國 立 交 通 大 學

電子工程學系 電子研究所碩士班

碩 士 論 文

MPEG-4 物件視訊解碼器在 PACDSP 平台上之

軟體實現

Software Implementation of MPEG-4 Object-Based Video Decoder on PACDSP Platform

研 究 生：許介遠

指導教授：林大衛 博士

中 華 民 國 九 十 六 年 六 月


E
S
A
1896

MPEG-4 物件視訊解碼器在 PACDSP 平台上之軟體實現

Software Implementation of MPEG-4 Object-Based Video Decoder on PACDSP Platform

研 究 生：許介遠　　Student: Chieh Yuan Hsu

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 2007

Hsinchu, Taiwan, Republic of China

中 華 民 國 九 十 六 年 六 月


E
S
A
1896

MPEG-4 物件視訊解碼器在 PACDSP 平台上之軟體實現

研究生：許介遠　　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 物件視訊解碼器之實現，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。我們也完成了雙核心的實現以提高整體的效能。

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 基於物件解碼器之運算複雜度並藉此找到有效率的實現方法。為了能減少運算量以及在 PACDSP 上實現，我們將離散餘弦反轉換（IDCT）轉為整數點運算(fixed point)，並且討論其效能及精確度。最後，我們的實現之精確度能夠符合 IEEE 1180-1190 標準之規範。同時，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。接著，我們討論了在雙核心平台上的實現方法以提高效能。為了加速執行時間，我們利用了 PACDSP

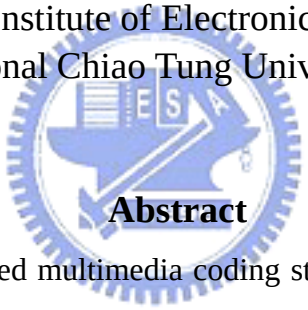
的特性，將規律之運算分佈於兩組以增加處理器之效能。我們也使用單指令多資料（SIMD）指令以及一般指令層級平行化來減少處理器之延遲。在演算法上，我們根據離散餘弦轉換（DCT）之特性來跳過多餘的運算。在所有的最佳化之後，我們在最差情況下，對於一個工作在 200MHz 的真實 PACDSP 晶片而言，能夠達到每秒 46 張的解碼，滿足每秒三十張即時解碼的要求。而整個程式的大小為 30 Kbytes，也小於 PACDSP 的程式快取記憶體大小 32 Kbytes。最後我們在 PSDK 平台上展示了雙核心的實現結果。

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、雙核心實現之設計、實作策略、最佳化方法、以及最後實現之結果。

Software Implementation of MPEG-4 Object-Based Video Decoder on PACDSP Platform

Student: Chieh-Yuan Hsu 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 the MPEG-4 object-based video decoder on the PACDSP platform, which consists of a VLIW digital signal processor (DSP) and an ARM920T processor. We complete many analyses to optimize the program flow and utilize the advantage of VLIW processor to achieve real-time decoding. Finally, a dual-core demonstration is completed and verified.

In our implementation, the MPEG-4 reference software, MoMuSys, is used as a model to verily our implementation. First, we analyze the computational complexity of the MPEG-4 object-based video decoder, and find efficient algorithms for the implementation. In order to reduce the complexity and to realize on PACDSP, we implement the fixed point inverse discrete cosine transform (IDCT), and then discuss the efficiency and accuracy. At last, our implementation can pass the accuracy test of

IEEE 1180-1190 standard and the performance of our algorithm is also competitive to other implementations. Then, we discuss the design of dual-core implementation to improve the performance. In order 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. For algorithmic optimization, we skip unnecessary computations according to the nature of discrete cosine transform (DCT). After all the optimizations, in the worst case, our implementation of decoder decodes 46 frame-per-second, which can achieve real-time decoding, 30 frame-per-second, for a real PACDSP chip running over 200 MHz. The code size is 30 Kbytes, which is smaller than the 32 Kbytes instruction cache on PACDSP. Finally, we demonstrate a 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, dual-core design, implementation strategies, the optimization methods, and the results of our implementation are discussed.

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，從踏入交通大學電子所開始，多虧老師的循循善誘，不但給予我在課業、研究上的幫助，使我學到了分析問題及解決問題的能力。同時老師樂觀的生活態度也影響了我，讓我更有勇氣面對各種困難。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 這個大家族的日子裡，實驗室所提供完善的研究資源。承蒙崑健、家揚、俊榮、朝雄、鴻志、亦善、榮煌等學長的提攜與照顧。而實驗室的同伴，政達、志岡、育群、柏昇、耀鈞、凱庭、錫祺、浩廷、育成、順成等，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。

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。僅將本論文獻給我敬愛的父母，許回福先生、廖淑香女士。

許介遠

民國九十六年六月于新竹


E
S
A
1896

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In modern day, compression of audio-visual information becomes more and more important, especially for applications on mobile devices. The higher the compression ratio, the greater the cost saving. Due to the increased demand on computing power, digital signal processors (DSPs) are popularly used in these mobile devices. We consider the implementation of the MPEG-4 object-based video decoder on the PACDSP platform.

The Moving Pictures Experts Group (MPEG) of the International Standardization Organization (ISO) produced the MPEG-4 stand aid for digital video and audio compression [5]. The MPEG-4 standard has been adopted widely in many consumer products. Our implementation of the video decoder is based on enhancing the functionality of the decoder of [6] . However, certain tools (such as error resilience and scalable coding) are left to potential future work.

PACDSP is a high performance, low cost VLIW (very long instruction word) DSP for multimedia applications [1]. The instruction set architecture (ISA) of PACDSP supports SIMD (single instruction multiple data) instructions, which are suitable for audio and video applications. 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 analyze complexity of the MPEG-4 reference software, and we also present our dual-core design and efficient implementation strategies of the MPEG-4 video decoder

on ARM. The optimization of the MPEG-4 video decoder on PACDSP is discussed in chapter 5. Chapter 6 shows the performance of our implementation, which includes the code size, data size and the decoding frame rate. Finally, we give some conclusions and list the future work in chapter 7.

Chapter 2

Overview of the MPEG-4 Video Standard

The contents of this chapter have been taken to a large extent from [5]–[9].

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

2.1 Structure of MPEG-4 Video Data

The concepts of video objects (VOs) and their temporal instances, video object planes (VOPs), are central to MPEG-4 video. 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 data in a top-down hierarchical structure. The meanings of the hierarchical layers are as follows.

- VideoSession (VS): A video session simply consists of an ordered collection of

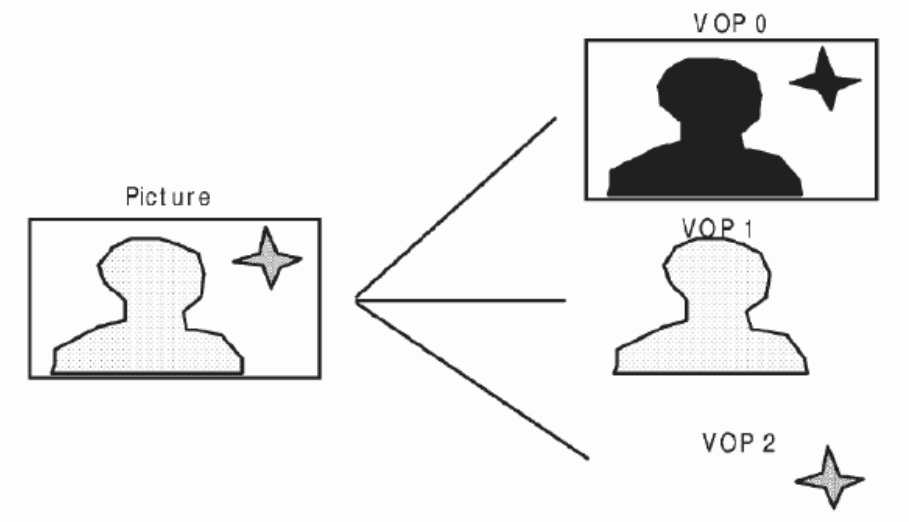


Figure 2.1: Segmentation of a frame into VOPs (from [7]).

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 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 (single layer) form, 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 video object planes together. GOVs can provide points in the bitstream where VOPs are encoded independently from one another, and can thus provide random access points into the bitstream.

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

There are four types of VOP defined in MPEG-4, as illustrated in Fig. 2.3. These are briefly explained below:

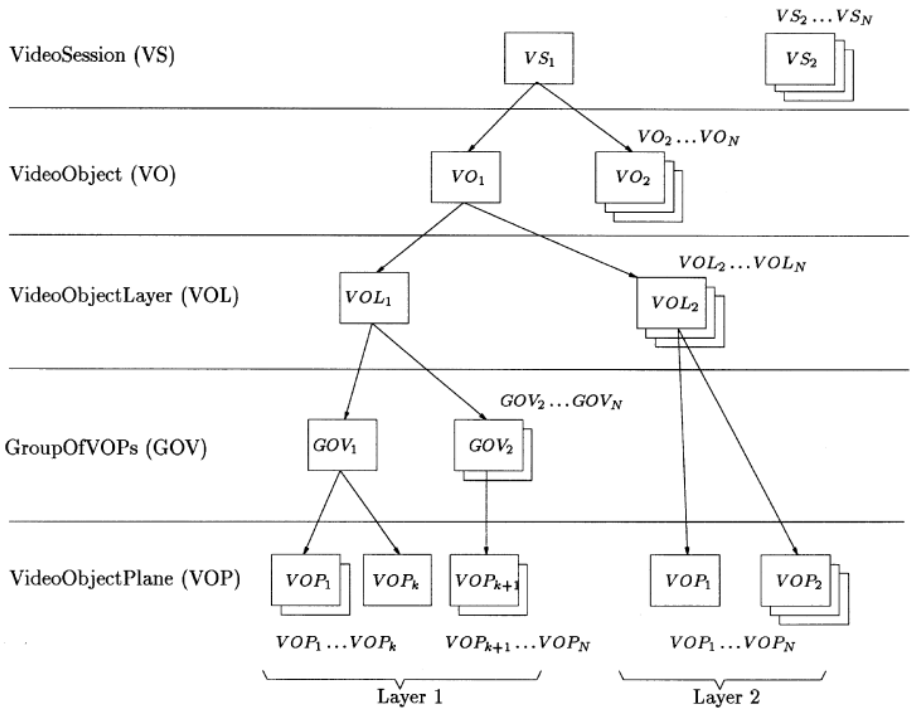


Figure 2.2: Structure of coded video data (from [8]).

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

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

3. A bidirectionally predictive-coded (B) VOP is a VOP that 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 that is coded using prediction

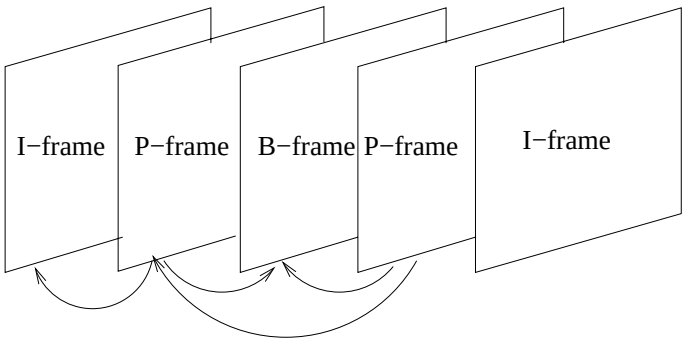


Figure 2.3: Types of VOP.

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 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 [5]–[9].

Fig. 2.5 is a structure of video decoder without any scalability feature. The decoder is mainly composed of three parts: shape decoder, motion decoder and texture decoder. The reconstructed VOP is obtained by combining the decoded shape, texture and motion information. The part of shape coding constitutes the major difference between frame-based and object-based coding.

2.2.1 Shape Coding

The ability to represent arbitrary shapes is an important capability of the MPEG-4 video standard. For each VO given as a sequence of VOPs of arbitrary shapes, the corresponding alpha planes is also given (generated via segmentation or via chroma-key). There are two kinds of alpha planes in MPEG-4, binary and gray scale. Binary alpha planes are encoded by modified context-based binary arithmetic encoding (CAE) and gray scale alpha planes are encoded by motion compensated discrete-cosine transform (DCT) similar to texture coding. An alpha plane is bounded by an extended rectangular bounding box. The bounded alpha plane is partitioned into blocks of 16×16 samples called alpha block and the encoding/decoding process is done per alpha block.

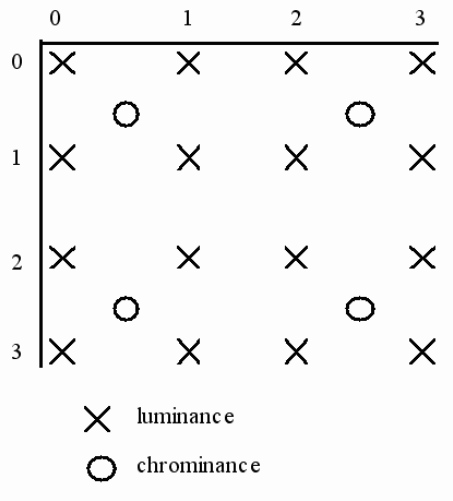


Figure 2.4: Positions of luminance and chrominance samples in 4:2:0 data (from [9]).

Binary Shape Coding

CAE and motion compensation are the basic tools for encoding binary alpha blocks (BABs) which are the primary unit in binary shape coding. Each BAB can be coded in one of the following modes:

1. The block is all transparent. In this case no coding is necessary. Texture information is not coded for such blocks either.
2. The block is all opaque. Shape coding is not necessary in this case, but texture information needs to be coded.
3. The block is coded using IntraCAE without use of past information.
4. Motion vector difference (MVD) is zero but the block is not updated.
5. MVD is non-zero, but the block is not updated.
6. MVD is zero and the block is updated. InterCAE is used for coding the block update.
7. MVD is non-zero, and the block is coded by InterCAE.

Table 2.1 shows the BAB types and VOP types they are used in.

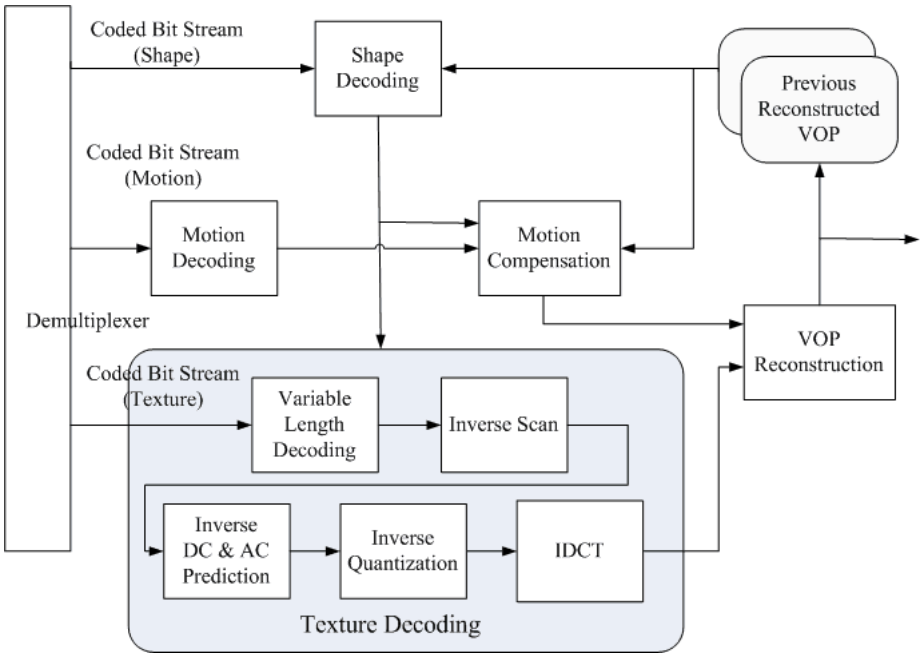


Figure 2.5: Simplified structure of the video decoder (from [5]).

CAE is used to code each binary pixel of the BAB. Prior to coding the first pixel, the arithmetic encoder is initialized. Each binary pixel is then encoded in raster order. The process for encoding a given pixel is as follows:

1. Compute a context number.
2. Index a probability table using the context number.
3. Use the indexed probability to drive an arithmetic encoder.

When the final pixel has been processed, the arithmetic code is terminated. Fig. 2.6 shows the templates for the context calculation for INTRA and INTER modes.

Gray Scale Shape Coding

The gray scale shape coding has a structure similar to that of binary shape with the difference that each pixel can take on a range of values (usually 0 to 255) representing the degree of the transparency of that pixel. The pixel value 0 corresponds to a completely

Table 2.1: List of BAB Types (from [5])

BAB Types	Semantic	Used in
0	MVDs==0 and No Update	P-, B-, and S(GMC)-VOPs
1	MVDs!=0 and No Update	P-, B-, and S(GMC)-VOPs
2	Transparent	All VOP Types
3	Opaque	All VOP Types
4	IntraCAE	All VOP Types
5	MVDs==0 and InterCAE	P-, B-, and S(GMC)-VOPs
6	MVDs!=0 and InterCAE	P-, B-, and S(GMC)-VOPs

Note: GMC = Global Motion Compensation.

transparent pixel and 255 to a completely opaque pixel. Intermediate values of the pixel correspond to intermediate degrees of transparencies of that pixel.

2.2.2 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. Furthermore, in order to perform the motion prediction for VOP of arbitrary shape, a special padding technique is required for the reference VOP before motion estimation.

Padding Process

The padding process defines the values of luminance and chrominance samples outside the VOP for prediction of arbitrarily shaped objects. Fig. 2.7 shows a simplified diagram of this process.

A decoded MB $d[y][x]$ is padded by referring to the corresponding decoded shape block $s[y][x]$. An MB that lies on the VOP boundary is padded by replicating the boundary samples of the VOP towards the exterior. This process is divided into horizontal repetitive padding and vertical repetitive padding. The remaining MBs that are completely outside

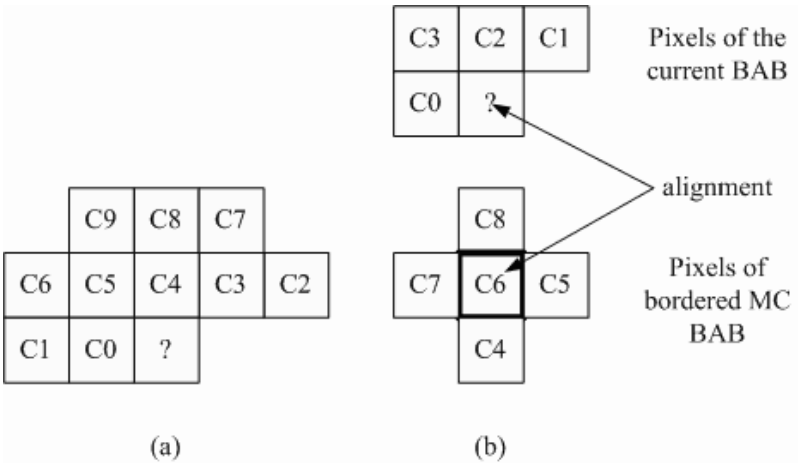


Figure 2.6: Pixel templates used for (a) INTRA and (b) INTER context calculation of BAB. The current pixel to be coded is marked with "?" (from [5]).

the VOP are filled by extended padding.

- Horizontal repetitive padding: Each sample at the boundary of a VOP is replicated horizontally to the left and/or right direction in order to fill the transparent region outside the VOP of a boundary block. If there are two boundary sample values for filling, the two sample values are averaged.
- Vertical repetitive padding: The remaining unfilled transparent region from above procedure are padded by similar process as the horizontal repetitive padding but in the vertical direction. After horizontal and vertical repetitive padding, the boundary MBs have been completely padded.
- Extended padding: Exterior MBs immediately next to boundary MBs are filled by replicating the samples at the border of the boundary MBs. If an exterior MBs is next to more than one boundary MBs, one of the MBs is chosen, according to the priority shown in Fig. 2.8. The remaining exterior MBs (not located next to any boundary MBs) are filled with 128.

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)

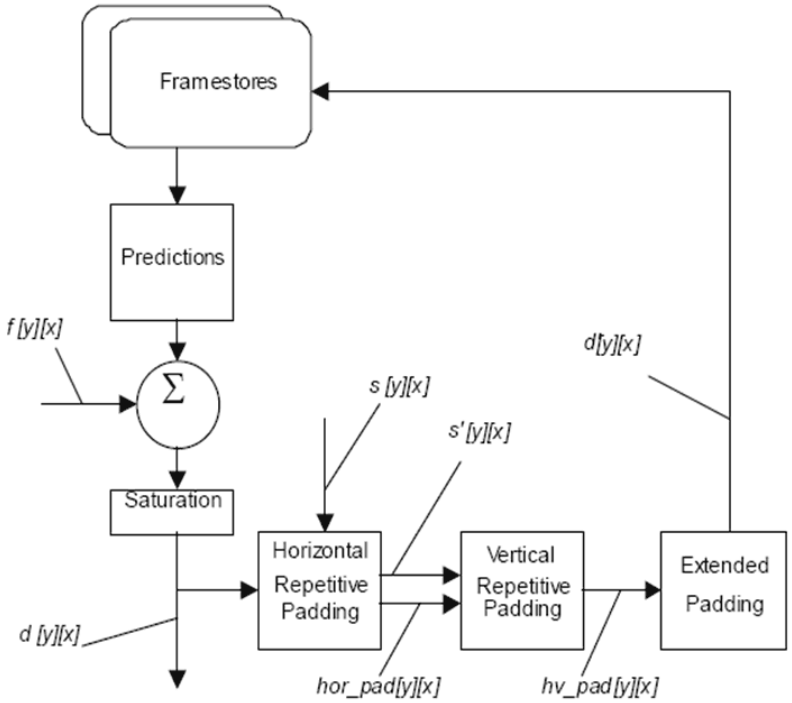


Figure 2.7: Simplified padding process (from [5]).

matching to handle arbitrary-shaped VOPs which is block-based method.

For an arbitrary shape VOP, the bounding rectangle of the VOP is first extended to the right-bottom side to multiples of MB size. The alpha value of the extended pixels is set to zero. The SAD is used for error measure, and is computed only for the pixels with nonzero alpha values.

The basic motion estimation may be performed on 16×16 luminance MBs. The motion vector is specified to half-pixel accuracy. 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 prediction errors more.

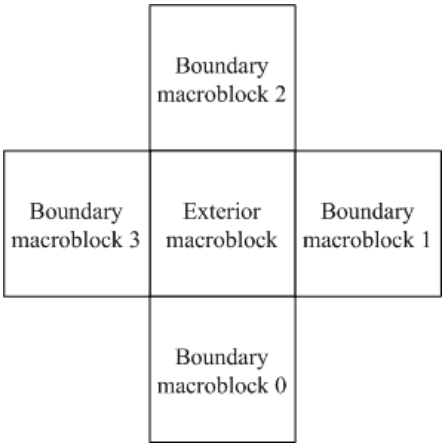


Figure 2.8: Priority of boundary MBs surrounding an exterior MB(from [5]).

Motion Vector Encoder

The motion vector must be coded when using INTER mode coding. Horizontal and vertical motion vectors are coded differentially by using a spatial neighborhood of three motion vectors that have already been coded, as illustrated in Fig. 2.9. These three motion vectors are candidate predictors for 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

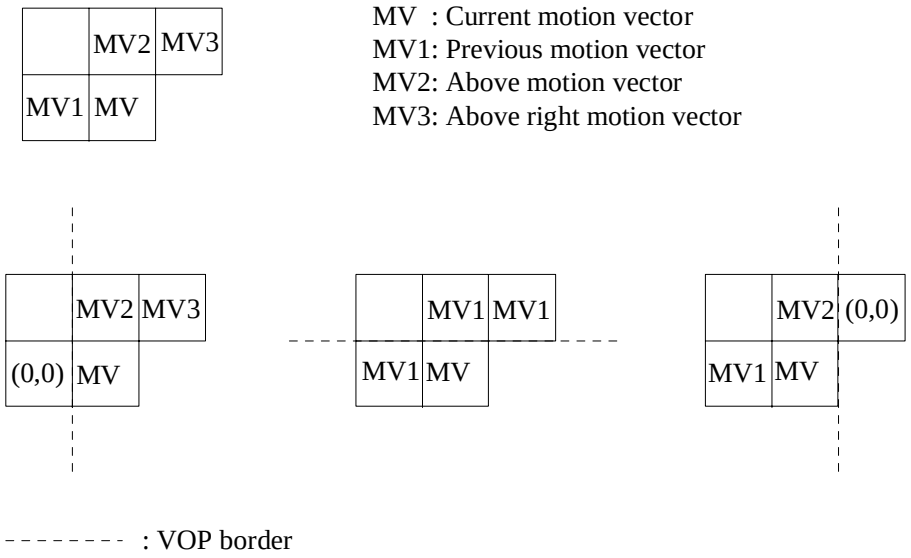


Figure 2.9: Motion vector prediction (from [9]).

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 alternatives are supported, namely, unrestricted motion compensation, four MV motion compensation and overlapped motion compensation.

For unrestricted motion compensation, the motion vectors are allowed to point outside the decoded area of a reference VOP. 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.

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 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 $H_2(i,j)$ are the weighting values of each pixel in the current block and neighbor blocks. The values of $H_0(i,j)$, $H_1(i,j)$, and $H_2(i,j)$ are shown in Table 2.2.

Since the VOP may be coded in P or B mode, there are three types of motion prediction, namely 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

Table 2.2: 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

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.

2.2.3 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 represents directly the values of the luminance and chrominance components. In the case of motion compensated VOPs the encoded texture information represents the residual values 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) 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 2.10 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 2.3, which can be changed if desired.

Furthermore, in order to provide a higher coding efficiency, a nonlinear scaler as shown in Table 2.4 is used for the DC coefficient of 8×8 block in MEPG-4 video. Note that the characteristics of nonlinear scaling are different between the luminance and chrominance blocks and 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 (DC and AC prediction). Intra prediction is a new operation used in MPEG-4 standards to reduce the spatial redundancy between 8×8 blocks.

Figure 2.11 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 method of 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 quantized DC (QDC) of block C is used as the prediction, else the QDC value of block A is used.

The AC prediction depends on DC prediction, as shown in Fig. 2.12. The AC coefficients 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.

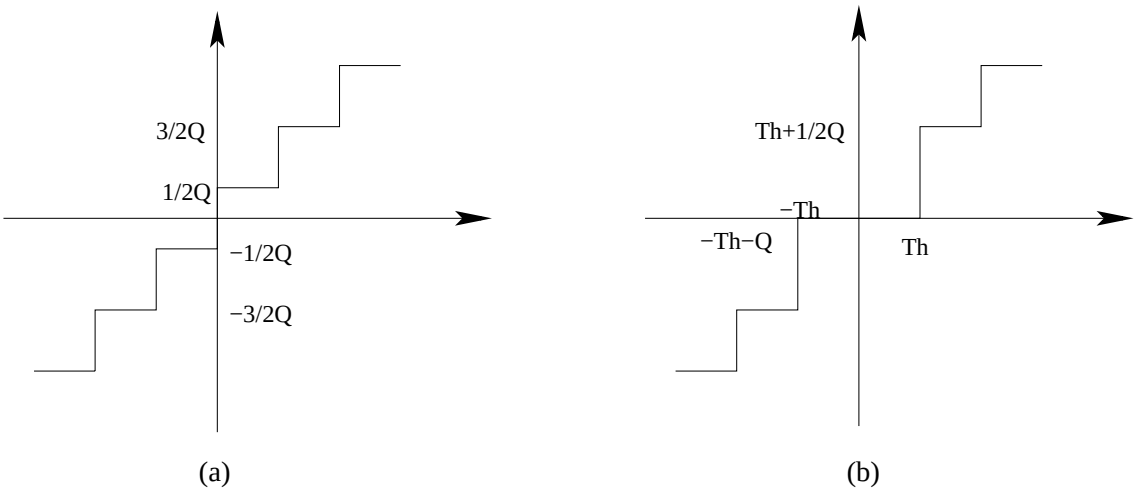


Figure 2.10: Quantizers in H.263. (a) For intra DC coefficient only. (b) For inter DC and all AC coefficients.

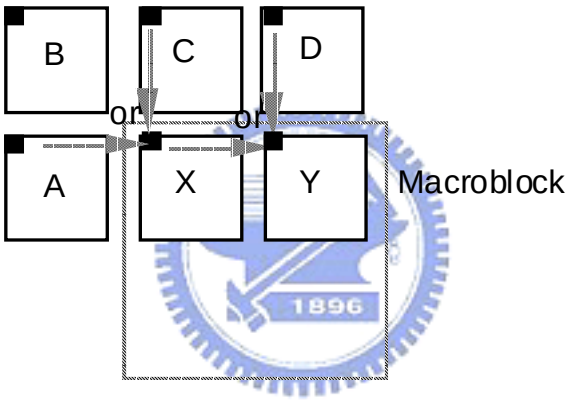


Figure 2.11: Prediction of DC coefficients of blocks in an intra MB (from [7]).

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 ways: 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. 2.13. The actual scan used depends on the coefficient prediction method used.

The coefficients after scan usually become data with many zeros at the end. This kind of 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 VLCs

Table 2.3: Default Quantization Matrix (Q) [5]

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 2.4: Nonlinear Scaler for DC Coefficients (from [5])

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

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.

2.2.4 Other Video Coding Tools [7]

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.

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:

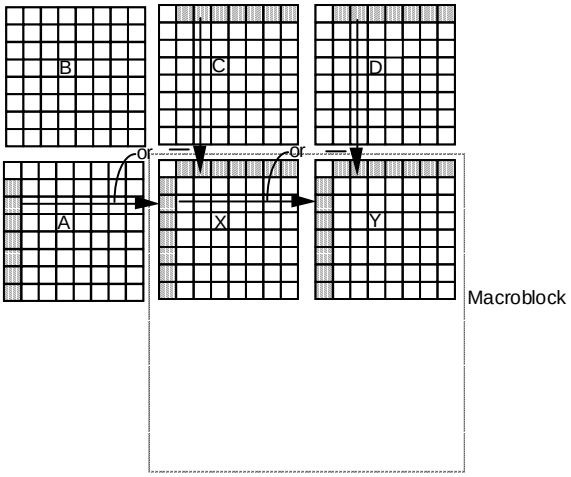


Figure 2.12: Prediction of AC coefficients of blocks in an intra MB (from [7]).

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 2.13: Scans for 8×8 blocks (from [5]).

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 bitstream 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 directions.

5. Intra update and scalable coding: To prevent error propagation, intra update is a simple method to solve the problem. However, intra coding will reduce the coding efficiency. Another method is scalable coding, which can prevent error propagation without more intra coding.

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.

Several different forms of scalability are provided in MPEG-4 video. Temporal and spatial scalability are the most basic scalability tools among them. A Fine Granularity Scalability (FGS) is also defined which supports continuous scalability of bit rate and video quality.

2.3 Profiles and Levels [5]

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 2.5.

Compared with 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.

Table 2.5: Profiles and Tools (from [5])

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 PACDSP

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

We consider implementation of MPEG-4 object-based video decoder on the PACDSP version 2.0. We focus on introducing it in this chapter. In the last section, we give a brief introduction to version 3.0, which is the latest version of the PACDSP.

3.1 Introduction

For high performance, the PACDSP is a VLIW processor with single instruction multiple data (SIMD) instruction set architecture (ISA). The software supported reducing hardware design complexity and 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 of the register files but also the power consumption of read/write operations. 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 register file 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 power consumption.
- Inter-cluster communication by memory controller for reusing hardware resource and reducing the port number of ping-pong register file 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 supported 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.

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.

3.2 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.

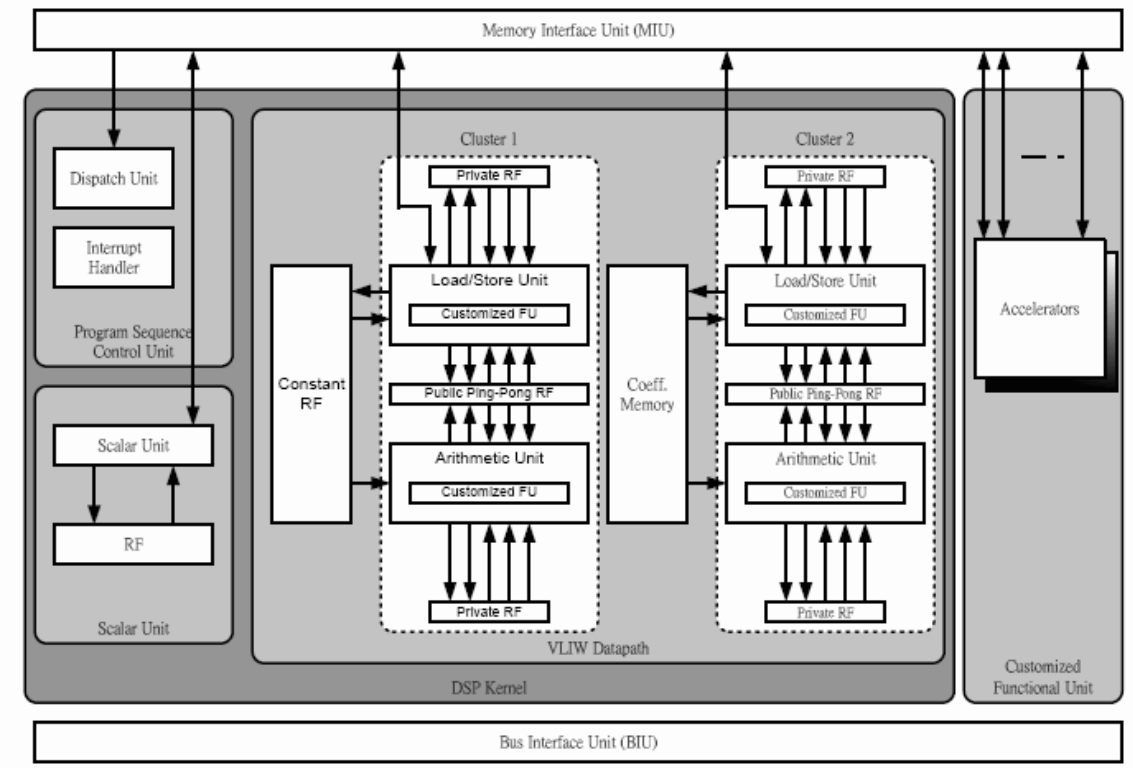


Figure 3.1: Architecture of the PACDSP (from [1]).

3.2.1 Branch Instructions

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:

- Program counter (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 defined in branch instruction.

- 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 independent instructions can be put in these delay slots.

3.2.2 Loop

The programmer can use the LBCB instruction to effect program loops. Loop Boundary Registers (RBC0 – RBC3), which are all 32-bit registers, can be used to record the loop counts. However, the maximum loop count is 65,536 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.

A constraint exists in using LBCB to control a nested loop, that is, 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.

3.2.3 Customized Function Units

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.

3.2.4 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). It is still aware of debug requests and will check the corresponding signal to see what kind of exceptions have happened.

3.2.5 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. In contrast, 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.

3.3 VLIW Datapath

The VLIW datapath is composed of two clusters which 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.

3.3.1 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 3.2 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].

3.3.2 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 3.3 shows how double and vector load/store work.

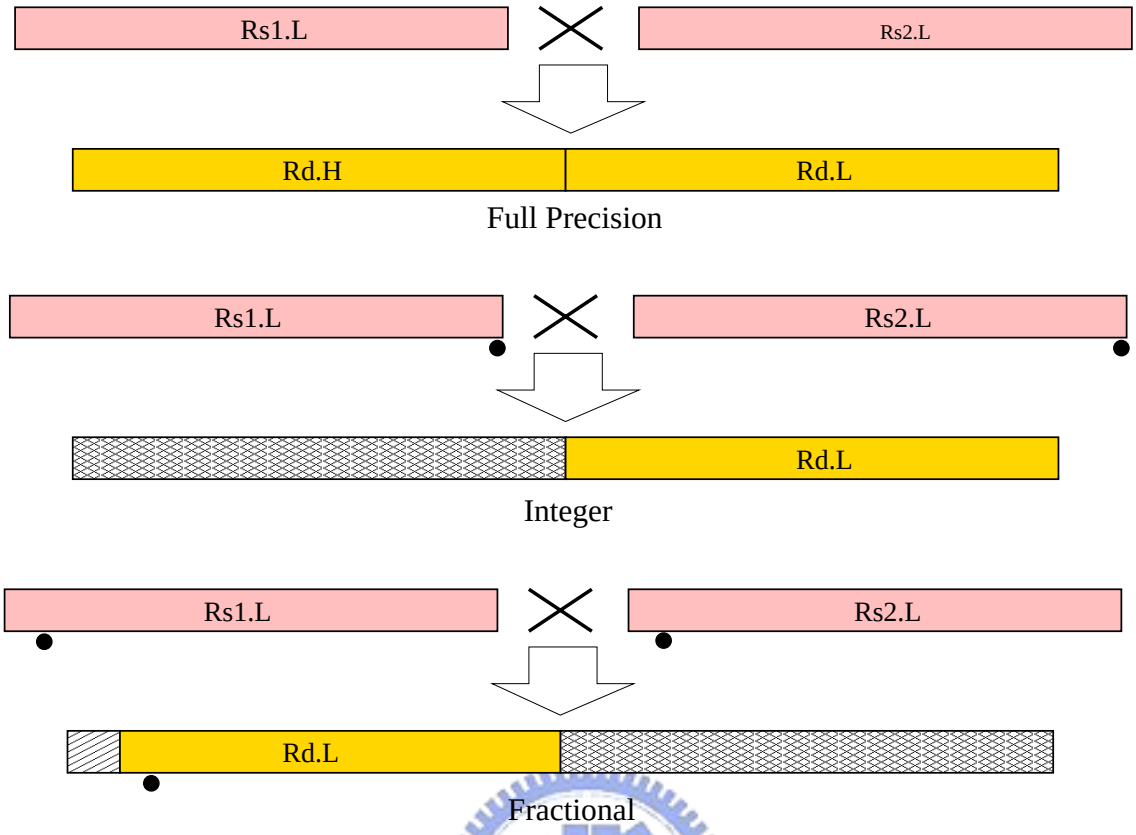


Figure 3.2: Illustration of multiplication instructions with different precisions (from [1]).

3.3.3 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. 3.4, 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 overhead, where the dynamic mapping is exposed to the VLIW ISA with two switching bits and is directly specified by the programmers for each instruction

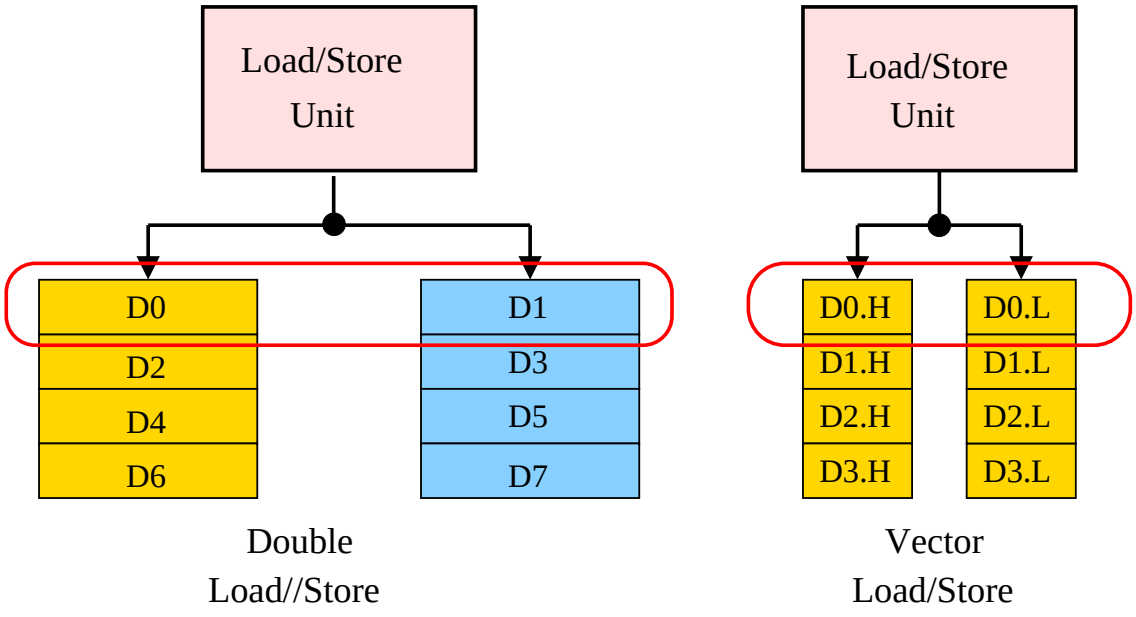


Figure 3.3: Different load/store instructions (from [1]).

packet.

3.3.4 Data/Address/Accumulator Registers

As shown in Fig. 3.5, 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. 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 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 ping-pong with 1-bit control and the word-length of these registers is 32.

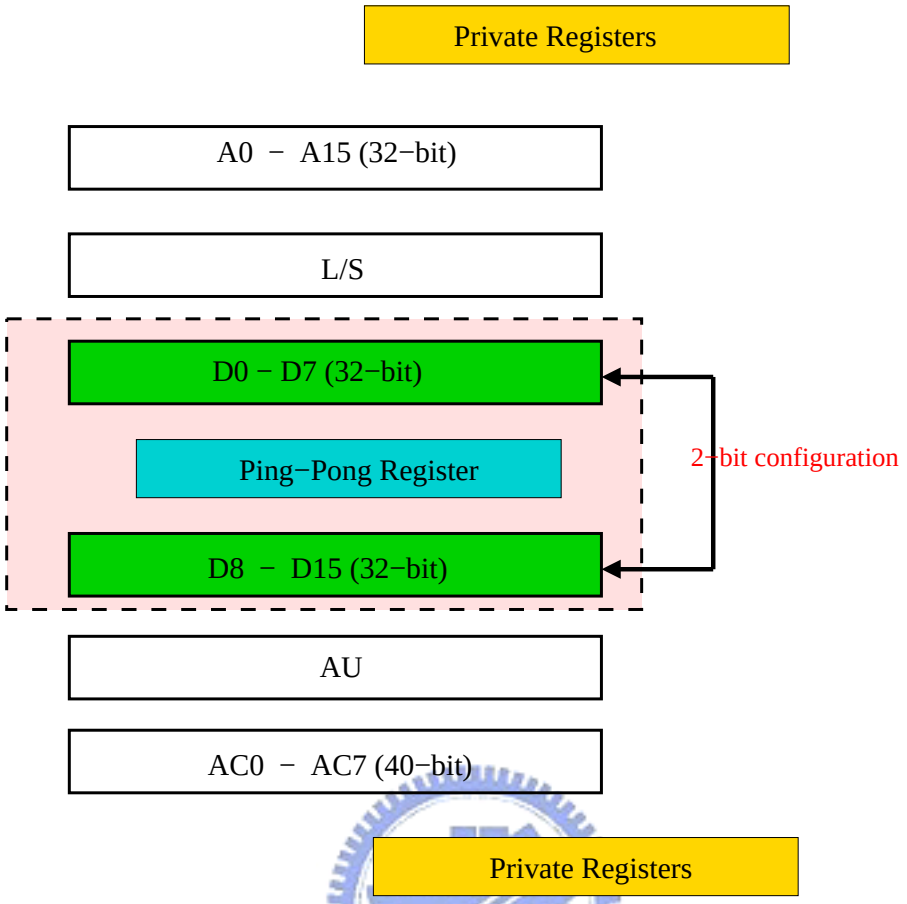


Figure 3.4: Ping-pong register file in one cluster (from [1]).

3.3.5 Status and Control Registers

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

Program Status Register (PSR)

The 16-bit program status register records the operation status in each cluster and the scalar unit. It includes Overflow, Negative, and Carry bits. It can only be read by user instructions.

Addressing Mode Control Register (AMCR)

The PACDSP provides three types of addressing modes:

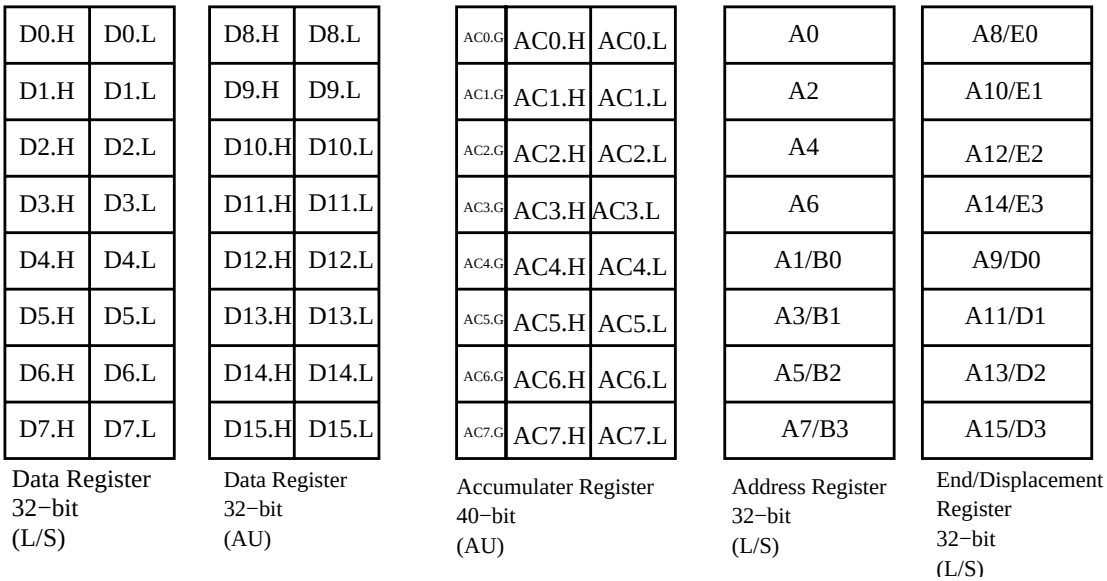


Figure 3.5: Available registers in one cluster (from [1]).

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

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.

3.3.6 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. These are briefly explained below.

1. Register direct mode: This 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. It is also used to specify a control register operand and a PC register operand for special instructions.

2. Address register indirect mode: This 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.

3. Immediate data mode: This 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 fast fourier transform (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 twiddle 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. Offset and modifier registers are also associated with the corresponding address registers in the same way.

3.3.7 Data Exchange

As shown in Fig. 3.6, the PACDSP provides a data exchange mechanism between any two of the scalar unit and the two clusters. Figure 3.7 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 in the future. 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:

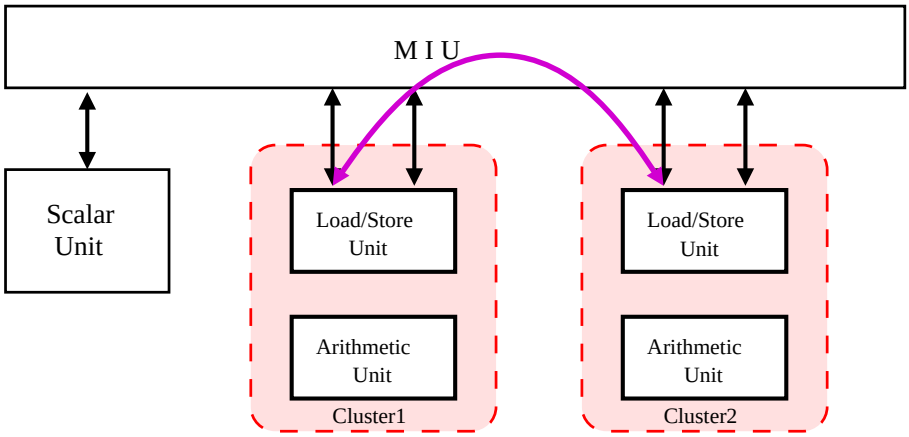


Figure 3.6: Data exchange between two clusters (from [1]).

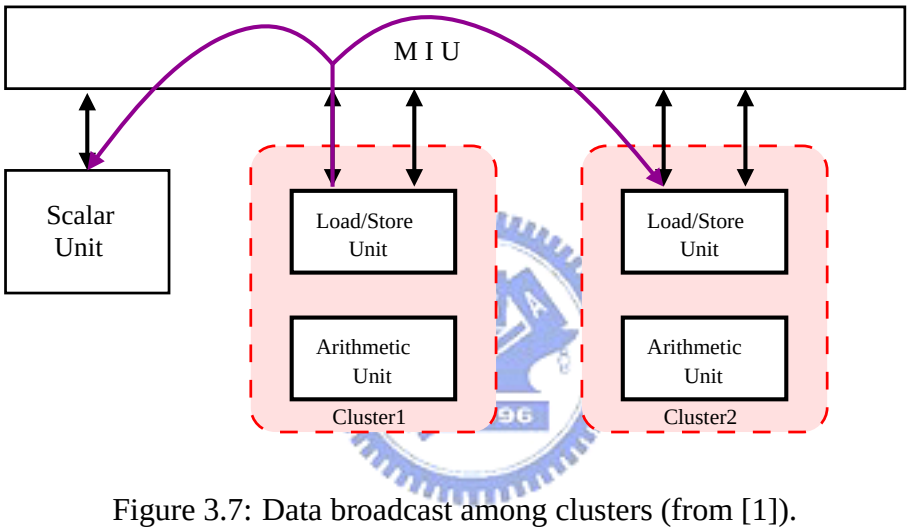


Figure 3.7: Data broadcast among clusters (from [1]).

Cluster1 instruction: DEX D1, D0

Cluster2 instruction: DEX D1, D2

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. 3.6.

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. 3.7.

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, cluster1 can do other operations without interference with this transmission.

3.3.8 Constant Register File

In many DSP algorithms, such digital filtering, there are many fixed data such as the filter coefficient. In order to avoid high frequency of data movement in the register file, the PACDSP provides a small 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 in comparisons, multiplications, multiplications and accumulations, 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. 3.8, 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.

3.4 Scalar Unit

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.

3.4.1 Scalar Unit

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 control/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.

3.4.2 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.

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.

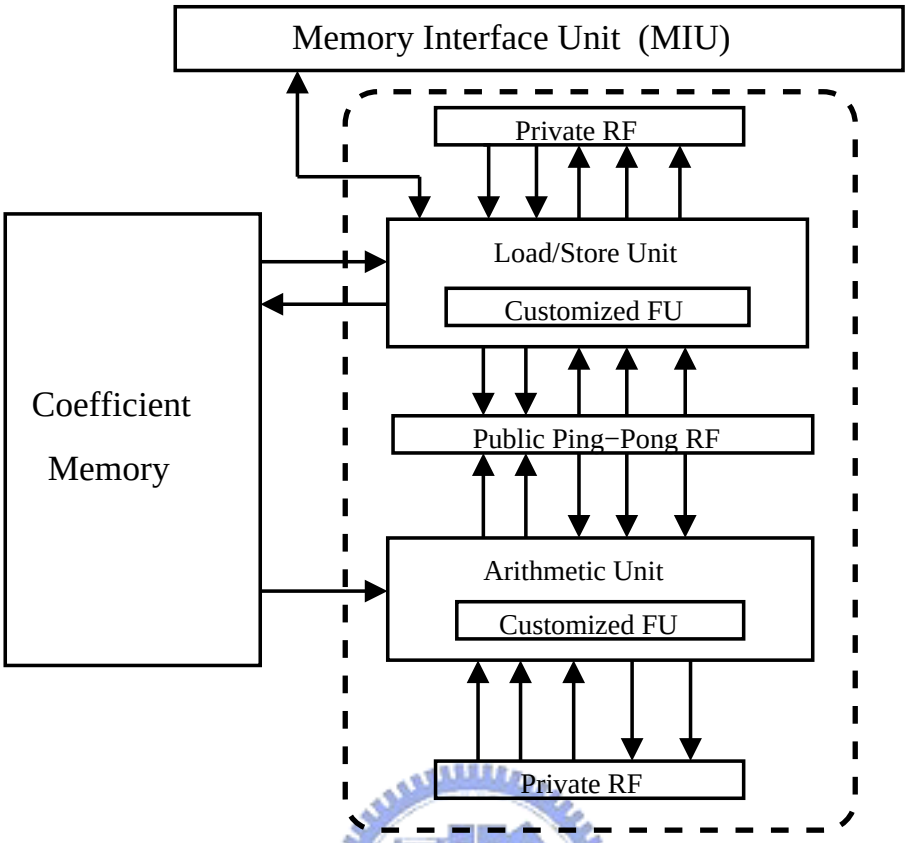


Figure 3.8: The Constant Register File of one cluster (from [1]).

3.4.3 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.

3.5 Conditional Execution Control

Unlike general purpose processors, the major mission of a DSP is to provide more computing power for numerical 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 3.1: Details of Control Register Files (from [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		

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.

Table 3.2: Memory-Mapped Control Registers (from [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

3.6 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 3.9 depicts the instruction set architecture (ISA) of the PACDSP.

Figure 3.10 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 described in Table 3.3.

3.7 DSP Running Modes

The PACDSP can work under several 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.

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. 3.11.

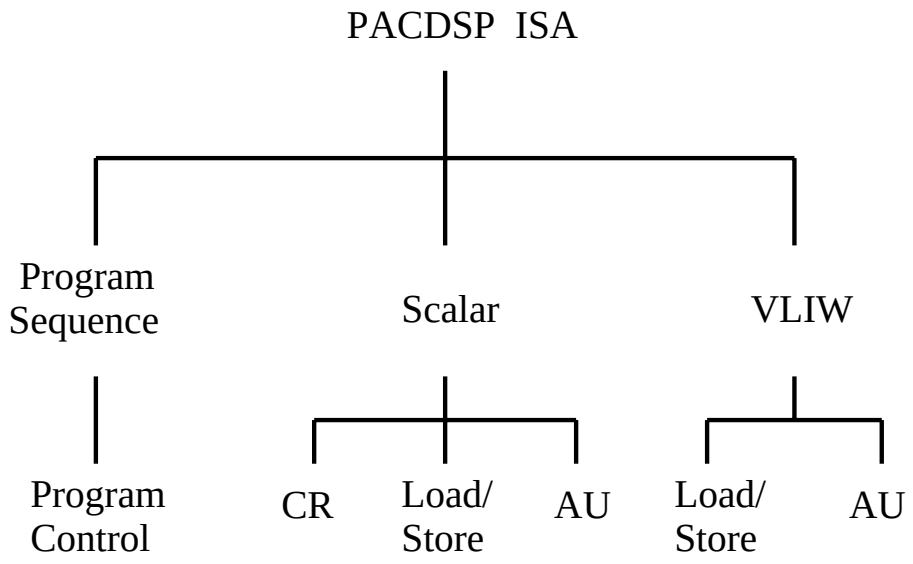


Figure 3.9: PACDSP instruction set architecture (from [1]).

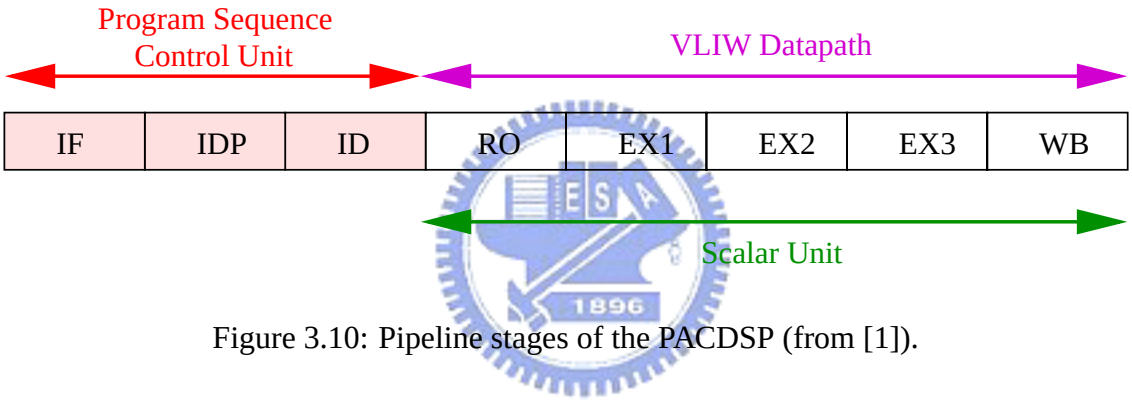


Figure 3.10: Pipeline stages of the PACDSP (from [1]).

3.8 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 braces, and slots within packet are separated by new-line characters. 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. 3.12. A NOP instruction should be placed in a slot where there is no instruction to be executed.

Table 3.3: Pipeline Stages and Their Descriptions (from [1])

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 Development Tools and Implementation Approach

3.9.1 Development Tools

At the present time, 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. However, we can choose only one optimization level currently. In addition, base utilities have been ported from the GNU binutils, and there is an assembler, a linker, and some other object handling tools. The debugger is ported from the GNU GDB (the GNU project debugger). The debugger can be connected to both the instruction set simulator (ISS) and the embedded ICE. These tool chains are developed by the Programming Language Laboratory of the Computer Science Department of National Tsing Hua University, Hsinchu, Taiwan, R. O. C.

The ISS is developed by SoC Technology Center (STC) of the Industrial Technology Research Institute, Hsinchu, Taiwan, R. O. C. The input file of the simulator is split through a parsing tool, “as2tic”, which parses the assembly code into the two parts of data and instructions. 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 Kbytes only. The ISS can be used on Linux operating systems only.

Table 3.4: Running Modes of the PACDSP (from [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

3.9.2 Implementation approach

Since the goal of our implementation is achieve a real-time MPEG-4 video decoder on PACDSP, the execution time and the code size are the most important issues. At the present time, the compiler cannot provide the performance of well-scheduled hand code. Moreover, the development of the compiler was not completed when we began our implementation. Therefore, our implementation employs assembly programming and optimization.

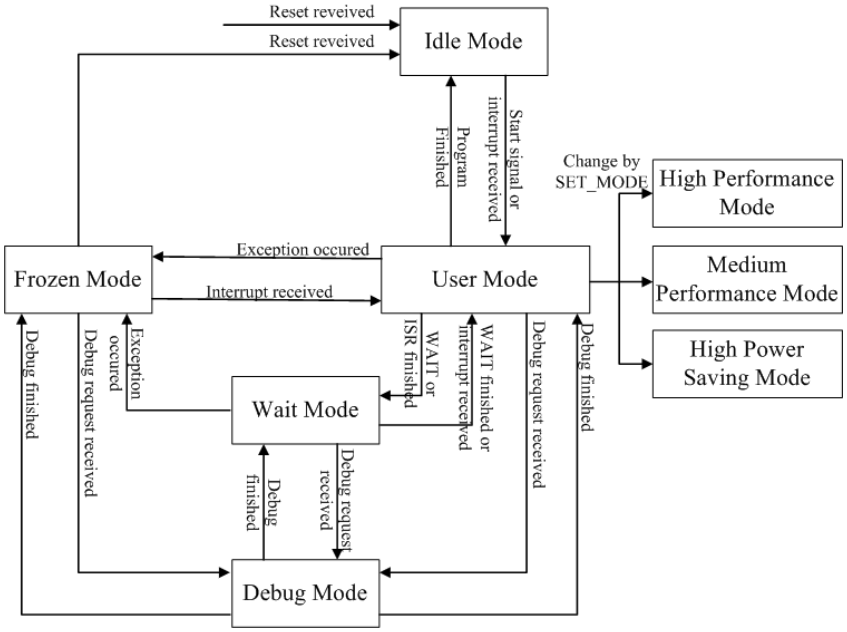


Figure 3.11: Transitions between DSP running modes (from [1]).

Table 3.5: Instruction Types in Each Instruction Slot (from [1])

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

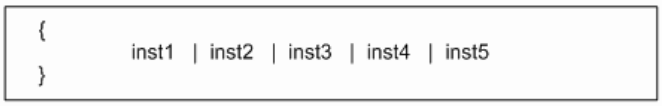


Figure 3.12: Simplified syntax of instruction packet (from [1]).

3.10 Overview of the PSDK 2.0 Platform

The PAC System Developer's Kit (PSDK) platform is developed by SoC Technology Center (STC) of Industrial Technology Research Institute (ITRI) in Taiwan. We demonstrate the implementation on it, which is a dual core system. It 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)

The PSDK 2.0 hardware modules are shown in Fig. 3.13. Since the PACDSP core module is replaced by an FPGA with the DSP design burned-in, the operating frequency of PACDSP is at most 22 MHz rather than a 200 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. 3.14, and it is noted that the start address of instruction is configurable and we set the instruction memory at 0xb0000000.

3.11 Overview of PACDSP v3.0

The contents of this section have been taken to a large extent from [2]–[4].

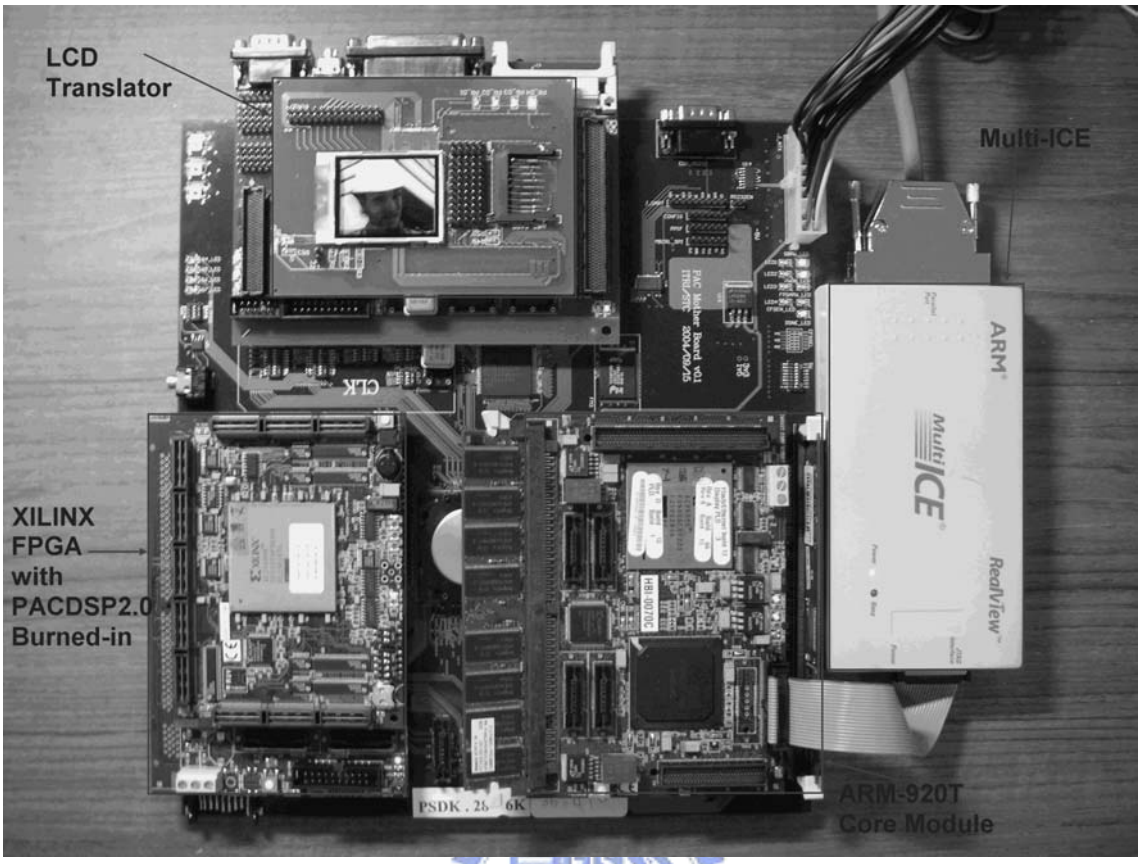


Figure 3.13: PAC System Developer's Kit (PSDK) 2.0.

In this section, we give a brief introduction to the PACDSP v3.0 which is the latest version of PACDSP. We focus our discussion on the difference between the v2.0 and the v3.0. Although our implementation is based on PACDSP v2.0, the information about the difference between the two versions can help us know the design trend of the PACDSP. It also can help us if we do implementation on PACDSP v3.0 in the future.

3.11.1 Architecture Overview

PACDSP v3.0 is also a VLIW DSP processor. The key features of the v3.0 are the same as the v2.0, which are already listed in section 3.1.1. Fig. 3.15 shows the architecture of PACDSP v3.0. Similar to v2.0, the core elements include the Program Sequence Control Unit (PSCU), Scalar Unit, Clusters (VLIW datapath), Customized Function Unit, and memory interface. The following briefly introduces the differences between v2.0 and

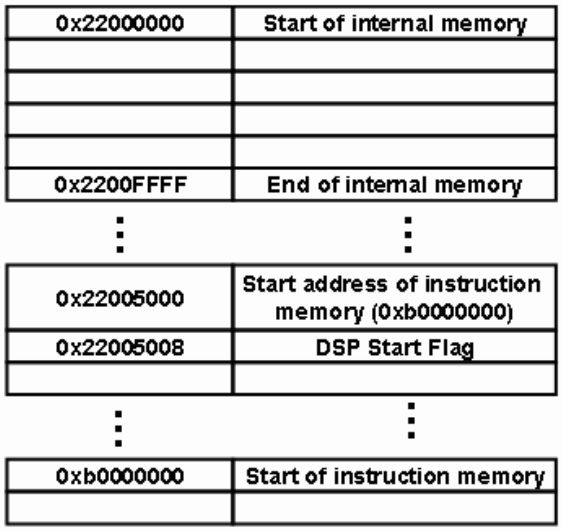


Figure 3.14: Memory map of the dualcore demonstration

v3.0 in PSCU, VLIW datapath, and the pipeline stages. Moreover, a comparison of some instructions that we frequently use between the two versions is given.

3.11.2 Program Control Sequence Unit (PSCU)

- There are five branch delay slots in PACDSP v3.0, compared to three in PACDSP v2.0.

- In the PACDSP v2.0, up to four levels of nested loop are supported with the use of the LBCB instruction, and the loop boundary registers (RBC0–RBC3) are used to record the loop counts. Instead of loop boundary registers, PACDSP v3.0 uses the general purpose registers (R0–R15) to record the loop counts. Up to sixteen levels of nested loop can be supported with the LBCB instruction in v3.0.

- Compared to PACDSP v2.0, PACDSP v3.0 has simplified scenarios of interrupt, debug, and exception. FIQ and IRQ are two types of interrupt supported by the PACDSP. In PACDSP v2.0, the minimum latency from interrupt request to the first ISR instruction to be executed is 3 cycles for both types of interrupt. The minimum latency is 4 cycles in PACDSP v3.0, however.

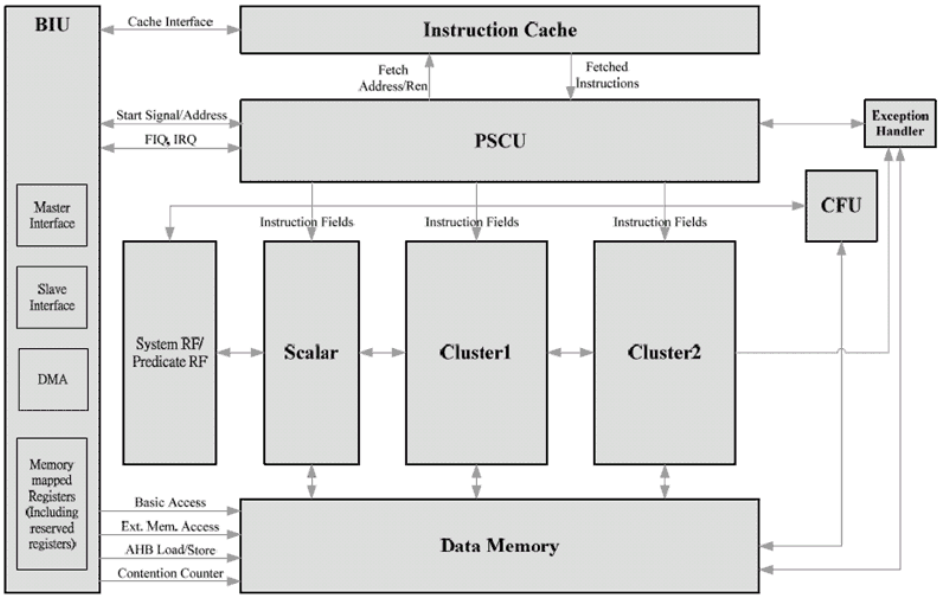


Figure 3.15: Architecture of PACDSP v3.0 (from [2]).

3.11.3 VLIW Datapath

- In PACDSP v2.0, the comparison instructions can only be executed in the L/S unit. In PACDSP v3.0, the comparison instructions can be executed in both L/S unit and arithmetic unit.
- The inter-cluster communication latency is 2 cycles for PACDSP v2.0. PACDSP v3.0 decrease the latency to 1 cycle.
- PACDSP v3.0 adds register relative addressing mode for L/S instructions.
- The addressing mode control register (AMCR) is a 32-bit register in PACDSP v2.0. In PACDSP v3.0, the AMCR is modified to a 16-bit register.
- In PACDSP v2.0, the constant register file in each clusters contains sixteen 32-bits registers (C0–C15), while PACDSP v3.0 only has eight (C0–C7).
- For the constant register file, additional pointer addressing mode is supported in PACDSP v3.0.

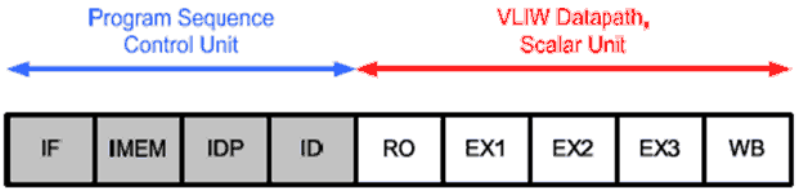


Figure 3.16: Pipeline stages of the PACDSP v3.0 (from [4]).

3.11.4 Pipeline Stages

Fig. 3.16 shows the pipeline stages of PACDSP v3.0. Compare to v2.0, PACDSP v3.0 divides the PSCU into four stages,which are IF, IMEM, IDP, and ID. The added stage, IMEM, accesses the instruction memory after the IF stage. The scalar unit operation and the VLIW datapath are both divided into five stages.

3.11.5 Instruction Set Comparison

Compared to PACDSP v2.0, PACDSP v3.0 adds some useful instructions and has enhanced some common by used instructions. Table 3.6 shows the modification of Load/Store instructions from PACDSP v2.0 to PACDSP v3.0 and their supporting units. Table 3.7 lists the comparison instructions supported in PACDSP v2.0 and PACDSP v3.0.

Table 3.6: Modification of Load/Store Instructions from PACDSP v2.0 to PACDSP v3.0

PACDSP v2.0			PACDSP v3.0			
Instruction	Scalar Unit	L/S Unit	Instruction	Scalar Unit	L/S Unit	Note
(D)LW	V	V	(D)LW	V	V	LW only in scalar unit
LNW		V	(D)LNW		V	
(D)LH(U)		V	LH(U)	V	V	
LB(U)		V	LB(U)	V	V	
(D)SW	V	V	(D)SW	V	V	SW only in scalar unit
without this instruction			(D)SNW		V	
(D)SH(U)		V	SH(U)	V	V	
(D)SB(U)		V	SB(U)	V	V	

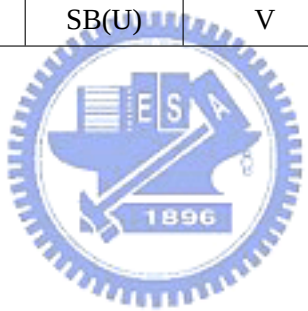


Table 3.7: Comparison Instructions Supported in PACDSP v2.0 and PACDSP v3.0

Category	PACDSP v2.0	PACDSP v3.0
Set Less Than	SLT(U) SLTI	SLT(U)[.L/.H] SLTI(U)
Set Greater Than	SGT(U) SGTI	SGT(U)[.L/.H] SGTI(U)
Set Equal	SEQ	SEQ[.L/.H] SEQI(U)

Chapter 4

Complexity Analysis of MPEG-4 Object-Based Video Decoder and Dual-Core Implementation Design

Prior to DSP implementation, 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 subsequent codec programming on the PAC platform. Fig. 4.1 shows the major blocks of MPEG-4 object-based video decoder, and our analysis will focus on some important blocks shown in this figure.

After the complexity analysis, we discuss the implementation of the IDCT of the MPEG-4 video decoder, which is a important function that consumes time. We address the efficiency and the accuracy of our algorithm, and then we show the performance of IDCT that is implemented on PACDSP. Finally, we show the design of the dual-core implementation on PSDK, and the optimization of the implementation on the ARM processor. We leave the optimization of implementation on PACDSP to the next chapter.

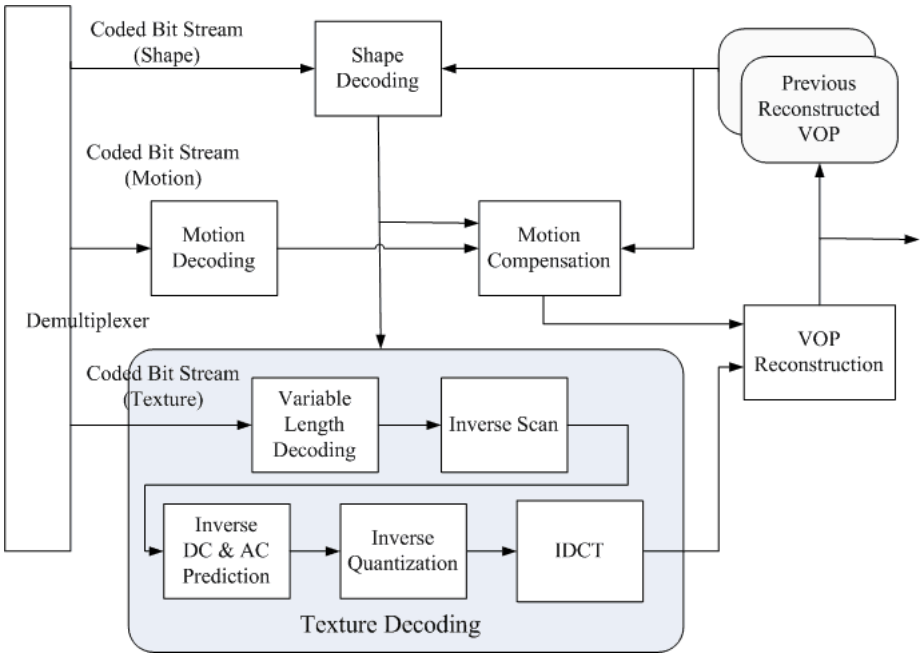


Figure 4.1: Block diagram of MPEG-4 object-based video decoder [5].

4.1 Profiles of the MPEG-4 Object-Based Video Decoder

In this section, we analyze the complexity of the MPEG-4 object-based video decoder. We focus on the execution time that the codec software spends in coding of practical video sequences. For this, we employ the MoMuSys [10] software as the base. There are three different sequence for our analysis, which are “stefan”, “foreman” and “akiyo”. Table 4.1 shows the VOP size of each sequence, which contains the width, height and the total number of pixels. We also show the first frame of each sequence in Fig. 4.2.

In the original codec of MoMuSys, the IDCT is implemented in floating-point, and it consumes much time. In order to reduce the complexity and to implement it on DSP, we modify the IDCT to fixed-point, which is discussed in the next section. After fixed-point IDCT, we will do the complexity analysis of execution time again to find out the amount of improvement.

The computational environment for the complexity analysis is a PC with a 2.0 GHz Pentium-M CPU and 768 MB of DDR RAM, running Windows-XP. We use Intel’s “VTune

Table 4.1: VOP Size of Each Test Sequence

Test Seq. (QCIF)	width (pixels)	height (pixels)	Total Num. of pixels
stefan	48	96	4,608
foreman	112	144	16,128
akiyo	144	128	18,432

Figure 4.2: First frame of each test sequence (a) stefan. (b) foreman. (c) akiyo.

Performance Analyzer" to run the profile of the MoMuSys software. The profiling result, shown in Table 4.2, is obtained from decoding 2 frames including one intra frame and one inter frame. And the encoder employs H.263 quantization with a fixed quantization step size (QP) of 4. Noted that QP affects the length of the bitstream, so a larger QP results in a smaller bitstream size and reduces the required encoding and decoding time. However, a large QP will reduce the quality of the output image.

In Table 4.2, "DecodeFirst" and "AlphaDecodeMB" are two key functions in shape decoding. "DecodeFirst" decodes the BAB type and "AlphaDecodeMB" decodes the alpha plane using context-based arithmetic coding according to the BAB type. Since we decoded one intra frame and one inter frame for the analysis, several functions are used in both I and P frames. In order to distinguish these functions, functions used in I frame are marked with underline I (_I), and functions used in P frame are marked with underline P (_P). However, certain functions, such as "VOPMotionCompensate" and "DecodeMB-

Table 4.2: Profile of Object-Based MPEG-4 Decoding of QCIF Sequence on VTune

	stefan_qcif		foreman_qcif		akiyo_qcif	
Function Name	Clockticks	%	Clockticks	%	Clockticks	%
DecodeVOLHeader	746	1.87	679	0.98	774	1.19
DecodeVOPHeader	337	0.84	320	0.46	316	0.48
VOPPadding	3,785	9.46	9,300	13.38	9,405	14.40
DecodeFirst_I	28	0.07	109	0.16	162	0.25
AlphaDecodeMB_I	1,727	4.32	4,430	6.37	4,786	7.33
DecodeMBHeader_I	18	0.05	70	0.10	76	0.12
VlcGetBlock_I	1,043	2.61	1,353	1.95	2,484	3.80
doDCACrecon_I	106	0.27	588	0.85	476	0.73
BlockIDCT_I	870	2.18	3,320	4.78	2,931	4.49
BlockDequantH263_I	128	0.32	367	0.53	408	0.62
DecodeFirst_P	37	0.09	123	0.18	106	0.16
AlphaDecodeMB_P	1,941	4.85	4,121	5.93	2,412	3.69
DecodeMBHeader_P	29	0.07	89	0.13	122	0.19
VlcGetBlock_P	623	1.56	383	0.55	4	0.01
BlockIDCT_P	1,067	2.67	3,534	5.09	602	0.92
BlockDequantH263_P	103	0.26	262	0.38	48	0.07
VOPMotionCompensate	1,457	3.64	4,275	6.15	4,104	6.29
DecodeMBMVs	90	0.23	233	0.34	36	0.06
WriteOutImage	15,940	39.85	15,588	22.43	15,921	24.38
Others	9,921	24.79	20,350	29.26	20,124	30.82
Total	39,996	100.00	69,494	100.00	65,297	100.00

MVs", are used in inter (P) frames only, and "doDCACrecon" is only called for intra (I) frames for our test sequences. Certain functions, like "DecodeVOLHeader", "DecodeVOPHeader" and "WriteOutImage", have regular operations that are almost independent of the test sequences. Hence we do not separate them for I frames and P frames, although they are called by both. Therefore, the execution time of these functions should be divided by two if we want to compare the computational complexity between them.

In the object-based video decoder, the VOP size is arbitrary in each frame. In our test sequences, "akiyo_qcif" has the biggest VOP size, "foreman_qcif" the next, and the VOP size of "stefan_qcif" is the smallest. The execution times of some functions, such as "VOPPadding" and those called for I frame decoding, are directly proportional to the VOP size. Hence they are the longest for "akiyo". However, for the functions called for P frames, not only the VOP size but also the sequence characteristics may affect the execution time. Take "akiyo_qcif" for example, though its VOP size is the biggest, because the motion in this sequence is very little, the execution time of the inter functions are less than "foreman_qcif", even less than "stefan_qcif" in some functions.

Though the test sequences have different VOP sizes and motion characteristics, we still can find in Table 4.2 that IDCT and shape decoding are very important parts in the decoding procedure, in the sense that they are time-consuming. We should pay more attention to these blocks. In the next section, we first discuss our study of fixed-point IDCT, and show the improvement of our optimization. The optimization of shape decoding will be left to the next chapter.

4.2 Fixed-Point IDCT

The DCT and IDCT in MPEG-4 are defined as

$$F(u,v) = \frac{2}{N} C(u)C(v) \sum_{x=0}^{N-1} \sum_{y=0}^{N-1} f(x,y) \cos\frac{(2x+1)u\pi}{2N} \cos\frac{(2y+1)v\pi}{2N}, \tag{4.1}$$

$$f(x,y) = \frac{2}{N} \sum_{u=0}^{N-1} \sum_{v=0}^{N-1} C(u)C(v)F(u,v) \cos\frac{(2x+1)u\pi}{2N} \cos\frac{(2y+1)v\pi}{2N}, \tag{4.2}$$

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.

4.2.1 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 [15]. One fast method reduces the required 1-D IDCTs from 16 to 8 [15]. However, since the number of required registers is very big in this algorithm, it is not appropriate for implementation on PACDSP. Similar to the derivation from discrete Fourier transform (DFT) to fast Fourier transform (FFT), a fast cosine transform (FCT) is proposed in [16]. A comparison of computational complexity of different algorithms is listed in Table 4.3.

Note that the computational complexity is estimated for floating-point computation. Since the transform coefficients used in [16] 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 algorithm. As a result, we first consider the IDCT algorithm of MoMuSys on PACDSP.

Table 4.3: Comparison of Computational Complexity for 8-point IDCT

	Direct Form	FCT [16]	MoMuSys	Even_Odd FCT [17]
Multiplications	64	12	16	20
Additions	56	29	26	28

4.2.2 Accuracy of IDCT

Since the PACDSP is not capable of floating-point computations, we have to convert the IDCT algorithm to fixed-point computation. In this, the accuracy is a critical issue. Since the native word length is 16-bit on PACDSP, we scale the floating-point cosine coefficients with 2^{15}. We then right shift 15 bits after multiplications, which rounds the product to the nearest integer.

The 1-D IDCT algorithm used in MoMuSys has the signal flow shown in Fig. 4.3. We need to check if the implementation is accurate enough. Some tests for the IDCT accuracy are defined in MPEG-4 [5], which are based on the IEEE Std. 1180-1190 with some modifications. The tests require five statistical measurements, which are as follows:

- For any pixel location, the peak error (ppe) shall not exceed 2 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.

The testing result of MoMuSys algorithm is shown in Table 4.4. We can see that this implementation is not accurate enough. It is because the simple rounding method introduces significant errors. Moreover, we see that the odd-indexed coefficients are rounded twice in this algorithm, yielding more serious rounding errors. Therefore, we try to use the even-odd decomposition algorithm [17] whose signal flow is shown in Fig. 4.4. In this algorithm, each coefficient is rounded once, which can reduce the rounding error. Moreover, we use the following rounding rules to improve the accuracy.

- Keep the shift as late as possible just enough to prevent the overflow.
- Minimize the bits shifted, just enough to prevent overflow.
- Minimize the number of shifts.

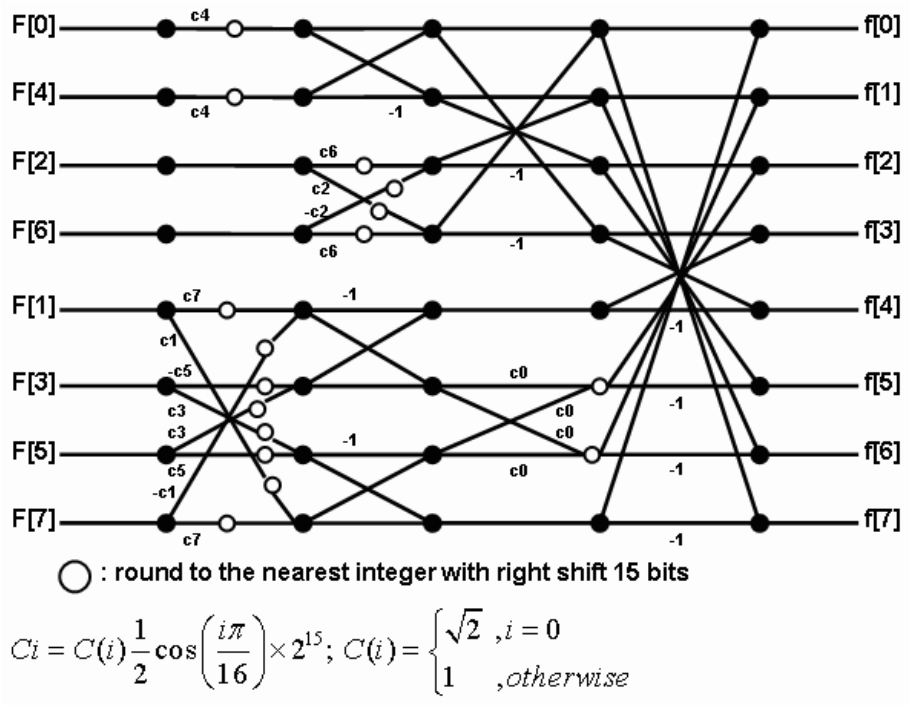


Figure 4.3: The IDCT algorithm used in MoMuSys [10].

Following the above rules, the rounding operations are postponed to the output stage and we can reduce the number of roundings. After the calculation of each row IDCT, we only do right shift of 11 bits for rounding to maximize the accuracy, so we need to do 19 bits of right shift after each column IDCT to keep the correct format. The accuracy testing result of our algorithm is also shown in Table 4.4. We can see that our fixed-point IDCT has enough accuracy to pass the test. Then we show the profiling of the software codec on PC which uses the fixed-point IDCT algorithm, and discuss the implement and optimization on PACDSP in the following sections.

4.2.3 Profile on PC with Fixed-Point IDCT

Table 4.5 shows the execution time comparison between the floating-point IDCT and the fixed-point IDCT obtained by profiling of the software codec. The clockticks gives the total execution time of the IDCT in decoding one intra frame and one inter frame. The percentage figures give the proportion of the clocktickes the IDCT consumes in the whole decoding procedure. We see that the execution time and the percentage used in IDCT

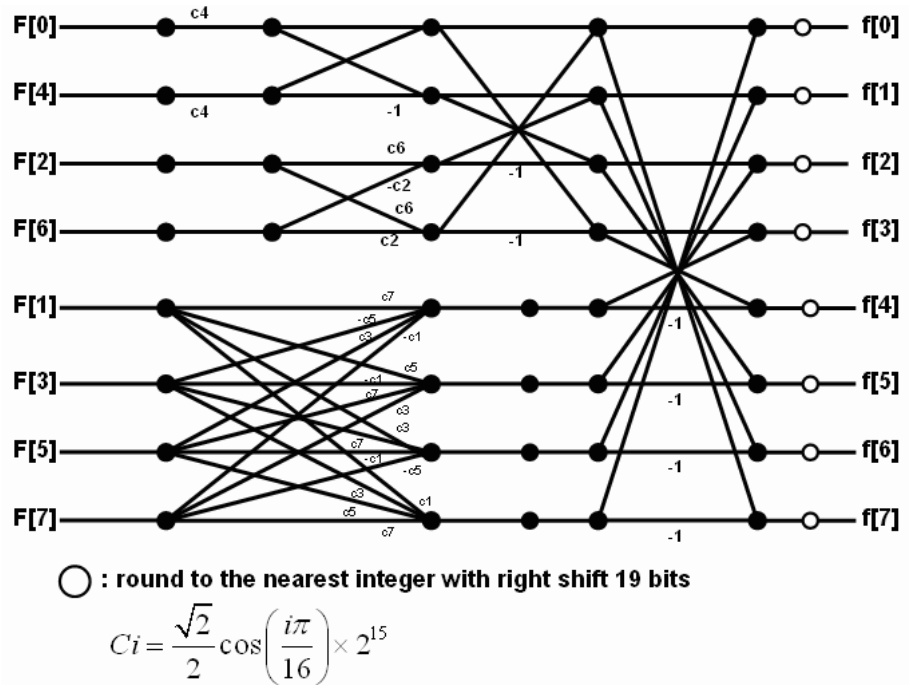


Figure 4.4: The even-odd decomposition IDCT algorithm [12].

both decrease much after our optimization. The implementation of fixed-point IDCT on PACDSP is left the next chapter.

4.3 Implementation of Decoder on Dual-Core PSDK

We now focus on our design of the MPEG-4 object-based video decoder for the dual-core system, where “dual-core” refers to ARM core and PACDSP core, especially for the P frame decoding.

Figure 4.5 shows a simple outline of the P frame decoding procedure. We first decode the shape and texture information, which includes the motion vectors and the prediction residuals. Then the padding procedure is executed on the reference frame (I frame in this figure) before the motion compensation. At last, we use the padded frame and the motion vectors to reconstruct the output frame and blend it according to the shape information.

In the decoding procedure, the padding process is independent of the bitstream, which is executed only on the reference frame. We assign the padding process to the ARM core. Then we can decode the bitstream information of the current frame with the DSP core at

Table 4.4: Test of Compliance for Modified IEEE Std. 1180-1190 in MPEG-4

Item	Modified IEEE 1180–1190	MoMuSys	Our Algorithm
ppe	$\leq$2	$>$2 (X)	$\leq$2 (◯)
$pmse$	$\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	◯	◯

Table 4.5: Execution Time Comparison of IDCT

Test Sequences	Original (Floating-Point)		Optimized (Fixed-Point)	
(QCIF)	Clockticks	%	Clockticks	%
stefan	1,937	4.85	559	1.47
foreman	6,854	9.87	1,976	3.07
akiyo	3,533	5.41	1,214	1.94

the same time. Moreover, we also use the ARM core to do the motion compensation (MC) and blending functions. Then we can use the DSP to decode the bitstream information of the next frame, when we do the MC and blending functions of the current frame. Fig. 4.6 illustrates the design.

Table 4.6 shows the total execution time on ARM core and PACDSP core, respectively. According to our design, the major functions on ARM core include three parts, which are "VOPPadding", "Motion Compensation", and "BlendVOP", whose execution times are listed in Table 4.6. The table shows that the execution time ratio between the two cores are nearly equal, except for the sequence "akiyo". For "akiyo", because of its stationary characteristic, the decoding work on PACDSP takes relatively little execution time due to a large percentage of zero motion vectors and residuals. However, the decoding work on

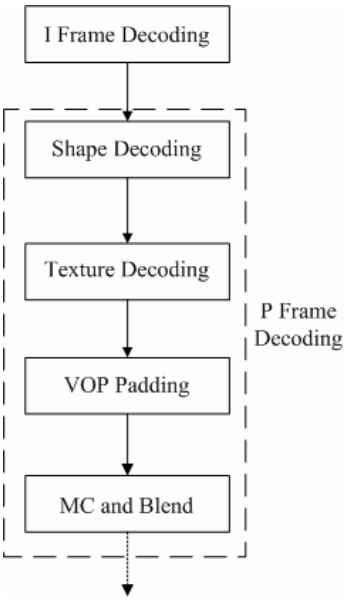


Figure 4.5: An outline of P frame decoding procedure.

the ARM, which does padding and MC, is independent on the VOP size rather than the characteristic of the sequence. Since the VOP size of "akiyo" is the biggest of three, it takes a significant greater execution time on the ARM core than the other two.

4.4 Optimization of Implementation on ARM

Before we discuss the optimization of implementation on PACDSP in the next chapter, we first discuss the optimization on the ARM core, which is focused on efficient motion compensation.

In the MPEG-4 object-based video decoder, the reference VOP needs to be interpolated before motion compensation when there are fractional motion vectors. In the MoMuSys reference software, three directional interpolations are executed for the whole VOP regardless whether the motion vectors are fractional or not. However, if the horizontal and vertical motion vectors are both integers, the interpolation is useless. Moreover, we need a large memory space for the interpolation results in this way. We decode twenty frames of each sequence and count the number of total motion vectors and fractional motion vectors, which is shown in Table 4.7. In the table "Both" means that both the

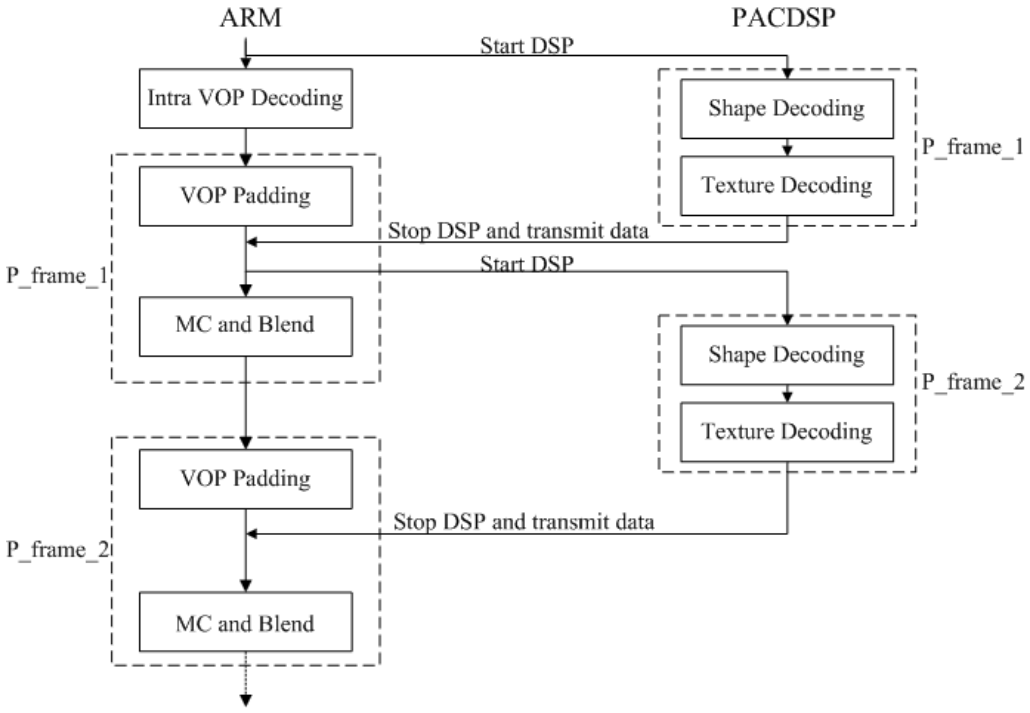


Figure 4.6: The dual-core P-frame decoding.

horizontal and the vertical motion vectors are fractional, “Hor.” and “Ver.” mean that the motion vector is fractional only in the horizontal and the vertical directions, respectively. In our test sequences, “stefan” has the largest percentage of fractional motion vectors. In addition, the “akiyo” has the least number of fractional motion vectors, although it has the largest VOP size.

In our implementation, we only do necessary interpolations for each block according to the type of motion vector. If the horizontal and vertical motion vectors are both integers, no interpolation will be executed. Compared with MoMuSys, our implementation only needs a memory space for storing the result of interpolation for each block. By removing the unnecessary operations, the execution time also decreases. Table 4.8 shows the result. Note that the proportion of fractional motion vectors and the VOP size will affect the amount of decrease time.

Table 4.6: Execution Time Analysis Between ARM and PACDSP

Test Sequences (QCIF)		Execution Time (Clockticks)		
		stefan	foreman	akiyo
ARM	VOPPadding	1,893	4,650	4,703
	Motion Compensation	1,457	4,275	4,104
	BlendVOP	585	1,207	1,492
	Total	3,935	10,132	10,299
PACDSP		4,529	9,961	3,851
ARM:PACDSP		0.869:1	1.017:1	2.674:1

Table 4.7: Analysis of Necessary Interpolation Using MoMuSys Encoder

Bitstream	Total MV	Fractional MV							
(QCIF)	Number	Total	%	Both	%	Hor.	%	Ver.	%
stefan	956	786	82.21	313	32.74	218	22.80	255	26.67
foreman	3,712	2,790	75.16	1,202	32.38	913	24.60	675	18.18
akiyo	2,184	383	17.54	106	4.85	48	2.20	229	10.49

Table 4.8: Execution Time of Motion Compensation after Eliminating Unnecessary Interpolations on ARM

Test Seqs.	Execution Time (cycles)			
(QCIF)	Original	Optimized	Decreases	Speedup (%)
stefan	524,725	255,944	268,781	51.22
foreman	1,144,238	516,038	628,200	54.90
akiyo	1,193,582	389,814	803,768	67.34

Chapter 5

Optimization of Implementation on PACDSP

In this chapter, we discuss the optimization of our implementation of the MPEG-4 object-based video decoder on PACDSP. The optimization contains three major parts, efficient implementation strategies, architectural optimization, and algorithmic optimization. At first, we discuss the efficient implementation strategies for several functions which utilize the advantage of PACDSP. The discussion follows the order of decoding procedure, which contains shape decoding and texture decoding. The improvement of each function is also shown.

After the reconstruction of the MPEG-4 object-based video decoder, we use the general architectural optimization in our assembly code to reduce stalls. At the last part, we use the characteristic of DCT to remove the unnecessary computations in the decoding procedure. Because this optimization is focused on algorithm and can be extend to other implementations of video decoder, we classify it as algorithmic optimization. We also show the results of architectural and algorithmic optimization, which focuses on the improvement of the whole decoder. Fig. 5.1 shows the flow of software development on PACDSP in our implementation.

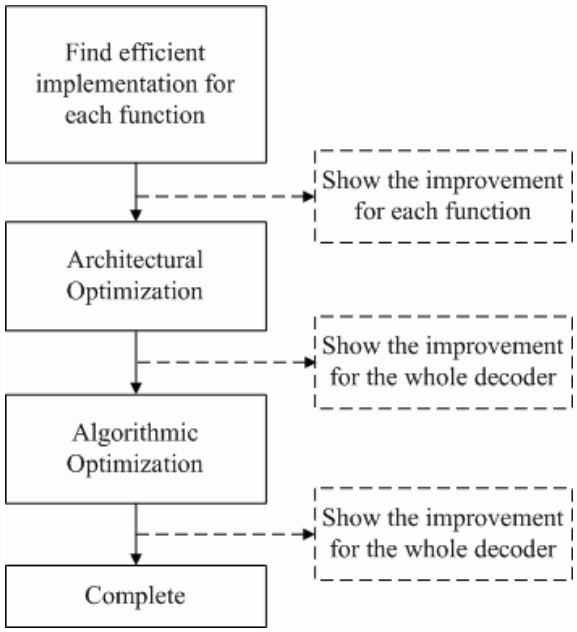


Figure 5.1: Flow of software development on PACDSP.

5.1 Implementation Strategies on PACDSP

In this section, we show several more efficient methods for our implementation on PACDSP. Our discussion can separate to two part, shape decoding and texture decoding. In the shape decoding, we discuss the efficient context calculation and efficient motion compensation in the context-based arithmetic coding. In the texture decoding, we first discuss the implementation of VLD, and then we show our optimization of DC/AC reconstruction. Finally, we shows our result of fixed-point IDCT implemented on PACDSP.

5.1.1 Efficient Context-Based Arithmetic Coding

Binary shape decoding is based on a block-based representation. The primary coding methods are block-based context-based arithmetic decoding and block-based motion compensation. From the profiling analysis, we know that the shape coding is a time-consuming part of the MPEG-4 object-based video decoder. We need to find out an efficient method to implement shape coding on PACDSP. In the following, we discuss our implementation of the two part, namely context calculation and shape motion compensation.

Fast Context Calculation

In the context-based arithmetic coding, we need to calculate the context to obtain the probability for arithmetic decoding. Therefore, for each 16×16 binary alpha block (BAB), 256 context calculations are needed. Fig. 5.2 shows the templates for intra and inter context calculation, where the current pixel to be coded is marked with "?". Because the large number of context calculations, it should help the performance of shape decoding if we can find an efficient implementation method.

Fig. 5.3 shows the intra context calculation of two successive pixels, which "P" and "C" means the previous and current pixel, respectively. In fact, there are only three new pixels we need to update for the context calculation of the current pixel, as shown with shadowed pixels in Fig. 5.3. Therefore, in our implementation, we store the context of the previous pixel and only load three pixel values to update the context for the current pixel. Fig. 5.4 shows the fast calculation. However, this method cannot be used for a pixel on the left column in each BAB. Table 5.1 shows the result in speed performance on PACDSP.

We also do the optimization for inter context calculation. Recall that there are two clusters on PACDSP that can perform computations simultaneously. We use one cluster to compute the context inex for pixels C0 to C3 and use the other cluster for that for pixels C4 to C8 at the same time. Then we can get the full context value by combining the two. The assembly code for fast inter context calculation is shown in Fig. 5.5. The result of our inter optimization is also shown in Table 5.1.

Table 5.1: Execution Time Comparison of Context Calculation for One BAB on PACDSP

	Original (Cycles)	Optimized (Cycles)	Speed Up (%)
Intra	13,824	7,824	43.40
Inter	10,240	6,144	40.00

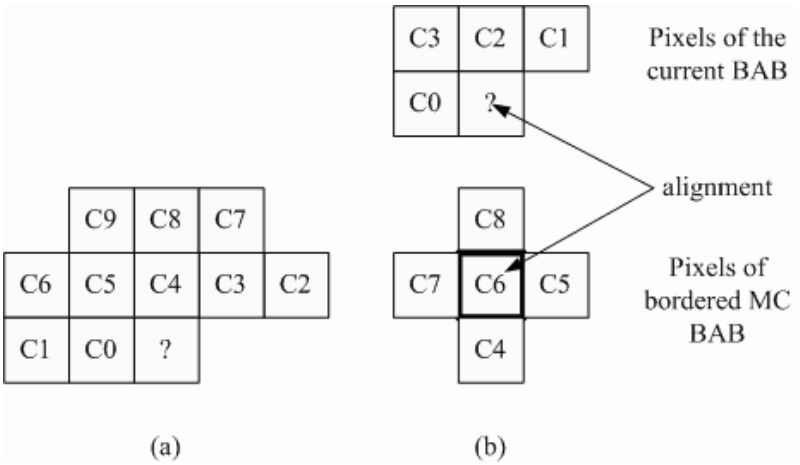


Figure 5.2: Pixel templates used for (a) INTRA and (b) INTER context calculation of BAB (from [5]).

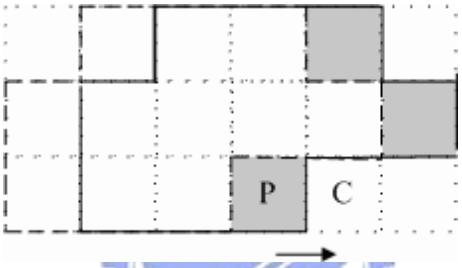


Figure 5.3: Intra context calculation.

Efficient Alpha Plane Motion Compensation

There is some difference of the motion compensation for shape decoding and that for texture decoding. In shape decoding, after getting the reference BAB, there are two types of motion compensation. We may directly use the reference BAB to represent the current by decoding BAB or use the reference BAB for inter context calculation. Whichever the type of motion compensation, we need to get the reference BAB according to the shape motion vector. Therefore, we discuss our optimization for getting the reference BAB.

Because of the need for context calculation, we need to get a 20×20 reference BAB. Pixels of reference BAB that fall outside of VOP are set to zero. In our initial implementation, we determine for each pixel in reference BAB if it is outside of VOP. Then we need 18184 cycles to get one reference BAB, which is inefficient. However, for most cases, the whole reference BAB is inside the VOP. We only need to check if the whole

```
Tmp = Prev_context & 0110111101;
Tmp = Tmp << 1 ;
Current_context = Tmp + new_C0
                      + new_C2 << 2
                      + new_C7 << 7;
```

Figure 5.4: Fast intra context calculation.

```
{ NOP  | ADDI D9,D6,-1    | NOP              | COPY D9,D6       | NOP  }
{ NOP  | ADDI D10,D5,-1   | NOP              | ADDI D10,D5,-1   | NOP  }
{ NOP  | NOP             | FMUL D10,D10,C11 | NOP              | FMUL D10,D10,C11  }
{ NOP  | ADD D10,D10,D9   | NOP              | ADD D10,D10,D9   | NOP  }
{ NOP  | ADD A4,A2,D10    | NOP              | ADD A4,A7,D10    | NOP  }
{ NOP  | LB D11,A4,0      | NOP              | LB D11,A4,0      | NOP  }
{ NOP  | LB D12,A4,1      | NOP              | ADDI A4,A4,19    | NOP  }
{ NOP  | LB D13,A4,2      | NOP              | LB D12,A4,0      | NOP  }
{ NOP  | SEQ D6,C13,p3,p4 | NOP              | LB D13,A4,1      | NOP  }
{ NOP  | SGTI D7,2,p5,p6  | NOP              | LB D14,A4,2      | NOP  }
{ NOP  | ANDP p7,p3,p5    | NOP              | SLLI D15,D11,8   | NOP  }
{ NOP  | ADDI A4,A4,20    | NOP              | SLLI D12,D12,7   | NOP  }
{ NOP  | LB D14,A4,0      | NOP              | ADDI A4,A4,21    | SLLI D13,D13,6  }
{ NOP  | (p7)COPY D13,D12 | NOP              | LB D11,A4,0      | NOP  }
{ NOP  | SLLI D11,D11,3   | NOP              | SLLI D14,D14,5   | NOP  }
{ NOP  | SLLI D12,D12,2   | NOP              | ADD D15,D15,D12  | NOP  }
{ NOP  | SLLI D13,D13,1   | NOP              | ADD D15,D15,D13  | NOP  }
{ NOP  | ADD D11,D11,D12  | NOP              | ADD D15,D15,D14  | NOP  }
{ NOP  | ADD D13,D13,D14  | NOP              | SLLI D11,D11,4   | NOP  }
{ NOP  | ADD D11,D11,D13  | NOP              | ADD D15,D15,D11  | NOP  }
{ NOP  | BDR D15          | NOP              | BDT D15          | NOP  }
{ NOP  | NOP  | NOP  | NOP  | NOP  }
{ NOP  | NOP  | NOP  | NOP  | NOP  }
{ NOP  | ADD D15,D15,D11  | NOP  | NOP  | NOP  }
```

Figure 5.5: Example assembly code for fast inter context calculation.

reference BAB is inside the VOP or not, instead of checking for each pixel. And when the whole BAB is inside the VOP, because the operation is independent between pixels, we can separate the calculation into the two clusters of the PACDSP, which is shown in Fig. 5.6. Therefore, we only need half the time to complete the operation of getting the reference BAB. Moreover, we can use the successive by changing characteristic of the index to simplify the calculations of coordinates. Fig. 5.7 shows the assembly code for getting the reference BAB. A comparison of the execution time on PACDSP for the case of whole BAB inside VOP is listed in table 5.2. After our optimization, the performance has improved much when the whole reference BAB is inside the VOP.

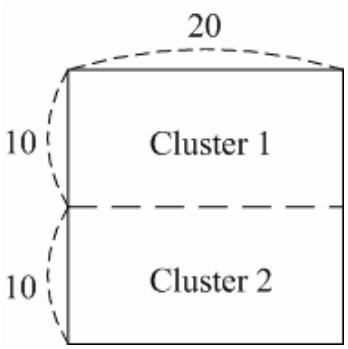


Figure 5.6: Calculation distribution of two clusters on PACDSP.

```
 (NOP   | NOP              | NOP              | ADDI A4,A4,200 | NOP              )
{ NOP   | NOP              | NOP              | ADD D9,D9,D10  | NOP              }
;inside VOP                                   ;Address offset
{ NOP   | NOP              | ADD D10,D9,D14   | NOP            | ADD D10,D9,D14   }
{ NOP   | NOP              | SRLI D11,D10,5   | NOP            | SRLI D11,D10,5   }
{ NOP   | NOP              | AND D12,D10,D15  | NOP            | AND D12,D10,D15  }
{ NOP   | NOP              | SLLI D11,D11,2   | NOP            | SLLI D11,D11,2   }
{ NOP   | ADD A5,A6,D11    | NOP              | ADD A5,A6,D11  | NOP              }
{ NOP   | LW  D11,A5,0     | NOP              | LW  D11,A5,0   | NOP              }
{ NOP   | NOP              | NOP              | NOP            | NOP              }
{ NOP   | SUB D12,D15,D12  | NOP              | SUB D12,D15,D12| NOP              }
{ NOP   | SRL D11,D11,D12  | NOP              | SRL D11,D11,D12| NOP              }
{ NOP   | AND D11,D11,C1   | NOP              | AND D11,D11,C1 | NOP              }
```

Figure 5.7: Example assembly code for getting reference BAB on PACDSP.

5.1.2 Efficient Variable Length Decoding (VLD)

In this subsection, we discuss a efficient method of VLD which uses the advantage of PACDSP. In additions, we also compare the performance of different VLD methods on PACDSP. The methods are proposed in [12] and [13]. We use the simple VLC table in Table 5.3 for the following comparison, which has thirteen entries in this table.

One Table Mapping with Magnitude-Offset

In this technique, we build a table containing all possible codewords. Each entry in the table has two elements, which are the corresponding VLC symbol and its code length.

Table 5.2: Execution Time of Getting One Reference BAB on PACDSP

	Original (Cycles)	Optimized (Cycles)	Speed Up (%)
Whole BAB in VOP	18,184	4,325	76.22

Table 5.3: Variable Length Codes for dct_dc_size_luminance [5]

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

Thus, because the maximum code length is 11 bits in this example, there would be 2^{11} entries in the table. We fetch the first 11 bits in the bitstream, whose magnitude gives the index the corresponding entry 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. 5.8.

Bit by Bit Matching

If the size of VLC 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 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

```
{ NOP   | MOVI.L D2,11   | 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   };D7 is code
{ 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 5.8: Example of one table mapping with magnitude-offset on PACDSP.

```
{ J Show_Bitstream,R1   | MOVI.L D2,2   | NOP   | NOP   | NOP   }
{ NOP   | MOVI.H D2,0   | NOP   | NOP   | NOP   }
{ NOP   | NOP   | NOP   | NOP   | NOP   }
{ NOP   | NOP   | NOP   | NOP   | NOP   }
{ NOP   | SEQ D7,C2,p4,p5   | NOP   | NOP   | NOP   }
{ NOP   | SEQ D7,C3,p8,p9   | NOP   | NOP   | NOP   }
{ NOP   | NOP   | NOP   | NOP   | NOP   }
{ (p4)B Get_DCT_DC_Delta   | (p4)MOVI.L D5,2   | NOP   | NOP   | NOP   }
{ NOP   | (p4)MOVI.H D5,0   | NOP   | NOP   | NOP   }
{ NOP   | (p8)MOVI.L D5,1   | NOP   | NOP   | NOP   }
{ NOP   | (p8)MOVI.H D5,0   | NOP   | NOP   | NOP   }
{ (p8)B Get_DCT_DC_Delta   | NOP   | NOP   | NOP   | NOP   }
{ NOP   | NOP   | NOP   | NOP   | NOP   }
{ NOP   | NOP   | NOP   | NOP   | NOP   }
{ NOP   | NOP   | NOP   | NOP   | NOP   }
{ J Show_Bitstream,R1   | MOVI.L D2,3   | NOP   | NOP   | NOP   }
{ NOP   | MOVI.H D2,0   | NOP   | NOP   | NOP   }
{ NOP   | NOP   | NOP   | NOP   | NOP   }
{ NOP   | NOP   | NOP   | NOP   | NOP   }
{ NOP   | SEQ D7,C1,p4,p5   | NOP   | NOP   | NOP   }
{ NOP   | SEQ D7,C2,p8,p9   | NOP   | NOP   | NOP   }
{ NOP   | SEQ D7,C3,p10,p11   | NOP   | NOP   | NOP   }
{ (p4)B Get_DCT_DC_Delta   | 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   }
```

Figure 5.9: Example of bit-by-bit matching on PACDSP.

a symbol will be long. The example assembly program of bit by bit matching on the PACDSP is shown in Fig. 5.9.

Multiple-Pass Matching

To reduce the frequency of accessing the bitstream, we may divide the VLC table into several subtables. Since the symbol with shorter code appears more frequently, we can search the subtable with shorter code length first. For example, we may divide the example table into two subtables. The first half with symbols 0–6 are grouped into one subtable and the second half with symbols 7–12 are grouped into the second subtable. In decod-

```
{ J Show_Bitstream,R1 | MOVI.L D2,5  | NOP  | NOP  | NOP  )
{ NOP  | MOVI.H D2,0  | NOP  | NOP  | NOP  )
{ NOP  | NOP  | NOP  | NOP  | NOP  }
{ NOP  | NOP  | NOP  | NOP  | NOP  };D7 is code
{ NOP  | SEQ D7,C0,p4,p5  | NOP  | NOP  | NOP  )
{ NOP  | SEQ D7,C1,p2,p3  | NOP  | NOP  | NOP  )
{ NOP  | NOP  | NOP  | NOP  | NOP  }
{ NOP  | NOP  | NOP  | NOP  | NOP  }
{ (p4)B Table_2  | NOP  | NOP  | NOP  | NOP  )
{ NOP  | NOP  | NOP  | NOP  | NOP  }
{ NOP  | NOP  | NOP  | NOP  | NOP  }
{ NOP  | NOP  | NOP  | NOP  | NOP  }
{ (p2)B Get_DCT_DC_Delta  | (p2)MOVI.L D2,5  | NOP  | NOP  | NOP  )
{ NOP  | (p2)MOVI.H D2,0  | NOP  | NOP  | NOP  )
{ NOP  | (p2)MOVI.L D15,6  | NOP  | NOP  | NOP  )
{ NOP  | (p2)MOVI.H D15,0  | NOP  | NOP  | NOP  )
{ NOP  | SEQ D7,C2,p2,p3  | NOP  | NOP  | NOP  )
{ NOP  | NOP  | NOP  | NOP  | NOP  }
{ NOP  | NOP  | NOP  | NOP  | NOP  }
{ (p2)B Get_DCT_DC_Delta  | (p2)MOVI.L D2,4  | NOP  | NOP  | NOP  )
{ NOP  | (p2)MOVI.H D2,0  | NOP  | NOP  | NOP  )
{ NOP  | (p2)MOVI.L D15,5  | NOP  | NOP  | NOP  )
{ NOP  | (p2)MOVI.H D15,0  | NOP  | NOP  | NOP  )
```

Figure 5.10: Example of multiple-pass matching on PACDSP.

ing, we read the first five bits in the bitstream and check if any code in the first subtable matches the bits. If not, then we read the next six bits and check the second subtable. The procedure is similar when there are more subtables. The example assembly program of multiple-pass matching on the PACDSP is shown in Fig. 5.10.

Optimized Multiple-Pass Matching

In our implementation, we use an idea similar to multiple-pass matching to realize the VLD on PACDSP. At first, we also divide the VLC table into two subtables in this example. However, without accessing the bitstream twice for the two subtables, we only access the bitstream once. The number of bits that we fetch from the bitstream is the longest code length in the VLC table. Then we can easily get the code from searching the table by shifts. In addition, because the predicate registers (p0–p15) are shared by the two clusters in the PACDSP, we can transmit the code to the other cluster and execute the comparison instruction at the same time. Then we can do the conditional execution according to the contents of the predicate registers. The example assembly program of optimized multiple-pass matching on the PACDSP is shown in Fig. 5.11.

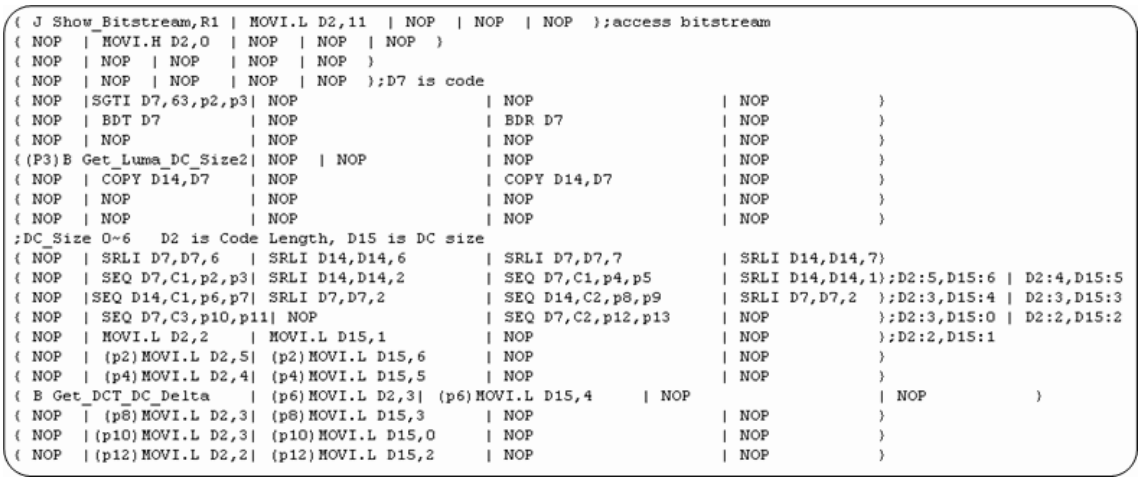


Figure 5.11: Example of optimized multiple-pass matching on PACDSP.

Comparison of Different VLD Methods

We decode a bitstream consisting of all possible symbols on PACDSP, which use the four different methods introduced above. The results are shown in Fig. 5.12 and Table 5.4. In the method "one table mapping with magnitude-offset," we only access the bitstream once and get the output by searching the table. Therefore, the execution time for decoding each symbol is all the same, only 35 cycles. The primary drawback of this method is the memory requirement of the lookup table because of the exponentially increasing table size with maximum code length.

The second method, "bit-by-bit matching," has the best performance for the shortest codeword. However, as the codeword gets longer, it is significant degraded in performance. Therefore, because of the characteristic of entropy coding which uses shorter codes to represent more frequently appearing symbols, the "bit-by-bit matching" method can be used when most symbols may be encoded with shorter codewords.

The third method, "multiple-pass matching," has a similar characteristic, where the performance is also degraded with longer codewords. However, because we only access the bitstream twice for the longest codeword, we need 89 cycles rather than 256 cycles in the worst case.

Finally, in our implementation, we use the advantage of PACDSP to optimize the multiple-pass matching and fetch the bitstream only one time. We see that the perfor-

Table 5.4: Execution Time of Different VLD Methods on PACDSP

Code Pattern	One Table Mapping with Magnitude-Offset	Bit-by-Bit Matching	Multiple-Pass Matching	Optimized Multiple-Pass Matching
10	35	27	34	38
11	35	31	41	38
001	35	54	48	38
010	35	58	55	38
011	35	62	62	38
0001	35	85	69	38
0000 1	35	108	75	38
0000 01	35	131	54	37
0000 001	35	154	61	37
0000 0001	35	177	68	37
0000 0000 1	35	210	75	37
0000 0000 01	35	233	82	37
0000 0000 001	35	256	89	37

mance of our implementation is very close to “one table mapping with magnitude-offset.” Moreover, there is no memory requirement for building a table in our implementation. Therefore, this method provides a good tradeoff between memory requirement and execution time.

5.1.3 Efficient AC/DC Reconstruction

There are two types of prediction, DC prediction and AC prediction, used in intra encoding of MPEG-4 video to reduce the spatial redundancy in texture coding, as shown in Fig. 5.13. In the MoMuSys reference, it uses much memory space for the prediction

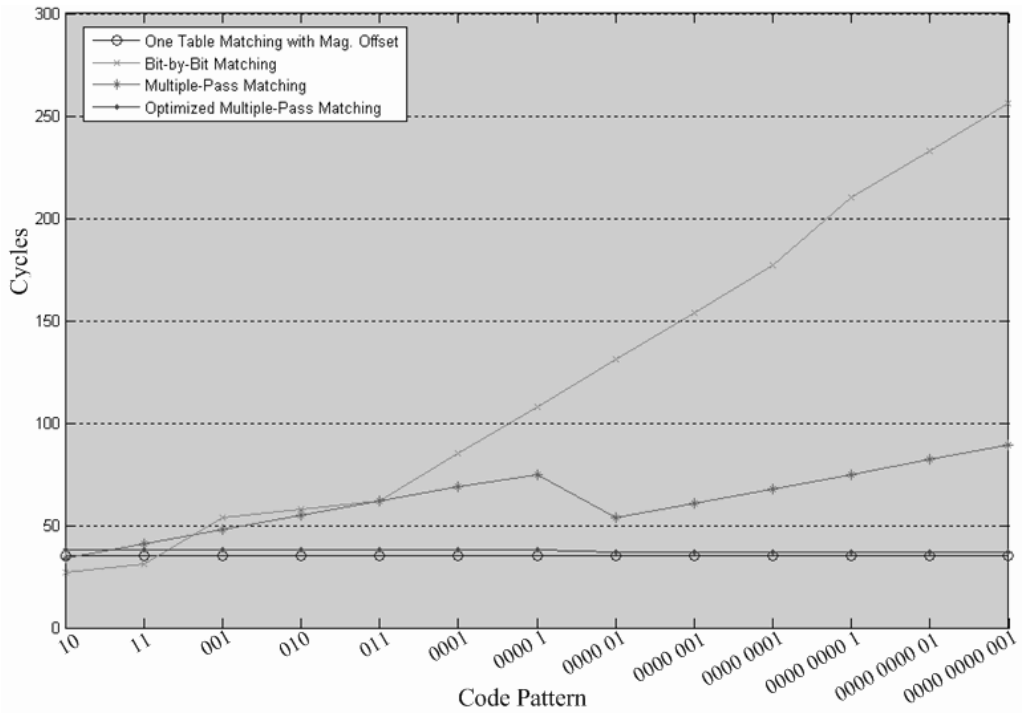


Figure 5.12: Comparison of different VLD methods on PACDSP.

operation. Our method can reduce the usage of memory substantially. We also modify the code flow for efficiency on PACDSP.

Memory Usage Reduction

There are a total of six blocks in one MB, which are four luminance blocks and two chroma blocks, as shown in Fig. 5.14(a). For the DC and AC prediction, we need to store pixels of the first row and the first column of the 8×8 block. The shaped area in Fig. 5.14(b) shows the pixels that we need to store. In MoMuSys, it stores the needed pixel values in all blocks of each MB in the VOP and it uses one word of memory space to store each pixel value. Therefore, for the worst case, it needs a memory space of 35,640 bytes for the DC and AC prediction in QCIF format. Obviously, the usage of memory is inefficient, especially when we have only 64 kB of data memory on PACDSP.

In our implementation, we first reduce the memory space for each pixel value to half word. Moreover, only the neighbor blocks are needed for the DC/AC prediction, which are the left block, the above-left block, and the block immediately above. Therefore, in-

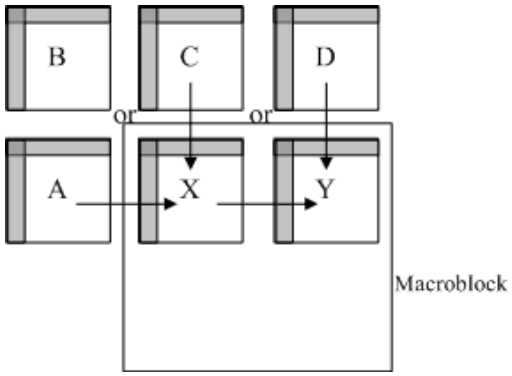


Figure 5.13: DC/AC prediction in MPEG-4 video decoder.

stead of storing the MBs of the whole VOP, we only store the necessary MBs. Fig. 5.15 shows the design of our implementation for two successive MBs. Only three parts of MBs are stored, which are the current MBs, the left MBs, and the MBs of the above row. Note that for the MBs of the above row, block 0 and block 1 shown in Fig. 5.14 are useless in DC and AC prediction. So we only store four blocks for the above row in our implementation. After this for memory usage reduction, we only need 1,680 bytes for DC and AC prediction in our implementation. Table 5.5 shows the detail of memory usage in DC and AC prediction and the comparison between the MoMuSys and our implementation.

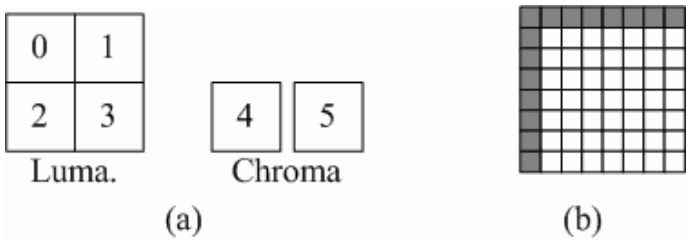


Figure 5.14: (a) Total blocks in one MB. (b) Pixels store for DC/AC prediction of one block.

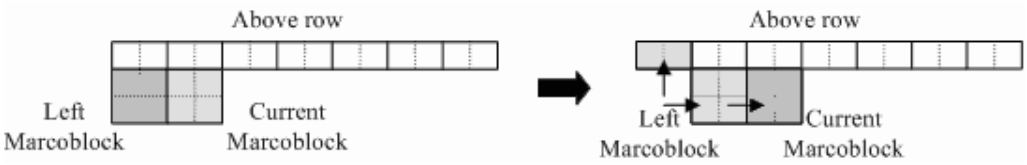


Figure 5.15: Memory usage design of DC/AC prediction for two successive MBs.

Table 5.5: Memory Usage Comparison of DC/AC Prediction on PACDSP

	MoMuSys	Our Implementation
Memory used for each pixel value	1 word	1 half word
MBs stored for prediction	Whole MBs in the VOP	Left, current MBs, and the above row without block 0 & block 1
Total memory used for QCIF video	$11 \times 9 \times 6 \times 15$ words = 35,640 bytes	$11 \times 4 \times 15 + 2 \times 6 \times 15$ half words = 1,680 bytes

Code Flow Modification

After the optimization of memory usage, we now discuss our modification of the code flow which can do the DC/AC prediction more efficiently. Fig. 5.16 shows the program flow of DC/AC prediction in MoMuSys. The DC prediction is executed first, which contains two steps. At first, we find the position of the prediction block, and then we get the prediction DC value from this block. After the DC prediction, we check the flag "ACPred_flag" to determine whether the AC prediction is necessary or not. Similar to DC prediction, the AC prediction also contains two steps, finding prediction block and getting prediction AC value. However, we use the same prediction block for DC and AC prediction.

In our implementation, we modify the code flow. When we find the prediction block, we check the "ACPred_flag" before getting the DC value. If we need to do the AC prediction, we get both DC and AC prediction values. Therefore, we can reduce the calculation for finding the prediction block in our implementation.

So far, we have shown several implementation strategies with optimization of the

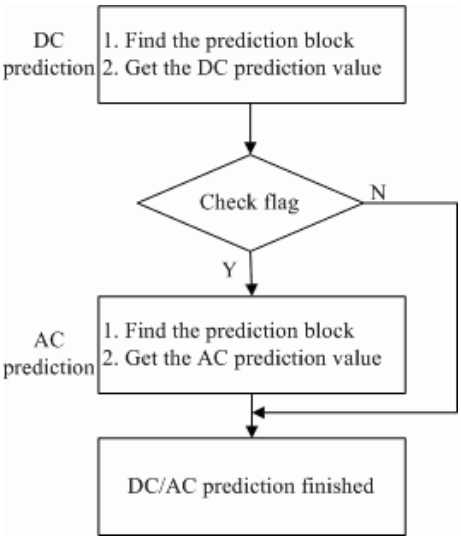


Figure 5.16: Program flow of DC/AC prediction in MoMuSys.

MPEG-4 object-based video decoder on PACDSP. Table 5.6 shows the performance of our implementation on PACDSP. Table 5.6 contains two parts, which are intra decoding and inter decoding. The inter decoding on PACDSP does not include includes the "VOP-Padding," "MotionCompensation," and the "Blend" functions, which are placed on ARM. In Table 5.6, the execution time of intra decoding are approximately proportional to the VOP size. Therefore, "akiyo" takes the longest execution time for intra decoding. However, both VOP size and sequence characteristic affect the performance of inter decoding. In the test sequences, the foreground object in "akiyo" is almost static. Thus most of the residuals for inter decoding in this sequence are zero, which lead to a short bitstream and fast decoding time. Therefore, the "akiyo" has the least execution time in inter decoding, even less than "stefan".

Table 5.6: Performance of MPEG-4 Object-Based Video Decoder on PACDSP

Test Seq. (QCIF)	Intra (cycles)	Inter (cycles)
stefan	1,114,552	1,040,929
foreman	2,510,208	1,795,598
akiyo	2,532,856	614,918

5.1.4 Optimization of IDCT on PACDSP

In the PACDSP, there are two clusters for doing computations at the same time. And for the IDCT, 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.

In addition, 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. After our optimization, we need 307 cycles to carry out a 8×8 block IDCT.

The performance of various IDCT implementation are listed in Table 5.7. In Table 5.7, we also use the number of processing units and the execution time to estimate the number of fetched instructions for each method. In this way, we can get a idea about the complexity in each method. We see that our implementation of IDCT on PACDSP is competitive, because of less arithmetic units required.

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 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 increase in throughput of computations.

Table 5.7: Comparison of IDCT on Different Platforms

Designs	Processing units	Clock (MHz)	2-D fast algo.	Cycles	Instruction counts
TI C62x [20]	2 MUL, 6 ALU	200	row-column	230	1,840
TI C64x [21]	2 MUL, 6 ALU	600	row-column	154	1,232
IDCT Core [20]	1 ALU	33	direct 2-D	1,208	1,208
PACDSP (ours)*	2 AU, 2 L/S	200	even-odd	307	1,228

*Note: If we consider the scalar unit, the instruction counts is 1,535 in our implementation

5.2.1 General Optimization Techniques

For our implementation on PACDSP, we should try to fill all the slots in an instruction packet to get a higher performance. Therefore, how to achieve a full-pipeline implementation is very important to a better performance. In this subsection, three general optimization techniques are discussed, which are static rescheduling, loop unrolling, and software pipelining [11]. The purpose of these techniques is to reduce the number of stalls resulting from hazards, and the appropriateness for PACDSP of these techniques are discussed as well.

For the discussion, we use an example of coefficients summing in a 1-D array, which contains eight 8-bit data. Fig. 5.17 shows the corresponding C program. 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 5.17: Example C code of vector addition.

Static Rescheduling

In the assembly code programming, the dependence of data may cause stalls in processor, and these stalls 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 of Fig. 5.18, we simply translate the C program in Fig. 5.17 to the PACDSP assembly code. We can see that because the dependency of the register D0 and the data loading from memory requires two cycle to be valid in PACDSP, two stalls are inserted after the "LB" instruction. In addition, the conditional branch, whose predicate register is p2, depends on the comparison instruction "SLTI." And the predicate register also need two cycles to be valid for conditional execution, so two stalls are inserted after the "SLTI" instruction. Therefore, there are totally 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. 5.18, we reschedule the order of the assembly code, which reduces the stalls 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 stalls in a single iteration.

To use the unrolling technique, we have to find the independent computations in 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 5.19 shows the assembly code before and after loop unrolling.

In Fig. 5.19, we see 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 tradeoff between execution time and corresponding code size. Although the stalls are all eliminated, the code size increases after loop unrolling. Therefore,

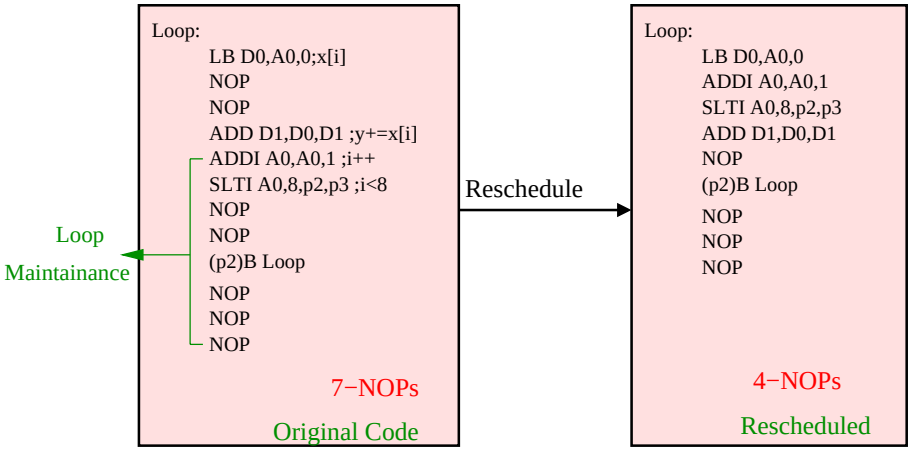


Figure 5.18: Example of static rescheduling technique.

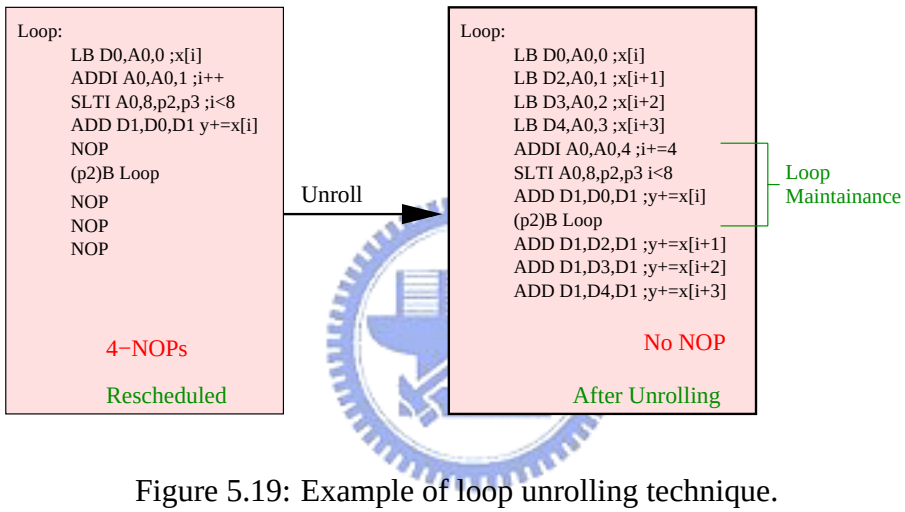


Figure 5.19: Example of loop unrolling technique.

we have to assess that if code size is critical or not. In addition, the number of available registers is a limitation to the use 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. 5.20.

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

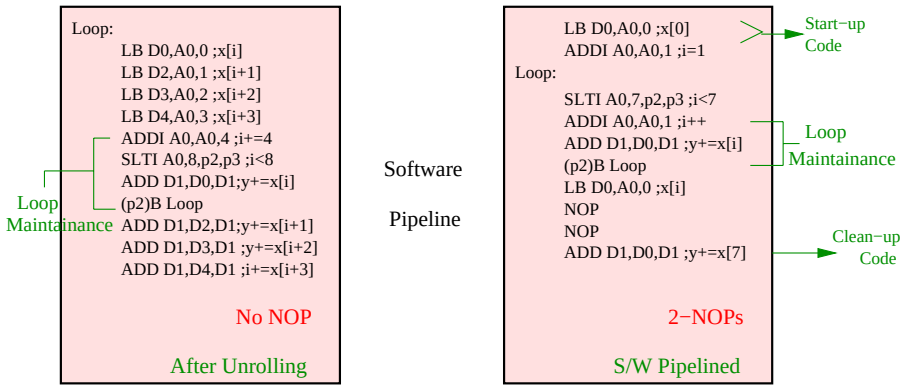


Figure 5.20: Example of software pipelining technique.

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, which are suitable to distribute into two clusters. Moreover, SIMD instructions are also very helpful for our optimization.

5.2.3 Experiment Result of Architectural Optimization

After our architectural optimization, including general optimization techniques and using the advantages of PACDSP, the improvement is shown in Table 5.8. We can find that

the architectural optimization introduces significant improvement, up to at most 28.27 percent. It is thus clear that the number of stalls affect the performance greatly. We can increase the performance of our implementation, if we reduce the stalls in the assembly code.

5.3 Algorithmic Optimization

In this section, we discuss the algorithmic optimization, which focuses on elimination of inverse scan, dequantization (IQ), and IDCT (IT) in texture decoding. We separate our discussion into two subsections. In the first subsection, we discuss the optimization of inverse scan, and then we consider the optimization of IQ and IT in the second subsection. At last, we show the improvement of our implementation on PACDSP after the algorithmic optimization.

5.3.1 Efficient Inverse Scan

Fig. 5.21 shows the simplified program flow of texture decoding in MPEG-4 object-based video decoder. In Fig. 5.21, two flags we should pay attention to are the "VLD_flag" and the "ACPred_flag." "VLD_flag" and "ACPred_flag" point out the necessity of VLD after reconstruction of DC coefficient and the necessity of AC prediction, respectively. In this subsection, we discuss our optimization for reducing the executed times of inverse scan.

After the reconstruction of DC coefficient and VLD, one of three inverse scans is performed, which are alternate-horizontal scan, alternate-vertical scan, and zigzag scan.

Table 5.8: Improvement After Architectural Optimization on PACDSP

Test Seqs.	I-Frames (Cycles)			P-Frames (Cycles)		
(QCIF)	Original	Optimized	%	Original	Optimized	%
stefan	1,114,552	799,518	28.27	1,040,929	766,657	26.35
foreman	2,510,208	1,826,742	27.23	1,795,598	1,416,963	21.09
akiyo	2,532,856	1,834,410	27.58	614,918	546,104	11.19

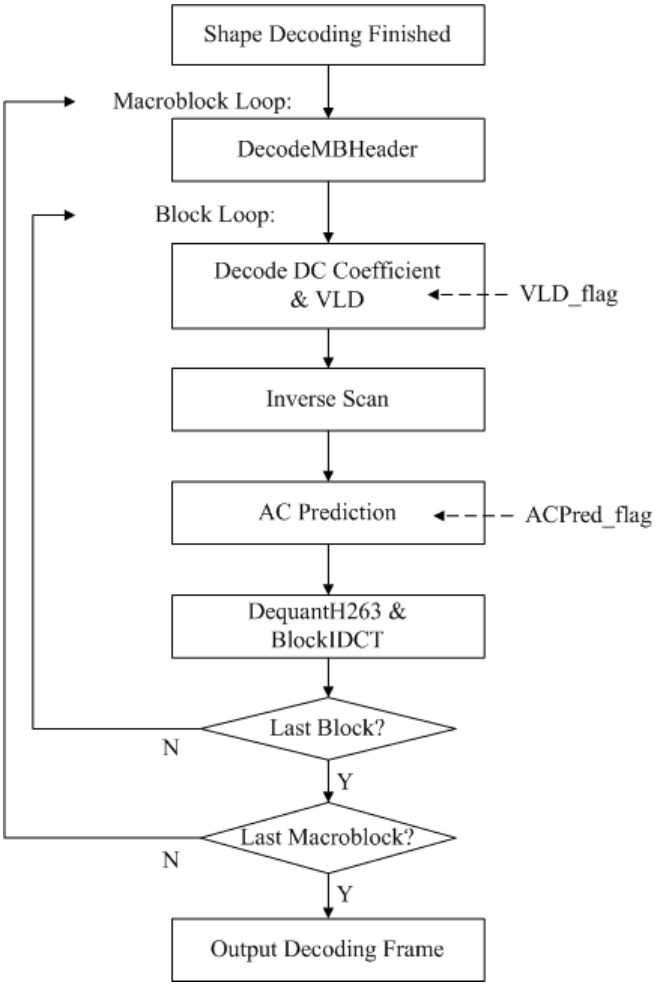


Figure 5.21: Program flow of texture decoding in MPEG-4 object-based video decoder.

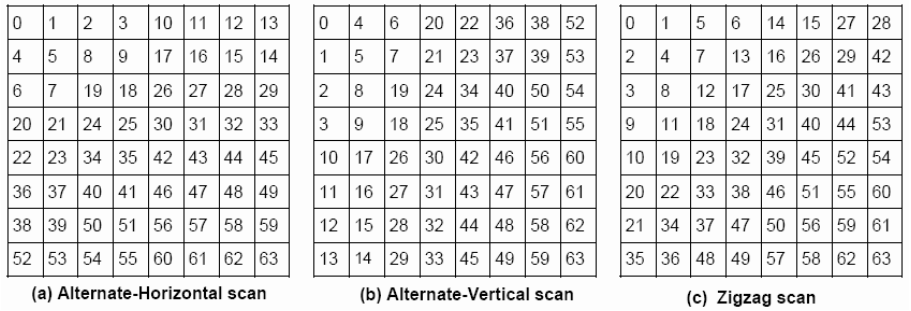


Figure 5.22: Scan orders for 8×8 blocks [5].

These scan orders are shown in Fig. 5.22. Then we can do the AC prediction (if necessary) after the inverse scan. However, if the VLD after the reconstruction of DC coefficient is unnecessary, which can be determined by checking the "VLD_flag," the decoded 8×8 block only has the DC component decoded from the bitstream. Then, there is no need of the inverse scan. Therefore, we can skip the procedure of inverse scan by checking the "VLD_flag."

By skipping the inverse scan, we can reduce the execution time of texture decoding. Table 5.9 shows the number of skipped blocks of different sequences by checking the "VLD_flag". We test twenty frames and nineteen frames for intra and inter decoding, respectively. It is obvious that the saving time is proportion to the number of skipped blocks. Note that because the texture decoding in fact decodes the residuals for inter frame, which has bigger chance to be zero. Therefore, the number of skipped blocks in inter frames is generally more than the number in intra frames.

Table 5.9: Number of Skipped Blocks in Twenty Intra Frames and Nineteen Inter Frames (Checking VLD_flag Only)

Test Seqs.	I-Frames (20 I)			P-Frames (19 P)		
(QCIF)	Total Blocks	Skipped Blocks	%	Total Blocks	Skipped Blocks	%
stefan	1,234	134	10.86	1,162	282	24.27
foreman	5,341	1,075	20.13	4,906	2,123	43.27
akiyo	5,281	922	17.46	2,710	1,801	66.46

5.3.2 Efficient IQ and IDCT

Besides the skip of inverse scan, we discuss the skipping possibility of dequantization (IQ) and IDCT (IT). As shown in Fig. 5.21, after the AC prediction, we need to do the IQ and IT of the decoded block to get the texture information. Similar to the skip of inverse scan discussed in the previous subsection, if we can find a method to skip the IQ and IT, the execution time can also be decreased. Fortunately, this idea can be realized by checking the "VLD_flag" and the "ACPred_flag."

As in the discussion about the "VLD_flag" before, we know that the decoded block only has the DC component if the VLD does not execute before the AC prediction. Moreover, if the AC prediction is not executed either, the decoded block for the IQ and IT still only has the DC component. In such a case, the texture information can be easily obtained without IQ and IT. In addition, because there is no AC prediction for inter MBs, we do not need to check the "ACPred_flag" in such a case.

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 only one coefficient at the DC after transform. In other words, if we can make sure that there is only a DC component in the decoded block, the corresponding output block data can be obtained with copying the DC component to the entire block, and such property is illustrated in Fig. 5.23. The assembly code of spreading DC value to the whole block is shown in Fig. 5.24. We need four iterations to complete one block, so the execution time

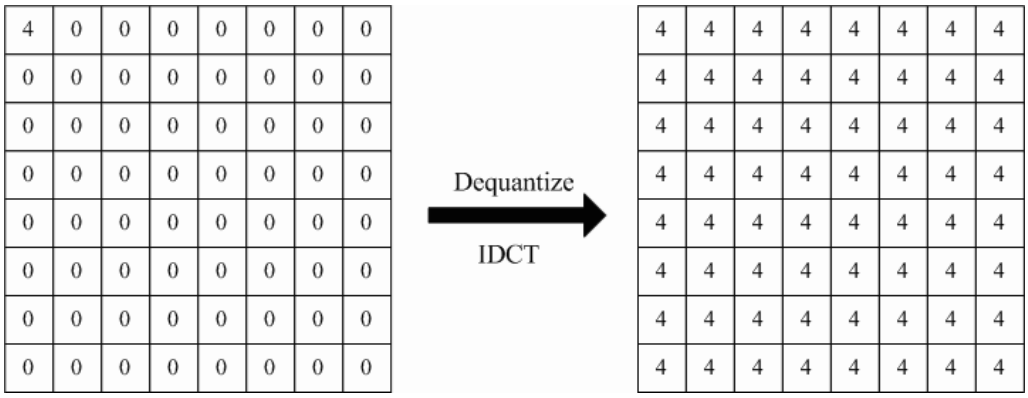


Figure 5.23: DC spreading from decoded coefficient to output block.

```
DC_Spreading: ; 4 iterations for one block
{ SET_LBCI RBCO,4 | MOVI.L A6,R_Block_2D | COPY D15,D14 | MOVI.L A6,R_Block_2D | COPY D15,D14 }
{ NOP | MOVI.L A6,R_Block_2D |  NOP  | MOVI.L A6,R_Block_2D |  NOP  };D14,D15 are DC value
{ NOP |         NOP          |  NOP  | ADDI A6,A6,128       |  NOP  }
Spread_DC_Coeff: ;iterations
{ LBCB RBCO,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 5.24: Assembly code of DC spreading.

Table 5.10: Number of Skipped Blocks in Twenty Frames and Nineteen Inter Frames form Checking VLD_flag and ACPred_flag (Intra Only)

Test Seqs.	I-Frames (20 I)			P-Frames (19 P)		
(QCIF)	Total Blocks	Skipped Blocks	%	Total Blocks	Skipped Blocks	%
stefan	1,234	119	9.64	1,162	282	24.27
foreman	5,341	655	12.26	4,906	2,123	43.27
akiyo	5,281	922	17.46	2,710	1,801	66.46

is 19 cycles including the setting of loop register and address registers. However, we still need several cycles to update the prediction data “DC_Store” for DC/AC prediction.

By checking the “VLD_flag” and “ACPred_flag” together, we can get the the number of skipped blocks for DC spreading as shown in Table 5.10, which includes twenty intra frames and nineteen inter frames. Compared to the earlier result of checking the “VLD_flag” only, because we check one more flag “ACPred_flag,” the number of skipped blocks is decreased. Fig. 5.25 shows the program flow of texture decoding after algorithmic optimization in our implementation.

5.3.3 Experiment Result of Algorithmic Optimization

Table 5.11 shows the improvement after the algorithmic optimizations, as discussed in this section. In Table 5.11, the decreased execution time is proportional to the number of skipped blocks. Because we have already done the optimization of those functions which we skip, such as fixed-point IDCT, not a high percentage of improvement is obtained by our algorithmic optimization. However, if the skipped functions are time-consuming, then

we can obtain much improvement by our algorithmic optimization.

5.4 Conclusion

In this chapter, we introduced several efficient implementation strategies for different function on PACDSP. We distributed the regular and independent computations into two clusters as much as possible. And we reduced the “NOP” instructions in the instruction packets. In addition, we also discussed the optimization on architecture and algorithm levels. The improvement in execution time of architectural and algorithmic optimization for intra frames and inter frames is shown in Figs. 5.26 and 5.27, respectively. Table 5.12 shows the overall improvement of our optimization on PACDSP. We can see that about 30% of execution time for decoding is reduced.

Table 5.11: Improvement After Algorithmic Optimization on PACDSP

Test Seqs.	I-Frames (Cycles)			P-Frames (Cycles)		
(QCIF)	Original†	Optimized	%	Original†	Optimized	%
stefan	799,518	783,725	1.98	766,657	737,283	3.83
foreman	1,826,742	1,743,121	4.58	1,416,963	1,214,174	14.31
akiyo	1,834,410	1,757,579	4.19	546,104	478,750	12.33

†Original means the execution time after architectural optimization.

Table 5.12: Overall Improvement After Optimization on PACDSP

Test Seqs.	I-Frames (Cycles)			P-Frames (Cycles)		
(QCIF)	Original†	Optimized	%	Original†	Optimized	%
stefan	1,114,552	783,725	29.68	1,040,929	737,283	29.17
foreman	2,510,208	1,743,121	30.56	1,795,598	1,214,174	32.38
akiyo	2,532,856	1,757,579	30.61	614,918	478,750	22.14

†Original means the execution time before optimization on PACDSP (architectural and algorithmic optimization).

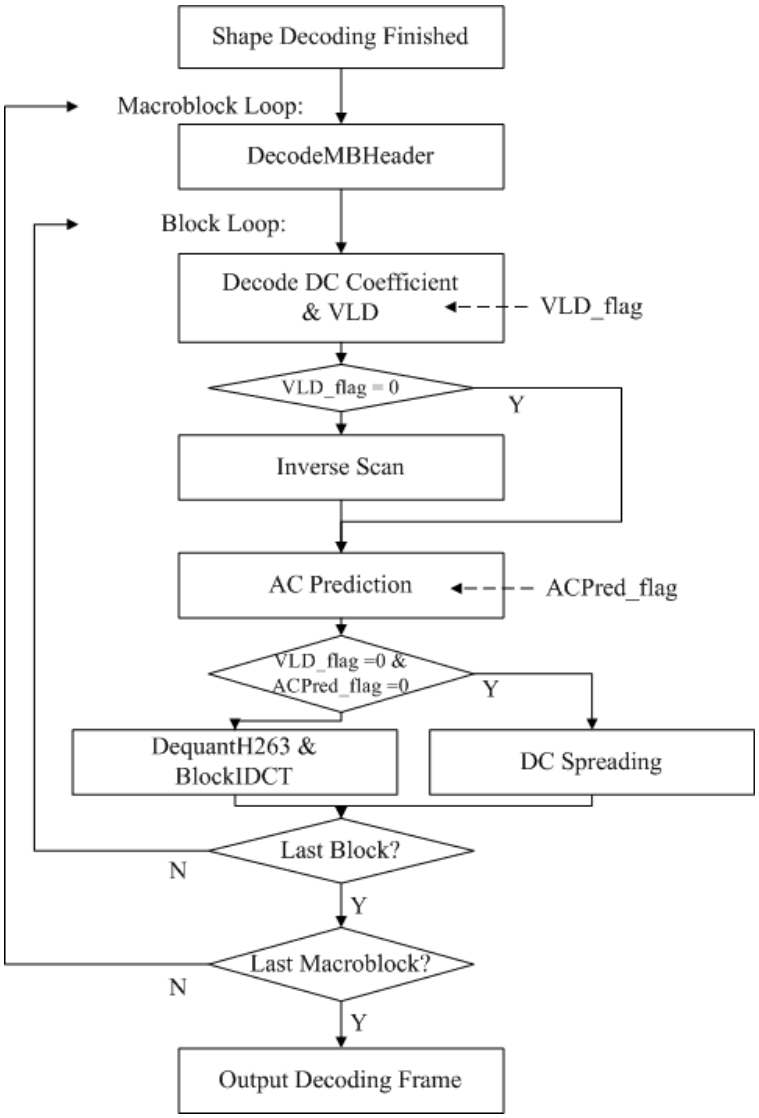


Figure 5.25: Program flow of texture decoding in MPEG-4 object-based video decoder after optimization.

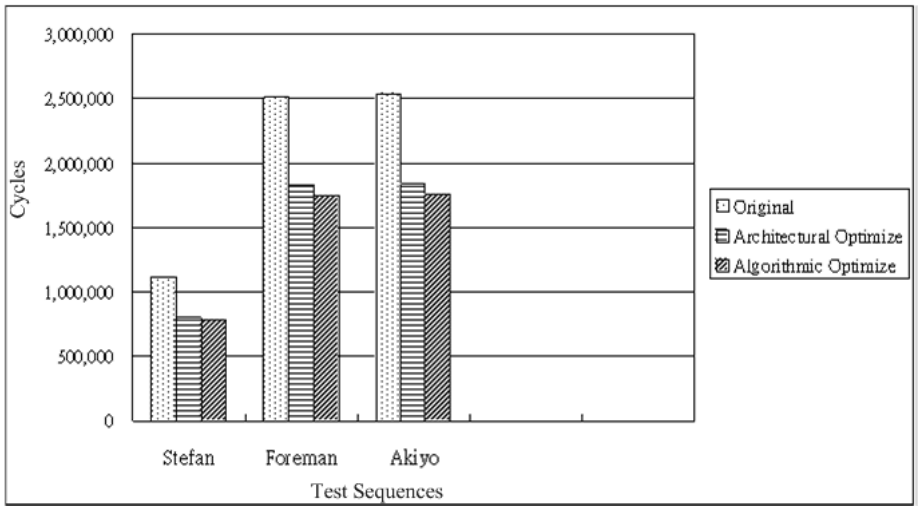


Figure 5.26: Improvement in execution time of architectural and algorithmic optimizations for I-frames on PACDSP.

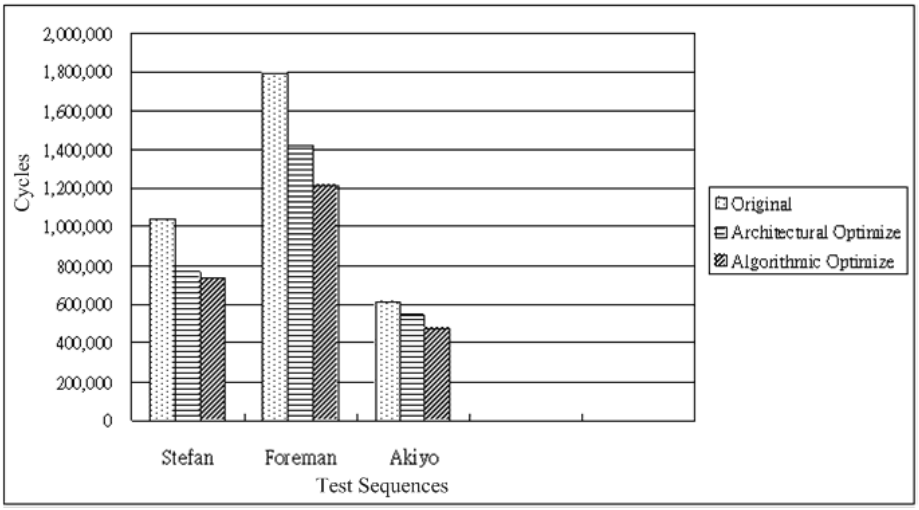


Figure 5.27: Improvement in execution time of architectural and algorithmic optimizations for P-frames on PACDSP.

Chapter 6

Overall Performance of the Implementation

In this chapter, we analyze the performance of our implementation of MPEG-4 object-based video decoder, includes the code size, data size and the decoding frame rate. At last, we discuss the effect of different QP values.

6.1 Performance Analysis

In this section, we discuss the code size and the data size of our implementation. In order to prevent the problem of cache miss, we must ensure that the sizes are smaller than the on-chip memory size provided by PACDSP, which are 32 kB and 64 kB for program and data, respectively. After analysis of the memory usage, we give an estimate of the frame rate of the implemented MPEG-4 object-based video decoder.

Code Size Analysis

Table 6.1 shows the code sizes of major functions in MPEG-4 object-based video decoder on PACDSP. The size of “AlphaDecodeMB” is the biggest, which does shape decoding. The size of “doDCACrecon” comes next. Since the instruction memory of PACDSP is 32 kB, we need to be concerned with the total size of our program. In our implementation, the total program size is 30,540 bytes, which is smaller than the instruction cache size.

Therefore, no cache miss will happen in our implementation.

Data Size Analysis

The data memory used in our implementation can be divided into several parts, which are shown in Table 6.2. The meaning of each item is as follows.

"Decoding Parameters" contain the header information of "VOLHeader" and "VOP-Header," which are set in the encoder. Before the decoding procedure, we must get them from the bitstream and store them in the data memory. In addition, because the number of registers is limited, we may need some memory space for storing the parameters, which are useful in the decoding procedure. Such memory space also belongs to "Decoding Parameters". The "Decoded VOP" means the memory we use to store the final output of the VOP. However, the output means the residuals of the VOP in the inter decoding. We still need to do the motion compensation and reconstruction to get the final reconstructed VOP. Since the format of our seqence is QCIF, the memory space of this part is $176 \times 144 \times 1.5 = 38,016$ bytes, which contains both luminance and chrominance.

The "Result Store" means the memory used to store the result of some functions, such as the motion vectors and the alpha plane. We separate the "Result Store" into three major parts, which are for shape decoding, texture decoding, and motion vectors. Table 6.3 shows the used memory space of each part. "Ref. Information" stores the information of the reference VOP for inter decoding. Finally, the total data memory used in our implementation is 57,775 bytes without the bitstream. The required memory size is smaller than the memory size provided by PACDSP, which is 64 kB. Therefore, no cache miss will happento degrade the performance. We put the bitstream in the remaining memory space. Therefore, we cannot decode too many frames if the bitstream size is large.

Frame Rate Estimation

We now estimate the frame rate of the implemented decoder. The results are shown in Tables 6.4 and 6.5. We demonstrate the MPEG-4 object-based video decoder on the PDSK, which is a dual-core system. The operating frequencies of the two cores and the

Table 6.1: Code Size Profile of Object-Based MPEG-4 Video Decoder on PACDSP

Function Name	Code Size (bytes)	%
DecodeVOPHeader	480	1.57
DecodeFirst	1,436	4.70
AlphaDecodeMB	8,124	26.60
DecodeMBHeader	3,208	10.50
VlcGetBlock	2,048	6.70
doDCACrecon	4,376	14.33
BlockIDCT	1,112	3.64
BlockDequantH263	360	1.18
DecodeMBMVs	3,712	12.15
BitstreamAccess	864	2.83
Others	5,300	15.80
Total	30,540	100.00

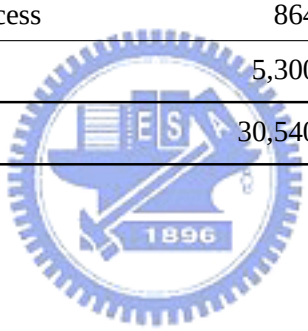


Table 6.2: Data Size Profile of Object-Based MPEG-4 Video Decoder on PACDSP

Usage	Memory Size (bytes)	%
Decoding Parameters	328	0.57
Decoded VOP	38,016	65.8
Result Store	9,412	16.29
Ref. Information	3,283	5.68
Table Information	6,736	11.66
Total	57,775	100.00

Table 6.3: Data Size Analysis of “Result Store” on PACDSP

Usage	Memory Size (bytes)	%
Shape Decoding	5,616	59.67
Texture Decoding	2,804	29.79
Motion Vectors	992	10.54
Total	9,412	100.00

transmitting frequency of the bus are shown below.

1. ARM core: 150 MHz.
2. PACDSP core: 200 MHz (real chip).
3. Bus: 22.5 MHz (32 bits width).

There are three major parts in Tables 6.4 and 6.5, which are ARM core, PACDSP core and the transmitted data between the two cores. The “cycles” of the “ARM” and “PACDSP” mean the execution times of ARM core and PACDSP core, respectively. We can get the execution times by dividing them by the operating frequencies. In addition, because the system is dual-core, we need to transmit data between two core modules, which is why Tables 6.4 and 6.5 contain entries called “Transmitted Data”. Note that the bus width is 32 bits. It means that we can transmit 32 bits of data at one time. We also can get the execution time of data transmission on bus by dividing the transmitting frequency. Moreover, the percentage of the total execution time for each part is shown in both tables. The total execution time of our implementation of MPEG-4 object-based video decoder is also shown in the tables. Then we can estimate the frame rate of each sequence, which is shown at the bottom of both tables. Note that for inter decoding, we separate the ARM core into two parts, which are “Padding&MC” and “Others.” According to our dual-core design, the “Padding&MC” procedure is overlapped with the procedure of PACDSP. Therefore, we only need to consider the longer part when we compute the total execution time. In other words, the percentage of the shorter part contributed nothing to the total execution time.

Table 6.4: Frame Rate Estimation for Intra Decoding of Our Implementation

Test Seq. (QCIF)		stefan	foreman	akiyo
ARM	(cycles)	231,080	807,932	923,210
	%	26.46	36.09	37.57
PACDSP	(cycles)	783,725	1,743,121	1,757,579
	%	67.35	56.71	53.70
Transmitted	(bytes)	32,272	112,912	129,040
Data	%	6.19	7.2	8.73
Execution Time	(ms)	5.82	15.36	16.37
Frame Rate	(fps)	171.8	65.1	61.1
Pixels Per Second		1,187,481.6	1,574,899.2	1,689,292.8

Table 6.5: Frame Rate Estimation for Inter Decoding of Our Implementation

Test Seq. (QCIF)		stefan	foreman	akiyo
ARM	(cycles)	918,189	1,858,485	1,808,689
Padding&MC	%	73.21	58.61	55.86
ARM	(cycles)	282,058	1,122,723	1,211,833
Others	%	22.49	35.38	37.42
PACDSP	(cycles)	737,283	1,214,174	478,750
	%	0	0	0
Transmitted	(bytes)	32,524	113,794	130,048
Data	%	4.30	6.01	6.72
Execution Time	(ms)	8.36	21.14	21.59
Frame Rate	(fps)	119.6	47.30	46.32
Pixels Per Second		826,675.2	1,144,281.6	1,280,655.4

For the sequence of “stefan” with the smallest VOP size, we can get the best frame rate which are 171.8 and 119.6 frames per second for the intra and inter decoding, respectively. For the sequences of “foreman” and “akiyo”, we still can reach about 60 fps and 45 fps for intra and inter decoding. However, in the demo system, the PACDSP core module is on FPGA with the DSP design burned in, whose operating frequency s 22 MHz rather than the 250 MHz for a real chip. The lower operating frequency will degrade the performance of our implementation. The real frame rate of the demo system is discussed as following.

Frame Rate Estimation of the Demo System

For the demo system, the PACDSP core module is replaced by a FPGA rather than a real chip. The operating frequency of each core and the bus transmitting rate of our demo system are list below:

- ARM core: 150 MHz.
- PACDSP core: 22 MHz (FPGA).
- Bus: 22 MHz (32 bits width).

Using above data, we can estimate the real frame rate of our demo system, which are shown in Table 6.6 and Table 6.7.

Because we choose the debug mode for compiling on ARM core, and add some extra functions that are necessary for displaying but useless for decoding procedure, the execution time of ARM core increases much. Moreover, because the operating frequency of the FPGA is much lower than the real chip, the execution time of FPGA core also increases. Therefore, the performance degrades greatly for the real demo platform.

6.2 Effect of Different Quantization Steps (QP)

In the MPEG-4 video encoder, quantization follows the DCT. Therefore, the quantization step size affects the block coefficients. In the above discussion, we let QP value be 4 in all cases. To have further understanding of how QP affects the video coding, we do some

Table 6.6: Frame Rate Estimation for Intra Decoding on Demo Platform

Test Seq. (QCIF)		stefan	foreman	akiyo
ARM	(cycles)	2,177,643	2,855,869	2,995,292
	%	22.12	14.16	14.63
PACDSP	(cycles)	1,114,552	2,510,208	2,532,856
	%	77.31	84.89	84.30
Transmitted	(bytes)	32,272	112,912	129,040
Data	%	0.57	0.95	1.07
Execution Time	(ms)	65.54	134.41	136.54
Frame Rate	(fps)	15.26	7.44	7.32
Pixels Per Second		105,477.1	179,988.5	202,383.4

Table 6.7: Frame Rate Estimation for Inter Decoding on Demo Platform

Test Seq. (QCIF)		stefan	foreman	akiyo
ARM	(cycles)	4,048,579	12,712,506	14,040,483
Padding&MC	%	0	52.76	55.22
ARM	(cycles)	9,156,179	11,187,675	11,162,323
Others	%	56.14	46.43	43.91
PACDSP	(cycles)	1,040,929	1,795,598	614,918
	%	43.52	0	0
Transmitted	(bytes)	32,524	113,794	130,048
Data	%	0.34	0.81	0.87
Execution Time	(ms)	108.72	160.62	169.50
Frame Rate	(fps)	9.20	6.23	5.90
Pixels Per Second		63,590.4	150,716.2	163,123.2

analysis for different QP values in this section. We consider three different QP values, which are 3, 4 and 8. And we discuss the effect on the number of skipped blocks for our algorithmic optimization. Since there are two kinds of algorithmic optimization in our implementation, we do the analysis for them separately.

Tables 6.8 and 6.10 show the numbers of skipped blocks under different QP values. In our analysis, we decode 20 I-frames and 19 P-frames as shown. Since a larger QP value introduces a rougher quantization, more block coefficients may be quantized to the same value. As a result, the coefficient after DC/AC prediction may be simpler, and the number of skipped blocks for our algorithmic optimization increases. When we increase the value of QP, the percentage of skipped blocks also increases. In addition, the block coefficients of inter coding have a larger probability to be zero, which results in a larger percentage of skipped blocks, especially for “akiyo” which is quite stationary. Obviously, the execution time will decrease when we increase the QP value, since the percentage of skipped blocks increases with the QP value. We show the execution time of each sequence with different QP in Tables 6.9 and 6.11.

Table 6.8: Number of Skipped Blocks in 20 Intra Frames with Different QP values

Test Seqs. (QCIF)	QP	Total Block No.	Check VLD_flag		Check VLD_flag & ACPred_flag	
			Skipped Blocks	%	Skipped Blocks	%
stefan	3	1,234	111	9.00	102	8.27
	4	1,234	134	10.86	119	9.64
	8	1,234	235	19.04	201	16.29
foreman	3	5,341	893	16.72	544	10.19
	4	5,341	1,075	20.13	655	12.26
	8	5,341	1,820	34.08	1,063	19.90
akiyo	3	5,281	777	14.71	615	11.65
	4	5,281	922	17.46	733	13.88
	8	5,281	1,604	30.37	1,073	20.32

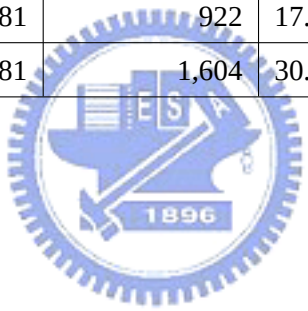


Table 6.9: Effects of Different QP to Execution Time of I-Frame Decoding on PACDSP

Test Seqs.	Execution Time (Cycles Per Frame)		
(QCIF)	QP = 3	QP = 4	QP = 8
stefan	837,280	783,725	674,367
foreman	1,822,649	1,743,121	1,540,830
akiyo	1,878,614	1,757,579	1,486,613

Table 6.10: Number of Skipped Blocks in 19 Inter Frames with Different QP

Test Seqs. (QCIF)	QP	Total Block No.	Check VLD_flag		Check VLD_flag & ACPred_flag	
			Skipped Blocks	%	Skipped Blocks	%
stefan	3	1,162	211	18.16	208	17.90
	4	1,162	282	24.27	282	24.27
	8	1,162	511	43.98	511	43.98
foreman	3	4,906	1,633	33.29	1,633	33.29
	4	4,906	2,123	43.27	2,123	43.27
	8	4,840	3,204	66.20	3,201	66.14
akiyo	3	3,487	2,158	61.89	2,146	61.54
	4	2,710	1,801	66.46	1,786	65.90
	8	2,023	1,573	77.76	1,573	77.76

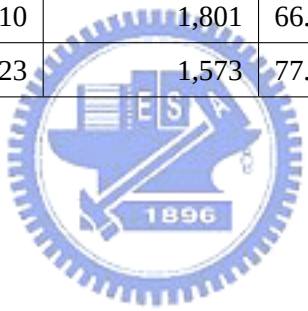


Table 6.11: Effects of Different QP to Execution Time of P-Frame Decoding on PACDSP

Test Seqs.	Execution Time (Cycles Per Frame)		
(QCIF)	QP = 3	QP = 4	QP = 8
stefan	833,126	737,283	697,521
foreman	1,355,579	1,214,174	1,068,700
akiyo	545,047	478,750	474,627

Chapter 7

Conclusion and Future Work

7.1 Conclusion

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

Before our implementation on PACDSP, we first analyzed the reference software of MPEG-4, MoMuSys, and done the profiling on the PC. By the analysis of the reference software, we had an initial understand of the decoding flow and the critical part of computation. We could design the more efficient strategies of our implementation according to the analysis. Since the PACDSP platform that we demonstrated on was a dual core system, we then discussed the dual core design of our implementation.

After the implementation on both processing core was verified and optimized, we also utilized several general software optimization techniques, such as static rescheduling, loop-unrolling, and software-pipelining to reduce the stalls. Moreover,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.

Finally, the optimization results were discussed. For the best case, stefan, which has the smallest VOP size, we can decode the MPEG-4 video bitstream over 171 frames and 119 frames per second for intra and inter decoding, respectively. And the program size is 30 KB, which is smaller than the instruction cache size. In addition, the used data

size was also under the limit of memory that provided on PACDSP. Therefore, no cache missing problem happened in our implementation. In conclusion, the performance of our implementation of MPEG-4 object-based video decoder on PACDSP is competitive.

7.2 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

  For the implementation on DSPs, the design of data structure is very important, which affects the performance highly. If we can design the more efficient data structure, the memory accesses can be significantly reduced, and the performance also can be improved.

- 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, the number and the size of decoding frames are also constrained. However, the internal memory of PACDSP can be accessed by the ARM core on the PSDK platform, then we can manage the memory through ARM core, and the usable memory size is enlarged.

  In addition, some functions like the VLD. Because it has many branch instructions in its decoding procedure, which degrades the performance of implementation on PACDSP. In other words, using PACDSP to implement the VLD has no advantage. We can redesign the dual-core implementation, and use the suitable core module to implement each functions.

- Implement on PACDSP v3.0

In this thesis, we consider the implementation of MPEG-4 video decoder on PACDSP v2.0. However, the latest version of PACDSP is version 3.0 which support some new and useful instructions. We can further implement the decoder on PACDSP v3.0, and use the new instructions to improve the performance.

- Add other MPEG-4 tools

  In our implementation, the tool of error-resilience in MPEG-4 simple profile is left. However, for the bitstream transmitted through a real channel, this tool is very important. We need to consider the implementation of error-resilience in the future. Moreover, we also can implement other advanced profiles of MPEG-4 video decoder for more decoding tools to extend the capability of PACDSP.

Bibliography

[1] SOC Technology Center, Industrual Technology Research Institute, *PACDSP v2.0 — Instruction Set Menu*. Doc. no. PACDSP2S0000, June 2005.

[2] SOC Technology Center, Industrual Technology Research Institute*PACDSP v3.0 — Software Developer's Bible — Vol. 1 Software Developer's Guide*. Doc. no. PACDSP3S0001, Feb. 2006.

[3] SOC Technology Center, Industrual Technology Research Institute*PACDSP v3.0 — Software Developer's Bible — Vol. 2 Instruction Set Manual*. Doc. no. PACDSP3S0002, April 2006.

[4] SOC Technology Center, Industrual Technology Research Institute *PACDSP v3.0 — Software Developer's Bible — Vol. 3 Programming Constraints and Optimization Guide*. Doc. no. PACDSP3S0003, May 2006.

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

[6] Chung-Yen Tsai, "Software implementation of MPEG-4 video decoder on PACDSP platform," M.S. thesis, Department of Electronics Engineering, National Chiao Tung University, Hsinchu, Taiwan, R.O.C., July 2006.

[7] 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.

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

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

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

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

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

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

[14] 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.

[15] 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.

[16] 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.

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

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

[19] Y. Chen and P. Hao, “Integer reversible transformation to make JPEG loseless,” in *Int. Conf. Siganl Processing*, Beijing, China, Sep. 2004, pp. 835–838.

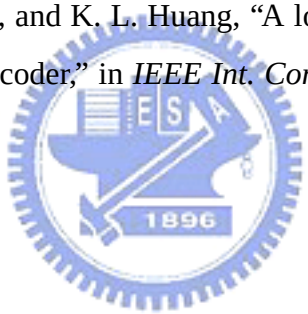
[20] T.S. Chang, C.S. Kung, and C.W. Jen, “A simple processor core design for DCT/IDCT transform,” *IEEE Trans. Circuits Syst. Video Technology*, vol. 10, no. 3 , pp. 439–447, Apr. 2000.

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

[22] N. Ventroux, J. F. Nezan, H. Raulet, and O. Deforges, “Rapid prototyping for an optimized MPEG-4 decoder implementation over a parallel heterogenous architecture,” in *Proc. Int. Conf. Multimedia Expo*, vol. 3, July 2003, pp. 417–420.

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

[24] J. H. Kuo, J. L. Wu, J. Shiu, and K. L. Huang, “A low-cost media-processor based real-time MPEG-4 video decoder,” in *IEEE Int. Conf. Consumer Electronics*, June 2002, pp. 272–273.



自傳

許介遠，男，民國七十二年二月十八日出生於台灣省高雄市。高中就讀於高雄中學。大學就讀國立交通大學電信工程學系，於民國九十四年六月畢業。並在同年九月進入交通大學電子工程研究所碩士班，於民國九十六年六月取得碩士學位，論文題目為:『MPEG-4 物件視訊解碼器在 PACDSP 平台上之軟體實現』。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。