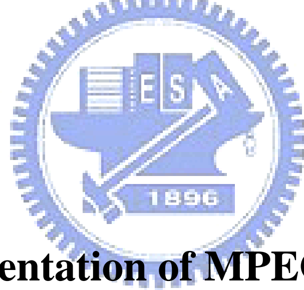
國　立　交　通　大　學

電子工程學系 電子研究所碩士班

碩　士　論　文

在 **PACDSP** 平台上之 **MPEG-4** 視訊解碼器軟體實現



Software Implementation of MPEG-4 Video Decoder

on PACDSP Platform

研 究 生： 蔡崇諺

指導教授： 林大衛 博士

中 華 民 國 九 十 五 年 六 月

在 PACDSP 平台上之 MPEG-4 視訊解碼器軟體實現

Software Implementation of MPEG-4 Video Decoder on PACDSP Platform

研 究 生：蔡崇諺　　Student : Chung-Yen Tsai
指導教授：林大衛 博士　　Advisor : Dr. David W. Lin

國 立 交 通 大 學
電子工程學系 電子研究所碩士班

碩士論文

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
June 2006
Hsinchu, Taiwan, Republic of China

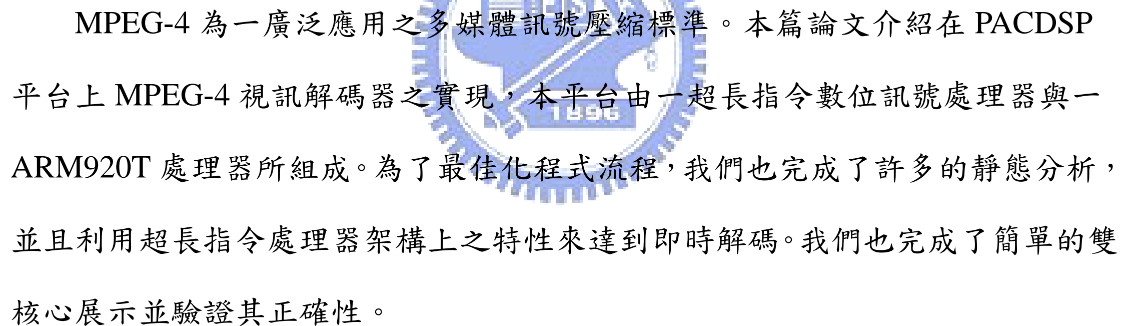
中 華 民 國 九 十 五 年 六 月

在 PACDSP 平台上之 MPEG-4 視訊解碼器軟體實現

研究生: 蔡崇諺　　　　　　　　　　　　　　　指導教授:林大衛 博士

國立交通大學電子工程學系　　電子研究所碩士班

摘要



MPEG-4 為一廣泛應用之多媒體訊號壓縮標準。本篇論文介紹在 PACDSP 平台上 MPEG-4 視訊解碼器之實現，本平台由一超長指令數位訊號處理器與一 ARM920T 處理器所組成。為了最佳化程式流程，我們也完成了許多的靜態分析，並且利用超長指令處理器架構上之特性來達到即時解碼。我們也完成了簡單的雙核心展示並驗證其正確性。

在我們的實作當中，我們使用了 MPEG-4 參考軟體，MoMuSys，當作驗證的比較對象。首先，我們分析了 MPEG-4 基於圖像解碼器之運算複雜度並藉此找到有效率的實現方法。接著，我們根據離散餘弦轉換（DCT）之特性來跳過多餘的運算，並且對於全零之剩餘方塊亦有許多可略過之計算。為了加速執行時間，我們將規律之運算分佈於兩組以增加處理器之效能。我們也使用單指令多資料（SIMD）指令以及一般指令層級平行化來減少處理器之延遲。我們討論了離散餘弦反轉換(IDCT)之效能與精確度，並且我們的實現能夠符合 IEEE1180-1190 標準之規範。我們所使用之演算法在效能上也具有與其他實現競爭的能力。在所有的最佳化之後，我們在最差情況下解碼一張 QCIF 格式之圖像需要 5,700,000

週期。也就是說，對一個工作在 175MHz 的真實 PACDSP 晶片而言，我們能夠達到每秒三十張畫面之即時解碼。而整個程式的大小為 27 Kbytes，也小於 PACDSP 的程式快取記憶體大小 32 Kbytes。最後我們在 PSDK 平台上展示了雙核心的實現。

在本篇論文當中，我們首先介紹了 MPEG-4 標準以及 PADSP 平台之概述。接著討論靜態分析、實作策略、最佳化方法、以及實驗結果。最後簡單介紹了展示雙核心實現的系統與機制。

Software Implementation of MPEG-4 Video Decoder on PACDSP Platform

Student: Chung-Yen Tsai Advisor: Dr. David W. Lin

Department of Electronics Engineering
Institute of Electronics
National Chiao Tung University

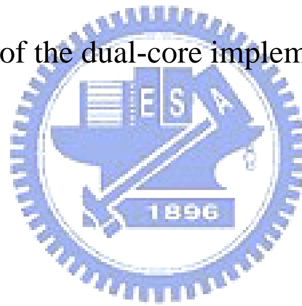
Abstract

MPEG-4 is a widely-applied multimedia coding standard. This thesis presents an implementation of MPEG-4 video decoder on the PACDSP platform, which consists of a VLIW digital signal processor (DSP) and an ARM920T processor. We complete many anlyses to optimize the program flow and utilize the advantage of VLIW processor to achieve real-time decoding. A simple dual-core demostration is completed and verified.

In our implementation, the MPEG-4 reference software, MoMuSys, is used as a golden model to verily our implementation. First, we analyze the computational complexity of the MPEG-4 frame-based video decoder, and find efficient algorithms for the implementation. Second, we skip some computations according to the nature of discrete cosine transform (DCT), and there are also lots of comutation skipped for all-zero residual blocks. Third, to speed up the execution time, we distribute the regular computations to both clusters to increase the efficiency of the processor. Single-instruction-multiple-data (SIMD) instructions and general instruction level parallelism also utilized to reduce the processor stalls. We also discuss the efficiency

and accuracy of IDCT, and the accuracy of our IDCT implementation can meet the IEEE 1180-1190 standard. The performance of our alogorithm is also competitive to other implementations. After all the optimizations, the worst-case computaion time for QCIF format is less than 5,700,000 cycles. That is, our implementation can achieve real-time decoding, 30 frame-per-second, for a real PACDSP chip running over 175 MHz. The code size is 27 Kbyte, which is smaller than the 32-Kbyte instruction cache on PACDSP. Finally, we demonstrate a simple dual-core implementation on the PAC System Developer's Kit (PSDK).

In this thesis, we first introduce the MPEG-4 standard and give an overview of the PACDSP platform. Then the static analysis, implementation strategies, the optimiztion methods, and the experiment results are discussed. Finally, we brief the system and mechanism for demonstration of the dual-core implementation on PSDK plarform.



誌謝

本篇論文的完成，誠摯地感謝我的指導老師 林大衛 博士，從電控系推甄進入電子所這個新環境時，多虧老師的循循善誘，不論在課業、研究、或者心理上遭遇挫折時的鼓勵與指導而能夠一次一次地解決困難，並且讓我學習了分析問題並加以解決的能力。而老師身體力行與樂觀積極的生活態度也深深地影響了我。在此，僅向老師及老師的家人致上最高的感謝之意。

感謝在電子研究所 CommLab 的日子裡實驗室所提供完善的研究資源。承蒙崑健、家揚、俊榮、朝雄等學長的提攜與照顧，在研究與生活上都能夠順利解決問題。而實驗室的同伴，鴻志、和璋、家賢、治傑、韋霖、旻弘、育彰、宗熹、德亘，以及室友浩緯、人中等在課業上的砥礪與生活上的幫助也讓我在忙碌的研究所生涯中仍舊擁有快樂的心情。此外，也要感謝學弟，政達、介遠，在你們的討論與幫助下，研究才能夠更加迅速與正確地完成。

最後，感謝我的家人，溫暖的家一直是我求學生涯中最強而有力的後盾，感謝你們的努力讓我能夠無後顧之憂地汲取知識，繼續升學。另外感謝我的女友，楊晨，在我求學過程一路相伴，面對壓力時不斷地鼓勵。僅將本論文獻給我敬愛的父母，蔡文彬先生、王阿珠女士，以及我摯愛的楊晨小姐。

蔡崇諺

二００六年六月于新竹

Contents

1 **Introduction** 1

2 **Overview of the MPEG-4 Video Standard** 3

2.1 Structure of MPEG-4 Video Data 3

2.2 MPEG-4 Video Texture Coding 6

2.3 Motion Coder 6

2.3.1 Texture Coder 10

2.3.2 Other Video Coding Tools [3] 15

2.3.3 Robust Video Coding 15

2.3.4 Scalable Coding 16

2.4 Profiles and Levels [2] 16

3 **Overview of The PACDSP** 19

3.1 Introduction 19

3.1.1 Architecture Features 20

3.2 Architecture Overview 21

3.3 Program Sequence Control Unit 22

3.3.1 Branch Instruction 22

3.3.2 Loop 23

3.3.3 Customized Function Unit (CFU) 24

3.3.4 Exception Handling 24

3.3.5 Interrupt Handling 24

3.4 VLIW Datapath 25

3.4.1 Ping-Pong Register File 25
3.4.2 Data/Address/Accumulator Registers 25
3.4.3 Status and Control Registers 26
3.4.4 Addressing Modes 28
3.4.5 VLIW Datapath 30
3.4.6 Data Exchange 31
3.4.7 Constant Register File 33
3.5 Scalar Unit 34
3.5.1 Overview 34
3.5.2 Control Registers 34
3.5.3 General Purpose Scalar Register File 35
3.6 Conditional Execution Control 36
3.7 ISA and Pipeline Stages 37
3.8 DSP Running Modes 38
3.9 Instruction Packet 39
3.10 Development Tools and Implementation Considerations 39
3.10.1 Development Tools 39
3.10.2 Implementation Considerations 42

4 Complexity Analysis and Implementation Strategy of MPEG-4 Framed-Based Video Decoder 43

4.1 Profiles of The MPEG-4 Frame-Based Video Decoder 44
4.1.1 Approach to Complexity Analysis 44
4.1.2 Profile on PC Using Intel VTune Performance Analyzer 45
4.1.3 Low-Level Computational Analysis 46
4.2 Implementation Strategies on PACDSP 49
4.2.1 Efficient Variable Length Decoding (VLD) 50
4.2.2 Efficient Motion Compensation 55
4.2.3 Profile on PACDSP of All Decoder Functions 56

5 Optimization of The Implementation on PACDSP 61

5.1 Algorithmic Optimization 61

5.1.1 Algorithmic Optimization for Intra Frames 61

5.1.2 Algorithmic Optimization for P-Frames 65

5.2 Architectural Optimization 67

5.2.1 General Optimization Techniques 68

5.2.2 Advantages of PACDSP 71

5.3 Experiment Results 72

5.3.1 Optimization of Dequantization 72

5.3.2 Implementation of IDCT 73

5.3.3 Overall Optimization of the implementation 79

5.4 Conclusion on Optimization 80

5.5 The Effect of Different Quantization Step (QP) 81

5.5.1 Effects of QP to I-Frame Decoding 82

5.5.2 Effects of QP to P-Frame Decoding 83

5.6 Comparison with Other Implementations 84

6 Conclusion and Future Work 89

6.1 Conclusion 89

6.2 Future Work 90

A Demonstration of MPEG-4 Frame-Based Video Decoder on Dual-Core PSDK 94

A.1 Overview of The PSDK 2.0 Platform 94

A.2 Introduction to Dual-Core Demonstration 96

A.2.1 I-Frames Decoding 96

A.2.2 P-Frames Decoding 97

B C Program and Assembly Code of IDCT 99

B.1 C Program of IDCT in MoMuSys 99

B.2 Original Assembly Code of IDCT 99

B.3 Optimized Assembly Code of IDCT 99

List of Figures

2.1 Segmentation of a frame into VOPs (from [3]). 4
2.2 Structure of coded video data (from [4]). 5
2.3 Types of VOP. 6
2.4 Positions of luminance and chrominance samples in 4:2:0 data (from [5]). 7
2.5 Motion vector prediction (from [5]). 9
2.6 Quantizers in H.263. (a) For intra DC coefficient only. (b) For inter DC and all AC coefficients. 12
2.7 Prediction of DC coefficients of blocks in an intra MB (from [3]). 14
2.8 Prediction of AC coefficients of blocks in an intra MB (from [3]). 14
2.9 Scans for 8×8 blocks (from [2]). 15

3.1 Architecture of the PACDSP [1]. 22
3.2 Ping-pong register file in one cluster [1]. 26
3.3 The available registers in one cluster [1]. 27
3.4 Illustration of the addressing mode control register (AMCR) [1]. 28
3.5 Illustration of multiplication instructions with different precisions [1]. . . 31
3.6 Different load/store instructions [1]. 32
3.7 Data Exchange between Two Clusters [1]. 33
3.8 Data broadcast among clusters [1]. 33
3.9 The Constant Register File of one cluster [1]. 35
3.10 PACDSP instruction set architecture [1]. 38
3.11 Pipeline stages of the PACDSP [1]. 38
3.12 Transitions between DSP running modes [1]. 41

3.13 Syntax of instruction packet [1]. 42
3.14 Simplified syntax of instruction packet [1]. 42

4.1 Block diagram of MPEG-4 frame-based video decoder [2]. 44
4.2 Example of bit by bit matching on PACDSP. 53
4.3 Example of one table mapping with magnitude-offset on PACDSP. 54
4.4 Example of multiple-pass matching on PACDSP. 55
4.5 Example of bounded multiple-pass lookup with magnitude-offset on PACDSP. 56
4.6 Comparison of different VLD methods on PACDSP 57

5.1 DC spreading from decoded coefficient to output block. 62
5.2 Assembly code of DC spreading. 62
5.3 Assembly code of new check in vertical AC reconstruction. 65
5.4 Example of vector addition. 69
5.5 Example of static rescheduling technique. 70
5.6 Example of loop unrolling technique. 70
5.7 Example of software pipelining technique 71
5.8 Original and optimized assembly code of IQ. 75
5.9 The IDCT algorithm used in MoMuSys. 78
5.10 The even-odd decomposition IDCT algorithm[8]. 79
5.11 Speed-up of different optimization methods for I-frames. 84
5.12 Speed-up of different optimization methods for P-frames. 85

A.1 PAC System Developer's Kit (PSDK) 2.0 95
A.2 Memory map of the dualcore demonstration 96
A.3 Co-processing mechanism for I-frames 97
A.4 Co-processing mechanism for P-frames 98

B.1 C program of IDCT in MoMuSys reference software including clipping. . 100
B.2 Assembly code of our initial IDCT implementation (horizontal processing).101
B.3 Assembly code of our initial IDCT implementation (vertical processing and clipping). 102

B.4 Assembly code of optimized IDCT implementation (horizontal processing). 103
B.5 Assembly code of optimized IDCT implementation (vertical processing and clipping). 104

List of Tables

2.1 Weighting Values $H_0(i, j)$, $H_1(i, j)$, and $H_2(i, j)$ 11

2.2 Default Quantization Matrix (Q) [2] 13

2.3 Nonlinear Scaler for DC Coefficients (from [2]) 13

2.4 Profiles and Tools (from [2]) 18

3.1 Details of Control Register Files [1] 36

3.2 Memory-Mapped Control Registers [1] 37

3.3 Pipeline Stages and Their Descriptions 39

3.4 Running Modes of the PACDSP [1] 40

3.5 Instruction Type in Each Instruction Slot 41

4.1 Profile of Frame-Based MPEG-4 Decoding of QCIF on PC 46

4.2 Complexity of Luminance Motion Compensation in One QCIF Frame . . 48

4.3 Complexity of Chrominance Motion Compensation in One QCIF Frame . 49

4.4 Complexity of Dequantization and IDCT for One 8×8 Block in MoMuSys Code 50

4.5 Variable Length Codes for dct_dc_size_luminance [2] 52

4.6 Execution Time of Different VLD Methods on PACDSP 58

4.7 Analysis of Necessary Interpolation Using MoMuSys 59

4.8 Estimated Profile of Frame-Based MPEG-4 Decoding of QCIF on PACDSP 60

5.1 Number of Skipped Blocks in 90 Intra Frames (Check CBP and ACPred_Flag Only) 64

5.2 Number of Skipped Blocks in 90 Intra Frames with Further Check After AC Prediction 66
5.3 Execution Time of Intra Frame Decoding on PACDSP 66
5.4 Number of Skipped Blocks in 89 P Frames 67
5.5 Execution Time of Inter (P) Frame Decoding on PACDSP 68
5.6 Analysis of Skipped Coefficients in Dequantization (90 I-frames) 74
5.7 Improvement after Optimization of Dequatization 76
5.8 Comparison of Computational Complexity for 8-point IDCT 76
5.9 Test of Compliance Using IEEE Std. 1180-1190 77
5.10 Comparison of IDCT on Different Platforms 80
5.11 Improvement After Optimization of IDCT 81
5.12 Overall Optimization after IDCT Optimization 82
5.13 Execution Time Before and After Optimizations 83
5.14 Number of Skipped Blocks in 90 Intra Frames with Different QP 86
5.15 Effects of Different QP to Execution Time of I-Frame Decoding on PACDSP 86
5.16 Number of Skipped Blocks in 89 P-Frames with Different QP 87
5.17 Percentage of Fractional Motion Vectors with Different QP 87
5.18 Effects of Different QP to Execution Time of P-Frame Decoding on PACDSP 88
5.19 Performance of MPEG-4 Video Decoder on Different Platforms 88

Chapter 1

Introduction

In modern industry, compression of audio-visual information becomes more and more important, especially for applications on mobile devices. Besides, digital signal processors (DSPs) are also popularly used on these mobile devices. Our goal is the implementation of MPEG-4 video decoder on the PACDSP platform.

The MPEG-4 standard for coding of audio-visual information has been widely adopted in various consumer products. There are several tools in the MPEG-4 standards, and they are used for different purposes. Since the present work is the first attempt to implement MPEG-4 video codecs on the PACDSP platform, we decide to implement the freme-based part of the MPEG-4 decoder first, and the corresponding encoder and other MPEG-4 video tools are left to the future work.

PACDSP is a high performance, low cost VLIW (Very Long Instruction Word) DSP for multimedia applications[1]. Optimized architecture for data stream applications gives a strong reason for system designers to use PACDSP to implement media codecs. The instruction set architecture (ISA) of PACDSP is optimized for audio and video applications, so PACDSP is suitable for products with multi-standard codec requirement. In addition, the low power design for PACDSP makes it possible to use PACDSP on portable devices.

This thesis is organized as follows. Chapter 2 is the overview of MPEG-4 standards. Chapter 3 introduces the architecture and specification of the PACDSP platform. Chapter 4 is the analysis of complexity for the reference software of MPEG-4. In addition, the implementation strategy of MPEG-4 video decoder is also simply introduced in this

chapter. The contents of chapter 5 are about the different optimization technologies and their experiment results. We also compare our implementation with that of other processors Finally, we will give some conclusions in chapter 6, and the future works are listed as well.

Chapter 2

Overview of the MPEG-4 Video Standard

2.1 Structure of MPEG-4 Video Data

The contents of this section have been taken to a large extent from [2]–[5].

A video sequence is composed of a succession of frames (or pictures). MPEG-4 divides a frame into a number of video object planes (VOPs). A succession of VOPs is termed a video object (VO). The idea of VOPs is illustrated in Fig. 2.1. Each VO is encoded separately and multiplexed to form a bitstream that users can access and manipulate. The encoder sends, together with VOs, information about scene composition to indicate where and when VOPs of a VO are to be displayed. Figure 2.2 shows the organization of the coded MPEG-4 video datta in a top-down hierarchical structure. A frame-based video can be interpreted as having only one VO. And in non-scalable coding, there is only one video object layer (VOL). The meanings of the hierarchical layers are as follows.

- VideoSession (VS): A video session simply consists of an ordered collection of video objects.
- VideoObject (VO): A video object is a complete scene or a portion of a scene with a semantic. In the simplest case this can be a rectangular frame, or it can be an

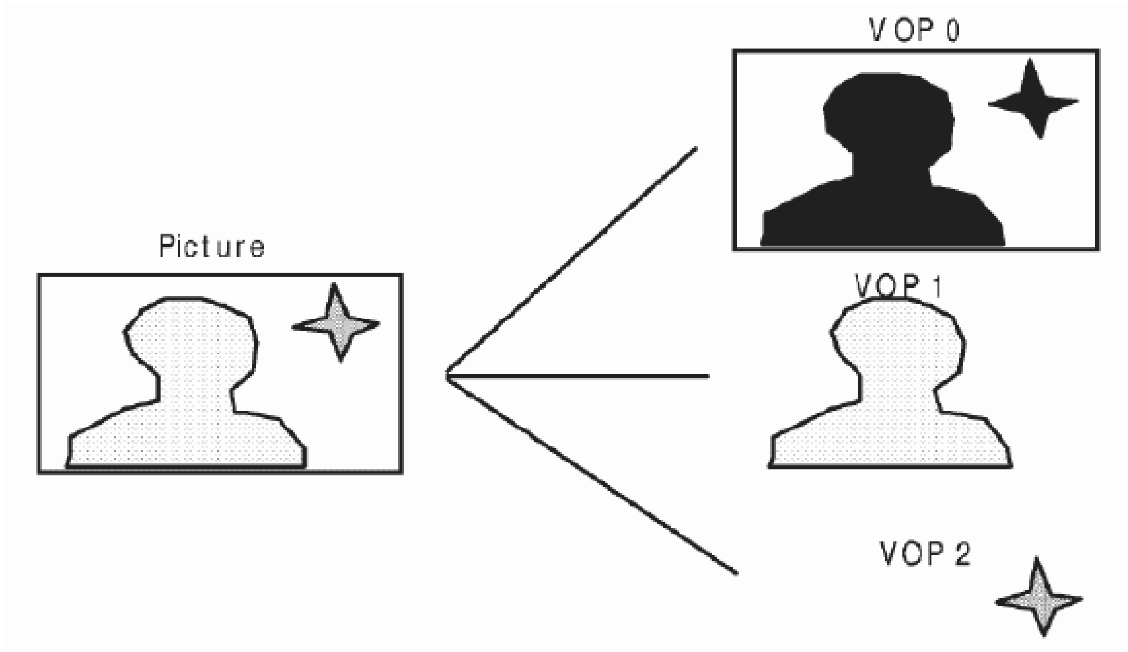


Figure 2.1: Segmentation of a frame into VOPs (from [3]).

arbitrarily shaped object corresponding to a physical object or background of the scene.

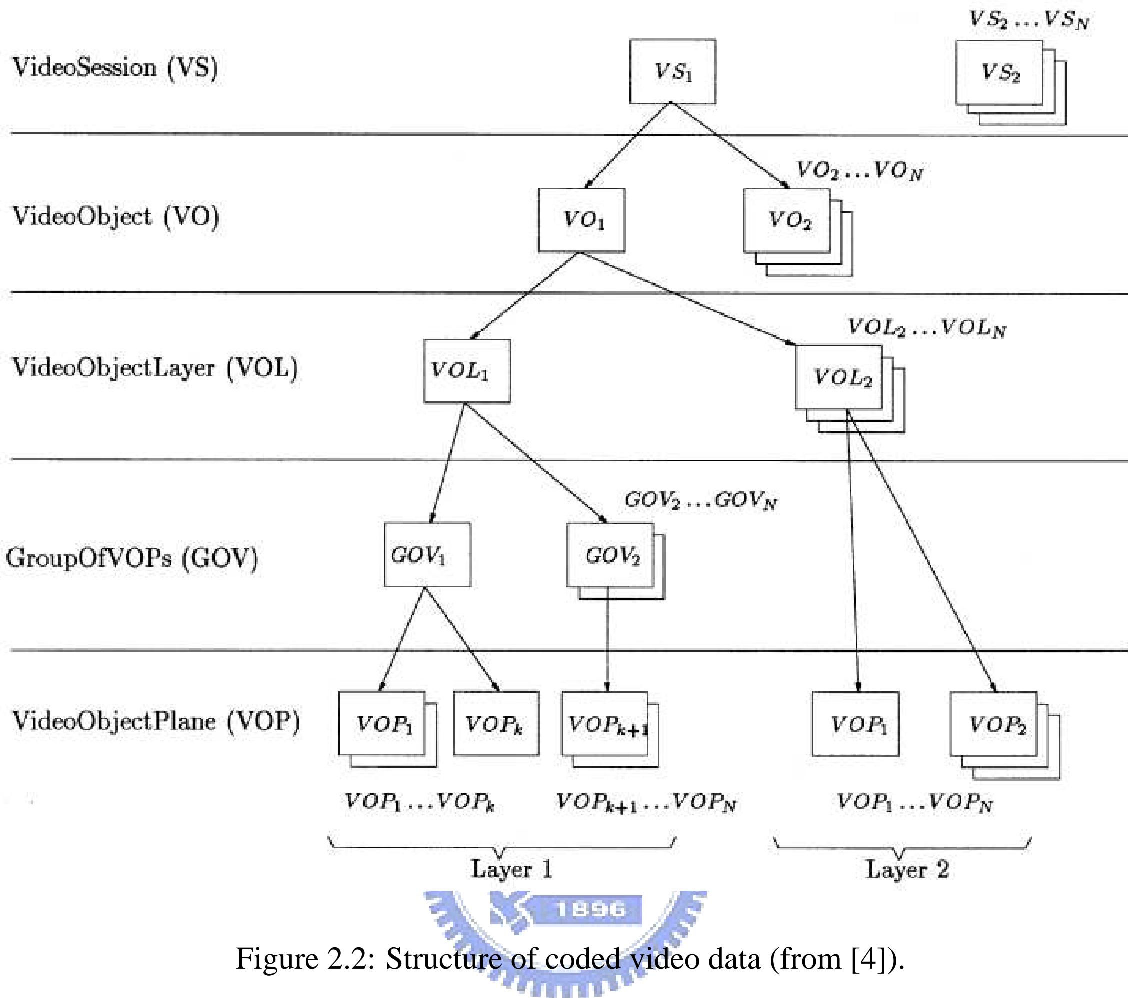
- VideoObjectLayer (VOL): Each video object can be encoded in scalable (multi-layer) or non-scalable form (single layer), depending on the application, represented by VOL. The VOL provides support for scalable coding. A video object can be encoded using spatial or temporal scalability, going from coarse to fine resolution.

- GroupOfVideoObjectPlanes (GOV): Group of video object planes are optional entities. The GOV groups together video object planes. GOVs can provide points in the bitstream where VOPs are encoded independently from each other, and can thus provide random access points into the bitstream.

- VideoObjectPlane (VOP): A VOP is a time sample of a video object.

As in the earlier MPEG standards, a VOP can be of the I, the P, or the B type, as illustrated in Fig. 2.3. In addition, there is a fourth type of VOP, called S, defined in MPEG-4. These are briefly explained below:

1. An intra-coded (I) VOP is coded using information only from itself.



1896

Figure 2.2: Structure of coded video data (from [4]).

2. A predictive-coded (P) VOP is a VOP which is coded using motion compensated prediction from a past reference VOP.

3. A bidirectionally predictive-coded (B) VOP is a VOP which is coded using motion compensated prediction from a past and/or future reference VOP(s).

4. A sprite (S) VOP 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. We omit further introduction of the S VOP.

The macroblock (MB) is a basic coding structure constructing VOP. An MB contains a section of the luminance component of 16×16 (horizontal $\times$ vertical) pixels in size, non-overlapping with each other, and the sub-sampled chrominance components in 4:2:0 format. The luminance and chrominance samples are positioned as shown in Fig. 2.4. In

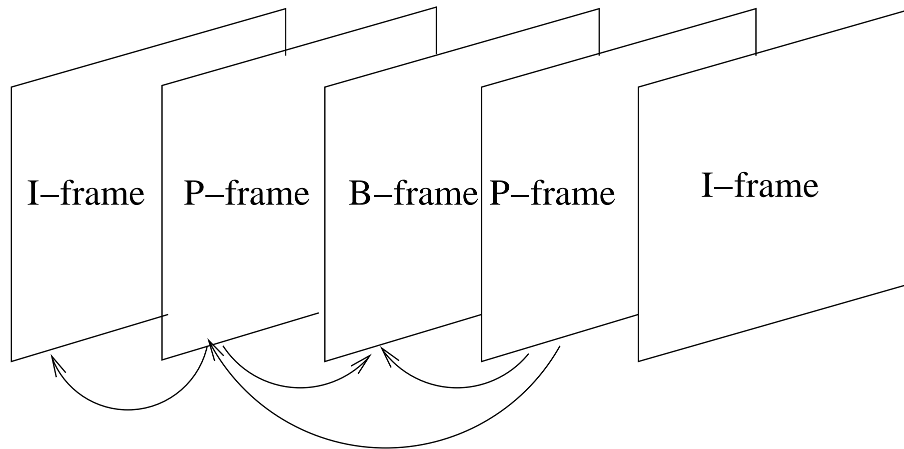


Figure 2.3: Types of VOP.

this format, an MB is divided into 4 luminance blocks and 2 chrominance blocks, each 8×8 pixels in size.

2.2 MPEG-4 Video Texture Coding

The contents of this section have been taken to a large extent from [3]–[5]. We concentrate on the techniques pertaining to frame-based video coding.

2.3 Motion Coder

Motion coding applies to P-VOP and B-VOP, for the purpose of reducing temporal redundancy. The motion coder consists of a motion estimator, motion compensator, previous/next VOPs store and motion vector (MV) predictor and coder.

Motion Estimation

The motion estimation (ME) techniques used in MPEG-4 can be seen as an extension of standard MPEG-1/2 or H.263 block matching techniques with modified block (polygon) matching to handle arbitrary-shaped VOPs. But this modification is of little concern to the current report.

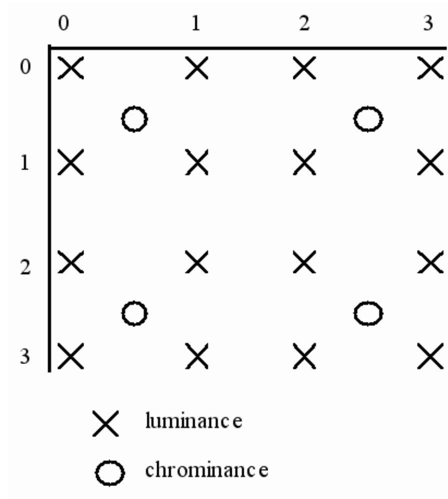


Figure 2.4: Positions of luminance and chrominance samples in 4:2:0 data (from [5]).

The basic motion estimation may be performed on 16×16 luminance MB. The motion vector is specified to half-pixel accuracy. In many coding software implementations, the motion estimation is performed by some search method to integer pixel accuracy vector and, using it as the initial estimate, a half pixel search is performed around it. Because the motion vector may be non-integer, sample interpolation is necessary. The interpolation is carried out only in half sample mode, where the half sample values are calculated by bilinear interpolation.

In the MPEG-4 standard, besides motion vector for 16×16 MB, motion vector can be sent for individual 8×8 blocks to reduce more prediction errors. Both the 8×8 block motion compensation and overlapped motion compensated prediction are referred to as advanced prediction in H.263 and are adapted in MPEG-4 to work with arbitrary shaped VOPs.

Motion Vector Encoder

When using INTER mode coding, the motion vector must be coded. Horizontal and vertical motion vector are coded differentially by using a spatial neighborhood of three motion vectors that have already been coded, as illustrated in Fig. 2.5. These three motion vectors are candidate predictors for the differential coding. The differential coding of

motion vectors is performed with reference to the reconstructed shape. In the special cases at the borders of the current VOP the following decision rules are applied:

1. If the MB of one and only one candidate predictor is outside the VOP, it is set to zero.

2. If the MBs of two and only two candidate predictors are outside the VOP, they are set to the third candidate predictor.

3. If the MBs of all three candidate predictors are outside the VOP, they are set to zero.

The motion vector coding is performed separately on the horizontal and vertical components. For each component, the median value of the three candidates for the same component is used as predictor, denoted P_x and P_y, respectively. After finding the predictors, the vector differences $MVD_x = MV_x - P_x$ and $MVD_y = MV_y - P_y$ are coded by variable length coding (VLC).

Motion Compensation

The motion compensator uses motion vectors to compute motion compensated prediction block, $pred[i][j]$, from the same reference VOP. In addition to basic motion compensation processing, three alternalties 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. When a sample referenced by a motion vector is outside the decoded VOP area, an edge sample is used. The $pred[i][j]$ is defined through the following:

$$xref = \min(\max(xcurr + dx, vhmcsr), xdim + vhmcsr - 1),$$

$$yref = \min(\max(ycurr + dy, vvmcsr), ydim + vvmcsr - 1),$$

where $vhmcsr$ = vop_horizontal_mc_spatial_ref, $vvmcsr$ = vop_vertical_mc_spatial_ref, $(ycurr, xcurr)$ is the coordinate of a sample in the current VOP, $(yref, xref)$ is the coordinate of a sample in the reference VOP, (dy, dx) is the motion vector, and $(ydim, xdim)$ is the dimension of the bounding rectangle of the reference VOP.

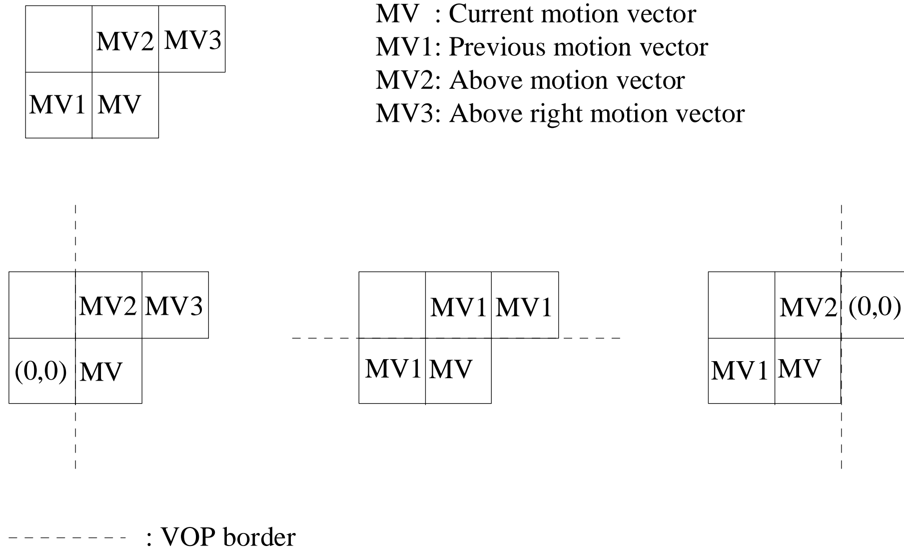


Figure 2.5: Motion vector prediction (from [5]).

One/two/four vectors decision is indicated by the MCBPC codeword and field_prediction flag for each MB. If one motion vector is transmitted for a certain MB, this is considered 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 MB, 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 MB.

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 as follows:

$$
\begin{aligned}
\bar{P}(i,j) &= [p(i+MV_x^0, j+MV_y^0)H_0(i,j) \\
&\quad + p(i+MV_x^1, j+MV_y^1)H_1(i,j) \\
&\quad + p(i+MV_x^2, j+MV_y^2)H_2(i,j) + 4]/8,
\end{aligned}
$$

where (MV_x^0, MV_y^0) denotes the motion vector for the current block, (MV_x^1, MV_y^1) the motion vector of the block above or below, (MV_x^2, MV_y^2) the motion vector of the block to the left or to the right, and $H_0(i,j)$, $H_1(i,j)$, and $H_2(i,j)$ the weighting of each pixel in the current block and neighbor blocks. The values of $H_0(i,j)$, $H_1(i,j)$, and $H_2(i,j)$ denote the weighting of each pixel in the current block and neighbor blocks, and they are shown in Table 2.1. It is noted that $H_0(i,j)$ is used for current luminance block, $H_1(i,j)$ for prediction of motion vectors of luminance blocks on top or bottom of current block, and $H_2(i,j)$ for prediction of motion vectors of luminance blocks on the left or right of current block.

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

1. Forward mode: Only the forward vector (MVFx,MVFy) 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 (MVBx,MVBy) is applied. 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 (MVFx,MVFy) and the backward vector (MVBx,MVBy) are applied. The prediction blocks $\bar{P}_y(i,j)$, $\bar{P}_u(i,j)$, $\bar{P}_v(i,j)$ are generated from the forward and the backward reference VOPs by doing the forward and the backward predictions and then averaging both predictions pixel by pixel.

2.3.1 Texture Coder

The texture information of a VOP is present in the luminance Y and two chrominance components Cb and Cr of the video signal. In the case of an I-VOP, the encoded texture information directly represents in the values of the luminance and chrominance components. In the case of motion compensated VOPs the encoded texture information rep-

Table 2.1: Weighting Values $H_0(i,j)$, $H_1(i,j)$, and $H_2(i,j)$

$H_0(i,j)$								$H_1(i,j)$								$H_2(i,j)$							
4	5	5	5	5	5	5	4	2	2	2	2	2	2	2	2	2	1	1	1	1	1	1	2
5	5	5	5	5	5	5	5	1	1	2	2	2	2	1	1	2	2	1	1	1	1	2	2
5	5	6	6	6	6	5	5	1	1	1	1	1	1	1	1	2	2	1	1	1	1	2	2
5	5	6	6	6	6	5	5	1	1	1	1	1	1	1	1	2	2	1	1	1	1	2	2
5	5	6	6	6	6	5	5	1	1	1	1	1	1	1	1	2	2	1	1	1	1	2	2
5	5	6	6	6	6	5	5	1	1	1	1	1	1	1	1	2	2	1	1	1	1	2	2
5	5	5	5	5	5	5	5	1	1	2	2	2	2	1	1	2	2	1	1	1	1	2	2
4	5	5	5	5	5	5	4	2	2	2	2	2	2	2	2	2	1	1	1	1	1	1	2

resents the residual error remaining after motion-compensated prediction. The texture coder includes padding process (for object-based coding, and applied only if needed), 8×8 two-dimensional (2D) discrete cosine transform (DCT), quantization, coefficient prediction, coefficient scan and VLC. We describe the last four elements below.

Quantization

MPEG-4 video supports two quantization techniques, one referred to as the H.263 quantization method and the other, the MPEG quantization method. The H.263 quantization method is uniform with dead zone for intra and inter AC coefficients and uniform for intra DC coefficients. The MPEG quantization method is uniform.

Figure 2.6 shows the quantizer characteristics in H.263. For inter DC and all AC coefficients, input between $-$Th and $+$Th is quantized to zero. All coefficients in an MB go through the same quantizer step size Q, which can be changed in increments of 2 from 2 to 62 as desired.

In the MPEG quantizer, each coefficient produced by 2D DCT is quantized with a uniform quantizer. The default quantizer matrix is defined as shown in Table 2.2, which can be changed if desired.

Typically, the DC coefficients of 8×8 blocks belonging to an intra MB are scaled by

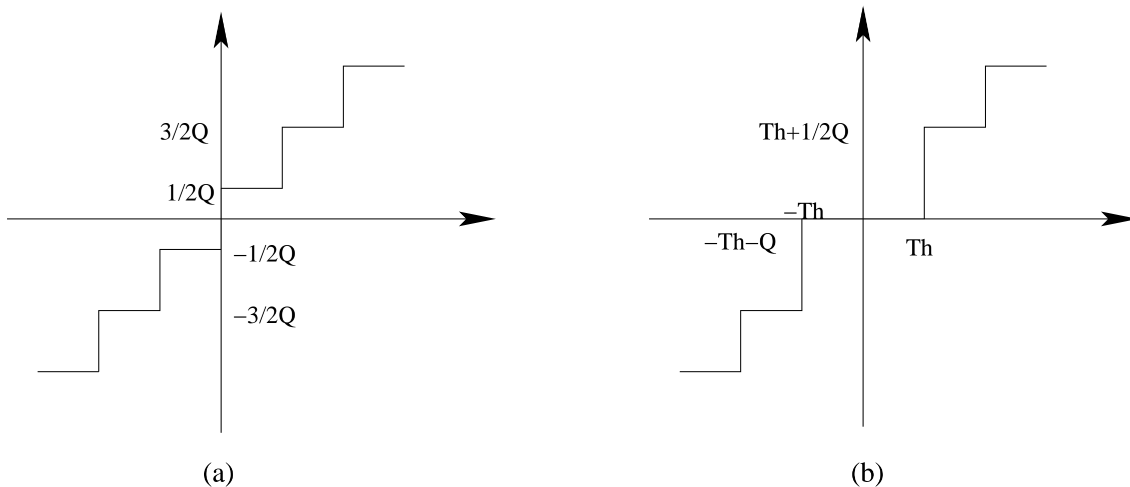
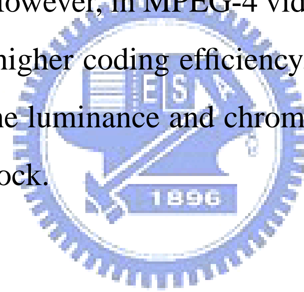


Figure 2.6: Quantizers in H.263. (a) For intra DC coefficient only. (b) For inter DC and all AC coefficients.

a constant scaling factor of 8. However, in MPEG-4 video, a nonlinear scaler as shown in Table 2.3 is used to provide a higher coding efficiency. The characteristics of nonlinear scaling are different between the luminance and chrominance blocks and further depend on the quantizer used for the block.



Intra Prediction

After quantization, the DC coefficients and many AC coefficients of an intra block are coded by intra prediction. Intra prediction is a new operation used in MPEG-4 standards to reduce the spatial redundancy between 8×8 blocks. There are two types of prediction, DC prediction and AC prediction.

Figure 2.7 shows the prediction of DC coefficients in intra 8×8 blocks. The quantized intra coefficients are predicted with three previous decoded DC coefficients. For example, the DC coefficients of block X is predicted from the DC coefficients of blocks A, B and C. Unlike MPEG-2, the prediction in MPEG-4 is gradient based. In computing the prediction of block X, if the absolute value of a horizontal gradient is less than the absolute value of a vertical gradient, then the QDC of block C is used as the prediction, else QDC value of block A is used.

The AC prediction depends on DC prediction, as shown in Fig. 2.8. The AC coeffi-

Table 2.2: Default Quantization Matrix (Q) [2]

Intra								Inter							
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

Table 2.3: Nonlinear Scaler for DC Coefficients (from [2])

Component	DC Scaler for Q Range			
	1–4	5–8	9–24	25–31
Luminance	8	2Q	Q+8	2Q+16
Chrominance	8	(Q+13)/2		Q+16

cients in the first row or in the first column are predicted with three previous decoded AC coefficients. The direction of prediction is the same as DC prediction.

Scan and VLC

The predicted DC and AC coefficients (as well as the un-predicted AC coefficients) of DCT blocks are scanned by one of three scans: alternate-horizontal, alternate-vertical and zigzag (the normal scan used in H.263 and MPEG-1) to change the 2D image to one dimensional data, as shown in Fig. 2.9. The actual scan used depends on the coefficient prediction method used. For instance, if the DC prediction refers to the horizontally adjacent block, alternate-vertical scan is selected for the current block. If the DC prediction refer to the vertically adjacent block, alternate-horizontal scan is used for the current block. For all other blocks, the 8×8 DCT blocks are zigzag scanned.

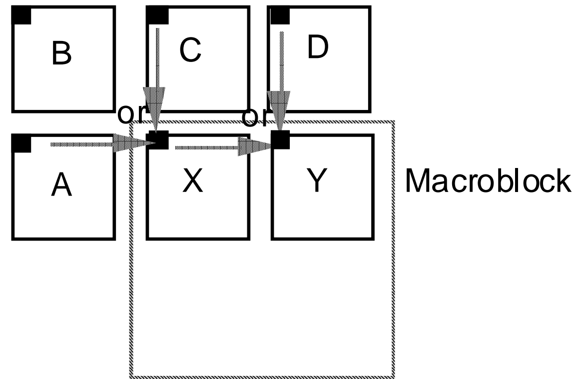


Figure 2.7: Prediction of DC coefficients of blocks in an intra MB (from [3]).

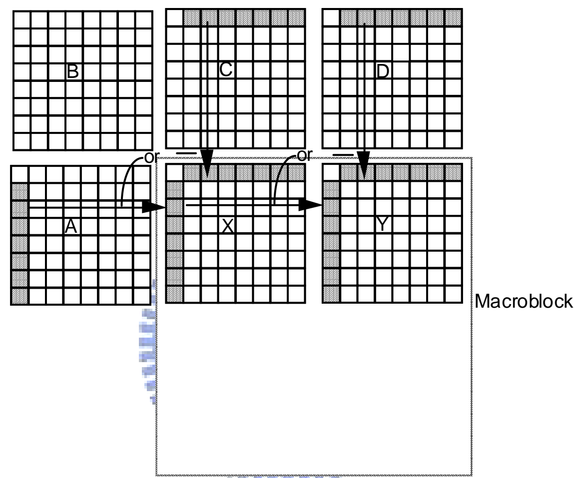


Figure 2.8: Prediction of AC coefficients of blocks in an intra MB (from [3]).

The coefficients after scan usually become data with many zeros at the end. This kind of a data stream is good for run-length coding. In MPEG-4, differential DC coefficients in intra blocks are encoded in VLC. But the AC coefficients are encoded by the variable length codes for EVENTs. An EVENT is a combination of a last non-zero coefficient indication, the number of successive zeros preceding the coded coefficient (RUN), and the non-zero value of the coded coefficient (LEVEL). Some statistically rare events have no VLC words to represent them. For them an escape coding method is used.

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

(a) Alternate-Horizontal scan

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

(b) Alternate-Vertical scan

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

(c) Zigzag scan

Figure 2.9: Scans for 8×8 blocks (from [2]).

2.3.2 Other Video Coding Tools [3]

In addition to texture video coding, there are some special tools defined in MPEG-4. We briefly introduce robust video coding and scalable coding here.

2.3.3 Robust Video Coding

Error resilience is a particular concern over wireless networks. In the error resilient mode, the MPEG-4 video offers a number of tools as follows:

1. Object priorities: The object based organization of MPEG-4 video facilitates prioritizing of the semantic objects based on their relevance. Further, the VOP types are a form of inherent prioritization since B-VOPs do not contribute to error propagation and thus can be transmitted at a lower priority or discarded in case of severe errors.

2. Resynchronization: The encoder can enhance error resilience by placing resynchronization (resync) markers in the bitstreams with approximately constant spacing, such as beginning of each MB.

3. Data partitioning: Data partitioning provides a mechanism to increase error resilience by separating the normal motion and texture data of all MBs in a video packet and send all of the motion data followed by a motion marker, followed by all of the texture data.

4. Reversible VLCs: The reversible VLCs offer a mechanism for a decoder to recover additional texture data in the presence of errors since the special design of reversible VLCs enables decoding of codewords in both the forward (normal) and the reverse direction.

5. Intra update and scalable coding: Intra update is a simple method to reduce error propagation. However, more intra updates means less coding efficiency. Another method is scalable coding, which can alleviate error propagation without more intra coding.

2.3.4 Scalable Coding

The scalability tools in MPEG-4 video are designed to support applications beyond that supported by single layer video, such as internet video, wireless video, multi-quality video services, video database browsing, etc. In scalable video coding, it is assumed that given a coded bitstream, decoders of various complexities can decode and display appropriate reproductions of coded video.

MPEG-4 video provides several different forms of scalability. The basic scalability tools offered are temporal scalability and spatial scalability. A Fine Granularity Scalability (FGS) is also defined which supports continuous scalability of bit rate and video quality.

2.4 Profiles and Levels [2]

Although there are many tools in the MPEG-4 standard, not every MPEG-4 decoder will have to implement all of them. Similar to MPEG-2, profiles and levels are defined as subsets of the entire bitstreams syntax of all the tools. The purpose of defining conformance points in the form of profiles and levels is to facilitate interchange of bitstreams among different applications. There are eight profiles defined in MPEG-4: simple, core, main, simple scalable, animated & mesh, basic animated texture, still scalable texture and simple face. The details are given in Table 2.4.

Compared with the previous standards, the simple profile of MPEG-4 is similar to the coding method in H.263. The difference is that the simple profile has error resilience but does not have B-frame coding. The simple scalable profile is simple profile with rectangular scalability. 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 purposes, such as 2D dynamic mesh coding and facial animation coding.

For frame-based coding and decoding, what concerns us is the main profile, excluding the shape coding, interlace, and sprite coding tools.

Table 2.4: Profiles and Tools (from [2])

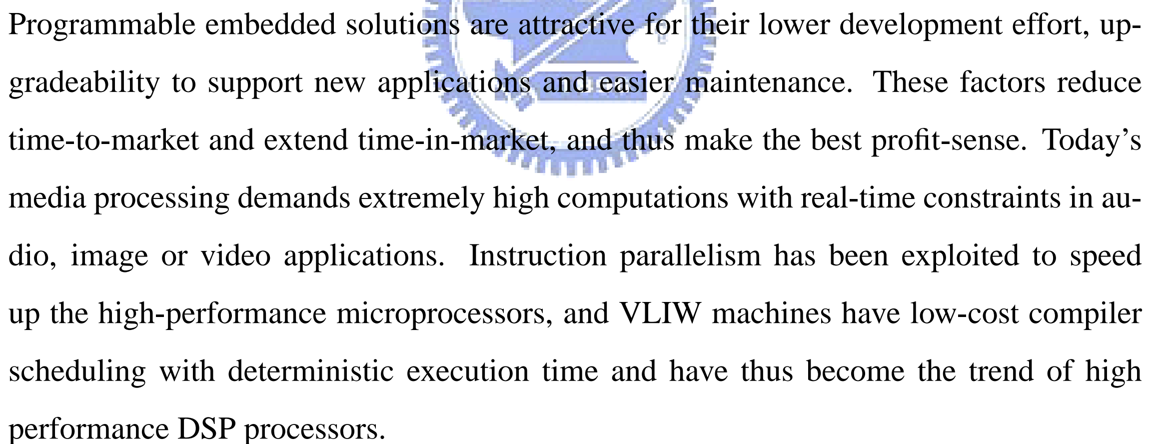
Tools	Simple	Core	Main	Simple Scalable	Animated 2D Mesh	Basic Animated Texture	Still Scalable Texture	Simple Face
Basic *1. I VOP* *2. P VOP* *3. AC/DC Prediction* *4. 4MV Unrestricted MV*	V	V	V	V	V			
Error resilience *1. Slice Resynchronization* *2. Data Partitioning* *3. Reversible VLC*	V	V	V	V	V			
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. Rectangular* *2. Arbitrary Shape*		V	V		V			
Binary Shape		V	V		V			
Gray 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

Overview of The PACDSP

The contents of this chapter have been taken to a large extent from [1].

3.1 Introduction



Programmable embedded solutions are attractive for their lower development effort, upgradeability to support new applications and easier maintenance. These factors reduce time-to-market and extend time-in-market, and thus make the best profit-sense. Today's media processing demands extremely high computations with real-time constraints in audio, image or video applications. Instruction parallelism has been exploited to speed up the high-performance microprocessors, and VLIW machines have low-cost compiler scheduling with deterministic execution time and have thus become the trend of high performance DSP processors.

Conventional VLIW processors are notorious for their poor code density, because the unused instruction slots must be filled by NOPs. Thus, the code density gets worse when the parallelism is limited. Variable-length VLIW instruction packet eliminates NOPs by dispatching instructions at run-time, compared to the conventional position-coded VLIW processors where each functional unit (FU) has a corresponding bit-field in the instruction packet. Indirect VLIW has an internal instruction buffer for the VLIW instruction packets. With this instruction buffer and the pre-fetch scheme, the VLIW processor can reduce instruction memory bandwidth requirement and power consumption of instruction

fetches.

The complexity of the register file (RF) grows exponentially as more and more FUs are integrated on a chip, which operate concurrently to achieve the performance requirements. The RF is frequently partitioned for execution clusters with explicit interconnection networks among the clusters to significantly reduce the complexity at the cost of small performance penalty.

For high performance, the PACDSP is a VLIW processor with single instruction multiple data (SIMD) instruction set architecture (ISA). The software supported schedule reduces the complexity of hardware design and the power consumption. Variable length instruction and instruction packet solve the poor code density problem of the conventional VLIW architecture. Another feature of the PACDSP, cluster architecture, reduces not only ports and entries of the register files but also the power consumption of read/write operations. More details about the features of PACDSP are discussed in the following sections.

3.1.1 Architecture Features

Key features of the PACDSP include the following items:

- Scalable VLIW datapath for easy extension of the performance.
- Variable instruction word/packet length to avoid the drawback of poor code density in the conventional VLIW architecture.
- Heterogeneous register files for more straightforward operations, less ports and smaller entries in each RF to improve the performance and reduce power and area.
- Constant register file in each cluster (32×32 bits) for storage of some fixed data in the applications to reduce the frequency of data movement which may cost significant of power consumption.
- Inter-cluster communication (ICC) by memory controller for reusing hardware resource and reducing the port number of ping-pong RF in order to reduce power and area and to increase the scalability.

- Optimized interrupt design with fast interrupt response time (3 clock cycles) with hardware supporting context switch to reduce the processing time of interrupt service routine (ISR).
- Hierarchical encoding scheme reducing the dependency between instructions and packets to reduce area and latency of the dispatch unit.
- Dynamic power management for power saving.
- Customized instruction set and functional unit interface for the accelerators that are used to enhance certain DSP operations.

3.2 Architecture Overview

There are three components in the PACDSP kernel: program sequence control unit, scalar unit and VLIW datapath. The accelerators that execute in different threads and synchronize the execution results through the scalar unit can enhance the computation power of the VLIW datapath. Figure 3.1 shows the architecture of the PACDSP.

The program sequence control unit dispatches instructions to the scalar unit and the VLIW datapath. It also executes control flow instructions and handles the interrupt and exception events. The scalar unit executes the scalar instructions whose characteristics are low parallelism and high data dependency. It also controls the power control interface and the customized functional unit interface.

The VLIW datapath composed of two clusters takes charge of complex data operations in the program. Each cluster contains a load/store unit (L/S) and an arithmetic unit (AU). Both units can execute instructions concurrently. Another feature of the PACDSP, the ping-pong register file, facilitates data transfers between these two units. With this feature, the typically high power consumption of the DSP kernel can be reduced. The maximum parallelism of the VLIW datapath in instruction and operation levels is 4 and 12, respectively.

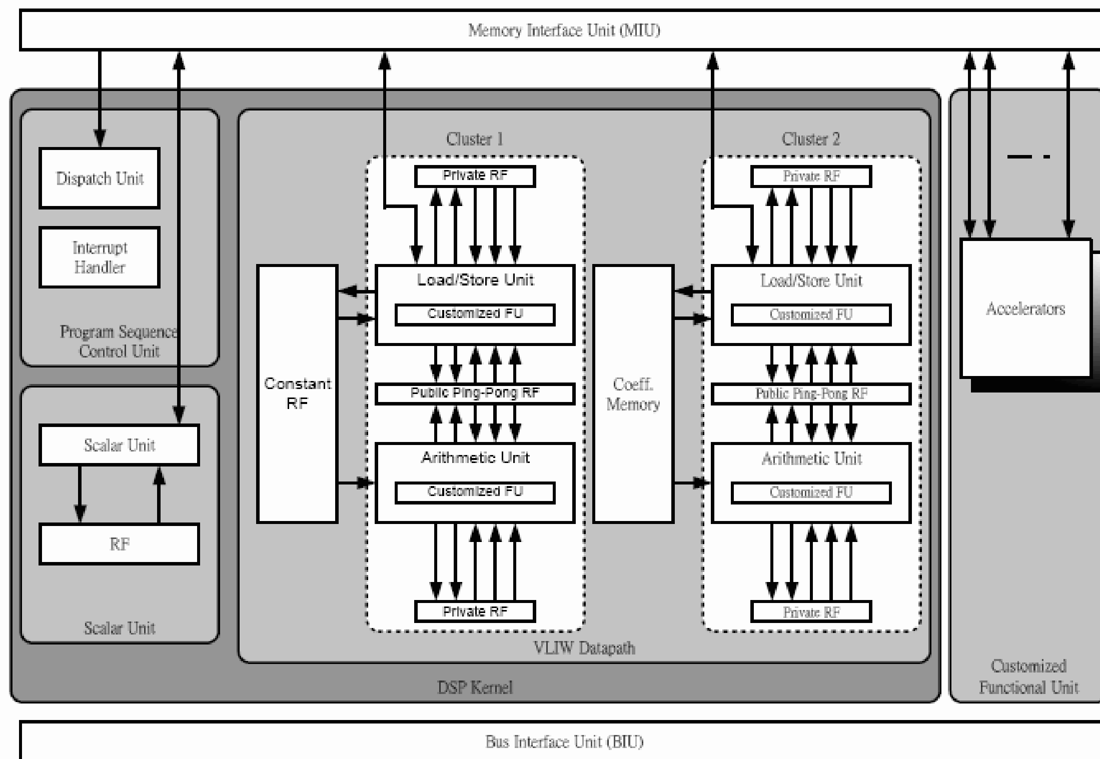


Figure 3.1: Architecture of the PACDSP [1].

3.3 Program Sequence Control Unit

The program sequence control unit is a main component in the DSP kernel. It dispatches instructions to the scalar unit and the VLIW datapath. It also executes the execution flow control instructions and handles the interrupt and exception events.

3.3.1 Branch Instruction

Branch instructions can be grouped into two categories, conditional branches and unconditional branches. There are three addressing modes defined in the PACDSP for generating the branch target address:

- PC-relative

  Add the 16-bit signed immediate offset to the address in the PC register, and take the result as the branch target address, i.e.,

$$TA = PC + OFFSET$$

where TA is the target address, PC is the address in PC register, and OFFSET is the 16-bit signed immediate value.

- Register

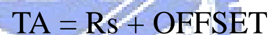
  Take the value in the register as the target address, i.e.,

$$TA = Rs$$

  where TA is the target address and Rs is the source register of address.

- Register-relative

  Add the 16-bit signed immediate offset to the address saved in the register and take the result as the branch target address, i.e.,



$$TA = Rs + OFFSET$$

  where TA is the target address, Rs is the source register saving the address, OFFSET is the 16-bit signed immediate value.

The branch instructions defined in the PACDSP support saving of the return address into the assigned register. The programmer should take care of the return addresses of nested loops. There are three branch delay slots in the PACDSP, and the independent instructions can be put in these delay slots.

3.3.2 Loop

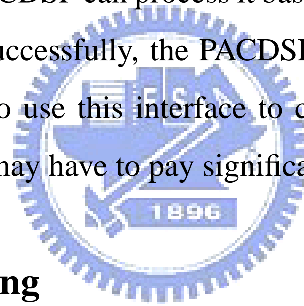
The programmer can use the LBCB instruction to effect program loops. Loop Boundary Register (RBC0 – RBC3), which are all 32-bit registers, can be used to record the loop counts. However, the maximum loop count is 65536 for each level. Since there are four Loop Boundary Registers, up to four levels of nested loop can be supported with the use of the LBCB instruction.

There is a constraint in using LBCB to control a nested loop. The outer loop should fully contain the inner loop. No exception will be generated if the constraints are violated, but the program behavior may be different from expectation.

However, conditional branches can be used inside the nested loop to implement some special branch behaviors in higher level languages, for example, “break” and “continue” in C.

3.3.3 Customized Function Unit (CFU)

The PACDSP provides Customized Function Unit Interface for extension purpose. The user can attach co-processors or customized function units to PACDSP and handle them through the scalar instructions. If some error happens in a customized function unit, it can inform the PACDSP and the PACDSP can process it based on the particular configuration. If the work given is finished successfully, the PACDSP can use its results and continue to work. It is recommended to use this interface to communicate with any added co-processor; otherwise, the user may have to pay significantly more effort to handle it.



3.3.4 Exception Handling

Unpredictable exceptions may occur during program execution. The exceptions need to be handled correctly for correct execution results. Exceptions may be caused by hardware (e.g., overflow), software, internal (e.g., undefined instruction), or external (e.g., coprocessor exception). When an exception happens, the DSP kernel will be frozen or listen to the main processing unit (MPU) deliverance. It is still aware of debug requests and will check the corresponding signal to see what kind of exceptions have happened.

3.3.5 Interrupt Handling

Two types of interrupt are supported by the PACDSP. One is fast interrupt request (FIQ), which has the higher priority, and the second is interrupt request (IRQ). The difference between them is that the FIQ uses hardware to reduce the time in saving the context and

the hardware resources used for the FIQ interrupt service routine (ISR) consist only of the scalar unit and program sequence control unit.

Contrarily, the IRQ can use all the hardware resources in PACDSP to deal with the IRQ request, but the ISR of IRQ needs to save the context by itself.

In the PACDSP, the minimum latency from interrupt request to the first ISR instruction to be executed is 3 cycles for both types of interrupt, and it may be postponed when the ISR experiences cache miss.

3.4 VLIW Datapath

3.4.1 Ping-Pong Register File

A centralized register file (RF) provides storage for and interconnects to each functional unit (FU), and each FU can read from or write to any register location. But in practical designs, the communication between FU is usually restricted by partitioning the RF to reduce the complexity significantly with some performance penalty. In other words, each FU can only read and write a limited subset of registers. In the ping-pong hierarchical RF, which is shown in Fig. 3.2, the RF is partitioned into private and ping-pong sub-blocks. Each FU (L/S or AU) can simultaneously access two sub-blocks, one of which is private (i.e., dedicated to the FU) and the other is dynamically mapped for inter-FU communications within one cluster. Therefore, each sub-block only requires the access ports for a single FU. The shared sub-blocks are organized in a ping-pong fashion to reduce the control overheads, where the dynamic mapping is exposed to the VLIW ISA with two switching bits and is directly specified by the programmers for each instruction packet.

3.4.2 Data/Address/Accumulator Registers

As shown in Fig. 3.3, the address registers (A0–A7) are all 32-bit and they are dedicated to the load/store unit (L/S) for memory accesses. In addition, A1, A3, A5, and A7 are also treated as the base registers which contain the base addresses in modulo addressing mode.

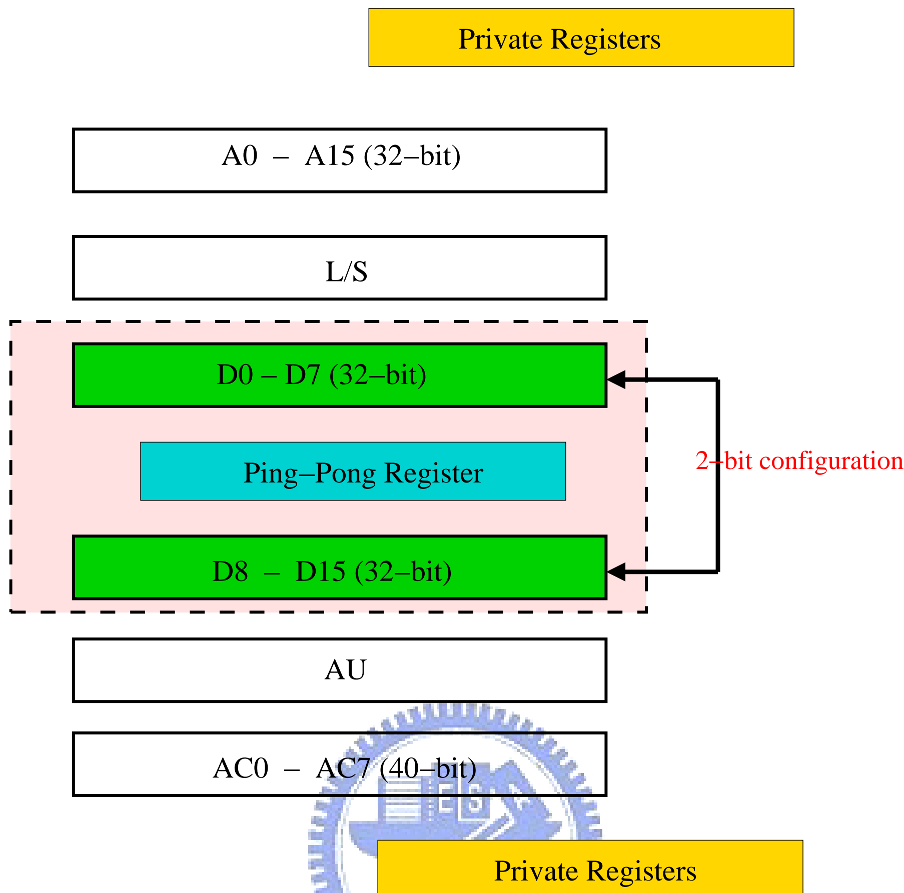


Figure 3.2: Ping-pong register file in one cluster [1].

E0–E3 (A8, A10, A12, and A14) and D0–D3 (A9, A11, A13, and A15) are individually treated as end registers and displacement registers which contain end addresses and displacements in modulo addressing mode. Nevertheless, in linear addressing mode, they can be treated as the address register like A0–A7. The accumulator registers (AC0–AC7) are 40-bit (8-bit as guard bits) and are dedicated to the arithmetic unit(AU) for data manipulations. The data registers(D0–D7 and D8–D15) are organized in the form of ping0pong with 1-bit control and the word-length of these registers are 32-bit.

3.4.3 Status and Control Registers

The status register and control register which are read and set by instructions can be used to monitor the DSP kernel status and handle the operation mode of DSP kernel.

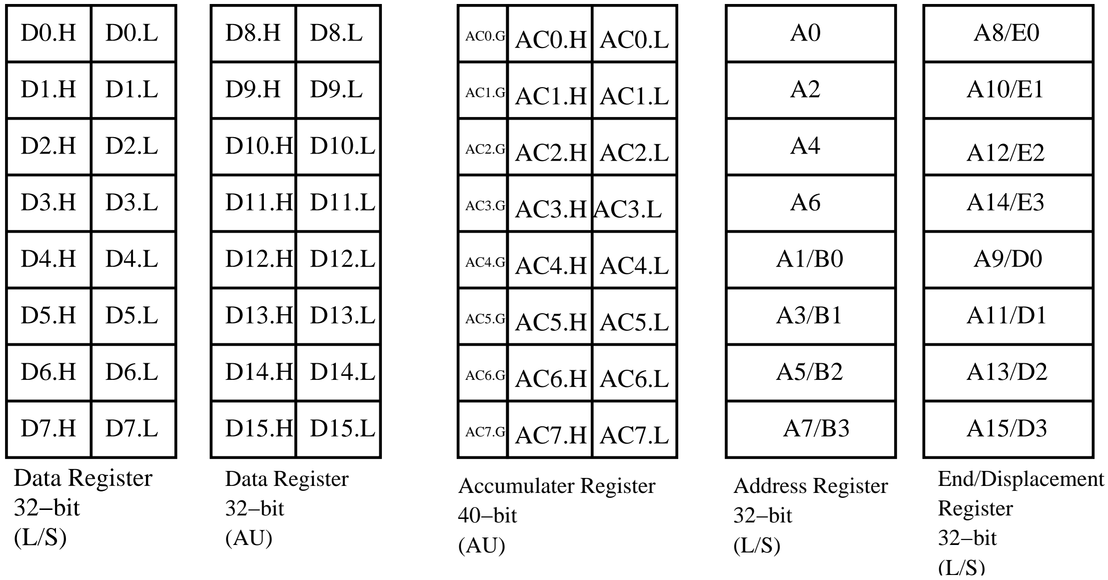


Figure 3.3: The available registers in one cluster [1].

Program Status Register

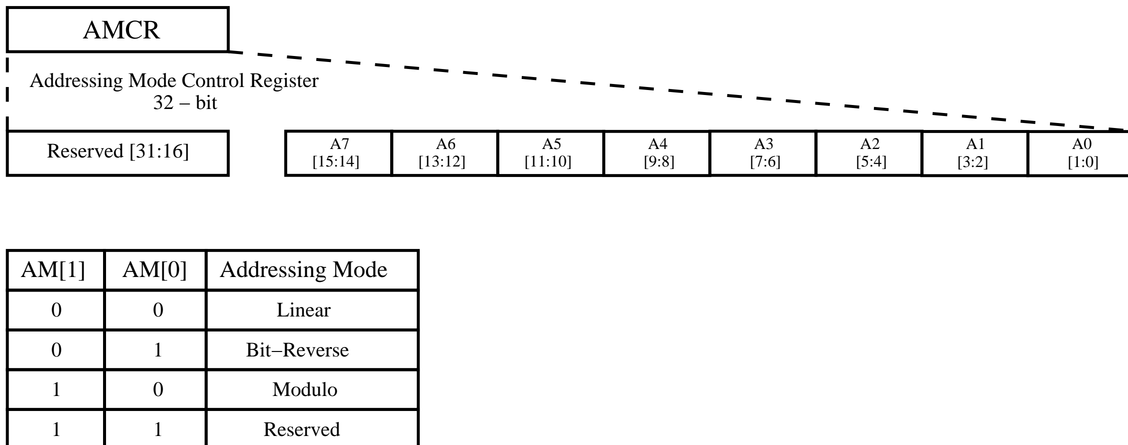
The 16-bit program status register records the operation status in each cluster and the scalar unit. It includes Overflow, Negative, and Carry bits, and instructions can only read the status register, not set it.

Addressing Mode Control Register (AMCR)

The PACDSP provides three types of addressing modes:

- Linear addressing mode,
- Bit-reverse addressing mode,
- Modulo addressing mode.

As shown in Fig. 3.4, the addressing mode control register (AMCR) is a 32-bit read/write register. This register is used to control the addressing mode of relative address registers. The addressing modes are related to where the operands are to be found and how the address calculations are to be made.



AM[1]	AM[0]	Addressing Mode
0	0	Linear
0	1	Bit–Reverse
1	0	Modulo
1	1	Reserved

Figure 3.4: Illustration of the addressing mode control register (AMCR) [1].

3.4.4 Addressing Modes

The addressing modes are related to where the operands are to be found and how the address calculations are to be made.

Linear Addressing Mode

There are three kinds of linear addressing mode, which are register direct mode, address register indirect mode, and immediate data mode.

The register direct addressing mode specifies that the operand is in one or more of the arithmetic unit (AU) registers, load/store unit (L/S) registers, control registers and program counter (PC) registers. This addressing mode is also used to specify a control register operand and a PC register operand for special instructions.

The address register indirect mode specifies that the address register is used to point to a memory location. The term indirect is used because the register contents are not the operand itself, but the operand address. This addressing mode specifies that an operand is in a memory location and specifies the effective address of that operand. There are still two sub-modes in the address register indirect mode:

- Pre-increment, +(Rs) offset

  The operand address is the sum of the contents of the address register and the offset. The data stored at the address of the sum of register value and offset will be loaded.

- Post-increment, (Rs)+ offset

  The operand is in the address register Rs. After the operand address is used, it is incremented by the offset and stored in the same address register. Incrementing the operand address by the offset places the next available address in the register. That is, the data stored at the location of the address register will be loaded first, and then the address is updated with the offset.

The immediate data mode does not use an address register. The instructions use an immediate value that is included in the instruction for the data value or address value.

Bit-Reverse Addressing Mode

Bit-reverse addressing mode is also called reverse-carry addressing mode. It is useful for 2^k-point FFT addressing. This mode is selected by setting the corresponding bits in AMCR, and address modification is performed in the hardware by propagating the carry from each pair of added bits in the reverse direction (from the MSB end toward the LSB end). It can also use the pre- or post-increment addressing mode.

This address modification is useful for addressing the twidle factors in 2^k point-FFT addressing as well as to unscramble 2^k-point FFT data.

Modulo Addressing Mode

Modulo address modification is useful for creating circular buffers for FIFO queues, delay lines, and sample buffers.

The definition of modulo addressing, using a base register (Bn) and a modulo register (Mi), enables the programmer to locate the modulo buffer at any address. The address pointer, An, is not required to start at the lower address boundary, nor to end on the upper address boundary. It can initially point to anywhere (aligned to its access width) within the defined modulo address range, $Bn \leq An < Bn + Mi$.

Modulo addressing can be selected by configuring corresponding bits in AMCR, and write the desired modulo to modulo registers. The range of modulo registers, Mi, is from 1 to $2^{32} - 1$.

Each base address register (Bn) is associated with an address register (B0 with A0, and so on). Offset and modifier registers are also associated with the corresponding address registers in the same way.

3.4.5 VLIW Datapath

The VLIW datapath of PACDSP is constructed in two clusters, and each contains an arithmetic unit (AU) and a load/store unit (L/S) as shown in Fig. 3.2. Therefore, it can execute four instructions simultaneously, and is thus called a four-way VLIW datapath.

Arithmetic Unit (AU)

The arithmetic unit (AU) comprises four 40-bit adders which can be reconfigured to two 16-bit adders or four 8-bit adders, two 16-bit multipliers, one shifter and one logical ALU. All data processing instructions in AU begin at the same stage, but not finish at the same time.

There are three types of precision in DSP — full, integer, and fractional. Figure 3.5 shows how it works.

- Full precision: Rd = Rs1.L $\times$ Rs2.L.
- Integer: Rd.L = (Rs1.L $\times$ Rs2.L)[15:0].
- Fractional: Rd.L = Rs1.L $\times$ Rs2.L)[30:15].

Load/Store Unit (L/S)

The load/store unit (L/S) comprises one address generation unit (AGU), one logical ALU, and one shifter. Similar to AU, all instructions in L/S begin at the same stage, but not finish at the same time.

The L/S unit supports powerful double load/store instructions, which can load or store two operands in one instruction. Figure 3.6 shows how double and vector load/store work.

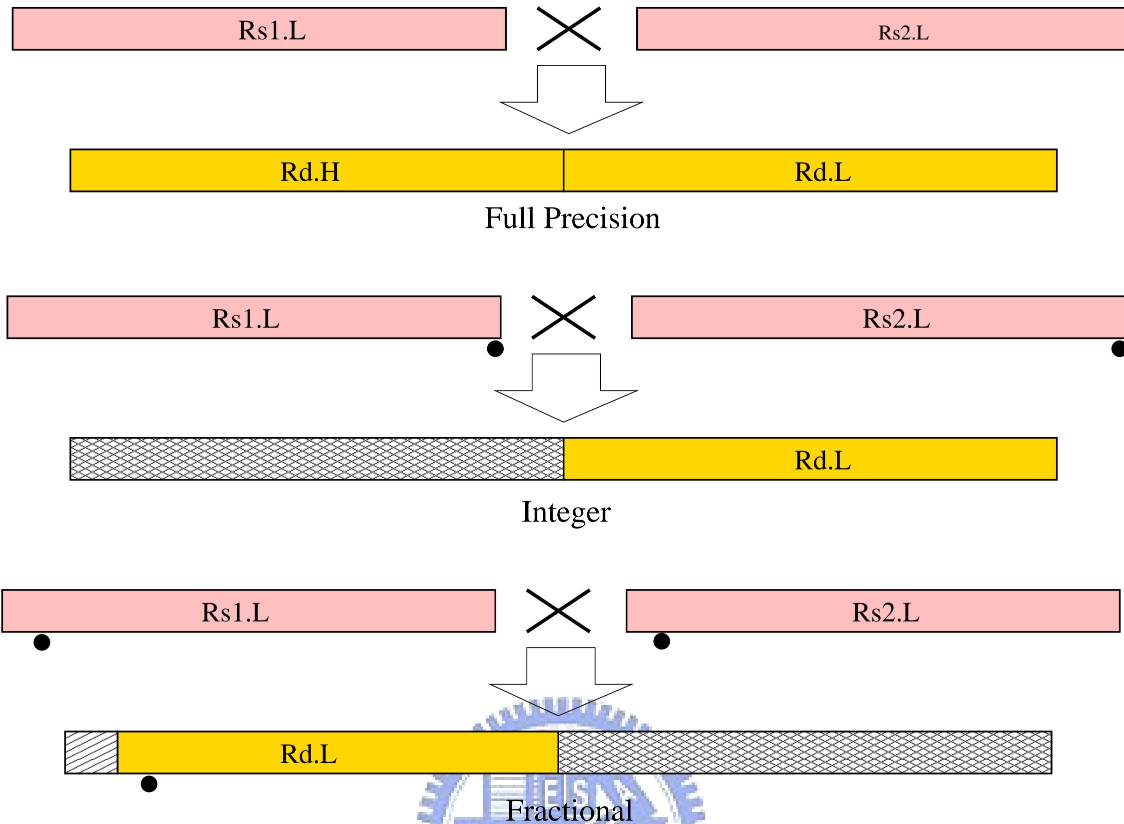


Figure 3.5: Illustration of multiplication instructions with different precisions [1].

3.4.6 Data Exchange

As shown in Fig. 3.7, the PACDSP provides a data exchange mechanism between any two of the scalar unit and the two clusters. Figure 3.8 shows that it can also provide data broadcast to facilitate one of them to broadcast its data to the others even though the number of clusters may be extended someday. This job is accomplished by using the ports of the memory interface unit (MIU) because MIU has connections with all register files of the scalar unit and the two clusters.

Data Exchange Between Clusters

The PACDSP provides a special instruction (DEX) to accomplish data exchange between clusters. For example:

Cluster1 instruction: DEX D1, D0

Cluster2 instruction: DEX D1, D2

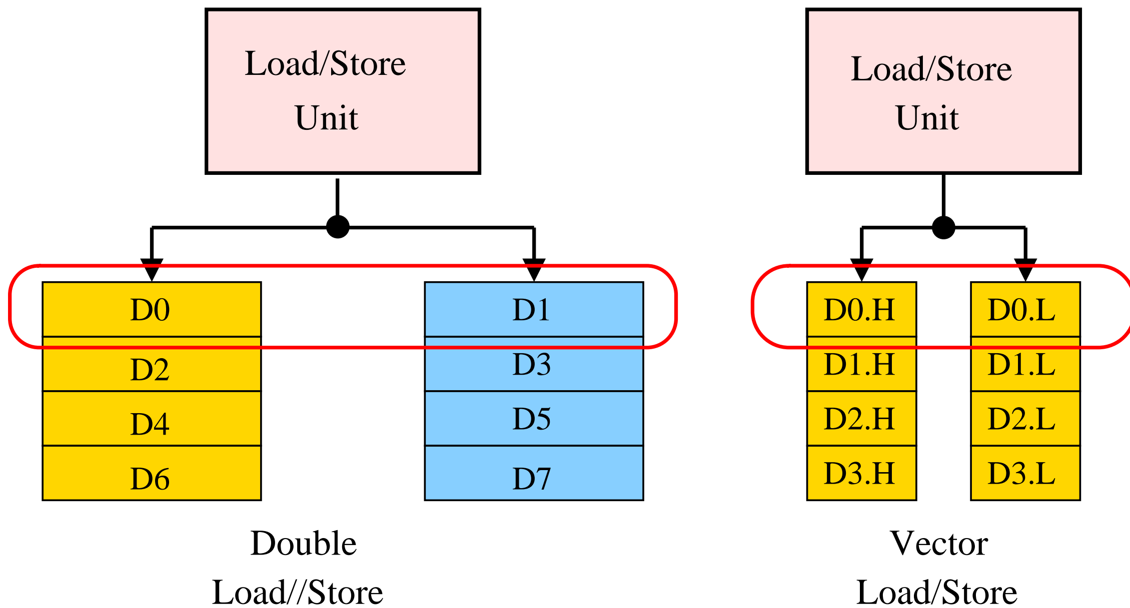


Figure 3.6: Different load/store instructions [1].

At compile time, this instruction pair will cause direct exchange of the contents of D0 and D2 through MIU and each cluster will store them in D1, as shown in Fig. 3.7.

Data Broadcast

Like data exchange between clusters, PACDSP also provides a special instruction pair (BDT and BDR) for data broadcast from one cluster to the others. For example:

Cluster1 instruction: BDT D0

Cluster2 instruction: BDR D3

Scalar instruction: BDR R0

At compile time, this set of instructions will broadcast data from cluster1 to cluster2 and the scalar unit as shown in Fig. 3.8.

On the other hand, if we just want to transmit data from one cluster to another (including the scalar unit), it can be considered a special case of data broadcast. For example:

Cluster1 instruction: ADD D0, D1, D2

Cluster2 instruction: BDR D7

Scalar instruction: BDT R0

In this example, the content of R0 is transmitted to D7 in cluster2. At the same time,

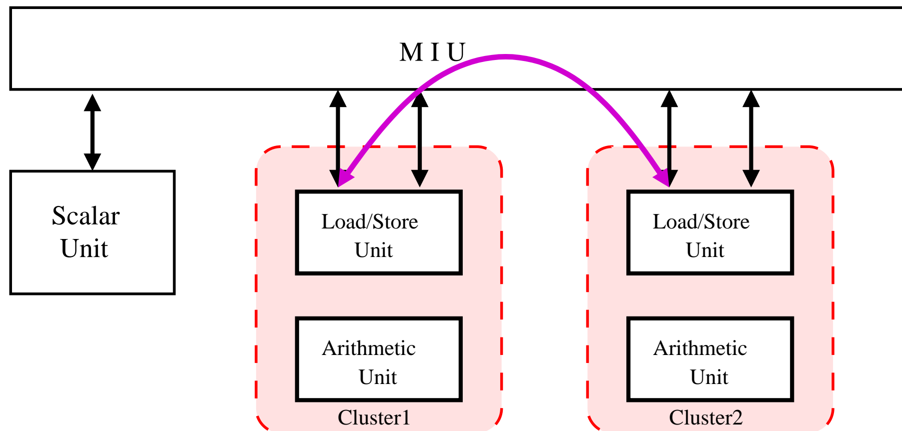


Figure 3.7: Data Exchange between Two Clusters [1].

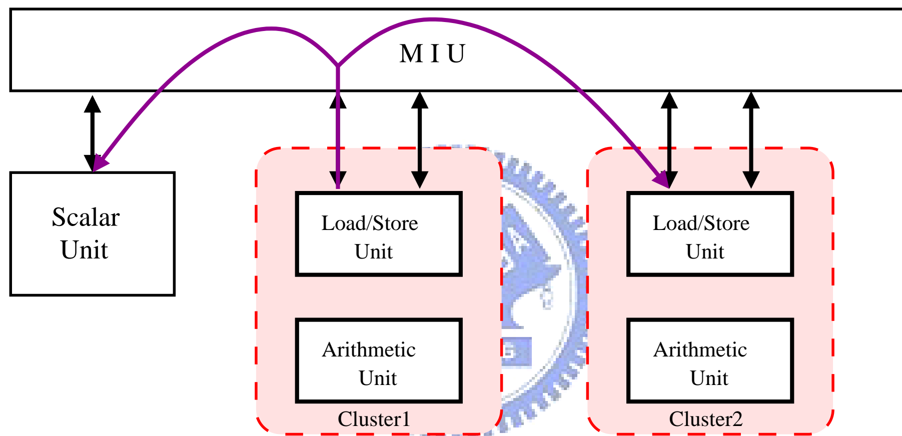


Figure 3.8: Data broadcast among clusters [1].

cluster1 can do other operations without affecting by this transmission.

3.4.7 Constant Register File

In many DSP algorithms, such as FIR, IIR, etc., there are many coefficient operations which use fixed data. In order to avoid high frequency of data movement in the register file, the PACDSP provides a small size memory, called Constant Register File to maintain the fixed data. We can also use it to store look up tables which contain fixed data for specific applications. It can reduce the frequency of data movement and thereby reduce power consumption in such operations.

Data contained in the Constant Register File can be used to do operations including

comparison, multiplication, multiplication and accumulation, etc. They are used as the second source operand in the instructions.

The specifications of Constant Register File (in one cluster) are as follows:

- 32×32 bits.
- Two read ports and one write port.

As shown in Fig. 3.9, the Constant Register File is initialized through the write port by MIU at the beginning of the program. Not only the L/S but also the AU has a read port for taking its value as one source operand. There are some rules when using the Constant Register File:

- It can only be modified by particular instructions in L/S.
- Read and write operations may not occur at the same time in L/S.

3.5 Scalar Unit

3.5.1 Overview

The Scalar Unit can perform three types of function, which are basic arithmetic operations, word and halfword-based load/store operations, and read/write operations performed on the contro/status registers.

Under some running modes, the DSP core may execute a program without activating the VLIW clusters. In this case, the scalar unit acts like a simple machine, handling some easy tasks.

Mostly, the scalar unit is in charge of the control-based work while the VLIW clusters are dealing with data processing. Data can be exchanged between the scalar unit and the VLIW clusters.

3.5.2 Control Registers

In the PACDSP kernel, there are 15 control registers. Table 3.1 shows the names and the widths of all the control registers in the PACDSP kernel.

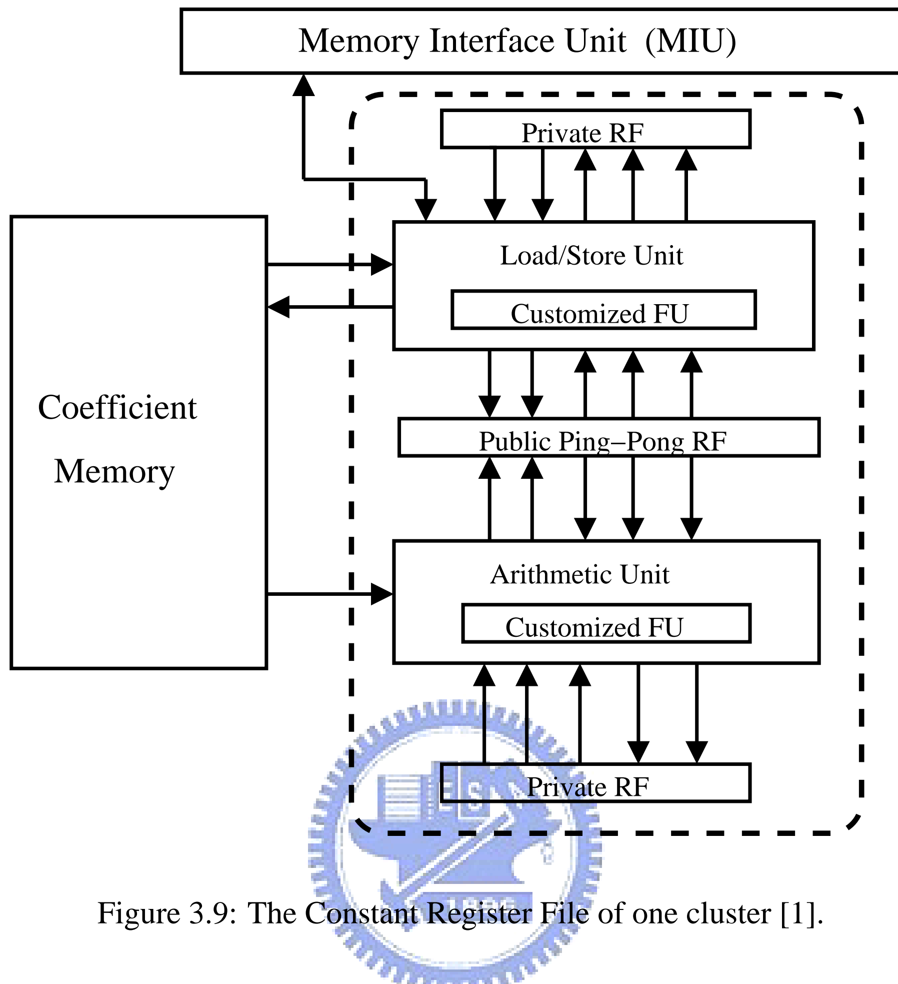


Figure 3.9: The Constant Register File of one cluster [1].

Several control registers are memory mapped and can be accessed by others outside the PACDSP kernel. Table 3.2 lists the memory mapped control registers and the mapping memory addresses.

The control registers can be read or write by the scalar instructions. When writing the control registers, we can assign a 16-bit immediate value to the destination, or set a general purpose scalar register as the source operand.

3.5.3 General Purpose Scalar Register File

In the scalar unit of the PACDSP kernel, there are sixteen 32-bit general purpose registers named R0 to R15.

Table 3.1: Details of Control Register Files [1]

Type	No	Name	Size(bits)	Note
Control	CR0	PREDN	16	Prediction information
	CR1	EN_INT	1	Interrupt enable flag
	CR2	MSK_EX	16	Mask inside exception
	CR3	SWI_EX	16	Software exception
	CR4	CF0	32	Custom function register 0
	CR5	CF1	32	Custom function register 1
	CR6	CF2	32	Custom function register 2
	CR7	CF3	32	Custom function register 3
Interrupt	CR8	SD_MIXIFN0	32	Mix information 0's shadow register
	CR9	SD_Rbc1	32	Loopboundary counter's shadow register1
	CR10	SD_Rbc2	32	Loopboundary counter's shadow register2
	CR11	SD_BCTG	32	Branch target shadow register
	CR12	SD_CPC	32	CPC's shadow register (ISR return address)
	CR13	SD_PREDN	16	Prediction's shadow register
	CR14	SD_R0	32	R0's shadow register
	CR15	Reserved		

3.6 Conditional Execution Control

Unlike general purpose processors, the major mission of a DSP is to provide more computing power for calculations. To reduce control overhead, the PACDSP supports conditional execution of instructions. Programmers can set predicates by Compare-and-Set instructions and then the instructions afterward can refer to the predicates to decide whether to execute or not. When the program calls a function, we can save the predicates and restore them after returning from the function call.

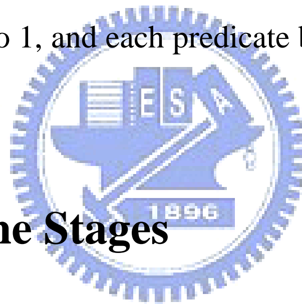
The Compare-and-Set instructions, such as SLT, SGT, etc., compare source operands

Table 3.2: Memory-Mapped Control Registers [1]

No	Name	Size	Note	Offset	R/W
00	Exception_Cause	32	Indicate inside exception cause	0x50020	R
01	Busy	1	DSP is busy	0x5000C	R
02	Start	1	Start signal	0x50008	R/W
03	Start_PC	32	Starting address	0x50000	R/W
04	MODE	4	DSP running mode	0x50040	R
05	VERSN	4	DSP version	0x50044	R

and save the results to the predicate registers, and the comparison results can be saved to the general purpose registers at the same time. The PACDSP provides 16 predicate bits (P0–P15), and a Compare-and-Set instruction updates 2 predicate bits at the same time.

However, P0 is always set to 1, and each predicate bit can be set by only one instruction at the same time.



3.7 ISA and Pipeline Stages

As said, the PACDSP architecture consists of the program sequence control unit, the scalar unit, and the VLIW datapath. Each of the three has corresponding function units. Therefore, the instruction set of PACDSP is classified according to the functional unit in which the instruction is executed. Figure 3.10 depicts the instruction set architecture (ISA) of the PACDSP.

Figure 3.11 shows the pipeline stages of the PACDSP. The program sequence control can be divided into three stages, which are IF, IDP, and ID. The scalar unit operation and the VLIW datapath are both divided into five stages, which are RO, EX1, EX2, EX3, and WB. The job of each pipeline stage is as described in Table 3.3.

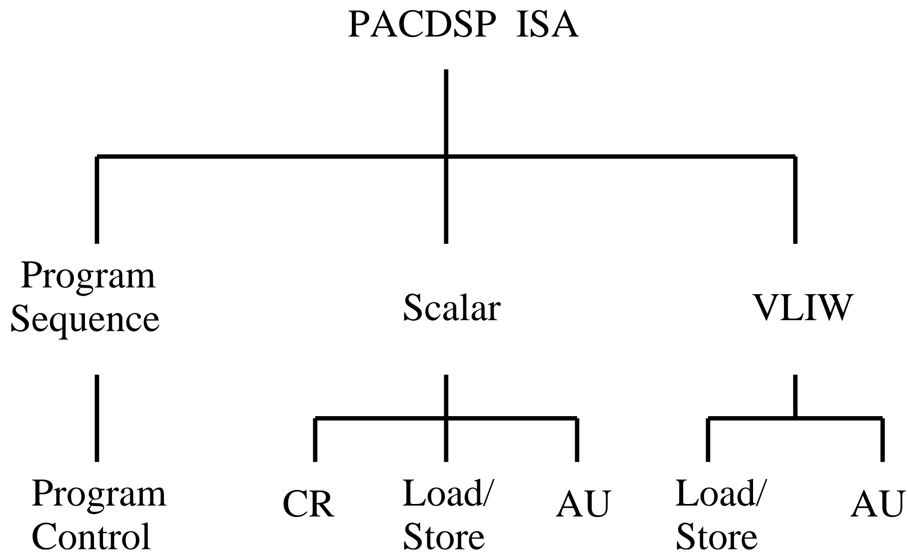


Figure 3.10: PACDSP instruction set architecture [1].

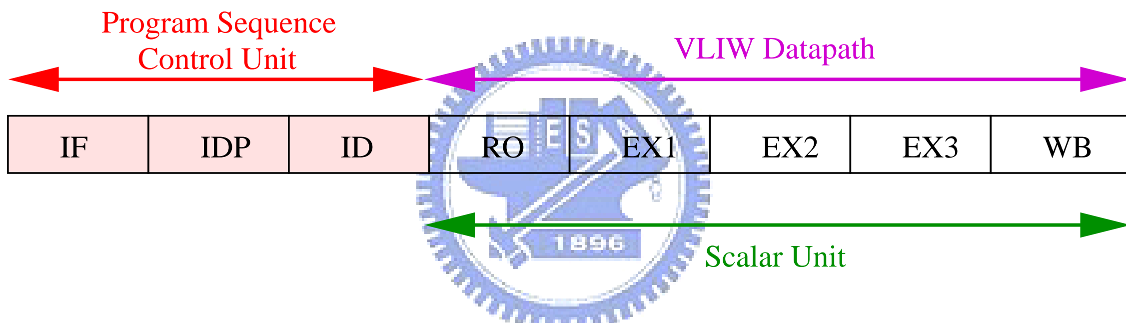


Figure 3.11: Pipeline stages of the PACDSP [1].

3.8 DSP Running Modes

The PACDSP can work under various running modes. Each mode has different hardware utilization. There are 7 different running modes. The corresponding hardware resource and a simple description of each running mode is given in Table 3.4.

It is noted that not all running modes can be chosen to be entered by the instructions. We can only change the three sub-modes of the the user mode by the instructions. The transitions between running modes are shown in Fig. 3.12.

Table 3.3: Pipeline Stages and Their Descriptions

Stage	Description
IF	Instruction Fetch
IDP	Instruction Dispatch
ID	Instruction Decode
RO	Read Operand
EX1	Execution One
EX2	Execution Two
EX3	Execution Three
WB	Write Back

3.9 Instruction Packet

The PACDSP can issue up to 5 instructions in one cycle. Instructions issued in the same cycle are packeted into an instruction packet. The five slots of the instruction packet and the types of instruction that can be contained in each slot are listed in Table 3.5.

The whole instruction packet is bounded by brackets, and slots within packet are separated by new-line characters. Figure 3.13 shows the syntax of a complete instruction packet. However, an instruction packet is allowed to be written in a single line, and be separated by a pipe character "|". The simplified syntax is shown in Fig. 3.14. It is noted that a NOP instruction should be placed in the slot where there is no instruction to be executed.

3.10 Development Tools and Implementation Considerations

3.10.1 Development Tools

We have a C-compiler ported from the well-known Open-Research-Compiler (ORC) on linux systems, and we can give parameters to optimize the performance of compiler. How-

Table 3.4: Running Modes of the PACDSP [1]

Running Modes		Description	Resources
Idle Mode		Idle after reset or trap	Execution control and interrupt interface
User Mode	High Performance	Process program which needs all resources	All available
	Medium Performance	Process program which does not need all resources	All except Cluster 2
	High power saving	Process FIQ ISR or scalar program	All except Cluster 1 and Cluster 2
Wait Mode		Wait for Customized Function Unit result	CFU, interrupt, debug interface, and exception handling unit
Frozen Mode		Froze DSP since exceptions happened	Debug and interrupt interface, exception handling unit
Debug Mode		Debugging	Debug interface, register files

ever, we can choose only one optimization level to the current status. In addition, base utilities are ported from the GNU binutils, and there are assembler, linker, and other object handling tools. The debugger is ported from the GNU GDB, and GDB is an abbreviation of GNU project debugger. The debugger can be connected to both the instruction set simulator (ISS) and embedded ICE. These tool chains are developed by Programming Language Laboratory of National Tsing Hua University in Hsinchu, Taiwan, R. O. C..

The ISS is developed by SoC Technology Center (STC) of Industrial Technology Research Institute of Taiwan, R. O. C.. The input file of the simulator is split through a parsing tool, “as2tic”, which parses the assembly code into two parts, data and instruction. We can configure the ISS to decide which kinds of information we want to print out to files. All the registers can be shown in each cycle, but the printable memory range is 8

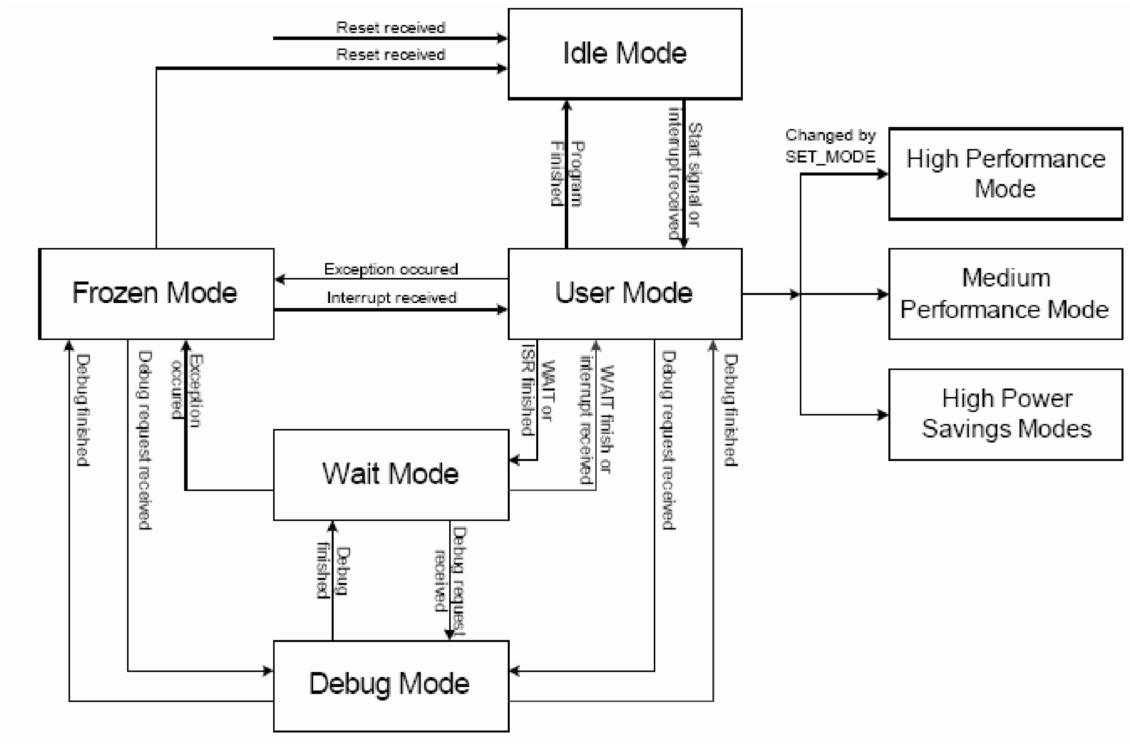


Figure 3.12: Transitions between DSP running modes [1].

Table 3.5: Instruction Type in Each Instruction Slot

Instruction Slot	Instruction Types
1 (Scalar Unit)	Program Sequence Control Instructions
2 (Cluster1)	VLIW Load/Store Instructions
3 (Cluster1)	VLIW Arithmetic Instructions
4 (Cluster2)	VLIW Load/Store Instructions
5 (Cluster2)	VLIW Arithmetic Instructions

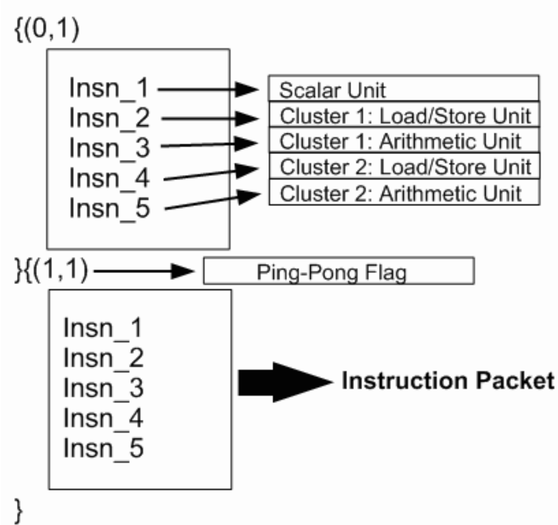


Figure 3.13: Syntax of instruction packet [1].

Figure 3.14: Simplified syntax of instruction packet [1].

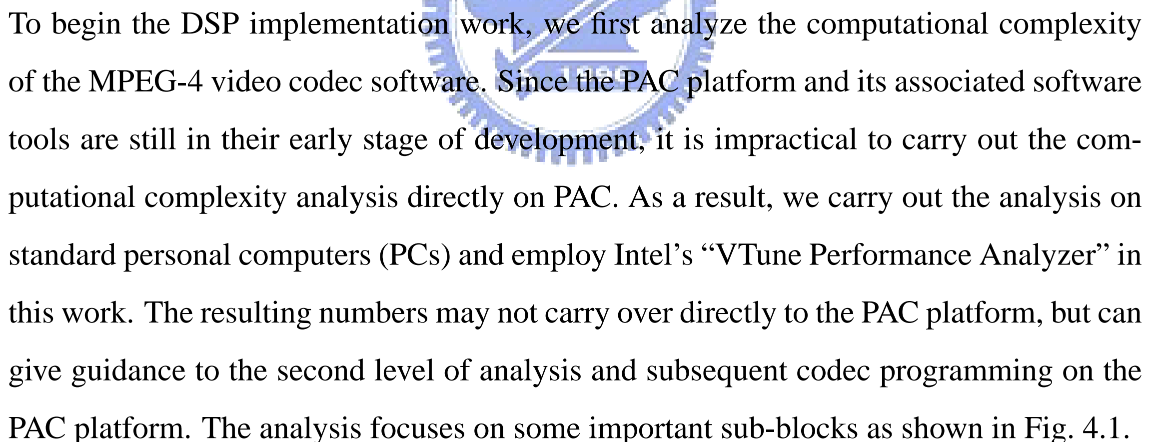
Kbytes. It is noted that the ISS can be used on linux operating systems only.

3.10.2 Implementation Considerations

Since the goal of our implementation is achieve a real-time MPEG-4 video decoder on PACDSP, the execution time is the most important issue that we care about. Although the compiler provides us facility for implementation, its performance is not better than well-scheduled hand code. Moreover, the development of compiler is not completed when we begin our implementation, so our implementation focuses on assembly code programming and its optimizations.

Chapter 4

Complexity Analysis and Implementation Strategy of MPEG-4 Framed-Based Video Decoder



To begin the DSP implementation work, we first analyze the computational complexity of the MPEG-4 video codec software. Since the PAC platform and its associated software tools are still in their early stage of development, it is impractical to carry out the computational complexity analysis directly on PAC. As a result, we carry out the analysis on standard personal computers (PCs) and employ Intel's "VTune Performance Analyzer" in this work. The resulting numbers may not carry over directly to the PAC platform, but can give guidance to the second level of analysis and subsequent codec programming on the PAC platform. The analysis focuses on some important sub-blocks as shown in Fig. 4.1.

After the complexity analysis, we discuss the implementation of bitstream accesses. In addition, different variable length decoding (VLD) methods on PACDSP and efficient interpolation technique are also discussed in this chapter. The execution time and corresponding code size of each sub-block are also listed in this chapter.

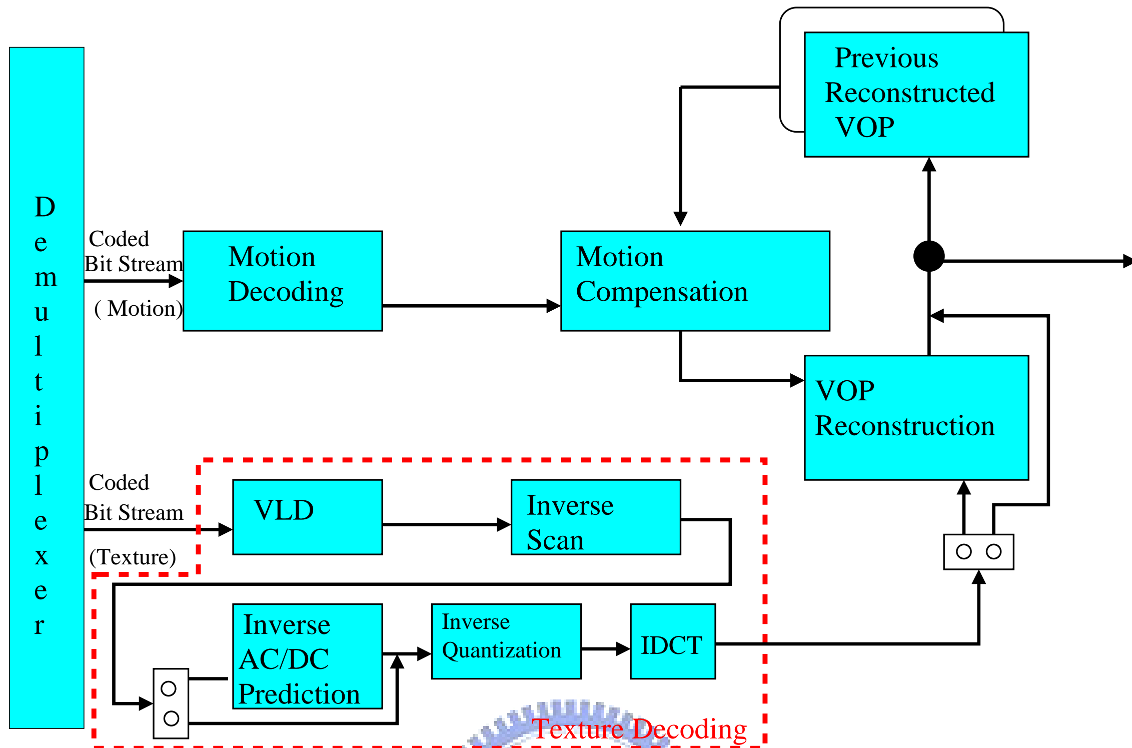


Figure 4.1: Block diagram of MPEG-4 frame-based video decoder [2].

4.1 Profiles of The MPEG-4 Frame-Based Video Decoder

4.1.1 Approach to Complexity Analysis

Our approach to codec complexity analysis consists of two levels, which may be viewed as employing a divide-and-conquer strategy.

The first level is an operational analysis of the time the codec software spends in coding of practical video sequences. Two major usages of this analysis are the identification the time-critical codec functions and the acquisition of some senses concerning the relative complexity of different codec functions in actual decoder operation. As a result, the complexities of various decoder components, such as the motion compensator and the VLD, are statistically variable and not a set of fixed numbers.

To capture the complexity variation over different video material, we consider several common test video sequences of different amount of motion that likely represent the type of material the PAC platform will largely address in its video coding applications for some

years. They are the QCIF (176×144) "grandmother" sequence and "stefan" sequence.

The second level of analysis is low-level computational analysis of the time-critical codec functions. We calculate the amount of computation (additions, multiplications, memory accesses, etc.). This prepares us for implementing these functions on the PAC platform. One way to carry out such analysis is to examine the block diagrams of the video codec and estimate the number of computations from the mathematical equations that define each block's function. But this way of analysis may overlook some overhead needed in a practical software implementation such as address computations. We thus also employ the MoMuSys software in this level of analysis [6], understanding that the results do not necessarily carry directly over to the PAC platform, but provide some reference data.

4.1.2 Profile on PC Using Intel VTune Performance Analyzer

The computational environment is a laptop with a 1.7 GHz Pentium-M CPU and 768 MB of DDR RAM, running Windows-XP. The profiling results, in Table 4.1, is obtained from encoding and decoding 2 frames employing H.263 quantization with a fixed quantization step size (QP), 4. It is noted 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.

In Table 4.1, it is noted that the data are calculated for two frames. However, some functions, such as "DecodeMBMVs" 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.

Nevertheless, we can still find in Table 4.1 that IDCT is a very important part in the decoding procedure, and the reason why IDCT consumes so much time is that the IDCT in the reference code is implemented in floating-point.

Moreover, the "bitstream access" includes accesses of decoding header and motion vectors, but most of the execution time is consumed in decoding of block coefficients.

Although the test sequences are all in QCIF format, the execution time of each sub-

Table 4.1: Profile of Frame-Based MPEG-4 Decoding of QCIF on PC

	stefan_qcif		grandmother_qcif	
Function Name	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

block varies with the characteristics. For example, the sub-block, "DecodeMBMVs," requires 1,544 clockticks for "stefan," but only 69 clockticks executed for "grandmother." The reason for such a result is that the amount of motion is less in the "grandmother" sequence, so the "skipped mode", which has no motion vectors, occurs more frequently.

4.1.3 Low-Level Computational Analysis

In the following analysis, 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 distin-

guishing "data" and "mem" operations is that many processors treat these two types of operation differently.

Motion Compensation

In the MPEG-4 simple-profile, there are four steps to complete the motion compensation for luminance blocks, and they are as follows:

1. The reference frame is padded with 16 pixels around the whole frame.
2. The padded frame is interpolated by two.
3. According to the corresponding motion vectors, we can find the reference block data.
4. Add the decoded residual data with the reference data.

We can easily find that there is much computation power consumed for the data accesses in memory. That is, the memory load/store and address calculations occupy most of the computational complexity for motion compensation. In Table 4.2, we estimate the necessary computations for completing the compensation of one frame.

According to Table 4.2, the storage requirement for luminance motion compensation is approximately as follows:

- 25,344 bytes for the previously decoded frame,
- 36,608 bytes for the padded frame,
- 146,432 bytes for the interpolated frame,
- 25,344 bytes for the reference frame data according to MVs,
- 25,344 bytes for the residual frame data.

The total memory space required is 253 KBytes, regardless of any memory-share skill. It is noted that we consider the forward predicted P-VOP only. The computational

Table 4.2: Complexity of Luminance Motion Compensation in One QCIF Frame

Operation	Padding	Interpolation	Find-Ref.	Add-Residual
data-add	0	181,508	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

1896 E S A

complexity and storage requirement of chrominance motion compensation are listed in Table 4.3. It is noted that the approximate complexity is analyzed under the 4:2:0 format.

Since the memory requirement and the amount of computation for addresses are very big, straightforward pointing of the C code for motion compensation to PACDSP will be inefficient with too many memory accesses. Therefore, we should further analyze the characteristics of motion compensation, and we leave the discussion to the next section.

Texture Decoding After VLD

The texture decoding steps after VLD involve inverse scan, inverse AC/DC prediction, inverse quantization (or dequantization), and IDCT. Among these, the inverse scan involves some memory manipulation. The inverse AC/DC prediction involves relatively small amount of computation (in the VTune analysis results). We concentrate on the dequantization and the IDCT below.

The MoMuSys code for dequantization and IDCT are relatively straightforward com-

Table 4.3: Complexity of Chrominance Motion Compensation in One QCIF Frame

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

pared to some other sections (such as motion compensation). Instead of carrying out a complexity analysis based on the algorithm as in the case of motion compensation, we analyze the MoMuSys code itself. The result, for one 8×8 block, is as shown in Table 4.4.

We now consider the storage requirement. The dequantization may need to store an 8×8 quantization matrix in addition to the data to be dequantized. The MoMuSys IDCT (and DCT) code is based on the conventional row-column computation algorithm. It only requires several words to store the DCT coefficients and several words to store the intermediate transform results, in addition to that required for the input and output data, where the input and output data can be colocated if desired.

4.2 Implementation Strategies on PACDSP

After the profiling on PC, we know that the bitstream decoding is a very important and time-consuming part in MPEG-4 video decoder. Especially, the memory accesses and

Table 4.4: Complexity of Dequantization and IDCT for One 8×8 Block in MoMuSys Code

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

load/store operations, are very critical issues in DSP implementation. Besides, motion compensation (MC) is also a most complex part in decoding of inter-encoded frames. Therefore, we do some analyses on both PC and PACDSP to find better methods for VLD and motion compensation. Discussions of dequantization and IDCT are left to the next chapter.

4.2.1 Efficient Variable Length Decoding (VLD)

A big issue concerning software implementation on a VLIW processor is that if there is any stall or program sequence branch, the entire processor has to stall or branch [7]. That is, we should try to synchronize the program sequence in both cluster to avoid inefficiency or incorrect programming. Otherwise, the computation in one cluster will be terminated by the change of program sequence caused by the other cluster. Besides, the register files are not shared between the two clusters, so we cannot access the bitstream in two clusters simultaneously. Therefore, we will compare the performance of different VLD methods

on PACDSP. The methods are proposed in [8] and [9]. We use the simple VLC table in Table 4.5 for the following comparison.

Bit by Bit Matching

If the size of variable length code table is not very big, we can simply check the bitstream bit by bit, and compare if any one symbol in the table is matched. The advantage of this method is its simplicity, but the number of memory accesses to acquire the bits and the number of comparison instructions are many. Therefore, the average execution time to decode a symbol will be long. The example assembly program of bit by bit matching on the PACDSP is shown in Fig. 4.2

One Table Mapping with Magnitude-Offset

We mentioned that the performance of a VLIW processor would degrade if there are many program sequence branches. In this technique, we build a table containing all possible code words. Each entry in the table has two elements, which are the corresponding VLC symbol and its code length. Thus, because the maximum code length is 11 bits in this example, there would be 2^{11} items in the new table, We fetch the first 11 bits in the bitstream. Then, the magnitude of the 11 bits gives the offset, which is used to fetch the corresponding item in the table. Note that we only have to access the bitstream once per symbol. The example assembly program of one-table mapping with magnitude-offset on the PACDSP is shown in Fig. 4.3

Multiple-Pass Matching

To reduce the frequency of accessing the bitstream, we may divide the VLC table into several subtables. The number of subtables is the number of bitstream accessing in worst case. Since the symbol with shorter code appears more frequently, we may appropriately enlarge the first subtable to further reduce bitstream accesses. For example, we divide the test table into three subtables. The first half with symbols 0–6 are grouped into the same table. Then, the second half is also divided into two parts with symbols 7–9 and 10–12, respectively. Therefore, we read the first five bits in the bitstream, and check if any code

Table 4.5: Variable Length Codes for dct_dc_size_luminance [2]

Variable length code	dct_dc_size_luminance
011	0
11	1
10	2
010	3
001	4
0001	5
0000 1	6
0000 01	7
0000 001	8
0000 0001	9
0000 0000 1	10
0000 0000 01	11
0000 0000 001	12

in the first subtable match the bits. If there is not a match, we read the next three bits and check the second subtable, and continue until the symbol is decoded. The example assembly program of multiple-pass matching on the PACDSP is shown in Fig. 4.4

Bounded Multiple-Pass Lookup with Magnitude-offset

The methods proposed previously have different advantages and disadvantages in speed and memory usage. We may take a compromise between execution time and memory requirement. Similar to the "multiple-pass matching" method, we still divide the test table into several subtables. However, the purpose of "multiple-pass matching" is to reduce the number of bitstream accessing, but the division in this technique is to reduce the huge table size in the "one table mapping with magnitude-offset" method. The required table

Bit-by-Bit Matching

```
{ NOP |MOVI.L D2,2 |NOP |NOP |NOP }
; get dc_size
{ NOP |MOVI.H D2,0 |NOP |NOP |NOP }
; no need to store dc_size
{ J Show_Bitstream,R1 |NOP |NOP |NOP |NOP }
{ NOP |NOP |NOP |NOP |NOP }
{ NOP |NOP |NOP |NOP |NOP }
{ NOP |NOP |NOP |NOP |NOP }
{ NOP |SEQ D7,C2,p4,p5 |NOP |NOP |NOP }
{ NOP |NOP |NOP |NOP |NOP }
{ NOP |NOP |NOP |NOP |NOP }
{ NOP |NOP |NOP |NOP |NOP }; symbol 2
{ (p4)B End |NOP |NOP |NOP |NOP }
{ NOP |(p4)MOVI.L D5,2 |NOP |NOP |NOP }
{ NOP |(p4)MOVI.H D5,0 |NOP |NOP |NOP }
{ NOP |NOP |NOP |NOP |NOP }
{ NOP |NOP |NOP |NOP |NOP }
{ NOP |SEQ D7,C3,p4,p5 |NOP |NOP |NOP }
{ NOP |NOP |NOP |NOP |NOP }
{ NOP |NOP |NOP |NOP |NOP }
{ NOP |NOP |NOP |NOP |NOP }
{ (p4)B End |NOP |NOP |NOP |NOP }
{ NOP |(p4)MOVI.L D5,1 |NOP |NOP |NOP }
{ NOP |(p4)MOVI.H D5,0 |NOP |NOP |NOP }
{ NOP |NOP |NOP |NOP |NOP }; symbo 1
{ NOP |MOVI.L D2,3 |NOP |NOP |NOP }
{ NOP |MOVI.H D2,0 |NOP |NOP |NOP }
{ J Show_Bitstream,R1 |NOP |NOP |NOP |NOP }
{ NOP |NOP |NOP |NOP |NOP }
{ NOP |NOP |NOP |NOP |NOP }
{ NOP |NOP |NOP |NOP |NOP }
{ NOP |SEQ D7,C1,p4,p5 |NOP |NOP |NOP }
{ NOP |NOP |NOP |NOP |NOP }
{ NOP |NOP |NOP |NOP |NOP }
{ NOP |NOP |NOP |NOP |NOP }
{ (p4)B End |NOP |NOP |NOP |NOP }
{ NOP |(p4)MOVI.L D5,4 |NOP |NOP |NOP }
{ NOP |(p4)MOVI.H D5,0 |NOP |NOP |NOP };
{ NOP |NOP |NOP |NOP |NOP }; symbol4
{ NOP |NOP |NOP |NOP |NOP }
{ NOP |SEQ D7,C2,p4,p5 |NOP |NOP |NOP }
{ NOP |NOP |NOP |NOP |NOP }
{ NOP |NOP |NOP |NOP |NOP }
{ NOP |NOP |NOP |NOP |NOP }
{ (p4)B End |NOP |NOP |NOP |NOP }
{ NOP |(p4)MOVI.L D5,3 |NOP |NOP |NOP }
{ NOP |(p4)MOVI.H D5,0 |NOP |NOP |NOP }
{ NOP |NOP |NOP |NOP |NOP };symbol3
{ NOP |NOP |NOP |NOP |NOP }
{ NOP |SEQ D7,C3,p4,p5 |NOP |NOP |NOP }
; symbol0
```

Figure 4.2: Example of bit by bit matching on PACDSP.

size in this technique is shown in Eq. 4.1. We only need to access the bitstream once per symbol, and the magnitude of code word is used to check which table to be searched in.

$$\begin{cases} Table\ \ size = \sum_{i=0}^{N} 2^{c_i} \\ \sum_{i=0}^{N} c_i = L \qquad \text{where } L \text{ is the max code word length.} \end{cases} \tag{4.1}$$

For example, the VLC table of the example is still partitioned into 3 parts. In this method, the number of entries in each search table are 32, 8, and 8, respectively. The first five bits are used to be the offset in the first table, and the following and last three bits are used in the second and third table, respectively. The search in each subtable is the same as “one table mapping with magnitude-offset” method. In conclusion, the total table size is 48 bytes, which is much smaller than that in the one table mapping method. The example assembly program of bounded multiple-pass lookup with magnitude-offset on the PACDSP is shown in Fig. 4.5

Comparison of Different VLD Methods

Using the methods introduced above, we decode a bitstream consists of all possible symbols on PACDSP. The results are shown in Fig. 4.6 and Table 4.6. We find that the best

One Table Mapping with Magnitude-Offset

```
{ NOP | MOVI.L D2,11 | NOP | NOP | NOP }; get dc_size
{ NOP | MOVI.H D2,0 | NOP | NOP | NOP }; no need to store dc_size
{ J Show_Bitstream,R1 | NOP | NOP | NOP | NOP }
{ NOP | NOP | NOP | NOP | NOP } ; get code word
{ NOP | NOP | NOP | NOP | NOP }; once access per symbol
{ NOP | NOP | NOP | NOP | NOP }
{ NOP | MOVI.L A2,DC_Table | NOP | NOP | NOP }
{ NOP | MOVI.H A2,DC_Table | NOP | NOP | NOP }
{ NOP | MOVI.L A3,DC_Size | NOP | NOP | NOP }
{ NOP | MOVI.H A3,DC_Size | NOP | NOP | NOP }
{ NOP | ADD A2,A2,D7 | NOP | NOP | NOP }
{ NOP | LBU D5,A2,0 | NOP | NOP | NOP }
{ NOP | NOP | NOP | NOP | NOP }
{ NOP | NOP | NOP | NOP | NOP }
{ NOP | SW D5,A3,0 | NOP | NOP | NOP }
```

Figure 4.3: Example of one table mapping with magnitude-offset on PACDSP.

performance occurs when the "bit-by-bit matching" method is applied with the shortest code word. However, there is significant degradation when the corresponding code pattern becomes longer. Therefore, because of the characteristic of entropy coding which uses shorter codes to represent more frquently appearing symbols, the "bit-by-bit matching" method can be used when most symbols may be encoded with shorter code words.

The performance of "multiple-pass matching" method has a similar characteristic, which is also affected by the length of code pattern. Compared to the first method, we need to fetch the bitstream only three times in the worst case, so we need 367 rather than 732 cycles for the longest code word.

In the third method, "one table mapping with magnitude-offset", we only access the bitstream once, so the execution time of decoding a symbol is the same for all cases in this example. Nevertheless, the primary drawback of this method is the memory requirement of the lookup table because of the exponentially increasing table size. Thus, this method is appropriate only when the lookup table is not large.

Finally, since the methods discussed above have different drawbacks, the fourth method provides a tradeoff between table size and execution time. We see that the execution time is very close to that of the third method, and the table size is 48 items rather than 2^{11}. In conclusion, the "bounded multiple pass lookups with magnitude-offset", is very effi-

Multiple-Pass Matching

```
{ NOP |MOVI.L D2,5 |NOP |NOP |NOP }
; get dc_size
{ NOP |MOVI.H D2,0 |NOP |NOP |NOP }
; no need to store dc_size
{ J Show_Bitstream,R1 |NOP |NOP |NOP |NOP }
{ NOP |NOP |NOP |NOP |NOP }
{ NOP |NOP |NOP |NOP |NOP }
{ NOP |NOP |NOP |NOP |NOP }
{ NOP |SEQ D7,C0,p4,p5 |NOP |NOP |NOP }
{ NOP |MOVI.L A3,DC_Size |NOP |NOP |NOP }
{ NOP |MOVI.H A3,DC_Size |NOP |NOP |NOP }
{ NOP |NOP |NOP |NOP |NOP }
{ (p4)B Check_Tab_1 |NOP |NOP |NOP |NOP }
{ NOP |NOP |NOP |NOP |NOP }
{ NOP |NOP |NOP |NOP |NOP }
{ NOP |NOP |NOP |NOP |NOP }
{ NOP |MOVI.L D5,0x3 |NOP |NOP |NOP }
{ NOP |SRLI D6,D7,3 |NOP |NOP |NOP }
{ NOP |SEQ D6,D5,p10,p11 |NOP |NOP |NOP }
{ NOP |NOP |NOP |NOP |NOP }
{ NOP |NOP |NOP |NOP |NOP }
{ (p10)B Flush_Update,R1 |NOP |NOP |NOP |NOP }
{ NOP |(p10)MOVI.L D2,2 |NOP |NOP |NOP }
{ NOP | (p10)MOVI.H D2,0 |NOP |NOP |NOP }
{ NOP |(p10)SW D5,A3,0 |NOP |NOP |NOP }
{ (p10)B End |NOP |NOP |NOP |NOP }
{ NOP |NOP |NOP |NOP |NOP };sybol1
{ NOP |NOP |NOP |NOP |NOP }
{ NOP |NOP |NOP |NOP |NOP }
{ NOP |MOVI.L D5,0x2 |NOP |NOP |NOP }
{ NOP |SRLI D6,D7,3 |NOP |NOP |NOP }
{ NOP |SEQ D6,D5,p10,p11 |NOP |NOP |NOP }
{ NOP |NOP |NOP |NOP |NOP }
{ NOP |NOP |NOP |NOP |NOP }
{ (p10)B Flush_Update,R1 |NOP |NOP |NOP |NOP }
{ NOP |(p10)MOVI.L D2,3 |NOP |NOP |NOP }
{ NOP | (p10)MOVI.H D2,0 |NOP |NOP |NOP }
{ NOP |(p10)SW D5,A3,0 |NOP |NOP |NOP }
{ (p10)B End |NOP |NOP |NOP |NOP }
{ NOP |NOP |NOP |NOP |NOP }
{ NOP |NOP |NOP |NOP |NOP }
{ NOP |NOP |NOP |NOP |NOP }
{ NOP |MOVI.L D5,0x2 |NOP |NOP |NOP }
{ NOP |MOVI.H D5,0 |NOP |NOP |NOP }
{ NOP |SRLI D6,D7,2 |NOP |NOP |NOP }
{ NOP |SEQ D6,D5,p10,p11 |NOP |NOP |NOP }
{ NOP |NOP |NOP |NOP |NOP }
{ NOP |NOP |NOP |NOP |NOP }
{ (p10)B Flush_Update,R1 |NOP |NOP |NOP |NOP }
{ NOP |(p10)MOVI.L D2,2 |NOP |NOP |NOP }
{ NOP | (p10)MOVI.H D2,0 |NOP |NOP |NOP }
{ NOP |(p10)SW D5,A3,0 |NOP |NOP |NOP }
{ (p10)B End |NOP |NOP |NOP |NOP }
{ NOP |NOP |NOP |NOP |NOP }; symbol0
{ NOP |NOP |NOP |NOP |NOP }
{ NOP |NOP |NOP |NOP |NOP }
```

Figure 4.4: Example of multiple-pass matching on PACDSP.

cient, but the simple "bit-by-bit matching" method is attractive if we can reduce the cycles required for bitstream accesses.

4.2.2 Efficient Motion Compensation

In the MoMuSys reference software, the motion compensation is done after all the residual data are decoded. Moreover, the reference frame is interpolated before the start of motion compensation. However, the internal memory of PACDSP is 64 KB only, so the interpolated frame is too large to be stored in the internal memory. Therefore, we propose a block-based interpolation and compensation method.

In compensation of the coefficient block by block, notice that the horizontal and the vertical block motion vectors may be both integers. That is, interpolation may not necessary. We analyze the motion vectors of luminance blocks using the MPEG-4 reference software. We count the amount of motion vectors, which are fractional in both horizontal and vertical directions, and the results are listed in Table 4.7. In Table 4.7, "Both"

Bounded Multiple Pass Lookup with Magnitude-Offset

```
{ NOP | MOVI.L D2,11 | NOP | NOP | NOP }; get dc_size
{ NOP | MOVI.H D2,0 | NOP | NOP | NOP }; no need to store dc_size
{ J Show_Bitstream,R1 | NOP | NOP | NOP | NOP }
{ NOP | NOP | NOP | NOP | NOP } ; get code word
{ NOP | NOP | NOP | NOP | NOP }; once access per symbol
{ NOP | NOP | NOP | NOP | NOP }
{ NOP | MOVI.L A2,DC_Table | NOP | NOP | NOP }
{ NOP | MOVI.H A2,DC_Table | NOP | NOP | NOP }
{ NOP | MOVI.L A3,DC_Size | NOP | NOP | NOP }
{ NOP | MOVI.H A3,DC_Size | NOP | NOP | NOP }
{ NOP | SGTI D7,63,p4,p5 | NOP | NOP | NOP }
{ NOP | SGTI D7,8,p6,p7 | NOP | NOP | NOP }
{ NOP | NOP | NOP | NOP | NOP }
{ (p4)B Check_Tab | NOP | NOP | NOP | NOP }
{ NOP | NOP | NOP | NOP | NOP }
{ NOP | NOP | NOP | NOP | NOP }
{ NOP | NOP | NOP | NOP | NOP }
{ (p6)B Check_Tab_1 | NOP | NOP | NOP | NOP }
{ NOP | NOP | NOP | NOP | NOP }
{ NOP | NOP | NOP | NOP | NOP }
{ NOP | NOP | NOP | NOP | NOP }
Check_Tab_2:
```

Figure 4.5: Example of bounded multiple-pass lookup with magnitude-offset on PACDSP.

means that both the horizontal and the vertical motion vectors are fractional. “Hor.” and “Ver.” mean that the motion vector is fractional only in horizontal and vertical direction, respectively.

According to the analysis in Table 4.7, we can also understand more about the details of different sequences, such as the directions of motion. Moreover, we know that the more than 50% of interpolation can be avoided in four of the six test sequences. Thus, if we can check the characteristic of the motion vectors in both luminance and chrominance motion compensation, much computation can be saved.

4.2.3 Profile on PACDSP of All Decoder Functions

Since the simulator of PACDSP is not equipped with the function of profiling, we will estimate the worst case execution cycles of each block and the corresponding code size,

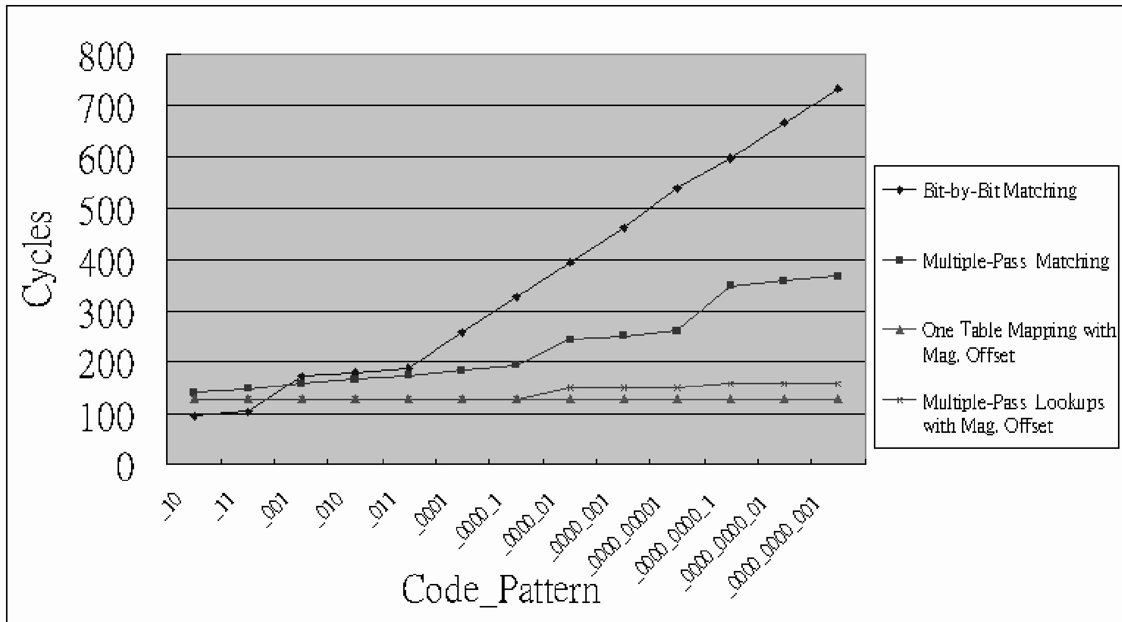


Figure 4.6: Comparison of different VLD methods on PACDSP

which will be evaluated by lines of assembly code. As mentioned in the previous chapter, PACDSP is a VLIW processor, and it can issue 5 instructions in parallel bounded in one instruction packet. That is, we can simply count the number of the instruction packets to estimate the execution time. The estimated profile is shown in Table 4.8. Note that we also list the category of each sub-function in Fig. 4.1.

The functionality of some important subblocks is as follows. "BitstreamShowBits" is used to access the bitstream stored in the corresponding buffer, and we can get the value of next n bit in the bitstream. "FlushUpdate" will flush the most significant m bit in the bitstream, where m and n should be given before entering these two functions, respectively. Therefore, the bitstream decoding, including headers and coefficients, can be completed with corresponding m and n. In addition, the headers of VOL, VOP, and MB are also decoded from bitstream with the aid of these two functions.

The purpose of "DecodeMBMVs" is to get the motion vectors of a macroblock. To get a correct motion vector, we need to get "mv_difference" with variable length decoding and "pmv", which is predicted from the previously decoded motion vectors. It is noted that both the difference and the prediction are composed of horizontal and vertical parts, and this function will be called only in inter-encoded frames.

Table 4.6: Execution Time of Different VLD Methods on PACDSP

Code Pattern	Bit-by-Bit Matching	Multiple-Pass Matching	One Table Mapping with Magnitude-Offset	Bounded Multiple-Pass Lookup with Magnitude-Offset
10	94	139	128	128
11	103	148	128	128
001	171	157	128	128
010	180	166	128	128
011	189	175	128	128
0001	258	184	128	128
0000 1	326	193	128	128
0000 01	394	243	128	149
0000 001	462	252	128	149
0000 0001	539	261	128	149
0000 0000 1	598	349	128	158
0000 0000 01	666	358	128	158
0000 0000 001	732	367	128	158

As to "DCACPrediction," it is involved only when the corresponding frame is encoded in "intra" mode. In this function, DC and AC predictions of an 8×8 block are completed with the rules in MPEG-4 standard [2]. In addition, there are three types of inverse zigzag scanning in intra encoded frames, and the inverse scanning is also completed in "DCACPrediction" in contrast to the function "Unzigzag" for inter encoded frames.

There are two types of quatization in the MPEG-4 standard. One is H.263 quantization and the other is MPEG-2 quantization. Our implementation is focused on simple profile, which supports H.263 quantization only.

The 8×8 two dimensional inverse discrete cosine transform (2-D IDCT) follows the

Table 4.7: Analysis of Necessary Interpolation Using MoMuSys

Bitstream	Total MV	Fractional MV							
(QCIF)	Number	Total	%	Both	%	Hor.	%	Ver.	%
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

definition in the MPEG-4 standard. In our implementation, this function includes clipping of the block coefficients, and the values after clipping range between 0 and 255.

"BlockInterpolation", "Luma_MC", and "Chroma_MC" are used for motion compensation for luminance and chrominance blocks. Finally, "Fill_VOP" is for filling the decoded block to the memory space, for output frames or reference frames.

1896

Table 4.8: Estimated Profile of Frame-Based MPEG-4 Decoding of QCIF on PACDSP

Function Name	Cycles	Code Size (KB)	Category
BitstreamShowBits	112	179	VLD
FlushUpdate	32	33	VLD
DecodeVOLHeader	3,704	1432	VLD
DecodeVOPHeader	1,745	149	VLD
DecodeMBHeader	677	199	VLD
DecodeMBMVs	697	538	Motion Decoding
DCACPrediction	2,985	1554	Inverse AC/DC Prediction
Unzigzag	749	36	Inverse Scan
BlockDequantH263	2,599	104	Inverse Quantization
BlockIDCT	1,749	189	IDCT
BlockInterpolation	2,361	96	Motion Compensation
MC_Luma	5,359	166	Motion Compensation
MC_Chroma	5,918	244	Motion Compensation
Fill_VOP	3,792	127	VOP Reconstruction

Chapter 5

Optimization of The Implementation on PACDSP

In this chapter, we discuss the optimization of our implementation of MPEG-4 frame-based video decoder on PACDSP. The optimization techniques are categorized into two types, algorithmic and architectural. We also discuss the performance of the optimization. Moreover, we compare the performance with some other reported implementation on other hardware platforms.

5.1 Algorithmic Optimization

Most of our optimizations on algorithm are on the elimination of dequantization and IDCT [8], [10]. We consider the intra- and inter-encoded frames, separately.

5.1.1 Algorithmic Optimization for Intra Frames

In intra frames decoding, there is a process, prediction of DC and AC coefficients, which is not applied to inter-encoded frames. However, since such predictions are time-consuming, if the frequency of these predictions can be reduced, much execution time can be saved.

In addition, an important property of DCT is that it concentrates signal energy in lower frequency coefficients. That is, if a block is filled with constant coefficients, there will be

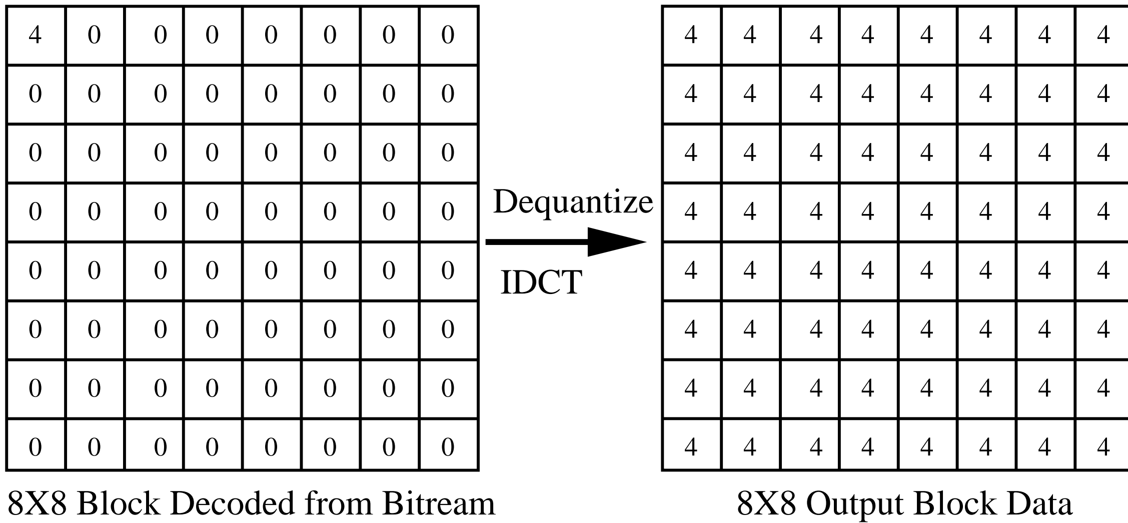


Figure 5.1: DC spreading from decoded coefficient to output block.

only one coefficient at the DC after the transform. In other words, if we can make sure that there is only a DC component 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 Fig. 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.

The assembly code of spreading DC value to the whole block is shown in Fig. 5.2. We need four iterations to complete one block, so the execution time is 19 cycles including the setting of loop register and address registers. However, we still need several cycles to

DC_Spread()

```
DC_Spreading: ; 4 iterations for one block
{ SET_LBCI RBC0,4 | MOVI.L A6,R_Block_2D | COPY D15,D14 | MOVI.L A6,R_Block_2D | COPY D15,D14 }
{ NOP | MOVI.H A6,R_Block_2D | NOP | MOVI.H A6,R_Block_2D | NOP }; D14 D15 are DC value
{ NOP | NOP | NOP | ADDI A6,A6,128 | NOP } ; 2nd half in 2nd cluster
Spread_DC_Coeff: ; iterations
{ LBCB RBC0,Spread_DC_Coeff | DSW D14,D15,(A6)+8 | NOP | DSW D14,D15,(A6)+8 | NOP }
{ NOP | DSW D14,D15,(A6)+8 | NOP | DSW D14,D15,(A6)+8 | NOP }
{ NOP | DSW D14,D15,(A6)+8 | NOP | DSW D14,D15,(A6)+8 | NOP }
{ NOP | DSW D14,D15,(A6)+8 | NOP | DSW D14,D15,(A6)+8 | NOP }; store 16 coefficient in one iteration
```

Figure 5.2: Assembly code of DC spreading.

update the prediction data “DC_Store”.

Check Skipped Blocks Using CBP and ACPred_Flag

In MPEG-4 video, there are two parameters encoded in the macroblock header which can help us reduce the amount of computation. The first one, CBP, standing for Coded Block Pattern, tells us which blocks in a macroblock are variable length encoded. The second, ACPred_Flag, informs us about the existence of AC coefficients prediction.

In order to find out the proportion of blocks that can be skipped, we choose the same test sequences as mentioned before. The simulation is done on PC with 90 frames to be encoded, and these frames are all encoded in intra type. The simulation results on PC are listed in Table 5.1.

In Table 5.1, we can see that the percentage of skipped block is not very high, and a slow-motion sequence such as “Akiyo” does not have the most skipped blocks among the six test sequences. The reason that the simulation results is not as what we expected is due to the parameter ACPred_Flag. Since the ACPred_Flag is set to 1 if there is any block in an MB predicted with AC coefficients, 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 previously simple checks cannot precisely indicate the blocks to be skipped, we add a check after the prediction of AC coefficients is completed. Similar to the previous method, we still need to check if the block data is variable length encoded through CBP in the MB header, CBP. If the corresponding bit in CBP is zero, we can skip this block because all the AC predicted coefficients are zero.

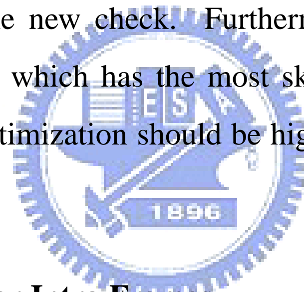
Consequently, we can further find out all the possible blocks to be skipped, but the effort also increases because of more conditions to be checked. We again do a simulation on PC to get the percentage of skipped blocks in 90 intra-encoded frames. The simulation results are listed in Table 5.2.

Compared to Table 5.1, we can see in Table 5.2 that the percentage of skipped blocks

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

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

gets higher with the aid of the new check. Furthermore, the test sequence "Grandmother_qcif" becomes the one which has the most skipped blocks, and it is expected that the performance of this optimization should be highly related the simulation results listed in Table 5.1 and 5.2.



Conclusion of Optimization for Intra Frames

Based on the analysis of the frequency of skipped blocks in intra-encoded frames, we apply the proposed means to our implementation on PACDSP. The simulation results are listed in Table 5.3, where noted that the execution time is gathered from the first encoded frame, not the average over 90 frames.

In Table 5.3, we can see that the performance of optimization varies from one sequence to another. The percentage of speedup on PACDSP is less than the percentage of skipped blocks, and this phenomenon can be explained by Ahmdahl's Law [7]. In other words, the skipped blocks do not reduce computations other than dequantization and IDCT, and we also need more cycles for the condition checking.

In conclusion, the above algorithmic optimization for intra-frame decoding is severely limited by the nature of the test sequences. To further improve the performance, we will take the advantage of VLIW architecture and SIMD instructions. The architectural

```
Vertical_AC_Reconstruction()
 vertical, top ROW of block C
 7 elements, so unroll the loop to 2 clusters
 A2 is Q_block, A3 is P_Coeff (AC)

; cluster1 1,   cluster2 5
{ NOP  | ADDI A2,A2,4  | CLR D12  | ADDI A2,A2,20  | CLR
D12  }
; D13 is index of Pred_A
{ NOP  | ADDI A3,A3,4  | NOP  | ADDI A3,A3,20  | NOP  }
{ NOP  | LW D14,A2,0  | NOP  | LW D14,A2,0  | NOP  }
{ NOP  | LW D15,(A3)+4  | NOP  | LW D15,(A3)+4  | NOP  }
; post increment
{ NOP |NOP  | NOP  | NOP  | NOP  }
{ NOP |NOP  | NOP  | NOP  | NOP  }
{ NOP |ADD D9,D14,D15|NOP | ADD D9,D14,D15 | NOP  }
{ NOP |SEQ D9,C0,p6,p7 |NOP | SEQ D9,C0,p8,p9  | NOP  }
{ NOP |SW D9,(A2)+4  NOP  | SW D9,(A2)+4  | NOP  }
; post increment
; cluster1 2,   cluster2 6
{ NOP  | LW D14,A2,0  | NOP  | LW D14,A2,0  | NOP  }
{ NOP  | LW D15,(A3)+4  | NOP  | LW D15,(A3)+4  | NOP  }
; post increment
{ NOP  | (p7)ADDI D12,D12,1  | NOP  | (p9)ADDI D12,D12,1
| NOP  }
{ NOP  | NOP  | NOP  | NOP  | NOP  }
{ NOP |ADD D9,D14,D15| NOP  | ADD D9,D14,D15 | NOP  }
{ NOP |SEQ D9,C0,p6,p7 | NOP  | SEQ D9,C0,p8,p9 | NOP  }
{ NOP |SW D9,(A2)+4  | NOP  | SW D9,(A2)+4  | NOP  }
; post increment
; cluster1 3,   cluster2 7
{ NOP  | LW D14,A2,0  | NOP  | LW D14,A2,0  | NOP  }
{ NOP  | LW D15,(A3)+4  | NOP  | LW D15,A3,0  | NOP  }
{ NOP  | (p7)ADDI D12,D12,1  | NOP  | (p9)ADDI D12,D12,1
| NOP  }
{ NOP  | NOP  | NOP  | NOP  | NOP  }
{ NOP | ADD D9,D14,D15|NOP  ADD D9,D14,D15 | NOP }
{ NOP | SEQ D9,C0,p6,p7 |NOP |SEQ D9,C0,p8,p9 | NOP }
{ NOP | SW D9,(A2)+4  | NOP  | SW D9,A2,0  | NOP  }
; cluster1 4,   cluster2 no
{ NOP | LW D14,A2,0  | NOP  | NOP  | NOP  }
{ NOP | LW D15,A3,0  | NOP  | NOP  | NOP  }
{ NOP | MOVI.L A7,Fake_AC_Pred | (p7)ADDI D12,D12,1  |
MOVI.L A7,Fake_AC_Pred  | (p9)ADDI D12,D12,1}
{ NOP  | MOVI.H A7,Fake_AC_Pred  | NOP  | MOVI.H
A7,Fake_AC_Pred  | NOP  }
{ NOP  | ADD D9,D14,D15 | NOP  | NOP  | NOP  }
{ NOP  | SEQ D9,C0,p6,p7  | NOP  | NOP  | NOP  }
{ NOP  | SW D9,A2,0  | NOP  | NOP | NOP  }
{ NOP  | NOP  | NOP  | NOP  | NOP  }
{ NOP  | (p7)ADDI D12,D12,1  | NOP  | NOP  | NOP  }
{ NOP | SEQ D12,C0,p10,p11 | NOP | SEQ D12,C0,p12,p13 |
NOP }
{ J End_of_One_Block | SW C0,A7,0 | NOP | NOP | NOP }
{ NOP  | NOP  | NOP  | NOP  | NOP  }
{ NOP  | (p10)SW C1,A7,0  | NOP  | NOP  | NOP  }
; iAC coeff. not zero
{ NOP  | NOP  | NOP  | (p12)SW C1,A7,0  | NOP  }
; Fake_AC_Pred  0
```

Figure 5.3: Assembly code of new check in vertical AC reconstruction.

optimization methods will be introduced and applied in the next section.

5.1.2 Algorithmic Optimization for P-Frames

Similar to the optimization for intra-encoded frames, we want to reduce the frequency that dequantization and IDCT are called. Therefore, we again do some analysis on PC to check how many blocks can be skipped, and we will apply some condition checking to skip the null residual blocks.

Table 5.2: Number of Skipped Blocks in 90 Intra Frames with Further Check After AC Prediction

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
akiyo	53,460	11,863	22.19
mobile	53,460	2,864	5.36
football	53,460	8,199	15.34

Table 5.3: Execution Time of Intra Frame Decoding on PACDSP

Test Seqs.	Execution Time (cycles)				
(QCIF)	Original	CBP and AC_Pred_flag Checked	speedup(%)	AC Prediction also Checked	Speedup(%)
grandmother	6,387,046	6,190,427	3.08	5,743,012	7.01
stefan	8,386,942	8,339,047	0.56	8,161,874	2.12
foreman	6,451,775	6,092,569	5.57	5,980,268	1.84
akiyo	6,183,448	5,885,550	4.82	5,695,724	3.23
mobile	10,211,299	10,189,775	0.21	10,128,500	0.60
football	7,087,360	6,973,907	1.60	6,920,920	0.76

Analysis of Null Residual Blocks

In this optimization, the condition to be checked is whether the residual blocks are variable length encoded or not. Therefore, we just check the CBP in the MB header to see if the dequantization and IDCT can be skipped. 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 test sequences that are more "static" skip more blocks.

Table 5.4: Number of Skipped Blocks in 89 P Frames

Test Seqs. (QCIF)	Total Block No.	Skipped Block No.	%
grandmother	52,866	40,475	76.56
stefan	52,866	14,082	26.64
foreman	52,866	23,261	44.00
akiyo	52,866	43,943	83.12
mobile	52,866	5,734	10.85
football	52,866	15,038	28.45

Conclusion of Optimization for Inter (P) Frames

As a result of many null residual blocks in P frames decoding, we apply the skip condition to our implementation on PACDSP. Similar to the simulation for intra frames, we gather the execution time for one frame, or the simulation time will be very long and inefficient. The execution time of decoding an inter-encoded(P) frame of six different sequences are list in Table 5.5.

We can see in Table 5.5 that a large amount of execution time is saved for more "static" sequences, such as "Akiyo". Nevertheless, the performance of this optimization is still limited by Amdahl's Law [7].

To further reduce the execution time in P frame decoding, we will avail ourselves of the assistance of VLIW architecture and SIMD instructions, which is discussed in the next section.

5.2 Architectural Optimization

An important issue of DSP implementation is the utilization of the architectural advantages. In this section, we introduce some general software optimization techniques, including static rescheduling, loop unrolling, and software pipelining.

In addition, the computations are dispatched to different units to utilize the advantage

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

Test Seqs.	Execution Time(cycles)		
(QCIF)	Original	CBP Checked	speedup(%)
grandmother	5,607,644	3,895,737	30.53
stefan	7,464,140	6,883,885	7.77
foreman	6,494,590	5,294,624	18.48
akiyo	4,693,963	3,159,598	32.69
mobile	8,861,251	8,527,807	3.76
football	8,470,472	7,942,794	6.23

of VLIW processor. Some special SIMD instructions of PACDSP are used to compute or load/store multiple data at the same time. The advantage of SIMD instructions is to increase the throughput of computations.

5.2.1 General Optimization Techniques

To get a higher performance, we should try to fill all the slots in an instruction packet. That is, how to achieve a full-pipeline implementation is very important to a better performance. In this subsection, we introduce three optimization methods, namely, static rescheduling, loop unrolling, and software pipelining. The purpose of these techniques is to reduce the stalls resulting from hazards, and the appropriateness for PADCDSP of these techniques are discussed as well.

In the discussion, we use an example of summing the coefficients in a 1-D array, which contains eight 8-bit data. The corresponding C program is shown in Fig. 5.4. In order to simplify the utilization of different techniques, we use only one instruction slot in the instruction packet.

```
for ( i=0; i<8; i++)
    y += x[i];
```

Figure 5.4: Example of vector addition.

Static Rescheduling

In the assembly code programming, dependence of data may cause stalls in processor, which increase the required computation time. There are three types of data hazard, namely, read-after-write (RAW), write-after-read (WAR), and write-after-write (WAW).

In the left half of Fig. 5.5, we simply translate the C program in Fig. 5.4 to the PACDSP assembly code. We can see that two stalls after the “LB” instruction are resulted from the dependence of the register D0, because data loading from memory requires two cycles to be valid in PACDSP.

In addition, the conditional branch, whose predicate register is p2, depends on the comparison instruction SLTI. Therefore, there are seven stalls (NOPs) in the direct translation with three delay slots, and these stalls significantly degrade the execution speed.

We can utilize the independence of instructions to eliminate the stalls as much as possible. In the right half of Fig. 5.5, we change the order of computation, and it is obvious that the stalls are reduced from seven to four. However, since the computation is not very complex, we cannot further reduce the number of stalls simply through rescheduling.

Loop Unrolling

Loop unrolling is a general technique to deal with the implementation of an iterative computation, especially, if there are any stalls in a single iteration.

To utilize the unrolling technique, we have to find what are the independent computations in the consecutive iterations. We can use different registers to store data from different iterations, and the instructions still need to be scheduled well to reduce the stalls. The number of unrolled loops depends on the stalls and independent computations in a single loop. Figure 5.6 shows the assembly code before and after loop unrolling.

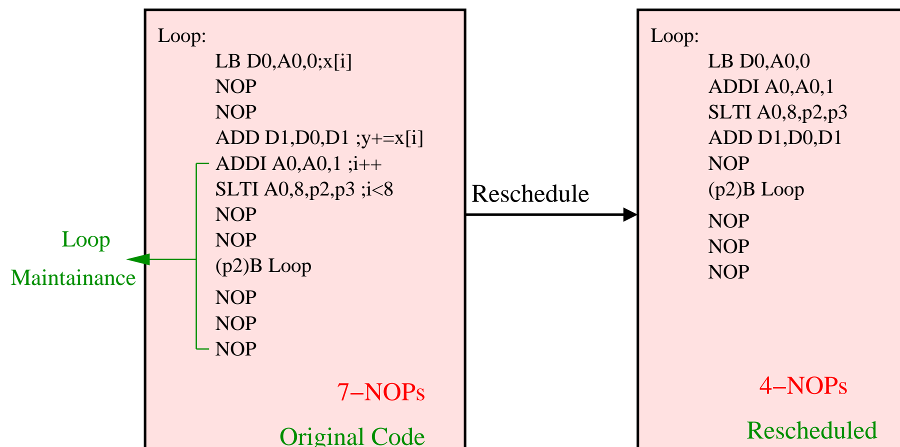


Figure 5.5: Example of static rescheduling technique.

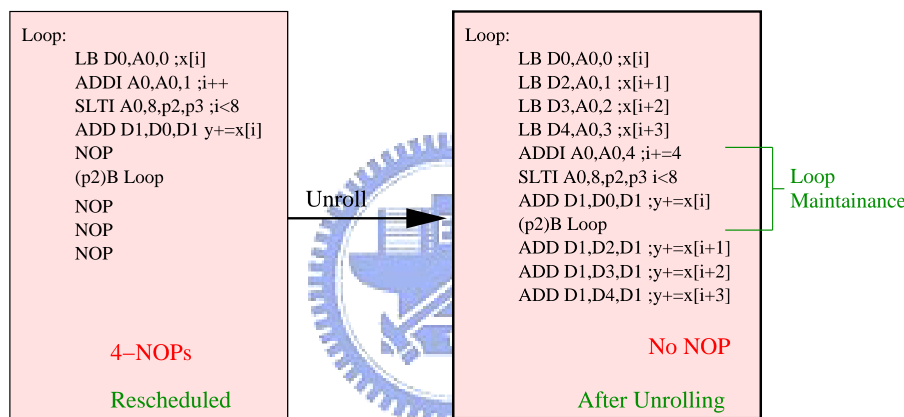


Figure 5.6: Example of loop unrolling technique.

In Fig. 5.6, we can find that all the stalls (NOPs) are eliminated. The loop maintenance code and branch condition should be changed to adjust the new iterative computations. However, there is a trade-off between execution time and corresponding code size. Although the stalls are all eliminated, the code size increases after loop unrolling. Therefore, we have to assess that if code size is critical or not. In addition, the number of available registers is a limitation to the utilization of loop unrolling.

Software Pipelining

The concept of software pipelining is to reorganize the loop and to interleave dependent instructions from different loop iterations to separate dependent instructions within the

original loop. Different from loop unrolling, we just reschedule the loop, so the stalls may not be entirely eliminated. An example of software pipelining is illustrated in Fig. 5.7.

It is noted that the start-up code and clean-up code are used to interleave the dependent code. Compared to loop unrolling, there are still 2 stalls. The advantage of software pipelining is the smaller code size. However, the loop overhead cannot be reduced through software pipelining. But we can apply loop unrolling and software pipelining to our implementation simultaneously and take the advantage of both techniques.

5.2.2 Advantages of PACDSP

In order to speed up our implementation on PACDSP, we can utilize the advantages of VLIW architecture and SIMD instructions. However, not all the computations can be distributed to both clusters, so we have to check if the feature of the computations are appropriate to apply the advantages of PACDSP.

In addition, since the branch instructions affects the program sequence of both clusters, it is better to put two regular and independent parts of computations in different clusters. For example, an iterative computation can be separated into two parts if the computations are independent in different iterations. Take the MPEG-4 frame-based video decoder for instance, dequantization (IQ) and IDCT (IT) are very regular computations, and we will discuss these two functions in the next section.

Moreover, SIMD instructions are also very helpful for optimization. Nevertheless,

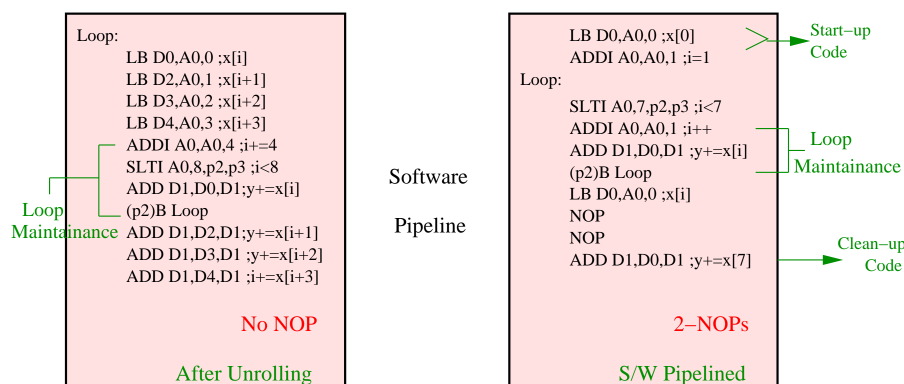


Figure 5.7: Example of software pipelining technique

most of the arithmetic SIMD instructions cannot be applied in our implementation because most of the data that we have length equal a word.

5.3 Experiment Results

In this section, we apply the architectural optimization techniques mentioned above. Since IQ and IT are very critical parts in the implementation of the video decoder, we particularly introduce the optimization of these two functions and the resulting improvement.

5.3.1 Optimization of Dequantization

In the MPEG-4 standard, there are two types of quantization, one is H.263 quantization and the other is MPEG quantization. Since our implementation focuses on the simple profile, we only need to support the H.263 quantization, and the its inverse quantization is defined as follows:

$$|F"[v][u]| = \begin{cases} 0, \text{if } QF[v][u]=0, \\ (2 \times |QF[v][u]| + 1) \times quantizer_scale, \text{if } QF[v][u] \neq 0,\ quantizer_scale \text{ is odd}, \\ (2 \times |QF[v][u]| + 1) \times quantizer_scale - 1, \text{if} QF[v][u] \neq 0,\ quantizer_scale \text{ is even}, \end{cases} \tag{5.1}$$

where $F"[0][0] = 8 \times QF[0][0]$ for intra frames and QF is the decoded block coefficients.

There are two main computations for the dequantization. First, we scale the coefficient according to the parameter "quantizer_scale," which is dependent on the value of QP. Secondly, we have to saturate the coefficients to the range, $(-2^{bits_per_piexel+3})$ – $(2^{bits_per_piexel+3} - 1)$.

The computations are very regular in the dequantization. As a result, we can separate the 8×8 block into two parts. For example, the first 32 coefficients are computed in the first cluster and the second half are done in the other cluster.

However, in the optimization of dequantization, we can skip some computations if the coefficient to be dequatized is zero. Thus, we need to check if the coefficients in both clusters are zero. We can deal with two consecutive coefficients simultaneously or the first 32 coefficients in the one cluster, and the second half in the other cluster.

We have to decide which strategy of distribution mentioned above is better for our optimization. Therefore, we gather the number of skipped coefficient pairs on PC with the MoMuSys reference software. We compress 90 frames in intra mode for each of the six test sequences, and the quatization step is four for all simulations. The results of the analysis are listed in Table 5.6.

In Table 5.6, we can easily find that it is better to work on two consecutive pixels in both clusters. Since the data structure in the implementation of dequatization is not appropriate for utilization of SIMD instructions and the limited number of registers also restrict the application of loop-unrolling and software piplelining, we apply the rescheduling technique to our implementation of dequantization. The original and optimized program are listed in Fig. 5.8

In the previous chapter, the simulation of 8×8 block requires 2599 cycles in worst case. The execution time is significantly reduced to 600–800 cycles after applying the above technique. Note that the required cycles depends on the number of consecutive zero coefficients. As a result, the original implementation of H.263 dequantization is replaced by the new design, and the simulation results of I- and P-frames are listed in Table 5.7.

5.3.2 Implementation of IDCT

The DCT and IDCT in MPEG-4 are defined as

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

$$f(x,y) = \frac{2}{N}\sum_{u=0}^{N-1}\sum_{v=0}^{N-1} C(u)C(v)F(u,v)\cos\frac{(2x+1)u\pi}{2N}\cos\frac{(2y+1)v\pi}{2N} \qquad (5.3)$$

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

and

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

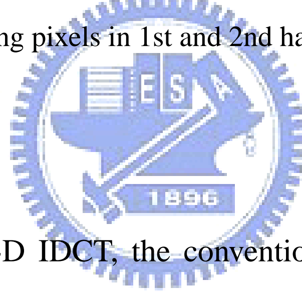
Many fast algorithms have been proposed for efficient computation. To implement IDCT on PACDSP, there are two critical issues, namely, efficiency and accuracy, which are discussed below.

Table 5.6: Analysis of Skipped Coefficients in Dequantization (90 I-frames)

Test Seqs.	Method 1 *		Method 2 **	
(QCIF)	No.	%	No.	%
grandmother	1,331,114	77.8	1,263,074	73.83
stefan	964,767	56.40	930,600	54.40
foreman	1,294,917	75.69	1,212,908	70.90
akiyo	1,394,676	81.53	1,296,934	75.81
mobile	668,798	39.09	609,066	35.60
football	1,303,186	76.18	1,249,742	73.05
Total pixel pairs: $176 \times 144 \times 1.5 \times 90 \div 2 = 1,710,720$				

* Two consecutive pixels as a pair.

**Corresponding pixels in 1st and 2nd half as a pair.



Efficiency of IDCT

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 [11]. Many fast algorithms for 2-D IDCT have been proposed, and one of them reduces the required 1-D IDCTs from 16 to 8 [11]. However, since the number of required registers is very big in this algorithm, it is not appropriate for the implementation on PACDSP.

We focuses the IDCT implementation on the efficiency of 1-D IDCT. Since the computational complexity of direct implementation is very high, there are also many fast algorithms for 1-D IDCT. Similar to the derivation from discrete Fourier transform (DFT) to fast Fourier transform (FFT), a fast cosine transform (FCT) is proposed in [12]. The comparison of computational complexity is listed in Table 5.8.

Note that the computational complexity is estimated for floating-point computation. Since the transform coefficients used in [12] are reciprocals of cosine values, the error increases because of limited accuracy in the fixed-point approximation on PACDSP. In addition, the number of multiplications is bigger in the even-odd decomposition algo-

Figure 5.8: Original and optimized assembly code of IQ.

rithm. As a result, we first implement the IDCT algorithm of MoMuSys on PACDSP.

Accuracy of IDCT

Since the PACDSP is not capable of floating-point computations, we have to convert the IDCT algorithm to an integer computation. There are also many approximation algorithms to floating-point IDCT. There are integer reversible algorithms for DCT/IDCT [14],[15], but they consist of several matrix computations, and the computational complexity should be much higher. Therefore, we do not implement a reversible transform.

Since there are two 16-bit multipliers in both clusters on PACDSP, we scale the floating-point cosine coefficients with 2^{15}. We then right shift 15 bits after the multiplication, and the multiplication is rounded to the nearest integer.

After applying this method to our implementation, and the execution time is about 1,200 cycles. We now check if the implementation is accurate enough.The IEEE Std.

Table 5.7: Improvement after Optimization of Dequatization

Test Seqs.	I-Frames (Cycles)			P-Frames (Cycles)		
(QCIF)	Original†	Optimized	%	Original†	Optimized	%
grandmother	5,743,012	5,160,154	10.15	3,895,737	3,613,810	7.24
stefan	8,161,874	7,386,839	9.50	6,883,885	6,257,262	9.10
foreman	5,980,268	5,327,327	10.92	5,294,624	4,859,341	8.22
akiyo	5,695,724	5,063,492	11.10	3,159,598	2,962,965	6.22
mobile	10,128,500	8,247,598	8.70	8,527,807	7,781,364	8.75
football	6,920,920	6,172,806	10.81	7,942,794	7,281,423	8.33

† Original means the execution time after algorithmic optimization.

Table 5.8: Comparison of Computational Complexity for 8-point IDCT

	Direct Form	FCT [12]	MoMuSys	Even_Odd FCT [13]
Multiplications	64	12	16	20
Additions	56	29	26	28

1180-1190, which is currently withdrawn, is usually used to test for the compliance of the implementation of IDCT algorithms. The compliance test requires five statistical measurements, which are as follows:

- For any pixel location, the peak error (ppe) shall not exceed 1 in magnitude.
- For any pixel location, the mean square error ($pmse$) shall not exceed 0.06.
- Overall, the mean square error ($omse$) shall not exceed 0.02.
- For any pixel location, the mean error (pme) shall not exceed 0.015 in magnitude.
- Overall, the mean error (ome) shall not exceed 0.0015 in magnitude.
- For all-zero input, the proposed IDCT shall generate all-zero output.

Table 5.9: Test of Compliance Using IEEE Std. 1180-1190

item	IEEE 1180–1190	MoMuSys	Even_Odd FCT [13]
ppe	$\leq$1	$>$1(X)	$\leq$1(◯)
$pmse$	$\leq$0.06	137.8279(X)	0.0081(◯)
$omse$	$\leq$0.02	5.2222(X)	0.0056(◯)
pme	$\leq$0.015	10.8429(X)	0.0019(◯)
ome	$\leq$0.0015	0.5742(X)	0.0001(◯)
$all\ zero\ input$	$all\ zero\ output$	◯	◯

We test the IDCT algorithm of MoMuSys with rounding to the nearest integer after each multiplication for the IEEE 1180-1190 compliance, whose results are listed in Table 5.9. We see that the simple rounding method introduces significant mismatch, so this algorithm does not comply with the IEEE 1180-1190 standard after converting to fixed-point computation, except in the last measurement.

Figure 5.9 shows signal flow in the algorithm. We see the odd-indexed coefficients are rounded twice. However, each rounding introduces corresponding error. Therefore, we try to use the even-odd decomposition algorithm [13]. In addition, since the multiplied coefficients are summed immediately, the number of roundings can be reduced if we postpone the roundings after the multiplied coefficients are summed. The signal flow graph is shown in Fig. 5.10. Note that the roundings are postponed to the output stage, so we have the right shift 19 bits to keep the correct format.

We also test the even-odd decomposition algorithm for the compliance of the IEEE 1180-1190 standard, whose results are also listed in Table 5.9. We can see that the even-odd decomposition algorithm complies with the standard, and the less rounding operations reduce the required execution time as well.

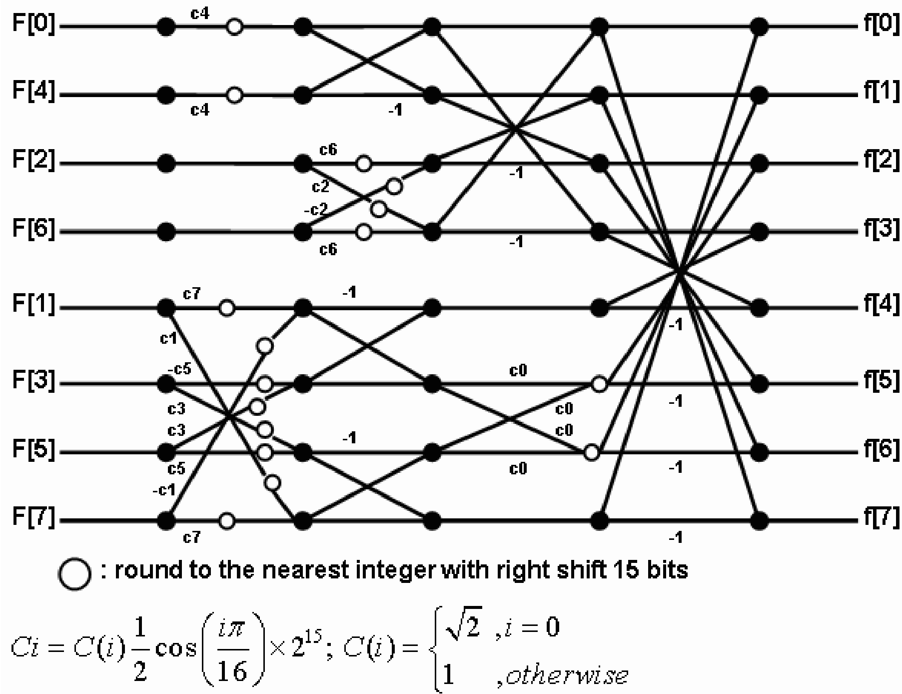


Figure 5.9: The IDCT algorithm used in MoMuSys.

Optimization of IDCT on PACDSP

There are two clusters in the PACDSP, and we can complete individual computations simultaneously because the computations of each row or column are independent. Therefore, we can simply distribute eight 1-D row-wise and column-wise IDCTs to both clusters. As a result, there are four iterations for both row and column computations.

According to the characteristics of the even-odd decomposition algorithm, we can use double-store, MAC, and butterfly instructions to facilitate the computation, where the butterfly instruction can sum and subtract the data in the two source registers at the same time.

The performance of various IDCT implementation are listed in Table 5.10. We see that the implementation on PACDSP is competitive, because of less arithmetic units required. The improvement to our implementation of the MPEG-4 video decoder is shown in Table 5.11.

In Table 5.11, we can find that the optimization of IDCT introduces at most 11.48 percent improvement. Moreover, since the number of skipped blocks is smaller for I-

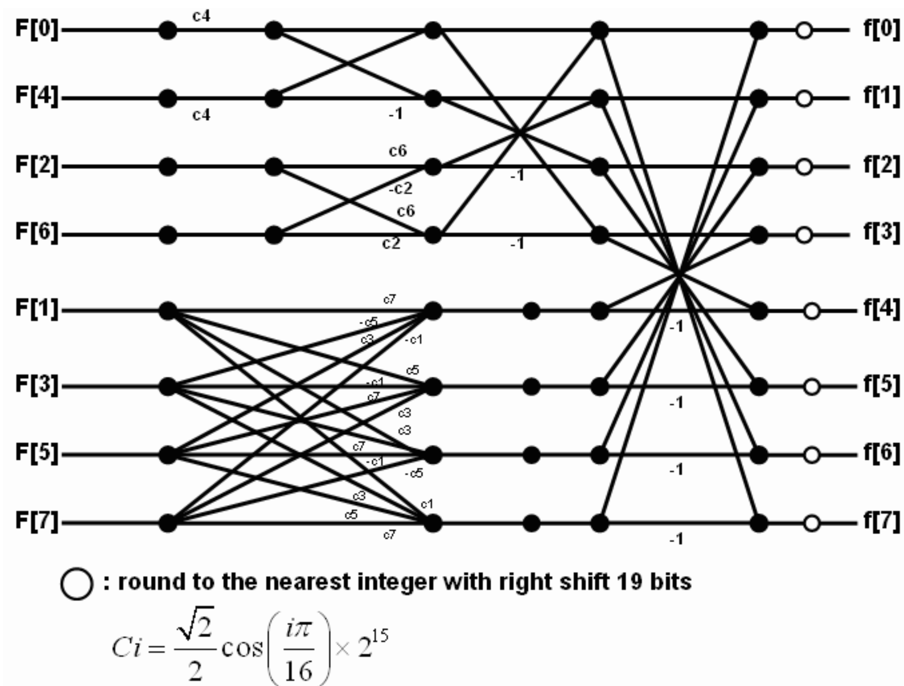


Figure 5.10: The even-odd decomposition IDCT algorithm[8].

frames, we have less improvement for P-frames. The C program , original ,and optimized assembly code of IDCT are listed in Appendix B.

5.3.3 Overall Optimization of the implementation

To further optimize the implementation of MPEG-4 video decoder on PACDSP, we review the entire program and apply the optimization methods introduced above.

First, we reschedule the program and try to eliminate all the unnecessary stalls, and the delay slots of the branches are filled as well. If there are any consecutive loads or stores, we replace the original program with double-loads or stores. Thus, the execution time and code size are both reduced. Second, we find out regular computations and employ the loop-unrolling and software-pipelining techniques to reduce the execution time.

The performance of the implementation after the overall optimization are listed in Table 5.12. We can find that all the execution time is about 5,600,000 cycles in worst case and 2,000,000 in best case. In other words, if the frequency of PACDSP is higher than 168MHz, then we can implement a real-time MPEG-4 video decoder for QCIF format

Table 5.10: Comparison of IDCT on Different Platforms

Designs	Processing units	Clock (MHz)	2-D fast algo.	Cycles
TI C30 [16]	1 MAC, 1 ALU	40	row-column	1344
TI C62x [16]	2 MUL, 6 ALU	200	row-column	226
TI C64x [17]	2 MUL, 6 ALU	600	row-column	154
IDCT Core [16]	1 ALU	33	direct 2-D	1208
PACDSP (ours)	2 AU, 2 L/S	200	even-odd	384

video on the PACDSP platform. Note that real-time means the decoding rate is higher than 30 frames per second. Since the instruction memory is a 32 KB cache, the program size of our implementation is 27 KB, which is smaller than the cache size. Therefore, the execution time is not degraded by the cache misses. In addition, the required data memory size is less than 64 KB depending on the size of bitstream to be decoded. That is, we cannot decode too many frames if the bitstream size is too big.

5.4 Conclusion on Optimization

In this chapter, we used several optimization techniques to improve the code. The simulation results before and after optimization are listed in Table 5.13.

We can see that about 50% of the execution time for both I- and P-frame decoding is reduced, except for the more complicated test sequences. The performance of our implementation will be compared with other implementations in the next section.

We show the speed-ups of different optimization steps for different test sequences in Figs. 5.11 and 5.12. The performance of algorithmic optimization is limited for I-frame decoding because the number of skipped blocks is not very big. However, the number of skipped blocks for P-frame decoding is very big especially for the test sequences that have less motion.

Moreover, the performance for IQ and IT optimizations are also not very significant.

Table 5.11: Improvement After Optimization of IDCT

Test Seqs.	I-Frames (Cycles)			P-Frames (Cycles)		
(QCIF)	Original††	Optimized	%	Original††	Optimized	%
grandmother	5,160,154	4,567,534	11.48	3,613,810	3,362,581	6.95
stefan	7,386,839	6,790,527	8.07	6,257,262	5,727,717	8.46
foreman	5,327,327	4,795,881	9.98	4,859,341	4,478,329	7.84
akiyo	5,063,492	4,541,150	10.32	2,962,965	2,768,631	6.56
mobile	8,247,598	8,606,904	6.93	7,781,364	7,163,932	7.93
football	6,172,806	5,536,382	10.31	7,281,423	6,725,999	7.63

†† Original means the execution time after optimization of dequantization.

Since we spend much execution time on decoding the bitstream, IQ and IT do not occupy much of the computation complexity. However, the performance of overall optimization is very impressive because many redundant stalls are removed after the optimization. Therefore, bitstream decoding as well as memory accesses are more efficient if we schedule the program better.

5.5 The Effect of Different Quantization Step (QP)

In the MPEG-4 video encoder, the quantization follows the IDCT computation. Therefore, the value of quantization step affects the floating-point block coefficients. In the previous implementation and discussion, we choose the QP value as 4 in all cases. To have a further understand of how QP affects the video encoding, we complete some analysis of different QP values.

We choose three different test sequences and three different QP values. The three sequences are akiyo, stefan, and mobile which are all in QCIF format, and they have low, medium, and high motion, respectively. Note that the three QP values are 3, 4, and 8. In the following, we discuss the effects to the I- and P-frame decoding.

Table 5.12: Overall Optimization after IDCT Optimization

Test Seqs.	I-Frames (Cycles)			P-Frames (Cycles)		
(QCIF)	Original†††	Optimized	%	Original†††	Optimized	%
grandmother	4,567,534	2,969,243	34.99	3,362,581	2,370,949	29.49
stefan	6,790,527	4,324,652	36.31	5,727,717	4,130,700	27.88
foreman	4,795,881	3,093,021	35.51	4,478,329	3,184,234	28.90
akiyo	4,541,150	2,918,622	35.73	2,768,631	2,050,519	25.94
mobile	8,606,904	5,606,101	34.87	7,163,932	5,035,200	29.71
football	5,536,382	3,429,212	38.06	6,725,999	4,743,437	29.48

††† Original means the execution time after optimization of IDCT.

5.5.1 Effects of QP to I-Frame Decoding

Since the larger QP value introduces a rougher quantization, more block coefficients may be quantized to the same value. As a result, the coefficients after DC/AC prediction may be simpler, and the number of skipped blocks in our algorithmic optimization may increase. The analyses of skipped blocks on PC using MoMuSys reference software are listed in Table 5.14.

In Table 5.14, we find that the number of skipped blocks increases with larger QP if we check CBP and ACPred_Flag only. However, if we further check the skip condition after AC prediction, the number of skipped block of "akiyo" does not increase with the increasing QP. Setting QP as 4, there are most skipped blocks in the "akiyo" sequence. Note that the number of skipped blocks are almost the same with setting QP as 8 even if we further check the skip condition after AC prediction.

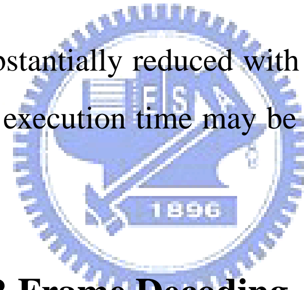
In addition, since different QP values affects the number of variable length coding, we also complete some simulation on our implementation. The execution time of I-frame decoding with different QP values are listed in Table 5.15. Note that we compare the execution time after all optimization methods applied.

In Table 5.15, we see significant affects of different QP values. The execution time of

Table 5.13: Execution Time Before and After Optimizations

Test Seqs. (QCIF)	I-Frames (Cycles)			P-Frames (Cycles)		
	Before	After	Speed-up (%)	Before	After	Speed-up (%)
grandmother	6,387,046	2,969,243	53.51	5,607,644	2,370,949	57.72
stefan	8,386,942	4,324,652	48.44	6,464,140	4,130,700	44.66
foreman	6,451,775	3,093,021	52.06	6,494,590	3,184,234	50.97
akiyo	6,183,448	2,918,622	52.80	4,693,963	2,050,519	56.32
mobile	10,211,299	5,606,101	45.10	8,861,251	5,035,200	43.18
football	7,087,360	3,429,212	51.62	8,470,472	4,743,437	44.00

the three test sequences are substantially reduced with increasing QP value. The reason for the significant reduction of execution time may be the reduction of the frequency of VLD.



5.5.2 Effects of QP to P-Frame Decoding

For the P-frame decoding, we still focus on two parts. One is the number of zero-residual blocks and the other is the percentage of fractional motion vectors.

First, we set different QP values and simulate on PC using MoMuSys reference software. The number of zero-residual blocks in 89 P-frames are listed in Table 5.16. We can find that the number of zero-residual blocks increases with the increasing QP values.

Second, we discuss the percentage of fractional motion vectors. Similar to the analysis in previous chapter, we also list the number of fractional motion vectors in Table 5.17, but we only show the total fractional motion vectors without checking the direction of motion vectors.

In Table 5.17, we see that the number of total motion vectors decrease with larger QP. That is, there are more MBs in skip mode. However, the number of fractional motion vectors increases. It may be resulted from the rougher quantization. The best motion

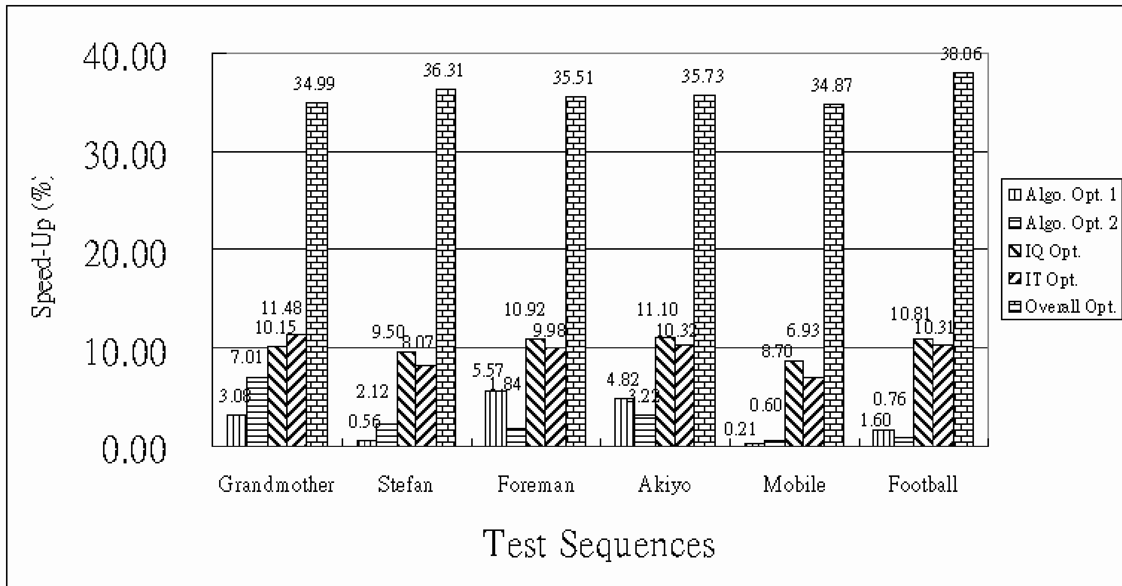


Figure 5.11: Speed-up of different optimization methods for I-frames.

vector of each block may be found in interpolated pixels with rougher quantization.

Finally, we also simulate P-frame decoding with different QP values in our implementation. the simulation results are listed in Table 5.18.

Although the number of zero-residual blocks increases with larger QP, there are more fractional motion vectors. Therefore, the execution time of P-frame decoding on PACDSP does not decrease with larger QP in the "akiyo" test sequence. However, the execution time is reduced with larger QP.

5.6 Comparison with Other Implementations

Since the MPEG-4 codec is widely used in the audio-visual compression, we can compare the performance of MPEG-4 video decoder on PACDSP with that reported for the other platforms. Table 5.19 lists some numerical information.

The requirement of MPEG-4 conformance test is 25-fps for QCIF [18]. The implementation on ARM7TDMI uses several architectural and algorithmic optimization techniques, and the frequency of memory access is significantly reduced. Thus, the performance of implementation on ARM7TDMI is much better than that on TI C6201.

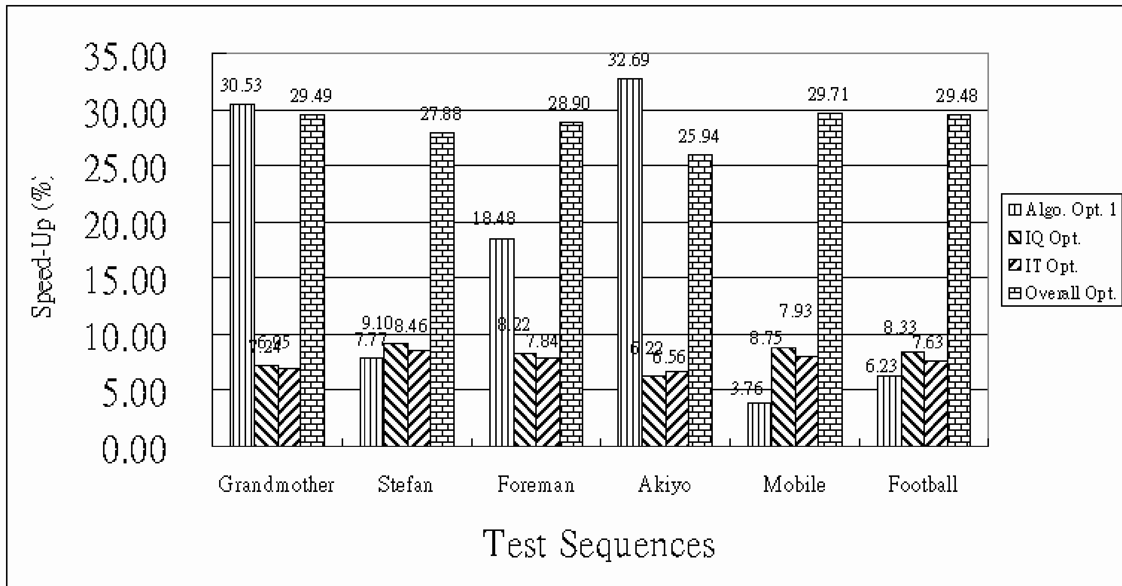


Figure 5.12: Speed-up of different optimization methods for P-frames.

The TriMedia CPU64 DSP is a powerful processor for multimedia applications, and it is a 5-issue VLIW processor with 27 function units. 64-bit and SIMD instructions are supported as well [21]. The 4CIF format is 704×576, which is 16 times larger than QCIF. However, the performance can also be a reference to compare other implementations.

The performance of our implementation is evaluated with 200 MHz frequency, and the decoding rate listed in Table 5.19 is estimated by the decoding of a P-frame for the test sequence akiyo. Since the cost of PACDSP is lower than other renowed processors, the performance of the implementation of MPEG-4 video decoder is competitive to other platforms.

Although our implementation can achieve the goal of real-time implementation, there are possible optimizations. For example, the data structure of our design is not appropriate for most SIMD instructions, and frequent memory accesses also consume much computations time. If we have a better plan for data structure and the usage of registers, much computation time can be saved.

Table 5.14: Number of Skipped Blocks in 90 Intra Frames with Different QP

Test Seqs. (QCIF)	QP	Check CBP and ACPred_Flag		Further Check after AC Prediction	
		Skipped Block No.	%	Skipped Block No.	%
akiyo	3	5,359	10.02	9,190	17.04
	4	6,574	12.30	11,863	22.19
	8	8,426	15.76	8,426	15.76
stefan	3	1,684	3.15	2,806	5.25
	4	2,041	3.82	4,679	8.79
	8	2,966	5.55	2,969	5.55
mobile	3	841	1.57	1,604	3.99
	4	1,422	2.66	2,864	5.36
	8	3,323	6.22	3,323	6.22
Total block number: 53,460					

Table 5.15: Effects of Different QP to Execution Time of I-Frame Decoding on PACDSP

Test Seqs. (QCIF)	Execution Time (Cycles Per Frame) QP = 3	QP = 4	QP = 8
akiyo	3,236,270	2,918,622	2,405,087
stefan	5,038,011	4,324,652	3,492,219
mobile	6,448,395	5,606,101	4,319,845

Table 5.16: Number of Skipped Blocks in 89 P-Frames with Different QP

Test Seqs. (QCIF)	QP	Check CBP and ACPred_Flag	
		Skipped Block No.	%
akiyo	3	43,345	81.99
	4	43,943	83.12
	8	49,113	92.90
stefan	3	11,842	22.40
	4	14,082	26.64
	8	26,550	50.22
mobile	3	4,385	8.29
	4	5,734	10.85
	8	13,849	26.20
Total block number: 52,866			

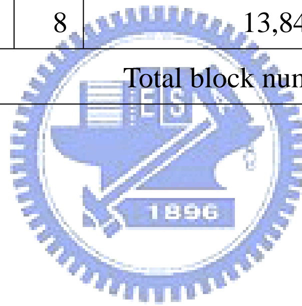


Table 5.17: Percentage of Fractional Motion Vectors with Different QP

Test Seqs. (QCIF)	QP	Total MV Number	Fractional MVs	%
akiyo	3	13,520	1,034	7.65
	4	13,552	1,225	9.04
	8	7,964	1,373	17.24
stefan	3	34,348	15,324	44.61
	4	33,744	15,385	45.59
	8	31,024	15,930	51.35
mobile	3	35,212	21,703	61.64
	4	35,192	21,663	61.56
	8	35,088	21,683	61.80

Table 5.18: Effects of Different QP to Execution Time of P-Frame Decoding on PACDSP

Test Seqs.	Execution Time (Cycles Per Frame)		
(QCIF)	QP = 3	QP = 4	QP = 8
akiyo	1,690,909	2,050,519	1,615,417
stefan	4,477,904	4,130,700	3,003,208
mobile	5,707,100	5,035,200	3,624,436

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

Processor	Freq. (MHz)	fps	Profile
TI C6201 [18]	200	28.57 (QCIF)	Not mentioned
ARM7TDMI [19]	12	15 (QCIF)	Simple profile
Philips TriMedia64 [20]	300	30 (4CIF)	Not mentioned
PACDSP	200	97.54 (QCIF)	Simple profile without error resilience

Chapter 6

Conclusion and Future Work

6.1 Conclusion

In this thesis, we consider the real-time implementation of MPEG-4 video decoder on PACDSP platform.

We first focused on the correct of decoding the bitstream, and the decoded frames have been verified with the reference software of MPEG-4, MoMuSys. Before the implementation, we analyzed the statistics of the MPEG-4 video decoder on PC. Therefore, we had an initial understand of the decoding flow and the critical part of computation. According to the analysis, we implemented the MPEG-4 video decoder on the PACDSP simulator.

After the implementation was verified, we further analyzed the characteristics of decoding procedure to find if there was any removable computation. Based on the analysis, we optimized the program sequence to reduce the computation complexity. The DC/AC predictions, IQs, and ITs were skipped with checking for the header and prediction results. In addition, we also utilized several general software optimization techniques, such as static rescheduling, loop-unrolling, and software-pipelining to reduce the stalls.

The optimization results were discussed and compared with other implementations. We can decode the MPEG-4 video bitstream over 97 frames per second in the best case, and the program size is 27 KB, which is smaller than the instruction cache size. In conclusion, the performance our implementation of MPEG-4 video decoder on PACDSP is competitive to other processors.

6.2 Future Work

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

- Combination of IQ and IDCT

  Since the computation of inverse quantization is followed by IDCT, we can simply combine these computations to reduce the number of memory load/store.

- Data structure refinement

  The data structure is very important to the implementation on DSPs. If the data structure is designed before the implementation, the memory accesses can be significantly reduced.

- Dual-core implementation

  Since the internal memory of PACDSP is 64 KB only and the access to external memory consumes much execution time, the amount of bitstream that is written to the memory is limited. Therefore, we can decode limited number of frames. Because the ARM core on the PACDSP platform can access the internal memory of PACDSP, we can manage the memory through ARM core, and the usable efficient memory size is enlarged.

- Add other MPEG-4 tools

  To simplify our implementation, the error-resilience tool in MPEG-4 simple profile is neglected. However, this tool is very important if the bitstream is transmitted through real channels. In addition, the special object-based compression technique can be implemented to extend the capability of PACDSP.

Bibliography

[1] SoC Technology Center of Industrial Technology Research Institute PACDSP2S0000, *PACDSP v2.0 — Instruction Set Menu*. June 2005.

[2] ISO/IEC 14496-2:2001, *Information Technology — Coding of Audio-Visual Objects — Part 2: Visual*. July 2001.

[3] A. Puri and A. Eleftheriadis, "MPEG-4: an object-based multimedia coding standard supporting mobile applications," *Mobile Networks Applic.*, vol. 3, pp. 5–32, 1998.

[4] A. Ebrahimi and C. Horne, "MPEG-4 natural video coding — an overview," *Signal Processing Image Commun.*, vol. 15, pp. 365–385, 2000.

[5] MPEG-4 Video Group, "MPEG-4 video verification model version 18.0," doc. no. ISO/IEC JTC1/SC29/WG11 N3908, Pisa, Jan. 2001.

[6] http://www.tnt.uni-hannover.de/project/eu/momusys.

[7] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach, 3rd ed.* San Francisco: Morgan Kaufmann Publishers, 2003.

[8] S. Sriram and C. Y. Hung, "MPEG-2 video decoding on the TMS320C6X DSP architecture," *IEEE Signal Systems Computer Conf.*, vol. 2, Nov. 1998, pp. 1735–1739.

[9] C. E. Fogg, "Survey of software and hardware VLC architectures," in *Proc. SPIE Image and Video Compression,* vol. 2186, May 1994, pp. 29-37.

[10] R. Prasad and R. Korada, "Efficient implementation of MPEG-4 video encoder on RISC core," *IEEE Trans. Consumer Electronics,* vol. 49, pp. 204-209, Feb. 2003.

[11] 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.

[12] 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.

[13] C. Y. Hung and P. Landman, "A compact IDCT design for MPEG video decoding," in *Proc. IEEE Workshop Signal Processing Systems*, Nov. 1997.

[14] G. Plonka and M. Tasche, "Reversible integer DCT algorithms," preprint, Gerhard-Mercator-Univ. Duisburg, 2002.

[15] Y. Chen and P. Hao, "Integer reversible transformation to make JPEG loseless," *Int. Conf. Siganl Processing, Beijing*, China, Sept. 2004, pp. 835–838.

[16] T.S. Chang, C.S. Kung, and C.W. Jen, "A simple processor core design for DCT/IDCT transform," *IEEE Trans. Circuits Syst. Video Technology*, vol. 10, no. 3 , pp. 439V-447, Apr. 2000.

[17] Texas Instuments, "*TMS320C64x Image/Video Processing Library — Programmers Reference*" Literature number SPRU023B, Oct. 2003.

[18] 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.

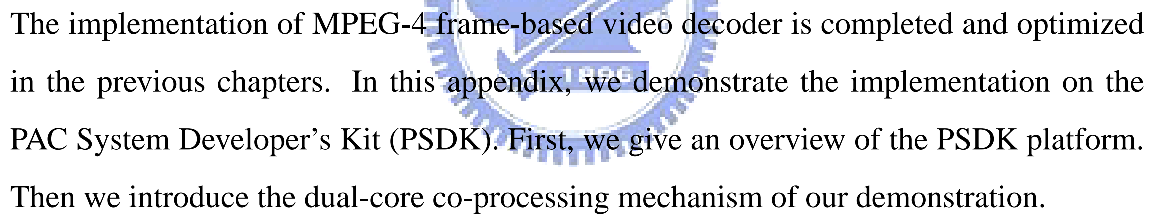
[19] K. Ramkishor and U. Gunashree, "Real time implementation of MPEG-4 video decoder on ARM7TDMI," in *Proc. Int. Symp. Intelligent Multimedia Video Speech Processing*, May 2001, pp. 522–526.

[20] 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.

[21] J. T. J. VanEijndhoven, J.T.J., *et al.*, "TriMedia CPU64 architecture," in *IEEE Int. Conf. Computer Design*, 1999

Appendix A

Demonstration of MPEG-4 Frame-Based Video Decoder on Dual-Core PSDK



The implementation of MPEG-4 frame-based video decoder is completed and optimized in the previous chapters. In this appendix, we demonstrate the implementation on the PAC System Developer's Kit (PSDK). First, we give an overview of the PSDK platform. Then we introduce the dual-core co-processing mechanism of our demonstration.

A.1 Overview of The PSDK 2.0 Platform

The PSDK platform is developed by SoC Technology Center (STC) of Industrial Technology Research Institute (ITRI) in Taiwan. The PSDK 2.0 system consists of following items:

- ARM Integrator-compatible Core Module: ARM920T CM
- Multi-ICE of ARM
- PACDSP Core Module (Burned in FPGA now)
- Generic peripherals (LCD translator)

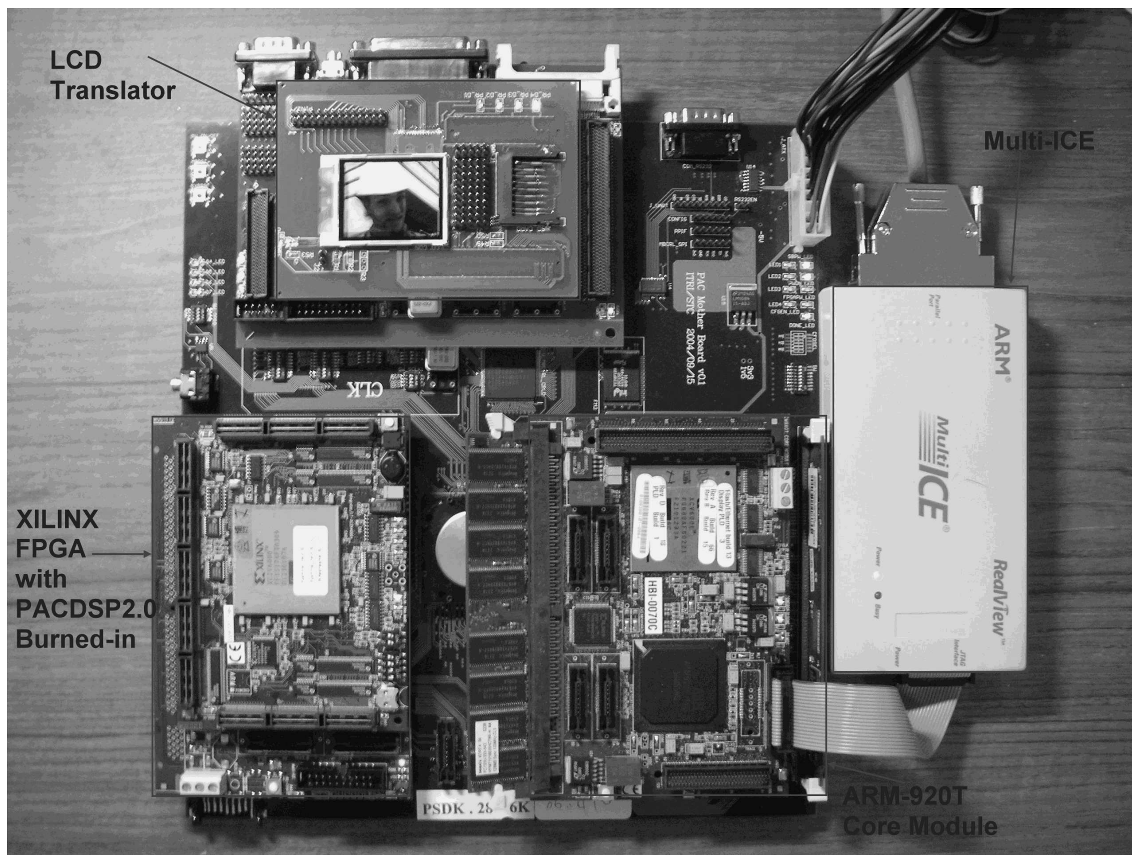


Figure A.1: PAC System Developer's Kit (PSDK) 2.0

The PSDK 2.0 hardware modules are shown in Fig. A.1. Since the PACDSP core module is replaced by an FPGA with the DSP design burned-in, the operating frequency of PACDSP is at most 33 MHz rather than a 250 MHz real chip. However, there is no difference for the functionality of a real chip and a burned-in FPGA.

It is noted that the operation of PACDSP is controlled by the ARM core, and its internal memory is accessible to the ARM core as well. For a PACDSP execution, we have to inform the DSP with its corresponding machine code of program and the data in the internal memory. Then we should give some signals to start the DSP execution. The memory map of our demonstration is shown in Fig. A.2, and it is noted that the start address of instruction is configurable and we set the instruction memory at 0xb0000000.

0x22000000	Start of internal memory
0x2200FFFF	End of internal memory
⋮	⋮
0x22005000	Start address of instruction memory (0xb0000000)
0x22005008	DSP Start Flag
⋮	⋮
0xb0000000	Start of instruction memory

Figure A.2: Memory map of the dualcore demonstration

A.2 Introduction to Dual-Core Demonstration

To keep the program sequence of our block-based implementation on PACDSP, we gather the block data whenever its decoding is completed. Since we have to write back the reference frame for P-frames decoding, the program flow of ARM core is different for I-frames decoding and P-frames decoding. We brief the co-processing mechanism in the following.

A.2.1 I-Frames Decoding

Since the utilization of interrupt is more complicated, we use the polling method to control the program flow in both processor. The internal memory of PACDSP is shared with ARM core, so we defined a flag, dec_block_complete, at the beginning address of internal memory, 0x22000000, to check if the decoding is completed. It is noted that the flag is set to 1 if one block data is decoded by PACDSP. The program flow is shown in Fig. A.3

Since the internal memory is a critical issue to our implementation and we have to store the headers and lookup tables, we can not store much bitstream in the internal memory. Our demonstration can show four consecutive intra-encoded frames of ”Foreman_qcif” sequence only.

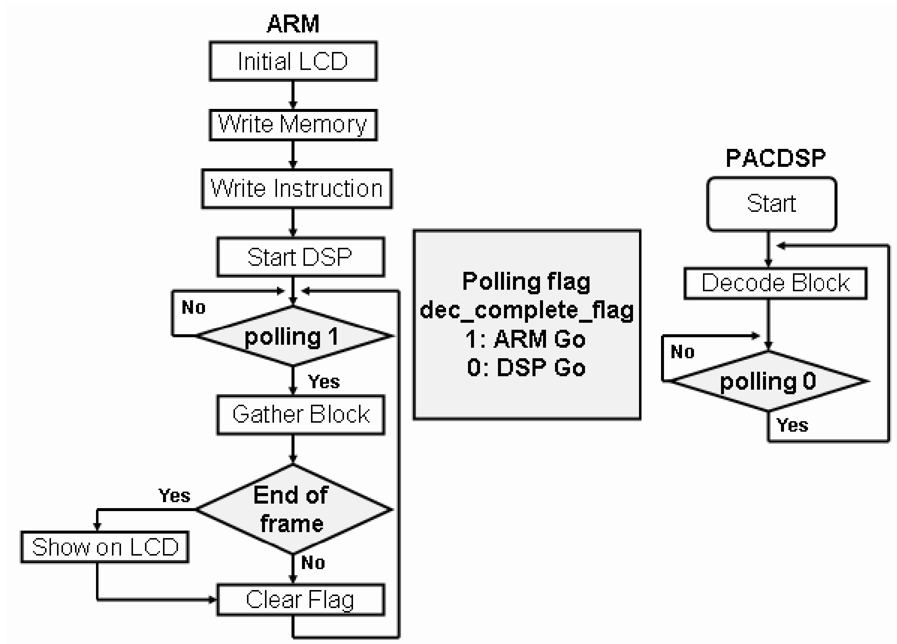


Figure A.3: Co-processing mechanism for I-frames

A.2.2 P-Frames Decoding

For P-frame decoding, because we suppose a decoded intra frame in the internal memory, we eliminate the bitstream of first intra-encoded frame. That is, all the frames decoded in this demonstration are P-frames. The program flow in P-frame decoding is shown in Fig. A.4.

Although we do not have to do DC and AC prediction in P-frame decoding, we have to store the reference frame, so we still can not store much bitstream in the internal memory. In this demonstration, we can decoding six consecutive P-frames of "Foreman_qcif" sequence and forty for "Akiyo_qcif" sequence.

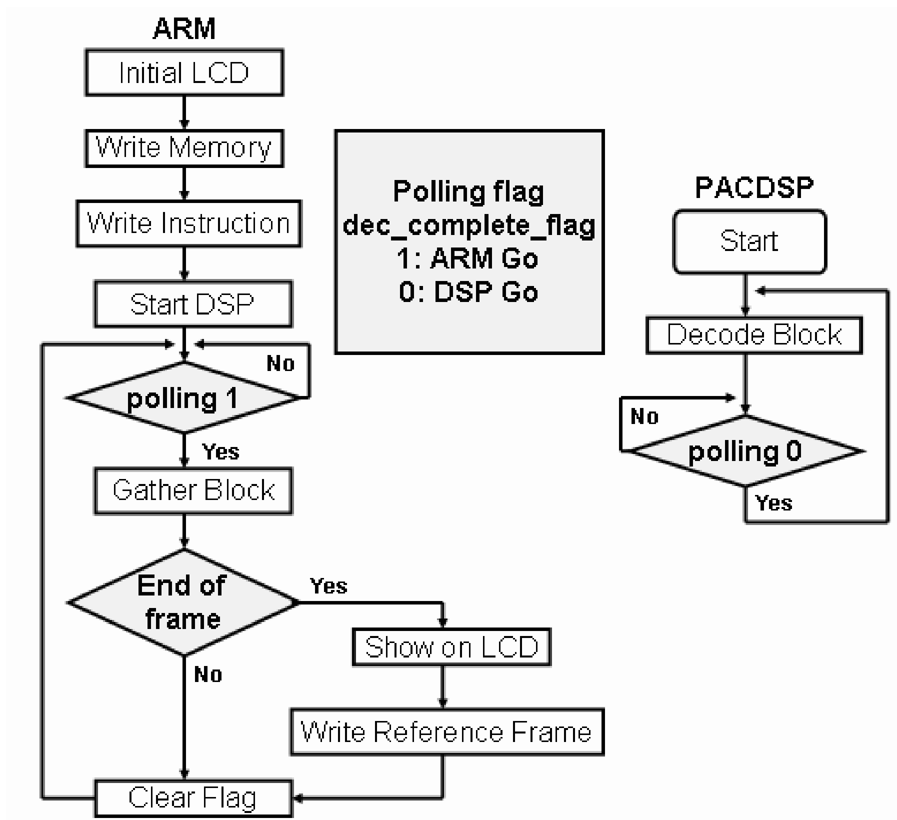
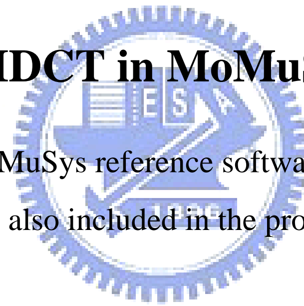


Figure A.4: Co-processing mechanism for P-frames

Appendix B

C Program and Assembly Code of IDCT

B.1 C Program of IDCT in MoMuSys



The C program of IDCT in MoMuSys reference software is shown in Fig. B.1. Note that clipping of block coefficients is also included in the program.

B.2 Original Assembly Code of IDCT

The initial IDCT design in of our implementation is listed in Fig. B.2 and B.3. Note that we use the instruction set of PACDSP.

B.3 Optimized Assembly Code of IDCT

The optimized IDCT design in of our implementation is listed in Fig. B.4 and B.5. Note that we use the instruction set of PACDSP.

Floating-Point Block_IDCT in MoMuSys (Int *coeff_in,Int block_out[][8], Int maxval)

```
Int          j1, i, j;
Double tmp[8], tmp1[8];
Double e, f, g, h,coeff[8][8],block[8][8];
static Double c0,c1,c2,c3,c4,c5,c6,c7;
Int v;
for (i=0;i<8;i++)
 for (j=0;j<8;j++)
  coeff[i][j] = (Double)coeff_in[i*8+j];
 c0=0.7071068;c1=0.4903926;c2=0.4619398;c
157348;c4=0.3535534;
 c5=0.2777851;c6=0.1913417;c7=0.0975452;
/* Horizontal */
/* Descan coefficients first */
for (i = 0; i < 8; i++) {
 for (j = 0; j < 8; j++) {
  tmp[j] =coeff[i][j];  }
 e = tmp[1] * c7 - tmp[7] * c1;
 h = tmp[7] * c7 + tmp[1] * c1;
 f = tmp[5] * c3 - tmp[3] * c5;
 g = tmp[3] * c3 + tmp[5] * c5;
 tmp1[0] = (tmp[0] + tmp[4]) * c4;
 tmp1[1] = (tmp[0] - tmp[4]) * c4;
 tmp1[2] = tmp[2] * c6 - tmp[6] * c2;
 tmp1[3] = tmp[6] * c6 + tmp[2] * c2;
 tmp[4] = e + f;
 tmp1[5] = e - f;
 tmp1[6] = h - g;
 tmp[7] = h + g;
 tmp[5] = (tmp1[6] - tmp1[5]) * c0;
 tmp[6] = (tmp1[6] + tmp1[5]) * c0;
 tmp[0] = tmp1[0] + tmp1[3];
 tmp[1] = tmp1[1] + tmp1[2];
 tmp[2] = tmp1[1] - tmp1[2];
 tmp[3] = tmp1[0] - tmp1[3];
 for (j = 0; j < 4; j++) {
  j1 = 7 - j;
  block[i][j] = tmp[j] + tmp[j1];
  block[i][j1] = tmp[j] - tmp[j1];
 }  }
/* Vertical */

 for (i = 0; i < 8; i++) {
  for (j = 0; j < 8; j++) {
   tmp[j] = block[j][i];
  }
  e = tmp[1] * c7 - tmp[7] * c1;
  h = tmp[7] * c7 + tmp[1] * c1;
  f = tmp[5] * c3 - tmp[3] * c5;
  g = tmp[3] * c3 + tmp[5] * c5;

  tmp1[0] = (tmp[0] + tmp[4]) * c4;
  tmp1[1] = (tmp[0] - tmp[4]) * c4;
  tmp1[2] = tmp[2] * c6 - tmp[6] * c2;
  tmp1[3] = tmp[6] * c6 + tmp[2] * c2;
  tmp[4] = e + f;
  tmp1[5] = e - f;
  tmp1[6] = h - g;
  tmp[7] = h + g;

  tmp[5] = (tmp1[6] - tmp1[5]) * c0;
  tmp[6] = (tmp1[6] + tmp1[5]) * c0;
  tmp[0] = tmp1[0] + tmp1[3];
  tmp[1] = tmp1[1] + tmp1[2];
  tmp[2] = tmp1[1] - tmp1[2];
  tmp[3] = tmp1[0] - tmp1[3];

  for (j = 0; j < 4; j++) {
   j1 = 7 - j;
   block[j][i] = tmp[j] + tmp[j1];
   block[j1][i] = tmp[j] - tmp[j1];
  }
 }

/*  Clipping */
for (i=0;i<8;i++)
     for(j=0;j<8;j++) {
     v = (Int) floor (block[i][j] + 0.5);
     block_out[i][j] =
(v<-maxval-1) ? -maxval-1 : ((v>maxval) ? maxval : v);
     }
```

Figure B.1: C program of IDCT in MoMuSys reference software including clipping.

Block_IDCT: (Horizontal Processing)

```
Block_IDCT:
{ NOP | NOP | NOP | MOVI.L C5,0x5A82 | NOP }
{ NOP | NOP | NOP | MOVI.L C8,0x3EC5 | NOP }
{ NOP | NOP | NOP | MOVI.L C9,0x3B21 | NOP }
{ NOP | NOP | NOP | MOVI.L C10,0x3537 | NOP }
{ NOP | NOP | NOP | MOVI.L C11,0x2D41 | NOP }
{ NOP | NOP | NOP | MOVI.L C12,0x238E | NOP }
{ NOP | NOP | NOP | MOVI.L C13,0x187E | NOP }
{ NOP | NOP | NOP | MOVI.L C14,0x0C7C | NOP }
{ NOP | NOP | NOP | MOVI.L A5,DCT_Block | NO
{ NOP | NOP | NOP | MOVI.H A5,DCT_Block | NO
{ NOP | NOP | NOP | MOVI.L A6,R_Block_2D | NO
{ NOP | NOP | NOP | MOVI.H A6,R_Block_2D | NC
{ SET_LBCI RBC2,0x8 | NOP | NOP | NOP | NOP }
Horizontal_Processing:
{ NOP | NOP | NOP | LW D0,A5 | NOP }
{ NOP | NOP | NOP | ADDI A5,A5,4 | NOP }
{ NOP | NOP | NOP | LW D8,A5 | NOP }
{ NOP | NOP | NOP | ADDI A5,A5,4 | NOP }
{ NOP | NOP | NOP | LW D1,A5 | NOP }
{ NOP | NOP | NOP | ADDI A5,A5,4 | NOP }
{ NOP | NOP | NOP | LW D9,A5 | NOP }
{ NOP | NOP | NOP | ADDI A5,A5,4 | NOP }
{ NOP | NOP | NOP | LW D2,A5 | NOP }
{ NOP | NOP | NOP | ADDI A5,A5,4 | NOP }
{ NOP | NOP | NOP | LW D10,A5 | NOP }
{ NOP | NOP | NOP | ADDI A5,A5,4 | NOP }
{ NOP | NOP | NOP | LW D3,A5 | NOP }
{ NOP | NOP | NOP | ADDI A5,A5,4 | NOP }
{ NOP | NOP | NOP | LW D11,A5 | NOP }
{ NOP | NOP | NOP | ADD D4,D0,D2 | FMUL
D12,D8,C14 }
{ NOP | NOP | NOP | NOP | SRAI D12,D12,15 }
;tmp[0]+tmp[4]  -- tmp[1]*c7
{ NOP | NOP | NOP | SUB D5,D0,D2 | FMUL
D13,D11,C8 }
{ NOP | NOP | NOP | NOP | SRAI D13,D13,15 }
;tmp[0]-tmp[4]  -- tmp[7]*c1
{ NOP | NOP | NOP | NOP | FMUL D14,D8,C8 }
{ NOP | NOP | NOP | NOP | SRAI D14,D14,15 }
;  -- tmp[1]*c1
{ NOP | NOP | NOP | NOP | FMUL D15,D11,C14 }
{ NOP | NOP | NOP | NOP | SRAI D15,D15,15 }
;  -- tmp[7]*c7
{ NOP | NOP | NOP | SUB D8,D12,D13 | FMUL
D0,D4,C11 }
{ NOP | NOP | NOP | NOP | SRAI D0,D0,15 }
;e = ...  -- tmp1[0] = (tmp[0]+tmp[4])*c4
{ NOP | NOP | NOP | ADD D11,D14,D15 | FMUL
D2,D5,C11 }
{ NOP | NOP | NOP | NOP | SRAI D2,D2,15 }
;h = ...  -- tmp1[1] = (tmp[0]-tmp[4])*c4
{ NOP | NOP | NOP | NOP | FMUL D12,D10,C10 }
{ NOP | NOP | NOP | NOP | SRAI D12,D12,15 }
;  -- tmp[5]*c3
{ NOP | NOP | NOP | NOP | FMUL D13,D9,C12 }
{ NOP | NOP | NOP | NOP | SRAI D13,D13,15 }
;  -- tmp[3]*c5
{ NOP | NOP | NOP | NOP | FMUL D14,D9,C10 }
{ NOP | NOP | NOP | NOP | SRAI D14,D14,15 }
;  -- tmp[3]*c3
{ NOP | NOP | NOP | NOP | FMUL D15,D10,C12 }
{ NOP | NOP | NOP | NOP | SRAI D15,D15,15 }
;  -- tmp[5]*c5
{ NOP | NOP | NOP | SUB D9,D12,D13 | FMUL D4,D1,C13 }
{ NOP | NOP | NOP | NOP | SRAI D4,D4,15 }
;f = ...  -- tmp[2]*c6
{ NOP | NOP | NOP | ADD D10,D14,D15 | FMUL D5,D3,C9 }
{ NOP | NOP | NOP | NOP | SRAI D5,D5,15 }
;g = ...  -- tmp[6]*c2
{ NOP | NOP | NOP | SUB D12,D8,D9 | FMUL D6,D1,C9 }
{ NOP | NOP | NOP | NOP | SRAI D6,D6,15 }
;tmp1[5] = e-f ...  -- tmp[2]*c2
{ NOP | NOP | NOP | SUB D13,D11,D10 | FMUL D7,D3,C13 }
{ NOP | NOP | NOP | NOP | SRAI D7,D7,15 }
;tmp1[6] = h-g ...  -- tmp[6]*c6
{ NOP | NOP | NOP | SUB D1,D4,D5 | ADD AC4,D8,D9 }
;tmp1[2] = tmp[2]*c6-tmp[6]*c2 ...  -- tmp[4] = e+f
{ NOP | NOP | NOP | ADD D3,D6,D7 | ADD AC7,D10,D11 }
;tmp1[3] = tmp[2]*c2+tmp[6]*c6 ...  -- tmp[7] = h+g
{ NOP | NOP | NOP | SUB D14,D13,D12 | ADD AC0,D0,D3 }
;d14 = tmp1[6]-tmp1[5] ...  -- tmp[0]= tmp1[0]+tmp1[3]
{ NOP | NOP | NOP | ADD D15,D13,D12 | ADD AC1,D2,D1 }
;d15 = tmp1[6]+tmp1[5] ...  -- tmp[1]= tmp1[1]+tmp1[2]
{ NOP | NOP | NOP | NOP | FMUL AC5,D14,C5 }
{ NOP | NOP | NOP | NOP | SRAI AC5,AC5,15 }
;tmp[5] = (tmp1[6]-tmp1[5])*c0
{ NOP | NOP | NOP | NOP | FMUL AC6,D15,C5 }
{ NOP | NOP | NOP | NOP | SRAI AC6,AC6,15 }
;tmp[6] = (tmp1[6]+tmp1[5])*c0
{ NOP | NOP | NOP | MOVI.L D15,28 | SUB AC2,D2,D1 }
;tmp[2] = tmp1[1]-tmp1[2]
{ NOP | NOP | NOP | MOVI.H D15,0 | SUB AC3,D0,D3 }
;tmp[3] = tmp1[0]-tmp1[3]
{ NOP | NOP | NOP | SUB A5,A5,D15 | ADD D0,AC0,AC7 }
{ NOP | NOP | NOP | SW D0,A5,0 | ADD D8,AC1,AC6 }
{ NOP | NOP | NOP | ADDI A5,A5,4 | ADD D1,AC2,AC5 }
{ NOP | NOP | NOP | SW D8,A5,0 | NOP }
{ NOP | NOP | NOP | ADDI A5,A5,4 | ADD D9,AC3,AC4 }
{ NOP | NOP | NOP | SW D1,A5,0 | NOP }
{ NOP | NOP | NOP | ADDI A5,A5,4 | SUB D2,AC3,AC4 }
{ NOP | NOP | NOP | SW D9,A5,0 | NOP }
{ NOP | NOP | NOP | ADDI A5,A5,4 | SUB D10,AC2,AC5 }
{ NOP | NOP | NOP | SW D2,A5,0 | NOP }
{ NOP | NOP | NOP | ADDI A5,A5,4 | SUB D3,AC1,AC6 }
{ NOP | NOP | NOP | SW D10,A5,0 | NOP }
{ NOP | NOP | NOP | ADDI A5,A5,4 | SUB D11,AC0,AC7 }
{ NOP | NOP | NOP | SW D3,A5,0 | NOP }
{ NOP | NOP | NOP | ADDI A5,A5,4 | NOP }
{ NOP | NOP | NOP | SW D11,A5,0 | NOP }
{ LBCB RBC2,Horizontal_Processing | NOP | NOP | NOP | NOP }
{ NOP | NOP | NOP | ADDI A5,A5,4 | NOP }
{ NOP | NOP | NOP | MOVI.L A5,DCT_Block | NOP }
{ NOP | NOP | NOP | MOVI.H A5,DCT_Block | NOP }
```

Figure B.2: Assembly code of our initial IDCT implementation (horizontal processing).

Block_IDCT：(Veritical and Clipping)

```
{ NOP | NOP | NOP | MOVI.L A6,R_Block_2D | NOP }
{ NOP | NOP | NOP | MOVI.H A6,R_Block_2D | NOP }
{ NOP | NOP | NOP | ADDI A6,A6,224 | NOP }
{ SET_LBCI RBC2,0x8 | NOP | NOP | NOP | NOP }
Vertical_Processing:
;{ NOP | ADDI D1,D1,1 | NOP | NOP | NOP }
;{ NOP | MOVI.L D7,4 | NOP | NOP | NOP }
{ NOP | NOP | NOP | LW D0,A5 | NOP }
{ NOP | NOP | NOP | ADDI A5,A5,32 | NOP }
{ NOP | NOP | NOP | LW D8,A5 | NOP }
{ NOP | NOP | NOP | ADDI A5,A5,32 | NOP }
{ NOP | NOP | NOP | LW D1,A5 | NOP }
{ NOP | NOP | NOP | ADDI A5,A5,32 | NOP }
{ NOP | NOP | NOP | LW D9,A5 | NOP }
{ NOP | NOP | NOP | ADDI A5,A5,32 | NOP }
{ NOP | NOP | NOP | LW D2,A5 | NOP }
{ NOP | NOP | NOP | ADDI A5,A5,32 | NOP }
{ NOP | NOP | NOP | LW D10,A5 | NOP }
{ NOP | NOP | NOP | ADDI A5,A5,32 | NOP }
{ NOP | NOP | NOP | LW D3,A5 | NOP }
{ NOP | NOP | NOP | ADDI A5,A5,32 | NOP }
{ NOP | NOP | NOP | LW D11,A5 | NOP }
{ NOP | NOP | NOP |ADD D4,D0,D2 | FMUL D12,D8,C14 }
{ NOP | NOP | NOP | NOP | SRAI D12,D12,15 }
;tmp[0]+tmp[4]  -- tmp[1]*c7
{ NOP | NOP | NOP |SUB D5,D0,D2 | FMUL D13,D11,C8 }
{ NOP | NOP | NOP | NOP | SRAI D13,D13,15 }
;tmp[0]-tmp[4]  -- tmp[7]*c1
{ NOP | NOP | NOP | NOP | FMUL D14,D8,C8 }
{ NOP | NOP | NOP | NOP | SRAI D14,D14,15 }
;  -- tmp[1]*c1
{ NOP | NOP | NOP | NOP | FMUL D15,D11,C14 }
{ NOP | NOP | NOP | NOP | SRAI D15,D15,15 }
;  -- tmp[7]*c7
{ NOP | NOP | NOP |SUB D8,D12,D13 | FMUL D0,D4,C11 }
{ NOP | NOP | NOP | NOP | SRAI D0,D0,15 }
;e = ...  -- tmp1[0] = (tmp[0]+tmp[4])*c4
{ NOP | NOP | NOP | ADD D11,D14,D15 |FMUL D2,D5,C11 }
{ NOP | NOP | NOP | NOP | SRAI D2,D2,15 }
;h = ...  -- tmp1[1] = (tmp[0]-tmp[4])*c4
{ NOP | NOP | NOP | NOP | FMUL D12,D10,C10 }
{ NOP | NOP | NOP | NOP | SRAI D12,D12,15 }
;  -- tmp[5]*c3
{ NOP | NOP | NOP |NOP | FMUL D13,D9,C12 }
{ NOP | NOP | NOP | NOP | SRAI D13,D13,15 }
;  -- tmp[3]*c5
{ NOP | NOP | NOP |NOP | FMUL D14,D9,C10 }
{ NOP | NOP | NOP | NOP | SRAI D14,D14,15 }
;  -- tmp[3]*c3
{ NOP | NOP | NOP |NOP | FMUL D15,D10,C12 }
{ NOP | NOP | NOP | NOP | SRAI D15,D15,15 }
;  -- tmp[5]*c5
{ NOP | NOP | NOP |SUB D9,D12,D13 | FMUL D4,D1,C13 }
{ NOP | NOP | NOP | NOP | SRAI D4,D4,15 }
;f = ...  -- tmp[2]*c6
{ NOP | NOP | NOP |ADD D10,D14,D15 | FMUL D5,D3,C9 }
{ NOP | NOP | NOP | NOP | SRAI D5,D5,15 }
;g = ...  -- tmp[6]*c2
{ NOP | NOP | NOP |SUB D12,D8,D9 |FMUL D6,D1,C9 }
{ NOP | NOP | NOP | NOP | SRAI D6,D6,15 }
;tmp1[5] = e-f ...  -- tmp[2]*c2
{ NOP | NOP | NOP | SUB D13,D11,D10 |
 FMUL D7,D3,C13 }
{ NOP | NOP | NOP | NOP | SRAI D7,D7,15 }
;tmp1[6] = h-g ...  -- tmp[6]*c6
{ NOP | NOP | NOP | SUB D1,D4,D5 | ADD AC4,D8,D9 }
;tmp1[2] = tmp[2]*c6-tmp[6]*c2 ...  -- tmp[4] = e+f
{ NOP | NOP | NOP | ADD D3,D6,D7 | ADD AC7,D10,D11 }
;tmp1[3] = tmp[2]*c2+tmp[6]*c6 ...  -- tmp[7] = h+g
{ NOP | NOP | NOP | SUB D14,D13,D12 | ADD AC0,D0,D3 }
;d14 = tmp1[6]-tmp1[5] ...  -- tmp[0]= tmp1[0]+tmp1[3]
{ NOP | NOP | NOP | ADD D15,D13,D12 | ADD AC1,D2,D1 }
;d15 = tmp1[6]+tmp1[5] ...  -- tmp[1]= tmp1[1]+tmp1[2]
{ NOP | NOP | NOP | NOP | FMUL AC5,D14,C5 }
{ NOP | NOP | NOP | NOP | SRAI AC5,AC5,15 }
;tmp[5] = (tmp1[6]-tmp1[5])*c0
{ NOP | NOP | NOP | NOP | FMUL AC6,D15,C5 }
{ NOP | NOP | NOP | NOP | SRAI AC6,AC6,15 }
;tmp[6] = (tmp1[6]+tmp1[5])*c0
{ NOP | NOP | NOP | MOVI.L D15,224 | SUB AC2,D2,D1 }
;tmp[2] = tmp1[1]-tmp1[2]
{ NOP | NOP | NOP | MOVI.H D15,0 | SUB AC3,D0,D3 }
;tmp[3] = tmp1[0]-tmp1[3]
{ NOP | NOP | NOP | SUB A6,A6,D15 | ADD D0,AC0,AC7 }
{ NOP | NOP | NOP | SUB A5,A5,D15 | NOP }
{ NOP | NOP | NOP | SW D0,A6,0 | ADD D8,AC1,AC6 }
{ NOP | NOP | NOP | ADDI A6,A6,32 | ADD D1,AC2,AC5 }
{ NOP | NOP | NOP | SW D8,A6,0 | NOP }
{ NOP | NOP | NOP | ADDI A6,A6,32 | ADD D9,AC3,AC4 }
{ NOP | NOP | NOP | SW D1,A6,0 | NOP }
{ NOP | NOP | NOP | ADDI A6,A6,32 | SUB D2,AC3,AC4 }
{ NOP | NOP | NOP | SW D9,A6,0 | NOP }
{ NOP | NOP | NOP | ADDI A6,A6,32 | SUB D10,AC2,AC5 }
{ NOP | NOP | NOP | SW D2,A6,0 | NOP }
{ NOP | NOP | NOP | ADDI A6,A6,32 | SUB D3,AC1,AC6 }
{ NOP | NOP | NOP | SW D10,A6,0 | NOP }
{ NOP | NOP | NOP | ADDI A6,A6,32 | SUB D11,AC0,AC7 }
{ NOP | NOP | NOP | SW D3,A6,0 | NOP }
{ NOP | NOP | NOP | ADDI A6,A6,32 | NOP }
{ NOP | NOP | NOP | SW D11,A6,0 | NOP }
{ LBCB RBC2,Vertical_Processing | NOP | NOP | NOP | NOP }
{ NOP | NOP | NOP | ADDI A5,A5,4 | NOP }
{ NOP | NOP | NOP | ADDI A6,A6,4 | NOP }
{ NOP | NOP | NOP | NOP | NOP }
{ SET_LBCI RBC2,64 | NOP | NOP | MOVI.L A6,R_Block_2D |
NOP }
{ NOP | NOP | NOP | MOVI.H A6,R_Block_2D | NOP }
Clip_Block:
{ NOP | NOP | NOP | LW D0,A6,0 | NOP }
{ NOP | NOP | NOP | NOP | NOP }
{ NOP | NOP | NOP | NOP | NOP }
{ NOP | NOP | NOP | SLTI D0,0,p12,p13 | NOP }
{ NOP | NOP | NOP | SGTI D0,255,p14,p15 | NOP }
{ LBCB RBC2,Clip_Block | NOP | NOP | SW D0,A6,0 | NOP }
{ NOP | NOP | NOP | (p12)SW C0,A6,0 | NOP }
{ NOP | NOP | NOP | (p14)SW C13,A6,0 | NOP }
{ NOP | NOP | NOP | ADDI A6,A6,4 | NOP }
{ JR R0 | NOP | NOP | NOP | NOP }
{ NOP | NOP | NOP | NOP | NOP }
{ NOP | NOP | NOP | NOP | NOP }
{ NOP | NOP | NOP | NOP | NOP }
```

Figure B.3: Assembly code of our initial IDCT implementation (vertical processing and clipping).

Optimized Block_IDCT: (Horizontal Processing)

```
Block_IDCT:
{ BDR R4 | BDT D0 | NOP | NOP | NOP } ;R4 = buffer1
{ BDR R5 | BDT D1 | NOP | NOP | NOP } ; R5 = buffer2
{ BDR R3 | BDT D4 | NOP | NOP | NOP } ;R3 = original rest
{ BDR R2 | BDT D3 | NOP | NOP | NOP } ; R2 = original bit count
{ BDR R6 | BDT D8 | NOP | NOP | NOP }
{ NOP | MOVI.L C13,0xFF | NOP | MOVI.L C13,0xFF | NOP }
{ NOP | MOVI.L C14,0xFF00 | NOP | MOVI.L C14,0xFF00 |NOP }
{ NOP | MOVI.L C14,0xFFFF | NOP | MOVI.L C14,0xFFFF |NOP }
{ NOP | MOVI.L C1,0x3EC5 | NOP | MOVI.L C1,0x3EC5 |NOP }
{ NOP | MOVI.L C2,0x3B21 | NOP | MOVI.L C2,0x3B21 | NOP }
{ NOP | MOVI.L C3,0x3537 | NOP |MOVI.L C3,0x3537 | NOP }
{ NOP | MOVI.L C4,0x2D41 | NOP | MOVI.L C4,0x2D41 |NOP }
{ NOP | MOVI.L C5,0x238E | NOP | MOVI.L C5,0x238E | NOP }
{ NOP | MOVI.L C6,0x187E | NOP | MOVI.L C6,0x187E | NOP }
{ NOP | MOVI.L C7,0x0C7C | NOP | MOVI.L C7,0x0C7C |NOP }
{ NOP | MOVI.L C8,0x0 | NOP |MOVI.L C8,0x0 | NOP }
{ NOP | MOVI.H C8,0x4 | NOP | MOVI.H C8,0x4 | NOP }
{ NOP | MOVI.L C9,0xC13B | NOP | MOVI.L C9,0xC13B |NOP } ;-C1 -3EC5
{ NOP | MOVI.L C10,0xCAC9 | NOP | MOVI.L C10,0xCAC9 | NOP };-C3
{ NOP | MOVI.L C11,0xDC72 | NOP | MOVI.L C11,0xDC72 | NOP };-C5
{ NOP | MOVI.L C12,0xF384 | NOP | MOVI.L C12,0xF384 |NOP };-C7
{ NOP | MOVI.L A5,DCT_Block | NOP |MOVI.L A5,DCT_Block |NOP } ;.word
{ NOP | MOVI.H A5,DCT_Block | NOP | MOVI.H A5,DCT_Block | NOP } ;.word
{ SET_LBCI RBC2,0x4 | NOP | NOP |ADDI A5,A5,128 | NOP }
Horizontal_Processing:
{ NOP | LW D8,A5,4 | NOP | LW D8,A5,4 | NOP }
{ NOP | LW D9,A5,12 | NOP | LW D9,A5,12 | NOP }
{ NOP | LW D10,A5,20 | NOP | LW D10,A5,20 | NOP }
{ NOP | LW D11,A5,28 | FMUL AC4,D8,C7 | LW D11,A5,28 |FMUL AC4,D8,C7 };tmp[1]*c7
{ NOP | LW D0,A5 | FMUL AC5,D8,C5 |LW D0,A5 | FMUL AC5,D8,C5 };tmp[1]*c5
{ NOP | LW D1,A5,8 | FMUL AC6,D8,C3 | LW D1,A5,8 | FMUL AC6,D8,C3 };tmp[1]*c3
{ NOP | LW D2,A5,16 | FMUL AC7,D8,C1 | LW D2,A5,16 | FMUL AC7,D8,C1 };tmp[1]*c1
{ NOP | LW D3,A5,24 | FMAC AC4,D9,C11 | LW D3,A5,24 | FMAC AC4,D9,C11 };-tmp[3]*c5
{ NOP | NOP | FMAC AC5,D9,C9 |NOP | FMAC AC5,D9,C9 } ;-tmp[3]*c1
{ NOP | NOP | FMAC AC6,D9,C12 |NOP | FMAC AC6,D9,C12 } ;-tmp[3]*c7
{ NOP | NOP | FMAC AC7,D9,C3 |NOP | FMAC AC7,D9,C3 } ;+tmp[3]*c3
{ NOP | NOP | FMAC AC4,D10,C3 |NOP | FMAC AC4,D10,C3 } ;+tmp[5]*c3
{ NOP | NOP | FMAC AC5,D10,C7 |NOP | FMAC AC5,D10,C7 } ;+tmp[5]*c7
{ NOP | ADD D4,D0,D2 | FMAC AC6,D10,C9 | ADD D4,D0,D2 |FMAC AC6,D10,C9 } ;-tmp[5]*c1
{ NOP | SUB D5,D0,D2 | FMAC AC7,D10,C5 | SUB D5,D0,D2 |FMAC AC7,D10,C5 } ;+tmp[5]*c5
{ NOP |NOP |FMUL D0,D4,C4 |NOP | FMUL D0,D4,C4 } ;tmp1[0]->69
{ NOP |NOP |FMUL D2,D5,C4 |NOP | FMUL D2,D5,C4 } ;tmp1[1]->76
{ NOP | NOP | FMUL D4,D1,C6 |NOP | FMUL D4,D1,C6 } ;tmp[2]*c6 ->77
{ NOP | NOP | FMUL D5,D3,C2 |NOP | FMUL D5,D3,C2 } ;tmp[6]*c2 ->84
{ NOP | NOP | FMUL D6,D3,C6 |NOP | FMUL D6,D3,C6 } ;tmp[6]*c6 ->85
{ NOP | NOP | FMUL D7,D1,C2 |NOP | FMUL D7,D1,C2 } ;tmp[2]*c2 ->92
{ NOP | NOP | FMAC AC4,D11,C9 |NOP | FMAC AC4,D11,C9 } ;-tmp[7]*c1
{ NOP | NOP | FMAC AC5,D11,C3 |NOP | FMAC AC5,D11,C3 } ;+tmp[7]*c3
{ NOP | SUB D1,D4,D5 | FMAC AC6,D11,C11 | SUB D1,D4,D5 | FMAC AC6,D11,C11 } ;-tmp[7]*c5
{ NOP | ADD D3,D6,D7 | FMAC AC7,D11,C7 |ADD D3,D6,D7 | FMAC AC7,D11,C7 } ;+tmp[7]*c7
{ NOP |NOP |BF AC0,D0,D3 |NOP |BF AC0,D0,D3 } ;USE BF
{ NOP |NOP |BF AC2,D2,D1 |NOP |BF AC2,D2,D1 } ;USE BF
{ NOP | NOP | BF D0,AC0,AC7 |NOP |BF D0,AC0,AC7 } ;USE BF
{ NOP | ADDI D0,D0,1024 | BF D8,AC2,AC6 | ADDI D0,D0,1024 | BF D8,AC2,AC6 }
{ NOP | ADDI D8,D8,1024 | SRAI D0,D0,11 | ADDI D8,D8,1024 | SRAI D0,D0,11 }
{ NOP | ADDI D1,D1,1024 | SRAI D8,D8,11 | ADDI D1,D1,1024 |SRAI D8,D8,11 }
{ NOP | ADDI D9,D9,1024 | SRAI D1,D1,11 | ADDI D9,D9,1024 | SRAI D1,D1,11 }
{ NOP | DSW D0,D8,A5,0 | SRAI D9,D9,11 | DSW D0,D8,A5,0 | SRAI D9,D9,11 }
{ NOP |DSW D9,D1,A5,24 |BF D4,AC3,AC5 |DSW D9,D1,A5,24 |BF D4,AC3,AC5 } ;USE BF
{ NOP | ADDI D4,D4,1024 | BF D10,AC1,AC4 | ADDI D4,D4,1024 | BF D10,AC1,AC4 }
{ NOP | ADDI D10,D10,1024 | SRAI D4,D4,11 | ADDI D10,D10,1024 |SRAI D4,D4,11 }
{ NOP | ADDI D5,D5,1024 | SRAI D10,D10,11 | ADDI D5,D5,1024 | SRAI D10,D10,11 }
{LBCB RBC2,Horizontal_Processing | ADDI D11,D11,1024 | SRAI D5,D5,11 | ADDI D11,D11,1024 | SRAI D5,D5,11 }
{ NOP | DSW D4,D10,A5,8 |SRAI D11,D11,11 | DSW D4,D10,A5,8 | SRAI D11,D11,11 }
{ NOP | DSW D11,D5,A5,16| NOP | DSW D11,D5,A5,16|NOP }
{ NOP | ADDI A5,A5,32 |NOP | ADDI A5,A5,32 | NOP }
```

Figure B.4: Assembly code of optimized IDCT implementation (horizontal processing).

Optimized Block_IDCT: (Vertical and Clipping)

```
;vertical
{ NOP | MOVI.L A5,DCT_Block | NOP | MOVI.L A5,DCT_Block | NOP }
{ NOP | MOVI.H A5,DCT_Block | NOP | MOVI.H A5,DCT_Block | NOP }
{ NOP | MOVI.L A6,R_Block_2D | NOP | MOVI.L A6,R_Block_2D | NOP }
{ NOP | MOVI.H A6,R_Block_2D | NOP | MOVI.H A6,R_Block_2D | NOP }
{ NOP | NOP | NOP | ADDI A5,A5,16 | NOP }
{ SET_LBCI RBC2,0x4 | NOP | NOP | ADDI A6,A6,16 | NOP }
Vertical_Processing:
{ NOP | LW D0,A5 | NOP | LW D0,A5 | NOP }
{ NOP | LW D8,A5,32 | NOP | LW D8,A5,32 | NOP }
{ NOP | LW D1,A5,64 | NOP | LW D1,A5,64 | NOP }
{ NOP | LW D9,A5,96 | NOP | LW D9,A5,96 | NOP }
{ NOP | ADDI A5,A5,128 | FMUL AC4,D8,C7 | ADDI A5,A5,128 | FMUL AC4,D8,C7 }
{ NOP | LW D2,A5,0 | FMUL AC5,D8,C5 | LW D2,A5,0 | FMUL AC5,D8,C5 }
{ NOP | LW D10,A5,32 | FMUL AC6,D8,C3 | LW D10,A5,32 | FMUL AC6,D8,C3 }
{ NOP | LW D3,A5,64 | FMUL AC7,D8,C1 | LW D3,A5,64 | FMUL AC7,D8,C1 }
{ NOP | LW D11,A5,96 | FMAC AC4,D9,C11 | LW D11,A5,96 | FMAC AC4,D9,C11 }
{ NOP | ADDI A5,A5,96 | FMAC AC5,D9,C9 | ADDI A5,A5,96 | FMAC AC5,D9,C9 }
{ NOP | NOP | FMAC AC6,D9,C12 | NOP | FMAC AC6,D9,C12 } ;-tmp[3]*c7
{ NOP | NOP | FMAC AC7,D9,C3 | NOP | FMAC AC7,D9,C3 } ;+tmp[3]*c3
{ NOP | NOP | FMAC AC4,D10,C3 | NOP | FMAC AC4,D10,C3 } ;+tmp[5]*c3
{ NOP | NOP | FMAC AC5,D10,C7 | NOP | FMAC AC5,D10,C7 } ;+tmp[5]*c7
{ NOP | ADD D4,D0,D2 | FMAC AC6,D10,C9 | ADD D4,D0,D2 | FMAC AC6,D10,C9 } ;-tmp[5]*c1
{ NOP | SUB D5,D0,D2 | FMAC AC7,D10,C5 | SUB D5,D0,D2 | FMAC AC7,D10,C5 } ;+tmp[5]*c5
{ NOP | NOP | FMUL D0,D4,C4 | NOP | FMUL D0,D4,C4 } ;tmp1[0]->69
{ NOP | NOP | FMUL D2,D5,C4 | NOP | FMUL D2,D5,C4 } ;tmp1[1]->76
{ NOP | NOP | FMUL D4,D1,C6 | NOP | FMUL D4,D1,C6 } ;tmp[2]*c6 ->77
{ NOP | NOP | FMUL D5,D3,C2 | NOP | FMUL D5,D3,C2 } ;tmp[6]*c2 ->84
{ NOP | NOP | FMUL D6,D3,C6 | NOP | FMUL D6,D3,C6 } ;tmp[6]*c6 ->85
{ NOP | NOP | FMUL D7,D1,C2 | NOP | FMUL D7,D1,C2 } ;tmp[2]*c2 ->92
{ NOP | NOP | FMAC AC4,D11,C9 | NOP | FMAC AC4,D11,C9 } ;-tmp[7]*c1
{ NOP | NOP | FMAC AC5,D11,C3 | NOP | FMAC AC5,D11,C3 } ;+tmp[7]*c3
{ NOP | SUB D1,D4,D5 | FMAC AC6,D11,C11 | SUB D1,D4,D5 | FMAC AC6,D11,C11 } ;-tmp[7]*c5
{ NOP | ADD D3,D6,D7 | FMAC AC7,D11,C7 | ADD D3,D6,D7 | FMAC AC7,D11,C7 } ;+tmp[7]*c7
{ NOP | NOP | BF AC0,D0,D3 | NOP | BF AC0,D0,D3 } ;USE BF
{ NOP | NOP | BF AC2,D2,D1 | NOP | BF AC2,D2,D1 } ;USE BF
{ NOP | NOP | BF D0,AC0,AC7 | NOP | BF D0,AC0,AC7 } ;USE BF
{ NOP | ADD D0,D0,C8 | BF D8,AC2,AC6 | ADD D0,D0,C8 | BF D8,AC2,AC6 }
{ NOP | ADD D8,D8,C8 | SRAI D0,D0,19 | ADD D8,D8,C8 | SRAI D0,D0,19 }
{ NOP | ADD D1,D1,C8 | SRAI D8,D8,19 | ADD D1,D1,C8 | SRAI D8,D8,19 }
{ NOP | ADD D9,D9,C8 | SRAI D1,D1,19 | ADD D9,D9,C8 | SRAI D1,D1,19 }
{ NOP | SW D0,A6,0 | SRAI D9,D9,19 | SW D0,A6,0 | SRAI D9,D9,19 }
{ NOP | SW D8,A6,32 | BF D4,AC3,AC5 | SW D8,A6,32 | BF D4,AC3,AC5 } ;USE BF
{ NOP | ADD D4,D4,C8 | BF D10,AC1,AC4 | ADD D4,D4,C8 | BF D10,AC1,AC4 }
{ NOP | ADD D10,D10,C8 | SRAI D4,D4,19 | ADD D10,D10,C8 | SRAI D4,D4,19 }
{ NOP | ADD D5,D5,C8 | SRAI D10,D10,19 | ADD D5,D5,C8 | SRAI D10,D10,19 }
{ NOP | ADD D11,D11,C8 | SRAI D5,D5,19 | ADD D11,D11,C8 | SRAI D5,D5,19 }
{ NOP | SW D4,A6,64 | SRAI D11,D11,19| SW D4,A6,64 | SRAI D11,D11,19 }
{ NOP | SW D10,A6,96 | NOP | SW D10,A6,96| NOP }
{ NOP | ADDI A6,A6,128 | NOP | ADDI A6,A6,128 | NOP }
{ NOP | SW D11,A6,0 | NOP | SW D11,A6,0 | NOP }
{ NOP | SW D5,A6,32 | NOP | SW D5,A6,32 | NOP }
{ LBCB RBC2,Vertical_Processing | SW D9,A6,64 | NOP | SW D9,A6,64 | NOP }
{ NOP | SW D1,A6,96 | NOP | SW D1,A6,96 | NOP }
{ NOP | ADDI A5,A5,-220 | NOP | ADDI A5,A5,-220 | NOP }
{ NOP | ADDI A6,A6,-124 | NOP | ADDI A6,A6,-124 | NOP }
;clipping
{ SET_LBCI RBC1,32 | MOVI.L A6,R_Block_2D | NOP | MOVI.L A6,R_Block_2D | NOP }
{ NOP | MOVI.H A6,R_Block_2D | NOP | MOVI.H A6,R_Block_2D | NOP }
{ NOP | NOP | NOP | ADDI A6,A6,128 | NOP }
Clip_Block:
{ NOP | LW D0,A6,0 | NOP | LW D0,A6,0 | NOP }
{ NOP | NOP | NOP | NOP | NOP }
{ NOP | NOP | NOP | NOP | NOP }
{ NOP | SLTI D0,-256,p2,p3 | NOP | SLTI D0,-256,p4,p5 | NOP }
{ NOP | SGTI D0,255,p6,p7 | NOP | SGTI D0,255,p8,p9 | NOP }
{ LBCB RBC1,Clip_Block | SW D0,A6,0 | NOP | SW D0,A6,0 | NOP }
{ NOP | (p2)SW C14,A6,0 | NOP | (p4)SW C14,A6,0 | NOP }
{ NOP | (p6)SW C13,A6,0 | NOP | (p8)SW C13,A6,0 | NOP }
{ NOP | ADDI A6,A6,4 | NOP | ADDI A6,A6,4 | NOP }
```

Figure B.5: Assembly code of optimized IDCT implementation (vertical processing and clipping).

自傳

蔡崇諺，男，民國七十年十一月三十日出生於台灣省桃園縣。高中就讀於國立桃園高級中學，民國九十三年六月畢業於交通大學電機與控制工程學系，並於同年九月進入交通大學電子工程研究所碩士班就讀，於民國九十五年六月取得碩士學位，論文題目為:『在 PACDSP 平台上之 MPEG-4 視訊解碼器軟體實現』，研究範圍與興趣為：軟、硬體和 DSP 平台上之系統整合與開發，主要應用範圍在多媒體訊號處理與壓縮方面。