

國 立 交 通 大 學

電 子 工 程 學 系 電 子 研 究 所 碩 士 班

碩 士 論 文

使用 ARM9 處理器實現 MPEG-4 視訊之軟體解碼



**Software Implementation of MPEG-4 Video Decoder
on ARM9 Processor**

研 究 生：吳和璋

指 導 教 授：林大衛 博士

中 華 民 國 九 十 五 年 九 月

使用 ARM9 處理器實現 MPEG-4 視訊之軟體解碼

**Software Implementation of MPEG-4 Video Decoder
on ARM9 Processor**

研究生：吳和璋
指導教授：林大衛 博士

Student : Ho-Chang Wu
Advisor : Dr. David W. Lin



A Thesis

Submitted to Department of Electronics Engineering & Institute of Electronics

College of Electrical and Computer Engineering

National Chiao Tung University

In Partial Fulfillment of the Requirements

For the Degree of

Master of Science

In

Electronics Engineering

September 2006

Hsinchu, Taiwan, Republic of China

中華民國九十五年九月

使用 ARM9 處理器實現 MPEG-4 視訊之軟體解碼

研究生: 吳和璋

指導教授: 林大衛 教授

國立交通大學電子工程學系

電子研究所碩士班

摘要

MPEG-4 是個眾所週知的視訊標準，它對通訊傳播及互動式應用提供了具有彈性的視訊及音訊編碼。在本篇論文中，我們在 ARM9 微處理器上實現了 MPEG-4 視訊解碼。我們的參考軟體是 MoMuSys，它是個包含 MPEG-4 main profile 編碼及解碼的軟體。

ARM 處理器是個簡化指令集電腦(RISC)微處理器。為了實作以及有更好的表現，我們善用了 ARM 處理器的特色，例如條件執行以及一次讀取多個字(word)。

在我們的實作中，先用 ARM 開發工具(ADS)其中的編譯器和 VTune 來得到 MoMuSys 的特性資料。其中 VTune 是個為英特爾處理器而設計的軟體開發工具。接下來我們分析了這些特性資料並找出相當耗費時間的程式。在最佳化前，因為 MoMuSys 是個根據每個物件來解碼的解碼器，而我們只需要根據每張圖來解碼的解碼器，所以我們減少了程式的大小以及記憶體的使用。

在我們最佳化的過程中，我們先針對整個解碼的流程並根據離散餘弦轉換(DCT)的特性來跳過多餘的運算。然後根據各個相當耗費時間的程式的特性，使用了 ARM 處理器的特性以及適用於他們各自的方法來做最佳化。雖然我們沒有把整個解碼器做最佳化，我們的 MPEG-4 simple profile 解碼器已經達到了即時解碼的目標。跟別人的實作比較，我們實作的性能是有競爭性的。

在本篇論文中，我們先介紹了 MPEG-4 標準，還有 ARM 處理器的概述。然後討論了複雜度分析、實作策略、最佳化技巧、以及實驗結果。論文最後做一結論並提出未來可再繼續發展的主題。


Software Implementation of MPEG-4 Video Decoder on ARM9 Processor

Student: Ho-Chang Wu

Advisor: Dr. David W. Lin

Department of Electronics Engineering
Institute of Electronics
National Chiao Tung University

Abstract



The MPEG-4 standard is a public standard which provides a flexible video and audio coding solution for broadcast and interactive applications. In this thesis, we consider implementation of an MPEG-4 video decoder on the ARM9 system. Our reference software is the codec of MoMuSys which is a software for MPEG-4 main profile encoding and decoding.

The ARM processor is a reduced instruction set computer (RISC) microprocessor. For implementation, the feature of the ARM processor, such as conditional execution and loading multiple words, are employed for better performance.

In our implementation, we first get the profile of the MoMuSys code by using the compiler of the ARM Developer Suite (ADS) and VTune which is a software development tool for Intel processors. Then we analyze the profiles and find the time-critical functions. Before optimization, we reduce the code size and the usage of memory because the MoMuSys decoder is for object-based decoding and we implement frame-based decoder only.

In our optimization, we first focus on the decoding flow and skip some computations according to the nature of discrete cosine transform (DCT). Then we employ the features of the ARM processor to optimize the time-critical functions according to their characteristics respectively.

Although we do not optimize the whole decoder, our MPEG-4 simple profile decoder can achieve the goal of real-time decoding. Compared with other implementations, the performance of our implementation is competitive.

In this thesis, we first introduce the MPEG-4 standard and give an overview of the ARM9 processor. Then the complexity analysis, implementation strategies, the optimization techniques, and the experiment results are discussed. Finally, we give a conclusion and point out some subjects for potential future work.



誌謝

本篇論文的完成，由衷感謝我的指導教授—林大衛老師。從踏入交大研究所，老師就給予許多指導與鼓勵。在老師熱情的指導下，專業知識從無到有，從討論中獲得許多做研究的方法，也讓我走過兩年充實的研究生生涯。研究過程中，老師樂觀的生活態度，影響了我對人生的態度，之後即使面對難題，也能提起勇氣面對。老師就像一盞明燈，指引我求學與人生的方向，著實受益良多。

在充滿人情味的實驗室，即使再辛苦，我清楚地知道，我並不孤單，有許多學長姐、同學以及學弟與我一起奮鬥。承蒙家揚、俊榮、朝雄、崑健等學長的照顧，在遇到難題時，總是能給予適當的協助與鼓勵。感謝實驗室的崇諺、治傑、家賢、鴻志、育彰、旻弘、德亘，在他們的砥礪與幫助下，論文才能順利完成，還有學弟介遠和政達，從旁也給予不少協助。另外，好友志龍、哲欣、郁民，在這兩年來給予很多生活上、精神上的幫助，很感謝他們。

感謝我的家人，父親吳清安先生、母親陳美玉女士、大姐吳彩華小姐、二姐吳佩涵小姐，在我的求學路中，是我強力的後盾，讓我在生活上無後顧之憂，溫暖的家成了我動力的來源。另外感謝我摯愛的女友佩瑩，陪伴我一路走來，給予我許多支持與鼓勵，是我精神上的支柱。

我的父親，辛苦工作一生，卻英年早逝，於去年底得肺癌後，病痛纏身半年多，於今年八月中逝世。從我小時候，我的父親就教導我是非對錯的觀念，他溫和的脾氣以及任勞任怨的工作態度，讓我相當景仰，也對我的人格教育產生莫大影響。在我的求學途中，他不斷地給予實質與精神上的支持，也充分尊重我的意見，讓我一路走來平坦順利。即使在他生病時，也時時關心我的課業與人生規劃，是個完完全全的好父親。如今，他未能看到我從研究所畢業，是我人生中的一大遺憾。在此，僅將本篇論文獻給我最敬愛的父親吳清安先生。

吳和璋



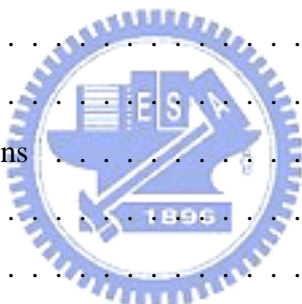
民國九十五年九月於新竹

Contents

1	Introduction	1
2	Overview of MPEG-4	3
2.1	Organization of the MPEG-4 Standard [2], [3]	3
2.2	MPEG-4 Video Coding Overview (from [4])	6
2.2.1	Structure of MPEG-4 Video Data	7
2.3	MPEG-4 Video Texture Coding (from [6], [7] and [8])	9
2.3.1	VOP Formation	10
2.3.2	Motion Coding	10
2.3.3	Texture Coder	15
2.4	Profiles and Levels (from [4])	19
2.4.1	Profiles and Levels	19
3	Environment of ARM9	22
3.1	Overview of the ARM Architecture (from [9])	22
3.1.1	Organization of the ARM9 Processor (from [10])	23
3.1.2	ARM Registers	24
3.1.3	Exceptions	25
3.1.4	Status Registers	26
3.2	ARM Instruction Set (from [9])	26
3.2.1	Branch Instructions	27
3.2.2	Data-Processing Instructions	27
3.2.3	Status Register Transfer Instructions	30

3.2.4	Load and Store Instructions	30
3.2.5	Coprocessor Instructions	32
3.2.6	Exception-Generating Instructions	33
3.3	The Thumb Instruction Set (from [9])	33
3.4	Memory and System Architectures	34
3.4.1	Write Buffers	36
3.4.2	Caches (from [11])	36
3.5	ARM Developer Suite (ADS) (from [12])	37
4	Analysis of Computational and Storage Complexity of MPEG-4 Framed- Based Video Decoder	41
4.1	Introduction to MoMuSys	41
4.2	Complexity Analysis of MoMuSys Decoder	42
4.2.1	Profile Using the Profiler of ADS	42
4.2.2	Low-Level Computational Complexity Analysis	46
4.3	Code Size and Memory Usage Reduction for Implementation	49
4.3.1	Code Size Reduction	49
4.3.2	Memory Usage Reduction	51
5	MPEG-4 Video Decoder Implementation and Optimization for ARM9	55
5.1	Algorithmic Optimization	55
5.1.1	Algorithmic Optimization for Blocks in Intra Frames with Null AC Coefficients [15]	56
5.1.2	Algorithmic Optimization for Null Residual Blocks of P-frames	59
5.1.3	Optimization for Image Interpolation and Padding	60
5.1.4	Optimization for Motion Compensation	65
5.2	Assembly/Architecture Level Optimization	67
5.2.1	Loop Overhead Reduction	67
5.2.2	Conditional Execution of Instructions	69
5.2.3	Reduction of Memory Accesses using LDM and STM	70

5.2.4	Experiment Results of Assembly/Architecture Level Optimization So Far	73
5.2.5	Optimization of IDCT	74
5.3	Conclusion on Optimization	78
5.3.1	Overall Improvement after Optimization	78
5.3.2	Profile Using the Profiler of ADS after Optimization	79
5.4	Comparison with Other Implementations	81
6	Conclusion and Future Work	84
6.1	Conclusion	84
6.2	Future Work	85
A	Assembly Code of Several Functions for Optimization	90
A.1	8×8 IDCT	90
A.2	VOP Reconstruction	94
A.3	Other Regular Functions	96
A.3.1	CopyImageI	96
A.3.2	Bzero	96
A.3.3	Putblock	97
A.3.4	MB_clip	97



List of Figures

2.1	A high level view of an MPEG-4 terminal (from [6]).	4
2.2	Segmentation of a picture to VOPs (from [6]).	8
2.3	Logical structure of coded video data (from [8]).	8
2.4	VOP types.	9
2.5	Positions of luminance and chrominance samples in 4:2:0 data (from [7]).	10
2.6	Detailed structure of VO encoder (from [6]).	11
2.7	Padding process (from [7]).	12
2.8	Interpolation scheme for half sample search.	12
2.9	Motion vector prediction (from [7]).	14
2.10	Quantizers of MPEG-4. (a) Quantizer for intra DC coefficient. (b) Quantizer for inter DC and all AC coefficients.	17
2.11	Prediction of DC coefficients of blocks in an intra MB (from [6]).	18
2.12	Prediction of AC coefficients of blocks in an intra MB (from [6]).	18
2.13	Scans for 8×8 blocks (from [4]).	20
3.1	5-stage organization of ARM9 (from [10]).	24
3.2	Format of the CPSR and the SPSRs (from [9]).	26
3.3	Formats of ARM instruction set (from [9]).	28
3.4	Formats of Thumb instruction set (from [9]).	35
3.5	ARM tools for developers.	39
4.1	Block diagram of MPEG-4 frame-based video decoder [4].	44
4.2	Flow of motion compensation.	48
4.3	Revised code using the #ifdef macros	52

5.1	DC spreading from decoded coefficient to output block (from [15]).	56
5.2	Modified flow of motion compensation with optimized interpolation.	63
5.3	Example of padding in the upper-left corner of a frame.	64
5.4	Modified flow of motion compensation with optimized padding.	65
5.5	Example code of interpolation and padding.	66
5.6	Example code of mode splitting.	68
5.7	Assembly code for for-loops generated by the compiler of ADS.	69
5.8	Our initial assembly code for for-loops.	69
5.9	Optimized assembly code for for-loops.	69
5.10	Example of zero-checking.	70
5.11	Saturation using conditional execution.	71
5.12	The IDCT algorithm used in MoMuSys [15].	78
5.13	The even-odd decomposition IDCT algorithm [20].	79



List of Tables

2.1	Default Quantization Matrix Q (from [4])	16
2.2	Nonlinear Scaler for DC Coefficients of DCT Blocks (from[4])	17
2.3	Profiles and Tools (from [4])	21
4.1	Functionalities of MoMuSys	43
4.2	Profile of Frame-Based MPEG-4 Decoding of QCIF Sequences on VTune (from [15])	45
4.3	Profile of Frame-Based MPEG-4 Decoding of QCIF Sequences on ADS .	46
4.4	Complexity of Luminance Motion Compensation for One QCIF Frame (from [15])	49
4.5	Complexity of Chrominance Motion Compensation for One QCIF Frame (from [15])	50
4.6	Complexity of Dequantization and IDCT for One 8×8 Block in MoMuSys	51
4.7	Files in MoMuSys That Are Not Needed for Simple Profile Implementa- tion	53
4.8	Functions with Memory Allocation Instructions	54
5.1	Number of Skipped Blocks in 90 Intra Frames (Check CBP and ACPred_Flag Only) (from [15])	58
5.2	Number of Skipped Blocks in 90 Intra Frames with Further Check After AC Prediction (from [15])	58
5.3	Execution Time of Intra Frame Decoding on ARM9	59
5.4	Number of Skipped Blocks in 89 P-Frames	59
5.5	Execution Time of Inter (P) Frame Decoding on ARM9	60

5.6	Analysis of Necessary Interpolation (from [15])	62
5.7	Execution Time and Storage Requirement of Image Interpolation and Padding on ARM9	63
5.8	Execution Time of P-Frame Decoding on ARM9 After Modification of Interpolation and Padding	65
5.9	Execution Time of P-Frame Decoding after Optimization of Motion Com- pensation on ARM9	67
5.10	Execution Time of Functions Optimized in Assembly/Architecture Level .	75
5.11	Execution Time of P-Frame Decoding after Optimization of VOP Recon- struction on ARM9	75
5.12	Improvement after Optimization for Regular Functions	76
5.13	Comparison of Computational Complexity for 8-point IDCT	76
5.14	Cycles of Floating-Point and Fixed-Point 1-D IDCT Using ADS Com- piler in Release Mode	76
5.15	Improvement after Optimization of IDCT	77
5.16	Overall Improvement after Optimization	79
5.17	Profile of Intra Frame Decoding after Optimization	81
5.18	Profile of P-Frame Decoding after Optimization	82
5.19	Performance of MPEG-4 Video Decoder on Different Platforms	83

Chapter 1

Introduction

Compression of audio-video data are becoming more and more important in multimedia communication. We have to make our existing communication channels and storage memory cost effective. The higher compression ratio we have, the more cost we save. However, speed is another issue for real-time implementation. It is important to find the balance in compression ratio, speed, and cost.

Various coding standards have been developed for different types of applications. The Moving Pictures Experts Group (MPEG) of the International Standardization Organization (ISO) formulated the MPEG-4 standard for compressing digital video and audio. The MPEG-4 video standard was originally for coding with high efficiency, very low bit-rate in mobile video communications, tele-shopping, interactive TV, etc. However, MPEG-4 video encoding and decoding requires significant amount of computation.

We consider implementation of an MPEG-4 video decoder on the ARM9 system. The implementation is based on modifying the code from MoMuSys [1], a main reference software of MPEG-4. The goal we want to reach is real-time implementation of MPEG-4 simple profile decoder and implementation of MPEG-4 main profile decoder.

Recently, the technique of embedded systems has been developed very well. The cause is that RISC cores are increasingly being used. A popular RISC processor in embedded applications is ARM core because of its low power consumption and small size in silicon, which are crucial for mobile applications. Using ARM9 system to implement MPEG-4 decoder, a key difficulty we face is optimizing the algorithms on a general pur-

pose RISC processor which is not explicitly designed for MPEG-4 applications.

This thesis is organized as follows. Chapter 2 is an overview of MPEG-4. Chapter 3 gives a brief description of ARM920T that we use. Chapter 4 discusses the analysis of the MPEG-4 simple profile video decoder for ARM920T. The optimization methods and the overall experimental results of the MPEG-4 decoder after optimization are described in chapter 5. Finally, chapter 6 contains the conclusion.



Chapter 2

Overview of MPEG-4

The ISO MPEG (Moving Picture Experts Group) committee, after successful completion of the MPEG-1 and the MPEG-2 standards, delivered MPEG-4 in 2000, the third MPEG standard. MPEG-4 was originally intended for very high compression coding of audio-visual information at very low bit-rates of 64 kbps or under. In general, MPEG-4 provides a flexible video and audio coding solution for broadcast and interactive applications. MPEG-4 builds success on of three fields:

- digital TV and HDTV,
- interactive graphics applications, and
- interactive multimedia (e.g., World Wide Web).

In this chapter, we introduce the organization of the MPEG-4 standard, its video texture encoding and decoding scheme, and other special video coding tools.

2.1 Organization of the MPEG-4 Standard [2], [3]

The MPEG-4 standard (ISO/IEC 14496) provides methods of audio-visual objects generic coding, as illustrated in Figure 2.1. It consists of the following basic parts.

1. ISO/IEC 14496-1: Systems

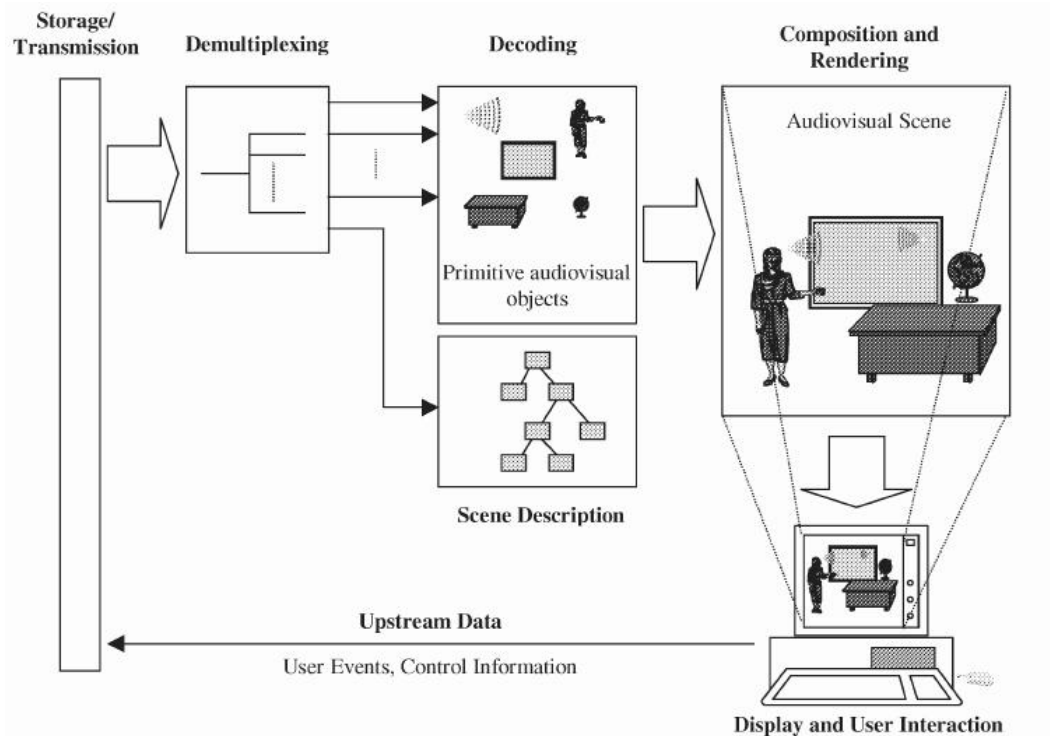


Figure 2.1: A high level view of an MPEG-4 terminal (from [6]).

The MPEG-4 Systems specification defines architecture and tools to create audio-visual scenes from individual objects. Scene description is a major tool for MPEG-4 systems. The MPEG-4 scene description, a totally new component in the MPEG specifications, is based on VRML (virtual reality modeling language) and specifies the spatial-temporal composition of objects in a scene. The scene description allows easy creation of compelling audio-visual content, and it is at the core of the systems specification.

2. ISO/IEC 14496-2: Visual

The MPEG-4 visual specification defines the main video codec. It consists of natural, arbitrary shape and synthetic video coding.

Similarly to MPEG-1 and MPEG-2, the main video coding tools are still texture coding for natural video coding. In order to reduce spatial redundancy, the MPEG-4 visual specification uses DCT, IDCT, intra prediction, quantization and de-quantization for intra coding. In order to reduce temporal redundancy, the MPEG-4 visual spec-

ification uses motion estimation and motion compensation for inter coding. In visual coding, the major difference from MPEG-1 and MPEG-2 is object coding. In MPEG-4, each picture is considered as consisting of objects, since some MPEG-4 functionalities require access not only to entire pictures but also to objects.

For synthetic video coding, in MPEG-4, mesh-based representation is useful. Triangular mesh-based representation of general objects is a tool in MPEG-4.

3. ISO/IEC 14496-3: Audio

ISO/IEC 14496-3 (MPEG-4 Audio) integrates many different types of audio coding: natural sound with synthetic sound, low bit-rate delivery with high-quality delivery, complex sound tracks with simple ones, speech with music, and traditional content with interactive and virtual-reality content. MPEG-4 Audio is a rather generic standard that applies to applications requiring the use of advanced sound compression, synthesis, manipulation, or playback. MPEG-4, unlike previous audio standards created by ISO/IEC and other groups, does not target at a single application such as real-time telephony or high-quality audio compression. The subparts specify state-of-the-art coding tools in several domains. However, MPEG-4 Audio is more than just the sum of its parts.

As the tools described are integrated with the rest of the MPEG-4 standard, new possibilities for object-based audio coding, interactive presentation, dynamic sound tracks, and other sorts of new media, are enabled.

4. ISO/IEC 14496-4: Conformance Testing

This part of ISO/IEC 14496 specifies how tests can be designed to verify whether bitstreams and decoders meet requirements specified in parts 1, 2, and 3 of ISO/IEC 14496. In this part of ISO/IEC 14496, encoders are not addressed specifically. An encoder may be said to be an ISO/IEC 14496 encoder if it generates bitstreams compliant with the syntactic and semantic bitstreams requirements specified in parts 1, 2 and 3 of ISO/IEC 14496.

5. ISO/IEC 14496-5: Reference Software

Reference software is normative in the sense that any conforming implementation of the software, taking the same conforming bitstreams, using the same output file format, will output the same file. Complying ISO/IEC 14496 implementations are not expected to follow the algorithms or the programming techniques used by the reference software.

6. ISO/IEC 14496-6: DMIF

Delivery Multi-media Integration Framework, DMIF, is an interface between the application and the transport, which enables the MPEG-4 application developer to be confident of the transport. Using the right DMIF instantiation, a single application can run on different transport layers.

MPEG-4 DMIF supports the following functionalities:

- A transparent MPEG-4 DMIF-application interface irrespective of whether the peer is a remote interactive peer, broadcast or local storage media.
- Support for mobile networks, developed together with ITU-T.
- User commands with acknowledgment messages.
- Management of MPEG-4 Sync Layer information.
- Control of establishing FlexMux channels.
- Homogeneous networks between interactive peers: IP, ATM, mobile, PSTN, Narrowband ISDN.

2.2 MPEG-4 Video Coding Overview (from [4])

MPEG-4 video provides standardized core technologies allowing efficient storage, transmission and manipulation of video data in multimedia applications. It provides technologies to view, access and manipulate objects, with great error robustness at a large range of bit-rates. Video activities in MPEG-4 aim at providing solutions in the form of tools and algorithms enabling functionalities such as efficient compression, object scalability, spatial and temporal scalability, error resilience, and fine granularity scalability.

2.2.1 Structure of MPEG-4 Video Data

Many of MPEG-4 functionalities have to access not only sequence of pictures, but also a whole object, and further, not only individual pictures, but also temporal instances of these objects within a picture. An input video sequence can be defined as a sequence of related pictures, separated in time.

MPEG-4 video applies to the concept of Video Objects (VOs) and their temporal instances, Video Object Planes (VOPs). Figure 2.2 is a diagram which describes the decomposition of a picture into a number of separate VOPs. Further, we show the class hierarchy used for representation of coded bitstreams in Figure 2.3.

- Video Session (VS)

Visual object sequence is the highest syntactic structure of the coded visual bitstream. The MPEG-4 scene may contain any 2-D or 3-D natural or synthetic objects.

- Video Object (VO)

Video Object represents a complete scene or a portion of a scene. The sequence, at the output of the decoding process, consists of a series of reconstructed VOPs separated in time and readied for display using the composition.

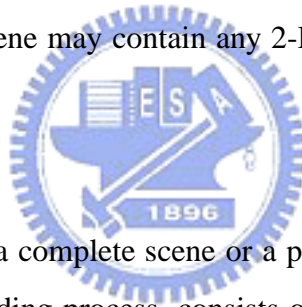
- Video Object Layer (VOL)

Coded video data consist of an ordered set of video bitstreams, called layers. Depending on the application, every video object can be encoded in scalable (multi-layer) or non-scalable form (single layer), represented by VOL. If there is only one layer, the coding process are called non-scalable video coding.

- Group of Video Object Planes (GOV)

Group of Video Object Planes (GOV) are optional entities and are essentially access units for editing, tune-in or synchronization.

- Video Object Plane (VOP)



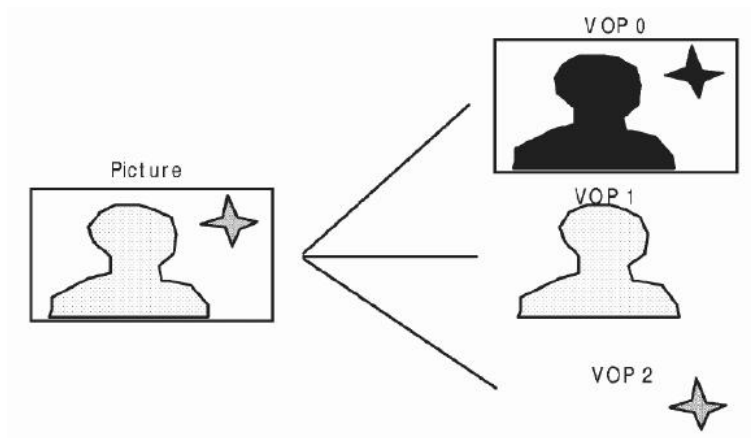


Figure 2.2: Segmentation of a picture to VOPs (from [6]).

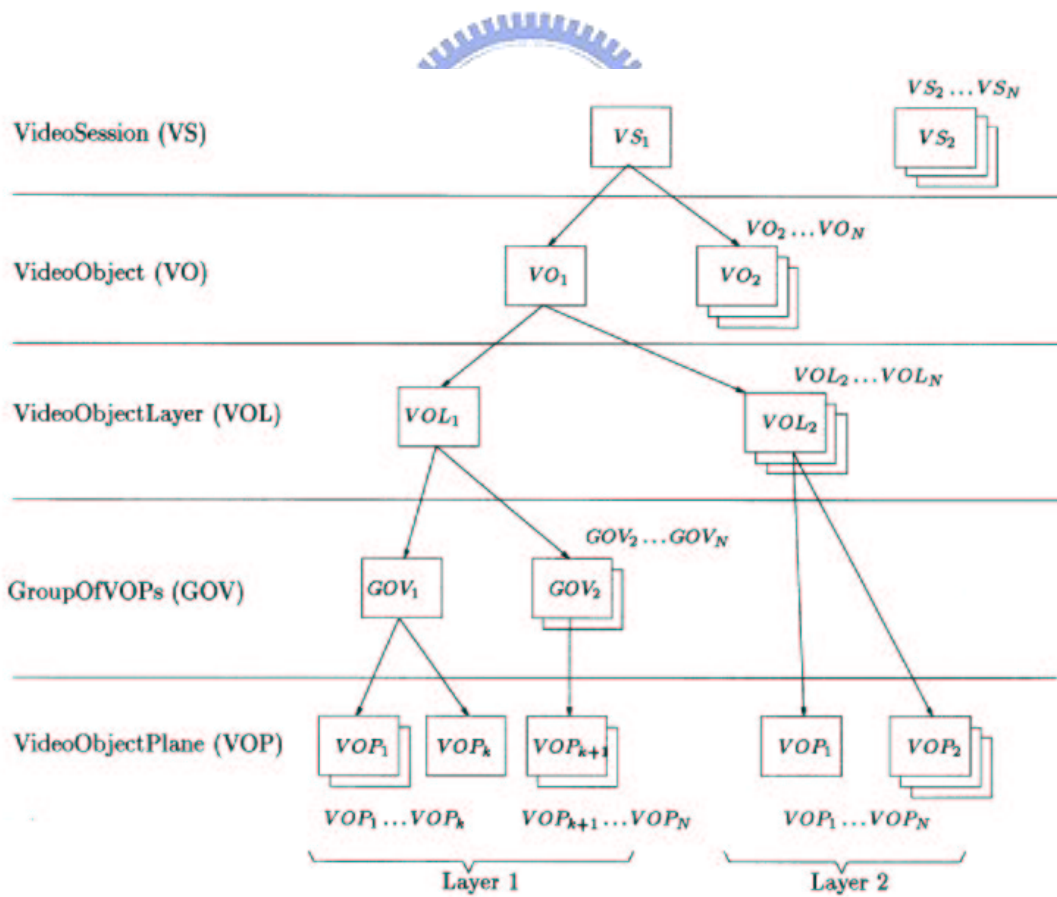


Figure 2.3: Logical structure of coded video data (from [8]).

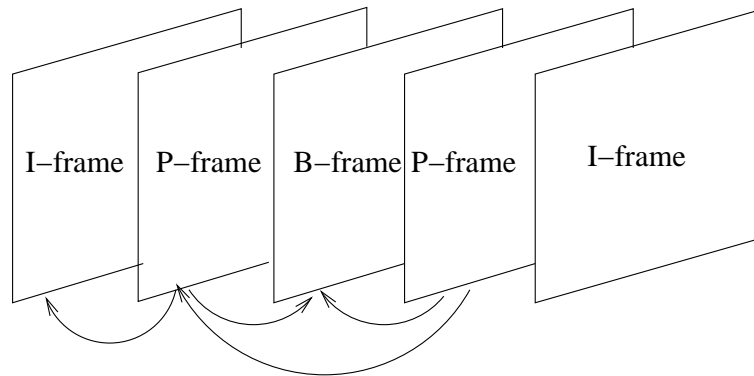


Figure 2.4: VOP types.

A Video Object Plane represents a snap shot in time of a Video Object. There are four types of VOP (Fig. 2.4) that use different coding methods:

1. I-VOP: An intra-coded VOP is coded using information only from itself.
2. P-VOP: A predictive-coded VOP is coded using motion compensated prediction from a past reference VOP.
3. B-VOP: A bidirectionally predictive-coded VOP is coded using motion compensated prediction from a past and/or future reference VOP(s).
4. S-VOP: A sprite is a VOP for a sprite object or a VOP which is coded using prediction based on global motion compensation from a past reference VOP.

The macroblock is a basic coding structure of VOP. It contains a section of the luminance component and the sub-sampled chrominance components in 4:2:0 format. There are 4 luminance blocks and 2 chrominance blocks in a macroblock. In this format, the luminance and chrominance samples are positioned as shown in Figure 2.5.

2.3 MPEG-4 Video Texture Coding (from [6], [7] and [8])

Because we consider implementation of frame based MPEG-4 decoder, we focus on introducing it, with only a small part addressing object-based MPEG-4 decoder.

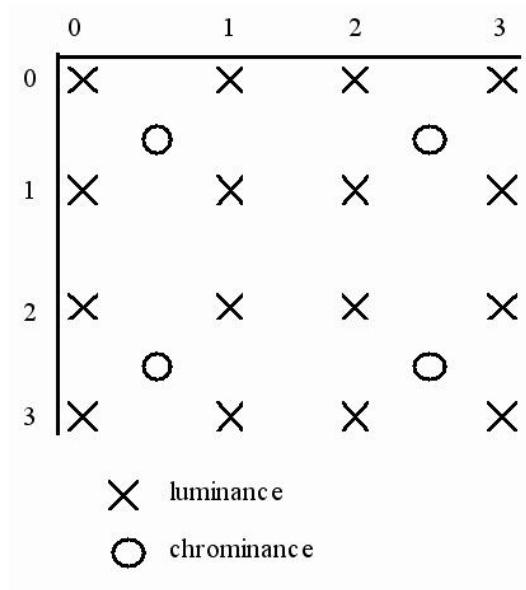


Figure 2.5: Positions of luminance and chrominance samples in 4:2:0 data (from [7]).

Figure 2.6 presents the internal structure of the VO encoder. The same encoding scheme is applied in coding all the VOPs of a given session. The encoder has an entirely new component compared to previous video coding standards: arbitrary shape coding.

2.3.1 VOP Formation

The video object shape information is obtained after segmentation. The shape information is hereafter referred to as alpha plane, which is used to form a VOP. The value 255 is assigned to pixels belonging to the objects and 0 is assigned to pixels outside the objects. However, as we consider frame-based coding, the pixels of an alpha plane are always 255.

2.3.2 Motion Coding

There are four types of VOPs (see Figure 2.4) that use different coding methods, I-VOP, P-VOP, B-VOP and S-VOP. Motion coding is essential for P-VOP and B-VOP to reduce temporal redundancy. The motion coder consists of a motion estimator, motion compensator, previous/next VOPs store and motion vector predictor and coder.

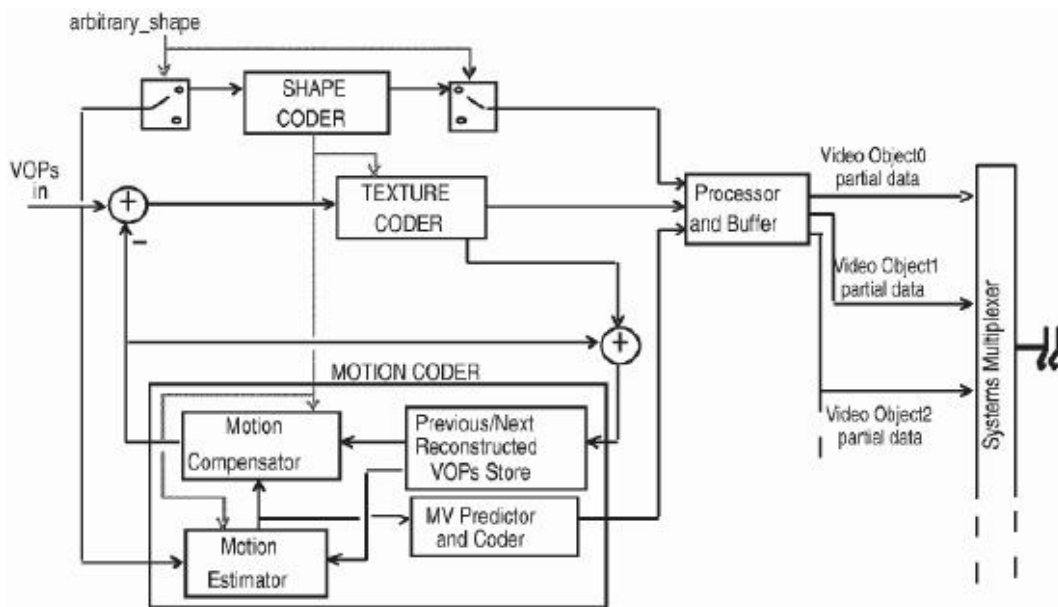


Figure 2.6: Detailed structure of VO encoder (from [6]).

Padding Process

Figure 2.7 shows a simplified diagram of the padding process. The values of luminance and chrominance samples outside the VOP are defined by the padding process.

By replicating the boundary samples of the VOP towards the exterior, a MB that lies on the VOP boundary is padded. This process is divided into horizontal repetitive padding and vertical repetitive padding.

Motion Estimation

Motion estimation (ME) is an important method for doing prediction between adjacent frames/pictures. Further, MPEG-4 encoder adopts block-based motion estimation technique, instead of pixel-based technique.

For every 16×16 luminance MB, the basic motion estimation is performed. Besides, motion vectors can be sent for individual 8×8 blocks to make prediction more accurate. One way of motion estimation is doing full search to integer pixel accuracy vector and, using it as the initial estimate, performing a half-pixel search around it.

Interpolation of MB is necessary because the motion vector may be non-integer numbers. Figure 2.8 illustrates the interpolation method. By bilinear interpolation, the half

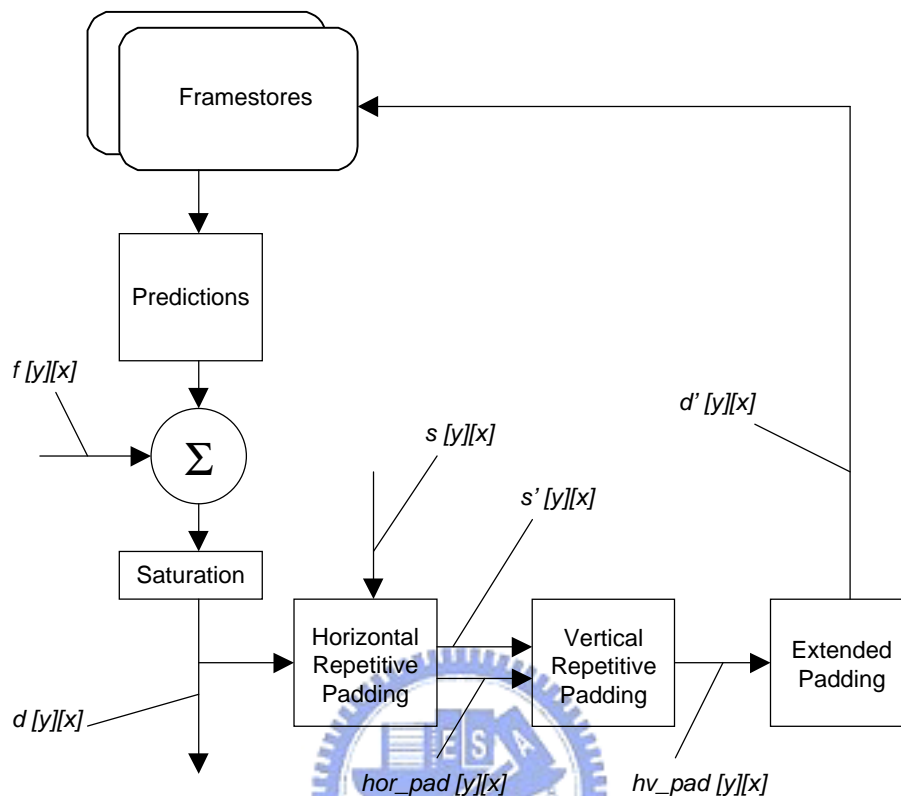
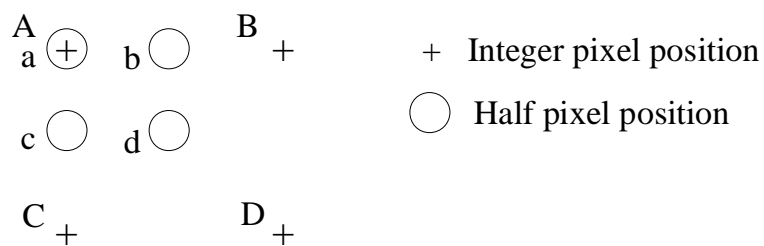


Figure 2.7: Padding process (from [7]).



$$\begin{aligned}
 a &= A, \\
 b &= (A + B + 1 - \text{rounding_control}) / 2 \\
 c &= (A + C + 1 - \text{rounding_control}) / 2, \\
 d &= (A + B + C + D + 2 - \text{rounding_control}) / 4
 \end{aligned}$$

Figure 2.8: Interpolation scheme for half sample search.

sample values can be calculated. Then, we will obtain the half-pixel motion vector by using interpolation.

Motion Vector Encoder

The motion vector will be coded when using INTER mode coding.

Motion vector is coded differentially by using a spatial neighborhood of three candidate MVs already coded (see Figure 2.9). At the borders of the current VOP, the following decision rules are applied:

1. If there is only one MB of candidate predictors outside the VOP, it is set to zero.
2. If there are two MBs of candidate predictors outside the VOP, they are set to the third candidate predictor.
3. If all three MBs of candidate predictors are outside the VOP, they are set to zero.

For horizontal and vertical components, the median value of the three candidates for the same component is used as predictor, denoted P_x and P_y , respectively:

$$P_x = \text{Median}(MV1_x, MV2_x, MV3_x),$$

$$P_y = \text{Median}(MV1_y, MV2_y, MV3_y).$$

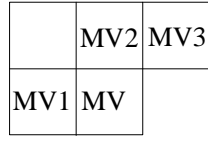
Then, the vector differences, $MVD_x (= MV_x - P_x)$ and $MVD_y (= MV_y - P_y)$, are coded by variable-length coding.

Motion Compensation

The motion compensation is performed on the prediction block, $pred[i][j]$, from the reference VOP. In addition to basic motion compensation processing, three alternatives are supported, namely, unrestricted motion compensation, four MV motion compensation and overlapped motion compensation.

For unrestricted motion compensation, the motion vectors are allowed to point outside the decoded area of a reference VOP. The $pred[i][j]$ is defined as:

$$x_{ref} = \min(\max(x_{curr} + dx, vhmcsr), x_{dim} + vhmcsr - 1),$$



MV : Current motion vector
 MV1: Previous motion vector
 MV2: Above motion vector
 MV3: Above right motion vector

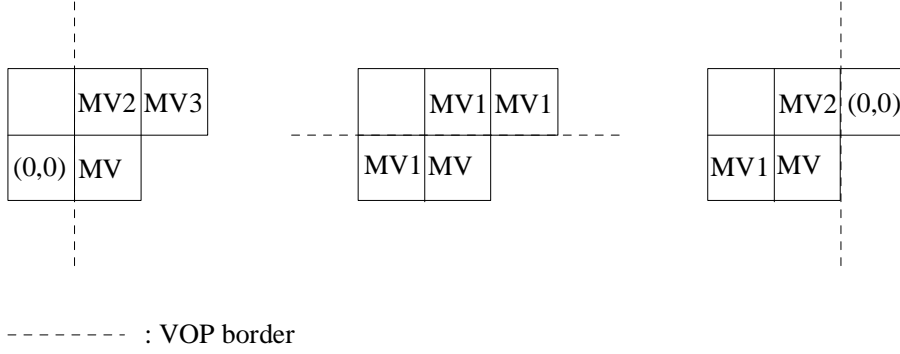


Figure 2.9: Motion vector prediction (from [7]).

$$y_{ref} = \min(\max(y_{curr} + dy, vvmcsr), ydim + vvmcsr - 1),$$

where $vhmcsr = vop_horizontal_mc_spatial_ref$, $vvmcsr = vop_vertical_mc_spatial_ref$, (y_{curr}, x_{curr}) are the coordinates of a sample in the current VOP, (y_{ref}, x_{ref}) are the coordinates of a sample in the reference VOP, (dy, dx) is the motion vector, and $(ydim, xdim)$ are the dimensions of the bounding rectangle of the reference VOP.

One/two/four vectors decision is indicated by the MCBPC codeword and field_prediction flag for each macroblock. If one motion vector is transmitted for a certain macroblock, this is defined as four vectors with the same value as the MV. When two field motion vectors are transmitted, each of the four block prediction motion vectors has the value equal to the average of the field motion vectors (rounded such that all fractional pixel offsets become half pixel offsets). If MCBPC indicates that four motion vectors are transmitted for the current macroblock, the information for the first motion vector is transmitted as the codeword MVD and the information for the three additional motion vectors is transmitted as the codewords MVD2–4. If four vectors are used, each of the motion vectors is used for all pixels in one of the four luminance blocks in the macroblock.

Overlapped motion compensation is performed when the flag $obmc_disable = 0$. Each pixel in an 8×8 luminance prediction block is a weighted sum of three prediction values, divided by 8. The creation of each pixel $\bar{P}(i, j)$, in an 8×8 luminance prediction block

is governed by the following equation:

$$\bar{P}(i, j) = \frac{(p(i+MV_x^0, j+MV_y^0)*H_0(i, j)+p(i+MV_x^1, j+MV_y^1)*H_1(i, j)+p(i+MV_x^2, j+MV_y^2)*H_2(i, j)+4)}{8},$$

where (MV_x^0, MV_y^0) denotes the motion vector for the current block, (MV_x^1, MV_y^1) denotes the motion vector of the block either above or below, (MV_x^2, MV_y^2) denotes the motion vector either to the left or right of the current block, and $H_0(i, j)$, $H_1(i, j)$, and $H_2(i, j)$ denote the weighting of each pixel in the current block and neighbor blocks.

Since the VOP may be coded in P or B mode, there are three types of motion vectors, forward mode, backward mode, and bi-directional mode. The different modes make different predictions $\bar{P}(i, j)$.

1. Forward mode

Only the forward vector (MVF_x, MVF_y) is applied in this mode. The prediction blocks $\bar{P}_y(i, j)$, $\bar{P}_u(i, j)$, $\bar{P}_v(i, j)$ are generated from the forward reference VOP.

2. Backward mode

Only the Backward vector (MVB_x, MVB_y) is applied in this mode. The prediction blocks $\bar{P}_y(i, j)$, $\bar{P}_u(i, j)$, $\bar{P}_v(i, j)$ are generated from the backward reference VOP.

3. Bi-directional mode

Both the forward vector (MVF_x, MVF_y) and the backward vector (MVB_x, MVB_y) are applied in this mode. The prediction blocks $\bar{P}_y(i, j)$, $\bar{P}_u(i, j)$, $\bar{P}_v(i, j)$ are generated from the forward and backward reference VOPs by doing the forward prediction and the backward prediction and then averaging both predictions pixel by pixel.

2.3.3 Texture Coder

There are three components, luminance Y and two chrominance components Cb and Cr, to present the texture information of a video signal. The texture information is directly in the luminance and chrominance components for an I-VOP. However, for a P-VOP and a B-VOP, the texture information represents only the residual after motion compensation.

The coding process for texture includes padding process (if needed), 8×8 block based DCT, quantization, coefficient prediction, coefficient scan and variable-length coding.

Discrete Cosine Transform Coding (DCT)

The transform coding in the MPEG-4 standard is based on 8×8 discrete cosine transform (DCT). Before quantization, the encoder does forward transform. Then the encoder does inverse transform after inverse quantization for reconstructing the VOP.

Quantization

MPEG-4 video supports two techniques of quantization (Q), the H.263 quantization method and the MPEG quantization method. We often use the H.263 quantization method, which is midtread quantizer for intra and inter AC coefficients and midrise quantizer for intra DC coefficients. The MPEG quantization method is a uniform quantizer with the default matrix.

The H.263 quantizer (see Figure 2.10) has uniform quantization for intra DC coefficients and nearly uniform midtread quantization for the inter DC and all AC coefficients. Values of AC data between $-Th$ and $+Th$ are quantized to zero. Coefficients in a macroblock are quantized with the same quantizer.

After DCT, each coefficient goes through a uniform quantizer. Table 2.1 is the default quantizer matrix.

In an intra macroblock, the DC coefficients of 8×8 blocks are scaled by a constant

Table 2.1: Default Quantization Matrix Q (from [4])

(intra)								(non intra)							
8	16	19	22	26	27	29	34	16	16	16	16	16	16	16	16
16	16	22	24	27	29	34	37	16	16	16	16	16	16	16	16
19	22	26	27	29	34	34	38	16	16	16	16	16	16	16	16
22	22	26	27	29	34	37	40	16	16	16	16	16	16	16	16
22	26	27	29	32	35	40	48	16	16	16	16	16	16	16	16
26	27	29	32	35	40	48	58	16	16	16	16	16	16	16	16
26	27	29	34	38	46	56	69	16	16	16	16	16	16	16	16
27	29	35	38	46	56	69	83	16	16	16	16	16	16	16	16

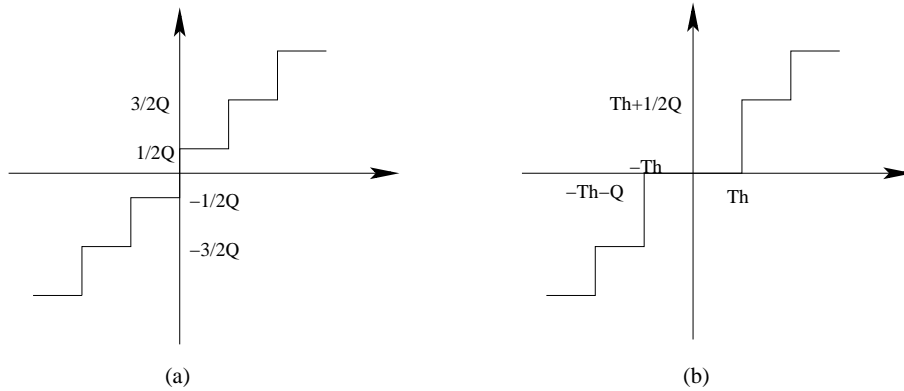


Figure 2.10: Quantizers of MPEG-4. (a) Quantizer for intra DC coefficient. (b) Quantizer for inter DC and all AC coefficients.

Table 2.2: Nonlinear Scaler for DC Coefficients of DCT Blocks (from[4])

component	DC scaler for Quantizer (Q) range			
	1-4	5-8	9-24	25-31
Luminance	8	$2Q$	$Q+8$	$2Q+16$
Chrominance	8	$\frac{Q+13}{2}$		$Q+16$

scaling factor of 8. Besides, Table 2.2 shows a nonlinear scaler which is used to provide a higher coding efficiency. We can see that the luminance and chrominance blocks use different quantizers.

Intra Prediction

Compared with inter prediction, intra prediction only uses self-information to predict the later data in the same VOP. When coding an intra block, the DC coefficient and many AC coefficients are coded by intra prediction. In the MPEG-4 standard, inter prediction is employed to reduce the spatial redundancy between 8×8 blocks.

DC prediction is illustrated in Figure 2.11. Taking three previous decoded DC coefficients as references, we can predict the DC value of the current block. For example, the DC value of block X is predicted from the DC coefficients of blocks A, B and C. The technique of prediction in MPEG-4 standards is gradient based. If the absolute value of a horizontal gradient is less than the absolute value of a vertical gradient, the QDC of

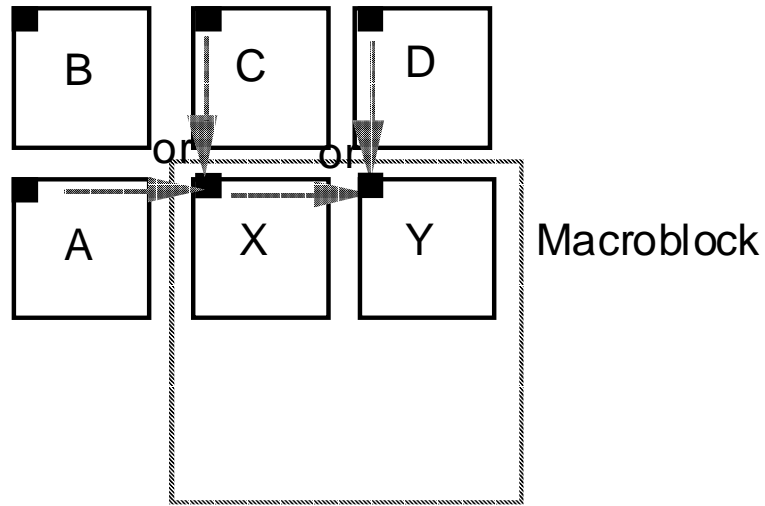


Figure 2.11: Prediction of DC coefficients of blocks in an intra MB (from [6]).

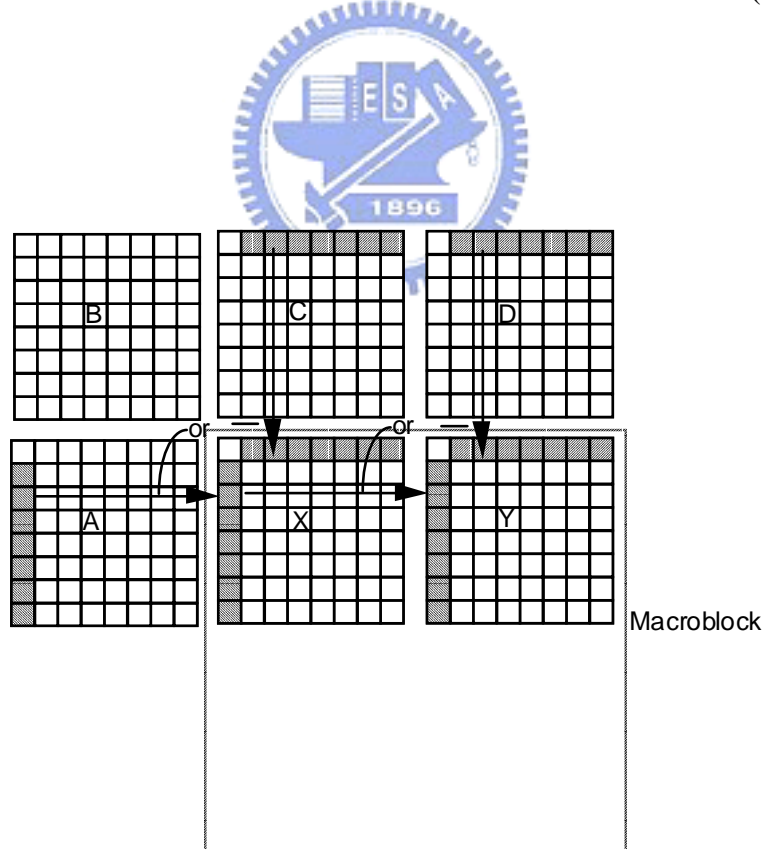


Figure 2.12: Prediction of AC coefficients of blocks in an intra MB (from [6]).

block C is used as the prediction, else QDC value of block A is used.

Figure 2.12 shows the AC prediction using the result of DC prediction. The direction of AC prediction is the same as DC prediction and only for the coefficients in the first row or in the first column.

Scan

Figure 2.13 shows three kinds of scan, alternate-horizontal, alternate-vertical and zigzag, to scan the DC and AC coefficients and change 2-D block data to 1-D data. The three scan types are chosen depending on the direction of DC prediction. If the direction is vertical, alternate-horizontal scan is used for the current block. If the direction is horizontal, alternate-vertical scan is selected for the current block. For all other blocks, zigzag scanned is used.

Variable-Length Coding (VLC)

After scan the coefficients become 1-D data, usually with many zeros. The data stream is good for run-length coding. The AC coefficients are encoded by the variable-length codes for EVENTS, where an EVENT consists of the triplet LAST, RUN, and LEVEL. LAST indicates that the code is the last code of the block or not. The value of RUN is the number of successive zeros preceding the coded coefficient. The value of LEVEL is the non-zero value of the coded coefficient.

2.4 Profiles and Levels (from [4])

2.4.1 Profiles and Levels

MPEG-4 defines many tools to compress the video data, but not all of them have to be implemented. Similar to MPEG-2, profiles and levels are defined as subsets of the entire bitstreams syntax containing all the tools. According to different applications, we can choose the profile we need. There are eight profiles defined by MPEG-4: simple, core, main, simple scalable, animated & mesh, basic animated texture, still scalable texture

0	1	5	6	14	15	27	28
2	4	7	13	16	26	29	42
3	8	12	17	25	30	41	43
9	11	18	24	31	40	44	53
10	19	23	32	39	45	52	54
20	22	33	38	46	51	55	60
21	34	37	47	50	56	59	61
35	36	48	49	57	58	62	63

zigzag

0	1	2	3	10	11	12	13
4	5	8	9	17	16	15	14
6	7	19	18	26	27	28	29
20	21	24	25	30	31	32	33
22	23	34	35	42	43	44	45
36	37	40	41	46	47	48	49
38	39	50	51	56	57	58	59
52	53	54	55	60	61	62	63

alternate-horizontal

0	4	6	20	22	36	38	52
1	5	7	21	23	37	39	53
2	8	19	24	34	40	50	54
3	9	18	25	35	41	51	55
10	17	26	30	42	46	56	60
11	16	27	31	43	47	57	61
12	15	28	32	44	48	58	62
13	14	29	33	45	49	59	63

alternate-vertical

Figure 2.13: Scans for 8×8 blocks (from [4]).

profile and simple face. Table 2.3 shows the detailed definitions.

The simple scalable profile is the same as simple profile, but with the rectangular scalability added. The core profile is the profile with all tools of the simple profile, temporal scalability, B-VOP coding and binary shape coding. The main profile is the profile with all tools in core profile, gray shape coding, interlace and sprite coding. The other profiles are for particular applications, such as 2D dynamic mesh coding and facial animation coding.

In the thesis, we focus on implementing the main profile decoder and optimizing the simple profile decoder.

Table 2.3: Profiles and Tools (from [4])

Visual Tools	Simple	Core	Main	Simple Scalable	Animated 2D Mesh	Basic Animated Texture	Still Scalable Texture	Simple Face Face
Basic 1. <i>I VOP</i> 2. <i>P VOP</i> 3. <i>AC/DC prediction</i> 4. <i>4MV unrestricted MV</i>	V	V	V	V	V			
Error resilience 1. <i>Slice resynchronization</i> 2. <i>Data partitioning</i> 3. <i>Reversible VLC</i>	V	V	V	V	V			
Short header	V	V	V		V			
B-VOP		V	V	V	V			
Method 1/Method 2 quantization		V	V		V			
P-VOP based temporal scalability 1. <i>Rectangular</i> 2. <i>Arbitrary shape</i>		V	V		V			
Binary shape		V	V		V			
Grey shape			V					
Interlace			V					
Sprite			V					
Temporal scalability (Rectangular)				V				
Spatial scalability (Rectangular)				V				
Scalable still Texture					V	V	V	
2D dynamic mesh with uniform topology					V	V		
2D dynamic mesh with Delaunay topology					V			
Facial animation Parameters								V

Chapter 3

Environment of ARM9

In this chapter, we introduce the environment of ARM9. The reduced instruction set computer (RISC) microprocessor ARM is a popular processor which was developed at Acorn Computers Limited of Cambridge, England, between 1983 and 1985. For MPEG-4 implementation, we use ARM920T as the core processor, which has a clock rate of 200 MHz and a 5-stage pipeline organization. We first introduce the architecture of ARM9. Then the instruction sets, memory architecture, and development tools will be introduced.



3.1 Overview of the ARM Architecture (from [9])

The ARM is a reduced instruction set computer (RISC), as it incorporates these typical RISC architecture features:

- a large uniform register file;
- a load/store architecture, where data-processing operations only operate on register contents, not directly on memory contents;
- simple addressing modes, with all load/store addresses being determined from register contents and instruction fields only; and
- uniform and fixed-length instruction fields, to simplify instruction decode.

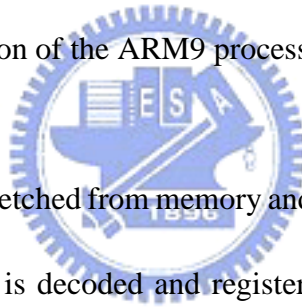
In addition, the ARM architecture has the following enhancements:

- control over both the Arithmetic Logic Unit (ALU) and shifter in every data-processing instruction to maximize the use of an ALU and a shifter;
- auto-increment and auto-decrement addressing modes to optimize program loops;
- “Load and Store Multiple” instructions to maximize data throughput; and
- conditional execution of all instructions to maximize execution throughput.

These enhancements to a basic RISC architecture allow ARM processors to achieve a good balance of high performance, low code size, low power consumption and low silicon area.

3.1.1 Organization of the ARM9 Processor (from [10])

Figure 3.1 shows the organization of the ARM9 processor. The functionalities of the five stages are:



- Fetch: The instruction is fetched from memory and placed in the instruction pipeline.
- Decode: The instruction is decoded and register operands read from the register files. There are 3 operand read ports in the register file so that most ARM instructions can source all their operands in one cycle.
- Execute: An operand is shifted and the ALU result generated. If the instruction is a load or store, the memory address is computed in the ALU.
- Buffer/Data: Data memory is accessed if required. Otherwise the ALU result is simply buffered for one cycle.
- Write back: The result generated by the instruction are written back to the register file, including any data loaded from memory.

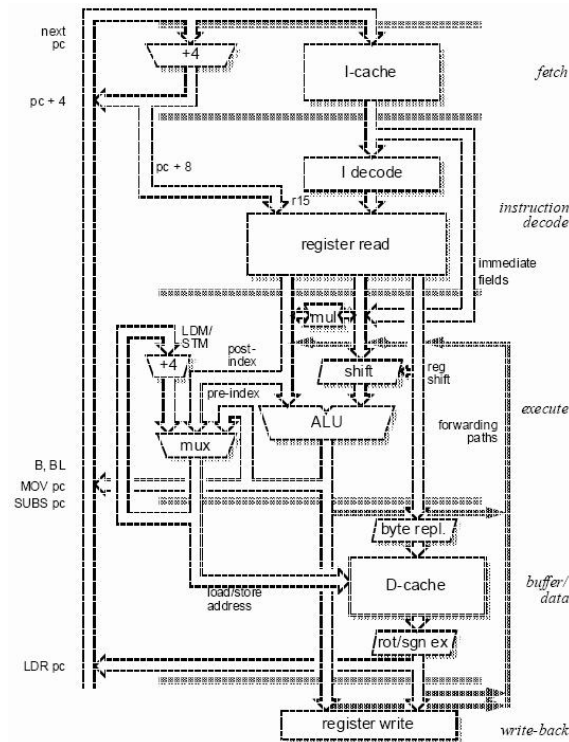


Figure 3.1: 5-stage organization of ARM9 (from [10]).

3.1.2 ARM Registers

ARM has 31 general-purpose 32-bit registers. At any one time, 16 of these registers are visible. The other registers are used to speed up exception processing. All the register specifiers in ARM instructions can address any of the 16 visible registers. Two of the 16 visible registers have special roles:

- Link register: Register 14 is the Link Register (LR). This register holds the address of the next instruction after a Branch and Link (BL) instruction, which is the instruction used to make a subroutine call. At all other times, R14 can be used as a general-purpose register.
- Program counter: Register 15 is the Program Counter (PC). It can be used in most instructions as a pointer to the instruction which is two instructions after the instruction being executed. All ARM instructions are four bytes long (one 32-bit word) and are always aligned on a word boundary. This means that the bottom two bits of the PC are always zero, and therefore the PC contains only 30 non-constant bits.

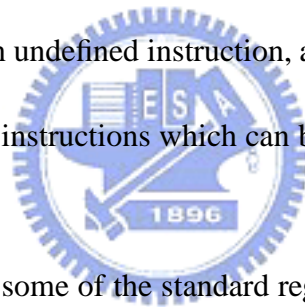
The remaining 14 registers have no special hardware purpose. Their uses are defined purely by software. Besides, software normally uses R13 as a Stack Pointer (SP).

3.1.3 Exceptions

ARM supports five types of exception, and a privileged processing mode for each type.

The five types of exceptions are:

- fast interrupt,
- normal interrupt,
- memory aborts, which can be used to implement memory protection or virtual memory,
- attempted execution of an undefined instruction, and
- software interrupt (SWI) instructions which can be used to make a call to an operating system.



When an exception occurs, some of the standard registers are replaced with registers specific to the exception mode. All exception modes have banked registers which are the replacements for R13 and R14. The fast interrupt mode has more registers for fast interrupt processing.

When an exception handler is entered, R14 holds the return address for exception processing. This is used to return after the exception is processed and to address the instruction that caused the exception.

When an exception occurs, the ARM processor halts execution after the current instruction and begins execution at one of a number of fixed addresses in memory, known as the exception vectors. There is a separate vector location for each exception.

An operating system installs a handler for every exception at initialization. Privileged operating system tasks are normally run in System mode to allow exceptions to occur within the operating system without state loss.

3.1.4 Status Registers

All processor states other than the general-purpose register contents is held in status registers. The current operating processor status is in the Current Program Status Register (CPSR). The CPSR holds:

- 4 condition code flags (Negative, Zero, Carry and overflow),
- 2 interrupt disable bits, one for each type of interrupt,
- 5 bits which encode the current processor mode, and
- 1 bit which encodes whether ARM or Thumb instructions are being executed execution throughput.

Each exception mode also has a Saved Program Status Register (SPSR) which holds the CPSR of the task immediately before the exception occurred. The CPSR and the SPSRs are accessed with special instructions. The bit fields of CPSR and SPSR are shown in Figure 3.2.



3.2 ARM Instruction Set (from [9])

The ARM instruction set can be divided into six broad classes of instruction:

- Branch instructions
- Data-processing instructions
- Status register transfer instructions
- Load and store instructions

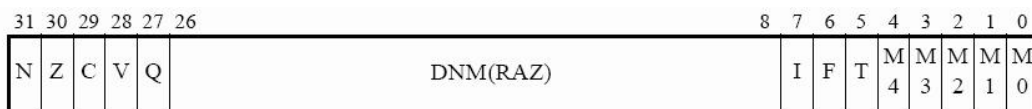


Figure 3.2: Format of the CPSR and the SPSRs (from [9]).

- Coprocessor instructions
- Exception-generating instructions

Most data-processing instructions and one type of coprocessor instructions can update the four condition code flags in the CPSR (Negative, Zero, Carry and oVerflow) according to their result.

Figure 3.3 shows the ARM instruction formats. We can see that almost all ARM instructions contain a 4-bit condition field. One value of this field specifies that the instruction is executed unconditionally.

3.2.1 Branch Instructions

As well as allowing many data-processing or load instructions to change control flow by writing the PC, a standard Branch instruction is provided with a 24-bit signed offset, allowing forward and backward branches of up to 32 MB.

There is a Branch and Link (BL) option that also preserves the address of the instruction after the branch in R14, the LR. This provides a subroutine call which can be returned from by copying the LR into the PC.

There are also branch instructions which can switch the instruction set, so that execution continues at the branch target using the Thumb instruction set. These allow ARM code to call Thumb subroutines, and ARM subroutines to return to a Thumb caller. Similar instructions in the Thumb instruction set allow the corresponding Thumb-to-ARM switches.

3.2.2 Data-Processing Instructions

The data-processing instructions perform calculations on the general-purpose registers. There are four types of data-processing instructions:

- Arithmetic/logic instructions
- Comparison instructions

		31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Data processing immediate shift	cond [1]	0	0	0	opcode	S	Rn	Rd	shift amount	shift	0	Rm																					
Miscellaneous instructions: See Figure 3-3	cond [1]	0	0	0	1	0	x	x	0	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
Data processing register shift [2]	cond [1]	0	0	0	opcode	S	Rn	Rd	Rs	0	shift	1	Rm																				
Miscellaneous instructions: See Figure 3-3	cond [1]	0	0	0	1	0	x	x	0	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	
Multiplies, extra load/stores: See Figure 3-2	cond [1]	0	0	0	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	
Data processing immediate [2]	cond [1]	0	0	1	opcode	S	Rn	Rd	rotate	immediate																							
Undefined instruction [3]	cond [1]	0	0	1	0	x	0	0	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	
Move immediate to status register	cond [1]	0	0	1	0	R	1	0	Mask	SBO	rotate	immediate																					
Load/store immediate offset	cond [1]	0	1	0	P	U	B	W	L	Rn	Rd	immediate																					
Load/store register offset	cond [1]	0	1	1	P	U	B	W	L	Rn	Rd	shift amount	shift	0	Rm																		
Undefined instruction	cond [1]	0	1	1	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x		
Undefined instruction [4,7]	1	1	1	1	0	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	
Load/store multiple	cond [1]	1	0	0	P	U	S	W	L	Rn	register list																						
Undefined instruction [4]	1	1	1	1	1	0	0	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	
Branch and branch with link	cond [1]	1	0	1	L	24-bit offset																											
Branch and branch with link and change to Thumb [4]	1	1	1	1	1	0	1	H	24-bit offset																								
Coprocessor load/store and double register transfers [6]	cond [5]	1	1	0	P	U	N	W	L	Rn	CRd	cp_num	8-bit offset																				
Coprocessor data processing	cond [5]	1	1	1	0	opcode1	CRn	CRd	cp_num	opcode2	0	CRm																					
Coprocessor register transfers	cond [5]	1	1	1	0	opcode1	L	CRn	Rd	cp_num	opcode2	1	CRm																				
Software interrupt	cond [1]	1	1	1	1	swi number																											
Undefined instruction [4]	1	1	1	1	1	1	1	1	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x		

Figure 3.3: Formats of ARM instruction set (from [9]).

- Multiply instructions
- Count Leading Zeros instruction

Arithmetic/Logic Instructions

There are twelve arithmetic/logic instructions which share a common instruction format. They perform an arithmetic or logical operation on up to two source operands, and write the result to a destination register. They can also optionally update the condition code flags based on the result. Of the two source operands:

- one is always a register,

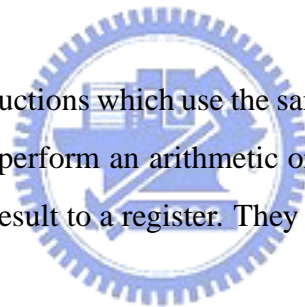
- the other has two basic forms:
 - an immediate value,
 - a register value, optionally shifted.

If the operand is a shifted register, the shift amount can be either an immediate value or the value of another register. Four types of shift can be specified. Every arithmetic/logic instruction can therefore perform an arithmetic/logic and a shift operation. As a result, ARM does not have dedicated shift instructions.

Because the Program Counter (PC) is a general-purpose register, arithmetic/logic instructions can write their results directly to the PC. This allows easy implementation of a variety of jump instructions.

Comparison Instructions

There are four comparison instructions which use the same instruction format as the arithmetic/logic instructions. They perform an arithmetic or logical operation on two source operands, but do not write the result to a register. They always update the condition flags based on the result.



The source operands of comparison instructions take the same forms as those of arithmetic/logic instructions, including the ability to incorporate a shift operation.

Multiply Instructions

Multiply instructions come in two classes. Both types multiply two 32-bit register values and store their result:

- 32-bit result: Normal. Stores the 32-bit result in a register.
- 64-bit result: Long. Stores the 64-bit result in two separate registers.

Both types of multiply instruction can optionally perform an accumulate operation.

Count Leading Zeros Instruction

The Count Leading Zeros (CLZ) instruction determines the number of zero bits at the most significant end of a register value, up to the first 1 bit. This number is written to the destination register of the CLZ instruction.

3.2.3 Status Register Transfer Instructions

The status register transfer instructions transfer the contents of the CPSR or an SPSR to or from a general-purpose register. Writing to the CPSR can:

- set the values of the condition code flags,
- set the values of the interrupt enable bits, or
- set the processor mode.

3.2.4 Load and Store Instructions

The following load and store instructions are available:

- Load and Store Register
- Load and Store Multiple registers
- Swap register and memory contents

Load and Store Register

Load Register instructions can load a 32-bit word, a 16-bit halfword or an 8-bit byte from memory into a register. Byte and halfword loads can be automatically zero-extended or sign-extended as they are loaded.

Store Register instructions can store a 32-bit word, a 16-bit halfword or an 8-bit byte from a register to memory.

Load and Store Register instructions have three primary addressing modes, all of which use a base register and an offset specified by the instruction:

- In offset addressing, the memory address is formed by adding or subtracting an offset to or from the base register value.
- In pre-indexed addressing, the memory address is formed in the same way as for offset addressing. As a side-effect, the memory address is also written back to the base register.
- In post-indexed addressing, the memory address is the base register value. As a side-effect, an offset is added to or subtracted from the base register value and the result is written back to the base register.

In each case, the offset can be either an immediate or the value of an index register. Register-based offsets can also be scaled with shift operations.

As the PC is a general-purpose register, a 32-bit value can be loaded directly into the PC to perform a jump to any address in the 4 GB memory space.

Load and Store Multiple Registers

Load Multiple (LDM) and Store Multiple (STM) instructions perform a block transfer of any number of the general-purpose registers to or from memory. Four addressing modes are provided:

- pre-increment,
- post-increment,
- pre-decrement, and
- post-decrement.

The base address is specified by a register value, which can be optionally updated after the transfer. As the subroutine return address and PC values are in general-purpose registers, very efficient subroutine entry and exit sequences can be constructed with LDM and STM:

- A single STM instruction at subroutine entry can push register contents and the return address onto the stack, updating the stack pointer in the process.
- A single LDM instruction at subroutine exit can restore register contents from the stack, load the PC with the return address, and update the stack pointer.

LDM and STM instructions also allow very efficient code for block copies and similar data movement algorithms.

Swap Register and Memory Contents

A swap (SWP) instruction performs the following sequence of operations:

1. It loads a value from a register-specified memory location.
2. It stores the contents of a register to the same memory location.
3. It writes the value loaded in step 1 to a register.

By specifying the same register for steps 2 and 3, the contents of a memory location and a register are interchanged.



3.2.5 Coprocessor Instructions

There are three types of coprocessor instructions as follows.

Data-Processing Instructions

These start a coprocessor-specific internal operation.

Data Transfer Instructions

These transfer coprocessor data to or from memory. The address of the transfer is calculated by the ARM processor.

Register Transfer Instructions

These allow a coprocessor value to be transferred to or from an ARM register.

3.2.6 Exception-Generating Instructions

Two types of instruction are designed to cause specific exceptions to occur.

Software Interrupt (SWI) Instructions

SWI instructions cause a software interrupt exception to occur. These are normally used to make calls to an operating system, to request an OS-defined service. The exception entry caused by a SWI instruction also changes to a privileged processor mode. This allows an unprivileged task to gain access to privileged functions, but only in ways permitted by the OS.

Software Breakpoint (BKPT) Instructions

BKPT instructions cause an abort exception to occur. If suitable debugger software is installed on the abort vector, an abort exception generated in this fashion is treated as a breakpoint. If debug hardware is present in the system, it can instead treat a BKPT instruction directly as a breakpoint, preventing the abort exception from occurring.

In addition to the above, the following types of instruction cause an Undefined Instruction exception to occur:

- coprocessor instructions which are not recognized by any hardware coprocessor, and
- most instruction words that have not yet been allocated a meaning as an ARM instruction.

In each case, this exception is normally used either to generate a suitable error or to initiate software emulation of the instruction.

3.3 The Thumb Instruction Set (from [9])

Compared with the ARM instructions, the Thumb instructions address the issue of code density. Thumb only uses 16-bit length and highly increases the code density. Besides, The Thumb instruction set is a re-encoded subset of the ARM instruction set.

Thumb does not alter the underlying programmer's model of the ARM architecture. It merely presents restricted access to it. All Thumb data-processing instructions operate on full 32-bit values, and full 32-bit addresses are produced by both data-access instructions and instruction fetches.

When the processor is executing Thumb instructions, eight general-purpose integer registers are available (R0 to R7) which are the same physical registers as R0 to R7 when executing ARM instructions. Some Thumb instructions also access the Program Counter (ARM Register 15), the Link Register (ARM Register 14) and the Stack Pointer (ARM Register 13).

Thumb does not provide direct access to the CPSR or any SPSR (as the MSR and MRS instructions do in the ARM instruction set). Thumb execution is flagged by the T bit (bit5) in the CPSR:

- T == 0: 32-bit instructions are fetched (and the PC is incremented by four) and are executed as ARM instructions.
- T == 1: 16-bit instructions are fetched (and the PC is incremented by two) and are executed as Thumb instructions.

Figure 3.4 shows the instruction formats of Thumb. The main difference between ARM and Thumb instructions is the condition field. For higher code density, there is no condition field in Thumb instructions. Besides, the offset field and immediate field are smaller.

3.4 Memory and System Architectures

ARM processors and software are designed to be connected to a byte-addressed memory. Word and halfword accesses to the memory ignore the alignment of the address and access the naturally-aligned value that is addressed (so a memory access ignores address bits 0 and 1 for word access, and ignores bit 0 for halfword accesses). The endianness of the ARM processor should normally match that of the memory system, or be configured to

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Shift by immediate	0	0	0	opcode [1]			immediate				Rm		Rd			
Add/subtract register	0	0	0	1	1	0	opc	Rm			Rn		Rd			
Add/subtract immediate	0	0	0	1	1	1	opc	immediate				Rn		Rd		
Add/subtract/compare/move immediate	0	0	1	opcode			Rd / Rn		immediate							
Data-processing register	0	1	0	0	0	0	opcode				Rm / Rs		Rd / Rn			
Special data processing	0	1	0	0	0	1	opcode [1]		H1	H2	Rm		Rd / Rn			
Branch/exchange instruction set [3]	0	1	0	0	0	1	1	1	L	H2	Rm		SBZ			
Load from literal pool	0	1	0	0	1	Rd			PC-relative offset							
Load/store register offset	0	1	0	1	opcode			Rm		Rn		Rd				
Load/store word/byte immediate offset	0	1	1	B	L	offset				Rn		Rd				
Load/store halfword immediate offset	1	0	0	0	L	offset				Rn		Rd				
Load/store to/from stack	1	0	0	1	L	Rd		SP-relative offset								
Add to SP or PC	1	0	1	0	SP	Rd		immediate								
Miscellaneous: See Figure 6-2	1	0	1	1	x	x	x	x	x	x	x	x	x	x	x	x
Load/store multiple	1	1	0	0	L	Rn		register list								
Conditional branch	1	1	0	1	cond [2]				offset							
Undefined instruction	1	1	0	1	1	1	1	0	x	x	x	x	x	x	x	x
Software interrupt	1	1	0	1	1	1	1	1	immediate							
Unconditional branch	1	1	1	0	0	offset										
BLX suffix [4]	1	1	1	0	1	offset										0
Undefined instruction	1	1	1	0	1	x	x	x	x	x	x	x	x	x	x	1
BL/BLX prefix	1	1	1	1	0	offset										
BL suffix	1	1	1	1	1	offset										

Figure 3.4: Formats of Thumb instruction set (from [9]).

match it before any non-word accesses occur (when the endianness is configurable and CP15 is implemented, bit7 of CP15 register 1 controls the endianness).

Memory that is used to hold programs and data should be marked as follows:

- Main (RAM) memory is normally set as cachable and bufferable.
- ROM memory is normally set as cachable, and should be marked as read only, so the bufferable attribute is not used and should be 1.

3.4.1 Write Buffers

Some ARM implementations incorporate a merging write buffer that subsumes multiple writes to the same location into a single write to main memory. Furthermore, some write buffers re-order writes, so that writes are issued to memory in a different order to the order in which they are issued by the processor.

For writes to bufferable areas of memory, memory aborts can only be signaled to the processor as a result of conditions that are detectable at the time the data is placed in the write buffer. Conditions that can only be detected when the data is later written to main memory (such as a parity error from main memory) must be handled by other methods (typically by raising an interrupt).

3.4.2 Caches (from [11])

Caches hold copies of the contents of memory locations. In general, these are memory locations that have been loaded from recently. These copies are automatically used in preference to off-chip memory.

Caches only give an advantage if the cached memory locations are used again. In a real system this is very common, for example:

- instruction loops, and
- frequently referenced data.

Cache operation is transparent to the programmer. However, a programmer must initialize the core to specify what off-chip memory locations are to be cached.

Harvard Architecture

ARM's cached Harvard cores have separate instruction and data caches, but use the same bus to access external memory. A programmer can define the properties of memory regions for data and instructions separately. For ARM920T, there is a 16 KB instruction memory and a 16 KB data memory.

Cache Between ARM and Coprocessor

ARM processors do not normally support cache coherence between the ARM and other system bus masters. Bus snooping is not supported. If memory data are to be shared between multiple bus masters without taking special software measures to ensure coherency, then the data must be mapped as:

- uncachable to ensure that all reads access main memory, and
- unbufferable to ensure that all write access main memory.

3.5 ARM Developer Suite (ADS) (from [12])

ADS consists of a suite of applications, together with supporting documentation and examples, that enable us to write and debug applications for the ARM family of RISC processors. We can use ADS to develop, build, and debug C, C++, and ARM assembly language programs.

The ADS toolkit consists of the following major components:

- command-line development tools,
- GUI development tools,
- utilities, and
- supporting software.

Figure 3.5 shows the flow of ARM tools. C or assembly source are input first, then the object code will be linked. Finally, the program will be loaded into the development board or the ARMulator.

Command-Line Development Tools

The following command-line development tools are provided:

- armcc** The ARM C compiler. The compiler is tested against the Plum Hall C Validation Suite for ANSI conformance. It compiles ANSI source into 32-bit ARM code.
- armcpp** This is the ARM C++ compiler. It compiles ISO C++ or EC++ source into 32-bit ARM code.
- tcc** The Thumb C compiler. The compiler is tested against the Plum Hall C Validation Suite for ANSI conformance. It compiles ANSI source into 16-bit Thumb code.
- tcpp** This is the Thumb C++ compiler. It compiles ISO C++ or EC++ source into 16-bit Thumb code.
- armasm** The ARM and Thumb assembler. This assembles both ARM assembly language and Thumb assembly language source.
- armlink** The ARM linker. This combines the contents of one or more object files with selected parts of one or more object libraries to produce an executable program. The ARM linker creates ELF executable images.
- armsd** The ARM and Thumb symbolic debugger. This enables source level debugging of programs. We can single-step through C or assembly language source, set breakpoints and watchpoints, and examine program variables or memory.

GUI Development Tools

The following Graphical User Interface (GUI) development tools are provided:

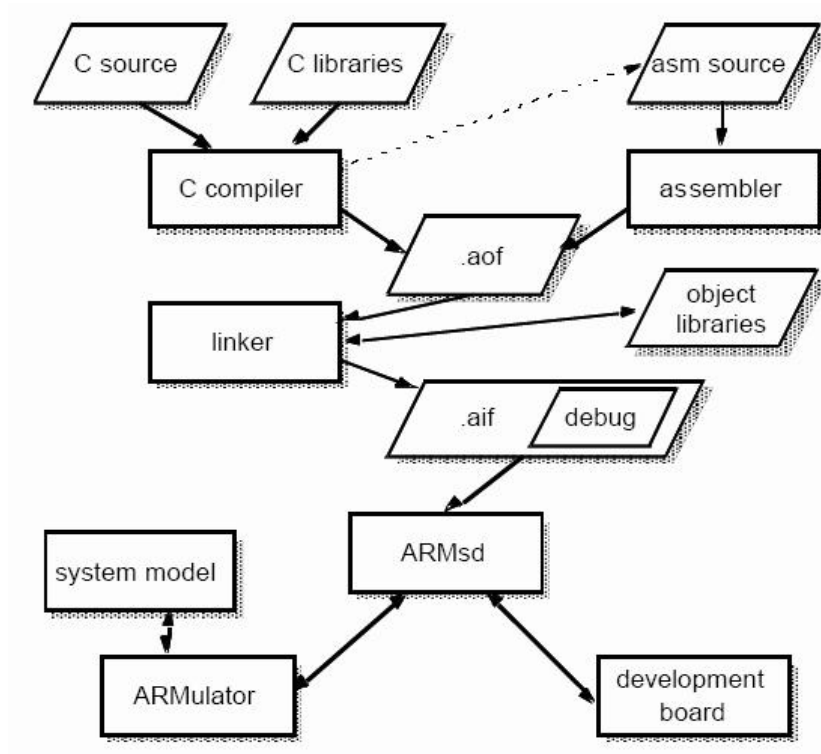


Figure 3.5: ARM tools for developers.

AXD

The ARM Debugger for Windows and UNIX. This provides a full Windows and UNIX environment for debugging C, C++, and assembly language source.

CodeWarrior IDE

The project management tool for Windows. This automates the routine operations of managing source files and building software development projects. The CodeWarrior IDE is not available for UNIX.

Utilities

The following utility tools are provided to support the main development tools:


- fromELF** The ARM image conversion utility. This accepts ELF format input files and converts them to a variety of output formats, including plain binary, Motorola 32-bit S-record format, Intel Hex 32 format, and Verilog-like hex format. fromELF can also generate text information about the input image, such as code and data size.
- armprof** The ARM profiler displays an execution profile of a simple program from a profile data file generated by an ARM debugger.
- armar** The ARM librarian enables sets of ELF format object files to be collected together and maintained in libraries. The programmer can pass such a library to the linker in place of several ELF files.

Supporting Software

The ARMulator support software is provided to enable the programmer to debug the programs, either under simulation, or on ARM-based hardware. It is an ARM core simulator which provides instruction-accurate simulation of ARM processors and enables ARM and Thumb executable programs to be run on non-native hardware. The ARMulator is integrated with the ARM debuggers.

Chapter 4

Analysis of Computational and Storage Complexity of MPEG-4 Framed-Based Video Decoder



Our implementation employs the public source MoMuSys as the base. In section 1, we introduce the MoMuSys software and show the profile. In section 2, we consider code size and memory usage reduction for implementation. And lastly in section 3, we discuss our approaches to code acceleration.

4.1 Introduction to MoMuSys

MoMuSys (Mobile Multimedia Systems) is project number AC098 in the ACTS (Advanced Communication Technology and Services) program, funded up to 50% from the European Commission. The rest is funded by the partners involved. The project started on September 1, 1995, and ran until the fall 1998. The prime project leader is Dirk Lappe from Bosch.

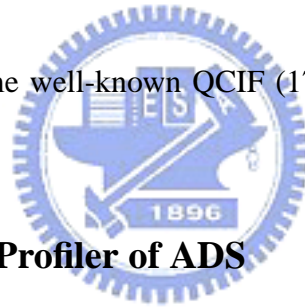
The MoMuSys donated its software for MPEG-4 main profile encoding and decoding to MPEG. Table 4.1 shows the functionalities that the MoMuSys software supports. The functionalities defined by MoMuSys conforms to the main profile of MPEG-4. However, to implement an MPEG-4 decoder on the ARM processor, the main profile appears too

complicated on first attempt. Therefore, we implement and optimize the simple profile first.

4.2 Complexity Analysis of MoMuSys Decoder

We consider two levels of complexity analysis. The first level is an operational analysis by actual decoding of video sequences. Because of the different characteristics of different sequences, the complexity figures of some decoder components, such as the motion compensator and the VLD, are statistically variable and not a set of fixed numbers. The second level is a low-level analysis, which is the ideal complexity computed by hand. At this level, time-critical blocks in the decoder block diagram (see Figure 4.1) are analyzed. But we overlook the overhead of some computations, especially that for address calculation.

In this section, we use some well-known QCIF (176x144) test sequences to do the analysis.



4.2.1 Profile Using the Profiler of ADS

In section 3.5, we have introduced the ARM Developer Suite (ADS) development tools for ARM processors. Now we employ the profile tools of ADS and VTune to do the first level analysis, where VTune is a software development tool for Intel processors [16]. The Intel VTune Analyzer helps to locate and remove software performance bottlenecks by collecting, analyzing, and displaying performance data from the system-wide level down to the source level. Since our PC contains Intel CPU, using VTune help us get accurate profiles.

The experiment environment of VTune is a laptop with a 1.7 GHz Pentium-M CPU and 768 MB of DDR RAM, running Windows-XP. The profiling results on VTune, in Table 4.2, is obtained from encoding and decoding 2 frames employing H.263 quantization with a fixed half quantization step size (QP) equal to 4. Note that the quantization step size affects the length of bitstream, so larger QP results in smaller bitstream size and reduce the required encoding and decoding time.

Table 4.1: Functionalities of MoMuSys

	Simple	Main	MoMuSys	Perference
Basic				
1. <i>I VOP</i>				
2. <i>P VOP</i>	V	V	V	
3. <i>AC/DC Prediction</i>				
4. <i>4MV Unrestricted MV</i>				Full search
Error resilience				
1. <i>Slice Resynchronization</i>	V	V	V	
2. <i>Data Partitioning</i>				
3. <i>Reversible VLC</i>				
Short header	V	V	V	
B-VOP		V	V	
Method 1/Method 2 quantization		V	V	
P-VOP based				
temporal scalability				
1. <i>Rectangular</i>		V	V	
2. <i>Arbitrary shape</i>				
Binary Shape		V	V	
Grey shape		V	V	
Rate control		V	V	TM5, VM4

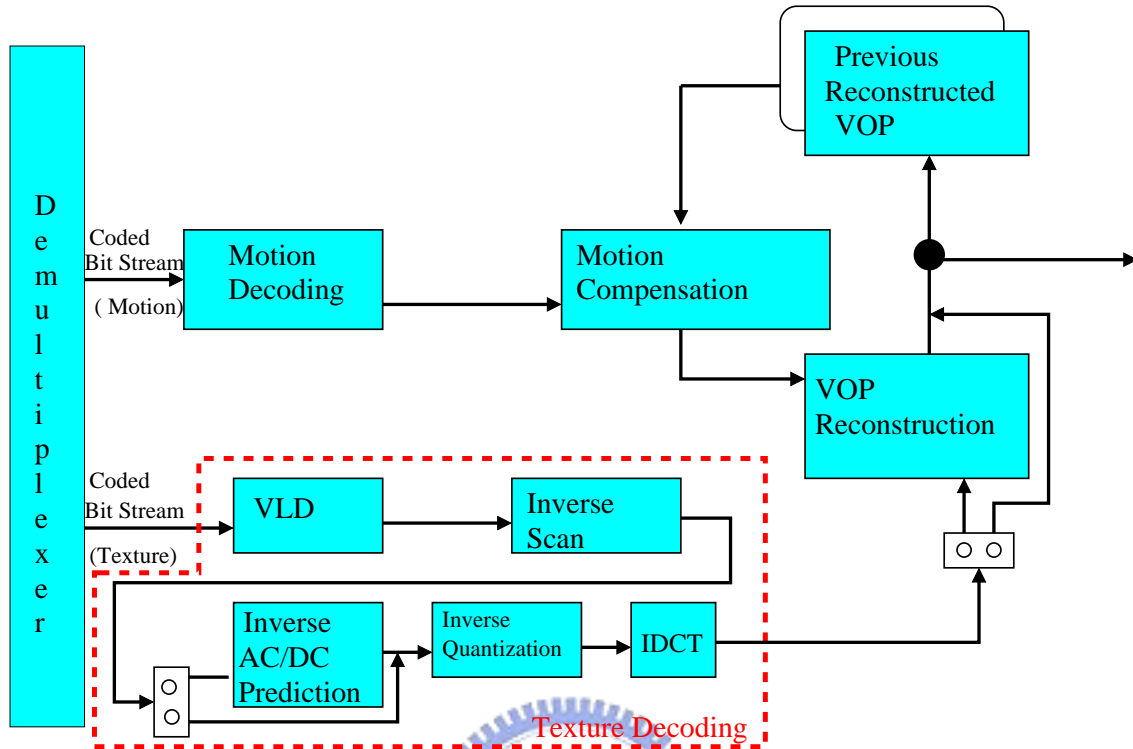


Figure 4.1: Block diagram of MPEG-4 frame-based video decoder [4].

We can see that the item “Others” occupies the most percentage of total time. Actually, most of “Others” is about writing output image. In the original code of MoMuSys, the output is written into files in yuv format. However, the instruction “fwrite” takes much time. It is included in “Others.” Writing the output into files is not necessary for every MPEG-4 application. For example, in some applications the output maybe shown on screen instead.

In Table 4.2, it is noted that the data are calculated for two frames. However, some functions, such as “DecodeMBVs” and “Motion Compensation,” are called for inter (P) frames only, and “DCACPrediction” is just for intra (I) frames. Therefore, the execution time of functions which are used for both I and P frames should be divided by two, when we want to compare the computational complexity of the MPEG-4 decoder.

The statistics on ADS are shown in Table 4.3 and we omit the components whose percentages are lower than 0.005%. Similar to Table 4.2, the item “Others” occupies the most percentage of total time and the percentages of other items are compatible between the two tables. In the table, we can find that the MoMuSys consumes more time on ARM

Table 4.2: Profile of Frame-Based MPEG-4 Decoding of QCIF Sequences on VTune
(from [15])

Function Name	stefan_qcif		grandmother_qcif	
	Clockticks	%	Clockticks	%
BitstreamAccess	1,695	1.35	1,865	1.91
DecodeVOLHeader	296	0.24	294	0.30
DecodeVOPHeader	26	0.02	23	0.02
DecodeMBHeader	495	0.40	264	0.27
DecodeMBMVs	1,544	1.23	69	0.07
DCACPrediction	2,584	2.06	2,621	2.69
BlockDequantH263	1,870	1.49	946	0.97
BlockIDCT	28,340	22.63	7,927	8.14
BlockInterpolation	1,170	0.93	1,165	1.20
Motion Compensation	8,066	6.44	7,203	7.40
Fill_VOP	424	0.34	413	0.42
Others	79,904	63.80	75,723	77.79
Total	125,244	100.00	97,348	100.00

processor. The reason is that the ARM processor is a fixed-point processor and it is very sensitive to the floating-pointing execution, such as IDCT, and file I/O operation.

Disregarding the item “Others,” we can see that the most time-critical components are “BlockIDCT” and “Motion Compensation” in both tables. The reason why IDCT consumes so much time is that the IDCT in the MoMuSys code is implemented in floating-point. Especially, the item “BlockIDCT” consumes more time on ARM processor relatively. Now we turn to the low-level analysis of these two time-critical items in the next subsection.

Table 4.3: Profile of Frame-Based MPEG-4 Decoding of QCIF Sequences on ADS

Function Name	stefan_qcif		grandmother_qcif	
	Clockticks	%	Clockticks	%
BitstreamAccess	1,412,708	0.44	524,648	0.26
DCACPrediction	2,086,955	0.65	2,098,591	1.04
BlockDequantH263	2,215,383	0.69	1,230,905	0.61
BlockIDCT	46,137,749	14.37	27,301,867	13.53
BlockInterpolation	1,733,778	0.54	1,735,374	0.86
Motion Compensation	5,939,794	1.85	3,934,859	1.95
Fill_VOP	706,354	0.22	706,257	0.35
Others	260,840,479	81.24	164,255,131	81.4
Total	321,069,930	100.00	201,787,632	100.00

4.2.2 Low-Level Computational Complexity Analysis

In this section, we consider detailed computational complexity for some time-critical functions. In the statistics of the following tables, the designation “data” in front of a dash indicates that the operation is associated with data values (memory contents), whereas the designation “mem” indicates that the operation is associated with memory addresses. The reason for distinguishing “data” and “mem” operations is that many processors treat these two types of operation differently.

Complexity Analysis of Motion Compensation

As Figure 4.2 shows, there are four steps to complete the motion compensation for luminance blocks. First, the reference frame is padded with 16 pixels around the whole frame for the motion vectors which point out of the frame. Second, the padded frame is interpolated by two. Third, according to the corresponding motion vectors, we can find the reference block data and get a compensated frame. Fourth, we add the decoded residual

data with the reference data and reconstruct the previous frame.

The complexity and memory requirement analysis of motion compensation for one frame is shown in Table 4.4. The data is calculated by hand in the ideal case. For example, data-load of interpolation comes from $(176 + 32) \times (144 + 32)$. We see that there are many memory accesses performed. That is, the memory load/store and address calculations occupy most of the computational complexity for motion compensation. So, it is necessary to lower the amount of memory accesses in our implementation.

About the storage requirement for luminance motion compensation, Table 4.4 also shows the memory requirement for every operation in motion compensation. The total memory space required is 253 KB, regardless of any memory-sharing skill. Note that we consider the forward predicted P-VOP only. Details of the storage requirement are as follows:

- 25,344 bytes for the previous decoded frame,
- 36,608 bytes for the padded frame,
- 146,432 bytes for the interpolated frame,
- 25,344 bytes for the motion-compensated reference frame data, and
- 25,344 bytes for the residual frame data.

We can see that the storage requirement is large and inefficient. Referring to the decoder block diagram (see Figure 4.1), the previous decoded frame is the result of motion compensation and the residual frame is the output of texture decoder. The storage requirements of the previous decoded frame and the residual frame are inevitable. We will discuss the methods of reducing the memory requirement of the padded frame and the interpolated frame in chapter 5.

Moreover, the computational complexity and storage requirement of chrominance motion compensation are listed in Table 4.5. Note that the computational complexity analysis is for the 4:2:0 format.

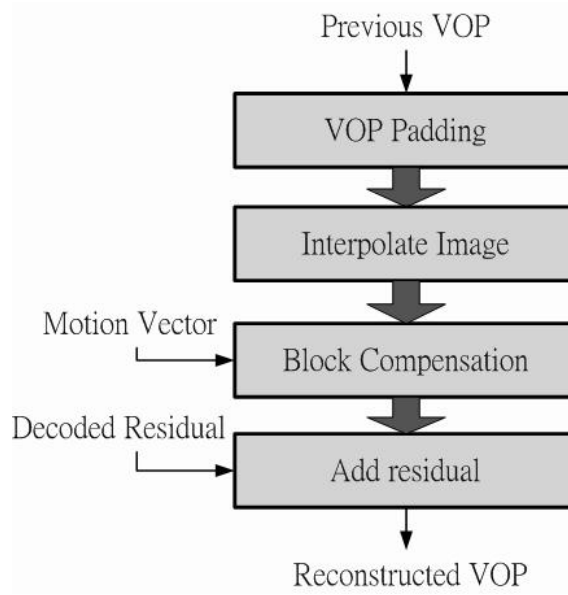


Figure 4.2: Flow of motion compensation.

Since the memory requirement and the amount of computation for addresses are very large, we have to analyze the characteristics of motion compensation and find some techniques to rearrange or reuse some memory spaces for higher performance. We leave the discussion to the next chapter.

Complexity Analysis of Texture Decoding

The texture decoding steps after VLD involve inverse scan, inverse AC/DC prediction, inverse quantization (or dequantization), and IDCT (inverse discrete cosine transform). However, the inverse scan and the inverse AC/DC prediction contain relatively small amount of computation. We concentrate on the dequantization and the IDCT below.

Table 4.6 shows the complexity of the MoMuSys software for dequantization and IDCT. Instead of carrying out a complexity analysis based on the algorithm as in the case of motion compensation, we analyze the MoMuSys code itself. In this table, note that there are many data-comparison instructions in dequantization and many data-add, data-multiply and data-load instructions in IDCT. We should pay attention to these parts in our implementation and optimization.

Table 4.4: Complexity of Luminance Motion Compensation for One QCIF Frame (from [15])

Operation	Padding	Interpolation	Find-Ref.	Add-Residual
data-add	0	181,507	0	25,344
data-shift	0	109,057	0	0
data-load	25,344	36,608	25,344	25,344
data-store	36,608	146,432	25,344	25,344
mem-add	36,608	146,432	26,136	25,344
mem-mult	0	0	396	0
Memory Req. (bytes)	61,952	146,432	25,344	25,344
Total Storage Requirement:				253 KBytes

4.3 Code Size and Memory Usage Reduction for Implementation

In this section, we discuss the code size and memory usage reduction before our acceleration.

4.3.1 Code Size Reduction

After compilation by the ADS compiler in release mode (-O3), the code size of MoMuSys is 595.52 KB and the total memory size (Code, RO, RW, and ZI data) is 647.34 KB. Although there is a 256 MB SDRAM on our platform, the instruction cache is only 8 KB. To reduce cache miss, we have to reduce the code size.

From earlier discussion, we know that there are many functionalities in MoMuSys

Table 4.5: Complexity of Chrominance Motion Compensation for One QCIF Frame (from [15])

Operation	Padding	Interpolation	Find-Ref.	Add-Residual
data-add	0	89,992	0	12,672
data-shift	0	54,530	0	0
data-load	12,672	18,304	12,672	12,672
data-store	18,304	73,216	12,672	12,672
mem-add	18,304	73,216	13,068	12,672
mem-mult	0	0	198	0
Memory Req. (bytes)	30,976	73,216	12,672	12,672
Total Storage Requirement:				126.5 KBytes

(see Table 4.1). Since we only implement the simple profile first, we can remove some functionalities and modify the code for our implementation. To retain the original code, we always use the `#ifdef` macros to do modification as shown in Figure 4.3. If we add the definition, `MAIN_PROFILE`, in the preprocessor of the compiler, we will get the original code without any change.

The files listed in Table 4.7 have the functionalities that we do not need. Thus we remove them completely. The method we use is to simply add `"#ifdef MAIN_PROFILE"` at the beginning of the file and add `"#endif"` at the end. Then, we will reduce a large amount of the code size and save much of the instruction memory. Some other files which are removed partly are not shown in this table.

After reducing the code size, our code size becomes 137.73 KB only and the total memory size (Code, Read-Only, Read-Write, and Zero-Initial data) is 167.89 KB. In conclusion, the MoMuSys code contains many functionalities and we only need a quarter of

Table 4.6: Complexity of Dequantization and IDCT for One 8×8 Block in MoMuSys

Operation	DeQuant	IDCT
data-comparison	320	0
data-add	192	544
data-mult	64	256
data-shift	128	0
data-load	256	576
data-floor	0	64
mem-add	0	64
mem-mult	0	64

the code size of the MoMuSys code for our simple profile implementation.

4.3.2 Memory Usage Reduction

The MoMuSys code was developed by many people. They use many “malloc” and “calloc” instructions to allocate memory spaces for utilization. That is, when they need an array or a matrix to record data, the “malloc” instruction will be employed to allocate an memory space and the “free” instruction will be employed to free the memory space in the end of the function.

Table 4.8 shows the greater part of the functions which have malloc or calloc instructions in the MoMuSys code. For our implementation, it is inefficient to allocate memory space so many times, because the operation takes much time. Our solution of reducing memory usage is to declare a global variable first and use macros to comment out the “malloc”, “calloc” and “free” instructions. Furthermore, there are many VOPs at the same execution time and occupy large memory spaces. In our work, we avoid repeated operations of memory allocation to get better performance.

```
#ifndef MAIN_PROFILE

    Original Code

#else

    Revised Code

#endif
```

Figure 4.3: Revised code using the #ifndef macros

It takes very much effort to remove all the instructions of memory allocation. So we focus on the part of code which allocates large memory spaces. However, there are some problems which are not easily solved. For example, the array named “A” is allocated by a function and pointed to a local pointer “B” in another function. At the end of the second function, “B” is freed but actually the freed array is “A”. When decoding the next frame or block, the array “A” is used again but the result is wrong because “A” is already freed. This kind of problem is hard to discover and hence we have spent lots of time dealing with it.

Finally, we use more fixed memory spaces (Zero-Initial data), and the speed performance is better.

Table 4.7: Files in MoMuSys That Are Not Needed for Simple Profile Implementation

In folder vm_dec	alp_dec_cae.c, alp_dec_grey.c, alp_dec_header.c, alp_dec_mc.c, alp_dec_mom.c, alp_dec_si.c, alp_dec_util.c, bin_ar_decode.c, concealment.c, drc_util_dec.c, mot_get_mvnum.c, newpred_d.c, sprite_dec_piece.c, sprite_dec_util.c
In folder vm_common	ac.c, alp_common_cae.c, alp_common_mc.c, alp_common_si.c, alp_common_util.c, BinArCodec.c, bitpack.c, boundary.c, computePSNR.c, context.c, deblock.c, decQM.c, do_bgc.c, download_filter.c, drc_util_common.c, dwt.c, dwt_aux.c, dwtmask.c, encQM.c, errorHandler.c, globalMC.c, idwt.c idwt_aux.c, idwtmask.c, imagebox.c, io_debug.c, io_sharp.c, io_yuv.c, mom_vo.c, msg.c, newpred_common.c, nr_util.c, PEZW_ac.c, PEZW_globals.c, PEZW_textureLayerBQ.c, PEZW_utils.c, post_filter.c, QMInit.c, QMUtils.c, quant.c, read_control_file.c read_image.c, sadct_blk.c, sadct_blk_kaup.c, sadct_blk_s_k.c, sadct_fprintf_mat.c, sadct_init.c, sadct_momusys.c, sadct_momusys_kaup.c, sadct_momusys_s_k.c, sadct_nrutil.c, sadct_vec.c, sadctq_blk.c, seg.c, shape.c, ShapeCommon.c, ShapeDeCoding.c, ShapeEnCoding.c, ShapeUtil.c, sprite_util.c, stats.c, text_quant_mat_def.c, Utils.c, vm_midproc.c, vm_vop_bound.c, wavelet.c, write_image.c, wvtPEZW.c, wvtpezw_tree_decode.c, wvtpezw_tree_encode.c, wvtpezw_tree_init_decode.c, wvtpezw_tree_init_encode.c, ztscan_dec.c, ztscan_enc.c, ztscanUtil.c


Table 4.8: Functions with Memory Allocation Instructions

function name	variable name	size (bytes)
BitstreamOpen	stream	2,092
AllocVop	vop	6,820
AllocImage	Image	28
	image.data	4
allocate_trace_file	trace	128
DecodeVopCombinedShapeTextureIntraErrRes	mblock	1,536
	DC_store	2,376
	slice_nb	396
	header_data	6,732
DecodeCombinedPacketInfoIntraErrRes	mblock	1,536
DecodeVopCombinedMotionShapeTextureInterErrRes	mblock	1,536
	DC_store	2,376
	slice_nb	396
	header_data	6,732
DecodeCombinedPacketInfoInterErrRes	mblock	1,536
*clone_bitstream	stream1	2,092
main	trace	128
DecodeVopNonScalable	mb_type	396
	mvda	396
SallocVop	vop	6,820
SallocVol	vol	6,196

Chapter 5

MPEG-4 Video Decoder

Implementation and Optimization for ARM9



We now consider the optimization of our implementation of the MPEG-4 frame-based video decoder on ARM9. The optimization techniques can be divided into two categories, algorithmic level and assembly/architecture level optimization. We focus on the functions which consume more cycles. At the end of this chapter, we give the experiment results.

5.1 Algorithmic Optimization

Algorithmic level optimization involves changing the computation flow in C language to reduce the computations. We modify our algorithms wherever possible to reduce the computations.

5.1.1 Algorithmic Optimization for Blocks in Intra Frames with Null AC Coefficients [15]

Analysis of DC/AC Prediction

In the process of intra prediction, the DC and AC values would be predicted from the previous decoded blocks. Then, after DC and AC reconstruction, inverse quantization and IDCT will be done for each intra block. However, many intra blocks have a non-zero DC coefficient but all zero AC coefficients because the property of DCT is that it concentrates signal energy in lower frequency coefficients. In other words, if we can make sure that there is only a DC coefficient decoded from the bitstream, the corresponding output block data can be obtained with copying the DC component to the entire block, and such property is illustrated in Figure 5.1. There are different methods to skip the prediction and transform, and we introduce the implementation techniques and show the analysis and simulation results in the following.

Check Skipped Blocks Using CBP and ACPred Flag

In the macroblock header, there are two parameters which contain important information for reducing the computation. The first one, Coded Block Pattern (CBP), gives information about blocks in macroblock being variable-length coded or not. It is a set of six bits, each of which representing one block in a macroblock. The status of the bits shows

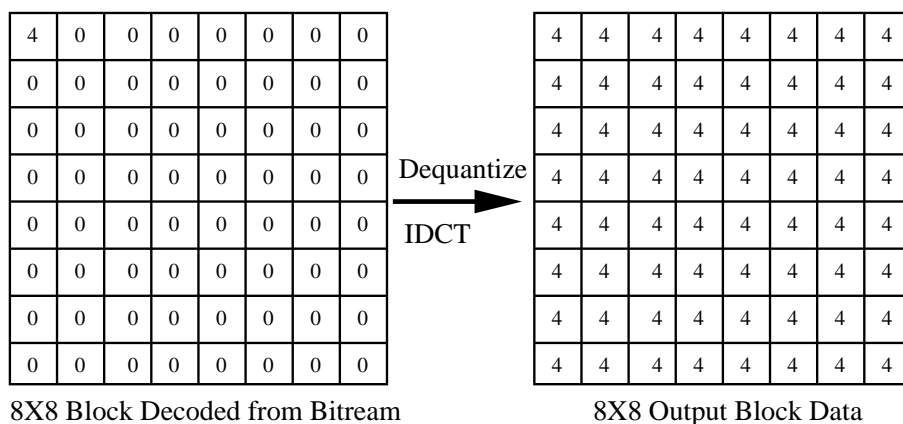


Figure 5.1: DC spreading from decoded coefficient to output block (from [15]).

whether the block is coded or not. If all coefficients of the block are zero, that block is not coded. The second, `ACPred_Flag`, tells us about the existence of AC prediction.

We choose some test sequences to compare the proportion of blocks that can be skipped. The simulation is done on PC with 90 frames for each sequence and these frames are all encoded in intra type. The simulation results are listed in Table 5.1.

In Table 5.1, we see that the percentage of skipped blocks is not very high. The reason that the simulation results is not as what we expected is due to the parameter `ACPred_Flag`. The `ACPred_Flag` is set to 1 if there is any block in an MB predicted with AC coefficients. In other words, if any block of the 6 blocks of the macroblock has AC prediction, the `ACPred_Flag` will be 1 but the rest of blocks whose AC values are zero will still do AC prediction. We cannot skip some blocks with DC component only but nonzero `ACPred_Flag`. Therefore, we should improve our method in finding the blocks that can be skipped.

Check Skipped Blocks After AC Prediction

Since the previous method was not precise enough, we did not skip all the blocks whose AC coefficients are all zero. In order to get higher precision, we add a check after the prediction of AC coefficients is completed. Similar to the previous method, we still have to check CBP. If the corresponding bit in CBP is zero, we can skip this block because all the AC predicted coefficients are zero.

Consequently, we can further find out all the possible blocks to be skipped, but the effort also increases because more conditions are checked. We again do a simulation on PC to get the percentage of skipped blocks in 90 intra-encoded frames. The simulation results are listed in Table 5.2.

Compared to Table 5.1, we can see that the percentage of skipped blocks gets higher with the aid of the new check. The test sequence “Grandmother_ycif” becomes the one which has the most skipped blocks. Consequently, the performance of this optimization should be highly related with the area of smooth region in each sequence.

Table 5.1: Number of Skipped Blocks in 90 Intra Frames (Check CBP and ACPred_Flag Only) (from [15])

Test Seqs. (QCIF)	Total Block No.	Skipped Block No.	%
grandmother	53,460	4,106	7.78
stefan	53,460	2,041	3.82
foreman	53,460	8,343	15.61
akiyo	53,460	6,574	12.30
mobile	53,460	1,422	2.66
football	53,460	5,568	10.42

Table 5.2: Number of Skipped Blocks in 90 Intra Frames with Further Check After AC Prediction (from [15])

Test Seqs. (QCIF)	Total Block No.	Skipped Block No.	%
grandmother	53,460	15,795	29.55
stefan	53,460	4,679	8.75
foreman	53,460	10,976	20.53

Optimization Result

Table 5.3 shows the optimization result. Obviously, the performance varies from one sequence to another. Moreover, the percentage of speedup on ARM9 is less than percentage of skipped blocks. The reason why the speedup is not much can be regarded as a demonstration of the Ahmdahl's Law [21]. That is, we only reduced the computations of inverse quantization and IDCT, and other parts like AC_DC_Prediction are not optimized. Besides, checking the conditions is the overhead that takes a few cycles.

In conclusion, the above algorithmic optimization for intra-frame decoding is severely limited by the characteristics of the test sequences. Furthermore, we can take the advan-

Table 5.3: Execution Time of Intra Frame Decoding on ARM9

Test Seqs. (QCIF)	Execution Time (cycles)				
	Original	CBP and AC_Pred_flag Checked	Speedup (%)	AC Prediction also Checked	Speedup (%)
grandmother	15,016,953	14,147,178	5.79	13,349,972	5.64
stefan	18,314,760	17,601,594	3.89	17,407,585	1.10
foreman	15,199,897	14,051,133	7.56	13,837,714	1.52

tage of ARM architecture to improve the performance. The architectural optimization methods will be introduced later.

5.1.2 Algorithmic Optimization for Null Residual Blocks of P-frames

Check Skipped Blocks for Null Residual Blocks

Similar to the optimization for intra-encoded frames, we want to simplify the decoding flow for the blocks whose residuals are all zero. Therefore, we again do some analysis on PC to check how many blocks can be skipped.

In the MB header of P-frame, the information, Coded Block Pattern (CBP), indicates whether the residual blocks are variable-length encoded or not. That is, if the corresponding bit in CBP is zero, the block is a null residual block. Therefore, we can check the CBP to see if the dequantization and IDCT can be skipped. For analysis, we still encode 90 frames with the first frame intra-encoded. The data listed in Table 5.4 are obtained from the statistics of the 89 inter-encoded P frames in each sequence.

In Table 5.4, the simulation results tell us that the less motion the test sequence has,

Table 5.4: Number of Skipped Blocks in 89 P-Frames

Test Seqs. (QCIF)	Total Block No.	Skipped Block No.	%
grandmother	52,866	42,746	80.86
stefan	52,866	16,495	31.20
foreman	52,866	25,590	48.41

the more blocks we can skip.

Optimization Result

Table 5.5 shows the average execution time for the test sequences with 89 frames. Compared with Table 5.4, we see that the percentage of speedup is quite related with the percentage of skipped blocks. Nevertheless, the performance of this optimization is still limited by Amdahl’s Law [21]. In conclusion, the above algorithmic optimization for P-frame decoding, similar to intra-frame decoding, is also limited by the characteristics of the test sequences. For further optimization in P-frame decoding, we consider the functionality of each process and try to reduce their computations. We will discuss the methods for optimization in the following sections.

5.1.3 Optimization for Image Interpolation and Padding

Analysis of Image Interpolation

Interpolation of each macroblock is necessary because the motion vector may be a non-integer number. In the original decoding process of motion compensation (see Figure 4.2), we interpolate the whole padded image and multiply the motion vector by 2. However, interpolation of the whole image consumes many cycles on ARM, but the effort is wasted where the motion vector is an integer. The other problem is that the storage requirement is 146,432 bytes which is large for our implementation. Therefore, we propose a block-based interpolation method, done only when the motion vector is not an integer.

Table 5.5: Execution Time of Inter (P) Frame Decoding on ARM9

Test Seqs. (QCIF)	Execution Time (cycles)		
	Original	CBP Checked	Speedup (%)
grandmother	18,600,472	12,428,187	33.18
stefan	20,837,319	19,004,118	8.80
foreman	19,114,456	16,360,197	14.41

Under the above approach, interpolation can be divided into 4 categories as follows, where MV_x is the horizontal motion and MV_y is the vertical motion for a block:

- Both MV_x and MV_y are integer numbers.
- MV_x is a half-integer number and MV_y is a integer number.
- MV_y is a half-integer number and MV_x is a integer number.
- Both MV_x and MV_y are half-integer numbers.

If both MV_x and MV_y are integer numbers, we can avoid the interpolation process. Moreover, we can interpolate only the horizontal direction, only the vertical direction, or both according to the category of the block.

To see how much saving is possible, we count the amount of motion vectors which are half-integer in either the horizontal or the vertical directions. The results are listed in Table 5.6 (from [15]). In Table 5.6, “Both” means that both the horizontal and the vertical motion are fractional. “ MV_x ” and “ MV_y ” mean that the motion vector is fractional only in horizontal and vertical direction, respectively.

From Table 5.6, we can also understand more about the different sequences, in particular directions of motion. Moreover, we see that more than 50% of interpolation can be avoided in four of the six test sequences. Thus, if we check the characteristics of the motion vectors before luminance and chrominance motion compensation of each blocks, many computations can be saved. Then the motion compensation flow becomes as shown in Figure 5.2.

Analysis of Padding

The objective of padding is to get more accurate motion estimation. Padding operation is necessary for the motion vectors which points out of the frame. However, it spends much time on repetitively copying the edge values to the exterior regions and the storage requirement of the padded frame is 36,608 bytes which is large for our implementation.

Table 5.6: Analysis of Necessary Interpolation (from [15])

Bitstream (QCIF)	Total MV Number	Half-integer MV							
		Total	%	Both	%	MV _x	%	MV _y	%
grandmother	18,204	2,064	11.34	550	3.02	497	2.73	1,017	5.59
stefan	33,744	15,385	45.59	1,954	5.79	10,478	31.05	2,953	8.75
foreman	34,128	15,585	45.67	4,658	13.65	5,994	17.56	4,933	14.45
akiyo	13,552	1,225	9.04	120	0.89	144	1.06	961	7.09
mobile	35,192	21,663	61.56	1,697	4.82	15,933	45.27	4,033	11.46
football	34,604	27,031	77.23	11,164	32.26	9,198	26.58	6,669	19.27

Moreover, in the original MoMuSys code, the frame which need to be padded is copied to the center of another bigger frame and padded latter. The reason of this operation is that the original frame size and the order of pixels are fixed in the MoMuSys code, so it is necessary to do the copy operation. Consequently, the overhead of calculating addresses is very considerable. If we can avoid copying the whole image, or even reducing the padding times, the performance will be more better.

Our approach is to skip the padding process and check if the target pixel is outside the frame. If it is, then we use the value of the edge pixel to do motion compensation. Take Figure 5.3 as an example. If the target pixel is at one of the positions marked a', we use the value of position a. Similar is the case for b' and b, h' and h, etc. Hence, we can totally skip the padding operation at the cost of some computations in block compensation. Finally, the flow of motion compensation becomes as shown in Figure 5.4.

Experiment Results

Based on the above analysis, we know that both interpolation and padding are time-critical processes because they are pixel-by-pixel operations. Their execution times for each P-frame decoding and storage requirements are shown in Table 5.7. For optimization, we

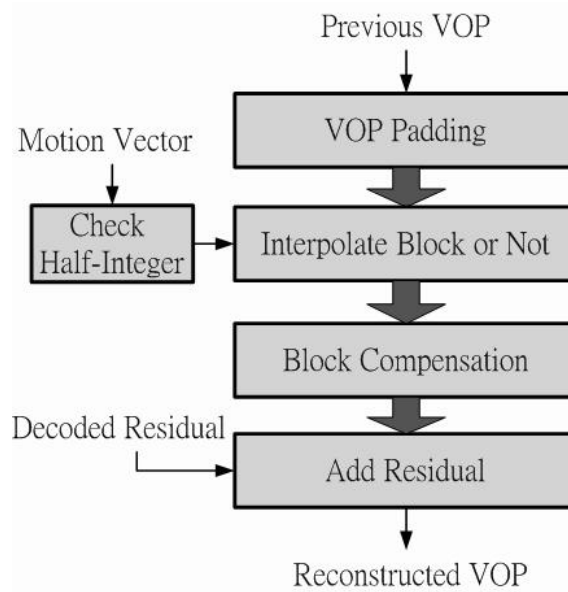


Figure 5.2: Modified flow of motion compensation with optimized interpolation.

Table 5.7: Execution Time and Storage Requirement of Image Interpolation and Padding on ARM9

Operation	Time (cycles)	Storage (bytes)
Interpolation	1,184,399	146,432
Padding	1,534,275	36,608
Total	2,718,674	183,040

alter the decoding flow to be as in Figure 5.4 and further optimize our code using the following methods:

- take the computations out of the loops as much as possible, and
- changing division operations into shift operations.

Note that the methods listed above are common optimization methods for programming. They are good for our design because the loops of interpolation and padding are large in number of cycles. Take Figure 5.5 as an example. This is a double for-loop and the pixels in the vertical direction need to be padded. We can see that the computations

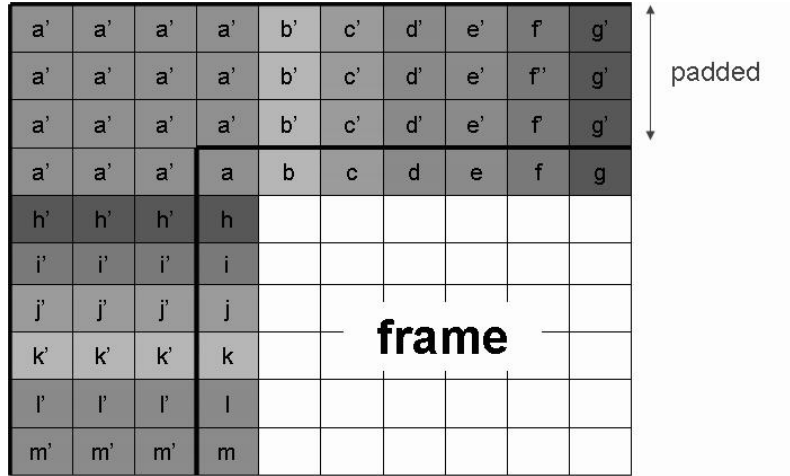


Figure 5.3: Example of padding in the upper-left corner of a frame.

for the variable “destination” is out of the second loop and thus we can save some computations. In other words, if the computations for the variable “destination” is in the second loop, the same computations will be done 8 times. Moreover, the interpolation is done by a shift operation instead of dividing by 2.

With our approach, the execution times of interpolation and padding are reduced, and the storage requirements are totally saved. Meanwhile, the time for calculating addresses is also reduced because the size of the previous frame is the same as that of the current frame. Therefore, the result of address calculation (partly) can be applied to both frames. The simulation results are listed in Table 5.8 and we can find that the speedup is very significant. Furthermore, the speedup of grandmother_qcif is more than other sequences. The reason why the speedup of different sequences is different is that their motion vectors are quite different in type. In grandmother_qcif, there is no motion vector which points outside the frame, and the motion of the sequence is relatively little that the number of fractional motion vectors is fewer. In conclusion, the speedup by the proposed method is also related to the characteristics of the test sequence. For further optimization, we focus on the modes of motion compensation and discuss our approach in the next section.

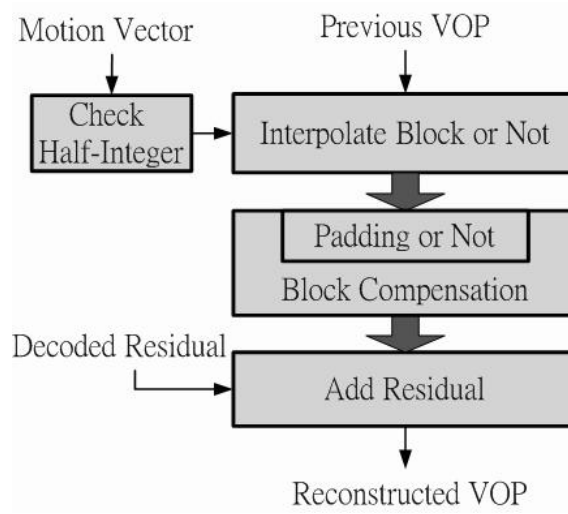


Figure 5.4: Modified flow of motion compensation with optimized padding.

Table 5.8: Execution Time of P-Frame Decoding on ARM9 After Modification of Interpolation and Padding

Test Seqs. (QCIF)	Execution Time (cycles)		
	Original	Optimized	Speedup (%)
grandmother	12,428,187	5,760,143	53.65
stefan	19,004,118	12,528,558	34.07
foreman	16,360,197	9,829,931	39.92

5.1.4 Optimization for Motion Compensation

Analysis of MB

Each macroblock (MB) in a frame has a compensation mode, which can be SKIPPED, INTER16, or INTER8. Their details are as follows:

- **SKIPPED mode:** The motion vector is (0,0) for the macroblock. We only take the macroblock in the same position in the previous frame as the reference block.
- **INTER16 mode:** The motion vector is not (0,0) for the macroblock. We obtain the reference block according to the motion vector.

```

for(n = 0; n < 16; n++)
{
    destination = comp->y_chan->f + (y+n) * width + x;

    for(m = 0; m < 16; m++)
    {
        x_rec = x + xint + m;
        y_rec = y + yint + n;
        y_rec2 = y_rec + 1;

        if(x_rec < 0)
            x_rec = 0;
        else if (x_rec >= (width - 1))
            x_rec = width - 1;
        if(y_rec < 0)
        {
            y_rec = 0;
            y_rec2 = 0;
        }
        else if (y_rec >= (height - 1))
        {
            y_rec = height - 1;
            y_rec2 = height - 1;
        }

        *(destination + m) = (*(rec_prev->y_chan->f + y_rec * width + x_rec) +
            *(rec_prev->y_chan->f + y_rec2 * width + x_rec) + 1 - rounding_control) >> 1;
    }
}

```

Figure 5.5: Example code of interpolation and padding.

- INTER8 mode: There are 4 motion vectors, one for each 8×8 block in the macroblock. We compensate the 4 blocks according to their motion vectors.

In the MoMuSys code, whatever the mode is, the computation is done according to what is needed for the INTER8 mode for regularity and simplicity. That is, we always compensate the 4 blocks in the macroblock according to their motion vectors, which may be (0,0) for SKIPPED mode. However, this consumes many cycles in calculating the addresses and the loop overhead is that for 4 double for-loops.

Mode Splitting and Experiment Results

Our approach is to optimize each mode for motion compensation and do compensation according to the mode. Meanwhile, common optimization methods, such as loop unrolling, are also applied. Take Figure 5.6 as an example of SKIPPED mode. The inner loop of the double for-loop is unrolled and hence the loop overhead is reduced. We can see that the code of dealing SKIPPED mode is very simple. If we use INTER8 mode to handle the MB in SKIPPED mode, the time we spend will be much.

Checking the modes and larger code size are small overheads and finally we will get

better performance from the optimization. The simulation result is shown in Table 5.9 and we can find that there is again a higher speedup in `grandmother_qcif`. The reason for the phenomenon is that the more blocks in skipped mode, the better performance from this approach.

In conclusion, we have optimized the algorithm of interpolation, padding, and motion compensation. For further optimization, we should employ architectural features on our design and we discuss the methods in the next section.

5.2 Assembly/Architecture Level Optimization

5.2.1 Loop Overhead Reduction

The assembly code generated by the compiler of ADS is inefficient for for-loops. Figure 5.7 shows an example code generated by the compiler. We can find that there are many branch instructions in the loop. However, there may be only one operation in the loop, such as ADD or MUL. The overhead is so large that we have to optimize the overhead of for-loops when we write the assembly code.

Instinctively, we write the program as Figure 5.8 shows. We use CMP instruction to test the termination condition of the loop. That is, the flags in the condition register (CPSR) will be set after CMP instruction and we use BGE instruction to do branch operation when the value in the register is greater than or equal to the constant value. However,

Table 5.9: Execution Time of P-Frame Decoding after Optimization of Motion Compensation on ARM9

Test Seqs. (QCIF)	Execution Time (cycles)		
	Original	Optimized	Speedup (%)
grandmother	5,760,143	5,281,289	8.31
stefan	12,528,558	12,486,879	0.33
foreman	9,829,931	9,800,037	0.30

```

if (mode == MBM_SKIPPED)
{
    for (n = 0; n < 16; n++)
    {
        temp = (y+n) * width + x;
        destination = comp->y_chan->f + temp;
        source = rec_prev->y_chan->f+ temp;

        *(destination) = *(source);
        *(destination +1) = *(source +1);
        *(destination +2) = *(source +2);
        *(destination +3) = *(source +3);
        *(destination +4) = *(source +4);
        *(destination +5) = *(source +5);
        *(destination +6) = *(source +6);
        *(destination +7) = *(source +7);
        *(destination +8) = *(source +8);
        *(destination +9) = *(source +9);
        *(destination +10) = *(source +10);
        *(destination +11) = *(source +11);
        *(destination +12) = *(source +12);
        *(destination +13) = *(source +13);
        *(destination +14) = *(source +14);
        *(destination +15) = *(source +15);
    } /* for(n=...) */
}
else if (mode == MBM_INTER16)
{
    :
}
else if (mode == MBM_INTER8 )
{
    :
}

```

Figure 5.6: Example code of mode splitting.

there are 4 instructions in the loop and the overhead is also large.

After many attempts to reduce the loop overhead, we employ the instruction SUBS whose function is to compare the value of destination register with zero after subtraction. Figure 5.9 shows an example optimized code in our approach. In order to employ SUBS, we should always write count-down-to-zero loops and use simple termination conditions. By using conditional execution, the CMP instructions can be avoided when exiting loops. For loops where the loop count is large, reduction of even one instruction in the loop is a big advantage.

Finally, we can find that there are only 2 loop control instructions in the loop and the original ADD/CMP instruction pair are replaced by a single SUBS instruction.

```

32      for (i = 0; i < 8; i++) {
000082d4 [0xe3a05000] mov     r5, #0
000082d8 [0xe3550008] cmp     r5, #8
000082dc [0xaa00008b] bge     0x8510 ; (BlockIDCT + 0x2d4)
000082e0 [0xea000001] b      0x82ec ; (BlockIDCT + 0xb0)
000082e4 [0xe2855001] add     r5, r5, #1
000082e8 [0xeaffffa] b      0x82d8 ; (BlockIDCT + 0x9c)
      :
      :
62      }
0000850c [0xeaffff74] b      0x82e4 ; (BlockIDCT + 0xa8)

```

Figure 5.7: Assembly code for for-loops generated by the compiler of ADS.

```

      MOV r12, #0      ;i=0
loop
  :
  ADD r12, r12, #1    ;i++
  CMP r12, #9        ;i<8
  BGE Exit
  B loop

```

Figure 5.8: Our initial assembly code for for-loops.

5.2.2 Conditional Execution of Instructions

Conditional execution is an important feature of the ARM processor. In the above discussion, we have realized the advantage of using conditional execution to reduce the loop overhead. Moreover, conditional executions can be employed in many places, such as zero-checking of input or saturation.

Conditional Execution for Zero-Checking

In some operations like dequantization and IDCT, there are many multiplications inside. It is essential to reduce the usage of multiplications because it takes 3 cycles to perform a multiplication. In architecture level, our solution is to employ conditional execution to

```

      MOV r12, #7      ;i=7
loop2
  :
  SUBS r12, r12, #1   ;i--
  BGE loop2          ;i>=0
  B Exit

```

Figure 5.9: Optimized assembly code for for-loops.

check the value of the input. If the input is zero, we can skip the multiplication because the result of multiplication must be zero. Take Figure 5.10 as an example, if the result of addition or subtraction is zero, the multiplication followed will be skipped and it takes one cycle only.

This example is an ideal case. However, the common case is that there is no addition or subtraction before multiplication and we have to use CMP instruction to set the flag. When the input is not zero, the CMP instruction would be redundant and consume one cycle. Consequently, the time we saved becomes one cycle from two cycles. So, the performance is better only when the probability of zero-input is more than 50%.

Conditional Execution for Saturation

Referring to Figure 4.1, when we do VOP reconstruction, we need to add the compensated image and the residual. However, saturation between 0 and 255 is essential after the addition. Usually, the saturation can be done using 4 instructions, two CMP and two MOV(cond). In consideration of the addition, we can still use 4 instructions as Figure 5.11 shows. The ADDS in this example is to do addition and to compare the destination register with zero. Then we can save one CMP operation in the code. Because VOP reconstruction is a pixel-by-pixel operation, we save one CMP operation for every pixel. That is, we save 176×144 cycles for every P-frame. However, the real condition of VOP reconstruction is more complicated and we will discuss it in the next section.

5.2.3 Reduction of Memory Accesses using LDM and STM

In ARM processor, there are two powerful instructions for memory access, LDM and STM. The LDM instruction is used to load multiple words from increasing or decreasing

```

LDR    r10, [r14], #4      ;tmp1[0] = (tmp[0] + tmp[4]) * c4
ADDS   r2, r4, r5         ;tmp1[1] = (tmp[0] - tmp[4]) * c4
MULNE  r2, r10, r2        ;r10 = c4
SUBS   r3, r4, r5
MULNE  r3, r10, r3

```

Figure 5.10: Example of zero-checking.

```

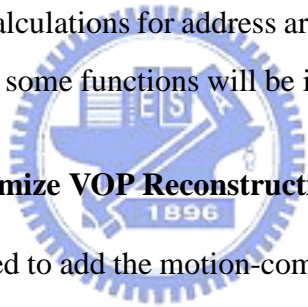
ADDS    r14, r11, r10    ; ADD r10 and r11, compare r14 with 0
MOVLT   r14, #0          ; Sat. to 0
CMPGT   r14, #255       ; Check >255
MOVGT   r14, #255       ; Sat. to 255

```

Figure 5.11: Saturation using conditional execution.

addresses into different registers and the STM instruction will store multiple data words in increasing or decreasing memory addresses. Usually, we use STM in the beginning of a function to push the register values into the stack and use LDM at the end of function to pop them.

We employ LDM and STM to load and store the input and the output, respectively. Similar to loop-unrolling, we make the 15 general purpose registers fully used and meanwhile the loop count is reduced. Furthermore, the greatest benefit from LDM and STM is that memory accesses and the calculations for address are much reduced. In the following discussion, the optimization for some functions will be introduced.



Using LDM and STM to Optimize VOP Reconstruction

For VOP reconstruction, we need to add the motion-compensated image and the residual. There are some conditions that we should take into account:

- The data type of the compensated image and the residual is 16-bit signed integer.
- The value of each pixel in the compensated image is among [0..255].
- The value of each pixel in the residual is among [-256..255].
- Allowing for program counter, stack pointer, loop counter and two input registers, we can use 11 general purpose registers.

Considering that the data type is short integer, if we use LDRSH instruction to load every short integer coefficient, the overhead will be very large because it takes 3 cycles to perform LDRSH. In our approach, LDM/STM is employed and the value in each register after using LDM will contain two coefficients. The higher halfword and the lower halfword contains a coefficient each. Unfortunately, the pixel value of the residual maybe

a negative number and hence we cannot add two registers directly. For example, if we add “0x00330033” and “0x00010001,” the result would be “0x00340034” and we can store the correct value into the memory directly. However, if the second value contains a negative number and becomes “0x0001FFFF”, the lower halfword will overflow after the addition and impact the value of the higher halfword.

In consideration of all conditions, we separate the additions for higher halfwords and lower halfwords. That is, we use shift operation to take out the halfwords we want. After addition and saturation, we pack the results, two halfwords, into a word and store them into the memory. Fortunately, there is a barrel shifter in front of the ALU of the ARM processor. Then, we can shift the second source register in the assembly instructions without any overhead.

The assembly code of VOP reconstruction, named AddClipImage, is given in Appendix A. We handle 8 pixels at the same time in a loop with the aid of LDM and STM instructions. The acceleration of VOP reconstruction is shown in Table 5.10.

Using LDM and STM to Optimize Regular Functions

For some regular functions, there are no calculations done on the input coefficients, but their functions are initializing an array or copying an array to another array, etc. In the following, our approach for these functions is introduced. The optimized assembly code are all given in the Appendix A.

- **Bzero and MBzero**

For some block-by-block operations, the array which contains the information must be initialized before the operation for the next block. We use the two functions, Bzero and MBzero, to fill the input array with zeros. Our optimization method is that we fill the registers with zeros and repeatedly use STMIA to store the zeros. Since the memory address would be auto-increased when using STMIA, we can save much time for calculating the address.

- **MB_clip**

At the end of intra frames decoding, each decoded block will be saturated. Each pixel in the block will be saturated to [0..255]. We use MB_clip to do saturation. Combining the method of conditional execution and the method of LDM/STM, we get better performance in saturating the blocks.

- PutBlock

In texture decoding process, each macroblock is split into 6 blocks, i.e., 4 luminance blocks and 2 chrominance blocks. At the end of texture decoding, we use PutBlock to pack the 6 blocks to a macroblock. This function checks the position of each block and puts them into a macroblock. Our optimizing method is that we load and store 8 coefficients at one time in the loop. After the loop is repeated 8 times, the operation is done.

- CopyImageI

When a frame is decoded, it is essential to save the current frame in the memory, which will be the source frame in motion compensation for the next frame. We use the function CopyImageI to achieve the goal. Since there is no mathematical calculation in the function, we repeatedly use LDMIA and STMIA to make each register fully used.

The simulation results of the functions above will be shown in the next section.

5.2.4 Experiment Results of Assembly/Architecture Level Optimization So Far

Table 5.10 shows the acceleration of the optimized functions. The frame size is QCIF for VOP reconstruction and CopyImageI, and the compiler is in release mode (-O3). We can find that the execution time for the functions is saved by more than 60%. Consequently, even compared with fully optimized compiler, the performance of regular functions can be highly improved in the assembly/architecture level. Moreover, some simple functions like Bzero which are called many times and consumes much power in the decoder should be optimized in architecture level as much as we can.

The speedup after optimizing VOP reconstruction is shown in Table 5.11. We can easily find that the time we save is the same for each sequence. The reason of this phenomenon is that VOP reconstruction is done once for every P-frame. So, the kind of optimization is unrelated with the characteristics of the test sequences.

Table 5.12 shows the simulation result after the regular functions are optimized. Note that we do the initialization in intra frame decoding, and we discard the time for initialization in P-frame decoding. As a result of the initialization, the speedup for intra frame is higher. Moreover, we can find that the percentage of speedup is significant because these functions are called many times in the decoding. Due to the regularities of the functions, the time we save for the test sequences are quite close. In conclusion, the optimization in architecture level is very important for the functions which contain large loops.

5.2.5 Optimization of IDCT

Studies of Efficient IDCT

In the past, there have been many methods developed for the fast computation of 2-D IDCT. The conventional approach is the row-column method, which requires 16 1-D IDCTs for the computation of an 8×8 IDCT [17]. Many fast algorithms for 2-D IDCT have been proposed, and one of them reduces the required 1-D IDCTs from 16 to 8 [17]. However, since the number of required registers is very big in this algorithm, it is not appropriate for implementation on ARM, which only has 15 general purpose registers.

For 1-D IDCT, we have to employ an efficient algorithm to implement it because the computational complexity of 1-D IDCT is very high. Furthermore, in the statistics shown before, the IDCT consumes considerable time in the total execution time. Since the computational complexity of direct implementation is very high, many fast algorithms for 1-D IDCT have been developed. Similar to the derivation from discrete Fourier transform (DFT) to fast Fourier transform (FFT), a fast cosine transform (FCT) is proposed in [18]. The computational complexity is compared in Table 5.13. Note that the computational complexity shown in the table is estimated for floating-point computation.

The ARM processor is a fixed-point processor. However, the transform coefficients

Table 5.10: Execution Time of Functions Optimized in Assembly/Architecture Level

Function	Execution Time (cycles)		
	Original	Optimized	Speedup (%)
VOP reconstruction	913,386	304,238	66.69
Bzero	391	111	71.61
MBzero	2,177	559	74.32
MB_clip	6,034	2081	65.51
PutBlock	594	209	64.81
CopyImageI	177,531	29,986	83.11

Table 5.11: Execution Time of P-Frame Decoding after Optimization of VOP Reconstruction on ARM9

Test Seqs. (QCIF)	Execution Time (cycles)			
	Original	Optimized	Speedup (cycles)	Speedup (%)
grandmother	5,281,289	4,672,140	609,149	11.53
stefan	12,486,879	11,877,730	609,149	4.88
foreman	9,800,037	9,190,888	609,149	6.22

used in [18] are cosine values. With the limited precision of fixed-point computation, the error will increase but we will get higher speed. Table 5.14 shows the cycles of floating-point and fixed-point 1-D IDCT using ADS compiler in release mode, and we can see that the time consumed by floating-point 1-D IDCT is much larger than the time consumed by fixed-point 1-D IDCT.

Table 5.12: Improvement after Optimization for Regular Functions

Test Seqs. (QCIF)	I-Frames (Cycles)			P-Frames (Cycles)		
	Original	Optimized	%	Original	Optimized	%
grandmother	13,349,972	10,515,382	21.23	4,672,140	3,861,477	17.35
stefan	17,407,585	14,608,605	16.08	11,877,730	10,797,637	9.09
foreman	13,837,714	11,016,629	20.39	9,190,888	8,079,428	12.09

Table 5.13: Comparison of Computational Complexity for 8-point IDCT

	Direct Form	FCT [18]	MoMuSys	Even_Odd FCT [19]
Multiplications	64	12	16	20
Additions	56	29	26	28

Implementation of IDCT on ARM9

The DCT and IDCT in MPEG-4 are defined as

$$F(u, v) = \frac{2}{N} C(u) C(v) \sum_{x=0}^{N-1} \sum_{y=0}^{N-1} f(x, y) \cos \frac{(2x+1)u\pi}{2N} \cos \frac{(2y+1)v\pi}{2N}, \quad (5.1)$$

$$f(x, y) = \frac{2}{N} \sum_{u=0}^{N-1} \sum_{v=0}^{N-1} C(u) C(v) F(u, v) \cos \frac{(2x+1)u\pi}{2N} \cos \frac{(2y+1)v\pi}{2N}, \quad (5.2)$$

Table 5.14: Cycles of Floating-Point and Fixed-Point 1-D IDCT Using ADS Compiler in Release Mode

Operation	Cycles
Floating-point 1-D IDCT	47,547
Fixed-point 1-D IDCT	3,017

where $u, v, x, y = 0, 1, 2, \dots, N - 1$, and

$$C(u), C(v) = \begin{cases} \frac{1}{\sqrt{2}}, & \text{for } u, v = 0, \\ 1, & \text{otherwise.} \end{cases}$$

Figure 5.12 shows the signal flow of the 1-D fixed-point IDCT algorithm used in MoMuSys. However, we can find that odd-indexed coefficients are rounded twice and each rounding introduces corresponding error. Figure 5.13 shows the signal flow of the even-odd decomposition algorithm, which provides a more accurate result because there is only one rounding operation for each coefficient. However, the even-odd decomposition algorithm contains more multiplications and each multiplication consumes 3 clockticks on ARM9. In consideration of the computational speed, we choose the 1-D IDCT algorithm of Figure 5.12. Compared with the original code of MoMuSys, the algorithm is not changed much but we use fixed-point computation and all architectural optimization methods to get higher computational speed. Our assembly code for IDCT is shown in Appendix A.

Based on the algorithm above and the optimization in architecture level, our IDCT takes 2,240 cycles to run once and the simulation result is shown in Table 5.15. Since IDCT is a time-critical function in decoding, the improvement of the function results in significant speedup of the total execution time.

Table 5.15: Improvement after Optimization of IDCT

Test Seqs. (QCIF)	I-Frames (Cycles)			P-Frames (Cycles)		
	Original	Optimized	%	Original	Optimized	%
grandmother	10,515,382	9,249,078	12.04	3,861,477	3,517,510	8.91
stefan	14,608,605	12,955,380	11.32	10,797,637	9,561,428	11.45
foreman	11,016,629	9,590,896	12.94	8,079,428	7,152,338	11.47

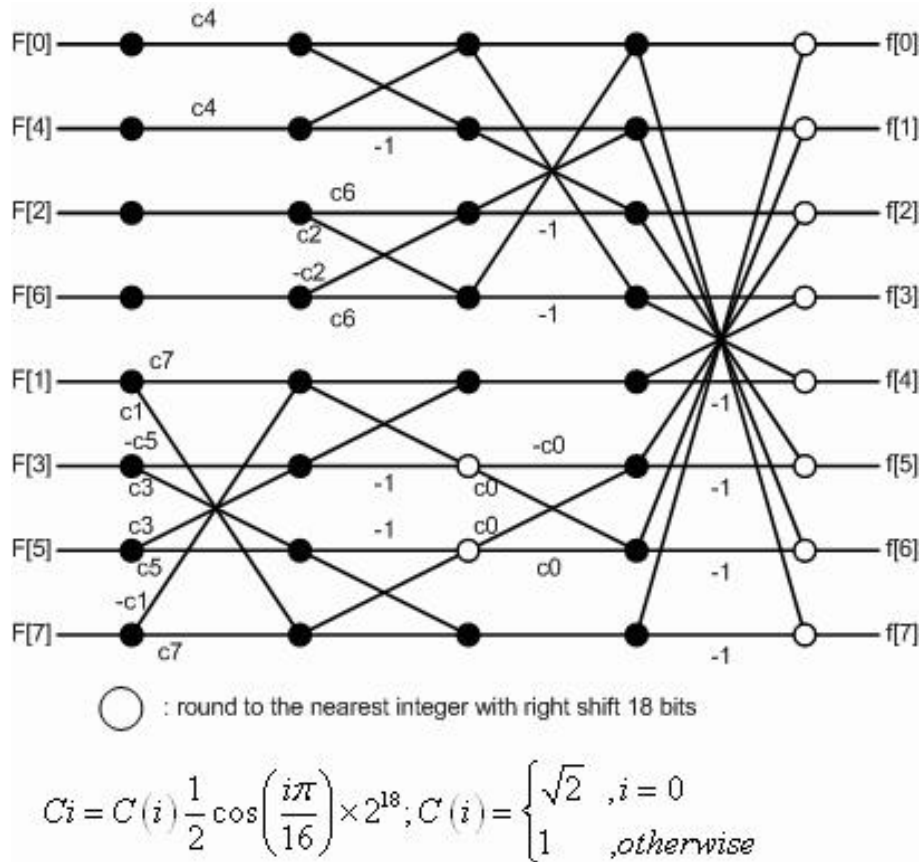


Figure 5.12: The IDCT algorithm used in MoMuSys [15].

5.3 Conclusion on Optimization

5.3.1 Overall Improvement after Optimization

In this chapter, we improved at both the algorithmic and the architecture levels. We first focused on algorithmic optimization and modified the decoding flow for the null texture or null residual blocks. Moreover, the processes of interpolation, padding, and motion compensation were optimized. After algorithmic optimization, we employed the architecture features, such as conditional execution, LDM/STM, and IDCT, to optimize the functions with large loops. The simulation results before and after optimization are listed in Table 5.16. We can see that there are about 30% speedup in intra decoding and more than 50% in P-frame decoding. Since the decoder is not optimized entirely, there are still some parts where we can optimize and we will do further analysis in the next

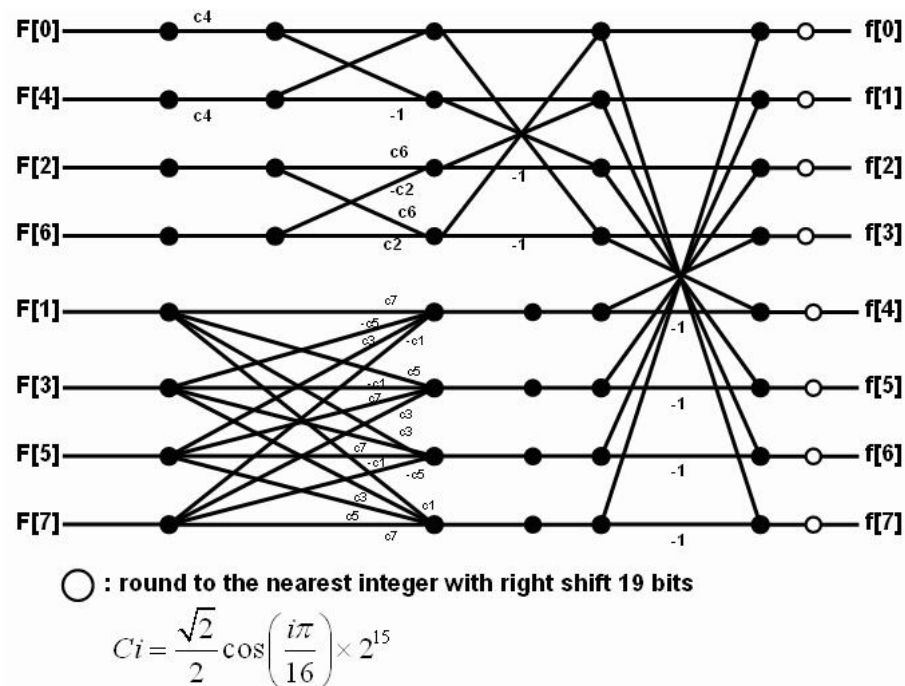
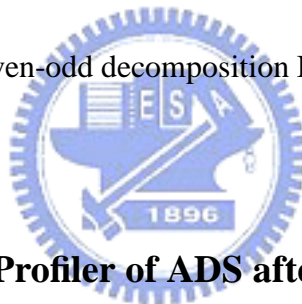


Figure 5.13: The even-odd decomposition IDCT algorithm [20].

section.



5.3.2 Profile Using the Profiler of ADS after Optimization

The analysis after optimization for each test sequence is shown in Table 5.17 and Table 5.18. Table 5.17 summarizes the profile of one intra frame decoding and Table 5.18 gives the average profile for 89 P-frames decoding. Note that:

- the profiles both exclude writing output files,

Table 5.16: Overall Improvement after Optimization

Test Seqs. (QCIF)	I-Frames (Cycles)			P-Frames (Cycles)		
	Original	Optimized	%	Original	Optimized	%
grandmother	15,016,953	9,249,078	38.41	18,600,472	3,517,510	81.09
stefan	18,314,760	12,955,380	29.26	20,837,319	9,561,428	54.11
foreman	15,199,897	9,590,896	36.90	19,114,456	7,152,338	62.58

- the items are named according to Figure 4.1,
- the cycles are calculated from the percentages which are only approximate numbers,
- there is no VOP reconstruction in intra frame decoding,
- the inverse scan has been included in DC/AC prediction in intra frame decoding,
- there is no DC/AC prediction in P-frame decoding, and
- the item “Others” mainly includes the initialization and some operations like clipping, putting an MB into a frame, putting zeros into an MB, or copying an array to another array.

In Table 5.17, we note that the test sequence, stefan_qcif, takes the most execution time among the three sequences. Unlike other test sequences, stefan_qcif has more rough regions which result in lower accuracy of intra prediction. Consequently, the bitstream size of stefan_qcif is bigger with more texture information and the decoding time becomes longer.

The code size after optimization is 117.75 KB, and the data memory size (Read-Only, Read-Write, and Zero-Initial data) is 110.06 KB. The program and data memory are saved more after optimization.

Since we have not optimized the processes of bitstream access, VLD, DC/AC prediction, and inverse quantization, we use the original code of MoMuSys to perform these processes in intra frame decoding. However, the execution time of the non-optimized codes occupies more than 50% of the total execution time. Finally, the performance in intra frame decoding is not good enough. We will discuss the reason in the next chapter.

In Table 5.18, we notice that the test sequence, stefan_qcif, again takes the most execution time. Due to the moving camera, there are many motion vectors for one P-frame and it results in much bitstream accessing time and decoding time. However, the execution time of motion compensation of foreman_qcif is more than stefan_qcif. The reason is that foreman_qcif contains more fractional motion vectors than stefan_qcif contains (refer to Table 5.6).

Table 5.17: Profile of Intra Frame Decoding after Optimization

Function Name	grandmother_qcif		stefan_qcif		foreman_qcif	
	Cycles	%	Cycles	%	Cycles	%
Bitstream Access	1,426,208	15.42	2,776,338	21.43	1,547,012	16.13
VLD	1,008,150	10.9	2,273,669	17.55	1,057,876	11.03
DC/AC Prediction	1,571,418	16.99	1,601,285	12.36	1,491,384	15.55
Inverse Quantization	935,082	10.11	1,381,044	10.66	1,016,635	10.6
IDCT	842,591	9.11	1,156,915	8.93	981,149	10.23
Others	3,465,630	37.47	3,766,129	29.07	3,496,841	36.46
Total	9,249,078	100	12,955,380	100	9,590,896	100

Since we have not optimized the processes of bitstream access, VLD, inverse scan, inverse quantization, and motion decoding, we use the original code of MoMuSys to perform these processes in P-frame decoding. However, 20% to 30% of the total execution time is consumed by the non-optimized processes. If we optimize the operations mentioned above, the speedup may be about 15%. Moreover, the item “Others” contains some regular functions which are not optimized. In case the whole decoder is optimized, the speedup can reach 50% possibly. In conclusion, we can have much better performance by optimizing the non-optimized parts of the decoder.

5.4 Comparison with Other Implementations

Since the MPEG-4 standard has been issued for several years, there are a few reports of implementations on other platforms. We compare our implementation with other implementations. The numerical comparison is listed in Table 5.19. Note that our data in the table comes from the average of 90 frames decoding in the best case.

Our main target for comparison is the implementation on ARM7TDMI [23]. Since the differences between ARM7TDMI and ARM9 are architecture and pipeline stages,

Table 5.18: Profile of P-Frame Decoding after Optimization

Function Name	grandmother_qcif		stefan_qcif		foreman_qcif	
	Cycles	%	Cycles	%	Cycles	%
Bitstream Access	148,791	4.23	1,220,994	12.77	454,889	6.36
VLD	100,953	2.87	1,008,731	10.55	334,014	4.67
Inverse Scan	155,474	4.42	531,615	5.56	399,100	5.58
Inverse Quantization	239,542	6.81	939,888	9.83	627,975	8.78
IDCT	263,110	7.48	899,730	9.41	675,896	9.45
Motion Decoding	17,236	0.49	54,500	0.57	58,649	0.82
Motion Compensation	676,769	19.24	1,274,538	13.33	1,397,567	19.54
VOP Reconstruction	215,975	6.14	207,483	2.17	210,279	2.94
Others	1,699,661	48.32	3,423,947	35.81	2,993,969	41.86
Total	3,517,510	100	9,561,428	100	7,152,338	100

the performance of ARM7TDMI processor is a little lower than the ARM9 processor. However, the performance of the implementation on ARM7TDMI is much better than other implementations. The possible reason is that the program on ARM7TDMI is fully optimized and employs the architectural features entirely.

The TriMedia CPU64 DSP, a powerful processor for multimedia applications, is a 5-issue VLIW processor with 27 function units. Further, 64-bit and SIMD instructions are supported as well [25]. The 4CIF format is 704×576 , which is 16 times larger than QCIF. However, the performance can also be compared with other implementations.

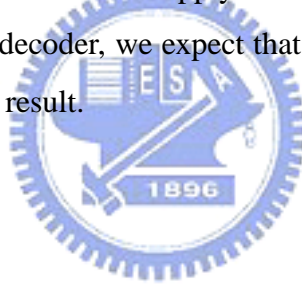
The PACDSP is a developing DSP with 5-issue VLIW architecture and SIMD instruction set [15]. Two load/store units and two ALU/MAC units can be assessed at the same time. Although the PACDSP is not issued yet, the performance on PACDSP is competitive to other platforms.

Although our MPEG-4 video decoder can achieve the goal of real-time implementa-

Table 5.19: Performance of MPEG-4 Video Decoder on Different Platforms

Processor	Freq. (MHz)	fps	Profile
TI C6201 [22]	200	28.57 (QCIF)	Not mentioned
ARM7TDMI [23]	12	15 (QCIF)	Simple profile
Philips TriMedia64 [24]	300	30 (4CIF)	Not mentioned
PACDSP [15]	200	97.54 (QCIF)	Simple profile without error resilience
ARM920T (Ours)	200 (I) 200 (P)	21.74 (QCIF) 57.14 (QCIF)	Simple profile without error resilience

tion, our design is not fully optimized. If we apply the methods of optimization which we mentioned before to the whole decoder, we expect that the speedup would be more than 50% compared with the current result.



Chapter 6

Conclusion and Future Work

6.1 Conclusion

We considered implementation of real-time MPEG-4 simple profile video decoder on ARM920T. The work was based on the source MoMuSys, which is a powerful but huge codec for MPEG-4 encoder and decoder.

Since the profile which the MoMuSys can support is main profile, we first used macros to reduce the code size for simple profile implementation. Moreover, the usage of memory is very inefficient because the MoMuSys code is written by many people together. We removed the memory allocation instructions as more as possible. After the efforts on reducing code size and memory usage, we reduced about 80% of the original code size.

For implementation, we analyzed the profile of MoMuSys using VTune and the profiler of ADS in chapter 4. Then we got some information about the time-critical processes in the decoding flow. Furthermore, the low-level computational complexity was analyzed for the time-critical processes. According to the analysis, we planned some strategies for optimization.

In chapter 5, the methods for optimization were divide into two categories, algorithmic and architecture level optimization. We first focused on the algorithmic level for intra and inter frame decoding. Then we found that IQ and IT can be skipped by checking the header information of each MB. The computations were saved for the null texture and null residual blocks. In architecture level optimization, we employed the architectural features,

such as conditional execution and LDM/STM to optimize the time-critical functions.

Although our MPEG-4 decoder is not fully optimized, the percentage of speedup for intra and inter frame decoding have achieved 38% and 81%, respectively in the best case. Since the frequency of the ARM9 processor is 200 MHz, we have achieved the goal of real-time implementation. Compared with other implementations, the performance of our implementation is competitive.

6.2 Future Work

There are several improvements and extensions can be considered in the future:

- Fully optimization

Since the decoder is not fully optimized, there are some processes can be optimized, such as bitstream access, VLD, inverse scan, inverse quantization, DC/AC prediction, and motion decoding. They may be optimized by employing the methods in architecture level.

- Combining processes

For reducing load/store, combining IQ and IT have been adopted in many implementations. The more processes combined, the more time we save for memory access.

- Dual-core implementation

Several specific DSPs have been developed for multimedia processing. The performance of them on some signal processing like IDCT is greater than the performance of ARM processor. If the time-critical functions can be done by DSPs, the performance will be much better.

- Optimization of the MPEG-4 encoder and the main profile decoder

Among the days of our research, we also implement the MPEG-4 encoder and the main profile decoder from the MoMuSys code without any optimization. Since the MoMuSys code is very inefficient, there must be a great improvement can be done.

Moreover, the optimization methods mentioned in the thesis also can be applied on them.



Bibliography

- [1] Mobile Multimedia Systems (MoMuSys) web site, <http://www.tnt.uni-hannover.de/js/project/eu/momusys/>
- [2] International Committee for Information Technology Standards, <http://www.ncits.org/>.
- [3] MPEG-4 Video Group, “MPEG-4 overview — (V.21 Jeju version),” Doc. no. ISO/IEC JTC1/SC29/WG11 N4668, Mar. 2002.
- [4] ISO/IEC 14496-2:2001, *Information Technology — Coding of Audio-Visual Objects — Part 2: Visual*. July 2001.
- [5] T. Sikora, “The MPEG-4 video standard verification model,” *IEEE Trans. Circuits Systems Video Tech.*, vol. 7, no. 1, pp. 19–31, Feb. 1997.
- [6] A. Puri and A. Eleftheriadis, “MPEG-4: An object-based multimedia coding standard supporting mobile applications mobile networks and applications,” *Mobile Networks Applic.*, vol. 3, pp. 5–32, 1998.
- [7] MPEG-4 Video Group, “MPEG-4 video verification model version 18.0,” Doc. no. ISO/IEC JTC1/SC29/WG11 N3908, Pisa, Jan. 2001.
- [8] A. Ebrahimi and C. Horne, “MPEG-4 natural video coding — an overview,” *Signal Processing Image Commun.*, vol. 15, pp. 365–385, 2000.
- [9] ARM, *ARM Architecture Reference Manual*. Doc. no. DDI0100E, 2000.

- [10] Tian-Sheuan Chang, "Lecture 3 — ARM processor architecture," in *Lecture and lab notes of SoC Design Lab*, Dept. of Electronics Engineering, National Chiao Tung University, 2005.
- [11] ARM, *ARM Developer Suite Version 1.2 – Developer Guide*. Doc. no. DUI0056D, 2001.
- [12] ARM document, *ARM Developer Suite Version 1.2 – Getting Started*. Doc. no. DUI0064D, 2001.
- [13] Meng-Yuan Liu, "Real-time implementation of MPEG-4 video encoder using SIMD-enhanced Intel processor," M.S. thesis, Degree Program of Electrical Engineering and Computer Science, National Chiao Tung University, Hsinchu, Taiwan, R.O.C., July 2004.
- [14] Pei-Yun Kuo, "Real-time implementation of MPEG-4 video encoder on digital signal processors," M.S. thesis, Department of Electronics Engineering, National Chiao Tung University, Hsinchu, Taiwan, R.O.C., July 2003.
- [15] Chung-Yen Tsai, "Software implementation of MPEG-4 video decoder on PACDSP platform," M.S. thesis, Department of Electronics Engineering, National Chiao Tung University, Hsinchu, Taiwan, R.O.C., July 2006.
- [16] Intel, *Getting Started With the VTune(TM) Performance Analyzer*. 2003.
- [17] N. I. Cho and S. U. Lee, "Fast algorithm and implementations of 2-D discrete cosine transform," *IEEE Trans. Circuit Syst.*, vol. 38, pp. 297–305, Mar. 1991.
- [18] B. G. Lee, "A new algorithm to compute the discrete cosine transform," *IEEE Trans. Acoust. Speech Signal Processing*, vol. 32, no. 6, pp. 1243–1245, Dec. 1984.
- [19] C. Y. Hung and P. Landman, "A compact IDCT design for MPEG video decoding," in *Proc. IEEE Workshop Signal Processing Systems*, Nov. 1997.

- [20] S. Sriram and C. Y. Hung, "MPEG-2 video decoding on the TMS320C6X DSP architecture," in *IEEE Signal Systems Computer Conf.*, vol. 2, Nov. 1998, pp. 1735–1739.
- [21] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach, 3rd ed.* San Francisco: Morgan Kaufmann Publishers, 2003.
- [22] N. Ventroux, J. F. Nezan, H. Raulet, and O. Deforges, "Rapid prototyping for an optimized MPEG-4 decoder implementation over a parallel heterogenous architecture," in *Proc. Int. Conf. Multimedia Expo*, vol. 3, July 2003, pp. 417–420.
- [23] K. Ramkishor and U. Gunashree, "Real time implementation of MPEG-4 video decoder on ARM7TDMI," in *Proc. Int. Symp. Intelligent Multimedia Video Speech Processing*, May 2001, pp. 522–526.
- [24] J. H. Kuo, J. L. Wu, J. Shiu, and K. L. Huang, "A low-cost media-processor based real-time MPEG-4 video decoder," *IEEE Int. Conf. Consumer Electronics*, June 2002, pp. 272–273.
- [25] J. T. J. VanEijndhoven, *et al.*, "TriMedia CPU64 architecture," in *IEEE Int. Conf. Computer Design*, 1999

Appendix A

Assembly Code of Several Functions for Optimization

A.1 8×8 IDCT

```
AREA idct8x8, CODE, READONLY
BlockIDCT2 FUNCTION
EXPORT BlockIDCT2
;=====
;           Horizontal idct
;=====
STMFD r13!,{r2-r12,r14}
MOV r12, #7 ;i = 7
ADD r1, r1, #256
loop1
LDR r14, =coeff ;set address of coefficient

;down 4 ; X(1) X(3) X(5) X(7)
ADD r11, r0, r12, LSL #5
LDR r6, [r11, #4] ;tmp[1]
LDR r7, [r11, #12] ;tmp[3]
LDR r8, [r11, #20] ;tmp[5]
LDR r9, [r11, #28] ;tmp[7]

LDMIA r14!, {r10-r11} ;e = tmp[1] * c7 - tmp[7] * c1
MUL r2, r6, r11 ;r10 = c1, r11 = c2
MUL r3, r9, r10
SUB r2, r2, r3

MUL r4, r9, r11 ;h = tmp[7] * c7 + tmp[1] * c1;
MLA r3, r6, r10, r4

LDMIA r14!, {r10-r11} ;f = tmp[5] * c3 - tmp[3] * c5;
MUL r4, r8, r10 ;r10 = c3, r11 = c5
MUL r5, r7, r11
SUB r4, r4, r5

MUL r10, r7, r10 ;g = tmp[3] * c3 + tmp[5] * c5;
```




```

MLA    r5, r8, r11, r10

ADD    r6, r2, r4        ;tmp[4] = e + f;
SUB    r7, r2, r4        ;tmp[5] = (e - f+131072)>>18;
ADD    r7, r7, #131072
MOV    r7, r7, ASR #18
SUB    r8, r3, r5        ;tmp[6] = (h - g+131072)>>18;
ADD    r8, r8, #131072
MOV    r8, r8, ASR #18
ADD    r9, r3, r5        ;tmp[7] = h + g;
LDR    r11, [r14], #4    ;tmp[5] = (tmp[6] - tmp[5]) * c0;
SUB    r10, r8, r7       ;tmp[6] = (tmp[6] + tmp[5]) * c0;
ADD    r8, r8, r7
MUL    r7, r10, r11
MUL    r8, r11, r8

;up 4 ; X(0) X(4) X(2) X(6) r2-r5,r10,r11
ADD    r11, r0, r12, LSL #5
LDR    r2, [r11, #0 ]    ;tmp[0]
LDR    r3, [r11, #16]    ;tmp[4]
LDR    r4, [r11, #8 ]    ;tmp[2]
LDR    r5, [r11, #24]    ;tmp[6]

LDR    r11, [r14], #4    ;tmp[0] = (tmp[0] + tmp[4]) * c4;
ADD    r10, r2, r3       ;tmp[1] = (tmp[0] - tmp[4]) * c4;
SUB    r3, r2, r3       ;r11 = c4
MUL    r2, r10, r11
MUL    r3, r11, r3

LDMIA  r14!, {r10-r11}   ;tmp[2] = tmp[2] * c6 - tmp[6] * c2;
MUL    r14, r5, r10     ;tmp[3] = tmp[6] * c6 + tmp[2] * c2;
MUL    r10, r4, r10     ;r10 = c2, r11 = c6
MLA    r5, r11, r5, r10
MUL    r11, r4, r11
SUB    r4, r11, r14

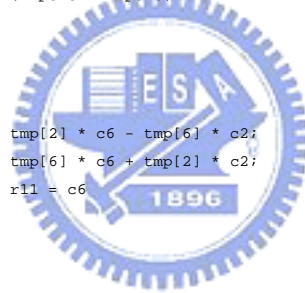
ADD    r10, r2, r5       ;tmp[0] = tmp[0] + tmp[3];
SUB    r5, r2, r5       ;tmp[3] = tmp[0] - tmp[3];
MOV    r2, r10
ADD    r10, r3, r4       ;tmp[1] = tmp[1] + tmp[2];
SUB    r4, r3, r4       ;tmp[2] = tmp[1] - tmp[2];
MOV    r3, r10

;    block[i][0] = ((tmp[0] + tmp[7])+131072)>>18;
;    block[i][7] = ((tmp[0] - tmp[7])+131072)>>18;
;    block[i][1] = ((tmp[1] + tmp[6])+131072)>>18;
;    block[i][6] = ((tmp[1] - tmp[6])+131072)>>18;
;    block[i][2] = ((tmp[2] + tmp[5])+131072)>>18;
;    block[i][5] = ((tmp[2] - tmp[5])+131072)>>18;
;    block[i][3] = ((tmp[3] + tmp[4])+131072)>>18;
;    block[i][4] = ((tmp[3] - tmp[4])+131072)>>18;

ADD    r10, r2, r9
SUB    r11, r2, r9
ADD    r2, r10, #131072
ADD    r9, r11, #131072
MOV    r2, r2, ASR #18
MOV    r9, r9, ASR #18

ADD    r10, r3, r8
SUB    r11, r3, r8

```



```

ADD r3, r10, #131072
ADD r8, r11, #131072
MOV r3, r3, ASR #18
MOV r8, r8, ASR #18

ADD r10, r4, r7
SUB r11, r4, r7
ADD r4, r10, #131072
ADD r7, r11, #131072
MOV r4, r4, ASR #18
MOV r7, r7, ASR #18

ADD r10, r5, r6
SUB r11, r5, r6
ADD r5, r10, #131072
ADD r6, r11, #131072
MOV r5, r5, ASR #18
MOV r6, r6, ASR #18

    STMDB r1!, {r2-r9} ;from the end of block

SUBS r12,r12,#1 ;i--
BGE loop1

;=====
; Vertical idct
;
; r0 is useless here, so we get one more register to use
;=====

Vertical
MOV r12, #7 ;i = 7
loop2
LDR r14, =coeff ;set address of coefficient

;down 4 ; X(1) X(3) X(5) X(7)
ADD r11, r1, r12, LSL #2 ;use r1, the result of Horizontal
LDR r6, [r11, #32] ;tmp[1]
LDR r7, [r11, #96] ;tmp[3]
LDR r8, [r11, #160] ;tmp[5]
LDR r9, [r11, #224] ;tmp[7]

LDMIA r14!, {r10-r11} ;e = tmp[1] * c7 - tmp[7] * c1;
MUL r2, r6, r11 ;r10 = c1, r11 = c7
MUL r3, r9, r10
SUB r2, r2, r3

MUL r4, r9, r11 ;h = tmp[7] * c7 + tmp[1] * c1;
MLA r3, r6, r10, r4

LDMIA r14!, {r10-r11} ;f = tmp[5] * c3 - tmp[3] * c5;
MUL r4, r8, r10 ;r10 = c3, r11 = c5
MUL r5, r7, r11
SUB r4, r4, r5

MUL r10, r7, r10 ;g = tmp[3] * c3 + tmp[5] * c5;
MLA r5, r8, r11, r10

ADD r6, r2, r4 ;tmp[4] = e + f;

SUB r7, r2, r4 ;tmp[5] = (e - f+131072)>>18;
ADD r7, r7, #131072

```



```

MOV    r7, r7, ASR #18

SUB    r8, r3, r5          ;tmp1[6] = (h - g+131072)>>18;
ADD    r8, r8, #131072
MOV    r8, r8, ASR #18

ADD    r9, r3, r5          ;tmp[7] = h + g;

LDR    r11, [r14], #4      ;tmp[5] = (tmp1[6] - tmp1[5]) * c0;
SUB    r10, r8, r7         ;tmp[6] = (tmp1[6] + tmp1[5]) * c0;
ADD    r8, r8, r7         ;r11 = c0
MUL    r7, r10, r11
MUL    r8, r11, r8

;up 4 ; X(0) X(4) X(2) X(6) r2-r5,r10,r11
ADD    r11, r1, r12, LSL #2 ;use r1, the result of Horizontal
LDR    r2, [r11, #0 ]      ;tmp[0]
LDR    r3, [r11, #128]     ;tmp[4]
LDR    r4, [r11, #64 ]     ;tmp[2]
LDR    r5, [r11, #192]     ;tmp[6]

LDR    r11, [r14], #4      ;tmp1[0] = (tmp[0] + tmp[4]) * c4;
ADD    r10, r2, r3         ;tmp1[1] = (tmp[0] - tmp[4]) * c4;
SUB    r3, r2, r3         ;r11 = c4
MUL    r2, r10, r11
MUL    r3, r11, r3

LDMIA  r14!, {r10-r11}     ;tmp1[2] = tmp[2] * c6 - tmp[6] * c2;
MUL    r0, r5, r10         ;tmp1[3] = tmp[6] * c6 + tmp[2] * c2;
MUL    r10, r4, r10        ;r10 = c2, r11 = c6
MLA    r5, r11, r5, r10
MUL    r11, r4, r11
SUB    r4, r11, r0

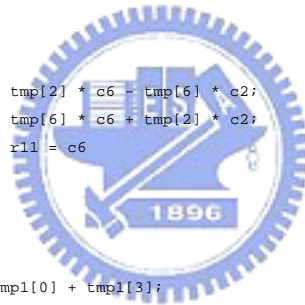
ADD    r10, r2, r5         ;tmp[0] = tmp1[0] + tmp1[3];
SUB    r5, r2, r5         ;tmp[3] = tmp1[0] - tmp1[3];
MOV    r2, r10
ADD    r10, r3, r4         ;tmp[1] = tmp1[1] + tmp1[2];
SUB    r4, r3, r4         ;tmp[2] = tmp1[1] - tmp1[2];
MOV    r3, r10

;    block[i][0] = ((tmp[0] + tmp[7])+131072)>>18;
;    block[i][7] = ((tmp[0] - tmp[7])+131072)>>18;
;    block[i][1] = ((tmp[1] + tmp[6])+131072)>>18;
;    block[i][6] = ((tmp[1] - tmp[6])+131072)>>18;
;    block[i][2] = ((tmp[2] + tmp[5])+131072)>>18;
;    block[i][5] = ((tmp[2] - tmp[5])+131072)>>18;
;    block[i][3] = ((tmp[3] + tmp[4])+131072)>>18;
;    block[i][4] = ((tmp[3] - tmp[4])+131072)>>18;

ADD    r10, r2, r9
SUB    r11, r2, r9
ADD    r2, r10, #131072
ADD    r9, r11, #131072
MOV    r2, r2, ASR #18
MOV    r9, r9, ASR #18

ADD    r10, r3, r8
SUB    r11, r3, r8
ADD    r3, r10, #131072
ADD    r8, r11, #131072
MOV    r3, r3, ASR #18

```



```

MOV r8, r8, ASR #18

ADD r10, r4, r7
SUB r11, r4, r7
ADD r4, r10, #131072
ADD r7, r11, #131072
MOV r4, r4, ASR #18
MOV r7, r7, ASR #18

ADD r10, r5, r6
SUB r11, r5, r6
ADD r5, r10, #131072
ADD r6, r11, #131072
MOV r5, r5, ASR #18
MOV r6, r6, ASR #18

ADD r11, r1, r12, LSL #2 ;address
STR r2, [r11, #0]
STR r9, [r11, #224]
STR r3, [r11, #32]
STR r8, [r11, #192]
STR r4, [r11, #64]
STR r7, [r11, #160]
STR r5, [r11, #96]
STR r6, [r11, #128]

SUBS r12, r12, #1 ; i--
BGE loop2

Exit
LDMFD r13!, {r2-r12,r14}
MOV pc, lr
ENDFUNC

AREA idct_coeff, DATA, READONLY
; c1 c7 c3 c5 c0 c4 c2 c6
coeff DCD 128553, 25571, 108982, 72820, 185364, 92682, 121095, 50159

```



A.2 VOP Reconstruction

```

AREA AddClipImage_code, CODE, READONLY
AddClipImage FUNCTION
EXPORT AddClipImage

STMFD r13!.,{r3-r12,r14}
MOV r12, r2 ; loop counter
loop
LDMIA r0!.,{r2-r5}
LDMIA r1!.,{r6-r9}

; first argument

MOV r10,r2, LSL #16 ; lower part move to higher part (for sign extension)
AND r11,r6,#0x000000FF ; save lower part of compensated frame
ADDS r14,r11,r10, ASR #16 ; ADD the first argument and compare with 0
MOVL T r14, #0 ; Sat. to 0
CMPGT r14, #255 ; Check >255

```

```

MOVGT  r14, #255          ; Sat. to 255

; second argument

MOV    r10,r2, ASR #16    ; higher part move to lower part (for add)
ADDS  r11,r10,r6, LSR #16 ; ADD the second argument and compare with 0
MOVL  r11, #0             ; Sat. to 0
CMPGT r11, #255          ; Check >255
MOVGT r11, #255          ; Sat. to 255
ORR   r2,r14,r11, LSL #16 ; pack the first and second argument

; 3rd argument

MOV    r10,r3, LSL #16    ; lower part move to higher part (for sign extension)
AND   r11,r7,#0x000000FF  ; save lower part of compensated frame
ADDS  r14,r11,r10, ASR #16 ; ADD the 3rd argument and compare with 0
MOVL  r14, #0             ; Sat. to 0
CMPGT r14, #255          ; Check >255
MOVGT r14, #255          ; Sat. to 255

; 4th argument

MOV    r10,r3, ASR #16    ; higher part move to lower part (for add)
ADDS  r11,r10,r7, LSR #16 ; ADD the 4th argument and compare with 0
MOVL  r11, #0             ; Sat. to 0
CMPGT r11, #255          ; Check >255
MOVGT r11, #255          ; Sat. to 255
ORR   r3,r14,r11, LSL #16 ; pack the 3rd and 4th argument

; 5th argument

MOV    r10,r4, LSL #16    ; lower part move to higher part (for sign extension)
AND   r11,r8,#0x000000FF  ; save lower part of compensated frame
ADDS  r14,r11,r10, ASR #16 ; ADD the 5th argument and compare with 0
MOVL  r14, #0             ; Sat. to 0
CMPGT r14, #255          ; Check >255
MOVGT r14, #255          ; Sat. to 255

; 6th argument

MOV    r10,r4, ASR #16    ; higher part move to lower part (for add)
ADDS  r11,r10,r8, LSR #16 ; ADD the 6th argument and compare with 0
MOVL  r11, #0             ; Sat. to 0
CMPGT r11, #255          ; Check >255
MOVGT r11, #255          ; Sat. to 255
ORR   r4,r14,r11, LSL #16 ; pack the 5th and 6th argument

; 7th argument

MOV    r10,r5, LSL #16    ; lower part move to higher part (for sign extension)
AND   r11,r9,#0x000000FF  ; save lower part of compensated frame
ADDS  r14,r11,r10, ASR #16 ; ADD the 7th argument and compare with 0
MOVL  r14, #0             ; Sat. to 0
CMPGT r14, #255          ; Check >255
MOVGT r14, #255          ; Sat. to 255

; 8th argument

MOV    r10,r5, ASR #16    ; higher part move to lower part (for add)
ADDS  r11,r10,r9, LSR #16 ; ADD the 8th argument and compare with 0
MOVL  r11, #0             ; Sat. to 0
CMPGT r11, #255          ; Check >255

```

```

MOVGT  r11, #255           ; Sat. to 255
ORR    r5,r14,r11, LSL #16 ; pack the 7th and 8th argument
STMDB  r0,{r2-r5}         ; store the 8 arguments

SUBS   r12, r12, #1       ; i--
BNE    loop               ; i>0

Exit
LDMFD  r13!,{r3-r12,pc}
ENDFUNC

```

A.3 Other Regular Functions

A.3.1 CopyImageI

```

CopyImage2 FUNCTION
EXPORT CopyImage2

STMFD  r13!,{r3-r12,r14}
MOV    r14, r2           ; loop counter

loop2
LDMIA  r0!,{r2-r12}      ; load 11 coefficients
STMIA  r1!,{r2-r12}      ; store 11 coefficients
SUBS   r14, r14, #1     ; i--
BNE    loop2            ; i>0

Exit2
LDMFD  r13!,{r3-r12,pc}
ENDFUNC

```



A.3.2 Bzero

```

Bzero FUNCTION
EXPORT Bzero

STMFD  r13!,{r1-r12,r14}
MOV    r1, #0
MOV    r2, #0
MOV    r3, #0
MOV    r4, #0
MOV    r5, #0
MOV    r6, #0
MOV    r7, #0
MOV    r8, #0
MOV    r9, #0
MOV    r10, #0
MOV    r11, #0
MOV    r12, #0
MOV    r14, #0

STMIA  r0!,{r1-r12, r14} ;store 64 zeros
STMIA  r0!,{r1-r12, r14}
STMIA  r0!,{r1-r12, r14}
STMIA  r0!,{r1-r12, r14}
STMIA  r0!,{r1-r12}

```

```

LDMFD r13!,{r1-r12,pc}
ENDFUNC

```

A.3.3 Putblock

```

PutBlock FUNCTION ;position of blocks in a MB (comp)
EXPORT PutBlock ; Y: 1 2 U: 5 V: 6
; 3 4

STMFD r13!,{r3-r12,r14}
MOV r12, #8 ; loop counter
CMP r0, #4
BEQ PutV
BGT PutU
; composition = 0~3

ANDS r3, r0, #1
ADDGT r2, r2, #32
ANDS r3, r0, #2
ADDGT r2, r2, #512

loop_Y
LDMIA r1!, {r3-r10}
STMIA r2!, {r3-r10}
ADD r2, r2, #32 ;next row
SUBS r12, r12, #1
BNE loop_Y
B Exit3

PutU
ADD r2, r2, #1024
loop_U
LDMIA r1!, {r3-r10}
STMIA r2!, {r3-r10}
SUBS r12, r12, #1
BNE loop_U
B Exit3

PutV
ADD r2, r2, #1280
loop_V
LDMIA r1!, {r3-r10}
STMIA r2!, {r3-r10}
SUBS r12, r12, #1
BNE loop_V

Exit3
LDMFD r13!,{r3-r12,pc}
ENDFUNC

```



A.3.4 MB_clip

```

MB_clip2 FUNCTION ;0~255
EXPORT MB_clip2

STMFD r13!,{r1-r12,r14}
MOV r14, #32

loop3

```

```

LDMIA r0!, {r1-r12}
CMP r1, #0
MOVLT r1, #0
CMP r1, #255
MOVGT r1, #255
CMP r2, #0
MOVLT r2, #0
CMP r2, #255
MOVGT r2, #255
CMP r3, #0
MOVLT r3, #0
CMP r3, #255
MOVGT r3, #255
CMP r4, #0
MOVLT r4, #0
CMP r4, #255
MOVGT r4, #255
CMP r5, #0
MOVLT r5, #0
CMP r5, #255
MOVGT r5, #255
CMP r6, #0
MOVLT r6, #0
CMP r6, #255
MOVGT r6, #255
CMP r7, #0
MOVLT r7, #0
CMP r7, #255
MOVGT r7, #255
CMP r8, #0
MOVLT r8, #0
CMP r8, #255
MOVGT r8, #255
CMP r9, #0
MOVLT r9, #0
CMP r9, #255
MOVGT r9, #255
CMP r10, #0
MOVLT r10, #0
CMP r10, #255
MOVGT r10, #255
CMP r11, #0
MOVLT r11, #0
CMP r11, #255
MOVGT r11, #255
CMP r12, #0
MOVLT r12, #0
CMP r12, #255
MOVGT r12, #255

SUBS r14, r14, #1
BNE loop3

LDMFD r13!, {r1-r12, pc}
ENDFUNC

```



自傳

吳和璋，男，民國七十一年四月二十一日出生於台灣省台北縣，高中就讀於國立臺灣師範大學附屬高級中學，民國九十三年六月畢業於交通大學電子工程學系，並於同年九月進入交通大學電子工程研究所碩士班就讀，於民國九十五年九月取得碩士學位，論文題目為：『使用 ARM9 處理器實現 MPEG-4 視訊之軟體解碼』，研究範圍與興趣為：ARM 處理器與 DSP 上之系統整合與軟體開發，主要應用範圍在多媒體訊號處理與壓縮方面。

