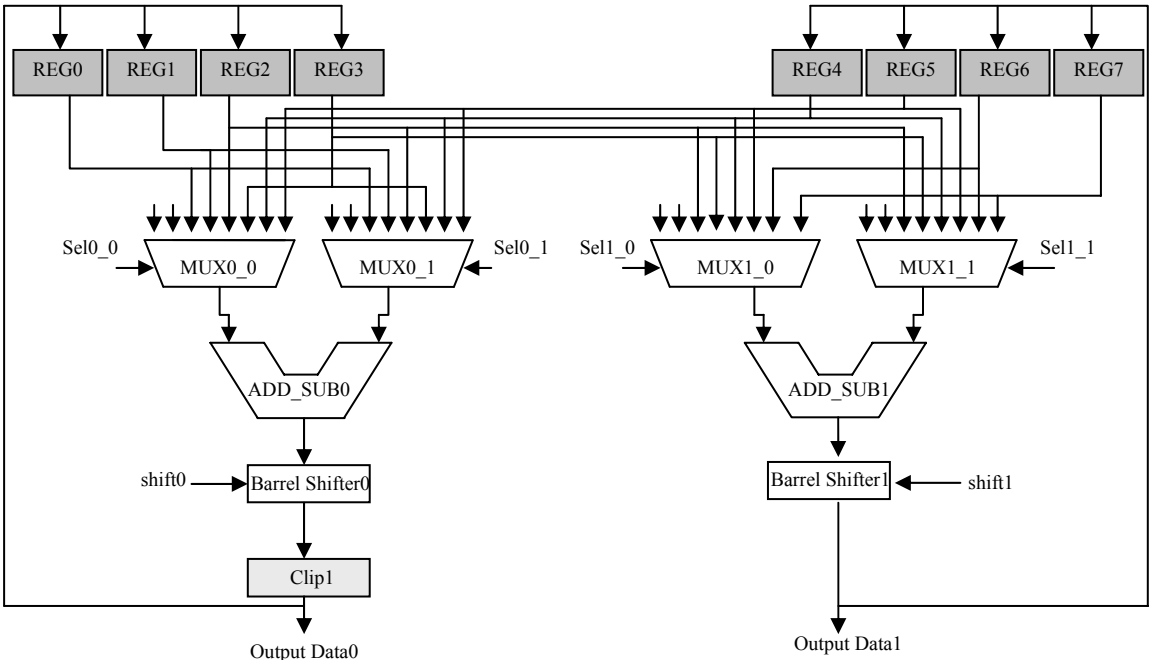



**Figure 2.13** Datapath for 16x16 Luma Prediction Modes

**Table 2.5** Clock Cycles Required for Performing Available 16x16 Luma Prediction Modes

| **Available Modes** | **Clock Cycles/MB** |
|---|---|
| Vertical | 257 |
| Horizontal | 257 |
| DC | 273 |
| Plane Mode | 340 |

(a) Top and Left Neighboring 16x16 Luma Blocks are available

| Available Modes | Clock Cycles/MB |
|---|---|
| Vertical | 257 |
| DC | 265 |

(b)Top Neighboring 16x16 Luma Block is available

| Available Modes | Clock Cycles/MB |
|---|---|
| Horizontal | 257 |
| DC | 265 |

(c) Left Neighboring 16x16 Luma Block is available

| Available Modes | Clock Cycles/MB |
|---|---|
| DC | 257 |

(d) Top and Left Neighboring 16x16 Luma Blocks are not available

#### 2.2.1.3 Proposed Hardware for 8x8 Chroma Prediction Modes

Since the 8x8 chroma prediction modes are similar to 16x16 luma prediction modes, the proposed hardware for 8x8 chroma prediction modes is also similar to the proposed hardware for 16x16 luma prediction modes. If both the left and top neighboring MBs of an 8x8 chroma block are available, it takes 302 clock cycles to generate the predicted pixels for that 8x8 chroma block using available 8x8 chroma prediction modes. Plane mode is also the most computationally demanding 8x8 chroma prediction mode.

The organized prediction equations for Plane mode are shown in Figure 2.14. The predicted pixels for an 8x8 chroma block are generated in 95 clock cycles using Plane mode. Clock cycles required for preprocessing and performing available 8x8 chroma predictions based on the availability of neighboring 8x8 chroma blocks for an 8x8 chroma block are given in Table 2.6.

a = (*p*[-1,15] + *p*[15,-1]) << 4, b = [(H << 2) + (H + 32)] >> 6
c = [(V << 2) + (V + 32)] >> 6

**C0 = [a – (3 * b) - (3 * c) + 16]**

pred[0, 0] = Clip1 [(C0) >> 5]
pred[0, 1] = Clip1 [(C0 + b) >> 5]
pred[0, 2] = Clip1 [(C0 + 2b) >> 5]
pred[0, 3] = Clip1 [(C0 + 3b) >> 5]

pred[1, 0] = Clip1 [(C0 + c) >> 5]
pred[1, 1] = Clip1 [((C0 + c) + b) >> 5]
pred[1, 2] = Clip1 [((C0 + c) + 2b) >> 5]
pred[1, 3] = Clip1 [((C0 + c) + 3b) >> 5]

pred[2, 0] = Clip1 [(C0 + 2c) >> 5]
pred[2, 1] = Clip1 [((C0 + 2c) + b) >> 5]
pred[2, 2] = Clip1 [((C0 + 2c) + 2b) >> 5]
pred[2, 3] = Clip1 [((C0 + 2c) + 3b) >> 5]

pred[3, 0] = Clip1 [(C0 + 3c)>> 5]
pred[3, 1] = Clip1 [((C0 + 3c) + b) >> 5]
pred[3, 2] = Clip1 [((C0 + 3c) + 2b) >> 5]
pred[3, 3] = Clip1 [((C0 + 3c) + 3b) >> 5]

**C1 = [a + (1 * b) - (3 * c) + 16] = C0 + 4b**

pred[0, 4] = Clip1 [(C1) >> 5]
pred[0, 5] = Clip1 [(C1 + b) >> 5]
pred[0, 6] = Clip1 [(C1 + 2b) >> 5]
pred[0, 7] = Clip1 [(C1 + 3b) >> 5]
pred[1, 4] = Clip1 [(C1 + c) >> 5]
pred[1, 5] = Clip1 [((C1 + c) + b) >> 5]
pred[1, 6] = Clip1 [((C1 + c) + 2b) >> 5]
pred[1, 7] = Clip1 [((C1 + c) + 3b) >> 5]

pred[2, 4] = Clip1 [(C1 + 2c) >> 5]
pred[2, 5] = Clip1 [((C1 + 2c) + b) >> 5]
pred[2, 6] = Clip1 [((C1 + 2c) + 2b) >> 5]
pred[2, 7] = Clip1 [((C1 + 2c) + 3b) >> 5]

pred[3, 4] = Clip1 [(C1 + 3c) >> 5]
pred[3, 5] = Clip1 [((C1 + 3c) + b) >> 5]
pred[3, 6] = Clip1 [((C1 + 3c) + 2b) >> 5]
pred[3, 7] = Clip1 [((C1 + 3c) + 3b) >> 5]

**C2 = [a – (3 * b) + (1 * c) + 16] = C0 + 4c**

pred[4, 0] = Clip1 [(C2) >> 5]
pred[4, 1] = Clip1 [(C2 + b) >> 5]
pred[4, 2] = Clip1 [(C2 + 2b) >> 5]
pred[4, 3] = Clip1 [(C2 + 3b) >> 5]

pred[5, 0] = Clip1 [(C2 + c) >> 5]
pred[5, 1] = Clip1 [((C2 + c) + b) >> 5]
pred[5, 2] = Clip1 [((C2 + c) + 2b) >> 5]
pred[5, 3] = Clip1 [((C2 + c) + 3b) >> 5]

pred[6, 0] = Clip1 [(C2 + 2c) >> 5]
pred[6, 1] = Clip1 [((C2 + 2c) + b) >> 5]
pred[6, 2] = Clip1 [((C2 + 2c) + 2b) >> 5]
pred[6, 3] = Clip1 [((C2 + 2c) + 3b) >> 5]

pred[7, 0] = Clip1 [(C2 + 3c) >> 5]
pred[7, 1] = Clip1 [((C2 + 3c) + b) >> 5]
pred[7, 2] = Clip1 [((C2 + 3c) + 2b) >> 5]
pred[7, 3] = Clip1 [((C2 + 3c) + 3b) >> 5]

**C3 = [a + (1 * b) + (1 * c) + 16] = C0 + 4c + 4b**

pred[4, 4] = Clip1 [(C3) >> 5]
pred[4, 5] = Clip1 [(C3 + b) >> 5]
pred[4, 6] = Clip1 [(C3 + 2b) >> 5]
pred[4, 7] = Clip1 [(C3 + 3b) >> 5]

pred[5, 4] = Clip1 [(C3 + c) >> 5]
pred[5, 5] = Clip1 [((C3 + c) + b) >> 5]
pred[5, 6] = Clip1 [((C3 + c) + 2b) >> 5]
pred[5, 7] = Clip1 [((C3 + c) + 3b) >> 5]

pred[6, 4] = Clip1 [(C3 + 2c)>> 5]
pred[6, 5] = Clip1 [((C3 + 2c) + b) >> 5]
pred[6, 6] = Clip1 [((C3 + 2c) + 2b) >> 5]
pred[6, 7] = Clip1 [((C3 + 2c) + 3b) >> 5]

pred[7, 4] = Clip1 [(C3 + 3c)>> 5]
pred[7, 5] = Clip1 [((C3 + 3c) + b) >> 5]
pred[7, 6] = Clip1 [((C3 + 3c) + 2b) >> 5]
pred[7, 7] = Clip1 [((C3 + 3c) + 3b) >> 5]

**Figure 2.14** Organized Prediction Equations for 8x8 Luma Plane Mode

**Table 2.6** Clock Cycles Required for Performing Available 8x8 Chroma Prediction Modes

| Available Modes | Clock Cycles/MB |
|---|---|
| Vertical | 65 |
| Horizontal | 65 |
| DC | 77 |
| Plane Mode | 95 |

(a) Top and Left Neighboring 8x8 Chroma Blocks are available

| Available Modes | Clock Cycles/MB |
|---|---|
| Vertical | 77 |
| DC | 65 |

(b)Top Neighboring 8x8 Chroma Block is available

| Available Modes | Clock Cycles/MB |
|---|---|
| Horizontal | 77 |
| DC | 65 |

(c)Left Neighboring 8x8 Chroma Block is available

| Available Modes | Clock Cycles/MB |
|---|---|
| DC | 65 |

(d) Top and Left Neighboring 8x8 Chroma Blocks are not available

#### 2.2.1.4 Implementation Results

A high performance and low cost hardware architecture for real-time implementation of intra prediction algorithm used in H.264 / MPEG4 Part 10 video coding standard is designed. The hardware design is based on a novel organization of the intra prediction equations. This hardware is designed to be used as part of a complete H.264 intra frame coder for portable applications.

The proposed architecture is implemented in Verilog HDL. The implementation is verified with RTL simulations using Mentor Graphics ModelSim SE. The Verilog RTL is then synthesized to a 2V8000ff1152 Xilinx Virtex II FPGA with speed grade 5 using Mentor Graphics Leonardo Spectrum. The resulting netlist is placed and routed to the same FPGA using Xilinx ISE Series 7.1i.

Clock cycle requirements of available 16x16 luma prediction modes for luma component of a MB depending on the availability of the neighboring 16x16 luma blocks are given in Table 2.7.

**Table 2.7** Clock Cycles Required for Performing Available 16x16 Luma Prediction Modes

| **Availability of Neighboring 16x16 Luma Blocks** | **Clock Cycles/MB** |
|---|---|
| None available | 257 |
| Left available, Top not available | 522 |
| Top available, Left not available | 522 |
| Both available | 1127 |

Clock cycle requirements of available 8x8 chroma prediction modes for chroma components of a MB depending on the availability of the neighboring 8x8 chroma blocks are given in Table 2.8.

**Table 2.8** Clock Cycles Required for Performing Available 8x8 Chroma Prediction Modes

| Availability of Neighboring 8x8 Chroma Blocks | Clock Cycles/MB |
|---|---|
| None available | 65 |
| Left available, Top not available | 142 |
| Top available, Left not available | 142 |
| Both available | 302 |

Clock cycle requirements of available 4x4 luma prediction modes for luma component of a MB depending on the availability of the neighboring 16x16 luma blocks are given in Table 2.9.

**Table 2.9** Clock Cycles Required for Performing Available 4x4 Luma Prediction Modes

| Availability of Neighboring 16x16 Luma Block | Clock Cycles/MB |
|---|---|
| None available | 1910 |
| Left available, Top not available | 1980 |
| Top available, Left not available | 1797 |
| Both available | 2640 |

As it can be seen from Table 2.7, 2.8, and 2.9, generating the predicted pixels using available 4x4 luma prediction modes for luma component of a MB is the most computationally demanding part in the proposed hardware. In the worst case, regardless of the availability of the neighboring pixels, generating the predicted pixels for a MB using 4x4 luma prediction modes is the bottleneck.

In a VGA frame (40x30 = 1200 MBs), there is only 1 MB (MB0) which has no available neighboring MBs, there are 39 MBs (the first row of MBs except MB0) which have only left-hand neighboring MBs available, there are 29 MBs (the first column of MBs except MB0) which have only upper neighboring MBs available, and there are 1131 MBs (remaining MBs) which have both left-hand and upper neighboring MBs available.

In addition to the number clock cycles given in Table 2.9, 16 clock cycles are required for loading the neighboring reconstructed pixels to the corresponding neighboring buffers for each 4x4 luma block. Therefore, generating the predicted pixels for a VGA frame using 4x4 luma prediction modes takes 1910 + (1980 x 39) + (1797 x 29) + (2640 x 1131) + (16x16x1200) = 3424283 clock cycles.

The FPGA implementation is verified to work at 90 MHz under worst-case PVT conditions with post place and route simulations. Since, in the proposed hardware, generating the predicted pixels for a MB using 4x4 luma prediction modes is the bottleneck, the FPGA implementation can process a VGA frame in 3424283 clock cycles per VGA frame x 11 ns clock cycle = 37.6 msec. Therefore, it can process 1000/37.6 = 27 VGA frames (640x480) per second.

The FPGA implementation including input and output register files and internal RAMs uses the following FPGA resources; 2002 Function Generators, 1001 CLB Slices, and 518 DFFs, i.e. %2.15 of Function Generators, %2.15 of CLB Slices, and %0.54 of DFFs.

### 2.2.2 Intra Prediction Hardware for Coder

The block diagram of the proposed intra prediction hardware architecture for the coder part of the H.264 intra frame coder is shown in Figure 2.15. The proposed hardware generates the predicted pixels for luma component of a MB using either selected 16x16 or 4x4 luma prediction modes. It generates the predicted pixels for chroma components of a MB using the selected 8x8 chroma prediction mode with different configurations. In the proposed hardware, there are two parts operating in parallel in order to perform intra prediction faster.

The upper part is used for generating the predicted pixels for the luma component of a MB using the selected 16x16 luma prediction mode if 16x16 luma prediction is selected for luma components of the MB by the mode decision hardware and for generating the predicted pixels for the chroma components of a MB using the selected 8x8 chroma prediction mode for the chroma components of the MB by the mode decision hardware.

The lower part is used for generating the predicted pixels for each 4x4 block in the luma component of a MB using the selected 4x4 luma prediction modes for each 4x4 luma block in a MB if 4x4 luma prediction is selected for the luma components of a MB by the

mode decision hardware. The lower part is more computationally demanding and it is the bottleneck in the proposed intra prediction hardware.

The size of register file that is used for the prediction buffer is 384x8, because they are used for storing both luma and chroma components of the predicted MB.

Two local neighboring buffers, local vertical register file and local horizontal register file, are used to store the neighboring pixels in the previously coded and reconstructed neighboring 4x4 luma blocks in the predicted MB. After a 4x4 luma block in the current MB is coded and reconstructed, the neighboring pixels in this block are stored in the corresponding local register files.

Local vertical register file is used to store the neighboring pixels d, h, l, and p in the left-hand previously coded and reconstructed neighboring 4x4 luma blocks in the predicted MB. Local horizontal register file is used to store the neighboring pixels m, n, o, and p in the upper previously coded and reconstructed 4x4 luma blocks in the predicted MB. The proposed hardware uses this data to determine the neighboring pixels in the left-hand and upper previously coded neighboring 4x4 luma blocks in the current MB.

Six global neighboring buffers, three global vertical neighboring buffers and three global horizontal neighboring buffers, are used to store the neighboring pixels in the previously coded and reconstructed neighboring MBs of the current MB.

Global luma vertical register file is used to store the neighboring pixels d, h, l, and p in the 4x4 luma blocks 5, 7, 13 and 15 of the previously coded MB. The proposed hardware uses this data to determine the neighboring pixels in the left-hand previously coded neighboring MB of the 4x4 luma blocks 0, 2, 8, and 10 in the current MB. Global Cb vertical register file and global Cr vertical register file are used for the chroma Cb and chroma Cr components of the MBs.

Global luma horizontal register file is used to store the neighboring pixels m, n, o, and p in the luma blocks 10, 11, 14, and 15 of the previously coded MBs in the previously coded MB row of the frame. The proposed hardware uses this data to determine the neighboring pixels in the upper previously coded neighboring MB of the 4x4 luma blocks 0, 1, 4, and 5 in the current MB. Global Cb horizontal register file and global Cr horizontal register file are used for the chroma Cb and chroma Cr components of the MBs.

Instead of using one large external SRAM, we have used 8 internal register files to store the neighboring reconstructed pixels in order to reduce power consumption. The power consumption is reduced by accessing a small register file for storing and reading a reconstructed pixel instead of accessing a large external SRAM. In addition, we have disabled the register files when they are not accessed in order to reduce power consumption.

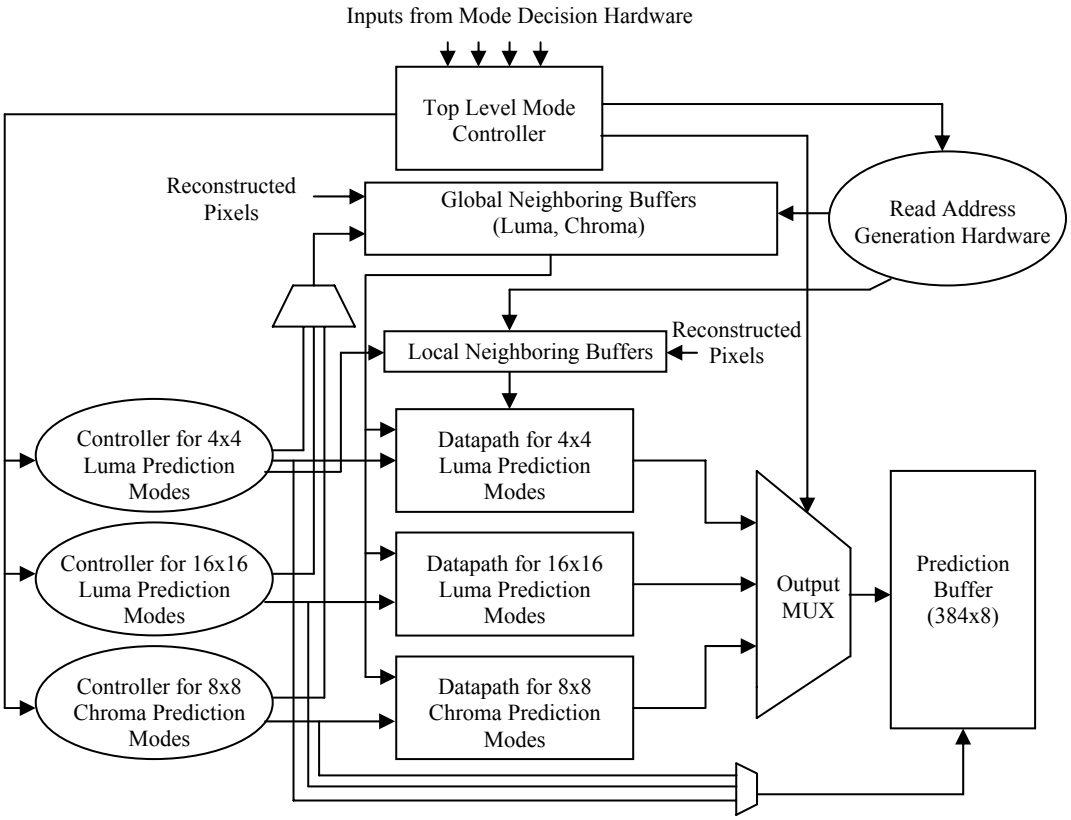



**Figure 2.15** Intra Prediction Hardware for Coder

#### 2.2.2.1 Proposed Hardware for 4x4 Luma Prediction Modes

After a careful analysis of the equations used in 4x4 luma prediction modes, it is observed that there are common parts in the equations and some of the equations are identical in each 4x4 luma prediction mode. The intra prediction equations are organized for exploiting these observations to reduce both the number of memory accesses and computation time required for generating the predicted pixels. The organized prediction equations for 4x4 luma prediction modes are shown in Figure 2.16.

Pred[0, 0] = Pred[1, 0] = Pred[2, 0] = Pred[3, 0] = A

Pred[0, 1] = Pred[1, 1] = Pred[2, 1] = Pred[3, 1] = B

Pred[0, 2] = Pred[1, 2] = Pred[2, 2] = Pred[3, 2] = C

Pred[0, 3] = Pred[1, 3] = Pred[2, 3] = Pred[3, 3] = D

(a) 4x4 Vertical Prediction Mode

Pred[0, 0] = Pred[0, 1] = Pred[0, 2] = Pred[0, 3] = I

Pred[1, 0] = Pred[1, 1] = Pred[1, 2] = Pred[1, 3] = J

Pred[2, 0] = Pred[2, 1] = Pred[2, 2] = Pred[2, 3] = K

Pred[3, 0] = Pred[3, 1] = Pred[3, 2] = Pred[3, 3] = L

(b) 4x4 Horizontal Prediction Mode

pred[y,x] = [(A + B) + (C + D) + (I + J) + (K + L) + 4] >> 3
(If the left and the top neighboring pixels are available)

Pred[y, x] = [(I + J) + (K + L) + 2] >> 2
(Else If only the left neighboring pixels are available)

pred[y, x] = [(A + B) + (C + D) + 2] >> 2
(Else If only the top neighboring pixels are available)

pred[y,x] = 128
(Else //If the left and the top neighboring pixels are not available)

(c) 4x4 DC Prediction Mode

| | |
|---|---|
| Pred[0, 0] | = [(A + B) + (B + C) + 2] >> 2 |
| Pred[0, 1] = Pred[1, 0] | = [(C + D) + (B + C) + 2] >> 2 |
| Pred[0, 2] = Pred[1, 1] = Pred[2, 0] | = [(C + D) + (D + E) + 2] >> 2 |
| Pred[0, 3] = Pred[1, 2] = Pred[2, 1] | = [(E + F) + (D + E) + 2] >> 2 |
| Pred[3, 0] | = [(E + F) + (D + E) + 2] >> 2 |
| Pred[1, 3] = Pred[2, 2] = Pred[3, 1] | = [(E + F) + (F + G) + 2] >> 2 |
| Pred[2, 3] = Pred[3, 2] | = [(G + H) + (F + G) + 2] >> 2 |
| Pred[3, 3] | = [(G + H) + (H +H) + 2] >> 2 |

(c) 4x4 Diagonal Down-Left Mode

| | |
|---|---|
| Pred[0, 2] = Pred[1, 3] | = [(A + B) + (B + C) + 2] >> 2 |
| Pred[0, 3] | = [(C + D) + (B + C) + 2] >> 2 |
| Pred[3, 0] | = [(J + K) + (K+ L) + 2] >> 2 |
| Pred[2, 0] = Pred[3, 1] | = [(J + K) + (I + J) + 2] >> 2 |
| Pred[1, 0] = Pred[2, 1] = Pred[3, 2] | = [(M + I) + (I + J) + 2] >> 2 |
| Pred[0, 0] = Pred[1, 1] = Pred[2, 2] = Pred[3, 3] | = [(M + I) + (M + A) + 2] >> 2 |
| Pred[0, 1] = Pred[1, 2] = Pred[2, 3] | = [(A + B) (M + A) + + 2] >> 2 |

(d) 4x4 Diagonal Down-Right Mode

```
Pred[3, 0]                  = [(I   + J)  + (J + K)  + 2] >> 2
Pred[2, 0]                  = [(I   + J)  + (M + I)  + 2] >> 2
Pred[1, 0] = Pred[3, 1]     = [(M + A) + (M + I) + 2] >> 2
Pred[1, 1] = Pred[3, 2]     = [(M + A) + (A + B) + 2] >> 2
Pred[1, 2] = Pred[3, 3]     = [(B + C) + (A + B) + 2] >> 2
Pred[1, 3]                  = [(B + C) + (C + D) + 2] >> 2
Pred[0, 2] = Pred[2, 3]     = [(B + C) + 1] >> 1
Pred[0, 1] = Pred[2, 1]     = [(A + B) + 1] >> 1
Pred[0, 3]                  = [(C + D) + 1] >> 1
Pred[0, 0] = Pred[2, 1]     = [(M + A) + 1] >> 1
```

(f) 4x4 Vertical Right Mode

```
Pred[0, 0] = Pred[1, 2]     = [(M + I) + 1] >> 1
Pred[1, 0] = Pred[2, 2]     = [(I + J) + 1] >> 1
Pred[2, 0] = Pred[3, 2]     = [(J + K) + 1] >> 1
Pred[3, 0]                  = [(K + L) + 1] >> 1
Pred[3, 1]                  = [(K + L) + (J + K) + 2)] >> 2
Pred[2, 1] = Pred[3, 3]     = [(I + J) + (J + K) + 2] >> 2
Pred[1, 1] = Pred[2, 3]     = [(I + J) + (M + I) + 2] >> 2
Pred[0, 1] = Pred[1, 3]     = [(M + A) + (M + I) + 2] >> 2
Pred[0, 2]                  = [(M + A) + (A + B) + 2] >> 2
Pred[0, 3]                  = [(B + C) + (A + B) + 2] >> 2
```

(g) 4x4 Horizontal Down Mode

```
Pred[1, 0]                  = [(A + B) + (B + C) + 2] >> 2
Pred[1, 1] = Pred[3, 0]     = [(C + D) + (B + C) + 2] >> 2
Pred[1, 2] = Pred[3, 1]     = [(C + D) + (D + E) + 2] >> 2
Pred[1, 3] = Pred[3, 2]     = [(E + F) + (D + E) + 2] >> 2
Pred[3, 3]                  = [(E + F) + (F + G) + 2] >> 2
Pred[2, 3]                  = [(E + F) + 1] >> 1
Pred[0, 0]                  = [(A + B) + 1] >> 1
Pred[0, 1] = Pred[2, 0]     = [(B + C) + 1] >> 1
Pred[0, 2] = Pred[2, 1]     = [(C + D) + 1] >> 1
Pred[0, 3] = Pred[2, 2]     = [(D + E) + 1] >> 1
```

(h) 4x4 Vertical Left Mode

Pred[0, 1] = [(I + J) + (J + K) + 2] >> 2
Pred[0, 3] = Pred[1, 1] = [(J + K) + (K + L) + 2)] >> 2
Pred[1, 3] = Pred[2, 1] = [(L + L) + (K + L) + 2)] >> 2
Pred[1, 2] = Pred[2, 0] = [1 + (K+ L)] >> 1
Pred[0, 0] = [1 + (I + J)] >> 1
Pred[0, 2] = Pred[1, 0] = [1 + (J + K)] >> 1
Pred[2, 2] = Pred[2, 3] = Pred[3, 0] = Pred[3, 1]
= Pred[3, 2] = Pred[3, 3] = L

(i) 4x4 Horizontal Up Mode

**Figure 2.16** Organized Prediction Equations for 4x4 Luma Prediction Modes

As it can be seen from the figure, (A + B), (B + C), (C + D), (D + E), (E + F), (F + G), (G + H), (J + K), (I + J), (M + I) and (M + A) are common in two or more equations, and some of the prediction equations (e.g. [(A + B) + (B + C) + 2] >> 2) are identical in each 4x4 luma prediction mode.

The proposed hardware starts to work if 4x4 luma prediction is selected for the luma component of a MB by the mode decision hardware. It first calculates the results of the common parts in the selected 4x4 luma prediction mode for a 4x4 luma block and stores them in temporary registers (i.e. performs preprocessing for the selected mode). It, then, calculates the results of the prediction equations in the selected 4x4 luma prediction mode for a 4x4 luma block using the values stored in these temporary registers. The neighboring buffers are only accessed during the preprocessing. Therefore, they are disabled after the preprocessing for reducing power consumption.

The proposed hardware calculates the results of the identical prediction equations in the selected 4x4 luma prediction mode for a 4x4 luma block only once. It, then, writes the predictions of the pixels specified by the identical prediction equations into the corresponding prediction buffer entries. Thus, no extra temporary registers are required for storing the results of the identical prediction equations in the selected 4x4 luma prediction mode for a 4x4 luma block.

The proposed datapath for generating predicted pixels for a 4x4 luma block using the selected 4x4 luma prediction modes for a 4x4 luma block is shown in Figure 2.17. REG0-REG7 are used to store the results of the common parts in the equations of the selected 4x4

luma prediction mode for a 4x4 luma block. Clock cycles required to perform each selected 4x4 luma prediction mode for a 4x4 luma block are given in Table 2.10. The clock cycles given in Table 2.10 includes the preprocessing in each selected mode. If both the left and top neighboring 4x4 luma blocks of a 4x4 luma block are available, it takes 24 clock cycles in the worst case to generate the predicted pixels for that 4x4 block using a selected 4x4 luma prediction mode.

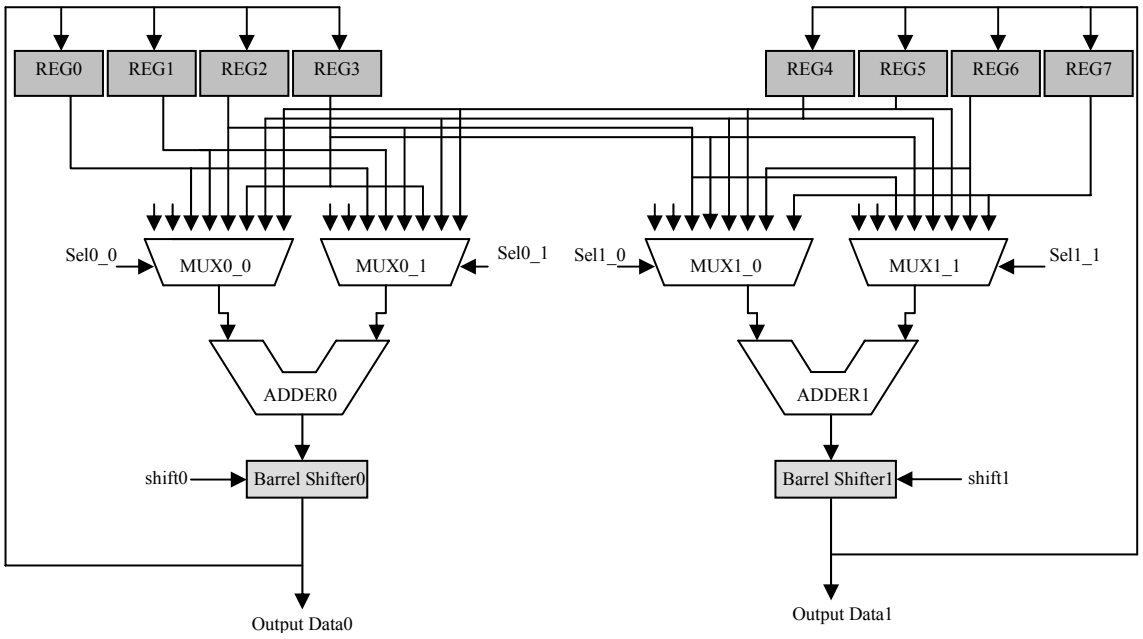



**Figure 2.17** Datapath for 4x4 Luma Prediction Modes

**Table 2.10** Clock cycles required for performing the selected 4x4 Luma Prediction Modes

| **Selected Mode** | **Clock Cycles / 4x4 Luma Block** |
|---|---|
| Vertical | 17 |
| Horizontal | 17 |
| DC | 21 |
| Diagonal Down-Left Mode | 24 |
| Diagonal Down-Right Mode | 24 |
| Vertical Right | 23 |
| Horizontal Down | 23 |
| Vertical Left | 22 |
| Horizontal Up | 20 |

(a) Top and Left Neighboring 4x4 Luma Blocks are available

| Selected Mode | Clock Cycles / 4x4 Luma Block |
|---|---|
| Vertical | 17 |
| DC | 21 |
| Diagonal Down-Left Mode | 24 |
| Vertical Left | 22 |

(b) Top Neighboring 4x4 Luma Block is available

| Selected Mode | Clock Cycles / 4x4 Luma Block |
|---|---|
| Horizontal | 17 |
| DC | 21 |
| Horizontal Up | 20 |

(c) Left Neighboring 4x4 Luma Block is available

| Selected Mode | Clock Cycles / 4x4 Luma Block |
|---|---|
| DC | 17 |

(d) Top and Left Neighboring 4x4 Luma Blocks are not available

Since the order of the equations used in a selected 4x4 luma prediction mode for a 4x4 luma block is not important for functional correctness, the equations are ordered to keep the inputs of the adders the same for as many consecutive clock cycles as possible. This avoids unnecessary switching activity and reduces the power consumption.

#### 2.2.2.2 Proposed Hardware for 16x16 Luma Prediction Mode

The proposed hardware for 16x16 luma prediction modes for the coder part of the H.264 intra frame coder is very similar to the one proposed in the section 2.2.1.2 for the search & mode decision part of the H.264 intra frame coder. The proposed hardware uses

the same datapath for 16x16 luma prediction modes given in 2.2.1.2. Control part is the main difference between the proposed hardware architectures for 16x16 luma prediction modes in the search & mode decision and the coder parts of the H.264 intra frame coder.

The proposed datapath for 16x16 luma prediction modes in the search & mode decision part is used for generating predicted pixels for a 16x16 luma block using all available 16x16 luma prediction modes. However, the proposed datapath for 16x16 luma prediction modes in the coder part is used for generating predicted pixels for a 16x16 luma block using only the selected 16x16 luma prediction mode if 16x16 luma prediction is selected for luma component of a MB by the mode decision hardware.

The organized prediction equations for Plane mode for the proposed hardware are same as the ones given in Figure 2.12.

Clock cycles required for generating the predicted pixels for a 16x16 luma block using the selected 16x16 luma prediction mode for luma component of a MB based on the availability of neighboring 16x16 luma blocks of that 16x16 luma block are given in Table 2.11.

Plane mode is the most computationally demanding 16x16 luma prediction mode. Therefore, using two parallel adders and shifters in the proposed datapath is especially important for Plane mode. The predicted pixels for a 16x16 luma block are generated in 340 clock cycles if Plane mode is selected for coding a 16x16 luma block.

**Table 2.11** Clock Cycles Required for Performing Selected 16x16 Luma Prediction Modes

| **Selected Mode** | **Clock Cycles/MB** |
|---|---|
| Vertical | 257 |
| Horizontal | 257 |
| DC | 273 |
| Plane Mode | 340 |

(a) Top and Left Neighboring 16x16 Luma Blocks are available

| **Selected Mode** | **Clock Cycles/MB** |
|---|---|
| Vertical | 257 |
| DC | 265 |

(b)Top Neighboring 16x16 Luma Block is available

| Selected Mode | Clock Cycles/MB |
|---|---|
| Horizontal | 257 |
| DC | 265 |

(c) Left Neighboring 16x16 Luma Block is available

| Selected Mode | Clock Cycles/MB |
|---|---|
| DC | 257 |

(d) Top and Left Neighboring 16x16 Luma Blocks are not available

#### 2.2.2.3 Proposed Hardware for 8x8 Chroma Prediction Modes

The proposed hardware for 8x8 chroma prediction modes for the coder part of the H.264 intra frame coder is very similar to the one proposed in the section 2.2.1.3 for the search & mode decision part of the H.264 intra frame coder. The proposed hardware uses the same datapath used for 8x8 chroma prediction modes in the search & mode decision part of the intra frame coder. Control part is the main difference between the proposed hardware architectures for 8x8 luma prediction modes in the search & mode decision and the coder part the H.264 intra frame coder.

The proposed datapath for 8x8 luma prediction modes in the search & mode decision part is used for generating predicted pixels for a 8x8 chroma block using all available 8x8 chroma prediction modes. However, the proposed datapath for 8x8 chroma prediction modes in the coder part is used for generating predicted pixels for an 8x8 chroma block using only the selected 8x8 luma prediction mode.

The organized prediction equations for Plane mode for the proposed hardware are same as the ones given in Figure 2.14.

Clock cycles required for generating the predicted pixels for an 8x8 chroma block using the selected 8x8 chroma prediction mode for chroma components of a MB based on the availability of neighboring 8x8 chroma blocks of that 8x8 chroma block are given in Table 2.12.

Plane mode is the most computationally demanding 8x8 chroma prediction mode. Therefore, using two parallel adders and shifters in the proposed datapath is especially important for Plane mode. The predicted pixels for an 8x8 chroma block are generated in 95 clock cycles if Plane mode is selected for coding an 8x8 chroma block.

**Table 2.12** Clock Cycles Required for Performing Selected 8x8 Chroma Prediction Modes

| **Selected Mode** | **Clock Cycles/MB** |
|---|---|
| Vertical | 65 |
| Horizontal | 65 |
| DC | 77 |
| Plane Mode | 95 |

(a) Top and Left Neighboring 8x8 Chroma Blocks are available

| **Selected Mode** | **Clock Cycles/MB** |
|---|---|
| Vertical | 77 |
| DC | 65 |

(b)Top Neighboring 8x8 Chroma Block is available

| **Selected Mode** | **Clock Cycles/MB** |
|---|---|
| Horizontal | 77 |
| DC | 65 |

(c) Left Neighboring 8x8 Chroma Block is available

| **Selected Mode** | **Clock Cycles/MB** |
|---|---|
| DC | 65 |

(d) Top and Left Neighboring 8x8 Chroma Blocks are not available

#### 2.2.2.4 Implementation Results

A high performance and low cost hardware architecture for real-time implementation of intra prediction algorithm used in H.264 / MPEG4 Part 10 video coding standard is designed. The hardware design is based on a novel organization of the intra prediction equations. This hardware is designed to be used as part of a complete H.264 intra frame coder for portable applications. The proposed architecture is implemented in Verilog HDL. The implementation is verified with RTL simulations using Mentor Graphics ModelSim SE. The Verilog RTL is then synthesized to a 2V8000ff1152 Xilinx Virtex II FPGA with speed grade 5 using Mentor Graphics Leonardo Spectrum. The resulting netlist is placed and routed to the same FPGA using Xilinx ISE Series 7.1i.

Generating the pixel predictions using selected 4x4 luma prediction modes is the most computationally demanding part in the proposed hardware. The proposed hardware generates the predicted pixels of a MB in the worst case in 24x16 = 384, 340, and 95 clock cycles using 4x4 luma, 16x16 luma, and 8x8 chroma predictions respectively.

The FPGA implementation for the proposed architecture including input and output register files and internal RAMs uses the following FPGA resources; 3034 Function Generators, 1517 CLB Slices, and 342 DFFs, i.e. %3.26 of Function Generators, %3.26 of CLB Slices, and %0.35 of DFFs.

# CHAPTER 3

# HARDWARE ARCHITECTURES FOR H.264 ENTROPY CODER

H.264 intra frame coder (baseline profile) supports coded sequences containing I-slices. I-slices contain intra-coded MBs in which each 16x16 or 4x4 luma block and each 8x8 chroma block is predicted from previously-coded blocks in the same slice. Figure 3.1 shows the syntax of a coded slice. The slice header defines the slice type and the coded picture that the slice 'belongs' to and may contain instructions related to reference picture management. The slice data consists of a series of coded MBs and/or an indication of skipped MBs. Each MB contains several header elements and coded residual data [4].

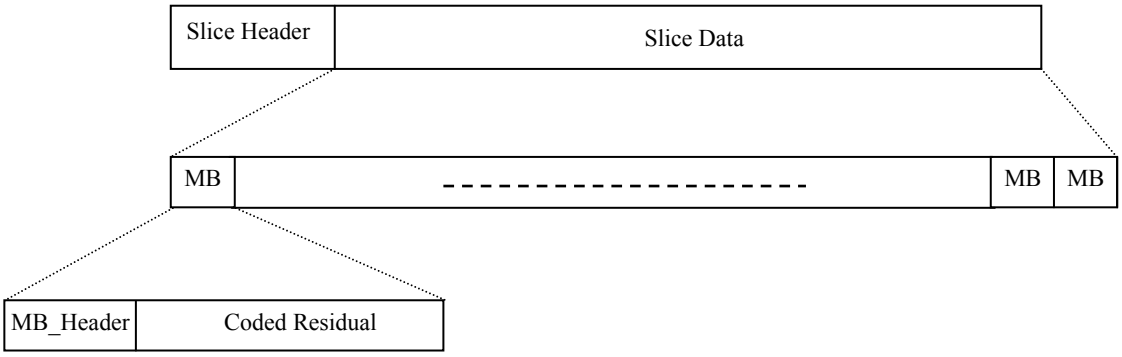



**Figure 3.1** Slice Syntax

Above the slice layer, syntax elements are encoded as fixed- or variable-length binary codes. At the slice layer and below, elements are coded using either variable-length

codes (VLCs) or context-adaptive arithmetic coding (CABAC) depending on the entropy coding mode. For the baseline profile, entropy coding mode is set to 0 and residual block data is coded using a context-adaptive variable length coding (CAVLC) scheme and all other syntax elements are coded using fixed-length or Exponential-Golomb Variable Length Codes. Macroblock syntax elements that should be encoded and transmitted in a baseline profile H.264 intra frame coder are shown in Table 3.1.

**Table 3.1** Macroblock Syntax Elements

| **Parameters** | **Description** |
|---|---|
| MB type | Prediction method for each coded MB |
| Coded block pattern | Indicates which blocks within a MB contain coded coefficients |
| Quantization parameter | Transmitted as a delta value from the previous value of QP |
| Residual data | Coefficient data for each 4x4 or 2x2 block |

## 3.1 H.264 Context-based Adaptive Variable Length Coding (CAVLC)

The video compression efficiency achieved in H.264 standard is not a result of any single feature but rather a combination of a number of encoding tools. As it is shown in the top-level block diagram of an H.264 Encoder in Figure 1.1, one of these tools is the Context Adaptive Variable Length Coding (CAVLC) algorithm used in the baseline profile of H.264 standard [3, 4, 5].

CAVLC algorithm is used to encode transformed and quantized 4x4 residual luminance and chrominance blocks. CAVLC algorithm uses multiple VLC tables for a syntax element. It adapts to the current context by selecting one of the VLC tables for a given syntax element based on the already transmitted syntax elements. This context-adaptivity provides better entropy coding performance in comparison to an entropy coding algorithm using a single VLC table. In addition, CAVLC algorithm improves the entropy coding performance by using coding techniques such as run-level and trailing ones coding

that are designed to take advantage of the characteristics of the 4x4 blocks of transformed and quantized residual data [4, 5, 9]. H.264 CAVLC algorithm achieves better coding results than the entropy coding algorithms used in the previous video compression standards. This coding gain, however, comes with an increase in encoding complexity which makes it an exciting challenge to have a real-time implementation of this algorithm.

### 3.1.1 H.264 CAVLC Algorithm Overview

CAVLC algorithm is used to encode transformed and quantized residual luminance and chrominance blocks in a MB in the order shown in Figure 3.2. Block -1 is formed by the DC coefficients of 4x4 luminance blocks only for the MBs that are coded in 16x16 Intra Mode. Blocks 16 and 17 are formed by the DC coefficients of 4x4 chrominance blocks for all the MBs. All the transformed and quantized 4x4 and 2x2 blocks for a MB are given as inputs to CAVLC algorithm in the order shown in Figure 3.2.

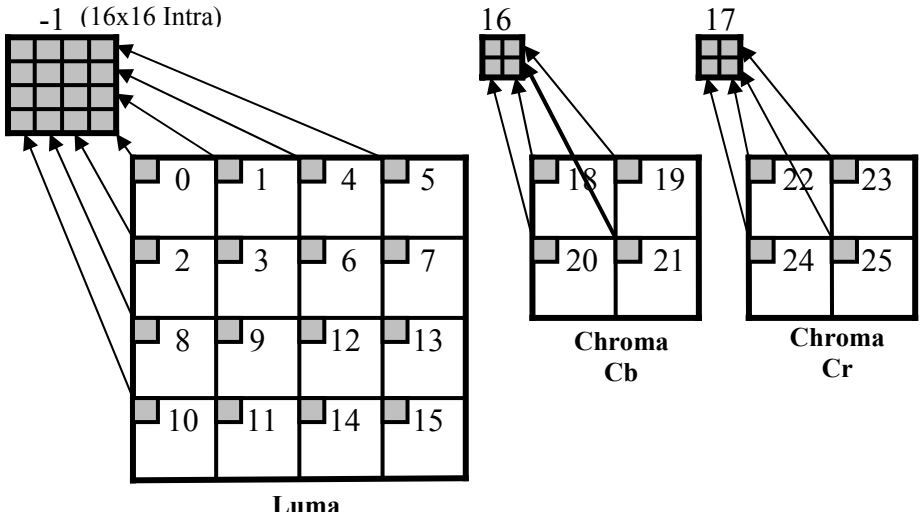



**Figure 3.2** Coding Order of Blocks in a Macroblock

CAVLC algorithm processes each 4x4 block in zig-zag scan order as shown in Figure 3.3 and each 2x2 block in raster scan order. It encodes each block in the following five steps [4, 5, 9].

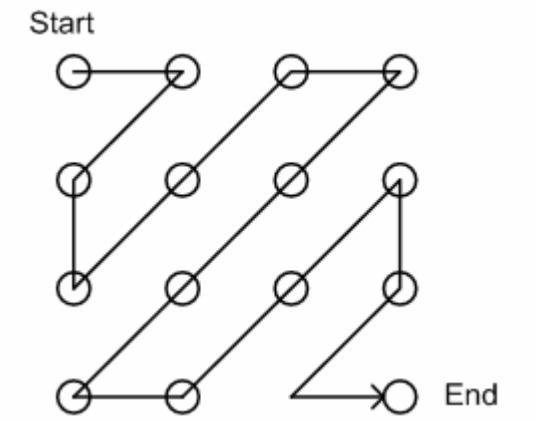



**Figure 3.3** Zig-zag scan for a 4x4 luma block

**Step 1.** It generates coeff_token, the variable length code that encodes both the number of non-zero coefficients (TotalCoeff) and the number of trailing ±1 values (TrailingOnes) in a block. Since the highest non-zero coefficients after the zig-zag scan are often sequences of ±1, CAVLC algorithm encodes the number of high-frequency ±1 coefficients (TrailingOnes) in coeff_token. Since the number of non-zero coefficients in neighbouring blocks is correlated, CAVLC algorithm generates coeff_token for a block context adaptively. It uses one of the four different VLC tables for generating the coeff_token for a block based on the number of non-zero coefficients in the neighboring blocks as follows. It first calculates a parameter nC based on the number of non-zero coefficients in the left-hand and upper previously coded blocks, nA and nB respectively. If upper and left blocks nB and nA are both available, nC = round ((nA + nB) /2). If only the upper is available, nC = nB; if only the left block is available, nC = nA; if neither is available, nC = 0. As a special case, for 2x2 dc chroma blocks, nC is always set to -1. It, then, selects the VLC table that will be used for generating the coeff_token based on the value of nC as shown in Table 3.2.

**Table 3.2** VLC Table for Coeff_Token

| nC | VLC Table for coeff_token |
|---|---|
| 0,1 | Table 1 |
| 2,3 | Table 2 |
| 4,5,6,7 | Table 3 |
| 8 or above | Table 4 |

**Step 2.** It encodes the sign of each TrailingOne with a single bit in reverse order starting with the highest-frequency TrailingOne.

**Step 3.** It encodes the level (sign and magnitude) of each remaining non-zero coefficient in the block in reverse order starting with the highest frequency coefficient and working back towards the DC coefficient. The codeword for a level consists of a prefix and a suffix. Since the magnitude of non-zero coefficients tends to be larger near the DC coefficient and smaller towards the higher frequencies, CAVLC algorithm adapts the suffix length for the level parameter depending on recently-coded level magnitudes. It sets the suffix length for the first level to 0 (unless there are more than 10 nonezero coefficients and less than 3 trailingOnes, in which case initialise to 1). It then increments the current suffix length, if the magnitude of the current level is larger than a predefined threshold for this suffix length. The thresholds are listed in Table 3.3. CAVLC algorithm generates the code length and the codeword for the current level based on its suffix length. When the suffix length for a level is 0, its codeword does not include a suffix. Otherwise, the codeword for the level includes a suffix. The codeword for a level always includes a prefix, but the prefix for a level is generated using different equations in the two cases; when the suffix length for the level is 0 versus when the suffix length for the level is greater than 0 [9].

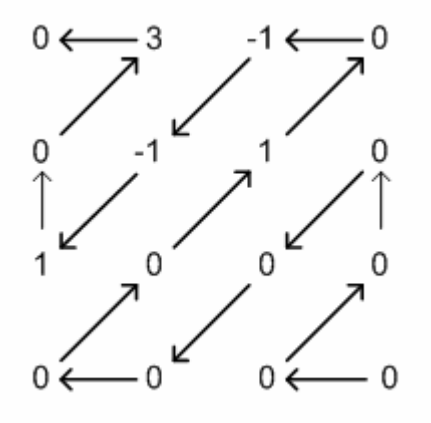
**Table 3.3** Thresholds for determining whether to increment SuffixLength

| Current SuffixLength | Threshold to increment SuffixLength |
|---|---|
| 0 | 0 |
| 1 | 3 |
| 2 | 6 |
| 3 | 12 |
| 4 | 24 |
| 5 | 48 |
| 6 | N/A (highest SuffixLength) |

**Step 4.** It encodes the total number of zeros before the last non-zero coefficient (Total_Zeros) using a VLC table.

**Step 5.** It encodes the number of zeros preceding each non-zero coefficient (Run_Before) in reverse order starting with the highest-frequency coefficient. Since after transformation and quantization, blocks typically contain mostly zeros, CAVLC algorithm uses run-length coding to represent strings of zeros compactly.

Let us take Figure 3.4 as an example of coding a 4x4 block by CAVLC.




(a) Reverse zig-zag scan

| **Element** | **Value** | **Code** |
|---|---|---|
| Coeff_token | TotalCoeffs = 5, TrailingOnes= 3 (use Table 1) | 0000100 |
| TrailingOne sign (4) | Positive | 0 |
| TrailingOne sign (3) | Negative | 1 |
| TrailingOne sign (2) | Negative | 1 |
| Level (1) | +1(use SuffixLength = 0) | 1 |
| Level (0) | +3(use SuffixLength = 1) | 0010 |
| TotalZeros | 3 | 111 |
| RunBefore(4) | ZerosLeft = 3; RunBefore = 1 | 10 |
| RunBefore(3) | ZerosLeft = 2; RunBefore = 0 | 1 |
| RunBefore(2) | ZerosLeft = 2; RunBefore = 0 | 1 |
| RunBefore(1) | ZerosLeft = 2; RunBefore = 1 | 01 |
| RunBefore(0) | ZerosLeft = 1; RunBefore = 1 | No code required |

(b) CAVLC Coding

**Figure 3.4** Example of coding a 4x4 block by CAVLC

### 3.1.2 Proposed Hardware Architecture

The proposed hardware architecture for H.264 CAVLC algorithm is shown in Figure 3.5. The proposed hardware performs context-adaptive variable length coding for a macroblock, in the worst case, in 2880 clock cycles. The worst-case occurs for the macroblocks that have no zero coefficients and trailing ±1 coefficients. Therefore, the proposed hardware can process 30 VGA frames per second at 104 MHz. In the following subsections, the hardware architecture will be explained in detail.

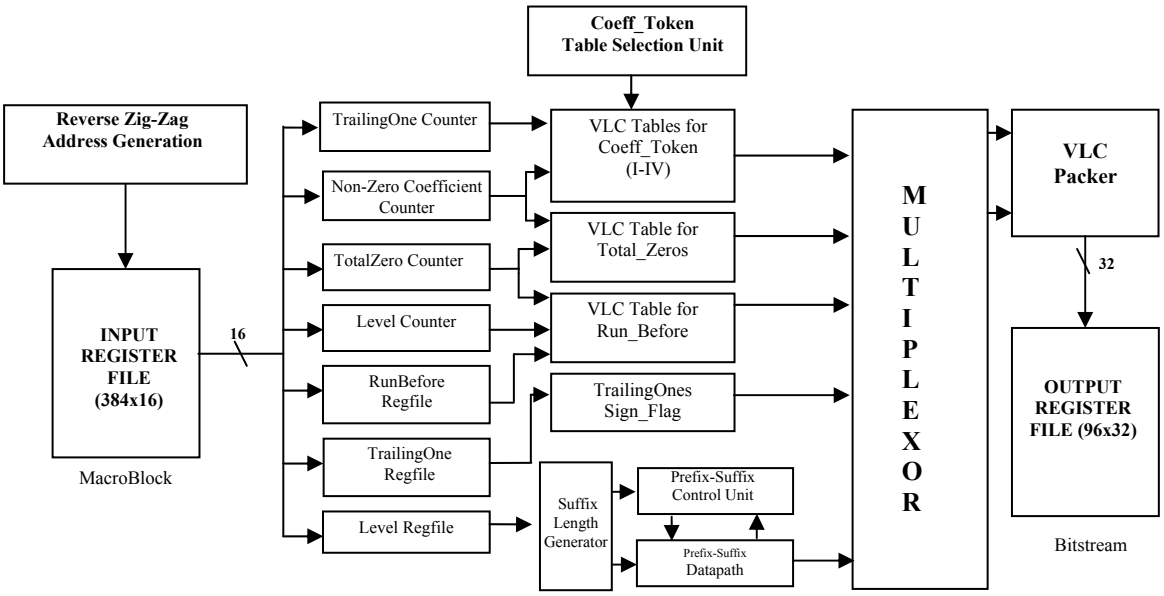



**Figure 3.5** CAVLC Block Diagram

#### 3.1.2.1 VLC Counters and Reverse Zig-zag Ordering

CAVLC hardware contains a number of counters and register files to store the information for a block that will be encoded by variable length codes. Non-Zero Coefficients counter is used to store the number of non-zero coefficients (TotalCoeff).

TrailingOnes counter is used to store the number of trailing ±1 values (TrailingOnes). TotalZeros counter is used to store the total number of zeros before the last non-zero coefficient (TotalZeros). Level counter is used to store the number of non-zero coefficients other than the TrailingOnes. TrailingOnes register file is used to store the sign of each TrailingOne. Level register file is used to store the level (sign and magnitude) of each non-zero coefficient other than the TrailingOnes. RunBefores register file is used to store the number of zeros preceding each non-zero coefficient.

CAVLC hardware begins the encoding for a 4x4 block by reading the coefficients from the input buffer in reverse zig-zag order. In each cycle, it reads one coefficient from the input buffer, and analyzes the coefficient and updates the information stored in the related counter and register file. At the end of this process, the counters and register files mentioned above contain all the information for the current block that will be encoded with variable length codes.

Reverse zig-zag scanning enables us to determine the necessary information for encoding a 4x4 block by reading and analyzing each coefficient only once. This reduces the power consumption by reducing the switching activity on the input buffer address and data signals.

It takes 16 cycles to read the coefficients and store the corresponding information in the counters and register files for each 4x4 luminance block in the macroblocks that are not coded in 16x16 Intra Mode. The same process takes 16 cycles for block -1 and 15 cycles for the other 4x4 luminance blocks in the macroblocks that are coded in 16x16 Intra Mode. Because DC coefficients in 4x4 luminance blocks in these macroblocks are coded in block -1. The same process takes 4 cycles for 2x2 chrominance blocks 16 and 17, and 15 cycles for the 4x4 chrominance blocks in all the macroblocks due to the same reason.

#### 3.1.2.2 CAVLC Hardware for Generating Coeff_Token

CAVLC hardware generates coeff_token, the variable length code that encodes both the number of non-zero coefficients (TotalCoeff) and the number of trailing ±1 values

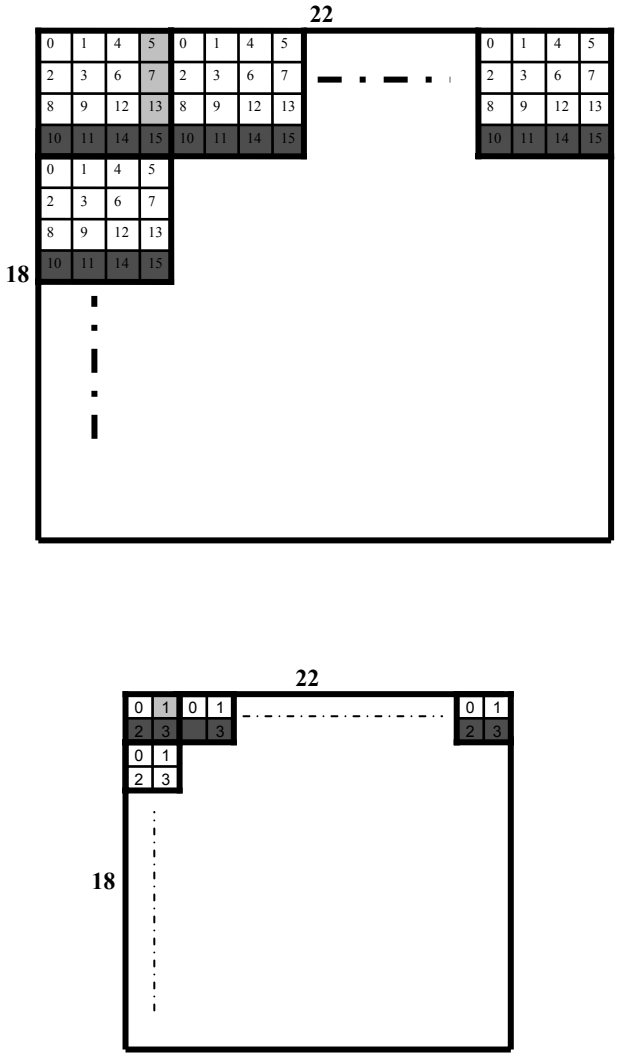
(TrailingOnes) in a block, by a VLC table lookup based on the values of Non-Zero Coefficients and TrailingOnes counters. CAVLC hardware uses one of the four different VLC tables for generating the coeff_token for a block based on the number of non-zero coefficients in the neighboring blocks. Coeff_Token Table Selection Unit (CT_TSU) shown in Figure 3.5 determines the VLC table that will be used for the current block as follows.

CT_TSU first calculates the parameter nC based on the number of non-zero coefficients in the left-hand and upper previously coded blocks, nA and nB respectively. It uses three internal SRAMs and six internal register files to determine the number of non-zero coefficients in the left-hand and upper previously coded neighboring blocks of a 4x4 block in a frame. The organization of the luminance and chrominance components of the MBs in a CIF frame is shown in Figure 3.6. The blocks within a MB are organized and numbered as shown in Figure 3.6.

CT_TSU uses three internal register files to store the number of non-zero coefficients in each 4x4 block in a MB; one for luminance, one for chrominance Cr, and one for chrominance Cb component. After CT_TSU processes a 4x4 block in the current MB, it updates the number of non-zero coefficients entry for this block in the corresponding register file. When it is later processing a new 4x4 block (cblk), if the neighboring 4x4 block (nblk) of cblk is in the same MB as cblk, CT_TSU deter-mines the number of non-zero coefficients in nblk using the corresponding register file entry.

CT_TSU uses three internal register files to store the number of non-zero coefficients in the blocks 5, 7, 13 and 15 of the previously coded MB; one for luminance, one for chrominance Cr, and one for chrominance Cb component. It uses this data to deter-mine the number of non-zero coefficients in the left-hand previously coded neighboring block of the blocks 0, 2, 8 and 10 in the current MB.

CT_TSU uses three internal SRAMs to store the number of non-zero coefficients in the blocks 10, 11, 14, and 15 of the MBs in the previously coded MB row of the frame; one for luminance, one for chrominance Cr, and one for chrominance Cb component. It uses this data to determine the number of non-zero coefficients in the upper previously coded neighboring block of the blocks 0, 1, 4 and 5 in the current MB. We have disabled the SRAMs when they are not accessed in order to reduce power consumption in CT_TSU.




(b)

**Figure 3.6** Macroblocks in a CIF Frame (a) Luma, (b) Chroma Cb and Cr

In addition, instead of using one large external SRAM, we have used three internal SRAMs and six internal register files to store the number of non-zero coefficients in previously coded blocks in order to further reduce power consumption in CT_TSU. This is achieved in two ways. First, instead of accessing one large external SRAM, one internal SRAM and two register files are used for a block. The other internal SRAMs and register files are not accessed saving power. Second, instead of using complex address generation logic for an external SRAM, much simpler address generation logic is used for the internal

SRAMs and register files. The blocks in the same locations of different MBs write to and read from the same SRAM and register file locations. So, if the left-hand and upper previously coded neighboring blocks of a 4x4 block are available, the read addresses for the number of non-zero coefficients in these blocks are generated by a table lookup based on its macroblock and block number. Consequently, CT_TSU calculates the parameter nC for the current block in just one cycle. It then selects the VLC table for the coeff_token based on the value of nC as shown in Table 3.2. It then generates coeff_token for the current block by a table lookup to the selected VLC table based on the values of Non-Zero Coefficients and TrailingOnes counters.

#### 3.1.2.3 CAVLC Hardware for Encoding Level

The codeword for a level consists of a prefix and a suffix. Suffix length generator shown in Figure 3.5 determines the suffix length for the current level based on the magnitude of previously coded levels. The prefix-suffix datapath and control unit shown in Figure 3.7 generate the code length and the codeword for the current level based on the suffix length provided by the suffix length generator. The prefix for a level is generated using different equations when the suffix length for the level is 0 versus when the suffix length for the level is greater than 0.

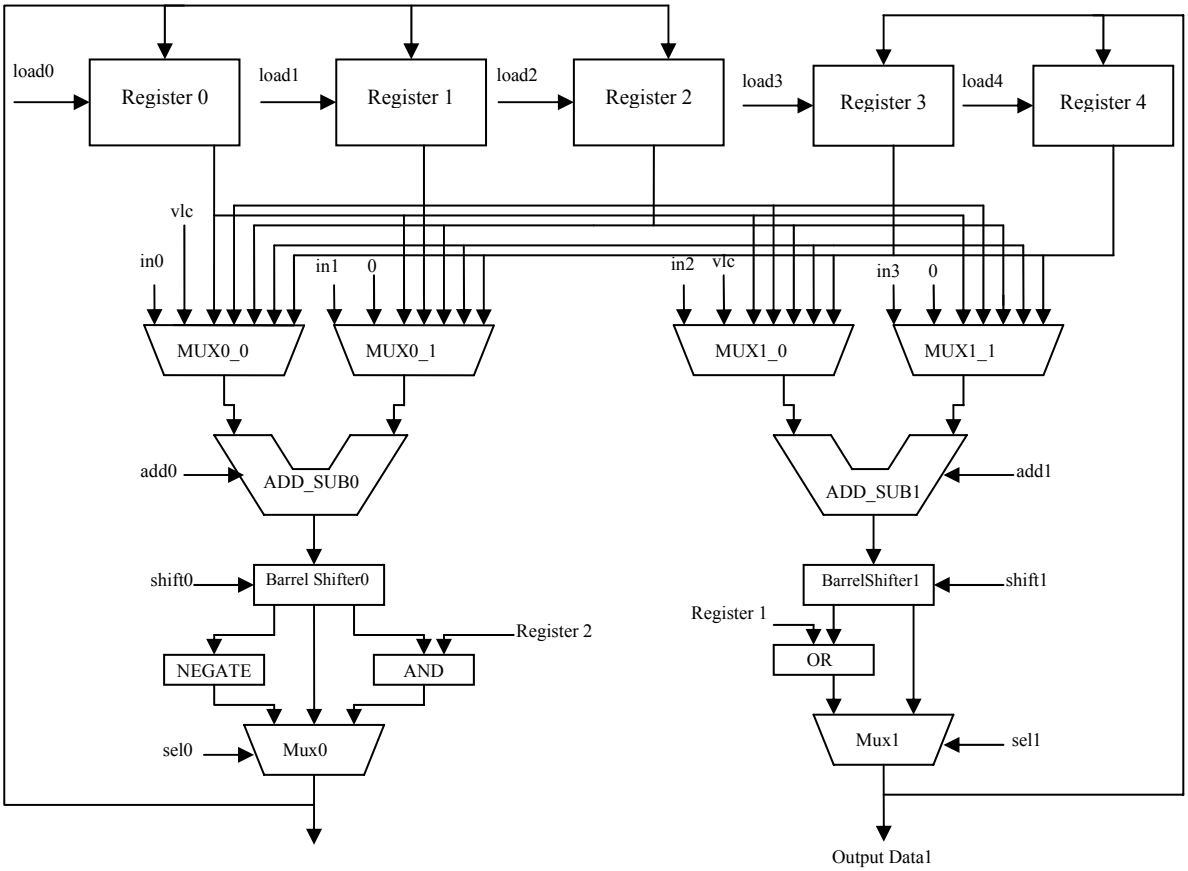



**Figure 3.7** Datapath for Coding Level Prefix and Level Suffix

The equations used in both cases are given in the Joint Model (JM) Reference Software Version 8.2 [9]. We have proposed the dual purpose prefix-suffix datapath shown in Figure 3.7 to implement these equations. In each case, the prefix-suffix control unit sends the appropriate control signals to the datapath. In the worst case, our suffix length generator, prefix-suffix datapath and control unit take 6 cycles to generate the code length and codeword for a level.

### 3.1.2.4 VLC Packer

Since CAVLC generates variable length codewords, consecutive codewords should be packed into fixed size words before being written to output register file. The datapath shown in Figure 3.8 is used to pack the variable length codewords into 32-bit words [10]. When a new codeword is sent to the VLC packer, the barrel shifter places this codeword next to the end of the bitstream stored in the 32-bit lower register. If the length of the resulting bitstream is larger than 31, the carry-out bit of the adder in the datapath is set to one. This indicates that a 32-bit bitstream is packed into the upper register. Thus, VLC packer outputs the content of the upper register, and it moves the content of the lower register into the upper register. This process is repeated for each codeword.

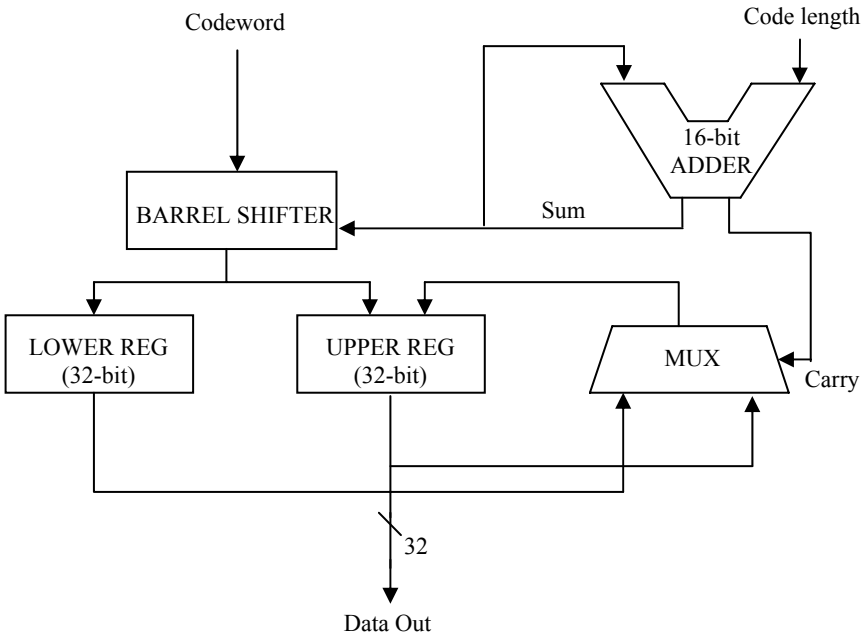



**Figure 3.8** VLC Packer Datapath

### 3.1.3 Implementation Results

A high performance and low power hardware architecture for real-time implementation of Context Adaptive Variable Length Coding algorithm used in H.264 / MPEG4 Part 10 video coding standard is presented. This hardware is designed to be used as part of a complete efficient H.264 intra frame coder hardware design. The proposed architecture is implemented in Verilog HDL. The implementation is verified with RTL simulations using Mentor Graphics ModelSim SE. The Verilog RTL is then synthesized to a 2V8000ff1157 Xilinx Virtex II FPGA with speed grade 5 using Mentor Graphics Leonardo Spectrum. The resulting netlist is placed and routed to the same FPGA using Xilinx ISE Series 7.1i.

The FPGA implementation including input and output register files as well is placed and routed at 76 MHz under worst-case PVT conditions. Since, in the worst-case, it takes 2880 clock cycles to process a MB, the FPGA implementation can code 22 VGA frames (640x480) per second. The FPGA implementation is verified to work in a Xilinx Virtex II FPGA on an Arm Versatile Platform development board.

The FPGA implementation including input and output register files as well used the following FPGA resources; 3946 Function Genera-tors, 1973 CLB Slices, 719 Dffs /Latches, and 6 Block RAMs, i.e. 4.23% of Function Generators, 4.23% of CLB Slices, 0.75% of Dffs /Latches, and 3.57% of Block RAMs.

The FPGA implementation excluding input and output register files used the following FPGA resources; 3849 Function Generators, 1925 CLB Slices, 719 Dffs /Latches, and 6 Block RAMs, i.e. 4.13% of Function Generators, 4.13% of CLB Slices, 0.75% of Dffs /Latches, and 3.57% of Block RAMs.

The Verilog RTL is also synthesized to Virtual Silicon UMC 0.18µ standard-cell library using Synopsys Design Compiler. The netlist excluding input and output register files has an area of 32K gates. The netlist including input and output register files has an area of 96K gates and it is verified to work at 233 MHz under worst-case PVT conditions with post synthesis simulations. This 0.18µ ASIC implementation can code 67 VGA frames (640x480) per second.

A hardware architecture for H.264 CAVLC algorithm similar to our design is presented in [7]. However, that architecture doesn't have the low-power techniques we have used in our design.

## 3.2 H.264 Exponential-Golomb Variable Length Entropy Coding

Some of the sequence, picture, slice and macroblock syntax elements are coded using exponential-golomb entropy coding in a baseline profile H.264 intra frame coder.

### 3.2.1 Exponential-Golomb Variable Length Entropy Coding Algorithm Overview

Exponential-Golomb Variable Length Entropy Coding algorithm is designed to produce short codewords for frequently occurring parameters and longer codewords for less common parameters.

#### 3.2.1.1 Exponential-Golomb Codes

Exponential Golomb (Exp-Golomb) codes are variable length codes with a regular construction [4]. It is clear from examining the first few codewords in Table 3.4 that they are constructed in a logical way:

[M zeros][1][INFO]

INFO is an *M*-bit field carrying information. The first codeword has no leading zero or trailing INFO. Codewords 1 and 2 have a single-bit INFO field, codewords 3–6 have a two-bit INFO field and so on. The length of each Exp-Golomb codeword is (2*M* + 1) bits and each codeword can be constructed by the encoder based on its index *code_num*:

$$M = \text{floor}(\log2[\text{code\_num} + 1])$$
$$\text{INFO} = \text{code\_num} + 1 - 2^{M}$$

**Table 3.4** Exp-Golomb codewords

| **code_num** | **codeword** |
|---|---|
| 0 | 1 |
| 1 | 010 |
| 2 | 011 |
| 3 | 00100 |
| 4 | 00101 |
| 5 | 00110 |
| 6 | 00111 |
| 7 | 0001000 |
| 8 | 0001001 |
| … | … |

A codeword can be decoded as follows:

1. Read in M leading zeros followed by 1.
2. Read M-bit INFO field.
3. code_num = $2^{M}$+ INFO – 1

(For codeword 0, INFO and M are zero.)

A parameter ***k*** to be encoded is mapped to code_num in one of the following ways:

**Mapping Type Description**

**ue(v):** Unsigned direct mapping, used for macroblock type, reference frame index and others. code_num = k.

**se(v):** Signed mapping, used for motion vector difference, delta QP and others. k is mapped to code_num as follows:

$$code\_num = 2|k| \ (k \leq 0)$$
$$code\_num = 2|k|-1 \ (k > 0)$$

**me(v):** Mapped symbols, parameter k is mapped to code_num according to a table specified in the H.264 standard [5].

#### 3.2.1.2 Sequence Syntax Elements

The sequence syntax elements are coded using either exp-golomb variable length entropy coding or fixed-length binary codes (u(n)) [5]. The sequence syntax elements and their corresponding mapping types for a baseline H.264 intra frame coder are shown in Table 3.5.

**u(n):** unsigned integer using n bits (fixed-length binary code mapping).

**Table 3.5** Sequence Syntax Elements

| **Syntax Element** | **Mapping Type** |
|---|---|
| profile_idc | u(8) |
| constrained_set0_flag | u(1) |
| constrained_set1_flag | u(1) |
| constrained_set2_flag | u(1) |
| reserved_zero | u(5) |
| level_idc | u(8) |
| seq_parameter_set_id | ue(v) |
| log2_max_frame_num_minus4 | ue(v) |
| pic_order_cnt_type | ue(v) |
| log2_max_pic_order_cnt_lsb_minus4 | ue(v) |
| num_ref_frames | ue(v) |
| gaps_in_frame_num_value_allowed_flag | u(1) |
| pic_width_in_mbs_minus1 | ue(v) |
| pic_height_in_map_units_minus1 | ue(v) |
| frame_mbs_only_flag | u(1) |
| direct_8x8_inference_flag | u(1) |
| frame_cropping_flag | u(1) |
| vui_parameters_present_flag | u(1) |

### 3.2.1.2 Picture Syntax Elements

The picture syntax elements are coded using either exp-golomb variable length entropy coding or fixed-length binary codes [5]. The picture syntax elements and their corresponding mapping types for a baseline H.264 intra frame coder are shown in Table 3.6.

**Table 3.6** Picture Syntax Elements

| **Syntax Element** | **Mapping Type** |
|---|---|
| pic_parameter_set_id | ue(v) |
| seq_parameter_set_id | ue(v) |
| entropy_coding_mode_flag | u(1) |
| pic_order_present_flag | u(1) |
| num_slice_groups_minus1 | ue(v) |
| num_ref_idx_l0_active_minus1 | ue(v) |
| num_ref_idx_l1_active_minus1 | ue(v) |
| weighted_pred_flag | u(1) |
| weighted_bipred_idc | u(2) |
| pic_init_qp_minus26 | se(v) |
| pic_init_qs_minus26 | se(v) |
| chroma_qp_index_offset | se(v) |
| deblocking_filter_control_present_flag | u(1) |
| constrained_intra_pred_flag | u(1) |
| redundant_pic_cnt_present_flag | u(1) |

### 3.2.1.3 Slice Syntax Elements

The slice syntax elements are coded using either exp-golomb variable length entropy coding or fixed-length binary codes. The following slice syntax elements are coded for a baseline H.264 intra frame coder. The corresponding mapping types are also indicated below [5].

**first_mb_in_slice - ue(v) :** specifies the address of the first Macroblock in the slice.

**slice_type - ue(v) :** specifies the coding type of the slice.

**pic_parameter_set_id - ue(v) :** specifies the picture parameter set in use.

**frame_num - ue(v) :** is used as a unique identifier for each short-term reference frame.

**idr_pic_id - ue(v) :** identifies an IDR picture.

**pic_order_cnt_lsb - u(v) :** specifies the picture order count

**no_output_of_prior_pics_flag - u(1) :** specifies how the previously-decoded pictures in the decoded picture buffer are treated after decoding of an IDR picture.

**long_term_reference_flag - u(1) :** equals to 0 specifies that the IDR picture is marked as "used for short-term reference", and equals to 1 specifies that the current IDR picture is marked "used for long-term reference".

**slice_qp_delta - se(v) :** specifies the initial value of QPY to be used for all the macroblocks in the slice until modified by the value of mb_qp_delta in the macroblock layer. The initial QPY quantization parameter for the slice is computed as:

$$SliceQPY = 26 + pic\_init\_qp\_minus26 + slice\_qp\_delta$$

#### 3.2.1.4 Macroblock Syntax Elements

The macroblock syntax elements are coded using either exp-golomb variable length entropy coding or fixed-length binary codes. The macroblock syntax elements and their corresponding mapping types for a baseline H.264 intra frame coder are shown in Table 3.7.

**Table 3.7** Macroblock Syntax Elements

| Syntax Element | Mapping Type |
|---|---|
| mb_type | ue(v) |
| mb_pred | u(n), ue(v) |
| Coded_Block_Pattern | me(v) |
| mb_qp_delta | se(v) |

**mb_type - ue(v) :**

Macroblock types for I-slices are listed in Table 3.9 [5].
I_4x4: MB is coded using Intra_4x4 prediction.

I_16x16_0_0_0, I_16x16_1_0_0, I_16x16_2_0_0, I_16x16_3_0_0, I_16x16_0_1_0, I_16x16_1_1_0, I_16x16_2_1_0, I_16x16_3_1_0, I_16x16_0_2_0, I_16x16_1_2_0, I_16x16_2_2_0, I_16x16_3_2_0, I_16x16_0_0_1, I_16x16_1_0_1, I_16x16_2_0_1, I_16x16_3_0_1, I_16x16_0_1_1, I_16x16_1_1_1, I_16x16_2_1_1, I_16x16_3_1_1, I_16x16_0_2_1, I_16x16_1_2_1, I_16x16_2_2_1, I_16x16_3_2_1: MB is coded using Intra_16x16 prediction.

*pred_mode* specifies the Intra_16x16 prediction mode.

CPBL (*CodedBlockPatternLuma*) specifies whether for the luma component non-zero AC transform coefficient levels are present. *CodedBlockPatternLuma* equal to 0 specifies that there are no AC transform coefficient levels in the luma component of the MB. *CodedBlockPatternLuma* equal to 15 specifies that there is at least one AC transform coefficient level in the luma component of the MB, requiring scanning of AC transform coefficient levels for all 4x4 blocks in the MB.

CBPC (*CodedBlockPatternChroma*) is explained in Table 3.8.

**Table 3.8** Specification of *CodedBlockPatternChroma*

| **CBPC** | **Description** |
|---|---|
| 0 | All chroma transform coefficient levels are equal to 0. |
| 1 | One or more chroma DC transform coefficient levels are non-zero.<br>All chroma AC transform coefficient levels are equal to 0. |
| 2 | Zero or more chroma DC transform coefficient levels are non-zero.<br>One or more chroma AC transform coefficient levels are non-zero. |

**Table 3.9** Macroblock types for I slices

| mb_type | name of mb_type | mb_pred | pred_mode | CPBC | CBPL |
|---|---|---|---|---|---|
| 0 | I_4x4 | Intra_4x4 | na | na | na |
| 1 | I_16x16_0_0_0 | Intra_16x16 | 0 | 0 | 0 |
| 2 | I_16x16_1_0_0 | Intra_16x16 | 1 | 0 | 0 |
| 3 | I_16x16_2_0_0 | Intra_16x16 | 2 | 0 | 0 |
| 4 | I_16x16_3_0_0 | Intra_16x16 | 3 | 0 | 0 |
| 5 | I_16x16_0_1_0 | Intra_16x16 | 0 | 1 | 0 |
| 6 | I_16x16_1_1_0 | Intra_16x16 | 1 | 1 | 0 |
| 7 | I_16x16_2_1_0 | Intra_16x16 | 2 | 1 | 0 |
| 8 | I_16x16_3_1_0 | Intra_16x16 | 3 | 1 | 0 |
| 9 | I_16x16_0_2_0 | Intra_16x16 | 0 | 2 | 0 |
| 10 | I_16x16_1_2_0 | Intra_16x16 | 1 | 2 | 0 |
| 11 | I_16x16_2_2_0 | Intra_16x16 | 2 | 2 | 0 |
| 12 | I_16x16_3_2_0 | Intra_16x16 | 3 | 2 | 0 |
| 13 | I_16x16_0_0_1 | Intra_16x16 | 0 | 0 | 15 |
| 14 | I_16x16_1_0_1 | Intra_16x16 | 1 | 0 | 15 |
| 15 | I_16x16_2_0_1 | Intra_16x16 | 2 | 0 | 15 |
| 16 | I_16x16_3_0_1 | Intra_16x16 | 3 | 0 | 15 |
| 17 | I_16x16_0_1_1 | Intra_16x16 | 0 | 1 | 15 |
| 18 | I_16x16_1_1_1 | Intra_16x16 | 1 | 1 | 15 |
| 19 | I_16x16_2_1_1 | Intra_16x16 | 2 | 1 | 15 |
| 20 | I_16x16_3_1_1 | Intra_16x16 | 3 | 1 | 15 |
| 21 | I_16x16_0_2_1 | Intra_16x16 | 0 | 2 | 15 |
| 22 | I_16x16_1_2_1 | Intra_16x16 | 1 | 2 | 15 |
| 23 | I_16x16_2_2_1 | Intra_16x16 | 2 | 2 | 15 |
| 24 | I_16x16_3_2_1 | Intra_16x16 | 3 | 2 | 15 |

**mb_pred - u(n) :**

The syntax element *mb_pred* specifies 4x4 luma prediction modes for a MB. The choice of 4x4 luma prediction mode for each 4x4 luma block must be signaled to the decoder and this could potentially require a large number of bits. However, 4x4 luma prediction modes for neighboring 4x4 luma blocks are often correlated. For example, let A, B and E be the left, upper and current 4x4 luma blocks respectively as shown in Figure 3.9. If previously-encoded 4x4 luma blocks A and B are predicted using mode 1, it is probable that the best mode for block E (current block) is also mode 1. To take advantage of this correlation, predictive coding is used to signal 4x4 luma prediction modes [4].

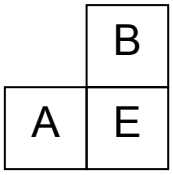



**Figure 3.9** Current and neighboring 4x4 luma blocks

For the current block E, the encoder and decoder calculate the most probable 4x4 prediction mode as the minimum of the prediction modes of A and B. If either of these neighboring 4x4 luma blocks is not available (outside the current slice or not coded in 4x4 luma prediction mode), the corresponding value for the 4x4 luma prediction mode is set to 2.

The encoder sends a flag for each 4x4 luma block, *prev_intra4x4_pred_mode*. If the flag is '1', the most probable 4x4 luma prediction mode is used. If the flag is '0', another parameter *rem_intra4x4_pred_mode* is sent to indicate the mode. If the current 4x4 luma prediction mode is smaller than the current most probable 4x4 luma prediction mode then *rem_intra4x4_pred_mode* is set to current 4x4 luma prediction mode, otherwise *rem_intra4x4_pred _mode* is set to current 4x4 luma prediction mode minus 1. In this way, only eight values of *rem_intra4x4_pred_mode* are required (0 to 7) to signal the current 4x4 luma prediction mode (0 to 8).

prev_intra4x4_pred_mode_flag - u(1)
rem_intra4x4_pred_mode - u(3)

If *prev_intra4x4_pred_mode_flag* = 1, only this parameter is coded as *mb_pred* (u(1)).

If *prev_intra4x4_pred_mode_flag* = 0, in addition to *prev_intra4x4_pred_mode_flag,* another parameter *rem_intra4x4_pred_mode* is coded as *mb_pred*(u(4)).

As an example for predictive coding of 4x4 luma prediction mode for a 4x4 luma block, assume that blocks A and B are predicted using 4x4 luma prediction modes 4 and 3 respectively and the 4x4 luma prediction mode for block E is not 3. Therefore, the most probable 4x4 luma prediction mode for block E is 3 and *prev_intra4x4_pred_mode_flag* is set to '0'. Depending on the value of current 4x4 luma prediction mode for block E, the *rem_intra4x4_pred_modes* will be coded as shown in Table 3.10.

**Table 3.10** Predictive coding of a 4x4 luma prediction mode
(most probable 4x4 luma prediction mode = 3 & *prev_intra4x4_pred_mode_flag* = 0 )

| **4x4 luma prediction mode** | **rem_intra4x4_pred_mode** |
|---|---|
| 0 | 0 |
| 1 | 1 |
| 2 | 2 |
| 4 | 3 |
| 5 | 4 |
| 6 | 5 |
| 7 | 6 |
| 8 | 7 |

Predictive coding is not used for signaling 16x16 luma prediction modes.

**mb_pred - ue(v) :**

The syntax element *mb_pred* specifies 8x8 chroma prediction modes for a MB. Predictive coding is not used for signaling 8x8 chroma prediction modes. Exp-Golomb codes for 8x8 chroma prediction modes are given in Table 3.11.

**Table 3.11** Exp-Golomb codes for 8x8 chroma prediction modes

| Chroma Prediction Mode | mb_pred | Exp-Golomb Code |
|---|---|---|
| 0 | 0 | 1 |
| 1 | 1 | 010 |
| 2 | 2 | 011 |
| 3 | 3 | 00100 |

**Coded_Block_Pattern – me(v) :**

*Coded_Block_Pattern* specifies which of the six 8x8 blocks - luma and chroma - contain non-zero transform coefficient levels. If the MB is coded using 4x4 luma prediction modes, *coded_block_pattern* should be present in the MB header. The assignment of *coded_block_pattern* to code_num is given in the H.264 Standard [5].

*Coded_Block_Pattern* has a 6-bit binary representation. The four least significant bits are for luma components of a MB and 2 most significant bits are for chroma components of a MB and equal to *CodedBlockPatternChroma* which is explained in detail before. The least significant bit of *Coded_Block_Pattern* indicates whether the $0^{th}$ 8x8 luma block of a MB is coded or not and the following bits of *Coded_Block_Pattern* indicate whether the $1^{st}$, $2^{nd}$ and $3^{rd}$ 8x8 luma blocks of a MB are coded or not. Several examples of calculating *Coded_Block_Pattern* for a MB are shown in Table 3.12.

**Table 3.12** Examples of calculating *Coded_Block_Pattern* for a MB

| **Coded Block Pattern** | **Coded 8x8 Blocks** |
|---|---|
| 47 (101111) | Four 8x8 Luma Blocks + Chroma Dc + Chroma Ac |
| 38 (100110) | $1^{st}$ and $2^{nd}$ 8x8 luma blocks + Chroma Dc + Chroma Ac |
| 15 (001111) | Four 8x8 Luma Blocks |
| 31 (011111) | Four 8x8 Luma Blocks + Chroma Dc |
| 2 (100000) | Chroma Dc + Chroma Ac |

**mb_qp_delta - ue(v) :**

*Slice_qp_delta* specifies the initial value of QPY to be used for all the macroblocks in the slice until modified by the value of *mb_qp_delta* in the macroblock layer. *Mb_qp_delta* is transmitted as a delta value from the previous value of QP and it can change the value of QPY in the macroblock layer. The value of *slice_qp_delta* shall be limited such that QPY is in the range of 0 to 51, inclusive.

### 3.2.2 Proposed Hardware Architecture

The block diagram of the proposed hardware architecture for header generation is shown in Figure 3.10. The proposed hardware generates the headers for sequence, picture, slice, and macroblock layer based on prediction and quantized transform coefficients.

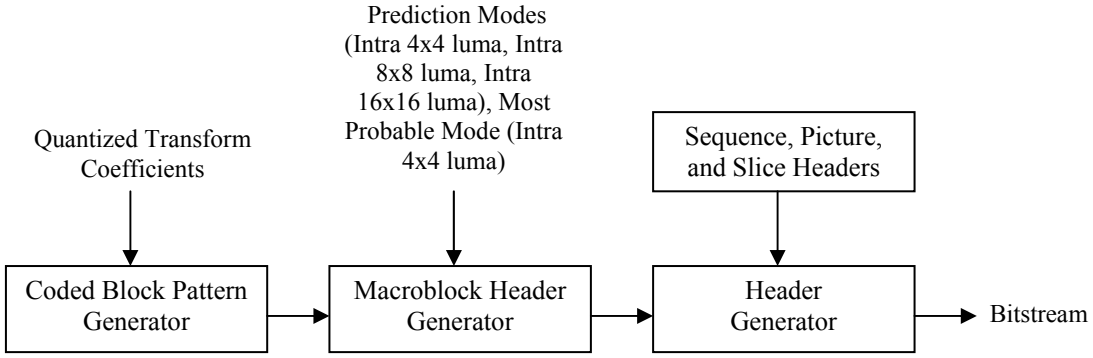



**Figure 3.10** Block Diagram of Header Generation Hardware

Coded Block Pattern Generator module shown in Figure 3.10 generates the following MB syntax elements for each MB: *CodedBlockPatternLuma*, *CodedBlockPatternChroma*, and *Coded_Block_Pattern*. The inputs to the Coded Block Pattern Generator are quantized transform coefficients. Based on the information from the Coded Block Pattern Generator and prediction modes for luma and chroma components of a MB, Macroblock Header

Generator module in Figure 3.10 calculates MB syntax elements and generates corresponding entropy codes for a MB using Exp-Golomb or fixed-length binary codes (MB header). Finally, Header Generator module concatenates sequence, picture and slice headers that are stored in a register with the MB header of the first MB in the slice and writes them to bitstream buffer. For the remaining MBs in the slice, Header Generator module writes their MB header directly to the bitstream buffer.

### 3.2.3 Implementation Results

A hardware architecture for real-time implementation of header generation algorithm used in H.264 / MPEG4 Part 10 video coding standard is presented. This hardware is designed to be used as part of a complete efficient H.264 intra frame coder hardware design. The proposed architecture is implemented in Verilog HDL. The implementation is verified with RTL simulations using Mentor Graphics ModelSim SE. The Verilog RTL is then synthesized to a 2V8000ff1157 Xilinx Virtex II FPGA with speed grade 5 using Mentor Graphics Leonardo Spectrum. The resulting netlist is placed and routed to the same FPGA using Xilinx ISE Series 7.1i.

The FPGA implementation is placed and routed at 76 MHz under worst-case PVT conditions. The FPGA implementation is verified to work in a Xilinx Virtex II FPGA on an Arm Versatile Platform development board.

The FPGA implementation used the following FPGA resources; 1812 Function Genera-tors, 906 CLB Slices, 434 Dffs /Latches, i.e. 1.94% of Function Generators, 1.94% of CLB Slices, and 0.45% of Dffs /Latches.

# CHAPTER 4

# TOP LEVEL H.264 INTRA FRAME CODER HARDWARE

The top-level block diagram of the proposed H.264 intra frame coder is shown in Figure 4.1.

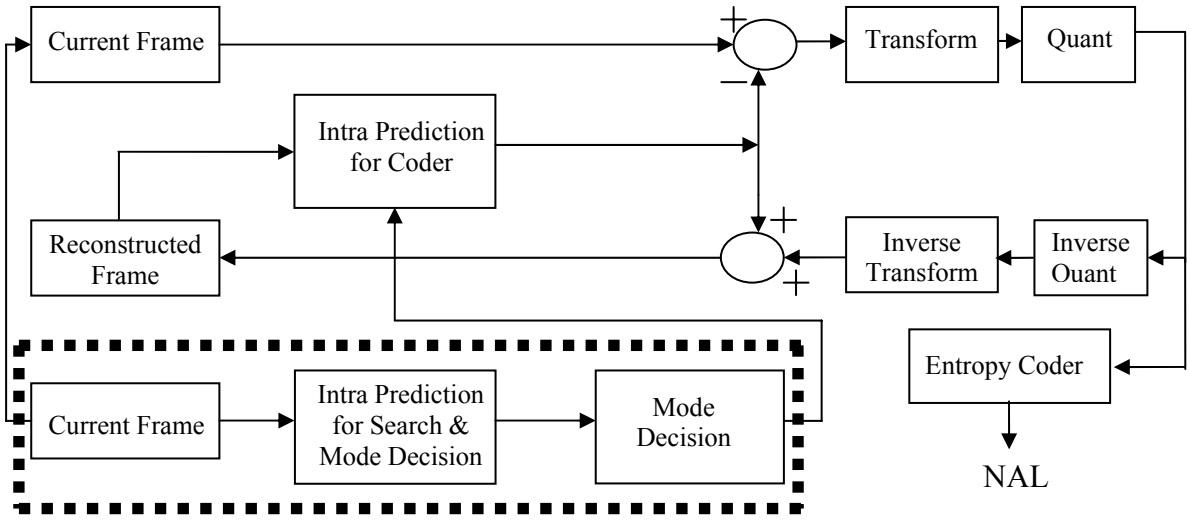



**Figure 4.1** H.264 Intra Frame Coder Block Diagram

## 4.1 Proposed Hardware Architecture

In the previous video coding standards, prediction part and coding part (DCT/Q/IQ/IDCT/VLC) can be clearly separated. After prediction part finishes processing a MB, coding part starts coding this MB and prediction part starts processing the next MB.

However, because of the intra prediction algorithm used in H.264 standard, after prediction of a MB, the next MB can not be predicted before corresponding reconstructed pixels at the output of the Transform/Quant/Inverse Quant/Inverse Transform are produced. The situation is even worse for 4x4 luma prediction modes. Prediction and mode decision of a 4x4 block cannot be performed until the previous 4x4 block is reconstructed. Therefore, the prediction part must wait the coding part which makes the MB pipelining impossible and the design of a real-time intra frame coder hardware very costly.

We, however, divided our proposed H.264 intra frame coder hardware into two main parts; the search & mode decision part and coder part. As shown in Figure 4.2, the search & mode decision hardware and the coder hardware work in a pipelined manner. After the first MB of the input frame is loaded to the input register file, search & mode decision hardware starts to work on determining the best mode for coding this MB. After search & mode decision hardware determines the best mode for the first MB, coder hardware starts to code the first MB using the selected best mode and search & mode decision hardware starts to work on the second MB. The entire frame is processed MB by MB in this order.

This is achieved by performing intra prediction in the search & mode decision hardware using the pixels in the current frame rather than the pixels in the reconstructed frame. However, intra prediction in the coder hardware is performed using the pixels in the reconstructed frame in order to be compliant with H.264 standard. This makes the MB pipelining and therefore the implementation of a low-cost H.264 intra frame coder hardware possible at the expense of a small PSNR loss in the video quality.

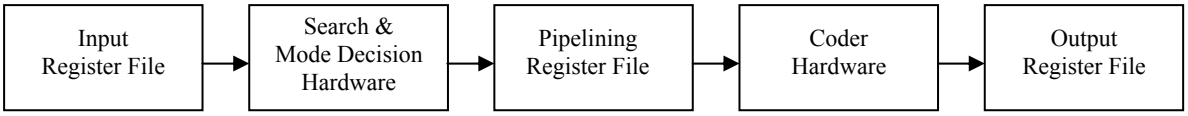



**Figure 4.2** H.264 Intra Frame Coder Block Diagram

### 4.1.1 Search & Mode Decision Hardware

The proposed search & mode decision hardware is shown in Figure 4.3. The proposed hardware includes Intra Prediction, Residue, Hadamard Transform and Mode Decision modules.

In the proposed hardware, there are two parts operating in parallel in order to complete the search & mode decision process faster. The upper part is used for finding the best 16x16 luma prediction mode for the luma component of a MB and the best 8x8 chroma prediction mode for the chroma components of a MB. The lower part is used for finding the best 4x4 luma prediction mode for each 4x4 block in the luma component of a MB.

Top level scheduling for the upper part of the search & mode decision hardware for 16x16 luma predictions is shown in Figure 4.4. First, the neighboring buffers in the intra prediction hardware are loaded with the corresponding neighboring pixels from the current MB register. Then, the intra prediction hardware generates the pixel predictions for the luma component of the current MB using the first available 16x16 luma mode and writes the predicted pixels to the prediction buffer. The Residue hardware, then, calculates the difference between the corresponding luma pixels in the current MB and the predicted MB. As the residue data associated with the first pixel position in a MB is calculated, Hadamard Transform module starts to calculate the Sum of Absolute Transformed Difference (SATD) for that mode using the residue data. So, Residue and Hadamard Transform modules are overlapped.

The overlapping between Residue and Hadamard Transform modules requires using two register files between them. While Residue module is calculating residue data for the next 4x4 luma block and writing the residue data to one of the register files, Hadamard Transform is calculating the SATD for the current 4x4 block based on the current residue data stored in the other register file.

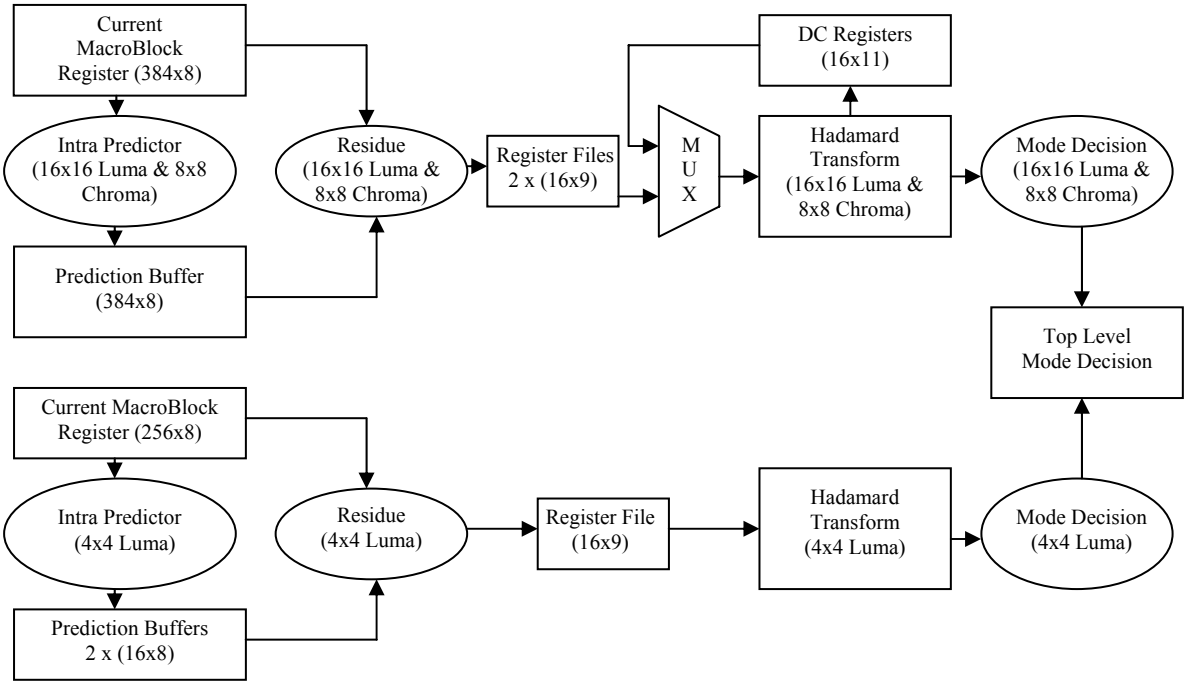



**Figure 4.3** Block Diagram of Search & Mode Decision Hardware

Hadamard transform for SATD calculations of 16x16 luma prediction modes requires storing DC coefficients in a register, because Hadamard transform has to be applied to these coefficients again. The multiplexer before the Hadamard Transform module in Figure 4.3 selects between DC coefficients and coefficients from the residue block.

After Hadamard Transform module finishes calculating SATD for an available 16x16 luma prediction mode of a MB, it decides whether it is the mode with lowest cost or not. After each available 16x16 luma prediction mode for a MB is searched, the prediction mode with the lowest cost and its cost information are sent to the Top Level Mode Decision hardware by the upper part of the search & mode decision hardware.

When the upper part of the search & mode decision hardware finishes with available 16x16 luma modes of a MB for luma samples, it starts to work with 8x8 chroma modes of the same MB for chroma samples. Top level scheduling for chroma samples is similar to that of luma samples.

The latencies of the modules in the upper part of the search & mode decision hardware are given in Table 4.1.

**Table 4.1** Latencies of the Modules in the Upper Part of the Search & Mode Decision Hardware

| Module | Latency |
|---|---|
| Neighbor Loader | 256 clock cycles |
| Hadamard Transform | 288 clock cycles |
| Residue Module | 256 clock cycles |
| Intra Prediction – Mode0 | 257 clock cycles |
| Intra Prediction – Mode1 | 257 clock cycles |
| Intra Prediction – Mode2 | 273 clock cycles |
| Intra Prediction – Mode3 | 340 clock cycles |

In the worst case, when all 16x16 prediction modes are available, intra search for a MB takes 256*4 (Neighbor Loader) + 1127 (Intra Prediction) + 288*4 (Hadamard Transform) + 1*4 (Top Level Mode Decision) = 3307 clock cycles.

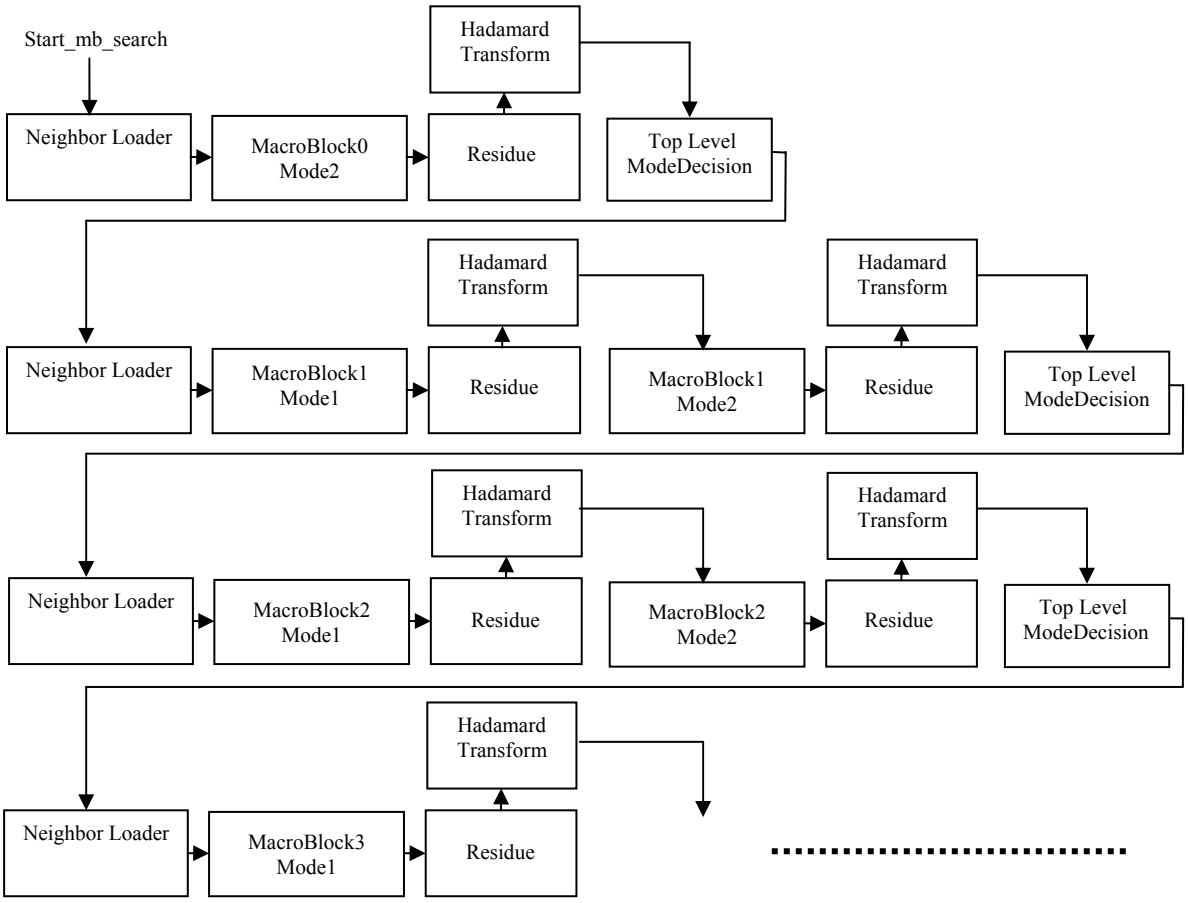



**Figure 4.4** Schedule for 16x16 Luma Prediction Modes

The lower part of the search & mode decision hardware is designed for finding the best 4x4 luma prediction mode for each 4x4 luma block in a MB. Top level scheduling for the lower part of the search and mode decision hardware is shown in Figure 4.6. Initially, we adapted the top level scheduling shown in Figure 4.5. However, this initial schedule could not achieve 30 fps CIF coding performance. We, therefore, improved the top level scheduling for the lower part of the search & mode decision hardware using pipelining.

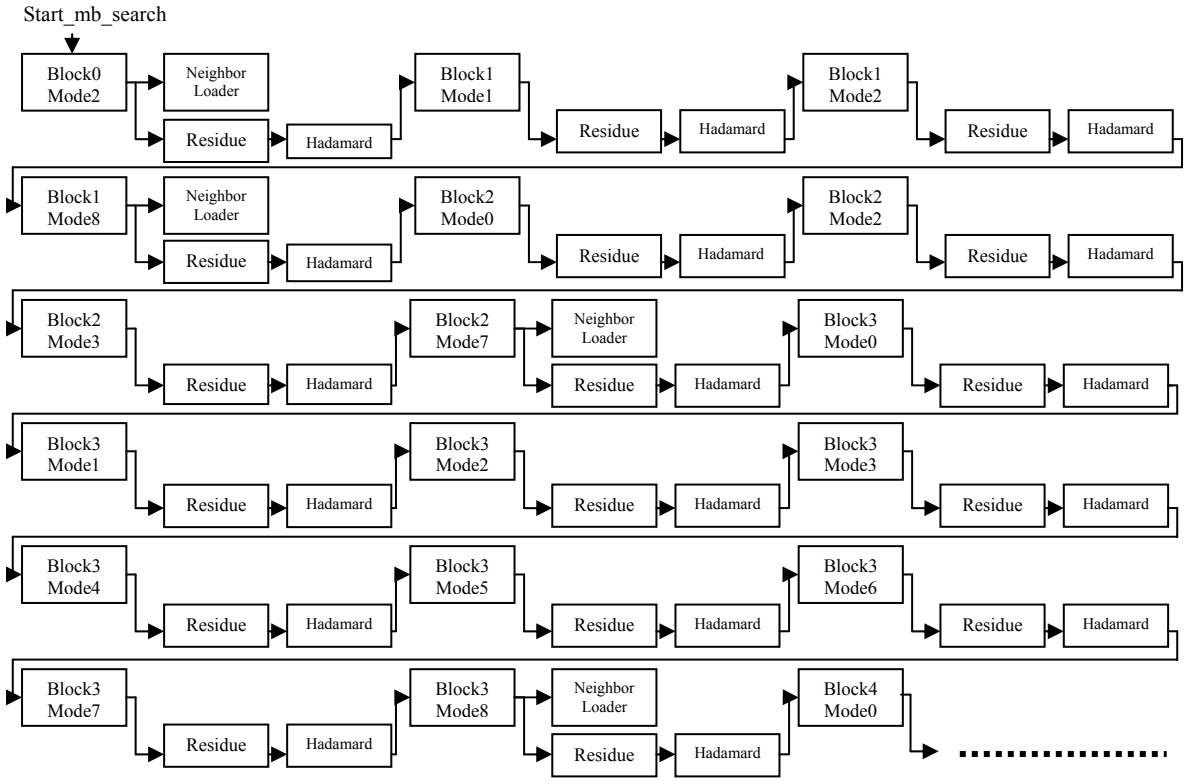



**Figure 4.5** Initial Schedule for 4x4 Luma Prediction Modes

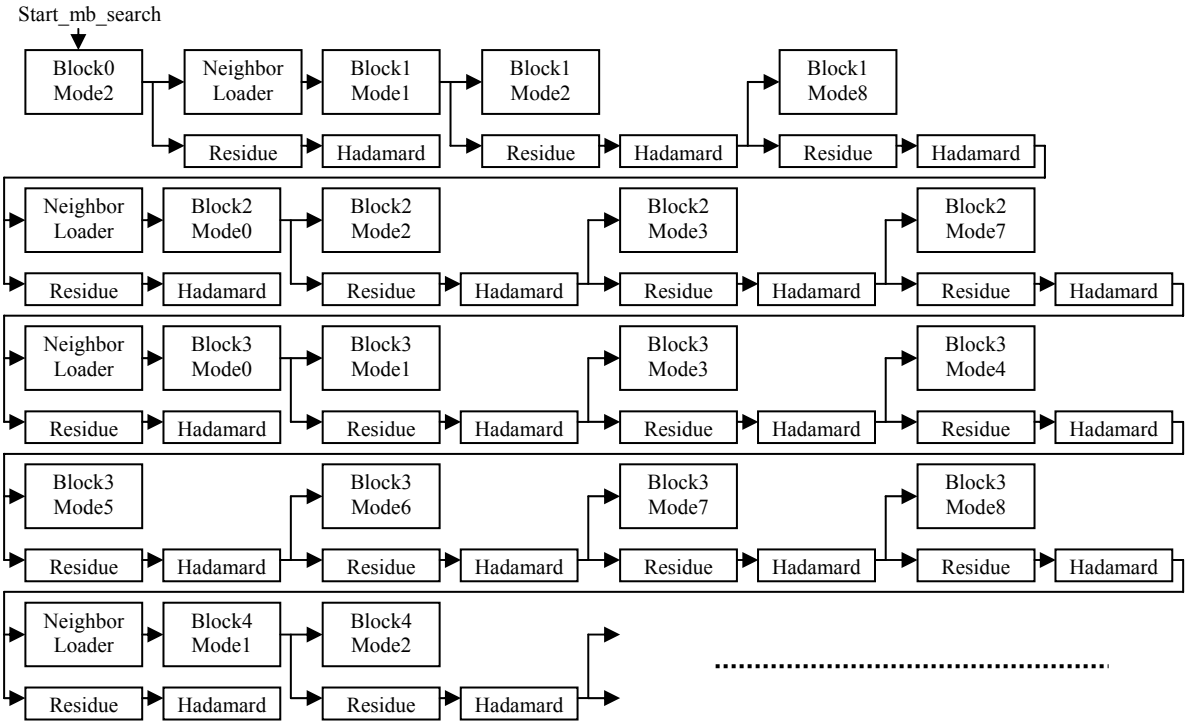



**Figure 4.6** Final Schedule for 4x4 Luma Prediction Modes

Before intra prediction hardware for the first available mode of a 4x4 luma block starts, the corresponding entries of the neighboring buffers for that 4x4 block in the prediction hardware are loaded with the neighboring pixels from the current MB register file. After generating pixel predictions of a 4x4 luma block using an available 4x4 luma prediction mode, the difference (residue) between the current 4x4 luma block and the predicted 4x4 luma block is calculated by Residue module. When the Residue module finishes the calculation of residue data for a 4x4 luma block for the current available 4x4 luma prediction mode, Hadamard Transform module starts to calculate SATD for that mode using the residue data. After Hadamard Transform finishes to calculate SATD for a 4x4 luma prediction mode, mode decision hardware for 4x4 luma blocks determines whether this prediction mode is the mode with lowest cost or not.

Intra prediction module is overlapped with Residue and Hadamard Transform modules. As the Residue and Hadamard Transform modules are working on the current available 4x4 luma prediction mode for a 4x4 luma block, intra prediction module starts to generate the prediction for the next available 4x4 luma prediction mode for the same 4x4

luma block if the current available 4x4 prediction mode is not the last available mode for the current 4x4 luma block.

If the current available 4x4 prediction mode is the last available 4x4 luma prediction mode for the current 4x4 luma block, Neighbor Loader module starts to load the corresponding neighboring pixels for the next 4x4 luma block from the current MB register file as the Residue module is working on the current 4x4 luma block. The Residue module is again followed by Hadamard Transform module. After Neighbor Loader finishes loading the neighboring pixels of the next 4x4 luma block, intra prediction module starts to generate the prediction for the first available 4x4 luma prediction mode for the next 4x4 luma block. All the 4x4 luma blocks in a MB are processed in this order.

The overlapping between Intra Prediction and Residue modules requires using two prediction buffers. While Intra Prediction module is generating the prediction for the next available 4x4 luma prediction mode and writing the results into one of the prediction buffers, Residue module is reading the predicted pixels for the current 4x4 luma prediction mode from the other prediction buffer for residue data calculation.

After intra prediction for all 4x4 blocks in a MB is finished, most probable mode calculation module determines the number of selected modes which are not the most probable mode for each 4x4 block in a MB and uses this information to calculate the cost of using intra 4x4 prediction for a MB (for each 4x4 block, $Cost_{4x4}$ = SATD + 4λR, where R=0 when selected mode is the most probable mode and R=1 otherwise). Most probable mode calculation module has vertical and horizontal buffers that are used for storing the most probable mode information of the 4x4 blocks in the MB boundaries. Vertical buffer is used for storing most probable mode information for the blocks 5, 7, 13 and 15 in the left-hand previously coded MB. Horizontal buffer is used for storing most probable mode information for the blocks 10, 11, 14, and 15 in the upper previously coded MBs in the previously coded MB row of the frame.

Finally, Top Level Mode Decision module in Figure 4.3 uses the results produced by the individual mode decision modules of the lower and upper parts of search & mode decision hardware to determine the prediction modes with lowest cost for a MB (one mode for luma samples and one mode for chroma samples) and sends this information to the coder hardware. In order to complete the SATD operations faster, a high speed Hadamard

transform implementation is taken from [15] and integrated into our design. The proposed hardware finishes SATD operations of a 4x4 block in 18 clock cycles.

The latencies of the modules in the lower part of the search & mode decision hardware are given in Table 4.2.

**Table 4.2** Latencies of the Modules in the Lower Part of the Search & Mode Decision Hardware

| **Module** | **Latency** |
|---|---|
| Neighbor Loader | 16 clock cycles |
| Hadamard Transform | 18 clock cycles |
| Residue | 18 clock cycles |
| Intra Prediction – Preprocessing | 8 clock cycles |
| Intra Prediction – Mode0 | 17 clock cycles |
| Intra Prediction – Mode1 | 17 clock cycles |
| Intra Prediction – Mode2 | 19 clock cycles |
| Intra Prediction – Mode3 | 18 clock cycles |
| Intra Prediction – Mode4 | 18 clock cycles |
| Intra Prediction – Mode5 | 17 clock cycles |
| Intra Prediction – Mode6 | 17 clock cycles |
| Intra Prediction – Mode7 | 17 clock cycles |
| Intra Prediction – Mode8 | 17 clock cycles |

For the schedule shown in Figure 4.5, after the prediction for each mode is generated, it takes 16 clock cycles for the Residue module to generate the residue block for that mode (loading neighbors is overlapped). After the residue block is generated, the Hadamard Transform module is started and it takes 18 clock cycles to finish the transform process. 4 extra cycles are required after the Hadamard Transform module before starting intra prediction for the next available mode of the same 4x4 block. So, in the worst case when all 4x4 modes are available, it takes 165 (Intra Prediction) + 16*9 (Residue) + 16*9 (Hadamard Transform) + 4*9 = 507 clock cycles for performing intra search for a 4x4 luma block. After intra search for all 4x4 luma blocks in a MB is done, total cost for the selected modes for each 4x4 luma block in a MB is calculated in 18 clock cycles. Most probable mode calculation for 4x4 blocks in a MB is, then, started and this calculation takes 36 clock cycles. Finally, cost comparison between 16x16 and 4x4 intra search is initiated and it takes 9 clock cycles. Therefore, intra search for a MB takes (16*507) + 18 + 36 + 9 = 8175 clock cycles when the initial schedule is used. Since 30 fps CIF coding cannot be achieved using this schedule, we used the schedule shown in Figure 4.6.

For the schedule in Figure 4.6, after the prediction for each mode is generated (hadamard transform is overlapped), it takes 16 cycles for the Residue module to generate the residue block for that mode (loading neighbors is overlapped). 1 extra cycle is required after the Hadamard Transform module before starting intra prediction for the next available mode of the same 4x4 block. So, in the worst case when all 4x4 modes are available, it takes 165 (Intra Prediction) + 16*9 (Residue) + 1*9 = 318 clock cycles for performing intra search for a 4x4 luma block. After intra search for all 4x4 luma blocks in a MB is done, total cost for the selected modes for each 4x4 luma block in a MB is calculated in 18 clock cycles. Most probable mode calculation for 4x4 blocks in a MB is, then, started and this calculation takes 36 clock cycles. Finally, cost comparison between 16x16 and 4x4 intra search is initiated and it takes 9 clock cycles. Therefore, intra search for a MB takes (16*318) + 18 + 36 + 9 = 5151 clock cycles when the final schedule is used.

### 4.1.2 Coder Hardware

The coder hardware is mostly taken from [15]. As shown in Figure 4.7, it includes Intra Prediction, Residue, Transform, Quant, Inverse Transform, Inverse Quant, Hadamard Transform, Reconstruction, and Entropy Coder modules.

After the search & mode decision hardware determines the best modes for luma and chroma components of a MB, the MB is loaded to the current MB register file in the coder hardware. As soon as this loading operation finishes, intra prediction hardware generates the predicted MB using the selected best mode. Then, the Residue module creates the residual data by taking the difference between the current MB and the predicted MB and it loads the residual data to the input register file of the Transform-Quant hardware. Reconstruction module adds the results of Inverse Transform module which is stored in a 16x16 register file and the corresponding intra predicted data from the predicted MB register and clips the result to the [0-255] range. The results obtained from the reconstruction process are loaded to the neighboring pixel buffers in the intra prediction hardware and the reconstructed MB register file.

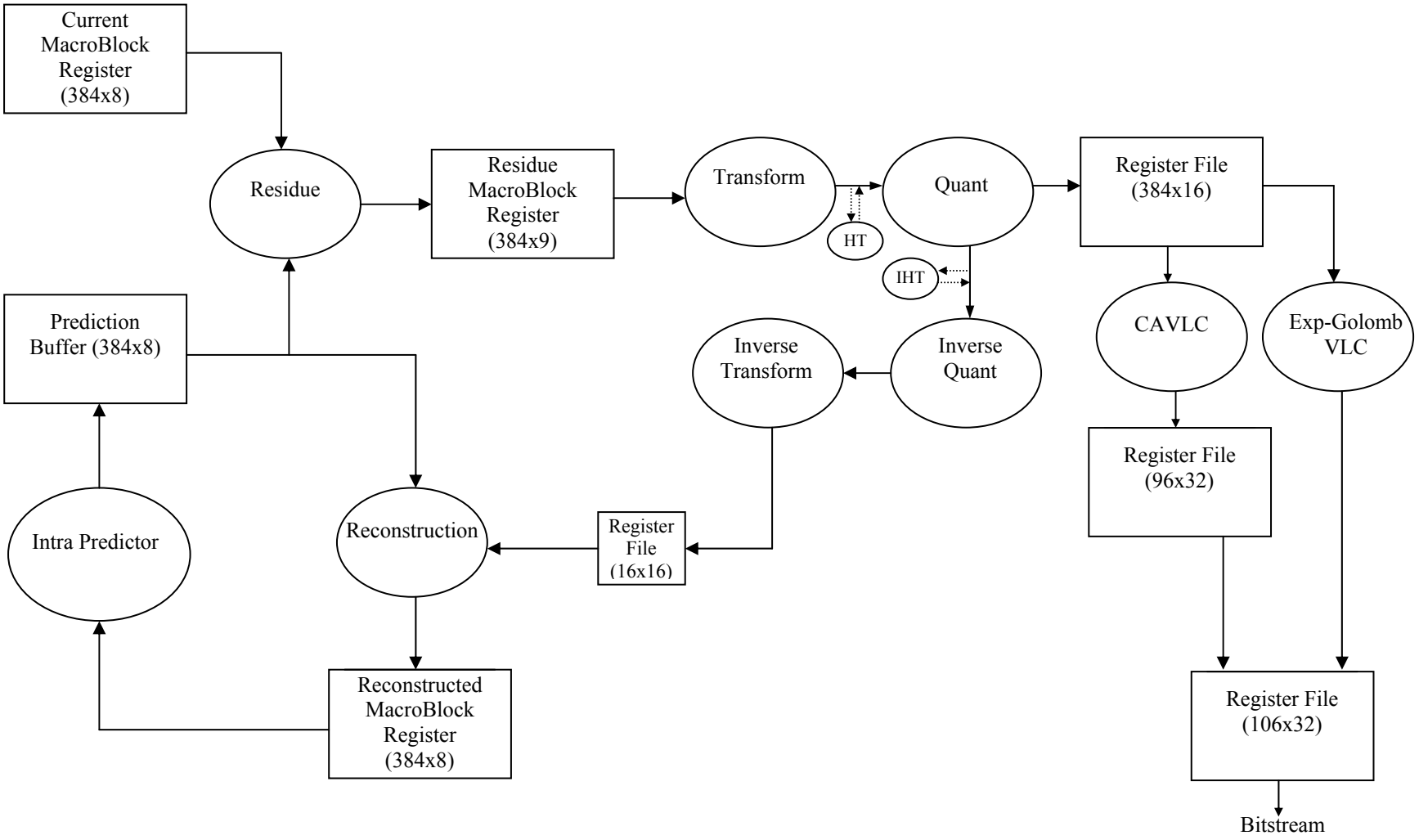



**Figure 4.7** Block Diagram of Coder Hardware

The scheduling of the Coder Hardware for a MB that will be coded with 4x4 luma prediction modes is shown in Figure 4.8 [15]. In the worst case, it takes 2676 clock cycles to code a MB that will be coded with 4x4 luma prediction modes. First, intra prediction hardware generates all pixel predictions for a MB based on the selected mode information for each 4x4 luma block and writes these results to the predicted MB register file. Then, the Residue block subtracts the predicted MB from the current MB. When the residual data for the first 4x4 luma block is available, Transform-Quant module starts to generate the quantized transform coefficients and loads these coefficients to the input register file of CAVLC hardware. After the quantized transform coefficients of the first 4x4 block are loaded, CAVLC and inverse Transform – Quant modules start to work. The bitstream generated by CAVLC module is stored in the output register file of CAVLC hardware. After Transform – Quant module finishes inverse quant and inverse transform operations for the first 4x4 block, reconstruction block starts to work. After the first 4x4 block of a MB is coded and reconstructed, the coder hardware starts to work on the second 4x4 block. In this way, all 4x4 blocks in a MB are coded and reconstructed.

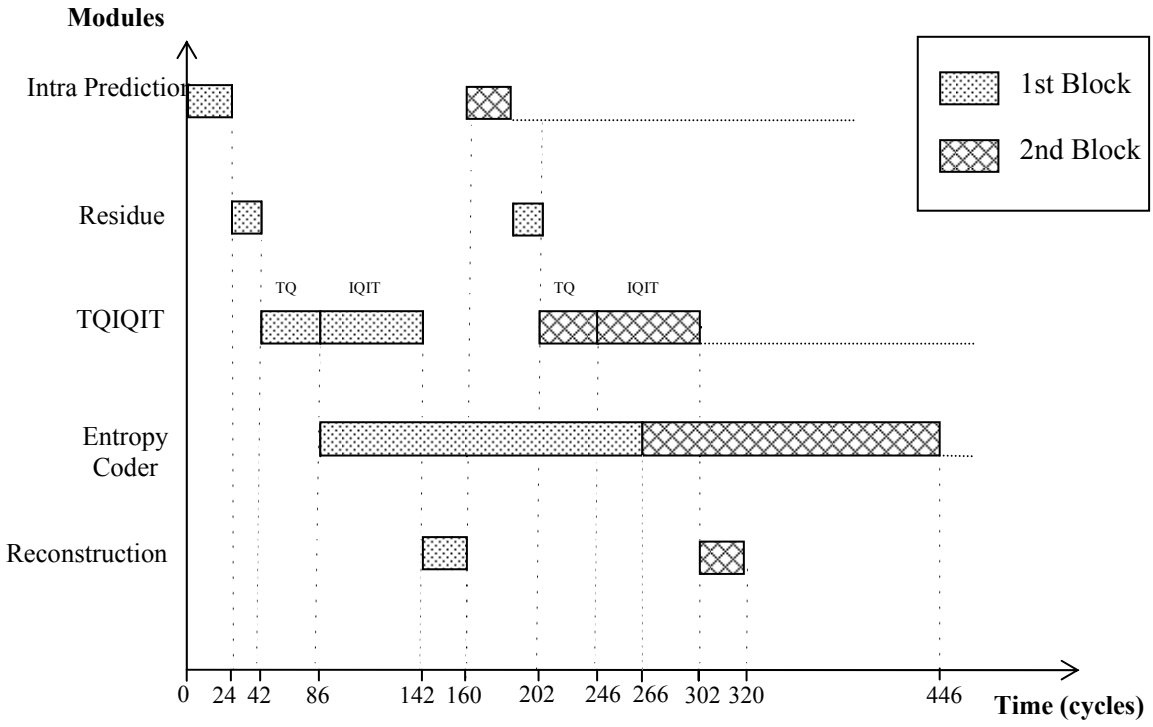



**Figure 4.8** Coder Hardware Scheduling for 4x4 Intra Modes

The scheduling of the Coder Hardware for a MB that will be coded with a 16x16 luma prediction mode is shown in Figure 4.9 [15]. In the worst case, it takes 3680 clock cycles to code a MB that will be coded with a 16x16 luma prediction mode. Hadamard Transform has to be applied to DC coefficients after 4x4 integer transforms. Therefore, inverse quant, inverse transform, CAVLC and reconstruction operations for the MB can only start after the Hadamard transform finishes.

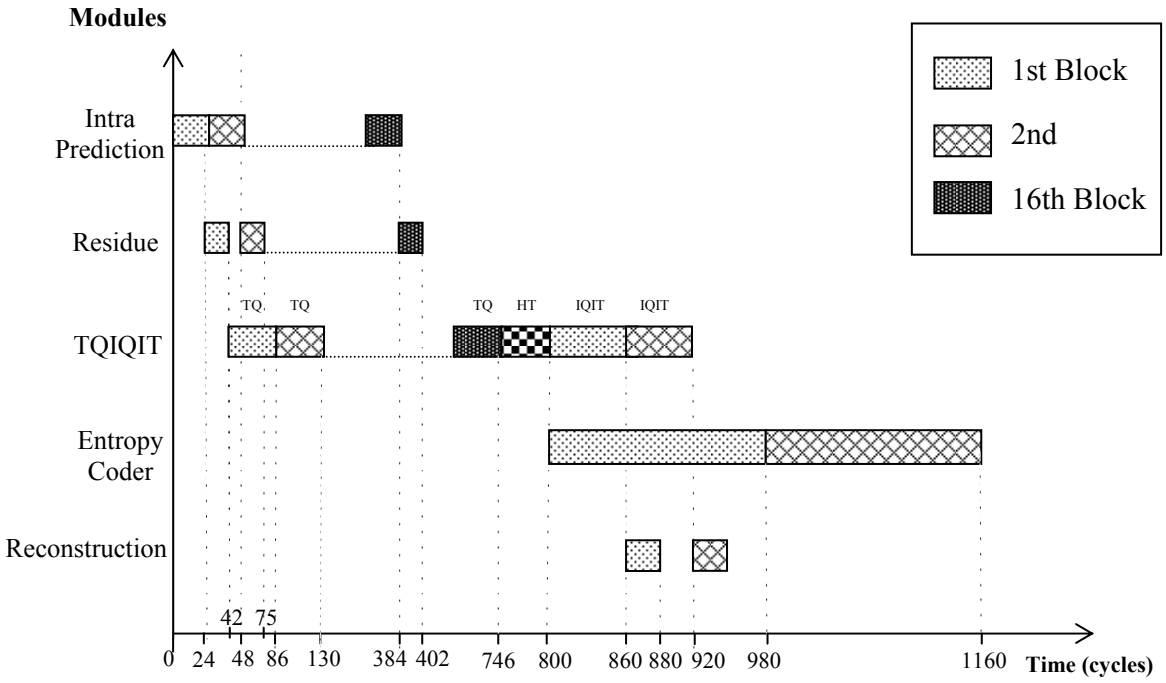



**Figure 4.9** Coder Hardware Scheduling for 16x16 Intra Modes

### 4.1.3 Implementation Results

The proposed architecture is implemented in Verilog HDL. The implementation is verified with RTL simulations using Mentor Graphics ModelSim SE. The Verilog RTL is then synthesized to a 2V8000ff1152 Xilinx Virtex II FPGA with speed grade 5 using Mentor Graphics Leonardo Spectrum. The resulting netlist is placed and routed to the same FPGA using Xilinx ISE Series 7.1i. The FPGA implementation is verified to work at 71 MHz on a Xilinx Virtex II FPGA on an ARM Versatile Platform development board.

The H.264 intra frame coder hardware is verified to be compliant with H.264 standard. As shown in Figure 4.10, the bitstream generated by the H.264 intra frame coder hardware for an input frame is successfully decoded by H.264 Joint Model (JM) reference software decoder and the decoded frame is displayed using a YUV Player tool for visual verification.

The proposed H.264 intra frame coder hardware includes a search & mode decision hardware and a coder hardware that work in a pipelined manner. Since, in the worst case, the search & mode decision hardware takes 5151 clock cycles for a MB and the coder hardware takes 3680 clock cycles for a MB, the intra frame coder hardware takes 5151 clock cycles for a MB. Therefore, the FPGA implementation can process a CIF frame in 396 MB * 5151 clock cycles per MB * 14 ns clock cycle = 28.5 msec. Therefore, it can process 1000/28.5 = 35 CIF (352x288) frames per second.

The FPGA implementation including input, output and internal RAMs and register files uses the following FPGA resources; 19589 Function Generators, 9795 CLB Slices, 3698 Dffs/Latches, and 1 Block Multiplier, i.e. %21.02 of Function Generators, %21.02 of CLB Slices, %3.83 of Dffs/Latches, and %0.6 of Block Multipliers.


**(a) Input Frame**


**(b) Encoded and Decoded Frame**

**Figure 4.10** Visual Verification of H.264 Intra Frame Coder Hardware

# CHAPTER 5

# CONCLUSIONS AND FUTURE WORK

## 5.1 Conclusions

In this thesis, we developed an efficient FPGA-based H.264 intra frame coder Hardware for portable applications targeting level 2.0 of baseline profile. We first designed a high performance and low cost hardware architecture for real-time implementation of entropy coding algorithms, context adaptive variable length coding and exp-golomb coding, used in H.264 video coding standard. The hardware is implemented in Verilog HDL and verified with RTL simulations using Mentor Graphics Modelsim. We then designed a high performance and low cost hardware architecture for real-time implementation of intra prediction algorithm used in H.264 video coding standard. This hardware is also implemented in Verilog HDL and verified with RTL simulations using Mentor Graphics Modelsim. We then designed and implemented the top-level H.264 intra frame coder hardware. The hardware is implemented by integrating intra prediction, mode decision, transform-quant and entropy coding modules. The H.264 intra frame coder hardware is verified to be compliant with H.264 standard and it can code 35 CIF (352x288) frames per second. The hardware is first verified with RTL simulations using Mentor Graphics Modelsim. It is then verified to work at 71 MHz on a Xilinx Virtex II FPGA on an ARM Versatile Platform development board.

## 5.2 Future Work

The FPGA-based H.264 intra frame coder implementation can be modified as an ASIC implementation and prototypes can be fabricated.

The power consumption of the H.264 intra frame coder hardware can be analyzed. Based on this analysis, low-power techniques such as clock gating and glitch reduction can be used to reduce its power consumption.

The H.264 intra frame coder hardware can be used to investigate the rate-distortion performance of the current intra frame search and mode decision algorithm and to develop better intra frame search and mode decision algorithms.

A complete H.264 video encoder hardware can be implemented by integrating motion estimation, motion compensation, de-blocking filter, intra vs. inter mode decision and rate control modules to the H.264 intra frame coder hardware.

## REFERENCES


[1] R. Schäfer, T. Wiegand and H. Schwarz, "The Emerging H.264/AVC Standard", *EBU Technical Review*, January 2003.

[2] Personal Communication with Hasan Ateş.

[3] T. Wiegand, G. J. Sullivan, G. Bjøntegaard, and A. Luthra "Overview of the H.264/AVC Video Coding Standard", *IEEE Trans. on Circuits and Systems for Video Technology vol.* 13, no. 7, pp. 560–576, July 2003.

[4] I. Richardson, H.264 and MPEG-4 Video Compression*,* Wiley, 2003.

[5] Joint Video Team (JVT) of ITU-T VCEG and ISO/IEC MPEG, Draft ITU-T. Recommendation and Final Draft International Standard of Joint Video Specification, ITU-T Rec. H.264 and ISO/IEC 14496-10 AVC, May 2003.

[6] O.Tasdizen, I. Hamzaoglu, "A High Performance And Low Cost Hardware Architecture for H.264 Transform And Quantization Algorithms", *13th European Signal Processing Conference*, September 2005.

[7] Yu-Wen Huang, Bing-Yu Hsieh, Tung-Chien Chen, and Liang-Gee Chen, "Hardware Architecture Design for H.264/AVC Intra Frame Coder," Proc. of IEEE ISCAS, pp. 269-272, April 2004.

[8] Yu-Wen Huang, Bing-Yu Hsieh, Tung-Chien Chen, and Liang-Gee Chen, "Analysis, Fast Algorithm, and VLSI Architecture Design for H.264/AVC Intra Frame Coder," IEEE Transactions on Circuits and Systems for Video Technology, Vol. 15, No. 3, March 2005.

[9] Joint Video Team (JVT) of ITU-T VCEG and ISO/IEC MPEG, *Joint Model (JM) Reference Software Version 8.2*, http://bs.hhi.de/suehring/.

[10] V. Bhaskaran and K. Konstantinides, *Image and Video Compression Standards: Algorithms and Architectures*, Kluwer Academic Publishers, 2nd Edition, 1997.

[11] D. Marpe, T. Wiegand, and S. Gordon, “H.264/MPEG4-AVC Fidelity Range Extensions: Tools, Profiles, Performance and Application Areas,” in *Proc. IEEE Int. Conf. Image Processing*, vol. 1, pp. 593-596, Genova, Italy, Sept. 2005.

[12] D. Marpe, V. George, H. L. Cycon, and K. U. Barthel, “Performance Evaluation of Motion-JPEG2000 in Comparison with H.264/AVC Operated in Intra Coding Mode, ” in *Proc. SPIE*, vol. 5266, pp. 129-137, Feb. 2004.

[13] G. Bjontegaard, “Calculation of Average PSNR Differences Between RD-Curves,” document VCEG-M33, Austin, USA, 2001.

[14] E. Sahin, I. Hamzaoglu, “A High Performance and Low Power Hardware Architecture for H.264 CAVLC Algorithm", *13th European Signal Processing Conference*, September 2005.

[15] O. Tasdizen, “H.264 Intra Frame Coder System Design”, MS Thesis, Sabanci University, August 2005.