OPTIMIZATION OF H.264 BASELINE DECODER ON ARM9TDMI PROCESSOR

by

SANDYA BASAVANAHALLI SHESHADRI

Presented to the Faculty of the Graduate School of

The University of Texas at Arlington in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

THE UNIVERSITY OF TEXAS AT ARLINGTON

December 2005

ACKNOWLEDGEMENTS

I am grateful to lot of people who helped me to shape up this thesis.

I am greatly indebted to Dr. K.R.Rao for his support, guidance and encouragement right from the beginning of my work. His courses Digital Image Processing and Video Coding Standards made my concepts about video processing much clearer. He also helped me in getting the industry experience in FastVDO LLC, Columbia, MD, where I am gaining a lot real world experience. I thank him for all the support.

Dr. Pankaj Topiwala, president of FastVDO LLC, was generous to allow me to run tests on FastVDO's Baseline decoder© and let me utilize all the resources and licenses for ADSv1.2, without which this work would not have been in reality. I am greatly thankful to him. I received many helpful comments and suggestions from Patrick Rault, Basavaraj Mudigoudar, Sachin Patil, and Tarun Batia who are my colleagues in FastVDO.

I thank Dr. Devarajan and Dr. Wang for accepting to be my committee members and review my thesis.

Last but not the least I thank my parents and my brother for their love and support in every walk of my life.

November 18, 2005

ABSTRACT

OPTIMIZATION OF H.264 BASELINE DECODER

ON ARM9TDMI PROCESSOR

Publication No. ______

Sandya Basavanahalli Sheshadri, MS

The University of Texas at Arlington, 2005

Supervising Professor: Dr. K. R. Rao

With the newly introduced features and advancements to the pre-existing features, the emerging H.264 video coding standard achieves significant improvements in coding performance over all existing standards, in a wide variety of applications. The coding-efficiency advantages of H.264, however, come at the expense of higher computational complexity. H.264 decoders can exhibit more than double the complexity of H.263 decoders. Furthermore, previous studies have shown that fractional-pixel motion-compensation interpolation and the loop filtering consume a significant amount of computational power in emerging H.264 decoders. Since these operations are part of the baseline profile of H.264, there is a need to evaluate new ways for minimizing complexity for H.264 decoders on low-complexity devices. In particular, new wireless devices have both complexity and bit rate constraints, yet the range of these constraints differ from traditional systems (e.g., powerful PCs that are networked over the best-effort Internet). Under common operational scenarios, a low complexity wireless handheld may have significantly greater complexity/power constraints than bit rate limitation (e.g., over a wireless access LAN).

This thesis analyzes the bottlenecks of H.264 decoders on ARM9TDMI processor, targeted for mobile devices, using performance-profiling tools. Optimizations are performed to achieve real time decoding. The code is built with Real View Compiler for ARM and ported on Symbian using Metroworks© Codewarrior© for Symbian V3.0 to achieve real time H.264 decoding on Nokia 6630 cellphone. The compiler flags were optimized for speed.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

LIST OF TABLES

LIST OF ACRONYMS

ARM9TDMI: Advanced RISC Microprocessor, Thumb, JTAG debug, fast multiplier, EmbeddedICE macrocell

AVC: Advanced Video Coding

DVD: Digital Video Disc

GOP: Group of Pictures

IDR: Intra Decoder Refresh

IEC: International Electro technical Commission

ISO: International Standards Organization

ITU: International Telecommunication Union

JVT: Joint Video Team

MPEG: Moving Picture Experts Group

NALU: Network Abstraction Layer Unit

PPS: Picture Parameter Set

RISC: Reduced Instruction Set Computer

SEI: Supplemental Enhancement Information

SI: Switching Intra

SP: Switching Prediction

SPS: Sequence Parameter Set

CHAPTER 1

INTRODUCTION

1.1 Overview: H.264 Video Coding Standard

Development of the international video coding standards such as MPEG-1, MPEG-2 and H.261 boosted a diverse range of multimedia applications, including digital video recording, and teleconferencing [1, 2, 3]. As a result of ongoing demand for better compression performance, advanced standards such as MPEG-4 and H.263 were also introduced [4, 5]. In order to provide better compression of video compared to previous standards, H.264 / MPEG-4 part 10 [4] video coding standard was recently developed by the JVT (Joint Video Team) [6] consisting of experts from VCEG (Video Coding Experts Group) and MPEG (Moving Pictures Experts Group). H.264 fulfills significant coding efficiency, simple syntax specifications, and seamless integration of video coding into all current protocols and multiplex architectures. Thus H.264 can support various applications like video broadcasting, video streaming, and video conferencing over fixed and wireless networks and over different transport protocols. The coding-efficiency advantages of H.264, however, come at the expense of higher computational complexity. For example, the study in [7] showed that H.264 decoders can exhibit more than double the complexity of H.263 decoders. Furthermore, previous studies have shown that fractional-pixel motion-compensation interpolation and the loop filtering consume a significant amount of computational power in emerging H.264

decoders [7, 8]. Since these operations are part of the baseline profile of H.264, there is a need to evaluate new ways for minimizing complexity for H.264 decoders on low-complexity devices. In particular, new wireless devices have both complexity and bit rate constraints, yet the range of these constraints differ from traditional systems (e.g., powerful PCs that are networked over the best-effort Internet). Under common operational scenarios, a low complexity wireless handheld may have significantly greater complexity/power constraints than bit rate limitation (e.g., over a wireless access LAN).

1.2 Applications and Design Feature Highlights

H.264 is designed for technical solutions including at least the following application areas

• Broadcast over cable, satellite, cable modem, DSL (Digital Subscriber Line), terrestrial, etc.

• Interactive or serial storage on optical and magnetic devices, DVD, etc.

• Conversational services over ISDN (Integrated Services Digital Network), Ethernet, LAN (Local Area Network), DSL, wireless and mobile networks, modems, etc. or mixtures of these.

• Video-on-demand (VOD) or multimedia streaming services over ISDN, cable modem, DSL, LAN, wireless networks, etc.

• Multimedia messaging services (MMS) over ISDN, DSL, Ethernet, LAN, wireless and mobile networks, etc.

Moreover, new applications may be deployed over existing and future networks. This raises the question about how to handle this variety of applications and networks. To address this need for flexibility and customizability, the H.264/AVC (Advanced Video Coding) design covers a Video Coding Layer (VCL), which is designed to efficiently represent the video content, and a Network Abstraction Layer (NAL), which formats the VCL representation of the video and provides header information in a manner appropriate for conveyance by a variety of transport layers or storage media.

Relative to prior video coding methods [16], some highlighted features of the design that enable enhanced coding efficiency include the following enhancements of the ability o predict the values of the content of a picture to be encoded:

• Variable block-size motion compensation with small block sizes: This standard supports more flexibility in the selection of motion compensation block sizes and shapes than any previous standard, with a minimum luma motion compensation block size as small as 4x4.

• Quarter-sample-accurate motion compensation: Most prior standards enable half sample motion vector accuracy at most. Accuracy improves by adding quarter sample motion vector, as first found in an advanced profile of the MPEG-4 Visual (part 2) standard [22], but further reduces the complexity of the interpolation processing compared to the prior design.

• Motion vectors over picture boundaries: While motion vectors in MPEG-2 and its predecessors were required to point only to areas within the previously-decoded reference picture, the picture boundary extrapolation technique first found as an optional feature in H.263 [22] is included in H.264/AVC.

• Multiple reference picture motion compensation: Predictive coded pictures (called "P" pictures) in MPEG-2 and its predecessors used only one previous picture to predict the values in an incoming picture. The new design extends upon the enhanced reference picture selection technique found in H.263++ [21] to enable efficient coding by allowing an encoder to select, for motion compensation purposes, among a larger number of pictures that have been decoded and stored in the decoder.

• Improved "skipped" and "direct" motion inference: In prior standards, a "skipped" area of a predictive-coded picture could not infer motion in the scene content. This had a detrimental effect when coding video containing global motion, so the new H.264/AVC design instead infers motion in "skipped" areas. For bi-predictive coded areas (called B slices), H.264/AVC also includes an enhanced motion inference method known as "direct" motion compensation, which improves further on prior "direct" prediction designs found in H.263+ and MPEG-4 Visual.

• Directional spatial prediction for intra coding: A new technique of extrapolating the edges of the previously decoded parts of the current picture is applied in regions of pictures that are coded as intra (i.e., coded without reference to the content of some other picture). This improves the quality of the prediction signal, and also allows prediction from neighboring areas that were not coded using intra coding (something not enabled when using the transform-domain prediction method found in H.263+ and MPEG-4 Visual).

• In-the-loop deblocking filtering: Block-based video coding produces artifacts known as blocking artifacts. These can originate from both the prediction and residual difference coding stages of the decoding process. Application of an adaptive deblocking filter is a well-known [15] method of improving the resulting video quality, and when designed well, this can improve both objective and subjective video quality. Building further on a concept from an optional feature of H.263+, the deblocking filter in the H.264/AVC design is brought within the motion-compensated prediction loop, so that this improvement in quality can be used in inter-picture prediction to improve the ability to predict other pictures as well.

• Small block-size transform: All major prior video coding standards used a transform block size of 8x8, while the new H.264/AVC design is based primarily on a 4x4 transform. The smaller block size is also justified partly by the advances in the ability to better predict the content of the video using the techniques noted above, and by the need to provide transform regions with boundaries that correspond to those of the smallest prediction regions.

• Hierarchical block transform: While in most cases, using the small 4x4 transform block size is perceptually beneficial, there are some signals that contain sufficient correlation to call for some method of using a representation with longer basis functions. The H.264/AVC standard enables this in two ways: by using a hierarchical transform to extend the effective block size use for low frequency chroma information to 8x8 array and by allowing the encoder to select a special coding type for intra coding, enabling extension of the length of the transform for luminance components for low-frequency information to a 16x16 block size in a manner very similar to that applied to the chroma.

• Short word-length transform: All prior standard designs have effectively required encoders and decoders to use more complex processing for transform computation. While previous designs have generally required 32-bit processing, the H.264/AVC design requires only 16-bit arithmetic for the 4x4 integer DCT.

• Exact-match inverse transform: In previous video coding standards, the transform used for representing the video was generally specified only within an error tolerance bound, due to the impracticality of obtaining an exact match to the ideal specified inverse transform. As a result, each decoder design would produce slightly different decoded video, causing a "drift" between encoder and decoder representation of the video and reducing effective video quality. H.264/AVC is the first standard to achieve exact equality of decoded video content from all decoders.

• Arithmetic entropy coding: An advanced entropy coding method known as arithmetic coding is included in H.264/AVC. While arithmetic coding was previously found as an optional feature of H.263, a more effective use of this technique is found in H.264/AVC to create a very powerful entropy coding method known as CABAC (context-adaptive binary arithmetic coding) which is not used in baseline profile.

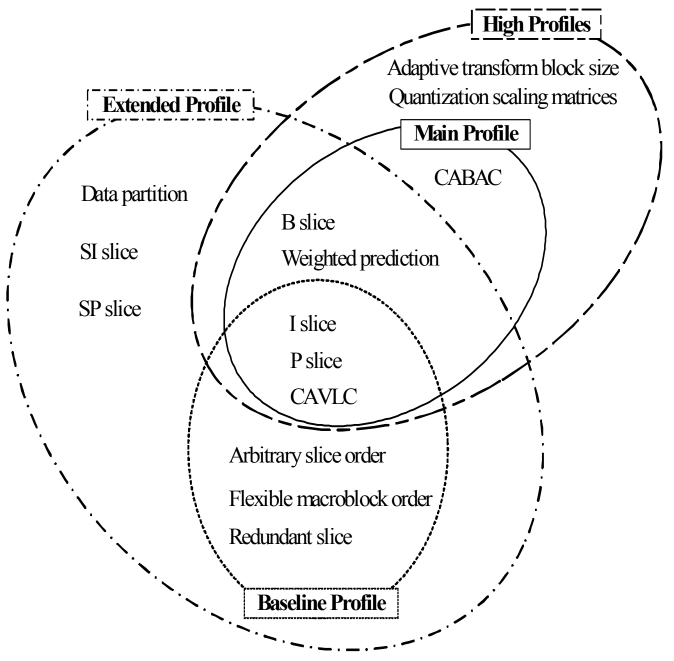


Figure 1.1: Profiles in H.264 [22]

• Context-adaptive entropy coding: The two entropy-coding methods applied in H.264/AVC, termed CAVLC (context-adaptive variable-length coding) and CABAC, both use context-based adaptivity to improve performance relative to prior standard designs.

• Arbitrary slice ordering (ASO): Since each slice of a coded picture can be (approximately) decoded independently of the other slices of the picture, the H.264/AVC design enables sending and receiving the slices of the picture in any order relative to each other. This capability, can improve end-to-end delay in real-time applications, particularly when used on networks having out-of-order delivery behavior (e.g., internet protocol networks).

• Redundant pictures: In order to enhance robustness to data loss, the H.264/AVC design contains a new ability to allow an encoder to send redundant representations of regions of pictures, enabling a (typically somewhat degraded) representation of regions of pictures for which the primary representation has been lost during data transmission.

• Data Partitioning: Since some coded information for representation of each region (e.g., motion vectors and other prediction information) is more important or more valuable than other information for purposes of representing the video content, H.264/AVC allows the syntax of each slice to be separated into up to three different partitions for transmission, depending on a categorization of syntax elements. Here the design is simplified by having a single syntax with partitioning of that same syntax controlled by a specified categorization of syntax elements

1.3 Layered Structure

The H.264 standard is designed in two distinct layers: a video coding layer (VCL), and a network adaptation layer (NAL) as shown in Figure 1.2. This work is concentrated on the performance of the VCL.

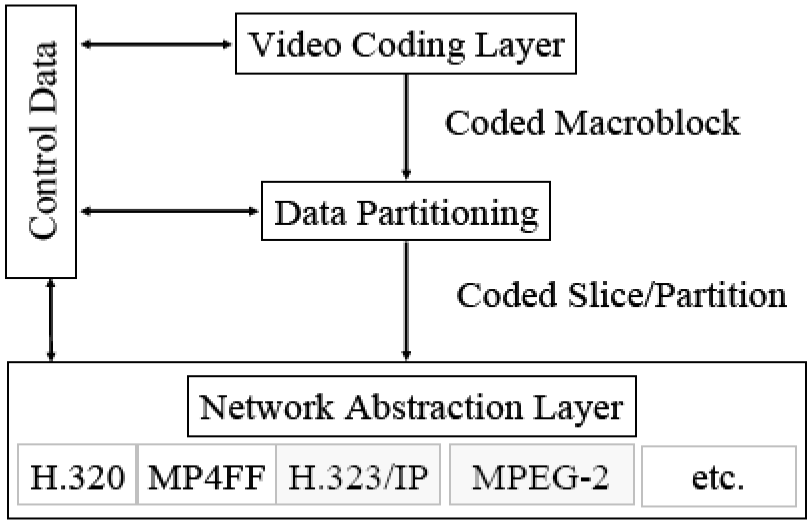


Figure 1.2: Layered Structure of H.264 Video [16]

1.3.1 Network Abstraction Layer

NAL abstracts the VCL data in a manner that is appropriate for conveyance on a variety of communication channels or storage media. For the friendliness of communication channel, a NAL unit specifies both byte-stream and packet-based formats. Also, non-VCL and VCL NAL units classify a NAL unit.

The non-VCL unit contains the additional information such as parameter setting. Previous standards contained header information about slice, picture and sequence that was coded at the start of each element. The loss of packet containing this header information would make the data dependent on this header, as useless. H.264 overcomes this shortcoming by making the packets transmitted synchronously in a real-time multimedia environment as self-contained [5]. Parameters that change very frequently are added to the slice layer.

1.3.2 Video Coding Layer

The VCL unit contains the core video coded data, which consists of video sequence, picture, slice, and macroblock. As in all prior ITU-T and ISO/IEC JTC1 video standards since H.261 [27], the VCL design follows the so-called block-based hybrid video coding approach, in which each coded picture is represented in block shaped units of associated luma and chroma samples called macroblocks. The basic source-coding algorithm is a hybrid of inter-picture prediction to exploit temporal statistical dependencies and transform coding of the prediction residual to exploit spatial statistical dependencies. A coded picture can represent either an entire frame or a single field, as was also the case for MPEG-2 video [23]. Generally, a frame of video can be considered to contain two interleaved fields, a top and a bottom field. The top field contains even-numbered rows 0, 2,…, and n, where n is an even number. The bottom field contains the odd-numbered rows. If the two fields of a frame were captured at different time instants, the frame is referred to as an interlaced frame, and otherwise it is referred to as a progressive frame (Figure 1.3). Baseline Profile is limited to progressive scan.

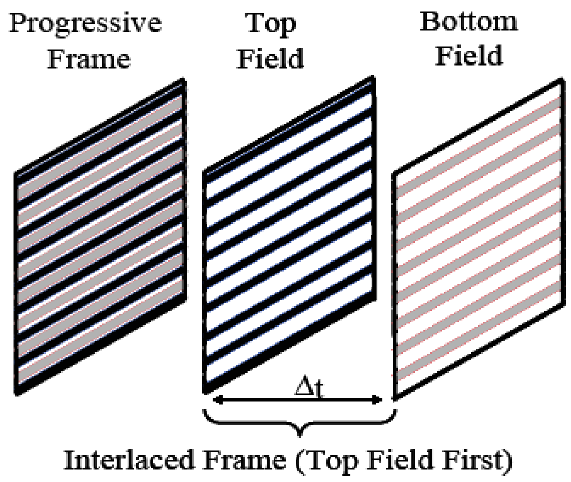


Figure 1.3: Progressive and interlaced frames and fields [16]

1.3.3 YCbCr color space and 4:2:0 sampling

The video color space used by H.264/AVC separates a color representation into three components called Y, Cb, and Cr. Component Y is called luma, and represents brightness. The two chroma components Cb and Cr represent the extent to which the color deviates from gray towards blue and red, respectively. Because the human visual system is more sensitive to luma than chroma, baseline profile of H.264/AVC uses a sampling structure in which the chroma component has one fourth of the number of samples than the luma component (half the number of samples in both the horizontal and vertical dimensions).This is called 4:2:0 sampling with 8 bits of precision per sample as in Fig.1.4 below.

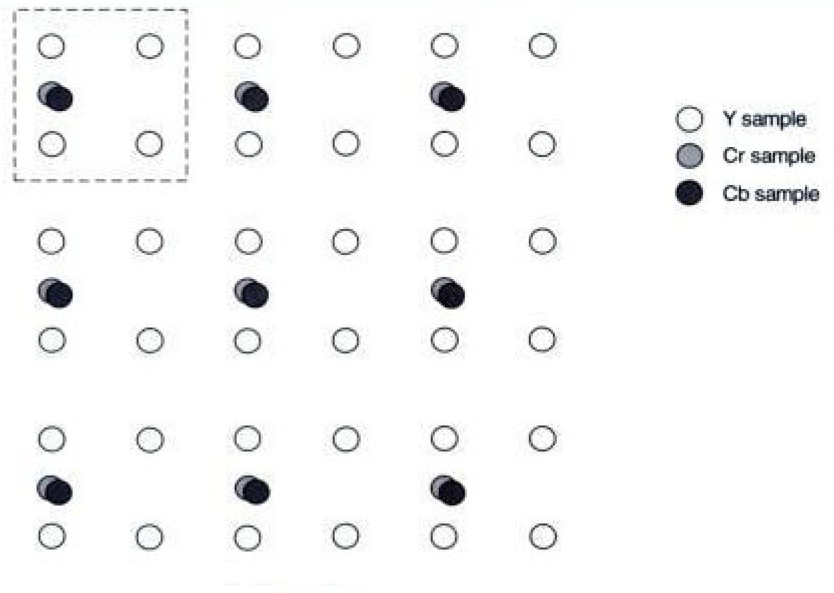


Figure 1.4: 4:2:0 sampling [39]

1.3.4 Division of the picture into macroblocks

A picture is partitioned into fixed-size macroblocks that each covers a rectangular picture area of 16x16 samples of the luma component and 8x8 samples of each of the two chroma components. This partitioning into macroblocks has been adopted into all previous ITU-T video coding standards since H.261. Macroblocks are the basic building blocks of the standard for which the decoding process is specified. The basic coding algorithm for a macroblock is described after explaining how macroblocks are grouped into slices.

1.3.5 Slices and slice groups

Slices are a sequence of macroblocks, which are processed in the order of a raster scan when not using flexible macroblock ordering (FMO) which is described in the next paragraph. A picture may be split into one or several slices as shown in Figure 1.5. A picture is therefore a collection of one or more slices. Slices are self-contained in the sense that given the active sequence and picture parameter sets, their syntax elements can be parsed from the bit-stream and the values of the samples in the area of the picture that the slice represents can be correctly decoded without use of data from other slices provided that utilized reference pictures are identical at encoder and decoder. Some information from other slices may be needed to apply the deblocking filter across slice boundaries.

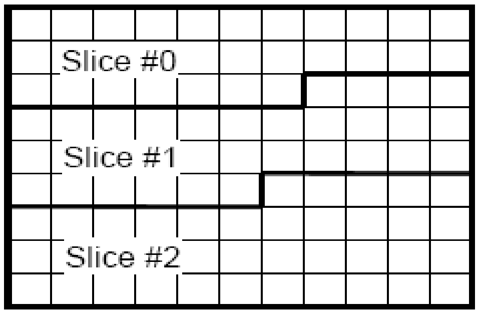


Figure 1.5: Subdivision of a picture into slices when not using FMO [16]

FMO [10] modifies the way in which pictures are partitioned into slices and macroblocks by utilizing the concept of slice groups. Each slice group is a set of macroblocks defined by a macroblock to slice group map, which is specified by the content of the picture parameter set and some information from slice headers. The macroblock to slice group map consists of a slice group identification number for each macroblock in the picture, specifying which slice group the associated macroblock belongs to. Each slice group can be partitioned into one or more slices, such that a slice is a sequence of macroblocks within the same slice group that is processed in the order of a raster scan within the set of macroblocks of a particular slice group. The case when FMO is not in use can be viewed as the simple special case of FMO in which the entire picture consists of a single slice group. Using FMO, a picture can be split into many macroblock-scanning patterns such as interleaved slices, a dispersed macroblock allocation, one or more "foreground" slice groups and a "leftover" slice group, or a checkerboard type of mapping. The latter two are illustrated in Figure 1.6. The left-hand side macroblock to slice group mapping has been demonstrated for use in region-of-interest type of coding applications. The right-hand side macroblock to slice group mapping has been demonstrated as useful for error concealment in video conferencing applications where slice group #0 and slice group #1 are transmitted in separate packets and when one of them is lost.

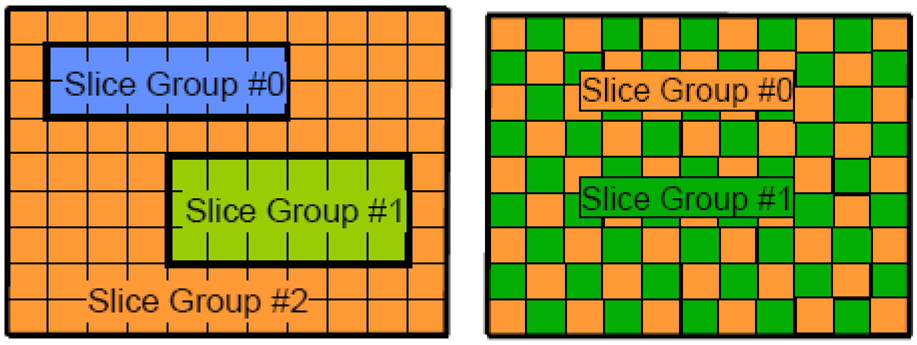


Figure 1.6: Subdivision of a QCIF frames into slices when using FMO [16]

Regardless of whether FMO is in use or not, each slice can be coded using different coding types as follows:

• I slice: A slice in which all macroblocks of the slice are coded using intra prediction.

• P slice: In addition to the coding types of I slice, some macroblocks of the P slice can also be coded using inter prediction with at most one motion compensated prediction signal per prediction block.

• B slice: In addition to the coding types available in I and P slices, some macroblocks of the B slice can also be coded using inter prediction with two motion compensated prediction signals per prediction block.

• SP slice: A so-called switching P slice that is coded such that efficient switching between different pre-coded pictures becomes possible.

• SI slice: A so-called switching I slice that allows an exact match of a macroblock in an SP slice for random access and error recovery purposes.

1.4 H.264 Codec

In common with earlier standards (such as MPEG-1, MPEG-2 and MPEG-4), the H.264 standard does not explicitly define a CODEC (encoder / decoder pair) [9]. Rather, the standard defines the syntax of an encoded video bit-stream together with the method of decoding this bit-stream. In practice, however, a compliant encoder and decoder are likely to include the functional elements shown in Figures 1.7 and 18. Whilst the functions shown in these figures are likely to be necessary for compliance, there is scope for considerable variation in the structure of the CODEC. The basic functional elements (prediction, transform, quantization, entropy encoding) are little different from previous standards (MPEG-1, MPEG-2, MPEG-4, H.261, H.263) [9]; the important changes in H.264 occur in the details of each functional element.

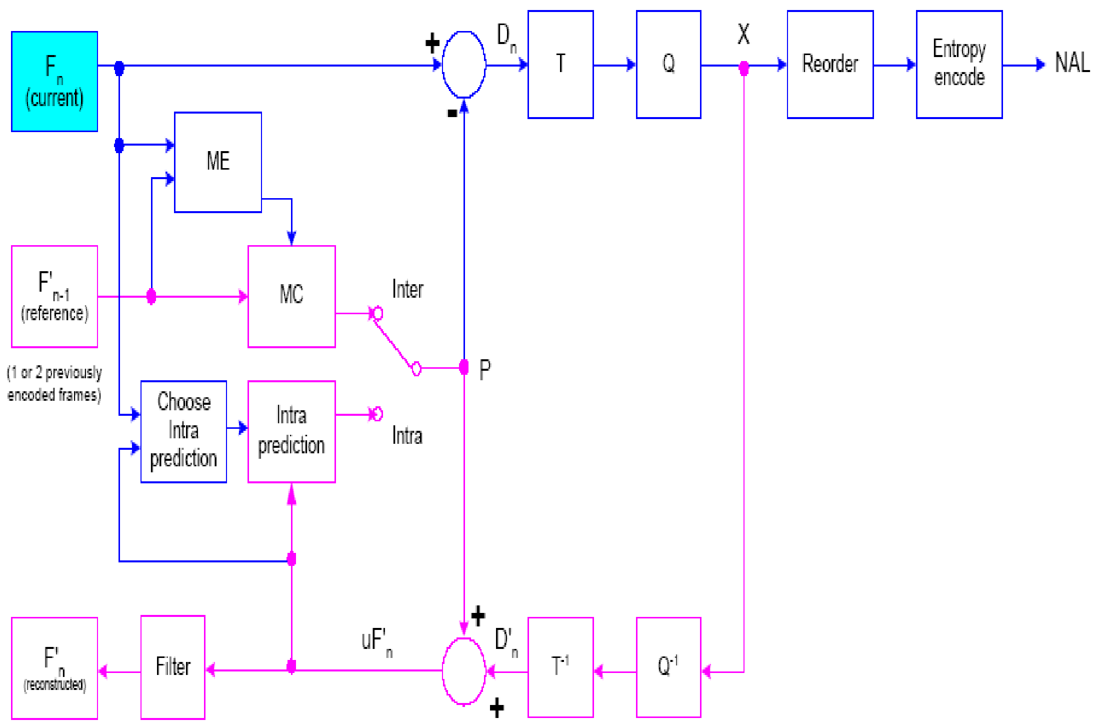


Figure 1.7: AVC Encoder [9]

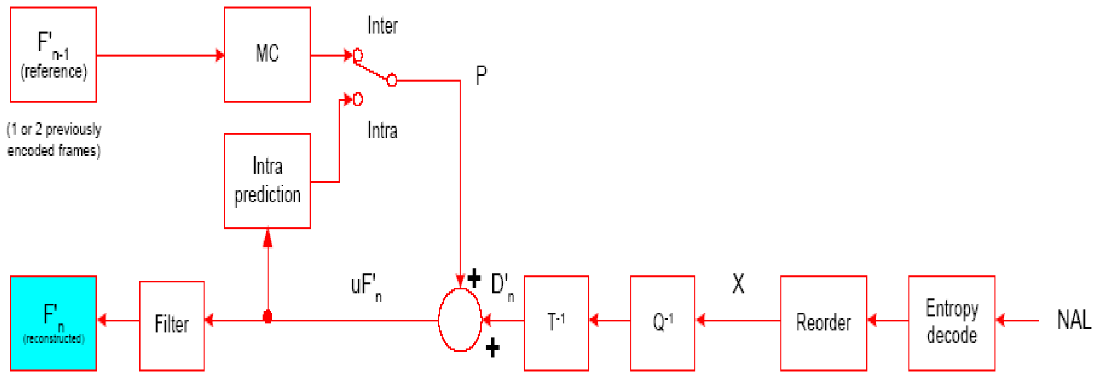


Figure 1.8: AVC Decoder [9]

The encoder (Figure 1.7) includes two dataflow paths, a “forward” path (left to right) and a “reconstruction” path (right to left). The dataflow path in the decoder (Figure 1.8) is shown from right to left to illustrate the similarities between encoder and decoder.

1.4.1 Encoder (forward path)

An input frame F_n is presented for encoding. The frame is processed in units of a macroblock (corresponding to 16x16 pixels in the original image). Each macroblock is encoded in intra or inter mode. In either case, a prediction macroblock P is formed based on a reconstructed frame. In Intra mode, P is formed from samples in the current frame n that have previously encoded, decoded and reconstructed (uF'_n in the Figure 1.7 and 1.8; note that the unfiltered samples are used to form P). In Inter mode, P is formed by motion-compensated prediction from one or more reference frame(s). In the Figures, the reference frame is shown as the previous encoded frame F'_n-1; however, the prediction for each macroblock may be formed from one or two past or future frames (in time order) that have already been encoded and reconstructed. The prediction P is subtracted from the current macroblock to produce a residual or difference macroblock D_n. This is transformed (using a block transform) and quantized to give X, a set of quantized transform coefficients. These coefficients are re-ordered and entropy encoded. The entropy encoded coefficients, together with side information required to decode the macroblock (such as the macroblock prediction mode, quantizer step size, motion vector information describing how the macroblock was motion-compensated, etc) form the compressed bitstream. This is passed to a network abstraction layer (NAL) for transmission or storage.

1.4.2 Encoder (reconstruction path)

The quantized macroblock coefficients X are decoded in order to reconstruct a frame for encoding of further macroblocks. The coefficients X are re-scaled (Q^{-1}) and inverse transformed (T^{-1}) to produce a difference macroblock D_n'. This is not identical to the original difference macroblock D_n; the quantization process introduces losses and so D_n' is a distorted version of D_n. The prediction macroblock P is added to D_n' to create a reconstructed macroblock uF'$_n$ (a distorted version of the original macroblock). A filter is applied to reduce the effects of blocking distortion and reconstructed reference frame is created from a series of macroblocks F'$_n$.

1.4.3 Decoder

The decoder receives a compressed bit-stream from the NAL. The data elements are entropy decoded and reordered to produce a set of quantized coefficients X. These are rescaled and inverse transformed to give D_n' (this is identical to the D_n' shown in the encoder). Using the header information decoded from the bit-stream, the decoder creates a prediction macroblock P, identical to the original prediction P formed in the encoder. P is added to D_n' to produce uF'$_n$ which this is filtered to create the decoded macroblock F'$_n$. It should be clear from the figures and from the discussion above that the purpose of the reconstruction path in the encoder is to ensure that both encoder and decoder use identical reference frames to create the prediction P. If this is not the case, then the predictions P in encoder and decoder will not be identical, leading to an increasing error or “drift” between the encoder and decoder.

1.5 Intra-frame Prediction

Each macroblock can be transmitted in one of several coding types depending on the slice-coding type. In all slice-coding types, the following types of intra coding are supported, which are denoted as Intra_4x4 or Intra_16x16 together with chroma prediction and I_PCM prediction modes. The Intra_4x4 mode is based on predicting each 4x4 luma block separately and is well suited for coding of parts of a picture with significant detail. The Intra_16x16 mode, on the other hand, does prediction of the whole 16x16 luma block and is more suited for coding very smooth areas of a picture. In addition to these two types of luma prediction, a separate chroma prediction is conducted. As an alternative to Intra_4x4 and Intra_16x16, the I_PCM coding type allows the encoder to simply bypass the prediction and transform coding processes and instead directly send the values of the encoded samples. The I_PCM mode serves the following purposes:

1) It allows the encoder to precisely represent the values of the samples

2) It provides a way to accurately represent the values of anomalous picture content without significant data expansion

3) It enables placing a hard limit on the number of bits a decoder must handle for a macroblock without harm to coding efficiency.

In contrast to some previous video coding standards [27] (namely H.263+ and MPEG-4 Visual), where intra prediction has been conducted in the transform domain intra prediction in H.264/AVC is always conducted in the spatial domain, by referring to neighboring samples of previously coded blocks which are to the left and/or above the block to be predicted. This may incur error propagation in environments with transmission errors that propagate due to motion compensation into inter-coded macroblocks. Therefore, a constrained intra-coding mode can be signaled that allows prediction only from intra-coded neighboring macroblocks. When using the Intra_4x4 mode, each 4x4 block is predicted from spatially neighboring samples as illustrated on the left-hand side of Figure 1.9. The 16 samples of the 4x4 block, which are labeled as a-p, are predicted using prior decoded samples in adjacent blocks labeled as A-Q. For each 4x4 block one of nine prediction modes can be utilized. In addition to "DC" prediction (where one value is used to predict the entire 4x4 block), eight directional prediction modes are specified as illustrated on the right-hand side of Figure 1.9. These modes are suitable to predict directional structures in a picture such as edges at various angles.

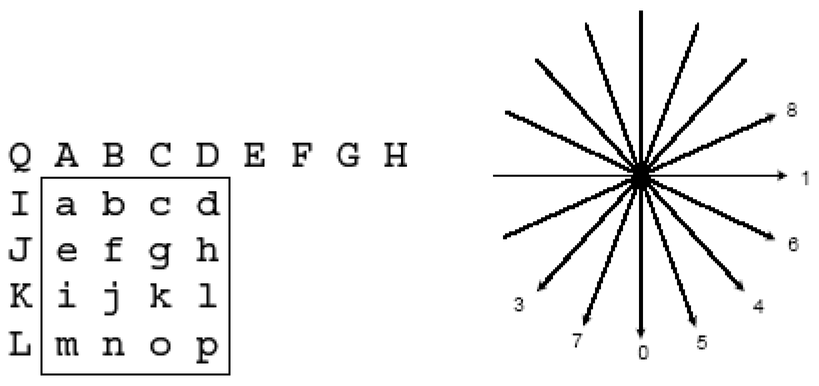


Figure 1.9: Left: Intra_4x4 prediction is conducted for samples a-p of a block using samples A-Q. Right: 8 "prediction directions" for Intra_4x4 prediction [16]

Figure 1.9 shows five of the nine Intra_4x4 prediction modes. For mode 0 (vertical prediction) the samples above the 4x4 block are copied into the block as indicated by the arrows. Mode 1 (horizontal prediction) operates in a manner similar to vertical prediction except that the samples to the left of the 4x4 block are copied. For mode 2 (DC prediction) the adjacent samples are averaged as indicated in Figure 1.9. The remaining 6 modes are diagonal prediction modes, which are called diagonal-down-left, diagonal-down-right, vertical-right, horizontal-down, vertical-left, and horizontal up prediction. As their names indicate, they are suited to predict textures with structures in the specified direction. The first two diagonal prediction modes are also illustrated in Figure 1.9. When samples E-H (see Figure 1.9) that are used for the diagonal-down-left prediction mode are not available (because they have not yet been decoded or they are outside of the slice or not in an intra-coded macroblock in the constrained intra-mode) these samples are replaced by sample D.

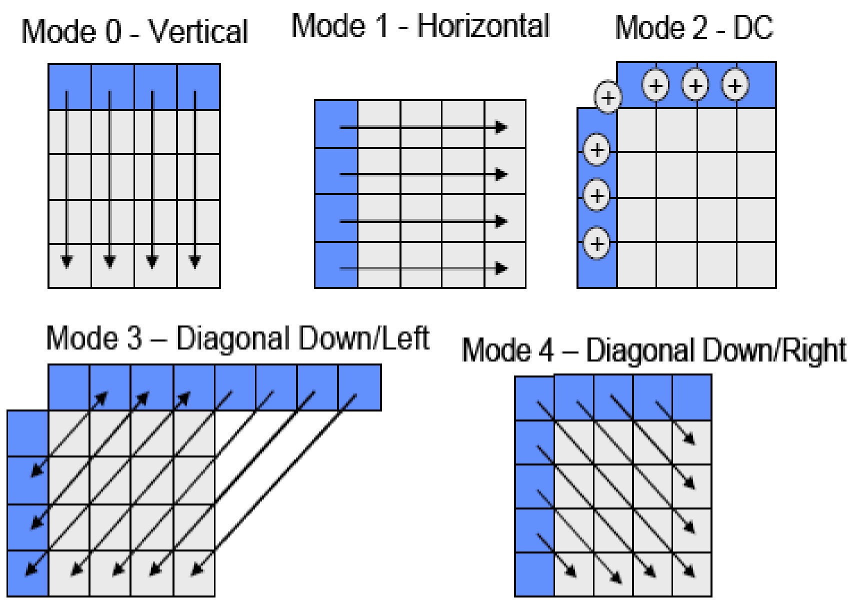


Figure 1.10: Five of the nine Intra_4x4 prediction modes [16]

When utilizing the Intra_16x16 mode, the whole luma component of a macroblock is predicted. Four prediction modes are supported. Prediction mode 0 (vertical prediction), mode 1 (horizontal prediction), and mode 2 (DC prediction) are specified similar to the modes in Intra_4x4 prediction except that instead of 4 neighbors on each side to predict a 4x4 block, 16 neighbors on each side to predict a 16x16 block are used. The chroma samples of a macroblock are predicted using a similar prediction technique as for the luma component in Intra_16x16 macroblocks, since chroma is usually smooth over large areas. Intra prediction (and all other forms of prediction) across slice boundaries is not used, in order to keep all slices independent of each other.

1.6 Inter-frame Prediction

1.6.1 Inter-frame Prediction in P Slices

In addition to the intra macroblock coding types, various predictive or motion-compensated coding types are specified as P macroblock types. Each P macroblock type corresponds to a specific partition of the macroblock into the block shapes used for motion-compensated prediction. Partitions with luma block sizes of 16x16, 16x8, 8x16, and 8x8 samples are supported by the syntax. In case partitions with 8x8 samples are chosen, one additional syntax element for each 8x8 partition is transmitted. This syntax element specifies whether the corresponding 8x8 partition is further partitioned into partitions of 8x4, 4x8, or 4x4 luma samples and corresponding chroma samples. Figure 1.11 illustrates the partitioning.

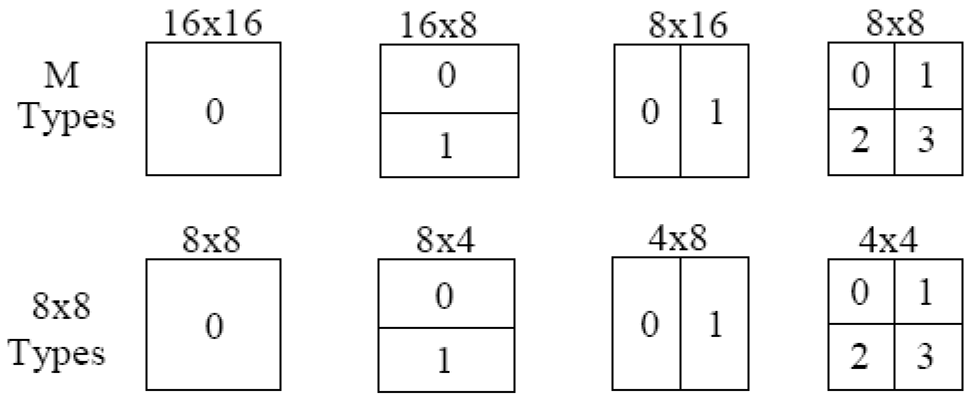


Figure 1.11: Segmentations of the macroblock for motion compensation. Top: segmentation of macroblocks, bottom: segmentation of 8x8 partitions. [16]

The prediction signal for each predictive-coded MxN luma block is obtained by displacing an area of the corresponding reference picture, which is specified by a translational motion vector and a picture reference index. Thus, if the macroblock is coded using four 8x8 partitions and each 8x8 partition is further split into four 4x4 partitions, a maximum of sixteen motion vectors may be transmitted for a single P macroblock. The accuracy of motion compensation is in units of one quarter of the distance between luma samples. In case the motion vector points to an integer-sample position, the prediction signal consists of the corresponding samples of the reference picture; otherwise the corresponding sample is obtained using interpolation to generate non-integer positions. The prediction values at half-sample positions are obtained by applying a one-dimensional 6-tap FIR filter, [1 -5 20 20 -5 1]/32, horizontally and vertically. Averaging samples at integer- and half-sample positions generates prediction values at quartersample positions.

Figure 1.12 illustrates the fractional sample interpolation for samples a-k and n-r. The samples at half sample positions labeled b and h are derived by first calculating intermediate values b_1 and h_1, respectively by applying the 6-tap filter as follows:

$$b_1 = (E - 5\,F + 20\,G + 20\,H - 5\,I + J) \qquad \ldots(1.1)$$

$$h_1 = (A - 5\,C + 20\,G + 20\,M - 5\,R + T\,) \qquad \ldots(1.2)$$

The final prediction values for locations b and h are obtained as follows and clipped to the range of 0 to 255.

$$b = (b_1 + 16\,) >> 5 \qquad \ldots(1.3)$$

$$h = (h_1 + 16\,) >> 5 \qquad \ldots(1.4)$$

The samples at half sample positions labeled as j are obtained by

$j_1 = cc - 5\,dd + 20\,h_1 + 20\,m_1 - 5\,ee + ff$, where intermediate values denoted as cc, dd, ee, m_1 and ff are obtained in a manner similar to h_1. The final prediction value j is then computed as $j = (j_1 + 512) >> 10$ and is clipped to the range of 0 to 255. The two alternative methods of obtaining the value of j illustrate that the filtering operation is truly separable for the generation of the half-sample positions.

The samples at quarter sample positions labeled as a, c, d, n, f, i, k, and q are derived by averaging with upward rounding of the two nearest samples at integer and half sample positions as, for example, by

$$a = (G + b + 1) >> 1 \qquad \ldots(1.5)$$

The samples at quarter sample positions labelled as e, g, p, and r are derived by averaging with upward rounding of the two nearest samples at half sample positions in the diagonal direction as, for example, by

$$e = (b + h + 1) >> 1 \qquad \ldots(1.6)$$

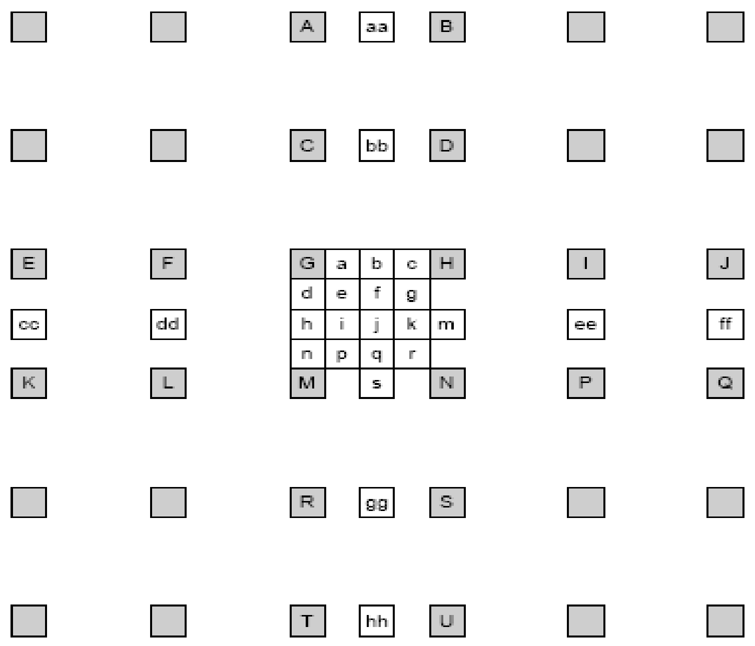


Figure 1.12: Filtering for fractional-sample accurate motion compensation (Upper-case letters indicate samples on the full-sample grid, while lower case samples indicate samples in between at fractional-sample positions) [16]

The prediction values for the chroma component are always obtained by bi-linear interpolation. Since the sampling grid of chroma has lower resolution than the sampling grid of the luma, the displacements used for chroma have one eighth sample position accuracy. The more accurate motion prediction using full sample, half sample and one-quarter sample prediction represent one of the major improvements of the present method compared to earlier standards for the following two reasons:

• The most obvious reason is more accurate motion representation

• The other reason is more flexibility in prediction filtering. Full sample, half sample and one-quarter sample prediction represent different degrees of low pass filtering which is chosen automatically in the motion estimation process. In this respect the 6-tap filter turns out to be a much better trade-off between necessary prediction loop filtering and has the ability to preserve high frequency content in the prediction loop.

A more detailed investigation of fractional sample accuracy is presented in [11]. The syntax allows so-called motion vectors over picture boundaries, i.e. motion vectors that point outside the image area. In this case, the reference frame is extrapolated beyond the image boundaries by repeating the edge samples before interpolation. The motion vector components are differentially coded using either median or directional prediction from neighboring blocks. No motion vector component prediction (or any other form of prediction) takes place across slice boundaries. The syntax supports multi-picture motion-compensated prediction [12] [13]. That is, more than one prior coded picture can be used as reference for motion-compensated prediction. Figure 1.13 illustrates the concept.

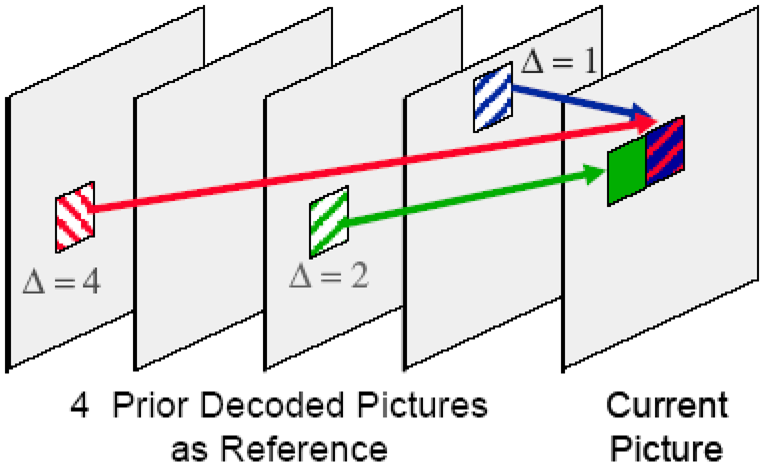


Figure 1.13: Multi-frame motion compensation[16].

In addition to the motion vector, also picture reference parameters (Δ) are transmitted. The concept is also extended to B slices as described below.

Multi-frame motion-compensated prediction requires both encoder and decoder to store the reference pictures used for inter prediction in a multi-picture buffer. The decoder replicates the multi-picture buffer of the encoder according to memory management control operations specified in the bit-stream. Unless the size of the multi-picture buffer is set to one picture, the index at which the reference picture is located inside the multi-picture buffer has to be signaled. The reference index parameter is transmitted for each motion-compensated 16x16, 16x8, 8x16, or 8x8 luma block. Motion compensation for smaller regions than 8x8 uses the same reference index for prediction of all blocks within the 8x8 region. In addition to the motion-compensated macroblock modes described above, a P macroblock can also be coded in the so-called P_Skip type. For this coding type, neither a quantized prediction error signal, nor a motion vector or reference index parameter is transmitted. The reconstructed signal is obtained similar to the prediction signal of a P_16x16 macroblock type that references the picture which is located at index 0 in the multi-picture buffer. The motion vector used for reconstructing the P_Skip macroblock is similar to the motion vector predictor for the 16x16 block. The useful effect of this definition of the P_Skip coding type is that large areas with no change or constant motion like slow panning can be represented with very few bits.

1.7 Transform, Scaling, and Quantization

Similar to previous video coding standards, H.264/AVC utilizes transform coding of the prediction residual. However, in H.264/AVC, the transformation is applied to 4x4 blocks, and instead of a 4x4 discrete cosine transform (DCT), a separable integer transform with similar properties as a 4x4 DCT is used. The transform matrix is given as

$$H = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 2 & 1 & -1 & -2 \\ 1 & -1 & -1 & 1 \\ 1 & -2 & 2 & -1 \end{bmatrix}$$

Since the inverse transform is defined by exact integer operations, inverse-transform mismatches are avoided. At the encoder the process includes a forward transform, zig-zag scanning, scaling and rounding as the quantization process followed by entropy coding. At the decoder, the inverse of the encoding process is performed except for the rounding. More details on the specific aspects of the transform in H.264/AVC can be found in [14].

An additional 2x2 transform is also applied to the DC coefficients of the four 4x4 blocks of each chroma component. The procedure for a chroma block is illustrated in Fig. 1.14. The small blocks inside the larger blocks represent DC coefficients of each of the four 4x4 chroma blocks of a chroma component of a macroblock numbered as 0,1,2, and 3. The two indices correspond to the indices of the 2x2 inverse Hadamard transform.

$$H = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

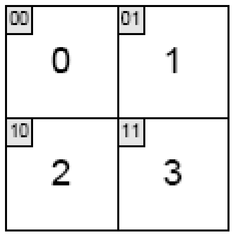


Figure 1.14: Repeated transform for chroma blocks. The four blocks numbered 0 to 3 indicate the four-chroma blocks of a chroma component of a macroblock.

A quantization parameter is used for determining the quantization of transform coefficients in H.264/AVC. The parameter can take 52 values. Theses values are arranged so that an increase of 1 in quantization parameter means an increase of quantization step size by approximately 12% (an increase of 6 means an increase of quantization step size by exactly a factor of 2). It can be noticed that a change of step size by approximately 12% also means roughly a reduction of bit rate by approximately 12%. The quantized transform coefficients of a block generally are scanned in a zigzag fashion and transmitted using entropy coding methods. The 2x2 DC coefficients of the chroma component are scanned in raster-scan order. All inverse transform operations in H.264/AVC can be implemented using only additions and bit-shifting operations of 16-bit integer values.

1.8 Entropy Coding

In H.264/AVC, two methods of entropy coding are supported. For transmitting the quantized transform coefficients a more efficient method called context-adaptive variable length coding (CAVLC) is employed. In this scheme, VLC tables for various syntax elements are switched depending on already transmitted syntax elements. Since the VLC tables are designed to match the corresponding conditioned statistics, the entropy coding performance is improved in comparison to schemes using a single VLC table.

The efficiency of entropy coding can be improved further if the context-adaptive binary arithmetic coding (CABAC) is used. On the one hand, the usage of arithmetic coding allows the assignment of a non-integer number of bits to each symbol of an alphabet, which is extremely beneficial for symbol probabilities that are greater than 0.5. On the other hand, the usage of adaptive codes permits adaptation to non-stationary symbol statistics. Another important property of CABAC is its context modeling.

1.9 In-Loop Deblocking Filter

One particular characteristic of block-based coding is the accidental production of visible block structures. Block edges are typically reconstructed with less accuracy than interior pixels and “blocking” is generally considered to be one of the most visible artifacts with the present compression methods. For this reason, H.264/AVC defines an adaptive in-loop deblocking filter [15], where the strength of filtering is controlled by the values of several syntax elements

Figure 1.15 illustrates the principle of the deblocking filter using a visualization of a one-dimensional edge. Whether the samples p0 and q0 as well as p1 and q1 are filtered is determined using quantization parameter (QP) dependent thresholds $\alpha(QP)$ and $\beta(QP)$. Thus, filtering of p0 and q0 only takes place if each of the following conditions is satisfied:

1. $| p0 - q0 | < \alpha(QP)$
2. $| p1 - p0 | < \beta(QP)$
3. $| q1 - q0 | < \beta(QP)$

where the $\beta(QP)$ is considerably smaller than $\alpha(QP)$.

Accordingly, filtering of p1 or q1 takes place if the corresponding condition below is satisfied:

$| p2 - p0 | < \beta(QP)$ or $| q2 - q0 | < \beta(QP)$

The basic idea is that if a relatively large absolute difference between samples near a block edge is measured, it is quite likely a blocking artifact and should therefore be reduced. However, if the magnitude of that difference is so large that it cannot be explained by the coarseness of the quantization used in the encoding, the edge is more likely to reflect the actual behavior of the source picture and should not be smoothed over.

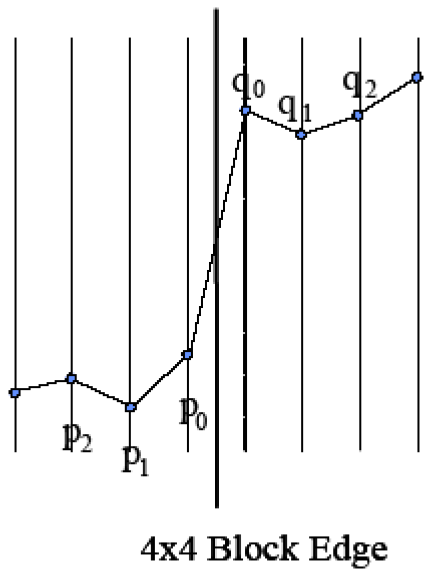


Figure 1.15: Principle of deblocking filter. [9]

The blockiness is reduced, while the sharpness of the content is basically unchanged. Consequently, the subjective quality is significantly improved. The filter reduces bit rate typically by 5-10% while producing the same objective quality as the non-filtered video. Figure 1.16 illustrates the performance of the deblocking filter.

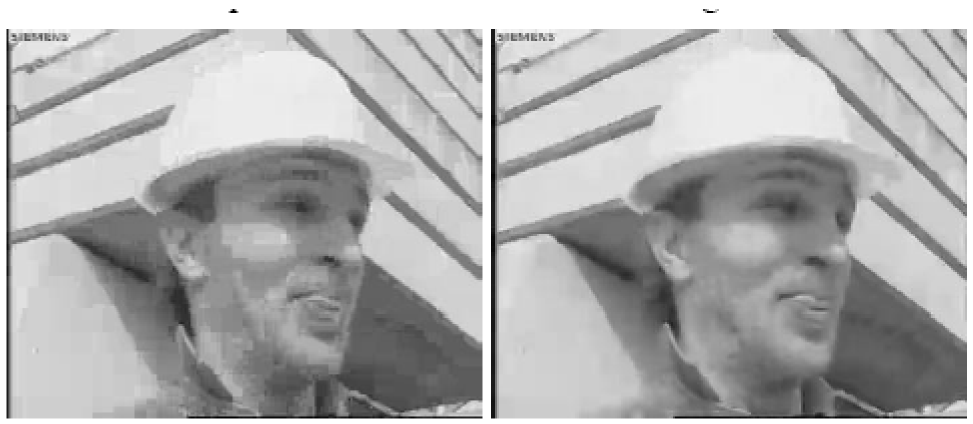


Figure 1.16: Performance of the deblocking filter for highly compressed pictures. Left: without deblocking filter, right: with deblocking filter. [16]

CHAPTER 2

ARM9TDMI

The Advanced RISC Processor (ARM) core is a key component of many successful 32-bit embedded systems. ARM cores are widely used in mobile phones, handheld organizers, and a multitude of other everyday portable consumer devices.

2.1 About the ARM9TDMI

The ARM9TDMI is a member of the ARM family of general purpose microprocessors. The ARM9TDMI is targeted at embedded control applications where high performance, low die size and low power are all important. The ARM9TDMI supports both 32-bit ARM and 16-bit Thumb instruction sets, allowing the user to trade off between high performance and high code density. The ARM9TDMI supports the ARM debug architecture and includes logic to assist in both hardware and software debug. The ARM9TDMI supports both bidirectional and unidirectional connection to external memory systems. The ARM9TDMI also includes support for coprocessors. But this thesis includes performance analysis of the core ARM9TDMI only.

The ARM9TDMI processor core is implemented using five-stage pipeline consisting of fetch, decode, execute, memory and write stages. The device has Harvard architecture, and the simple bus interface eases connection to either a cached or static random access memory (SRAM) based memory system. A simple handshake protocol is provided for coprocessor support.

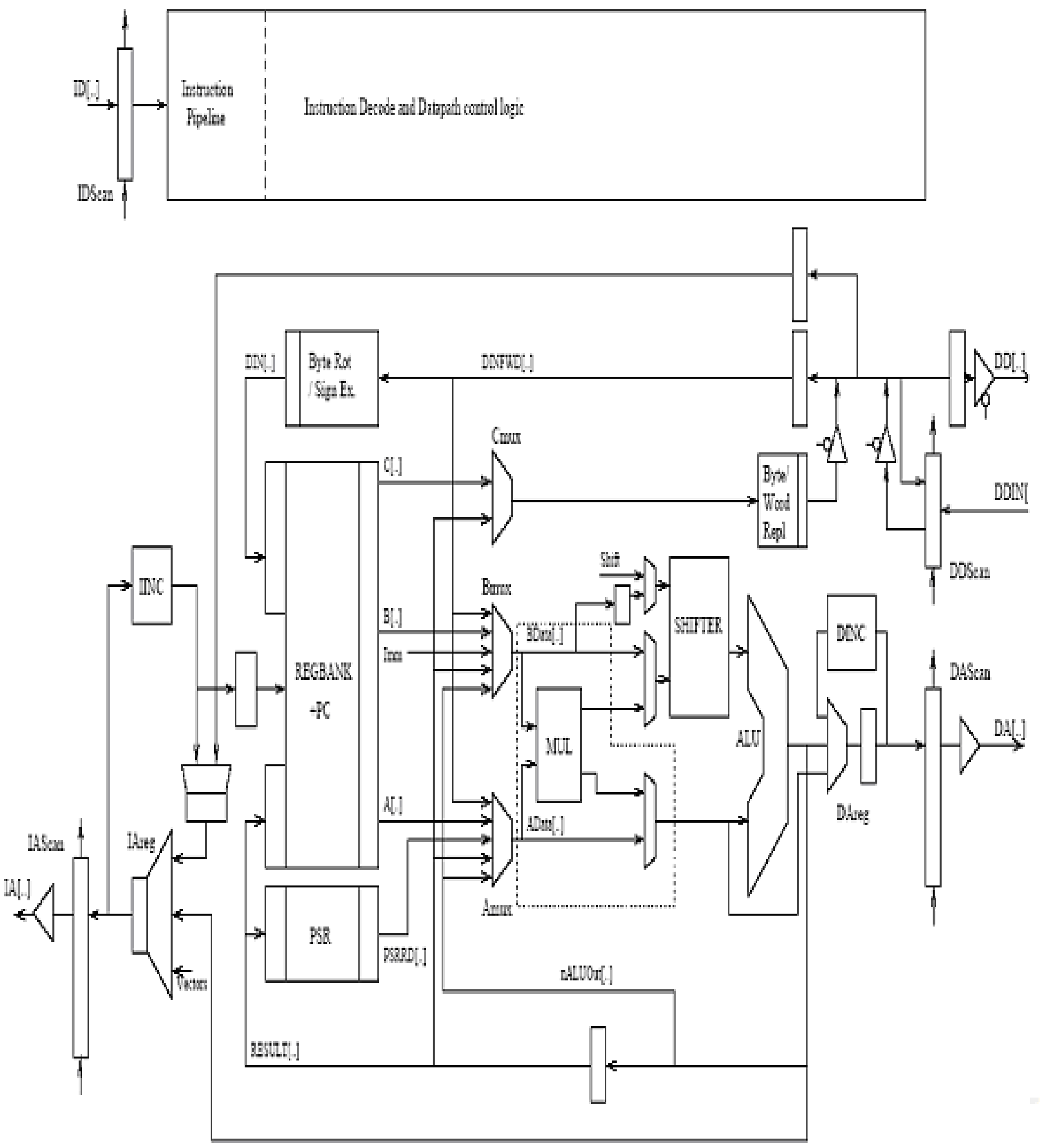


Figure 2.1: Processor Block Diagram [37]

2.2 Programmer's Model

2.2.1 Hardware Fundamentals

The ARM9TDMI processor core implements ARM architecture v4T, and so executes the ARM 32-bit instruction set and the compressed Thumb 16-bit instruction set. The ARM processor can be abstracted into eight components – Arithmetic and Logic Unit (ALU), barrel shifter, Multiply Accumulate Unit (MAC), register file, instruction decoder, address register, incremented, and sign extend. The register file contains 37 registers, but only 17 or 18 registers are accessible at any point in time; the rest are banked according to processor mode.

2.2.2 Instruction set extension spaces

ARM architecture v4 and v4T introduced a number of instruction set extension spaces to the ARM instruction set. These are:

- Arithmetic instruction extension space
- Control instruction extension space
- Coprocessor instruction extension space
- Load/store instruction extension space

For more information on instruction set refer [18].

2.2.3 Pipeline implementation and interlocks

The ARM9TDMI implementation uses a five-stage pipeline design. These five stages are

- Instruction fetch (F)

- Instruction decode (D)
- Execute (E)
- Data memory access (M)
- Register write (W)

ARM implementation is fully interlocked, so that software will function identically across different implementations without concern for pipeline effects. Interlocks do affect instruction execution times. For example, the following example suffers a single cycle penalty due to a load-use interlock on register RO:

LDR RO, [R7]

ADD R5, R0, R1

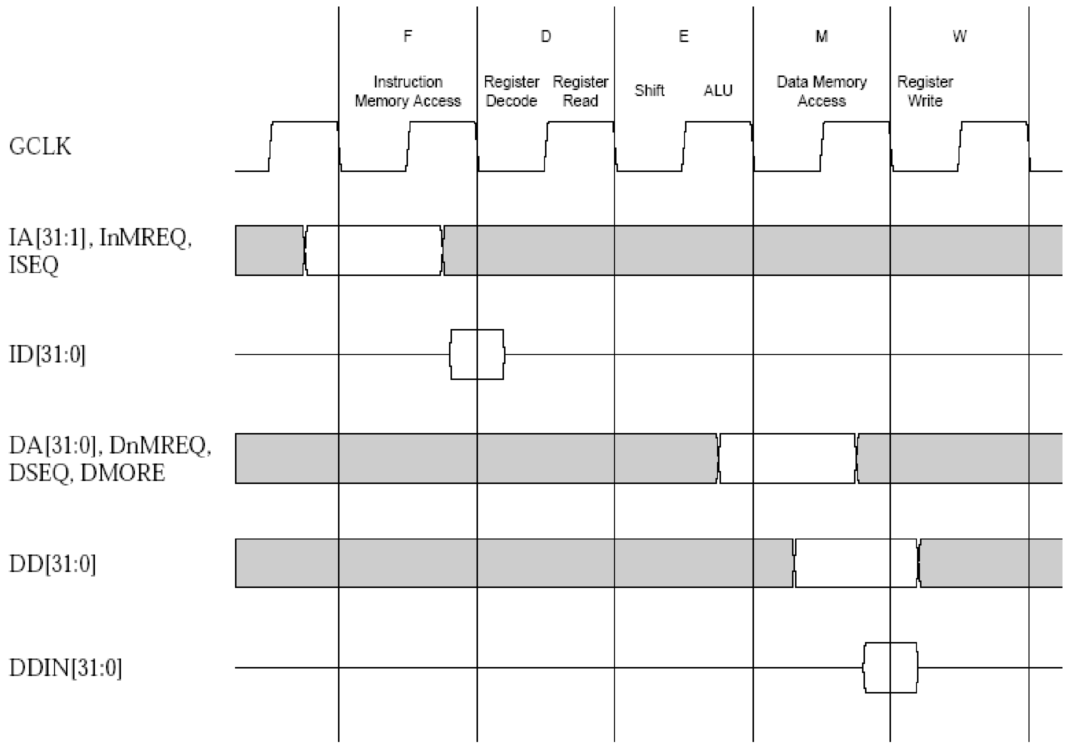


Figure 2.2:ARM9TDMI processor core instruction pipeline [37].

2.2.4 Memory Interface

The ARM9TDMI has Harvard bus architecture with separate instruction and data interfaces. This allows concurrent instruction and data accesses, and greatly reduces the CPI of the processor. For optimal performance, single cycle memory accesses for both interfaces are required, although the core can be wait-stated for non-sequential accesses, or slower memory systems.

For both instruction and data interfaces, the ARM9TDMI process core uses pipelined addressing. The address and control signals are generated the cycle before the data transfer takes place, giving any decode logic as much advance notice as possible. All memory accesses are generated from GCLK. For each interface there are different types of memory access:

- Non-sequential
- Sequential
- Internal
- Coprocessor transfer

CHAPTER 3

DESIGNING AND OPTIMIZING FOR ARM9TDMI

ARM9TDMI processor is mainly targeted for mobile devices and handheld organizers. These devices have constraints like low power consumption, less memory and so on. In order to achieve real time decoding, considering the complexity of the H.264 video coding standard, optimizing the software to run efficiently on the target processor is very important. Optimizing code takes time reduces source code readability. Usually, it is worth optimizing functions that are frequently executed and important for performance. Performance profiling tool, found in ARM simulator ARM Developer Suite v1.2 [37] is used to find these frequently executed functions. To write efficient C code it is important to be aware of areas where the C compiler has to be conservative, the limits of the processor architecture the C compiler is mapping to, and the limits of specific C compiler. FastVDO's [39] H.264 decoder for ARM9TDMI is tested on *armcc* from ARM Developer suite version 1.2 (ADS1.2) [44]. This compiler or a later version, can be licensed directly from ARM [37].

3.1 Optimization Techniques

3.1.1 Basic C Data Types

ARM processor has 32-bit registers and 32-bit data processing operations. The ARM architecture is RISC load/store architecture. In other words the data has to be loaded from memory to registers before acting on them. There is no arithmetic or logical instructions that manipulate values in memory directly. Compiler *armcc* use the data-type mappings in Table 3.1 for an ARM target. The exceptional case for type *char* is worth noting as it can cause problems in porting the code from processor architecture. A common example is using a char type variable "j" as a loop counter, with loop continuation condition j>=0. As j is unsigned for the ARM compilers, the loop will never terminate.

Table 3.1: C compiler data type mappings

C Data Type	Implementation
Char	Unsigned 8-bit byte
Short	Signed 16-bit half word
Int	Signed 32-bit word
Long	Signed 32-bit word
Long long	Signed 64-bit double word

- Local Variable Type: ARMv4-based processors can efficiently load and store 8, 16, and 32- bit data. However most ARM data processing operations are 32-bit only. For this reason, use of 32-bit data type, int or long, for local variables wherever possible is suggested. Avoid using short or char as local variables, even if manipulation is performed on 8 or 16-bit data.

- Function arguments type: As local variables, converting function arguments from type char or short to type int increases performance and reduces code size.
- For array entries and global variables held in main memory, use the type with the smallest size possible to hold the required data. This saves memory footprint.

3.1.2 Conditional Execution

All ARM instructions are conditional. Each instruction contains a 4-bit field which is a condition code; the instruction is only executed if the ARM flag bits indicate that the specified condition is true. Typically a conditionally executing code sequence starts with a compare instruction setting the flags, followed by a few conditionally executed instructions.

For example:

```
CMP x, #0
MOVGE y, #1
MOVLT y, #0
```

This saves two branch instructions and on average 2.5 ARM9 cycles [17].

Conditional execution reduces the number of branch instructions, and therefore improves code-size and performance. However, when more than about four instructions are conditional, performance could be worse in some cases (since branches take three cycles or less on ARM). The compiler therefore limits the number of conditionally executed instructions.

It is therefore beneficial to keep the bodies of if and else statements as simple as possible, so that they can be conditionalized. Relational expressions should be grouped into blocks of similar conditions.

The following example shows how the compiler uses conditional execution:

```
int g(int a, int b, int c, int d)
{ if (a > 0 && b > 0 && c < 0 && d < 0) /* grouped conditions */
return a + b + c + d;
return -1;
}
```

The above code snippet in assembly, compiled by the ADS1.2 compiler, is as shown below.

```
g
        CMP a1,#0
        CMPGT a2,#0
BLE |L000024.J4.g|
        CMP a3,#0
        CMPLT a4,#0
        ADDLT a1,a1,a2
        ADDLT a1,a1,a3
        ADDLT a1,a1,a4
        MOVLT pc,lr
|L000024.J4.g|
        MVN a1,#0
        MOV pc,lr
```

Because the conditions were grouped, the compiler was able to conditionalize them.

3.1.3 Comparison with zero

The ARM flags are set after a compare (CMP) instruction. The flags can also be set by other operations, such as MOV, ADD, AND, MUL, which are the basic arithmetic and logical instructions (the data-processing instructions). If a data-processing instruction sets the flags, the N and Z flags are set the same way as if the result was compared with zero. The N flag indicates whether the result is negative, the Z flag indicates that the result is zero. For example:

```
ADD R0, R0, R1
CMP R0, #0
```

This produces identical N and Z flags as:

```
ADDS R0, R0, R1
```

The N and Z flags on the ARM correspond to the signed relational operators x < 0, x >= 0, x == 0, x != 0, and unsigned x == 0, x != 0 (or x > 0) in C.

Each time a relational operator is used in C, the compiler emits a compare instruction. If the operator is one of the above, the compiler can remove the compare if a data processing operation preceded the compare. For example:

```
int g(int x, int y)
{ if (x + y < 0)
return 1;
else
return 0;
}
```

The compiler output for the above code snippet will be:

```
g
ADDS a1,a1,a2
MOVPL a1,#0
MOVMI a1,#1
MOV pc,lr
```

If possible, arrange for critical routines to test the above conditions. This often allows you to save compares in critical loops, leading to reduced code size and increased performance.

3.1.4 Loops

Loops are a common construct in most programs; a significant amount of the execution time is often spent in loops. It is therefore worthwhile to pay attention to time-critical loops.

The loop termination condition can cause significant overhead if written without caution. Count-down-to-zero loops and simple termination conditions should be used.

Take the following two sample routines, which calculate n!. The first implementation uses an incrementing loop, the second a decrementing loop.

```
int fact1 (int n)
```

```
{ int i, fact = 1;
for (i = 1; i <= n; i++)
fact *= i;
return (fact);
}
int fact2 (int n)
{ int i, fact = 1;
for (i = n; i != 0; i--)
fact *= i;
return (fact);
}
```

The following code is produced:

```
fact1
        MOV a3,#1
        MOV a2,#1
        CMP a1,#1
        BLT    |L000020.J5.fact1|
|L000010.J4.fact1|
        MUL a3,a2,a3
        ADD a2,a2,#1
        CMP a2,a1
        BLE    |L000010.J4.fact1|
|L000020.J5.fact1|
        MOV a1,a3
        MOV pc,lr
fact2
        MOVS a2,a1
        MOV a1,#1
        MOVEQ pc,lr
|L000034.J4.fact2|
        MUL a1,a2,a1
        SUBS a2,a2,#1
        BNE    |L000034.J4.fact2|
        MOV pc,lr
```

The difference between fact1 and fact2 is that the original ADD/CMP instruction pair is replaced by a single SUBS instruction. This is because a compare with zero could be optimized away, as described in section 3.1.3.

In addition to saving an instruction in the loop, the variable n does not need to be saved across the loop, so a register is also saved. This eases register allocation, and leads to more efficient code elsewhere in the function (two more instructions saved).

This technique of initializing the loop counter to the number of iterations required, and then decrementing down to zero, also applies to while and do statements. Small loops can be unrolled for higher performance, with the disadvantage of increased codesize. When a loop is unrolled, a loop counter needs to be updated less often and fewer branches are executed. If the loop iterates only a few times, it can be fully unrolled, so that the loop overhead completely disappears. The ARM compilers currently do not unroll loops automatically, so any unrolling should be done in the source code.

3.1.5 Register Allocation

The most important optimization supported by the ARM compilers is called register allocation. This is a process where the compiler allocates variables to ARM registers, rather than to memory. This has the advantage that those variables can be accessed quickly whenever needed, without needing instructions to transfer them from/to memory. As a result of register allocation, most variables are kept in registers, resulting in dramatic improvement in codesize and performance. Code can be written which enables the compiler to achieve a more optimal register allocation.

All basic integer, pointer and floating-point types can be allocated to registers. Fields of structures and complete structures can also be kept in registers. The following rules define when a variable is considered for allocation in a register:

A variable may be allocated to a register if:

· it is a local variable or a function parameter, and

· its address is never taken, or its address is taken, but not assigned to another variable.

A field in a structure may be allocated to a register if:

· it is declared locally or a function parameter, and

· the structure is not assigned directly with the result of a function call, and

· neither the address of the structure nor any of its fields is taken, or if any of these addresses is taken, it is not assigned to another variable.

Pointers are a powerful part of the C language. However, they must be used carefully or poor code may result. If the address of a variable is taken, the compiler must assume that the variable can be changed by any assignment through a pointer or by any function call, making it impossible to put it into a register. This is also true for global variables, as they might have their address taken in some other function. This problem is known as pointer aliasing, because the pointer is known as an alias of the variable it points to.

CHAPTER 4

H.264/AVC BASELINE PROFILE DECODER COMPLEXITY ANALYSIS

This chapter briefs about the analysis of the computational complexity of a software-based H.264/AVC baseline profile decoder [39]. The analysis is based on determining the number of CPU cycles required by the H.264 baseline decoder to perform the key decoding sub-functions. The bit-streams for the test are generated using JM9.7 encoder for a variety of content and bit rates. Previous studies indicate that an H.264/AVC baseline decoder is approximately 2.5 times more time complex than an H.263 baseline decoder [16].

4.1 Introduction

To estimate the computational complexity of an H.264/AVC baseline decoder implementation, it is important to understand its two major components: time complexity and space (or storage) complexity. Time complexity is measured by the number of CPU cycles required to execute a specific implementation of an algorithm. A clever implementation of a given algorithm may have significantly lower time complexity than a less clever one for a given hardware platform despite the fact that the two implementations produce identical results. Storage complexity is measured by the approximate amount of memory required to implement an algorithm. It is a function of the algorithm, though it may be affected somewhat by implementation design specifics. The cost of hardware for a given application is a function of both time and storage complexities as both affect the size of the underlying circuits. In this thesis, the computational complexity of the FastVDO's H.264/AVC baseline decoder [39] is studied and analyzed in the context of a software implementation on ARM9TDMI processor which is the core processor of most of the handheld devices.

4.2 Experiment Setup

As mentioned earlier, H.264/AVC contains a number of optional coding features that can have a significant effect on decoder time complexity. To develop experimental results, bit-streams were generated using public JM9.7 [30] H.264/AVC baseline encoder, which performs an exhaustive search of all possible motion vectors in a search range around the median predicted value and coding modes in order to achieve near-optimal rate-distortion performance.

To generate the bit-streams, a set of six QCIF-resolution video sequences from FastVDO LLC are used. The QCIF video sequences are shown in the Fig.4.1. Each video sequence was encoded by the encoder at different bit rates of 18, 24, 48, and 64Kbps, with and without deblocking filter. The selected rates represent typical video data rates used in videoconferencing systems.

Table 4.1: Experiment Setup

Platform	ARM9TDMI processor
H.264 Decoder	FastVDO LLC Baseline High Profile H.264 Decoder©
Video Sequences	Girl, Golf, Karate, Plane, Foreman, Carphone (Fig. 4.2)
Encoder	JM 9.7 [30]
Profiler	ADS1.2 ARM simulator's Performance Analyzer

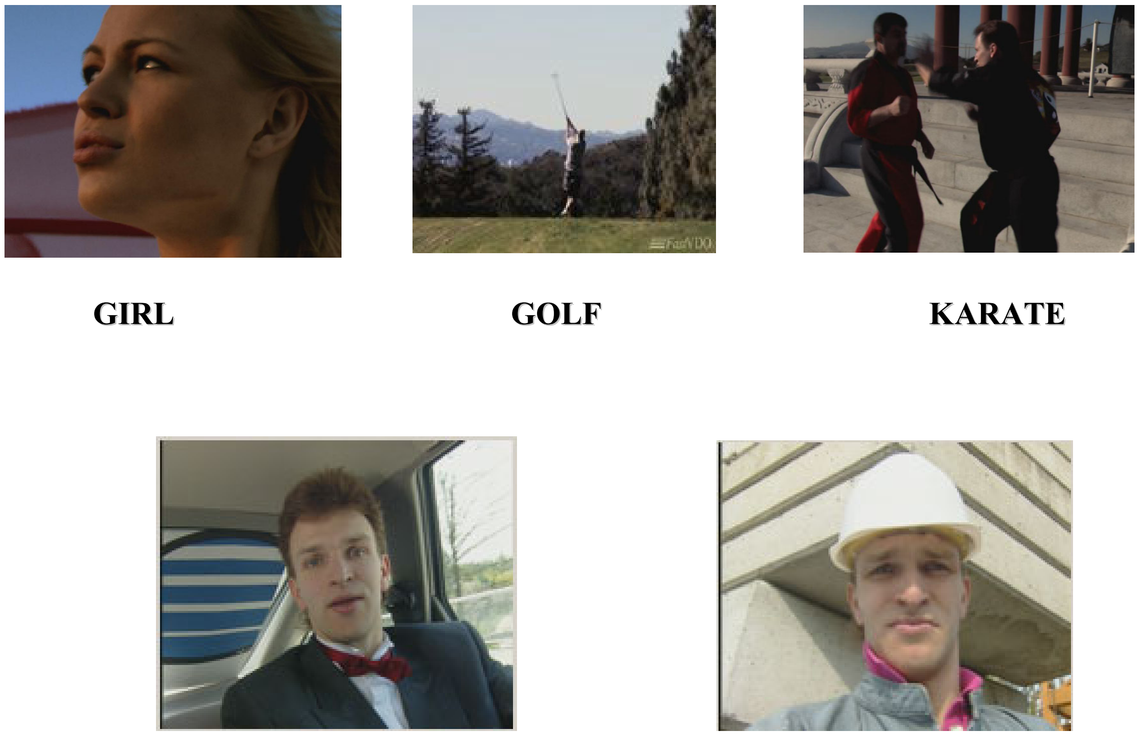


Figure 4.1: Video sequences used for performance analysis [39]

4.3 Experimental Analysis

The first stage in any optimization process is to identify the critical routines and measure their current performance. The "hotspots" in the application code are sections of code which utilize maximum of processor time out of the total time taken by the application to complete one execution. Application profiler is a tool that measures the proportion of time or processing cycles spent in each subroutine. For the performance analysis of FastVDO LLC's H.264 Baseline profile decoder, the profiler used is ADSv1.2 ARM simulator's performance analyzer [37]. This performance analyzer is a powerful software-profiling tool. It helps in understanding the performance characteristics of a software application at all levels. The analyzer gives results in two formats:-

- Flat Sampling: It can be time based and event based
- Call Graph: Flow of control in the application

4.3.1 Results of Performance analysis of H.264 Decoder

Results of time-based sampling of H.264 decoder using ADSv1.2 ARM simulator's performance analyzer are listed in Table 4-2. Figure 4.2 shows the pie chart, which clearly indicates the functions where most of the execution time is spent for "Foreman" QCIF sequence. The results indicate that the most time consuming computation modules in the H.264 are motion compensation, deblocking filter and IDCT 4x4 (Inverse Discrete Cosine Transform). Therefore, these modules are selected as the target for optimization.

Table 4.2: Execution time in percentage of the total CPU time

Video Sequences	Deblocking Filter	Motion Compensation	4x4 IDCT	Others
Foreman	37.83 %	19.83%	10.08%	32.36%
Carphone	34.97%	18.82%	10.65%	35.56%
Girl	38.92%	14.25%	11.84%	34.99%
Karate	35.71%	20.03%	10.3%	33.96%

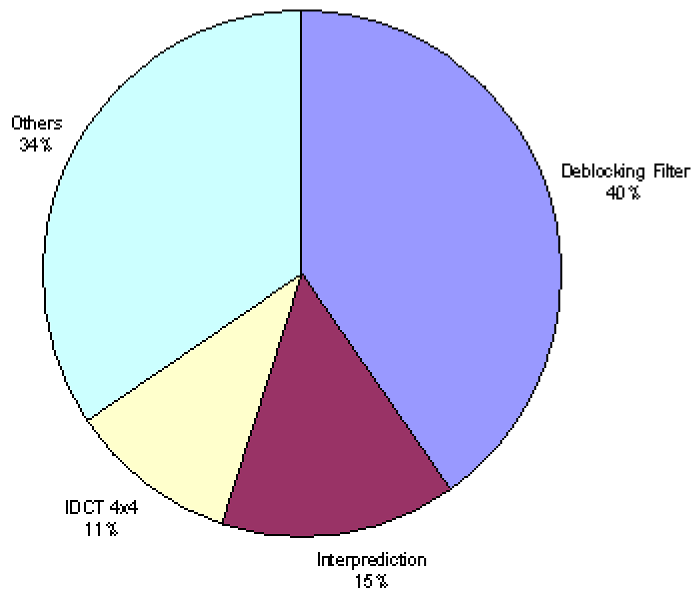


Figure 4.2: Percentage of the total execution time spent in major subroutines for Foreman QCIF sequence.

4.3.2 Optimization of IDCT

Figure 4.2 shows flow-graph of the inverse transforms which are applied to rows and columns of each 4x4 block. The steps of implementation [40] of the 1-D inverse transform are listed as follows: -

ei0 = di0 + di2, with i = 0..3

ei1 = di0 – di2, with i = 0..3

ei2 = (di1 >> 1) – di3, with i = 0..3

ei3 = di1 + (di3 >> 1), with i = 0..3

fi0 = ei0 + ei3, with i = 0..3

fi1 = ei1 + ei2, with i = 0..3

fi2 = ei1 – ei2, with i = 0..3

fi3 = ei0 – ei3, with i = 0..3

where, eii is the input to the transformation process. The resulting 4x4 matrix obtained after 1-D IDCT is then transposed and 1-D IDCT is applied to its results to get the IDCT 4x4 module. The final coefficients are obtained by adding 32 and dividing (right shifting) by 64 for the purpose of rounding.

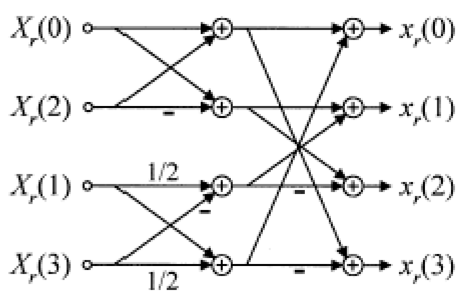


Figure 4.3: Fast implementation of the H.264 inverse transform. No multiplications are needed, only additions and shifts [14].

Since the entire 4x4 block needs to be loaded back from the memory into the registers to perform IDCT and final coefficients have to be stored back in the memory, IDCT is memory intensive routine. ARM9TDMI supports multiple load and store instructions. Sequential multiple load/store instruction takes N cycles, where N is the number of registers to be loaded, as compared to other processor like ARM7TDMI takes 2+Nt cycles [37], where t is the number of cycles required for each sequential access to memory. This is one of the advantages of ARM9 processor along with the preprocessing shift using barrel shifter. To take advantage of the 32 bit ARM registers we can load four pixels at a time. This boils down to just 4 cycles instead of 16 cycles to load the entire 4x4 block from the memory to the registers. Also if byte aligned load/store is used, the value will be available for further processing only after two cycles. If the data has to be processed right after a byte aligned load, the processor waits for two cycles and thus resulting in stalls. This is termed as interlocks [17]. This can be avoided if multiple load instructions are used. The division by 64 is done by right shifting five bits using barrel shifter.

Since all the pixels are manipulated in the same way, we can pack 2 pixels into a single 32-bit register. This is called single instruction multiple data (SIMD) processing [17]. To process multiple pixels at a time, four pixels are loaded at once using a word load as in Fig.4.3.

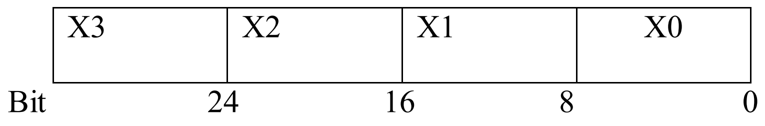


Figure 4.4: Four pixels packed in 32-bit register

Then the 8-bit data is unpacked and promoted to 16-bit data using an AND with a mask register as in Fig.4.4

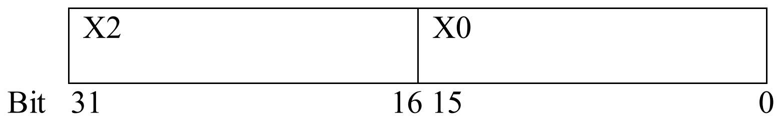


Figure 4.5: Two pixels packed in 32-bit register

Note that even for signed values [a, b] + [c, d] = [a + b, c + d], if [a, b] is interpreted using the mathematical equation $a2^{16} + b$. Therefore SIMD operations can be performed on these values using normal arithmetic instructions [16].

As described above, IDCT is implemented on two pixels at a time. Appendix B includes the steps of implementation. It is shown that, by the proposed optimization technique, 32 cycles are reduced per each 4x4 IDCT block, excluding the interlocks.

The table 4.3 shows the CPU cycles taken by the IDCT routine in the decoding of the test sequences. Figure.4.4 shows the speedup of the IDCT sub-function in the decoding process of several test streams.

Table 4.3: ARM9TDMI mega cycles/frame reduction before and after IDCT optimization

Sequences	Before Optimization (Mega Cycles/ Frame)	After Optimization (Mega Cycles/ Frame)
Foreman	0.634	0.347
Carphone	0.648	0.305
Girl	0.642	0.298
Karate	0.542	0.317

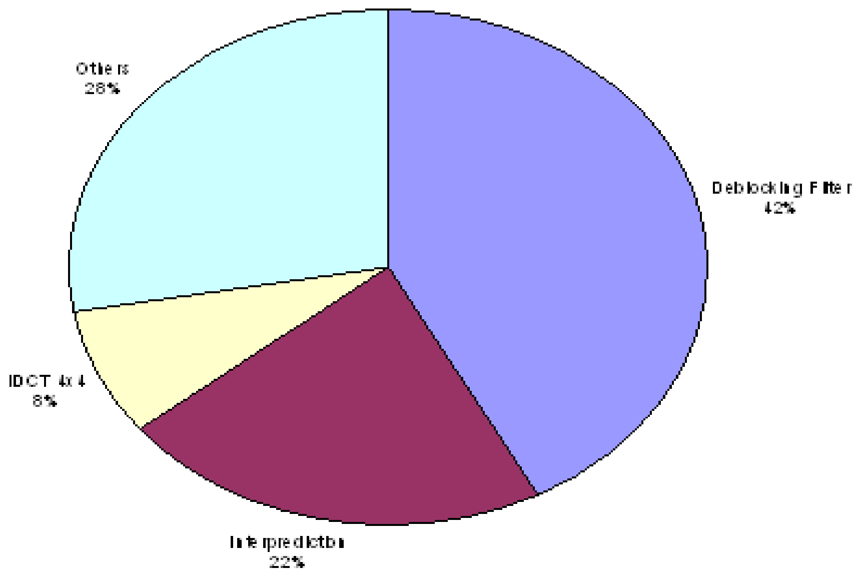


Figure 4.3: Chart showing percentage execution time of subroutines after IDCT optimization for "Foreman" QCIF sequence.

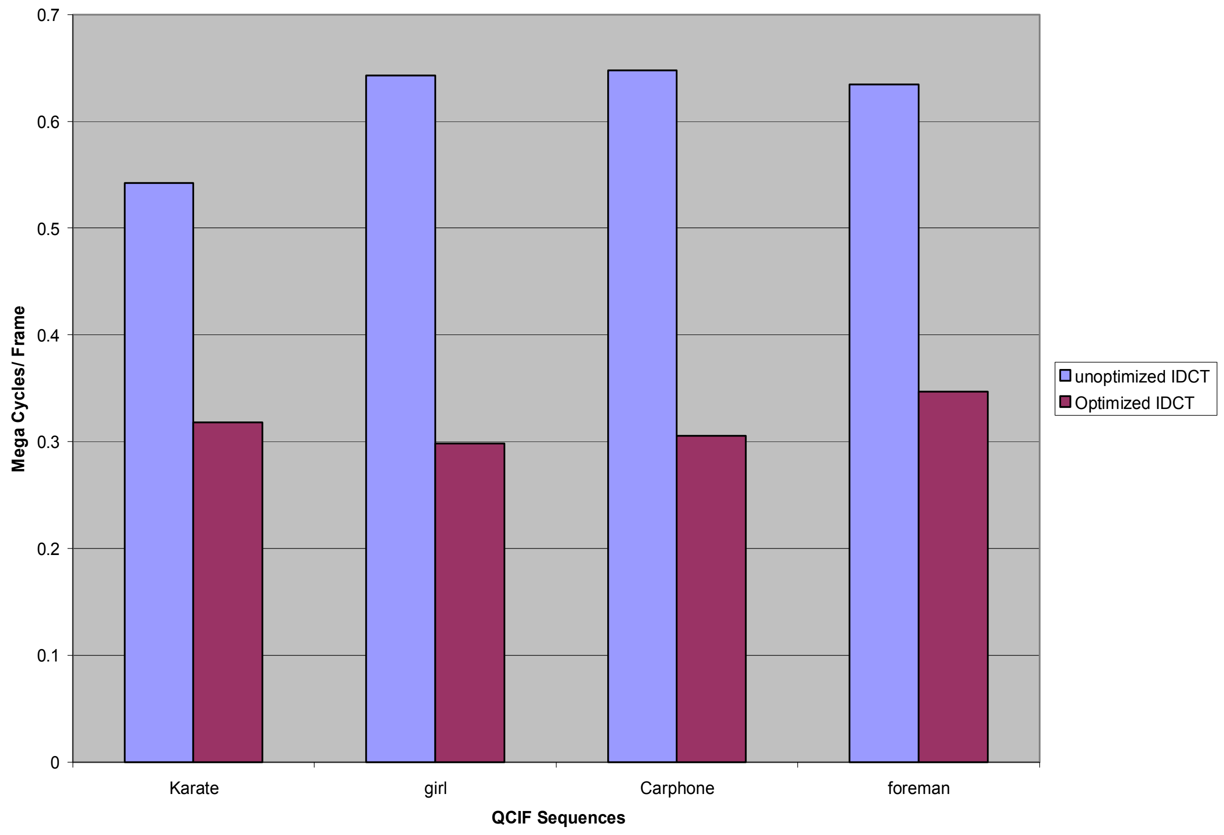


Figure 4.6: Reduction in ARM cycles after IDCT optimization for test sequences

4.3.3 Optimizing inter-frame prediction

Motion vectors are coded differentially using either median or directional prediction, depending on the partitioning mode that is used for the macroblock. For each motion vector, a predicted block must be computed by the decoder and then arranged with other blocks to form the predicted macroblock. Motion vectors for the luminance component are specified with quarter-sample accuracy. Interpolation of the reference video frames is necessary to generate the predicted macroblock using fractional pel motion vectors. The complexity of the required interpolation filter varies depending on the phase specified by the motion vector. To generate a predicted macroblock using a motion vector that indicates a half-sample position, an interpolation filter that is based on a 6-tap windowed sinc function is employed as in section 1.5. In the case of prediction using a motion vector that indicates a quarter-sample position, filtering consists of averaging two integer- or half-sample position values. A bilinear filter is used to interpolate the chrominance frames when subsampled motion vectors are used to predict the underlying chrominance blocks.

As clearly observed most of the time spent in this sub-routine is due to the intensive arithmetic calculations for interpolation is to get half-pel and quarter-pel values. The main section of the interpolation is the ½-pixel resolution applying the 6-tap filter [1 -5 20 20 -5 1]/32. The samples at half sample position is derived by first calculating intermediate value b_1 by applying the 6-tap filter as follows: $b_1 = (E - 5 F + 20 G + 20 H - 5 I + J)$ as in (1.1) and the final prediction value for location b is calculated by $b = (b_1 + 16) >> 5$ as in (1.3) and clipped to the range of 0 to 255.

For horizontal half pel value, as explained in section 4.3.2 four pixels are loaded at a time. So it takes only 2 cycles to load all the six pixels required. After masking, the data is arranged as in Fig. 4.5.

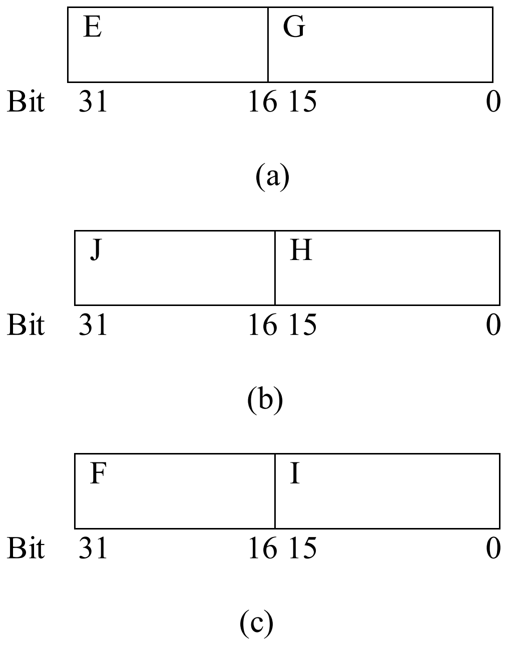


Figure 4.7: Packed pixels in 32-bit ARM registers ((a) Reg0, (b) Reg1, (c) Reg2)

Now perform

```
                                                                    Cycles

ADD R1, R0, R1              ; [E + J, G + H]                           1
AND R3, #0XFF, R1 LSL #4    ; [0, (G+H)* 16]                           1
ADD R3, R3, R3 LSL #2       ; [0, (G+H)* 16 + (G+H)* 4]                1
ADD R2, R2, R2 LSL #2       ; [(F*4) + F, (I*4) + I]                   1
ADD R4, R2, R2 LSR #16                                                 1
AND R4, R4, #0XFF           ; [0, (F+I) * 5]                           1
SUB R3, R3, R4              ; [0, ((G+H)* 20) - ((F+I) * 5)]           1
ADD R1, R3, R1 LSR #16      ; b1                                       1
```

.It does not take more than seven cycles to pack the pixels. So, effectively it takes seventeen cycles to get the result after applying 6 – tap filter including loading pixels from the memory. Whereas it does not take less than twenty two cycles to calculate the same before optimization.

Vertical interpolation with optimization gives a significant reduction in cycles as compared to optimizing horizontal interpolation. Six pixels are loaded in each of the four columns of block as in Fig. 4.6. Then masking and arranging 2 pixels in each register, we can perform the same operation to interpolate on two columns simultaneously. Since four pixels are loaded at once, memory access overhead is reduced to one fourth of the original.

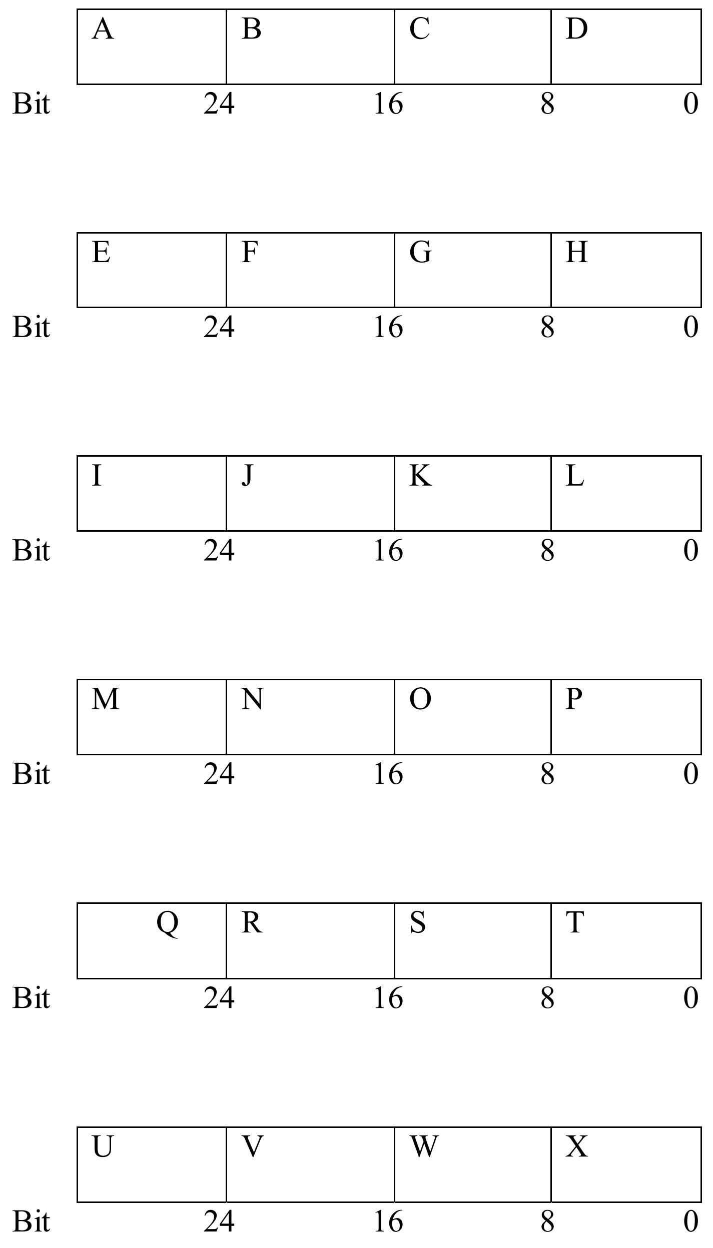


Figure 4.8: Four pixels in four columns are loaded in one 32-bit register

The profile data collected by the ADS ARM simulator shows that by using the above optimization in both horizontal and vertical interpolation of the 4x4 block, the over all decoder has achieved a significant speedup. The Table 4.4 and the Fig.4.7 explain the same.

Table 4.4: ARM9TDMI mega cycles/frame reduction before and after Interpolation optimization

Sequences	Before Optimization (Mega Cycles/ Frame)	After Optimization (Mega Cycles/ Frame)
Foreman	1.248	0.521
Carphone	1.145	0.658
Girl	0.773	0.436
Karate	1.054	0.75

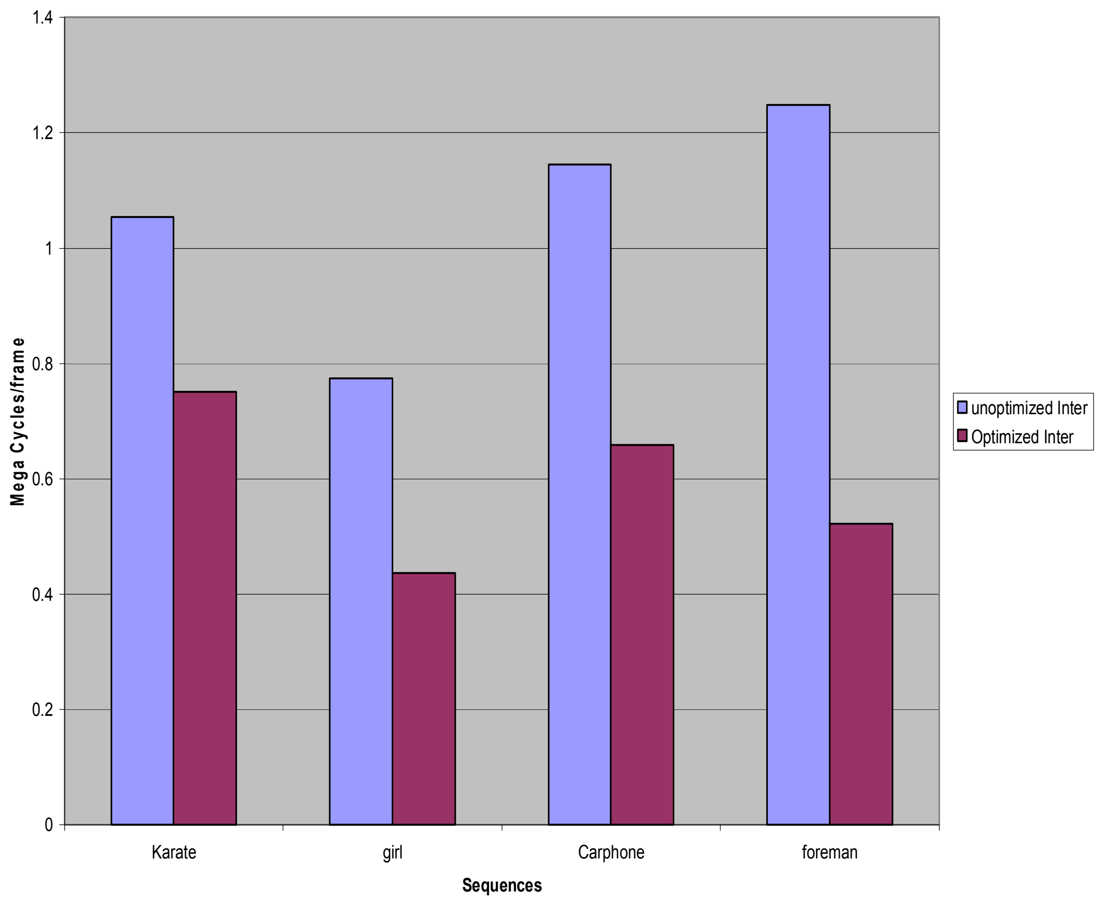


Figure 4.9: Reduction in ARM cycles after interpolation optimization for test sequences

4.3.4 Optimization of De-blocking Filter

De-blocking has been proved to be the most computational complex part in H.264 decoding [38]. The basic operation of de-blocking is filtering operation on the edges of 4x4 sub-blocks as in Fig.4.8, which is pixel level manipulation. For de-blocking procedure, there are two major tasks:

Get Strength: This part calculates the filtering strength of each edge based on the context information which involves large amount of conditional branches. For example, as is shown in Fig. 4.7, the filtering strength of sub-block b0 depends on only not the feature of macroblock A and B but also that of sub-block a0 and b0. And the filtering strength of b0b3 can be calculated along multiple data in parallel, by packing pixels in one 32-bit register, because they are independent to each other. The ARM Instruction Set Architecture (ISA) has conditional instruction as in section which can be used to execute conditional branches.

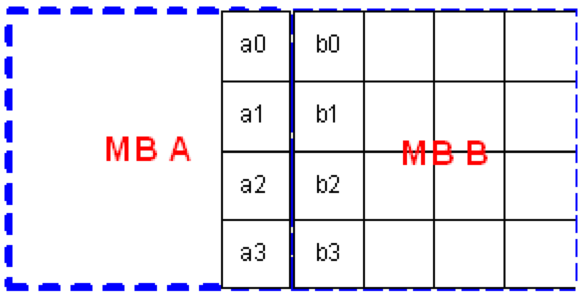


Figure 4.10: Get Strength context [35]

Loop-filter: Loop-filter is mainly multi-tap filter applied to the edge pixels in the decoded frame as in section. Thus similar optimization techniques used by interpolation can also be applied to this part. Still conditional branches can be found in this part, thus conditional instructions of ARM ISA can be efficiently used to achieve a significant reduction in processing time.

The results obtained, before and after optimization of deblocking filter, by running tests on different test streams are tabulated in the Table 4.5 and the Fig.4.9 explains the same.

Table 4.5: ARM9TDMI mega cycles/frame reduction before and after loop filter optimization

Sequences	Before Optimization (Mega Cycles/ Frame)	After Optimization (Mega Cycles/ Frame)
Foreman	2.381	1.221
Carphone	2.127	1.132
Girl	2.112	1.242
Karate	1.879	1.284

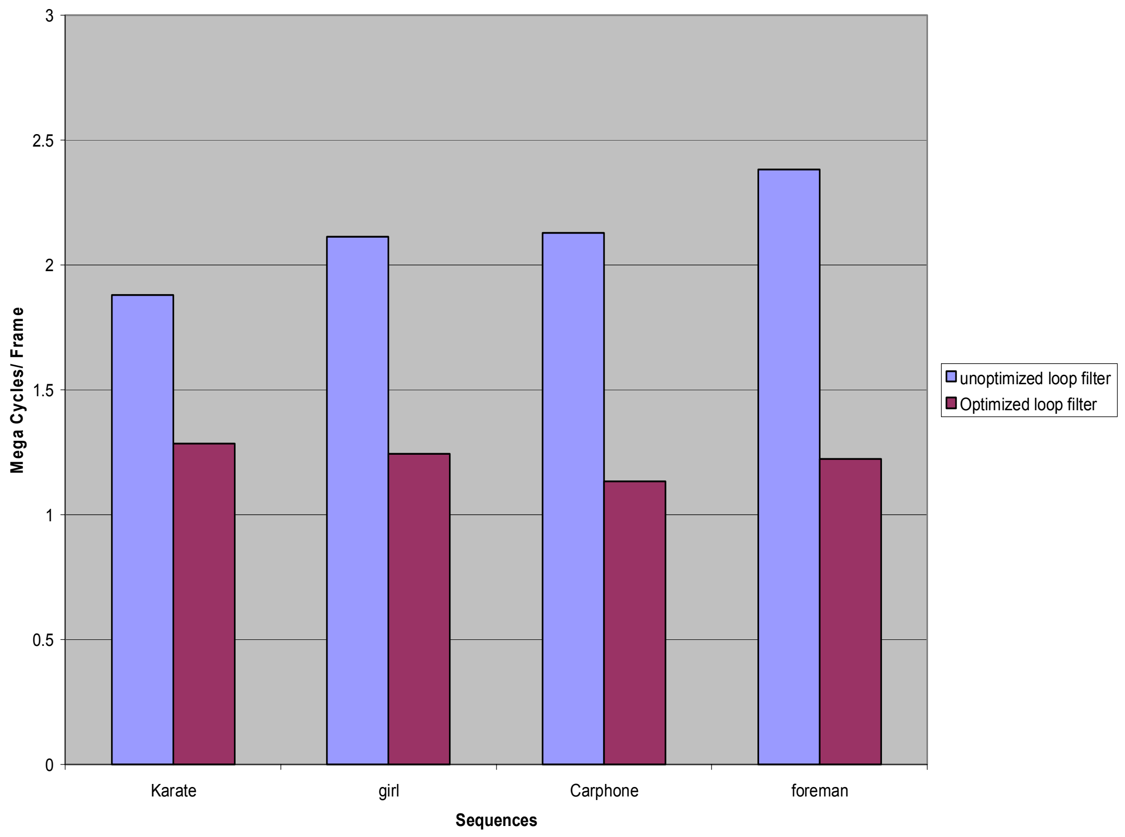


Figure 4.11: Reduction in ARM cycles after loop filter optimization for test sequences

CHAPTER 5

CONCLUSIONS AND FUTURE WORK

This thesis evaluates complexities of different building blocks of H.264 decoder for embedded platforms with ARM core. Many modifications are made to streamline the code and to improve performance. The optimizations are mainly done on 4x4 2D-IDCT, de-blocking filter and the algorithms for intra and inter predictions, which contribute to a significant decoding time.

The ARM9TDMI architecture supports several ways to tailor the H.264 decoder to run faster and efficient on it. These techniques are exploited to get the best results possible. The analysis is based on determining the number of CPU cycles required by the H.264 decoder to perform the key decoding sub-functions. The bit-streams for the test were generated using JM9.7 encoder [30] for a variety of contents and bit rates. With the help of profiling tools from ADSv1.2, it can shown that, with the proposed optimizing techniques, FastVDO LLC's H.264 baseline decoder takes on an average of 3.8 megacycles, along with the RGB conversion, to decode a 176x144 QCIF resolution frame.

The frequencies of ARM9TDMI core processor vary from 104MHz to 150MHz. The optimized decoder is ported to symbian [45] operating system for Nokia [46] 6630 which has ARM9 core running at 150MHz. With the overhead of the symbian OS the decoder on the cell phone decodes 15 to 20 frames a second. The test results for standard sequences show that the decoder is optimized to get an excellent performance to power consumption ratio.

ARM9TDMI core can be used as hardware accelerators for H.264 encoding on System on chip (SOC). Future work is to recognize the sub-modules in H.264 encoding and optimize those modules for ARM9 core.

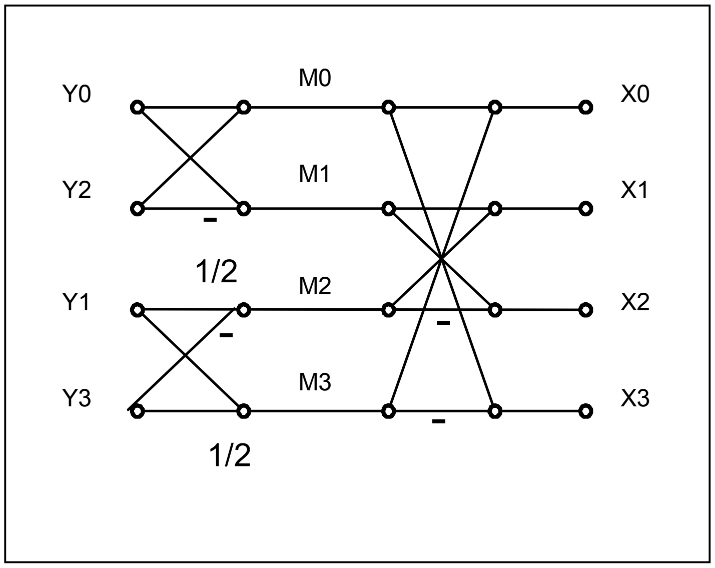
On the decoding side, this work can be extended to achieve real time decoding for sequences up to QVGA (320x240), which is a good resolution for cell phone applications. The existing decoder for ARM9 can be plugged into symbian multimedia framework, MMF [45], to achieve streaming solutions on cell phones.

APPENDIX A

IMPLEMENTATION STEPS OF 4X4 IDCT

Transforms via butterflies

The 2-D inverse transform can also be computed as 1-D row transforms followed by 1-D column transforms. Each of these 1-D transforms is computed by the equations below:



M0 = Y0 + Y2;

M1 = Y0 - Y2;

M2 = (Y1 >> 1) - Y3;

M3 = Y1 + (Y3 >> 1);

X0 = M0 + M3;

X3 = M0 - M3;

X1 = M1 + M2;

X2 = M1 - M2;

The Fig.A.1 below shows the pixel numbering of a 4x4 block. It is seen that the entire sixteen 8-bit pixels can be loaded into four 32-bit ARM registers using multiple load word ARM instructions. So, effectively it takes four cycles to load all the sixteen pixels of a 4x4 block.

```
LDMIA base, {R0-R3}        ; load registers R0 – R3 with sixteen 8 bit
                           ; pixels, takes four CPU cycles.
```

Now arrange these pixels in eight ARM registers, packing two pixels in each. This is shown in Fig.A.2.

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

Figure A.1: 4X4 Block Pixel Numbering

R4	4	0
R5	5	1
R6	6	2
R7	7	3
R8	9	8
R9	11	10
R10	13	12
R11	15	14
Bit	31 16	0

Figure A.2: Pixel arrangements in ARM registers

The assembly code to arrange four pixels is below

```
AND R5, R1, 0xFFFF                 ; Mask to get R5 = [0 0 P5 P4]

AND R4, R0, 0xFFFF                  ; Mask to get R4 = [0 0 P1 P0]

ORR R4, R5, LSL #16                ; R4 = [P5 P4 P1 P0]

AND R5, mask, R4, LSR #8           ; mask = #0xFF0FF, R5 = [0 P5 0 P1]

AND R4, R4, 0xFF0FF                 ; R4 = [0 P4 0 P0]
```

It takes 5 cycles to arrange four pixels in the required order. So it takes twenty four cycles in total to load and arrange as required.

Now, it can be clearly seen that it simply takes eight cycles to perform 1-D inverse transform on two rows at once. So it takes 40, (16 + 24), cycles in total to perform 1-D row transform.

REFERENCES

[1] D. LeGall, “MPEG: A video compression standard for multimedia applications,” Commun., ACM, vol. 34, pp. 46-58, Apr. 1991

[2] “The MPEG-2 international standard,” ISO/IEC, Reference number ISO/IEC 13818-2, 1996.

[3] “Video coding for audiovisual services at px64 kbits” ITU-T, ITU-T Recommendation H.261, Mar. 1993.

[4] T. Sikora, “The MPEG-4 video standard verification model,” IEEE Trans CSVT, vol. 7, pp. 19-31, Feb. 1997.

[5] “Video coding for low bit rate communications,” ITUT, ITU-T Recommendation H.263, ver. 1, 1995.

[6] JVT website: ftp://standards.polycom.com

[7] V. Lappalainen, et al, “Complexity of Optimized H.26L Video Decoder Implementation,” IEEE Trans. CSVT, vol 13., pp. 717-725, July 2003.

[8] M. Horowitz, A. Joch, F. Kossentini, and A. Hallapuro, “H.264/AVC Baseline Profile Decoder Complexity Analysis,” IEEE Trans. CSVT, vol 13., pp. 704-716, July 2003.

[9] H.264 / MPEG-4 Part 10 White Paper, Iain E.G. Richardson, www.vcodex.com.

[10] S. Wenger: “H.264/AVC over IP,” IEEE Trans. CSVT July 2003.

[11] T. Wedi, et al, " Motion- and aliasing-compensated prediction for hybrid video coding”, IEEE Trans. CSVT, pp 577-586, July 2003.

[12] T. Wiegand, et al, “Long-Term Memory Motion-Compensated Prediction,” IEEE Trans. CSVT, vol. 9, pp. 70-84, Feb. 1999.

[13] T. Wiegand and B. Girod, “Multi-frame Motion- Compensated Prediction for Video Transmission,” Kluwer Academic Publishers, Sept. 2001.

[14] H. Malvar, et al, “Low-Complexity Transform and Quantization in H.264/AVC,” in IEEE Trans. CSVT, pp 598-603, July 2003.

[15] P. List, et al, “Adaptive Deblocking Filter,” in IEEE Trans. CSVT, Vol.13, pp 614-619, July 2003.

[16] Wiegand, et al “Overview of the H.264 / AVC Video Coding Standard”, IEEE Trans. CSVT, Volume 13, pp 560 - 576 July 2003

[17] ARM system Developer’s Guide – Designing and optimizing System Software, Andrew N. Sloss, Dominic Symes, Chris Wright, Morgan Kaufmann Publishers, ISBN: 1-55860-874-5.

[18] http://www.arm.com/documentation/Instruction_Set/ - Instruction set manual.

[19] http://www.vcodex.com/h264.html - Overview of H.264

[20] http://www.cs.iastate.edu/~prabhu/Tutorial/CACHE/amdahl.html - Amdahl’s Law

[21] An introduction to the ITU-T H.263 video compression standard http://www.4i2i.com/h263_video_codec.htm

[22] Soon-Kak Kwon, A. Tamhankar and K.R. Rao, “Overview of H.264 / MPEG-4 Part 10”, Special issue on “Emerging H.264/AVC video coding standard, J. Visual Communication and Image Representation, vol.17, 2006.

[23] Information Technology – Generic coding of moving pictures and associated audio information: Video, ITU-T Rec. H.262 (2000 E).

[24] H.263: International Telecommunication Union, “Recommendation ITU-T H.263: Video Coding for Low Bit Rate Communication,” ITU-T, 1998.

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

[26] K. R. Rao and P. Yip, Discrete Cosine Transform, Orlando, FL: Academic Press, 1990.

[27] I. E.G. Richardson, H.264 and MPEG-4 Video Compression: Video Coding for Next-generation Multimedia, Wiley, 2003.

[28] M. Ghanbari, “Standard Codecs : Image Compression to Advanced Video Coding,” Hertz, UK: IEE, 2003.

[29] G. Sullivan, P. Topiwala and A. Luthra, “The H.264/AVC Advanced Video Coding Standard: Overview and Introduction to the Fidelity Range Extensions,” SPIE Conference on Applications of Digital Image Processing XXVII, vol. 5558, pp. 53-74, Aug. 2004.

[30] H.264 / AVC JM reference software - (JM 9.7) http://iphome.hhi.de/suehring/tml/download/

[31] Game Developers Conference, “3D Graphics optimizations for ARM architecture” – Gopi K. Kolli, Intel Corporation

[32] Application Note 34, writing efficient C for ARM, Document number: ARM DAI 0034A, January 1998

[33] Complexity-Distortion Analysis of H.264/JVT Decoders on Mobile Devices, Ray and Radha, Michigan State University.

[34] “Performance comparison of the emerging H.264 video coding standard with the existing standards” – Kamaci and Altunbasak, Center for Signal and Image Processing, Georgia Institute of Technology, Atlanta, GA, USA.

[35] "Optimization of a baseline H.263 video encoder on the TMS320C600" - R. Sheikh, Banerjee, Evans, and C. Bovik

[36] "Prediction based directional fractional pixel motion estimation for the H.264 video coding " - Libo Yang, Keman Yu, Jiang Li, and Shipeng Li.

[37] ARM9TDMI – Technical Reference Manual- www.arm.com

[38] Lappalainen, et al ,"Complexity of Optimized H.26L Video Decoder Implementation", in IEEE Trans. CSVT, vol.13, pp 717-725, July 2003.

[39] www.fastvdo.com

[40] Joint Video Team (JVT) of ISO/IEC MPEG & ITU-T VCEG (ISO/IEC JTC1/SC29/WG11 and ITU-T SG16 Q.6), Document: JVT-B038, "Low Complexity Transform and Quantization – Part I: Basic Implementation", Hallapuro, Karczewicz, Malvar.

[41] Tsu-Ming Liu, et al, "An 865-µW H.264/AVC Video Decoder for Mobile Applications", IEEE Asian solid state circuits conf., Hsinchu, Taiwan, Nov-2005.

[42] Jörn Ostermann, Lappalainen, et al, "Video coding with H.264/AVC: Tools, Performance, and Complexity", Circuits and Systems magazine, IEEE, Vol 4, pp 7-28, Jan 2004.

[43] "A high-speed low-cost DCT architecture for HDTV applications", Z-J. Mou and F. Jutand, Proceedings of International Conference on Acoustic, Speech, and Signal Processing ICASSP'91, ville, pays, pages 1153-1156, 1991.

[44] ARM Developer's Suite v1.2- codewarrior IDE guide, ARM DUI 0065D; www.arm.com

[45] www.symbian.com – All the symbian programming documents and example codes.

[46] www.forum.nokia.com – Developer's forum to develop applications for Nokia cell phones.

[47] A.Puri, X. Chen and A.Luthra, "Video coding using the H.264/MPEG-4 AVC compression standard", Signal processing: Image communication, Vol.19, pp.793-849, Oct.2004.

[48] Symbian Installation files to install H.264 decoder on series 60 symbian cell phones - http://www.fastvdo.com/H.264Mobile.html

BIOGRAPHICAL INFORMATION

The author, Sandya B Sheshadri, earned her bachelors degree in Electronics and Communications from Vishweshwaraiah Technological University, Karnataka, India in 2003. She joined University of Texas, Arlington in fall 2003 to study Masters in Electrical Engineering, specialization being video processing. She is currently working on developing H.264 codecs for mobile devices in particular on symbian operating systems. Her interests include research and development in image and video processing, mathematics, DSP processors, microcontrollers, ASIC, random signals and noise.