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 物件視訊編碼器在 PACDSP 平台上之軟體實現

Software Implementation of MPEG-4 Object-based Video Encoder on PACDSP Platform

研 究 生： 江政達

指導教授： 林大衛 博士

中 華 民 國 九 十 六 年 六 月

MPEG-4 物件視訊編碼器在 PACDSP 平台上之軟體實現

Software Implementation of MPEG-4 Object-based Video Encoder on PACDSP Platform

研 究 生 ：江政達　　Student: Cheng-Ta Chiang

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

MPEG-4 物件視訊編碼器在 PACDSP 平台上之軟體實現

研究生：江政達　　　　　　　　　　　　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 v3.0 平台上 MPEG-4 物件視訊編碼器之實現，本平台由一超長指令數位訊號處理器與一 ARM926EJ-S 處理器所組成。為了最佳化程式流程，我們也完成了許多的靜態分析，並且利用超長指令處理器架構上之特性來達到即時編碼。我們已可在 ARM 平台上呈現簡單的展示，並在指令集模擬器上驗證 DSP 部分之正確性。

在我們的實作當中，我們使用了 MPEG-4 參考軟體，MoMuSys，當作驗證的比較對象。首先，我們分析了 MPEG-4 物件視訊編碼器之統計特性並且對編碼流程有了初步的瞭解。接著，我們分析編碼之運算複雜度並且藉此找到有效率的實現方法。在移動估測編碼中，我們利用螺旋搜尋法中的一項參數來降低運算複雜度，並且沒有犧牲太多的影像品質。在形狀編碼中，我們使用多重符號之內容基礎的算術編碼(CAE)來壓縮二元形狀資訊，並在 inter 編碼模式中做調整以降低運算複雜度。在紋理編碼中，我們根據離散餘弦轉換(DCT)之特性來跳過多餘的運算。

為了加速執行時間，我們將規律之運算分佈於兩組以增加處理器之效能。我們也使用單指令多資料(SIMD)指令以及一般指令層級平行化來減少處理器之延遲。我們討論了離散餘弦轉換(DCT)和離散餘弦反轉換(IDCT)之效能與精確度，並且我們的離散餘弦反轉換(IDCT)實現能夠符合 IEEE 1180-1190 標準之規範。在所有的最佳化之後，我們在最好的情況下可分別在 intra 和 inter 編碼模式下達到每秒 33 和 43 張的 QCIF 畫面即時編碼。而整個程式的大小為 27 Kbytes，也小於 PACDSP 的程式快取記憶體大小 32 Kbytes。

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、最佳化方法、整體實作設計、以及實驗結果。最後簡單介紹了雙核心實現的系統與機制。

Software Implementation of MPEG-4 Object-based Video Encoder on PACDSP Platform

Student: Cheng-Ta Chiang Advisor: Dr. David W. Lin

Department of Electronics Engineering
Institute of Electronics
National Chiao Tung University

Abstract

MPEG-4 is a widely-applied multimedia coding standard. This thesis presents an implementation of MPEG-4 object-based video encoder on the PACDSP v3.0 platform, which consists of a VLIW digital signal processor (DSP) and an ARM926EJ-S processor. We complete many analysis to optimize the program flow and utilize the advantage of VLIW processor to achieve real-time encoding. We have done a simple demonstration on ARM core, and the encoding on DSP part is verified by instruction set simulator.

In our implementation, the MPEG-4 reference software, MoMuSys, is used as a golden model to verily our implementation. First, we analyze the statistics of the MPEG-4 object-based video encoder, and have an initial understand of the encoding flow. Second, we analyze the computation complexity of the coding, and find efficient algorithms for the implementation. In the motion coding, we use a parameter of spiral search to simplify the computation complexity without too much quality loss. In shape coding, we use multi-symbol CAE to compress the binary shape information and give some modification for inter mode coding to reduce computation complexity.

In texture coding, we skip some computations according to the mature of discrete cosine transform (DCT).

Third, to speed up the execution time, we distribute the regular computations to both clusters to increase the efficiency of the processor. Single instruction multiple data (SIMD) instructions and general instruction level parallelism also utilized to reduce the processor stalls. We also discuss the efficiency and accuracy of DCT and IDCT, and the accuracy of our IDCT implementation can meet the IEEE 1180-1190 standard. After all the optimizations, we can encode the MPEG-4 video data for QCIF format over 33 and 43 frames per second in the best case for intra and inter encoding. The code size is 27 Kbytes, which is smaller than the 32-Kbyte instruction cache on PACDSP.

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

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。

另外要感謝的，是實驗室的蔡崇諺學長和吳和璋學長。謝謝你們熱心地幫我解決了許多方面的疑問。

感謝通訊電子與訊號處理實驗室(commlab)，提供了充足的軟硬體資源，讓我在研究中不虞匱乏。感謝崑健、俊榮、鴻志、家揚、朝雄...等博班學長的指導，以及 94 級介遠、志岡、柏昇、耀鈞、順成、凱庭、錫祺、浩廷、育成、耀佮等實驗室成員，平日和我一起唸書，一起討論，也一起打混，讓我的研究生涯充滿歡樂又有所成長。期待大家畢業之後都能有不錯的發展。

最後，要感謝的是我的家人，他們的支持讓我能夠心無旁鶩的從事研究工作。另外感謝我的女友，牛怡婷，在我的求學過程一路相伴，面對壓力時不斷地鼓勵。

謝謝所有幫助過我、陪我走過這一段歲月的師長、同儕與家人。謝謝！

誌於 2007.6 風城交大

政達

Contents

List of Figures

List of Tables

Chapter 1

Introduction

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

The MPEG-4 standard for coding of audio-visual information has been widely adopted in various consumer products. There are several tools in the MPEG-4 standards, and they are used for different purposes. Since the present work is the first attempt to implement MPEG-4 video codecs on the dual-core system, we decide to implement the object-based part (arbitrary binary shape) of the MPEG-4 encoder first, and support the tools in simple profile without error-resilience. Some video tools of MPEG-4 video standard are left to the future work.

PACDSP is a high performance, low cost VLIW (Very Long Instruction Word) DSP for multimedia applications [3]. Optimized architecture for data stream applications gives a strong reason for system designers to use PACDSP to implement media codecs. The instruction set architecture (ISA) of PACDSP is optimized for audio and video applications, so PACDSP is suitable for products with multi-standard codec requirement. In addition, the low power design for PACDSP makes it possible to use PACDSP on portable devices.

For the best case in our dual-core implementation, we can encode the MPEG-4 video data over 33 frames and 43 frames per second in QCIF size for intra and inter encoding, respectively.

This thesis is organized as follows. Chapter 2 is the overview of MPEG-4 standards. Chapter 3 introduces the architecture and specification of the PACDSP platform. Chapter 4 is the development of our C code and the overall system design for our implementation. In addition, the algorithm analysis of MPEG-4 video encoder is also discussed in this chapter. The contents of chapter 5 are about the architecture optimization technologies and their experiment results. We also compare our implementation with that of other processors. The performance of dual-core implementation is shown in chapter 6. Finally, we will give some conclusions in chapter 7, and the future works are listed as well.

Chapter 2

Overview of the MPEG-4 Video Standard

The contents of this chapter have been taken to a large extent from [5]–[8].

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

An input video sequence can be defined as a sequence of related frames or pictures, separated in time. MPEG-4 divides a frame into a number of video object planes (VOPs). A succession of VOPs is termed a video object (VO). Fig. 2.1 shows the decomposition of a picture into a number of separate VOPs. 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.

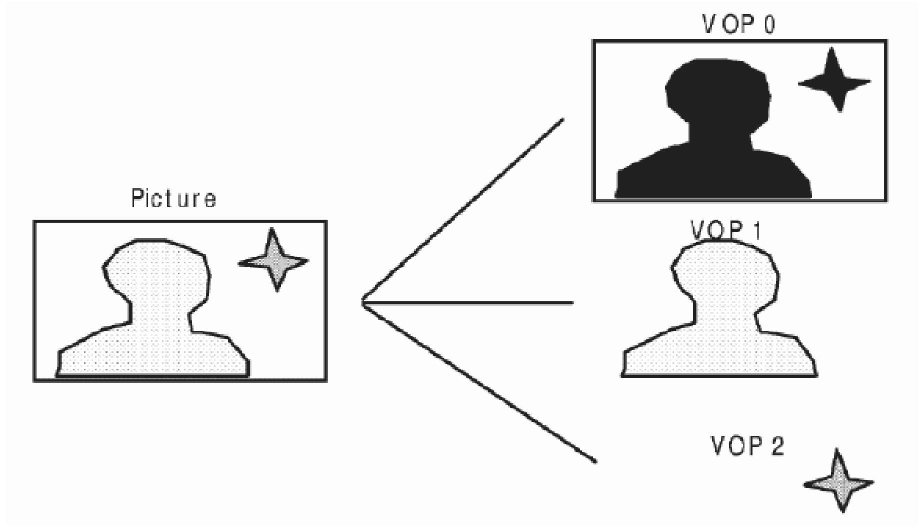


Figure 2.1: Segmentation of a frame into VOPs (from [6]).

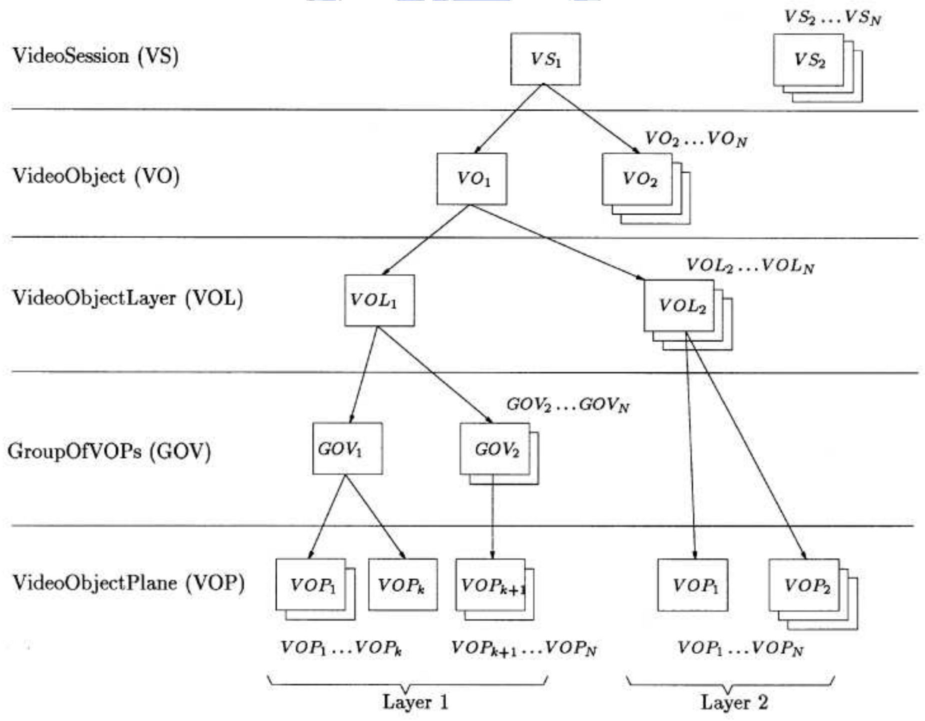


Figure 2.2: Structure of coded video data (from [7]).

- VideoSession (VS): A video session is the highest syntactic structure of the coded visual bitstream and simply consists of an ordered collection of video objects.

- VideoObject (VO): A video object represents 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.

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

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

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

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

4. A sprite (S) VOP is a VOP for a sprite object or a VOP that is coded using prediction based on global motion compensation from a past reference VOP. We omit further introduction of the S VOP.

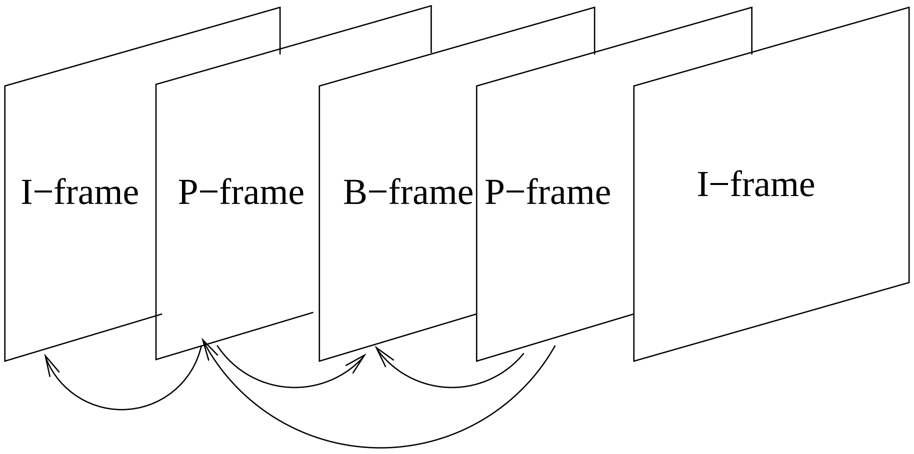


Figure 2.3: Types of VOP.

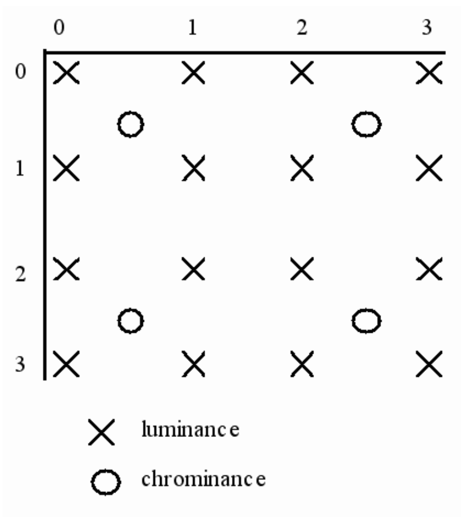


Figure 2.4: Positions of luminance and chrominance samples in 4:2:0 data (from [8]).

The macroblock (MB) is a basic coding structure constructing VOP. An MB contains a section of the luminance component of 16×16 pixels in size, 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]–[8].

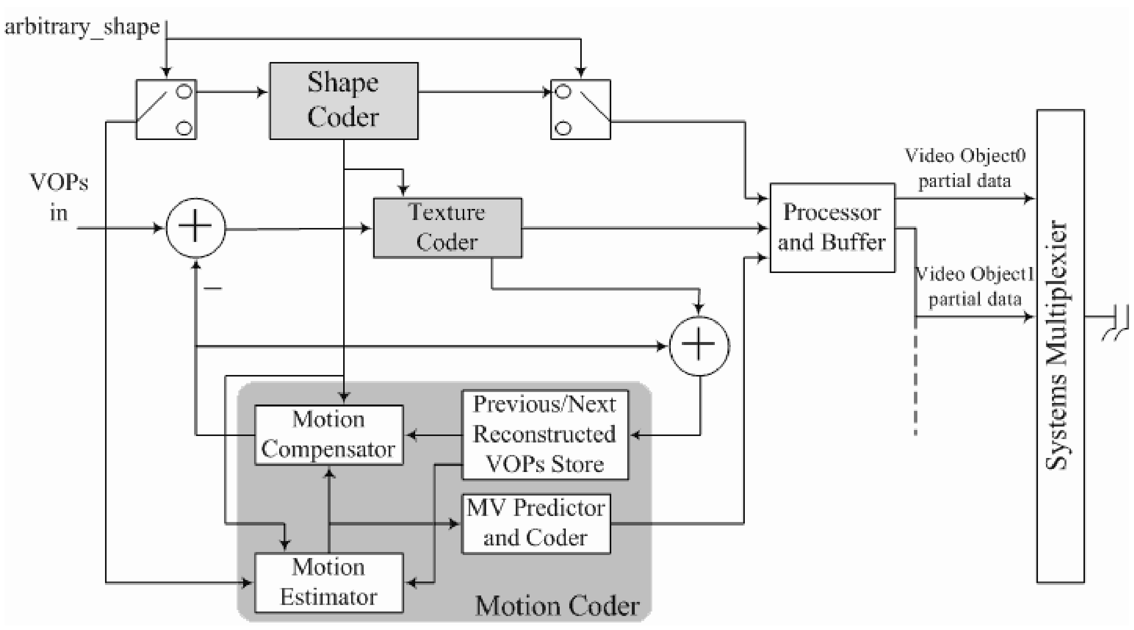


Figure 2.5: Detailed structure of VO encoder (from [6]).

Fig. 2.5 presrnts the detailed structure of the VO encoder. The encoder is mainly composed of three parts: shape encoder, motion encoder and texture coder. The reconstructed VOP is obtained by combining the shape, texture and motion information. The part of shape coding constitutes the major difference between frame-based and object-based coding.

2.2.1 VOP Formation

The video object shape information is obtained after segmentation. The shape information is hereafter referred to as alpha plane, which is used to form a VOP. There are two kinds of alpha planes in MPEG-4, binary alpha plane and gray scale alpha plane. For the binary alpha plane, the value 255 is assigned to pixels belonging to the objects and 0 is assigned to pixels outside the objects. The value of gray scale alpha plane is used for hybrid (of natural and synthetic) scenes generated by blue screen composition and is represented by an 8-bit component.

For the binary alpha plane, a rectangular bounding box enclosing the shape to be coded is formed such that its horizontal and vertical dimensions are extended to multiples

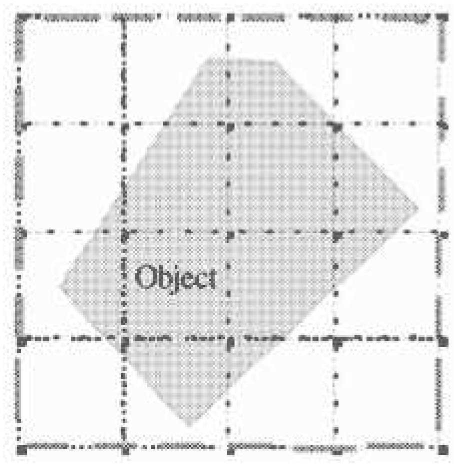


Figure 2.6: A VOP in bounding box (from [6]).

of 16 pixels (MB size). For efficient coding, it is important to minimize the number of macroblocks contained in the bounding box. Fig. 2.6 shows an example of an arbitrary shape VOP with bounding box and the MB structure.

2.2.2 Shape Coding

After VOP formation, the alpha plane of VOP will be coded prior to coding motion vector and texture based on the VOP image bounding box. Binary alpha planes are encoded by modified context-based arithmetic encoding (CAE) while grey scale alpha planes are encoded by motion compensated DCT similar to texture coding. An alpha plane is also 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.

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. InterCAE and IntraCAE are the variants of the CAE algorithm used with and without motion compensation, respectively. The motion vectors which are differentially coded can be computed by searching for a best match position. Each BAB can be coded in one of the following modes:

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.

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.

If the encoder need rate control and rate reduction, the encoder realizes these through size-conversion of binary alpha information. To be specific, a 4:1 downsampled binary alpha block is used first and if the shape errors are greater than a designed threshold value, a 2:1 downsampled binary alpha block is used next, again if it is found unacceptable, an unsubsampled binary alpha block is used.

Table 2.2: Shape Coding Modes and Their Main Usages (from [5])

	Mode	Main Usage
1	intra	I frames, arbitrarily shaped still texture object, error resilience
2	inter, inter MC	P frames
3	horizontal/vertical scanning	Low-bitrate shape coding
4	Subsampling to a block size 8×8 or 4×4	Low-bitrate lossy shape coding

The MPEG-4 standard allows for 18 coding modes of each BAB: (intra/inter/inter MC)×(horizontal/vertical scanning)×(subsampling factor 0/1/2). The influence of different shape coding modes on the performance of the coder in terms of coding efficiency but also in terms of computational complexity is of interest. Table 2.2 shows the main usage for each coding mode.

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

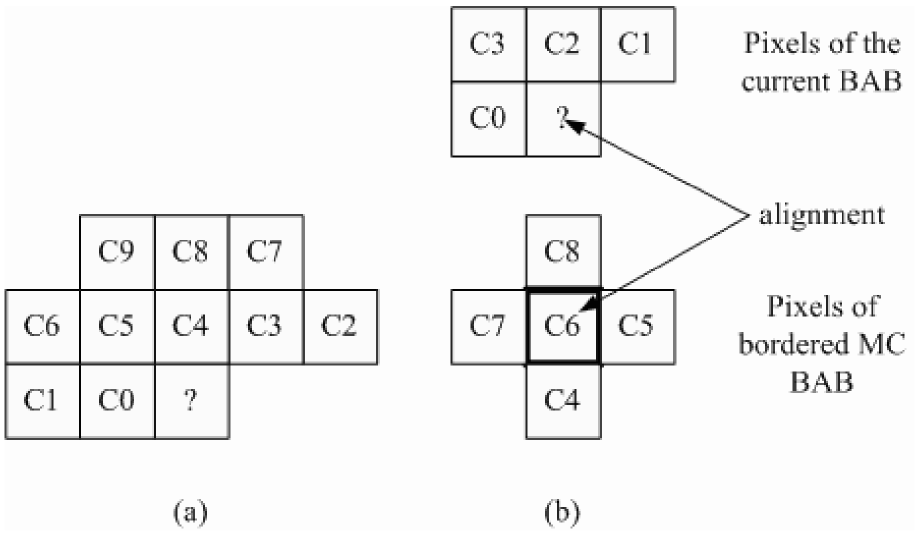


Figure 2.7: Pixel templates used for (a) INTRA and (b) INTER context calculation of BAB. The current pixel to be coded is marked with “?” (from [5]).

2.2.3 Motion Coder

Motion coding is essential for P-VOP and B-VOP to reduce temporal redundancy. The motion coder consists of a motion estimator, motion compensator, previous/next VOPs store and motion vector (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

Fig. 2.8 shows a simplified diagram of the padding process. The value of luminance and chrominance samples outside the VOP are defined by the padding 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 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

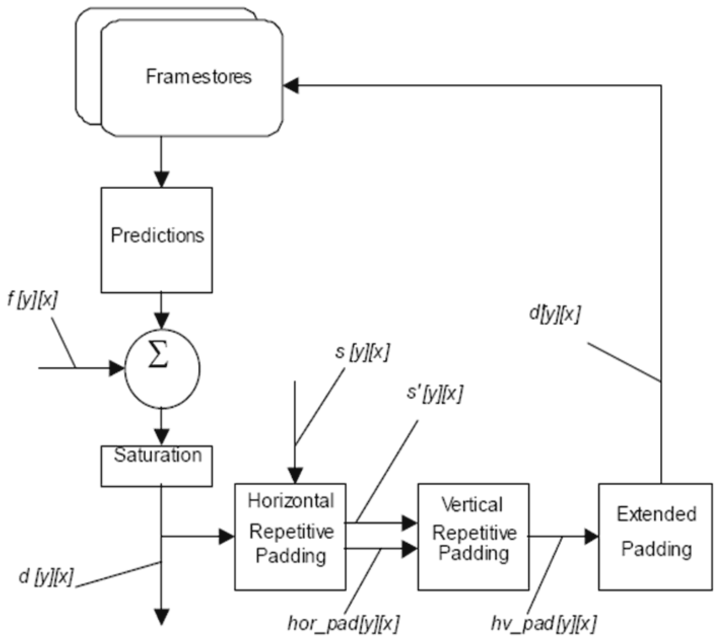


Figure 2.8: Simplified padding process (from [5]).

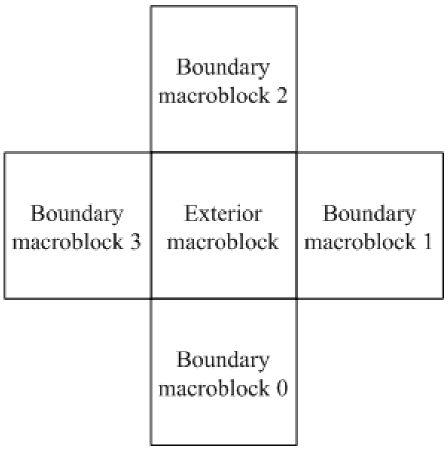


Figure 2.9: Priority of boundary MBs surrounding an exterior MB (from [5]).

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.

- 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. 2.9. The remaining exterior MBs (not located next to any boundary MBs) are filled with 128.

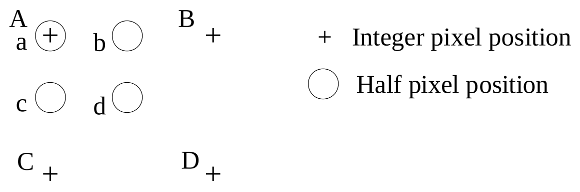
Motion Estimation

Motion estimation (ME) is a method of prediction between adjacent frames/pkctures. In general, the ME techniques used in MPEG-4 can be seen as an extension of standard MPEG-1/2 or H.263 block matching techniques with modified block (polygon) matching to handle arbitrary-shaped VOPs which is block-based method.

For an arbitrary shape VOP, the bounded 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. In many coding software implementations, the motion estimation is performed by full search to integer pixel accuracy vector and, using it as the initial estimate, a half pixel search is performed around it. Interpolation of MB is necessary because the motion vector may be non-integer. Fig. 2.10 illustrates the bilinear interpolation method.

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.



a = A,
b = (A + B + 1 - rounding_control) / 2
c = (A + C + 1 - rounding_control) / 2,
d = (A + B + C + D + 2 - rounding_control) / 4

Figure 2.10: Interpolation scheme for half sample search (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 (see Fig. 2.11). 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.

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

$$Px = Median(MV1x, MV2x, MV3x),$$

$$Py = Median(MV1y, MV2y, MV3y).$$

Then, the vector differences, $MVDx\ (= MVx - Px)$ and $MVDy\ (= MVy - Py)$, are coded by variable-length coding (VLC).

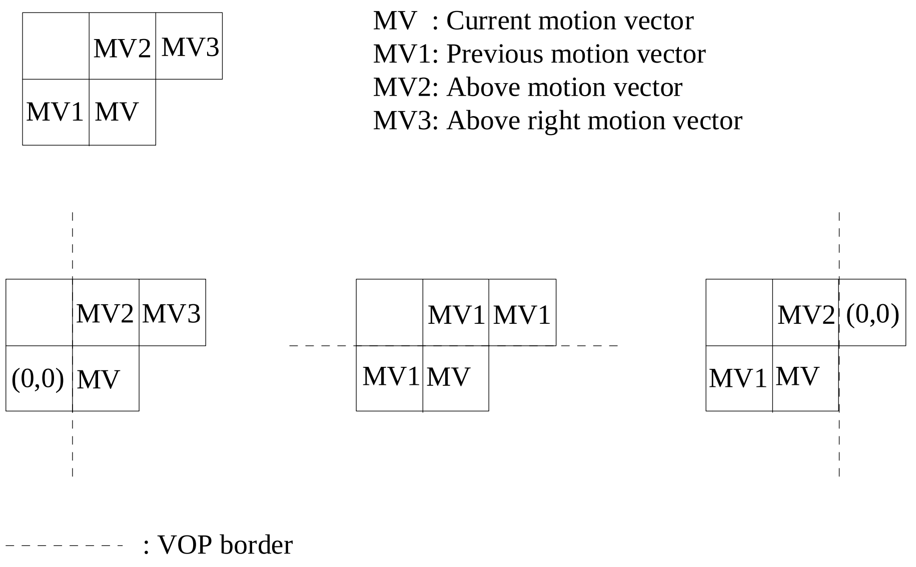


Figure 2.11: Motion vector prediction (from [8]).

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. The $pred[i][j]$ is defined as follows:

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

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

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. The creation of each pixel $\overline{P}(i, j)$, in an 8×8 luminance prediction block is governed by the following equation:

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

where (MV_x^0, MV_y^0) denotes the motion vector for the current block, (MV_x^1, MV_y^1) 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 value of each pixel in the current block and neighbor blocks.

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

2.2.4 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. The texture information is directly in the luminance and chrominance components for an I-VOP. However, for a P-VOP and a B-VOP, the texture information represents 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) discrete cosine transform (DCT), quantization, coefficient prediction, coefficient scan and variable length coding (VLC).

Padding Process

When the shape of the VOP is arbitrary, two types of MB exits, those that lie inside the VOP and those that lie on the boundary of the VOP. The MBs that lie completely inside the VOP are coded using a technique identical to the technique used in H.263. The MBs that lie on the boundary of the shape need to be padded before texture coding. For residual error blocks after motion compensation, the region outside the VOP within the blocks are padded with zero. For intra blocks, the padding is performed in a three-step procedure called low pass extrapolation (LPE). This procedure is as follows:

1. Compute the arithmetic mean value m of the pixels $f(i,j)$ in the blocks that belong to the VOP as
$$m = (1/N) \sum_{(i,j) \in VOP} f(i,j)$$
where N is the number of pixels situated with the VOP.

2. Assign m to each block pixel situated outside of the VOP region.

3. Apply the following filtering operation to each block pixel $f(i,j)$ outside of the VOP region, in raster-scan oeder:

$$f(i,j) = \frac{f(i,j-1) + f(i-1,j) + f(i,j+1) + f(i+1,j)}{4}.$$

If one or more of the four pixels used for filtering are outside the block, the corresponding pixels are not included into the filtering operation and the divisor 4 is reduced accordingly.

Discrete Cosine Transform (DCT) Coding

Similar to MPEG-1 and MPEG-2, the transform coding in the MPEG-4 standard is based on 2D 8$\times$8 DCT. Before quantization, the encoder does forward transform. Then the encoder does inverse transform after inverse quantization for reconstructing the VOP.

Quantization

MPEG-4 video supports two quantization techniques, one referred to as the H.263 quantization method and the other, the MPEG quantization method. The H.263 quantization method is with dead zone for intra and inter AC coefficients and with no dead zone for intra DC coefficients. The MPEG quantization method is uniform quantizer with the default matrix as shown in Table 2.3.

Figure 2.12 shows the quantizer characteristics in H.263. It has uniform quantization for intra DC coefficients and nearly uniform midtread quantization for the inter DC and all AC coefficients. All coefficients in a MB go through the same quantizer step size Q, which can be changed in increments of 2 from 2 to 62 as desired.

Furthermore, in order to provide a higher coding efficiency, Table 2.4 shows a nonlinear scaler which 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

When coding an intra block, the DC coefficients and many AC coefficients are coded by intra prediction. Intra prediction is an operation used in MPEG-4 standards to reduce the spatial redundancy between 8×8 blocks.

DC prediction is illustrated in Fig. 2.13. The quantized intra coefficients are predicted with three previous decoded DC coefficients. For example, the DC coefficients of block X

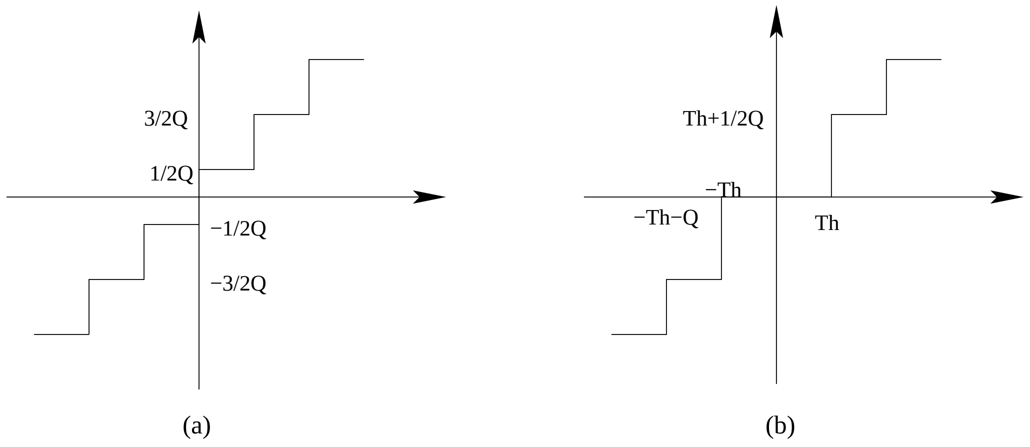


Figure 2.12: Quantizers in H.263. (a) For intra DC coefficient only. (b) For inter DC and all AC coefficients.

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	$(Q+13)/2$		$Q-16$

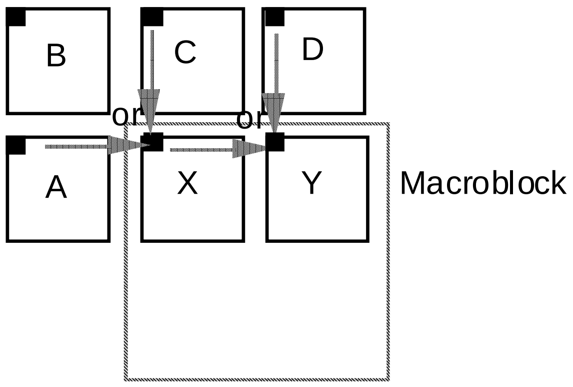


Figure 2.13: Prediction of DC coefficients of blocks in an intra MB (from [6]).

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. 2.14. 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.

Scan and VLC

Figure 2.15 shows three kinds of scan, alternate-horizontal, alternate-vertical and zigzag (the normal scan used in H.263 and MPEG-1), to scan the DC and AC coefficients and change the 2D block data to 1D data. The actual scan used depends on the coefficient prediction method used. If the direction is vertical, alternate-horizontal scan is used for the current block. If the direction is horizontal, alternate-vertical scan is selected for the current block. For all other blocks, zigzag scanned is used.

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 the MPEG-4 standard, differential DC coefficients in intra blocks are encoded in VLC. However, the AC coefficients are encoded by the variable length codes for EVENTs, where an EVENT consists of a last non-zero coefficient indication (LAST), the number of successive zeros preceding the

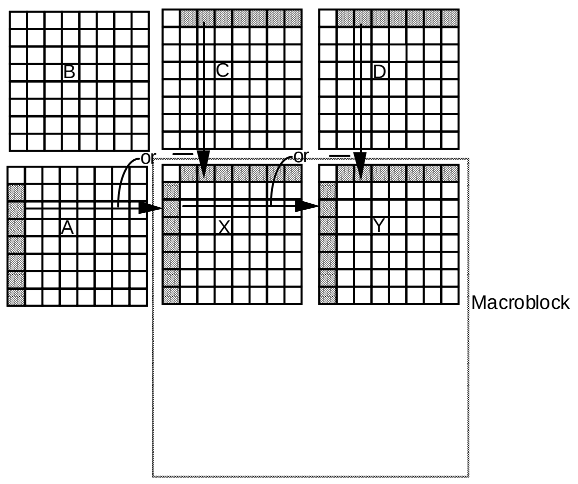
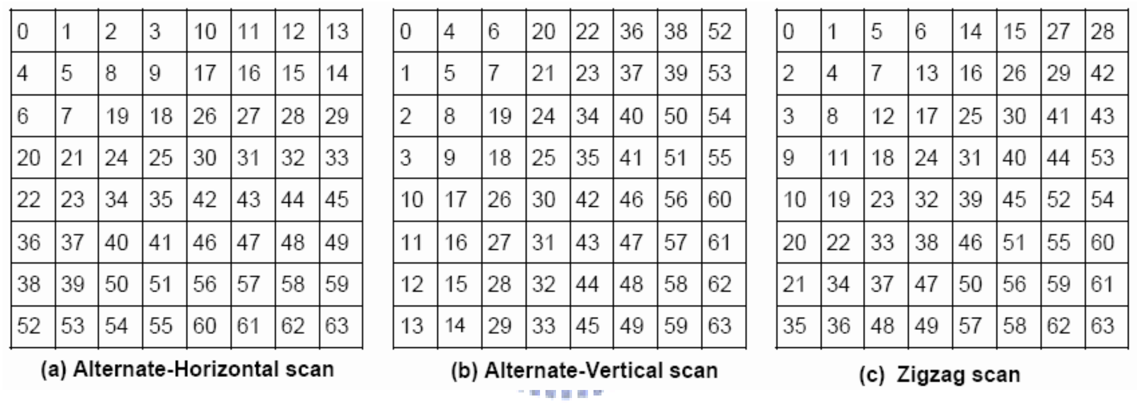


Figure 2.14: Prediction of AC coefficients of blocks in an intra MB (from [6]).



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

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.

2.2.5 Other Video Coding Tools [6]

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

Robust Video Coding

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

1. Object priorities

   The object based organization of MPEG-4 video facilitates prioritizing of the semantic objects based on their relevance. Further, the VOP types lend themselves to a form of automatic prioritization since, B-VOPs are noncausal and 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 reduce the problem. However, more 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. The Fine Granularity Scalability (FGS) which supports continuous scalability of bit rate and video quality is also defined.

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

Table 2.5: Profiles and Tools in MPEG-4 Video (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 [3]–[4].

3.1 Introduction

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

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

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

For high performance, the PACDSP is a VLIW processor with single instruction multiple data (SIMD) instruction set architecture (ISA). The software supported scheduling reduces hardware 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 and entries 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 computing power.
- Variable instruction word/packet length for compensating the drawback of poor code density in the VLIW architecture.
- Heterogeneous register files for more straightforward operations, less port number and smaller entries in each RF to improve the performance and reduce power and area.
- Constant register file in each cluster for the storage of fixed data used in the applications to reduce the frequency of data movement which may cost significant of power consumption.
- Inter-cluster communication by memory controller for reusing hardware resource and reducing the port number of ping-pong RF in order to reduce power and area and to increase the scalability.
- Optimized interrupt design with fast interrupt response time 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 FU interface that can be used to enhance DSP functionalities.

The architecture of the PACDSP v3.0 is shown in Fig. 3.1. The following sections will briefly introduce its pipeline stages and its core elements, including the Program Sequence Control Unit (PSCU), Scalar Unit, Clusters (VLIW datapath), and Customized Function Unit (CFU). 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.

3.2 ISA and Pipeline Stages

There are three major divisions in the PACDSP instruction set architecture (ISA): Program Sequence Control, Scalar and VLIW Data Path. In each division, the instructions are divided into categories by function units. Figure 3.2 depicts the ISA of the PACDSP.

Figure 3.3 shows the pipeline stages of PACDSP. The program sequence control unit operation can be divided into four stages, which are IF, IMEM, IDP, and ID. Scalar unit and VLIW datapath operation are both divided into five stages, namely RO, EX1, EX2, EX3, and WB. The job of each pipeline stage is shown in Table 3.1.

3.3 Program Sequence Control Unit

The program sequence control unit (PSCU) is a main component in the DSP kernel. Basically, we can regard it as the combination of the control path and the instruction path. The control path affects the program counter updating, address fetch, pipeline control, hardware context shadowing, interrupt handling, exception handling, etc., according to the input control signals from elsewhere in the PACDSP. In addition, the instruction path is responsible for fetching, dispatching, and decoding of the instruction packets.

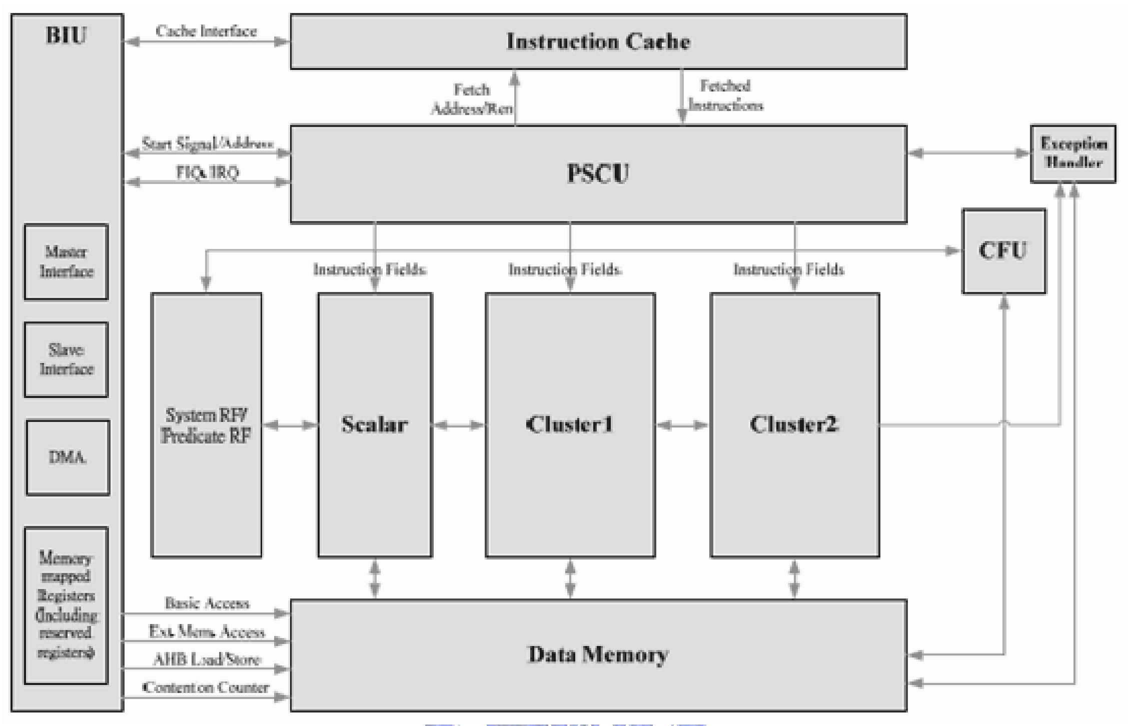


Figure 3.1: Architecture of the PACDSP [2].

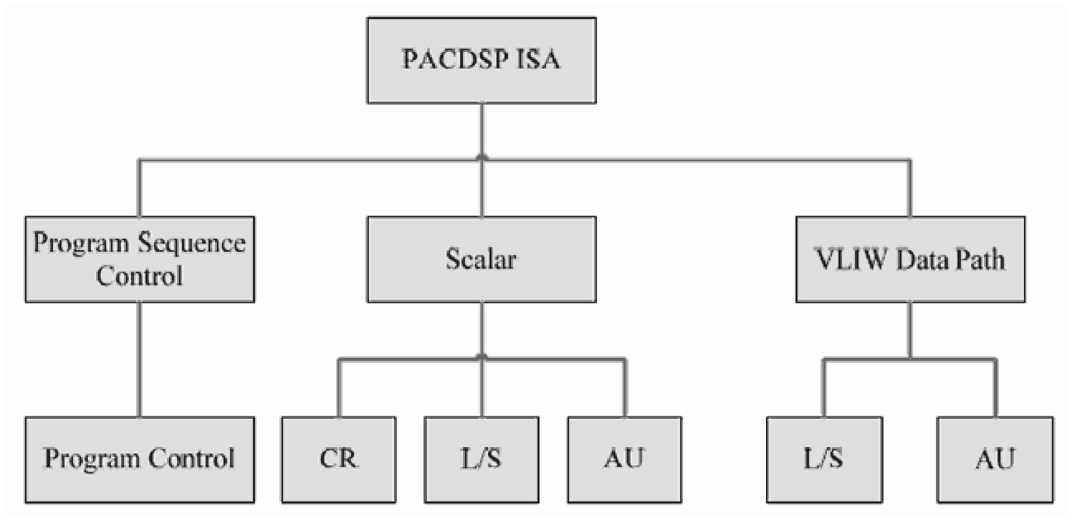


Figure 3.2: PACDSP instruction set architecture [4].

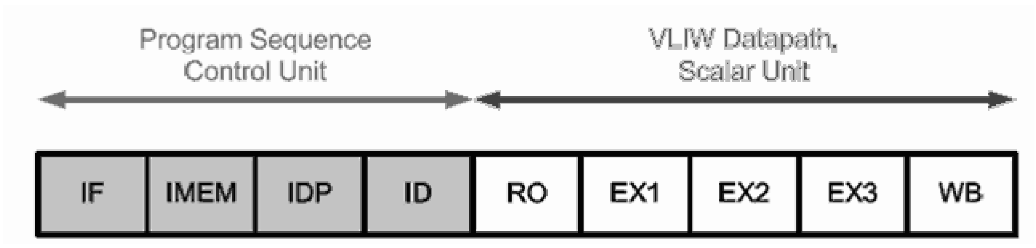


Figure 3.3: Pipeline stages of the PACDSP [4].

Table 3.1: Pipeline Stages and Their Jobs

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

3.3.1 Branch Instructions

Branch instructions can be grouped into two categories, conditional branches and unconditional branches. There are three addressing modes defined in the PACDSP v3.0 for generating the branch target address:

- PC-relative

  Add up to 32-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 Program Counter, and OFFSET is the immediate value defined in branch instruction.

- Register

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

  $$\text{TA} = \text{Rs}$$

  where TA is the target address and Rs is the source register defined in branch instruction.

- Register-relative

  Add up to 32-bit signed immediate offset to the address saved in the register and take the result as the branch target address, i.e.,

  $$\text{TA} = \text{Rs} + \text{OFFSET}$$

  where TA is the target address, Rs is the source register defined in branch instruction, and OFFSET is the immediate value defined in branch instruction.

In some circumstances, a branch operation may need to save the return address to ensure correct working of the program when it returns. 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 five branch delay slots in the PACDSP, and the programmer could put the branch-independent instructions in the delay slots.

There are some constraints about instructions in the delay slots. Reference [4] gives the details of the programming constraints.

3.3.2 Loops

The programmer can use the LBCB or B instruction to describe program loops. LBCB is similar to branch, but instead of checking a predicate register (P0–P15), LBCB checks a general purpose register (R0–R15) to decide whether to branch or not. Since there are 16 general purpose registers (R0–R15), up to 16 levels of nested loop can be supported with the use of the LBCB instruction.

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

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

3.3.3 Customized Function Units (CFUs)

The PACDSP provides Customized Function Unit Interface for extension usage. The user can attach co-processors or customized function units to PACDSP and handle them through the scalar instructions. If 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 given work has finished successfully, the PACDSP can use its results and continue to work. It is recommended that if a co-processor is used, communication with it be made through this interface, or the user will have to pay much more effort to handle it.

3.3.4 Exception Handling

Unpredictable exceptions may occur during program execution. The exceptions need to be handled correctly for correct execution results. Exceptions may be caused by hardware (e.g., overflow), software, internal (e.g., undefined instruction), or external (e.g., coprocessor exception). When an exception happens whether PACDSP is running a program or not, PACDSP will check for mask information. If the exception is masked, PACDSP will ignore the exception and return to normal execution. If the exception is unmasked, it will be taken. PACDSP will freeze its pipeline, finish the instructions before the PC which introduced the exception, and recover the states for consistence. After the state is recovered, PACDSP will issue exception handling ISR to inform the MPU and the Embedded ICE, waiting for different commands to resolve the exception.

3.3.5 Interrupt Handling

Two types of interrupt are supported by the PACDSP. One is fast interrupt request (FIQ), which has the higher priority, and the second is interrupt request (IRQ). The difference between them is that the FIQ has fixed ISR address and IRQ needs ISR to check the IRQ source to obtain the proper ISR address.

In the PACDSP, the minimum latency from interrupt request to the first ISR instruction to be executed is 4 cycles for both types of interrupt, and it may be postponed when the ISR experiences cache miss.

3.4 Scalar Unit

The scalar unit plays an important role in handling control-based task for PACDSP. It also has a simple capacity for data computing. Thus, the scalar unit is like a RISC machine. Programmers can exploit computing capacity of the scalar unit to increase overall instruction-level parallelism (ILP) in compute-based task.

The scalar unit mainly consists of one adder, one down-counter, one comparator, one shifter and one logical ALU. The scalar unit has four major functions as follows:

- Program flow control function.
- Data processing function.
- Memory access function.
- Data transfer function.

3.4.1 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. These registers are viewed as the loop boundary counter, the timer and the address register in the LBCB, WAIT and Branch/Load/Store instructions, respectively. In other instructions, they are viewed as data registers.

3.4.2 System Register and Predication Register

There are 16 system registers named as SR0 to SR15 in PACDSP. Table 3.2 shows the names, the widths, the meaning of all the system registers in PACDSP. Note that each bit in SR0 is used as the predication register and are named P0 to P15, where the value of P0 is always true. Most instructions of PACDSP can be executed conditionally according to the values of predication registers.

3.5 VLIW Datapath

As shown in Fig. 3.4, the VLIW datapath of PACDSP is constructed with distributed register file: ping-pong registers, accumulator registers, address registers, constant registers and some control flags.

If the instruction must write into two consecutive destination registers, for example, DLW and FMUL.D, the destination register number has to be even because of banked structure.

The VLIW datapath of PACDSP is constructed in two clusters, and each contains an arithmetic unit (AU) and a load/store unit (L/S) as shown in Fig. 3.5. Therefore, it can execute four instructions simultaneously, and is thus called a four-way VLIW datapath. The VLIW datapath supports SIMD (Single Instruction Multiple Data) operation. It executes in three modes: Single (32-bit or 40-bit), Dual (16-bit) and Quad (8-bit). There are also three types of precision in the datapath of PACDSP: Full, Integer and Fractional.

Arithmetic Unit (AU)

The arithmetic unit comprises 40-bit modules which are divided according to functions. The function types supported by the AU are shown below:

- Arithmetic and comparison instructions.
- Data transfer instructions.
- Bit manipulation instructions.

Table 3.2: System Register File [2]

No	Name	Size(bits)	Note
SR0	PREDN	16	Predication information
SR1	EN_INT	1	Interrupt enable flag
SR2	MSK_EXC	16	Mask inside exception
SR3	SWI_EXC	16	Software exception
SR4	CF0	32	Custom function register 0
SR5	CF1	32	Custom function register 1
SR6	CF2	32	Custom function register 2
SR7	CF3	32	Custom function register 3
SR8	SD_Status	8	Mix information 0's shadow register
SR9	SD_CPC	32	CPC's shadow register (ISR return address)
SR10	SD_BCTG	32	Branch target's shadow register
SR11	SD_R0	32	R0's shadow register
SR12	Mode	4	Power mode register
SR13	CFU_Info_Sel	4	CFU_Info select register
SR14	EXC_Cause	16	Exception cause
SR15	Reserved	32	N.A.

- Multiplication and accumulation instructions.
- Special instructions.

All data processing instructions in AU begin at the same stage but not finish at the same time due to different computing complexity.

Load/Store Unit (L/S)

The load/store unit (L/S) comprises 32-bit modules except for one 16-bit address generation unit (AGU) which is used to support the different addressing modes. The functional types supported by L/S are as follows:

- Arithmetic and comparison instructions.

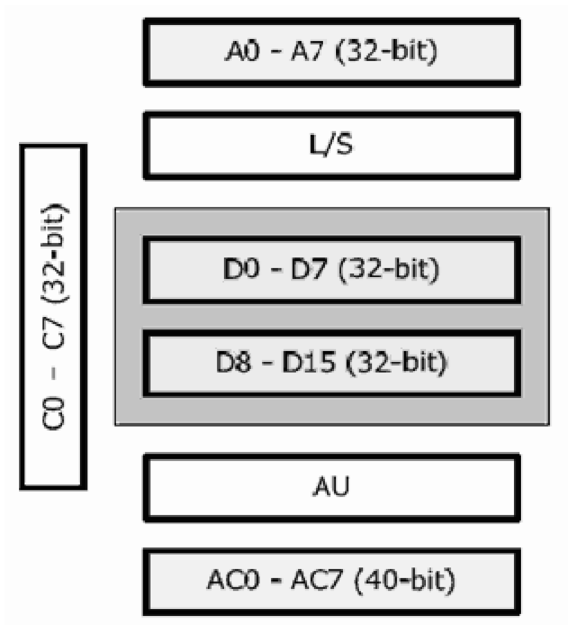


Figure 3.4: The VLIW datapath register organization [2].

- Data transfer instructions.
- Bit manipulation instructions.
- Load and store instructions.
- Special instructions.

Like AU, all instructions in L/S begin at the same stage but not finish at the same time due to different computing complexity.

The L/S unit supports powerful double load/store instructions, which can load or store two operands in one instruction. It also supports instructions that load and store by bytes or half-words. These instructions make memory access easier and more convenient.

3.5.1 Ping-Pong Register File

The ping-pong register file contains sixteen 32-bit registers which are divided into two groups: D0–D7 and D8–D15. The AU and the L/S unit can access the ping-pong register file at the same time but it has to be in different groups. In other words, both units

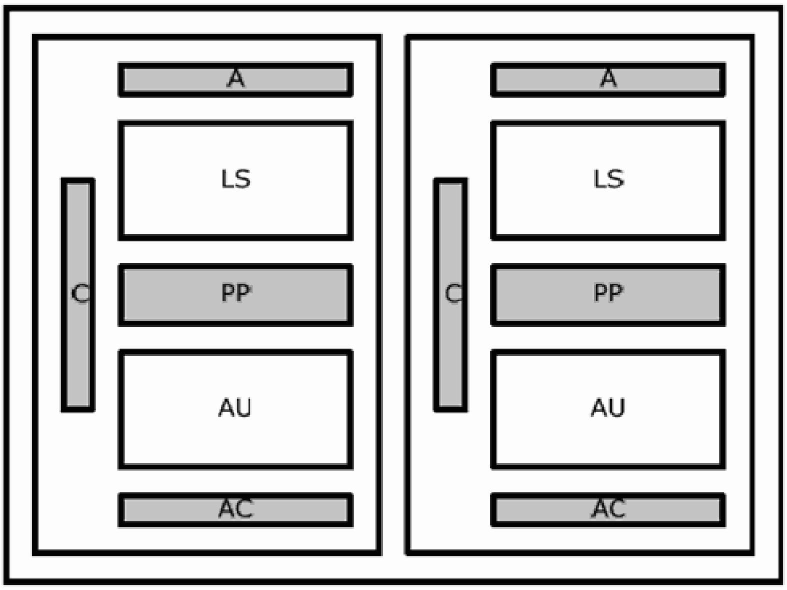


Figure 3.5: The four-way VLIW datapath of PACDSP [2].

cannot read or write the same group simultaneously. All possible access conditions are as follows:

- LS reads D0–D7 and writes D0–D7, and AU reads D8–D15 and writes D8–D15.
- LS reads D0–D7 and writes D8–D15, and AU reads D8–D15 and writes D0–D7.
- LS reads D8–D15 and writes D0–D7, and AU reads D0–D7 and writes D8–D15.
- LS reads D8–D15 and writes D8–D15, and AU reads D0–D7 and writes D0–D7.

3.5.2 Address/Accumulator Registers

As shown in Fig. 3.4, the address registers (A0–A7) are all 32-bit and they are dedicated to the load/store (L/S) unit for memory accesses. PACDSP supports several addressing modes. In modulo addressing mode, A0 and A2 are treated as pointers. A1 and A3 contain base addresses. A4 and A6 contain the values of end address plus one. A5 and A7 are treated as displacements. So it can support two groups of modulo addressing: (A0,A1,A4,A5) and (A2,A3,A6,A7). In other addressing modes, they can be used as address storage or data processing storage according to the design of the user.

The accumulator registers (AC0–AC7) are 40-bit registers which are dedicated to the arithmetic unit (AU) for data manipulations. The most significant eight bits are guard bits for accumulation operations.

3.5.3 Constant Registers

To avoid high frequency of data movement in the register file, PACDSP provides a small constant register file to keep fixed data. The constant register file has eight 32-bit registers (C0–C7). They can be read as either the first operand or the second operand in instructions that use them. But one instruction cannot access the constant register file as both of its source operands simultaneously.

The constant register file can be read by both the AU and the L/S unit but can only be written by the L/S unit. All accesses to the constant register file must be pointed by the control flags CF0 and CF1, which are pointers to the constant registers. And they are calculated from the values contained in CF2 and CF3, which are the contents of the pointers.

3.5.4 Status and Control Registers

The status register and control register can be used to monitor the DSP kernel status and handle the operation mode of the DSP kernel.

Program Status Register

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

Addressing Mode Control Register (AMCR)

There are several addressing modes supported by PACDSP. The addressing mode control register (AMCR) is a 16-bit register. This register is used to set the addressing mode for each address register. The addressing modes are related to where the operands are to be found and how the address calculations are to be made. The definitions are shown in Table 3.3.

3.5.5 Addressing Modes

The addressing modes are related to where the operands are to be found and how the address calculations are to be made. PACDSP supports Linear Addressing Mode, Bit-Reverse Addressing Mode, and Modulo Addressing Mode for memory access. They can be altered by setting the AMCR. Table 3.4 shows the syntax of addressing modes that be used and the associated supporting units.

Fig. 3.6 shows that A0–A7 are the address register file and they are classified into even and odd banks in linear and bit-reverse addressing modes. Some addressing modes use 2 address registers, RsA and RsB, at the same time. They must be consecutive registers with RsA in the even bank and RsB in the odd bank.

Linear Addressing Mode

- Offset by immediate (RsA, displacement)
  The operand address is the sum of the contents of the address register (RsA) and the displacement (up to 24-bit signed integer, but the value range depends on the implementation of data memory).

- Offset by register (RsA, RsB)
  The operand address is the sum of the contents of the address register (RsA) and the contents of the address register (RsB).

Table 3.3: Definitions of AMCR (from [2])

AM[1]	AM[0]	Addressing Mode
0	0	Linear
0	1	Bit-reversed
1	0	Modulo
1	1	Reversed

Table 3.4: Syntax of Address Modes and Supporting Units [3]

Addressing Mode	Syntax	Support Unit	
1. Linear		Scalar	Cluster
Offset by Immediate	RsA, displacement	V	V
Offset by Register	RsA, RsB	V	V
Post-increment by Immediate	RsA, displacement+	V	V
Post-increment by Register	RsA, RsB+	V	V
2. Modulo		Scalar	Cluster
Post-increment by Register	RsA, RsB+	-	V
Post-increment by Immediate	RsA, displacement+	-	V
3. Bit-Reverse		Scalar	Cluster
Post-increment by Immediate	RsA, displacement+	-	V
Post-increment by Register	RsA, RsB+	-	V

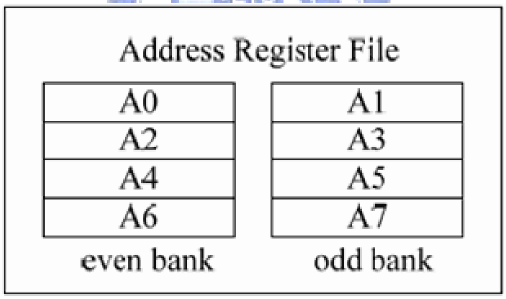


Figure 3.6: Address register file [2].

- Post-increment by immediate (RsA, displacement+)

  The operand address is in the address register RsA. After the operand address is used, it is incremented by the displacement (up to 24-bit signed integer, but the value range depends on the implementation of data memory) and stored in the same address register.

- Post-increment by register (RsA, RsB+)

  The operand address is in the address register RsA. After the operand address is used, it is incremented by the register (RsB) and stored in the same address register.

Bit-Reverse Addressing Mode

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

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

Modulo Addressing Mode

Modulo address modification is useful for creating circular buffers for FIFO queues, delay lines, and sample buffers. This addressing mode only supports post-increment by immediate and post-increment by register. The definition of modulo addressing, using a base register (Bn) and a end register (En), enables the programmer to locate the modulo buffer at any address. The current address register, An, can initially point anywhere (aligned to its access width) within the defined modulo address range, $Bn \leq An < En$.

Modulo addressing can be selected by configuring corresponding bits in AMCR. The range of modulo registers is from 1 to $2^{16} - 1$.

3.5.6 Data Communication

The PACDSP provides fast data communication mechanism among scalar unit and two clusters. As shown in Fig. 3.7, it provides a data exchange mechanism between any two of the scalar unit and the two clusters. Figure 3.8 shows that it can also provide data broadcast to facilitate one of them to broadcast its data to the others. 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. It only needs one instruction latency.

Data Exchanges

We can use the instruction DEX to exchange 32-bit data between any two units. Or we can use the instruction DDEX to exchange 64-bit data between the L/S units in two clusters.

Data Broadcast

We can use the instruction pair BDT and BDR to broadcast 32-bit data from one unit to the others. Or we can use the instruction pair DBDT and DBDR to translate 64-bit data between two clusters.

3.6 Conditional Execution Control

A DSP processor is focused on the 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.

All the PACDSP instructions are conditional, except TRAP, ROE, WAIT, TEST and LBCB. If a instruction is conditionally executed, the predicates referred to will be read in the RO (read operand) stage.

The compare-and-set instructions, such as SLT, SGT, etc., compare source operands and save the results to the predicate registers, and the comparison results can be saved

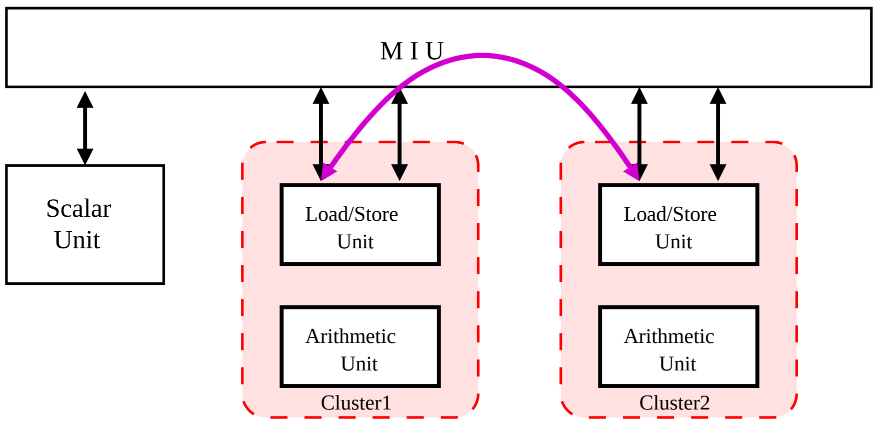


Figure 3.7: Data exchange between two clusters [2].

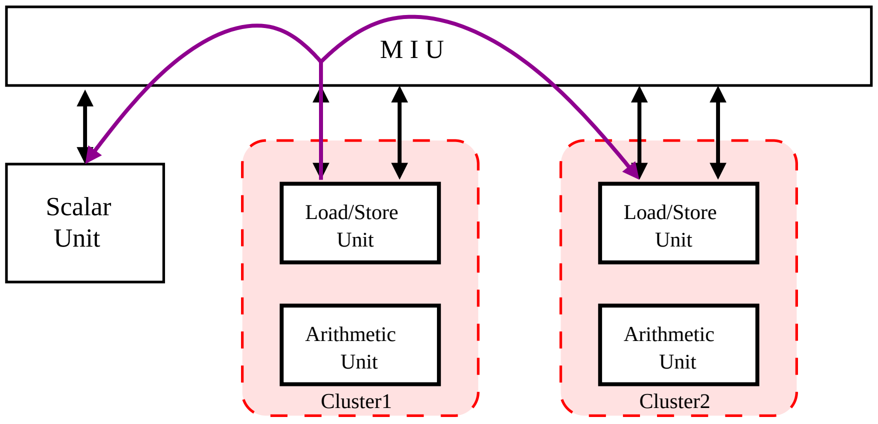


Figure 3.8: Data broadcast among clusters [2].

to the general purpose registers at the same time. For compiler friendliness, PACDSP saves both positive and negative boolean results for the compare-and-set instructions concurrently. However, P0 is always set to 1, and each predicate bit can be set by only one instruction at the same time.

3.7 Instruction Packet

PACDSP v3.0 can process at most five instructions concurrently. 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.

An instruction packet is enclosed in a pair of braces and can be expressed in either the horizontal or the vertical format. Figure 3.9 shows the syntax of a complete instruc-

Table 3.5: Instruction Type in Each Instruction Slot

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

tion packet. However, an instruction packet is allowed to be written in a single line and separated by a pipe character "|". The simplified syntax is shown in Fig. 3.10. A NOP instruction should be placed in the slot where there is no instruction to be executed.

3.8 DSP Running Modes

The PACDSP can work under various running modes. Each mode has different hardware utilization. We can change the running modes using the assembly instructions. Table 3.6 lists the running modes and the corresponding hardware resource.

3.9 Versions of PACDSP

PACDSP v3.0 is the latest version of PACDSP as of April 2007. The former version is PACDSP v2.0 whose chip was taped out in August 2006. We briefly introduce the features added from v2.0 to v3.0 and the differences between them in this section.

3.9.1 Pipeline Stages

Figure 3.11 shows the pipeline stages of PACDSP v2.0. The PSCU of PACDSP v2.0 is divided into three stages, which are IF, IDP and ID. Compared to Fig. 3.2, we see that PACDSP v3.0 divides the PSCU into four stages, where the added stage, IMEM, accesses the instruction memory after the IF stage.

3.9.2 Program Sequence Control Unit (PSCU)

A brief list of the differences between PACDSP v2.0 and v3.0 in PSCU is as follows:

- There are 5 branch delay slots in PACDSP v3.0, while only 3 in PACDSP v2.0.
- PACDSP v3.0 uses general purpose registers (R0–R15) to record the loop counts. Up to 16 levels of nested loop can be supported with the use of the LBCB instruction. In PACDSP v2.0, loop boundary registers (RBC0–RBC3) are used to record

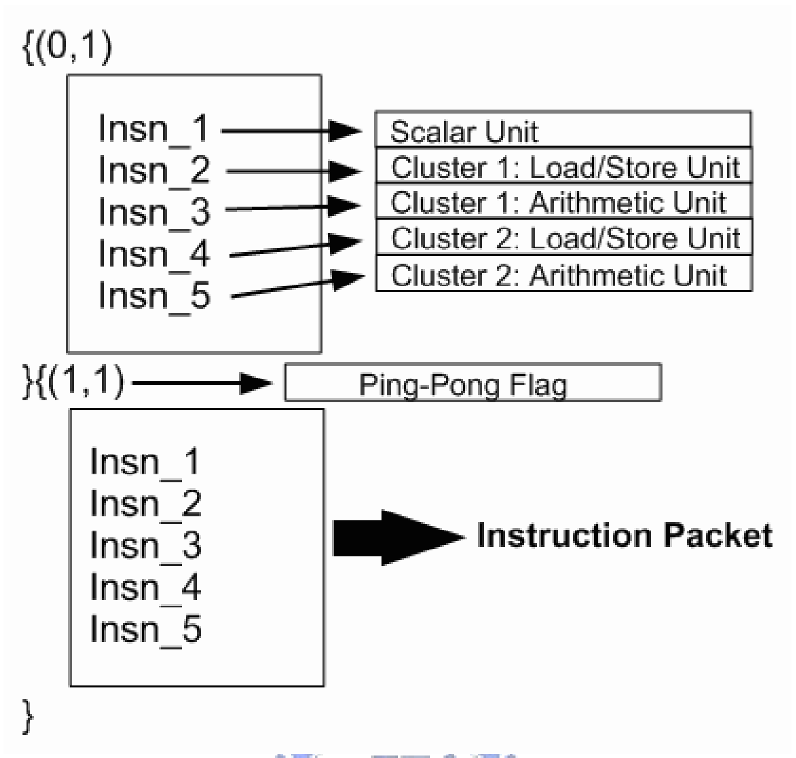


Figure 3.9: Syntax of instruction packet [3].

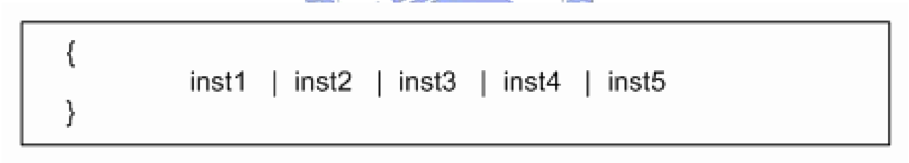


Figure 3.10: Simplified syntax of instruction packet [3].

the loop counts, and only up to 4 levels of nested loop can be supported with the use of the LBCB instruction.

- Compared to PACDSP v2.0, PACDSP v3.0 simplifies the scenarios of interrupt, debug, and exception.

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

Table 3.6: Running Modes of the PACDSP v3.0 [2]

Running Modes	Description	Resources	Binary Value
High Performance	Process performance-oriented programs which need all resource for high performance	All instruction slots are available	0x0
Medium Performance	Process programs which only need partial resource to achieve performance constraints	Scalar and Cluster 1 instruction slots are available	0x2
High power saving	Process power-oriented programs which care power consumption more than performance	Only Scalar instruction slot is available	0x3

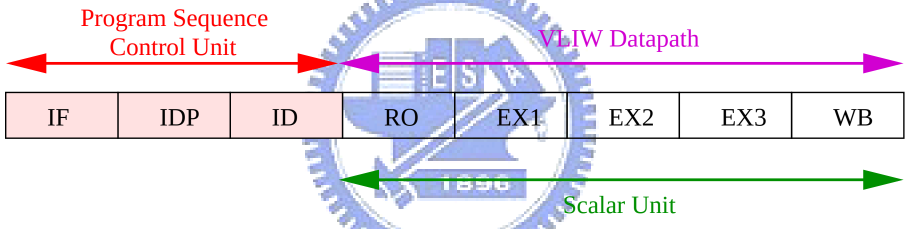


Figure 3.11: Pipeline stages of PACDSP v2.0 [3].

3.9.3 VLIW Data Path

- In PACDSP v3.0, the comparison instructions can be processed in both AU and L/S unit. But in PACDSP v2.0, only the L/S unit can process the comparison instructions.
- The inter-cluster communication latency is 2 cycles in PACDSP v2.0. PACDSP v3.0 decreases the latency to 1 cycle.
- PACDSP v3.0 adds register relative addressing mode for L/S instructions.
- The addressing mode control register (AMCR) is 32-bit in PACDSP v2.0. In PACDSP v3.0, it is a 16-bit register.

- In PACDSP v2.0, the constant register file contains sixteen 32-bit registers (C0–C15), while in PACDSP v3.0 it only has eight 32-bit registers (C0–C7).
- PACDSP v3.0 supports pointer addressing for the constant register file.

3.9.4 Conditional Execution Control

In PACDSP v2.0, the comparison instructions read the comparing sources in the RO pipeline stage, and the comparison is performed in the EX1 pipeline stage. The compared result is valid in the EX2 stage. Some program sequence control instructions, such as the branch instruction, can be conditionally executed. Their referred predicate registers are read in the ID stage. If a VLIW instruction is conditionally executed, the referred predicate register will be read in the RO stage.

Comparatively, in PACDSP v3.0, the results of comparison instructions are valid in the EX1 stage. Both program sequence control instructions and VLIW instructions can be conditionally executed, where the referred conditional predicates are read in the RO stage.

3.9.5 Instruction Set

Compared to PACDSP v2.0, PACDSP v3.0 adds some useful instructions and has enhanced some commonly used instructions. Table 3.7 shows the modified Load/Store instructions from PACDSP v2.0 to PACDSP v3.0 and their supporting units. Table 3.8 lists the comparison instructions supported by PACDSP v2.0 and PACDSP v3.0.

3.10 Dual-Core Platform and the Tool Chains

We focus on the introduction of PACDSP v3.0 in previous sections. In this section, the dual-core platform where we will demonstrate our implementation will be simplified introduced.

The dual-core platform is developed by SoC Technology Center (STC) of Industrial Technology Research Institute (ITRI). The PAC system consists of following items:

Table 3.7: Modified 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	

- ARM Integrator-compatible Core Module: ARM926EJ-S
- Multi-ICE of ARM
- PACDSP v3.0 Core Module (Burned in XILINX FPGA now)
- Generic peripherals (LCD translator in VGA size)

The dual-core platform is shown in Fig 3.12. The operation of PACDSP is controlled by the ARM core, and its internal memory is accessible to the ARM core as well. For a

Table 3.8: Comparison Instructions Supported by 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)

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.

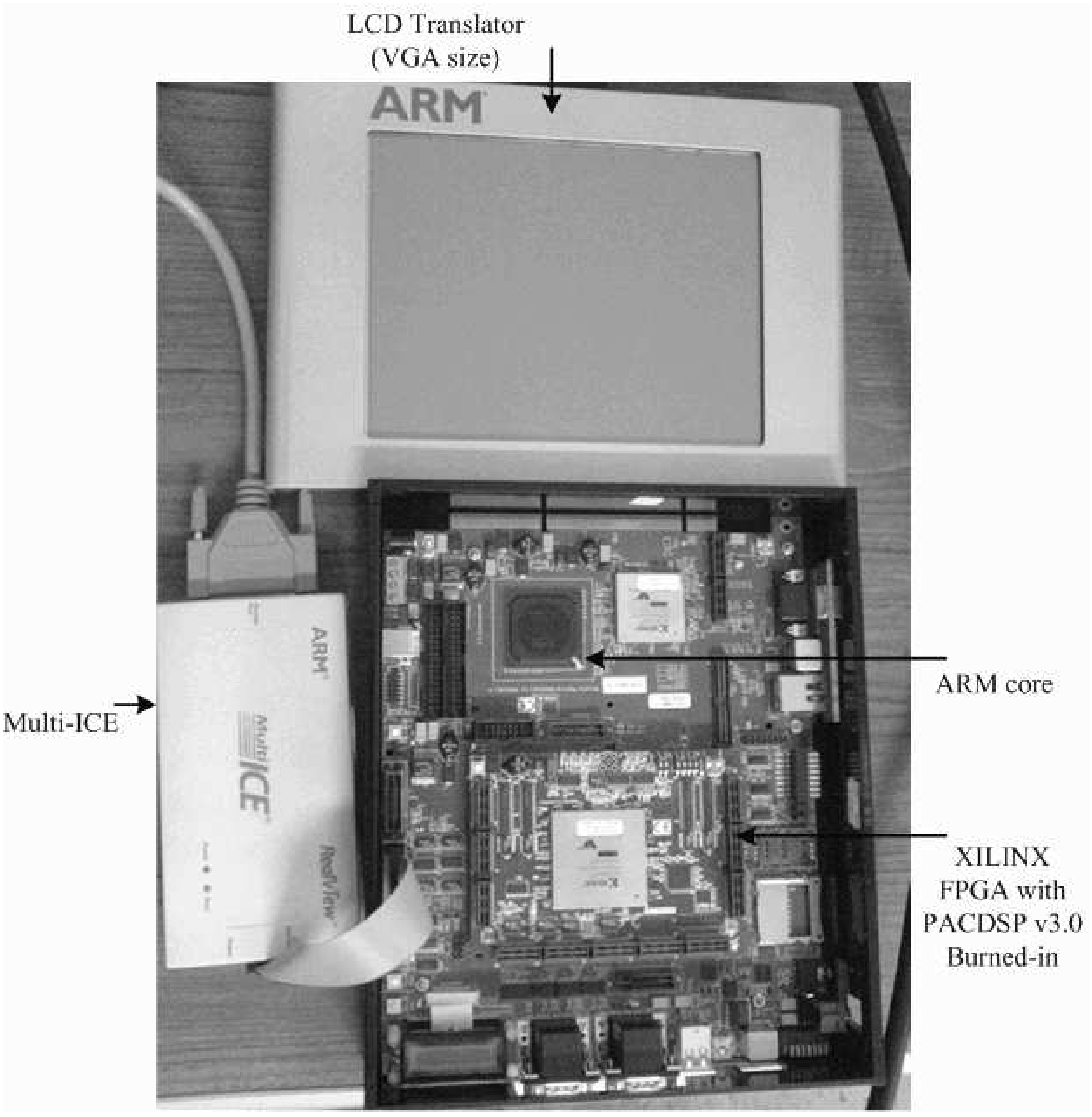


Figure 3.12: Dual-Core platform of PAC v3.0 system.

Chapter 4

C Code Development and System Design

Figure 4.1 shows the flow of our program development for the video encoder on PACDSP. In section 1, we introduce the first phase of the flow and present the profile. In sections 2–4, we analyze the computational complexity of the motion coder, shape coder, and texture coder, respectively, and discuss our approaches to algorithm optimization. And lastly in section 5, we give the implementation strategies on PACDSP. The third phase of the work, namely assembly codes optimization, is discussed in the next chapter.

4.1 Initial Code Development

Our C code development employs the public source MoMySys (Mobile Multimedia Systems) as the base [9]. The MoMuSys donated its software for MPEG-4 main profile encoding and decoding to MPEG. However, to implement an MPEG-4 encoder on the PACDSP platform, the main profile appears too complicated on first attempt. Therefore, we implement the simple profile plus binary shape coding without error resilience. Table 4.1 shows the functionalities that our implementation support.

Table 4.1: Functionalities of Our Implementation

	Simple	Main	Our Implementation
Basic *1. I VOP* *2. P VOP* *3. AC/DC Prediction* *4. 4MV Unrestricted MV*	V	V	V
Error resilience *1. Slice Resynchronization* *2. Data Partitioning* *3. Reversible VLC*	V	V	
Short header	V	V	
B-VOP		V	
Method 1/Method 2 quantization		V	
P-VOP based temporal scalability *1. Rectangular* *2. Arbitrary shape*		V	
Binary Shape		V	V
Grey shape		V	
Rate control		V	

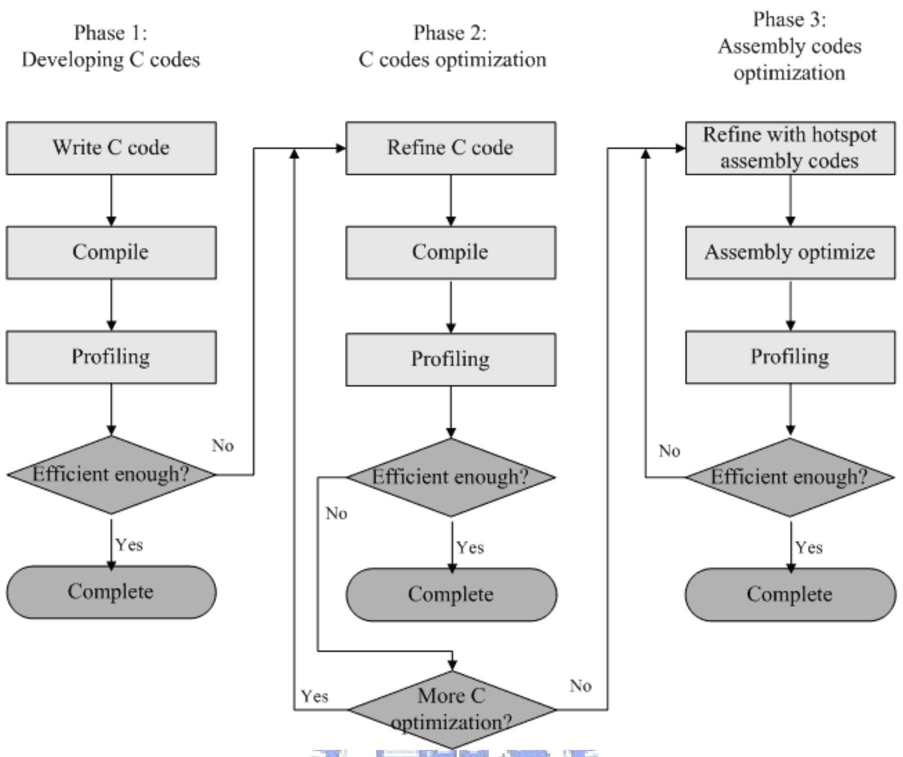


Figure 4.1: Flow of software development.

4.1.1 Profile Using the Profiler of ADS

To capture the complexity variation over different video material, we consider several common test video sequences of different amount of motion that likely represent the type of material the PACDSP platform will largely address in its video coding applications for some years. They are the QCIF (176×144) “foreman,” “akiyo,” and “stefan” sequences.

We employ the profile tools of ADS (ARM Developer Suite) to do the first level analysis, where ADS is the development tools for ARM processors. The profiling results, in Tables 4.2 and 4.3, are obtained from encoding an I-VOP and a P-VOP, respectively. We employ H.263 quantization with a fixed quantization step (QP), 4. Note that the quantization step size affects the length of bitstreams, so larger QP results in shorter bitstream and reduces the required encoding time.

The execution clockticks of the motion coder, the shape coder, and the texture coder

Table 4.2: Profile of Object-Based MPEG-4 Encoding of QCIF I-VOP on ADS

	foreman_qcif		akiyo_qcif		stefan_qcif	
Function Name	Clockticks	%	Clockticks	%	Clockticks	%
TextureCoding	*41,409,898*	*78.25*	*42,557,578*	*74.24*	*12,047,483*	*62.22*
BlockDCT	17,368,343	32.82	17,867,992	31.17	4,798,081	24.78
BlockIDCT	18,156,851	34.31	18,452,700	32.19	5,212,443	26.92
ShapeCoding	*3,958,416*	*7.48*	*3,674,489*	*6.41*	*2,228,649*	*11.51*
CAE_MB	2,799,468	5.29	2,505,073	4.37	1,547,081	7.99
Others	*7,551,684*	*14.27*	*11,092,257*	*19.35*	*5,086,585*	*26.27*
Total	*52,919,998*	*100.00*	*57,324,324*	*100.00*	*19,362,717*	*100.00*

are denoted as “MotionEstimation,” “ShapeCoding,” and “TextureCoding,” respectively, in Tables 4.2 and 4.3. The execution clockticks of the critical functions belonging to each are also shown in the tables. Besides the three coders, the remaining execution clockticks are included in “Others,” which contains VOP formation, writing header bitstream, VOP padding, etc.

In I-VOP encoding, we can find in Table 4.2 that the most time-critical components are “BlockDCT” and “BlockIDCT” of “TextureCoding.” The reason why DCT and IDCT consume so much time is that the DCT and IDCT in the reference code is implemented in floating-point. Moreover, the function “CAE_MB,” which does the context-based arithmetic coding for binary alpha blocks, is an important part of “ShapeCoding.”

As we can see in Table 4.3, most computation is spent on the functions related to motion estimation, which occupies about 40% to 50% of the execution time in P-VOP encoding. Since the mode “inter MC” is added to “ShapeCoding” of P-VOP encoding, the function “ShapeInterMB,” which finds the best matching of binary alpha block, is another time-critical function.

In the object-based video encoder, the VOP size is arbitrary in each frame. Among the three test sequences, “akiyo_qcif” has the biggest VOP size, “foreman_qcif” the sec-

Table 4.3: Profile of Object-Based MPEG-4 Encoding of QCIF P-VOP on ADS

	foreman_qcif		akiyo_qcif		stefan_qcif	
Function Name	Clockticks	%	Clockticks	%	Clockticks	%
MotionEstimation	*79,675,422*	*50.20*	*48,952,190*	*45.19*	*24,251,478*	*41.60*
FullPelMotionEstMB	71,951,245	45.34	40,752,077	37.62	22,069,388	37.86
FindSubPel	7,703,016	4.85	8,183,547	7.55	2,174,324	3.73
TextureCoding	*37,139,540*	*23.40*	*38,101,536*	*35.17*	*10,856,089*	*18.62*
BlockDCT	16,004,337	10.08	15,611,662	14.41	4,768,944	8.18
BlockIDCT	16,252,774	10.24	16,749,806	15.46	4,564,249	7.83
ShapeCoding	*35,191,526*	*22.17*	*12,907,962*	*11.91*	*18,419,663*	*31.60*
ShapeInterMB	30,833,225	19.43	10,436,508	9.63	15,631,836	26.82
CAE_MB	3,231,739	2.04	1,636,398	1.51	2,351,171	4.03
Others	*6,694,822*	*4.22*	*8,372,839*	*7.73*	*4,764,480*	*8.17*
Total	*158,701,310*	*100.00*	*108,334,527*	*100.00*	*58,291,710*	*100.00*

ond, and “stefan_qcif” the smallest. Therefore, we see that the execution times of some functions for I-VOP encoding, such as DCT and IDCT, are proportional to the VOP size. For functions which only operate on boundary macroblocks, such as “CAE_MB,” the execution times are proportioned to the boundary MB counts. However, for the functions called for P-VOP encoding, 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, since the motion of this sequence is little, the execution times of the inter functions are less than “foreman_qcif” and even less than “stefan_qcif” sometimes.

Table 4.4: Major Function in Motion Estimation (ME)

Function Name	Execution Time Percentage in Total for ME		
	foreman_qcif	akiyo_qcif	stefan_qcif
Obtain_SR	0.40%	0.61%	0.26%
SAD_MB	81.65%	71.07%	83.38%
SAD_Block	3.16%	3.86%	2.43%
ChooseMode	0.53%	0.85%	0.48%
FindSubPel	9.67%	16.72%	7.78%
Others	4.59%	6.89%	5.67%

4.2 Motion Coder Analysis and Design

The functions related to motion estimation costs the most computation in P-VOP encoding. Hence our first target is to reduce the complexity of these functions. Table 4.4 summarizes the major functions in motion estimation and the percentage complexity of each function in the total for motion estimation.

The search method in the original reference software is full search in raster-scan order with check for early termination each row. The search range is $[-16,16)$, and the motion vector is specified to half-pixel accuracy. As we can see in Table 4.4, the most complex function is "SAD_MB," which is used to calculate the SAD (sum of absolute differences) in the 16×16 MB at integer pixel displacements. After searching for the MB motion vector, additional search is made for each 8×8 block. The integer block motion estimation uses the MB motion vector as the search center and the search range is ±2 pixels. "SAD_Block" is the function to calculate the SAD of an 8×8 block.

In this section, we introduce some methods to reduce the complexity of SAD calculation.

4.2.1 Modification of Search Order

In order to increase the probability of early termination, we try to use the spiral full search order to replace the original order in the reference software. Since, statistically, experience shows that most motions are within ±5 pixels, spiral search may reduce the complexity of SAD calculation by more frequent early termination. Figure 4.2 shows the concept of spiral search.

Table 4.5 shows the percentage of early termination in SAD calculation under two different scan orders: raster-scan order and spiral in order. Three test sequences of different motion characteristics are used here, and we have run 10 inter frames each.

4.2.2 Use of A Tier Parameter

As illustrated in Figure 4.2, the spiral search algorithm consists of a number of nested loops from tier0 to tier16. We consider using a tunable parameter, "tier_para," to reduce the complexity of SAD calculation by terminating the search procedure when we find a local minimum SAD. A termination test is added at the end of every tier's motion estimation. The modified flow of spiral search is illustrated in Fig. 4.3, and the parameters used are as follow:

- curr_tier is the tier number where the ME algorithm searches on,
- SAD_0 means the SAD of zero motion vector (the search point of tier_0),
- min_SAD records the minimum SAD value during the search procedure, and

Table 4.5: Percentage of Early Termination in SAD Calculation Under Different Scan Orders

Scan Order	foreman_qcif	akiyo_qcif	stefan_qcif
Raster-scan order	46.62%	55.33%	43.24%
Spiral order	66.00%	80.66%	60.37%

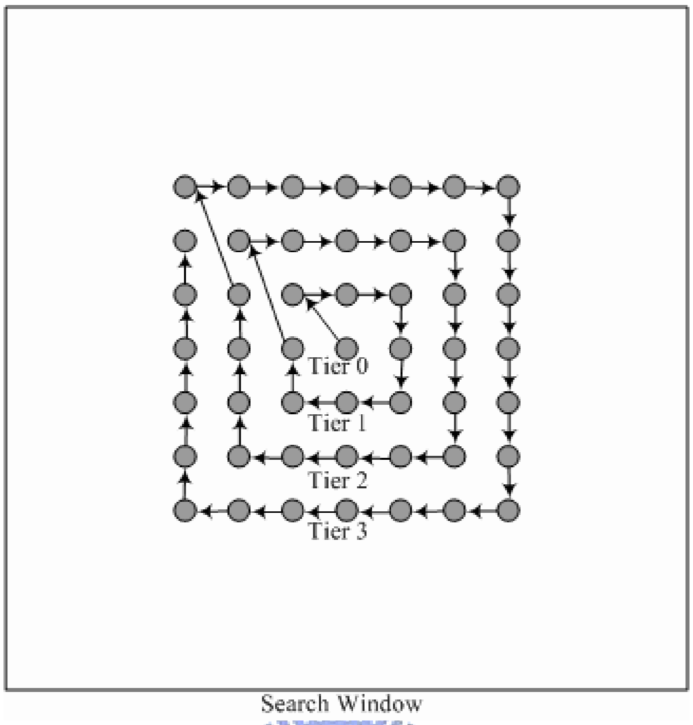


Figure 4.2: Concept of spiral search.

- min_tier records the tier number of the search point whose SAD value is minimum during the search procedure.

A "tier_para" value of N means that if we find a best match at tier m and the best match is unchanged between tier m and tier $m+N$, then we terminate the search procedure and take the best match as the search result. Note that when "tier_para" is equal to 16, the modified search procedure is equivalent to full spiral search.

The following results are obtained by encoding 10 P-frames for each sequence at a fixed quantization step size (QP) of 4. Figure 4.4 shows the early termination percentage of SAD calculation with different tier parameter values. The video quality is measured by PSNR (peak signal to noise ratio) and Fig. 4.5 shows the average PSNR in two cases: without residual coding and with residual coding. In the first case, without residual coding, we measure the PSNR using the motion compensated frames and the original frames. We see that, even with small "tier_para," the quality is still very close to full search. With residual coding, the quality loss is compensated by adding reconstructed residual. As illustrated in Fig. 4.5, the three curves for "with residual coding" are nearly horizontal.

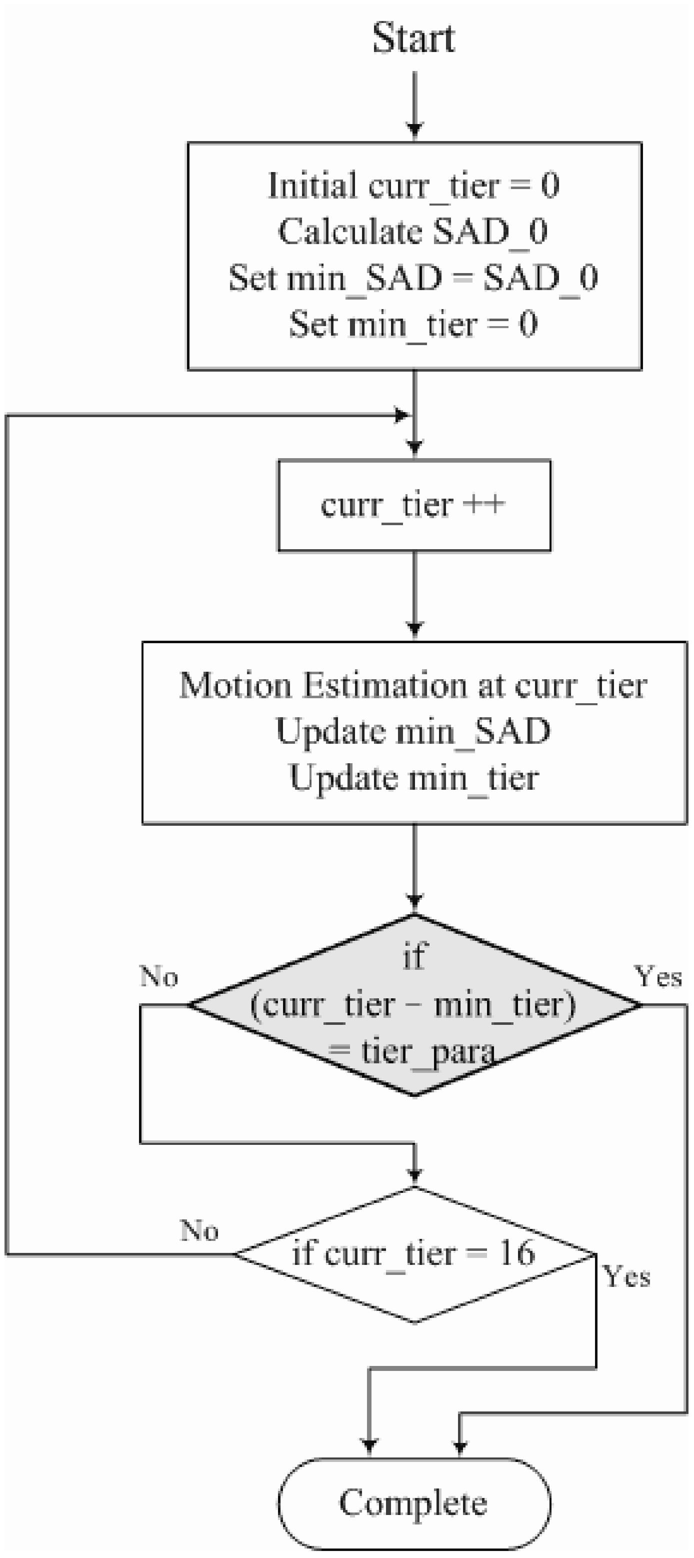


Figure 4.3: Dataflow of spiral search with tier parameter.

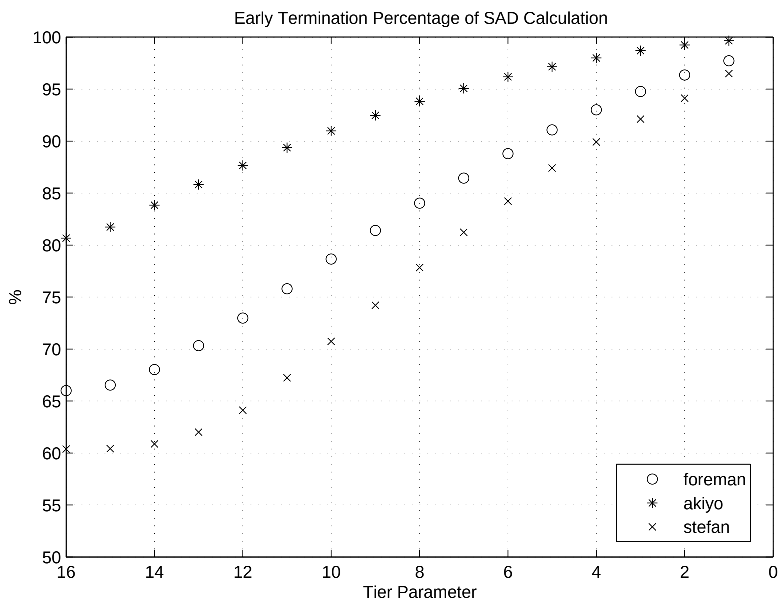


Figure 4.4: Early termination percentage of SAD calculation with different tier_para values.

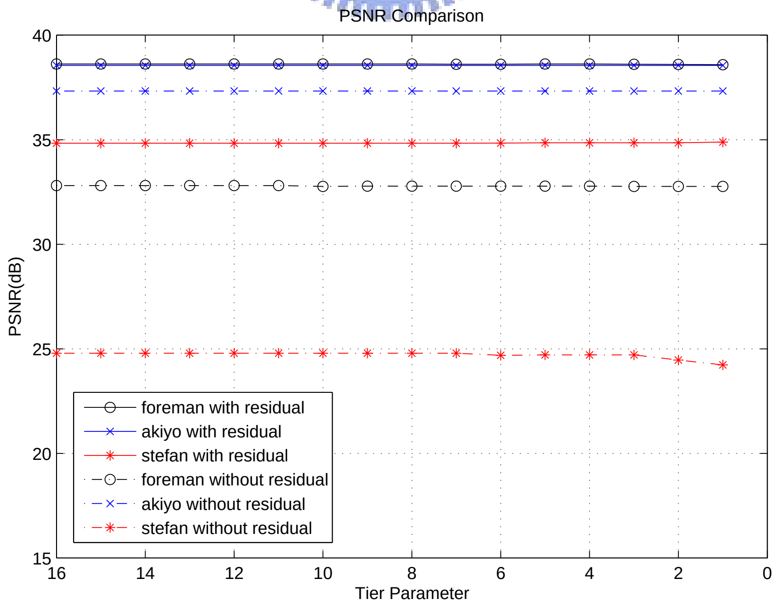


Figure 4.5: PSNR values with different tier_para values.

However, choosing too small a tier parameter may cause an originally inter-coded block to be coded in intra mode, which makes the related bit-rate rise. The amount of increase in bitrate is dependent on the QP of texture coding.

4.3 Shape Coder Analysis and Design

For lossless shape coding in our implementation, only four modes are supplied for context-based arithmetic encoding (CAE). Table 4.6 shows the four CAE modes and their supporting VOP. In I-VOP coding, only two modes are available for ShapeCoding, and Table 4.2 shows that most time of ShapeCoding are spend on CAE operation. However, all the four modes are available in P-VOP coding. As shown in Table 4.3, the function "ShapeInterMB" occupies about 10% to 30% (depending on motion characteristic) of the execution time in P-VOP encoding.

In this section, we firstly introduce a multi-symbol CAE algorithm to encode the shape information efficiently. Then, we show some methods to simplify the data flow of ShapeCoding in P-VOP coding.

4.3.1 Multi-Symbol CAE [10]

Since the CAE algorithm has a complicated coding procedure and strong data dependency, it is hard to exploit the parallel processing capability of PACDSP. In the reference code, the arithmetic coder encodes one symbol at a time, and the bottleneck of CAE is its sequential processing nature. The proposed multi-symbol CAE design is based on the

Table 4.6: CAE Modes and Associated VOP Types

Mode	Intra / Inter MC	Scanning	Supporting VOP
1	Intra	Horizontal	I-VOPs, and P-VOPs
2	Intra	Vertical	I-VOPs, and P-VOPs
3	Inter MC	Horizontal	P-VOPs
4	Inter MC	Vertical	P-VOPs

inherent characteristics of binary alpha blocks as well as the numerical properties of the probabilities indexed by the contexts, and it is capable of encoding either a singe symbol or multiple symbols within each coding.

In the multi-symbol CAE algorithm, only the symbols with a particular set of contexts are chosen to be multi-symbol encoded. We denote S_{T0} as a symbol whose context has either all-zero or all-one bits, $S_0(n)$ as the occurrence of n successive S_{T0} with the same context, and $S_C(n)$ as the total symbol count of $S_0(n)$, i.e., $S_C(n) = S_0(n) \times n$. Note that $S_0(1)$ and $S_C(1)$ also record the occurrence and the number of symbols that do not belong to S_{T0}. Figure 4.6 illustrates examples for counting $S_0(n)$ in INTRA mode. There are nine successive symbols whose contexts have all-zero bits, thus $S_0(9)$ is increased by one. In the right-hand side of Fig. 4.6, there are three successive symbols whose context has all-one bits and therefore $S_0(3)$ is increased by one. Because each video object is usually a connected body, the neighboring pixels around the coded symbol tend to be all zero or all one, i.e., S_{T0} should tend to appear in clusters, and this situation tend to happen to successively coded symbols.

Figure 4.7 shows the distributions of $S_0(n)$ and $S_C(n)$ for some test sequences. Although $S_0(1)$ dominates the occurrence count, $S_C(1)$ has a proportion of 27% only . On the other hand, the total symbol count of the symbols which can be multi-symbol coded has a proportion of 73%; this implies that S_{T0} tends to appear in clusters.

We can design our multi-symbol CAE algorithm based on the analysis above. The flowchart is shown in Fig. 4.8, where the meanings of RU and RN are as follows:

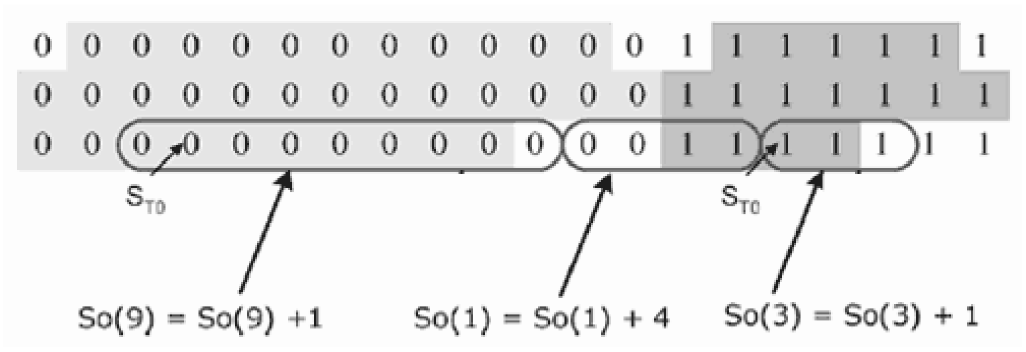


Figure 4.6: Examples of counting $S_0(n)$ in INTRA mode (from [10]).

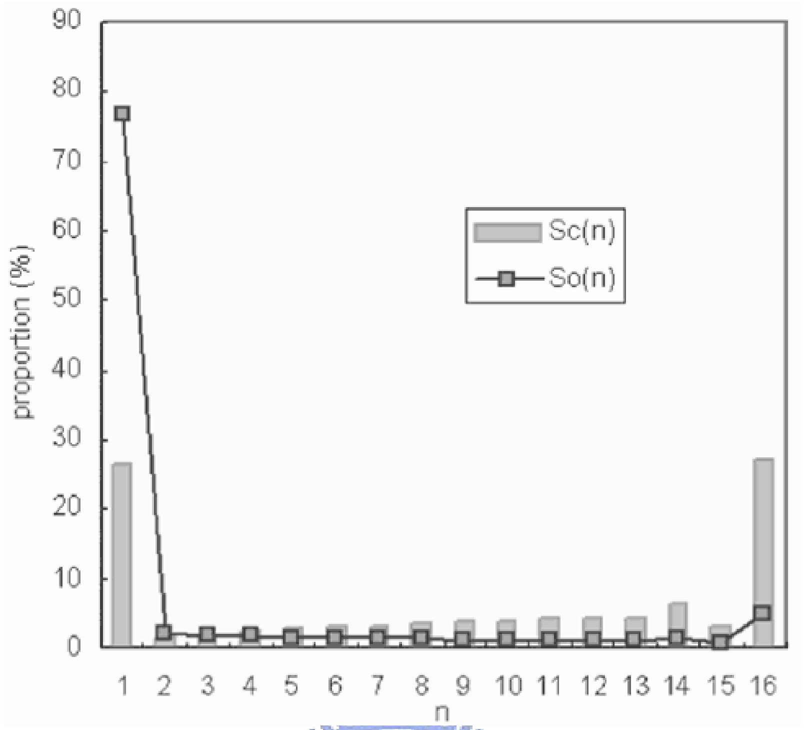


Figure 4.7: Distribution of $S_C(n)$ and $S_0(n)$ (from [10]).

- Range Update (RU): The RU stage updates the range (R) and lower bound (L) of the interval.
- Renormalization (RN): The RN stage renormalizes R when R is smaller than QUARTER. This stage could also output the encoded bits of the arithmetic coder.

After context generation and probability look-up of the binary pixel to be coded, we see if the pixel belongs to S_{T0}. If so, then use multi-symbol coder to encode the successive pixels with the same context, otherwise, use the single-symbol coder to encode it. In the multi-symbol coder, the first step is to count the number of successive pixels with the same context, and denote the counts as n. Then, we do a multi-symbol RN test by looking up a table to check if the range (R) is larger than QUARTER after encoding n symbols. If the test is passed, it can perform the multi-symbol RU stage. On the other hand, failing of the multi-symbol RN test means that the range (R) should be renormalized during encoding n successive symbols. Therefore, the single-symbol coder is used instead of the multi-symbol coder. From some statistics [10], about 6.18% of coded symbols need to perform

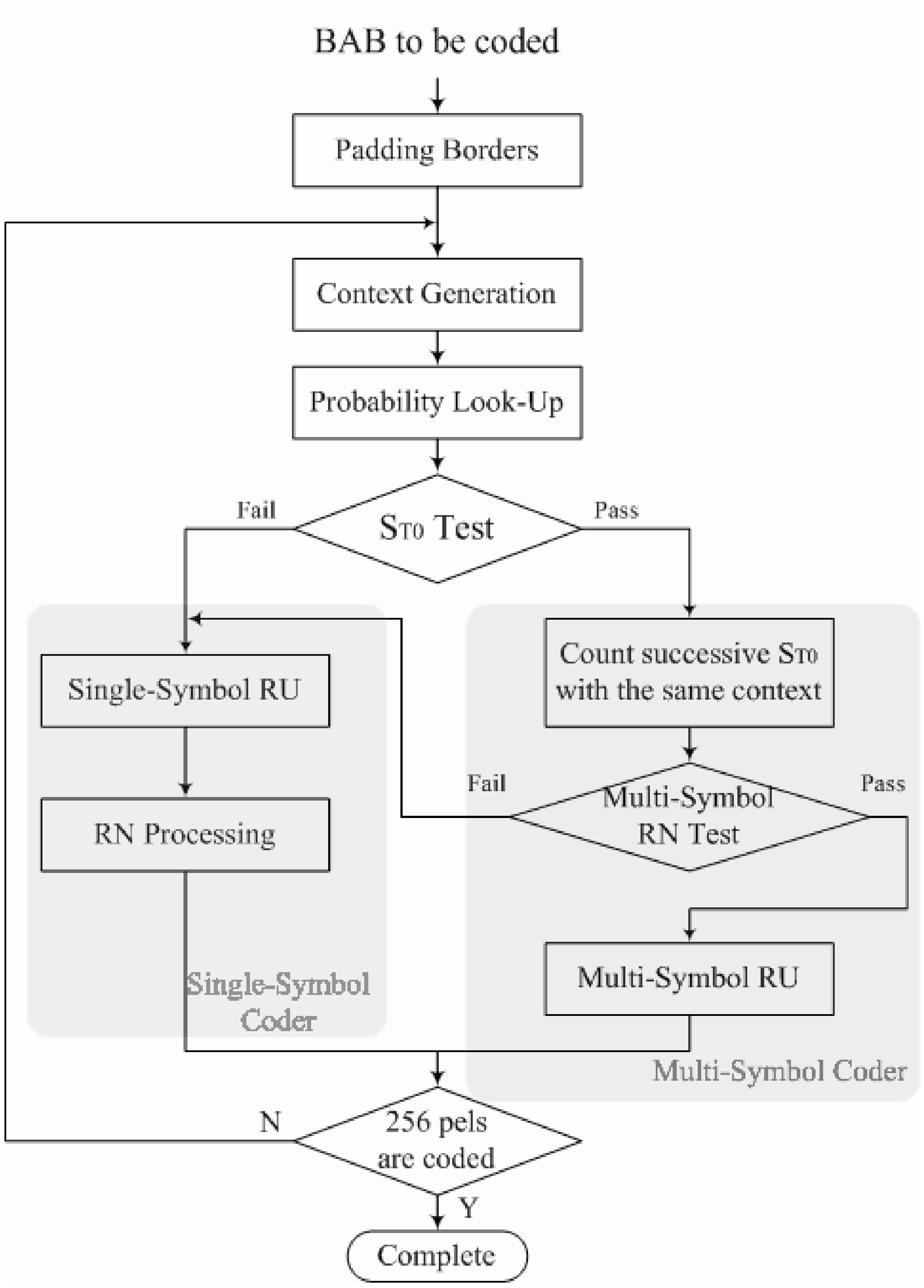


Figure 4.8: Flowchart of multi-symbol CAE.

renormalizations. When the coded pixel is one of n successive S_{T0}, only 0.76% of the coded symbols need to perform renormalizations.

In order to see the performance improvement by multi-symbol CAE, we show the implementation results of single-symbol CAE and multi-symbol CAE on the PACDSP in Fig. 4.9. The results are obtained by encoding one I-VOP of foreman_qcif. Twenty-six BABs are within the VOP, and each is encoded in both horizontal scanning and vertical scanning. Compared to single-symbol CAE, about 40% of execution time is reduced by using multi-symbol CAE.

4.3.2 Modification of Mode Selection Method

All four CAE modes are available for ShapeCoding in P-VOP coding. The one offering highest compression is usually chosen to be the coding mode. Our analysis shows that it is possible to skip hte intra mode.

"ShapeInterMB" is an important function for ShapeCoding in P-VOP coding. It is similar to motion estimation, but perform search in binary alpha plane. A predicted motion vector, MVPs, is taken as the search center, and then a full search is performed over the search window $[-8, 8)$. The MVPs is determined by analyzing certain candidate motion vectors of shape (MVs) and motion vectors of selected texture blocks (MV) around the MB corresponding to the current BAB. They are located and denoted as shown in Fig. 4.10 where MV1, MV2 and MV3 are rounded up to integer values. By traversing MVs1, MVs2, MVs3, MV1, MV2 and MV3 in this order, MVPs is determined by taking the first encountered MV that is defined. If no candidate motion vectors are defined, MVPs = (0,0).

After the search, the motion compensated BAB having the least difference with current BAB is obtained. Then IntraCAE and InterCAE are done separately, and the mode selection criterion is as follows:

$$ShapeBits_{INTRA} \gtrless ShapeBits_{INTER} + offset$$

where $offset$ consists of coded bits for the shape mode and that for MVDs. However, we find that under the two conditions below the odds are in favor of choosing the inter mode:

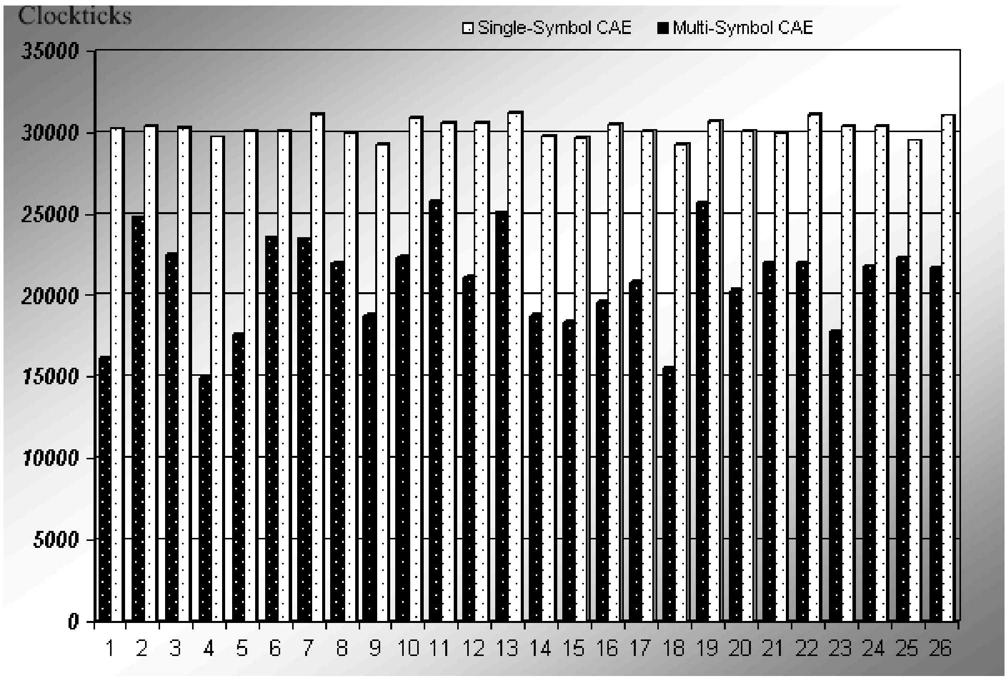


Figure 4.9: Performance improvement by multi-symbol CAE.

1. Number of different pels between motion compensated BAB and current BAB are small.
2. The offset is small.

Both the different pixels and offset are known before CAE operation. Therefore, two thresholds, “D_th” and “offset_th,” are introduced to test if the different pixels and offset are small enough. If the different pixels and offset are smaller than “D_th” and “offset_th,” respectively, then coding by inter mode should yield a good compression ratio. Then we skip the coding by intra mode and take the coding results of inter mode as the output bitstream. We simulate 100 P-VOPs of stefan_qcif and present the results in Table 4.7, where “SR” is the ratio of skipping intra mode coding and “bpv” is the related shape bits per VOP. We also collect the results of SR with different thresholds in Fig. 4.11. We find that, even in the right-bottom region of Table 4.7, more than 50% intra mode coding are skipped but the amount of increase in shape bits are still less than 1%. It is a good idea to trade bit-rate with computation complexity, and the ShapeCoding is still lossless.

There is a interesting phenomenon in the simulation results. That is, in some cases

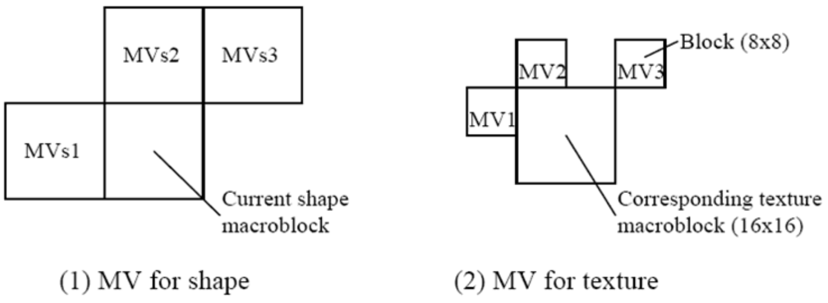


Figure 4.10: Candidates for MVPs.

we skip more coding but a smaller bpv is obtained. The reason is that, more Inter mode coding would increase the candidates for MVPs, and a different MVPs would form a different search region. It triggers a "chain reaction" in the prediction coding flow.

4.3.3 Reducing Candidates for MVs

An interesting property observed in motion estimation is that there is no big difference in SAD values between two blocks located 1 pixel away from each other. This is because the luminance value of a pixel differs only a little from its neighbor's values. However, there are only two different pixel values in binary alpha images: 255 for opaque pixel, and 0 for transparent pixel. Thus we can take the different pixels (DP) as the criterion value for motion estimation in binary alpha plane.

The function "ShapeInterMB" which performs a full search on binary alpha plane aims to find an optimal match over the search range. Based on the characteristics mentioned above, there must be a sub-optimal match at nearby positions of the best match. Therefore, we use a search step equal to 2 in both the horizontal and the vertical directions to reduce the number of candidates for MVs. This results in omitting the comparison of roughly 3/4 of the number of the blocks, thus decreasing the complexity. However, the cost is that the shape bits are increased when a sub-optimal match is taken. We show the experiment results on ADS in Table 4.8 where the execution time (cycles) is obtained by encoding 1 P-VOP on ADS, and the shape bits (bpv) are statistically averages from

Table 4.7: Simulation Results of Skip Ratio (SR) and Shape Bits per VOP (bpv)

D_th		0	1	2	3	4	5	6	7	8
Offset_th	SR (%)	0	0.18	0.37	0.46	1.01	1.10	1.19	1.28	1.28
= 0	bpv	315.55	315.55	315.55	315.55	315.50	315.50	315.50	315.50	315.50
Offset_th	SR (%)	0	1.19	2.38	2.93	4.30	5.22	5.86	6.04	6.41
= 1	bpv	315.55	315.55	315.55	315.55	315.50	315.50	315.50	315.50	315.50
Offset_th	SR (%)	0	1.19	2.38	2.93	4.30	5.22	5.86	6.04	6.41
= 2	bpv	315.55	315.55	315.55	315.55	315.50	315.50	315.50	315.50	315.50
Offset_th	SR (%)	0	1.65	3.11	3.75	5.31	6.50	7.51	7.88	8.33
= 3	bpv	315.55	315.55	315.55	315.55	315.50	315.50	315.50	315.50	315.50
Offset_th	SR (%)	0	2.47	4.58	5.95	8.42	11.26	13.37	14.47	15.11
= 4	bpv	315.55	315.57	315.57	315.57	315.51	315.50	315.44	315.44	315.44
Offset_th	SR (%)	0	3.85	6.87	9.16	12.82	16.48	19.05	20.42	21.25
= 5	bpv	315.55	315.57	315.55	315.55	315.51	315.51	315.45	315.45	315.50
Offset_th	SR (%)	0	4.85	8.88	12.09	17.12	21.79	25.18	28.39	30.49
= 6	bpv	315.55	315.71	315.77	315.77	315.83	315.97	315.87	315.77	315.88
Offset_th	SR (%)	0	5.86	11.26	15.93	22.71	28.02	32.60	36.54	39.47
= 7	bpv	315.55	315.71	315.81	315.87	315.87	316.03	316.00	315.92	316.07
Offset_th	SR (%)	0	6.32	12.45	18.32	26.28	32.23	38.37	42.86	46.25
= 8	bpv	315.55	315.75	315.81	315.89	315.88	316.07	316.05	316.08	316.29
Offset_th	SR (%)	0	6.96	13.55	20.51	29.58	36.81	43.41	48.17	51.56
= 9	bpv	315.55	315.69	315.85	315.95	315.98	316.24	316.17	316.21	316.45
Offset_th	SR (%)	0	6.96	14.29	21.89	31.68	39.38	46.15	52.11	56.32
= 10	bpv	315.55	315.69	315.92	316.04	315.94	316.38	316.28	316.34	316.69

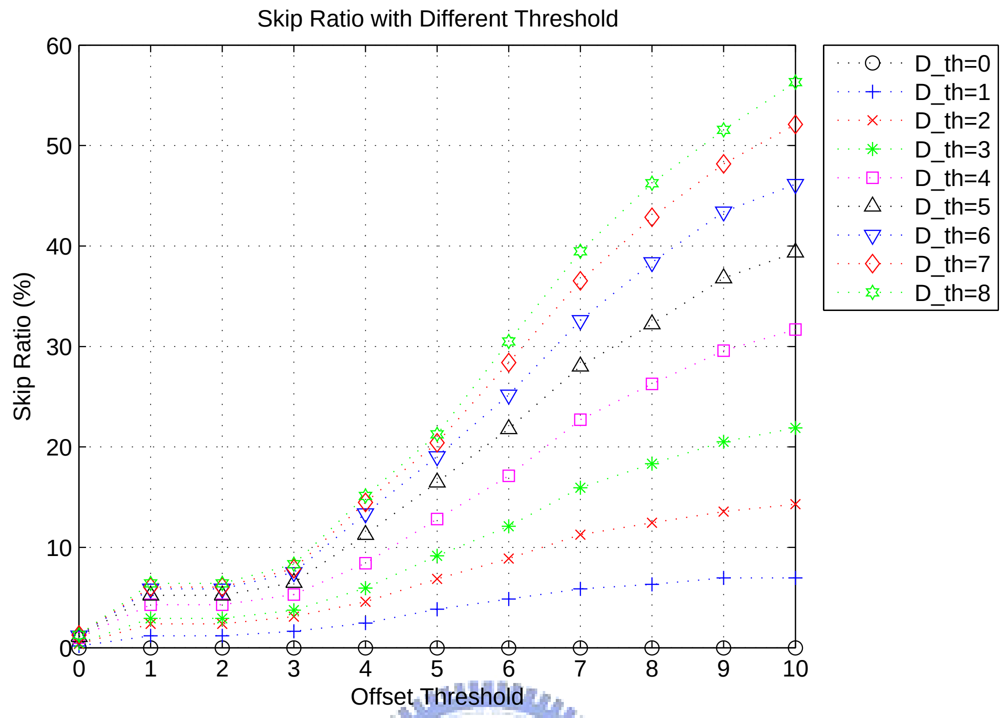


Figure 4.11: Skip ratio (SR) with different thresholds.

encoding 100 P-VOPs.

4.4 Texture Coder Analysis and Design

The floating-point DCT and IDCT of the texture coder are time-consuming functions. Implementing the transforms in fixed-point is essential for PACDSP. We will discuss this

Table 4.8: Execution Results on ADS of Reduced-Complexity ShapeInterMB Function

Test Seq.	Execution Time (cycles)			Shape Bits (bpv)		
(QCIF)	Original	Modified	%	Original	Modified	%
foreman	30,833,225	9,251,863	69.99	555.85	610.14	9.77
akiyo	10,436,508	3,319,540	68.19	230.71	225.31	-2.34
stefan	15,631,836	4,502,918	71.19	315.55	338.25	7.19

subject in the next chapter. In this section, we do some analysis to eliminate dequantization and inverse transform in some situation.

An important property of DCT is that it concentrates signal energy in lower frequency coefficients [11]. For example, if a block is filled with constant coefficients, there will be only one coefficient at the DC after the transform. In other words, if we can make sure that there is only a DC component in the quantized block, the corresponding output block data can be obtained with copying the DC component to the entire block. This is illustrated in Fig. 4.12.

Coded block pattern (CBP) is a parameter coded in macroblock header to tell the decoder which blocks in a MB are variable length coded. “FindCBP” is the function to set the coded block pattern by scanning the quantized block. The procedure of checking skipped blocks is similar to the function “FindCBP”. We combine the checking procedure with the function “FindCBP”. This technique can be applied in both intra-mode coding and inter-mode coding. The simulation results on PC are listed in Table 4.9. We see that the skipped rate is highly related to the motion characteristics of the test sequence and the quantization step size (QP). Thus we can reduce the computation complexity of reconstructed loop of video encoder in our implementation.

4.5 Implementation Strategy on Dual-Core Platform

Since the data memory and instruction cache on PACDSP is limited to 64kB and 32kB, respectively, it is hard to implement all the encoder functions on chip. We implement the encoder on the dual-core system illustrated in Fig 4.13. We now focus on the dual-core mechanism of MPEG-4 object-based video encoder.

Figure 4.14 shows the basic design of the software implementation. We encode the shape information on DSP and do the texture padding on the ARM at the same time. Furthermore, the texture coder is split into two parts: the transformer and the bitstream coder. After forward transform, we transmit the quantized coefficients to the bitstream coder on ARM. Then, variable length coding and the inverse transform are performed at the same time. At last, we pad the reconstructed VOP to be the reference VOP for coding

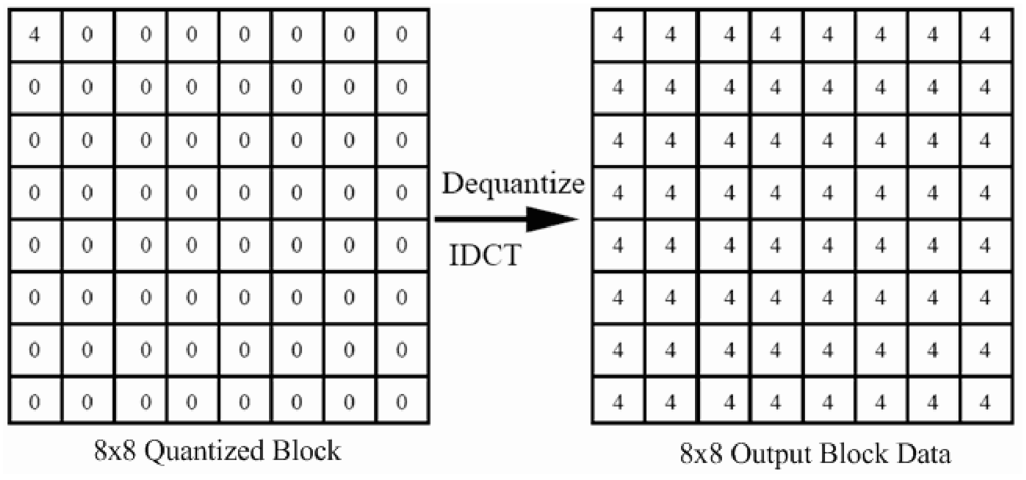


Figure 4.12: DC spreading from quantized coefficient to output block.

Table 4.9: Number of Skipped Blocks in 101 Frames (1 I, 100 P)

Test Seqs. (QCIF)	Transformed Block No.	Skipped Block No.		%
foreman	27,401	QP=2	8,133	29.68
		QP=4	13,682	49.93
		QP=7	17,865	65.20
akiyo	26,732	QP=2	14,389	53.83
		QP=4	19,088	71.41
		QP=7	21,657	81.02
stefan	5,874	QP=2	630	10.73
		QP=4	1,421	24.19
		QP=7	2,321	39.51

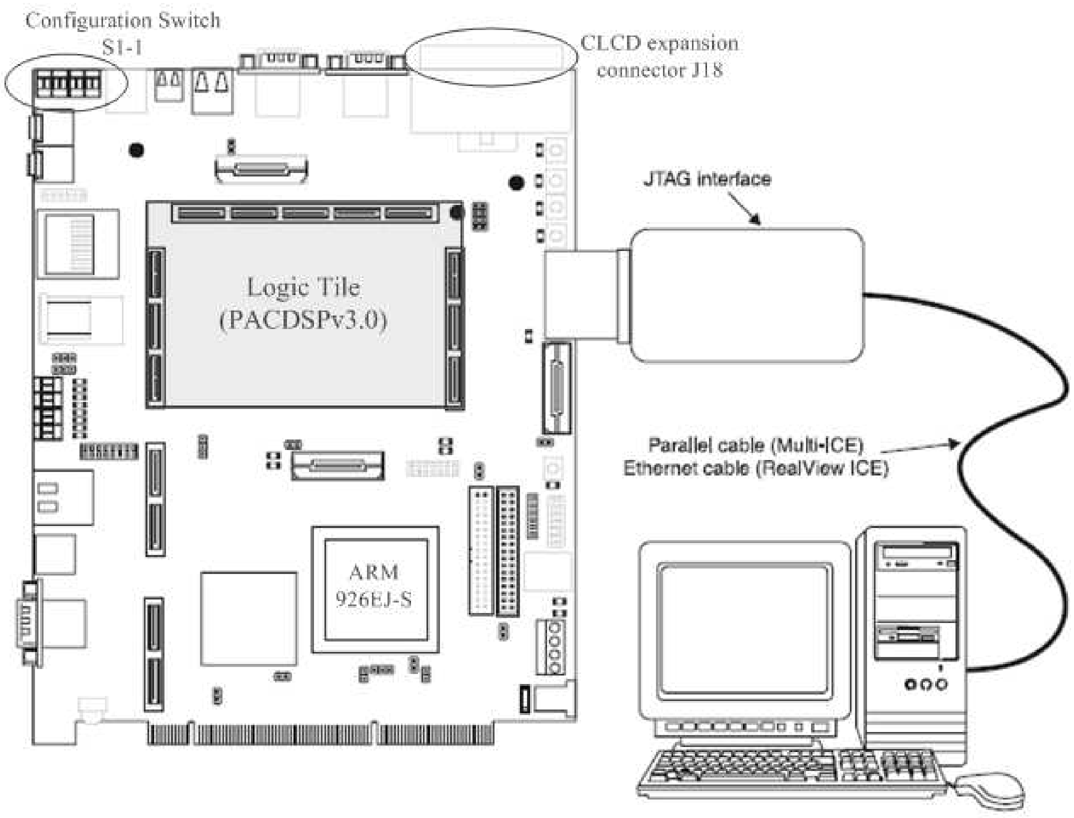

Configuration Switch S1-1
CLCD expansion connector J18
JTAG interface
Logic Tile (PACDSPv3.0)
Parallel cable (Multi-ICE)
Ethernet cable (RealView ICE)
ARM 926EJ-S

Figure 4.13: PACDSP v3.0 system.

of the next frame.

Implementation Approach

Since the goal of our implementation is to achieve a real-time MPEG-4 video encoder on PACDSP v3.0, the execution time and the code size are the most important issues. Although we can program in a high-level language, experience shows that the current compiler cannot address these issues to the desired level. Moreover, the development of compiler was not complete when we began our implementation. Thus our implementation uses assembly programming.

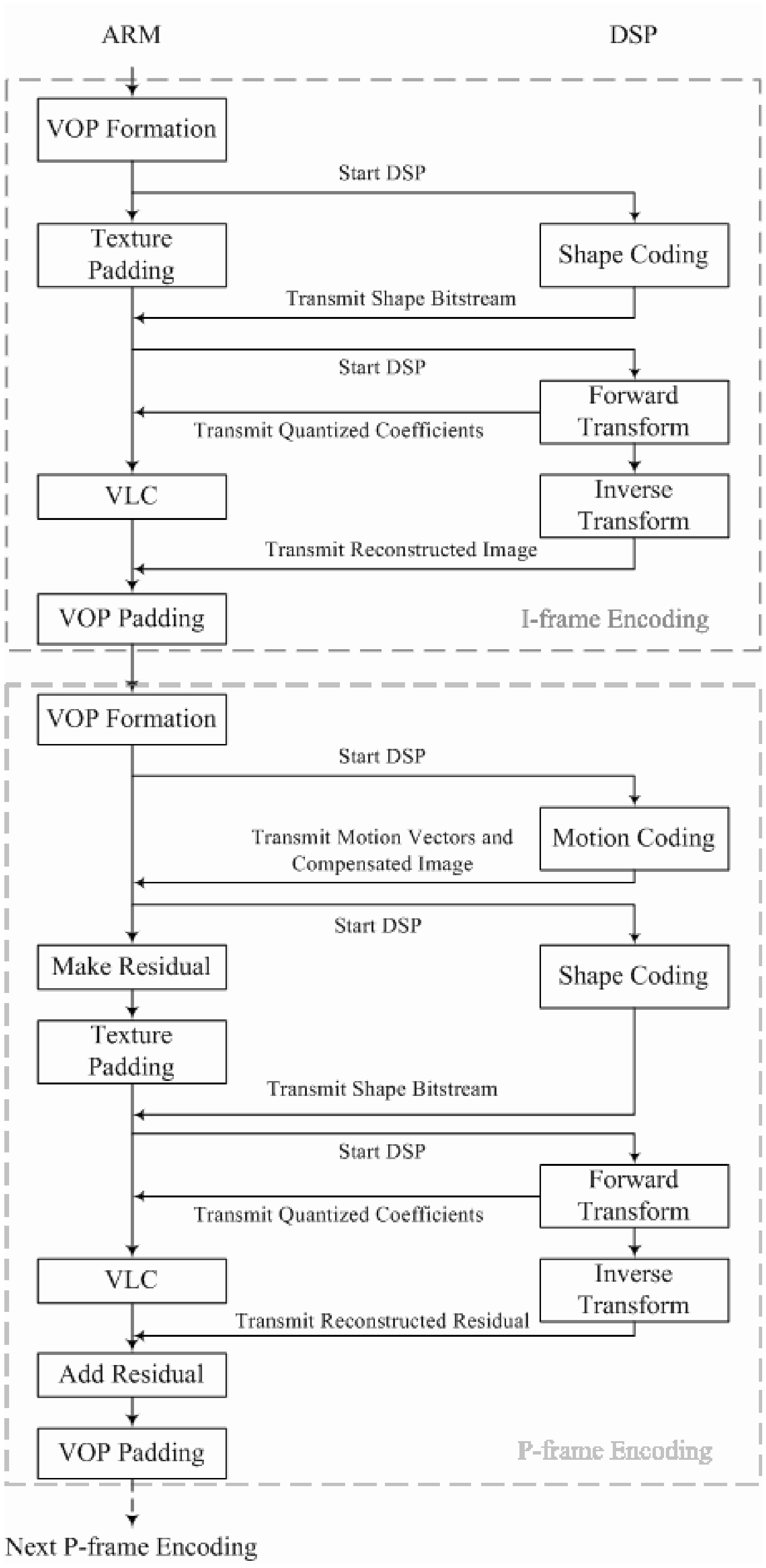


Figure 4.14: Our basic dual-core software encoder design.

Chapter 5

Optimization of Implementation on PACDSP

In this chapter, we discuss the optimization of our implementation of the MPEG-4 object-based video encoder on PACDSP. First, some general techniques of code optimization are introduced. Then, we present the fixed-point design of DCT, IDCT, and quantization. We also discuss the performance of the optimization. In addition, we compare the performance with some other reported implementations on other hardware platforms.

5.1 General Techniques of Code Optimization

The utilization of architectural advantages is important in DSP implementation of complicated algorithms such as video encoder. 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 the VLIW processor. Some special SIMD instructions of PACDSP are used to compute or load/store multiple data at the same time. The advantage of SIMD instructions is to increase the throughput of computations.

5.1.1 General Optimization Techniques

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

In the following discussion, we use an example of summing the coefficients in a 1-D array, which contains eight 8-bit data. The corresponding C program is shown in Fig. 5.1. In order to simplify the utilization of different techniques, we use only one instruction slot in the instruction packet.

Static Rescheduling

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

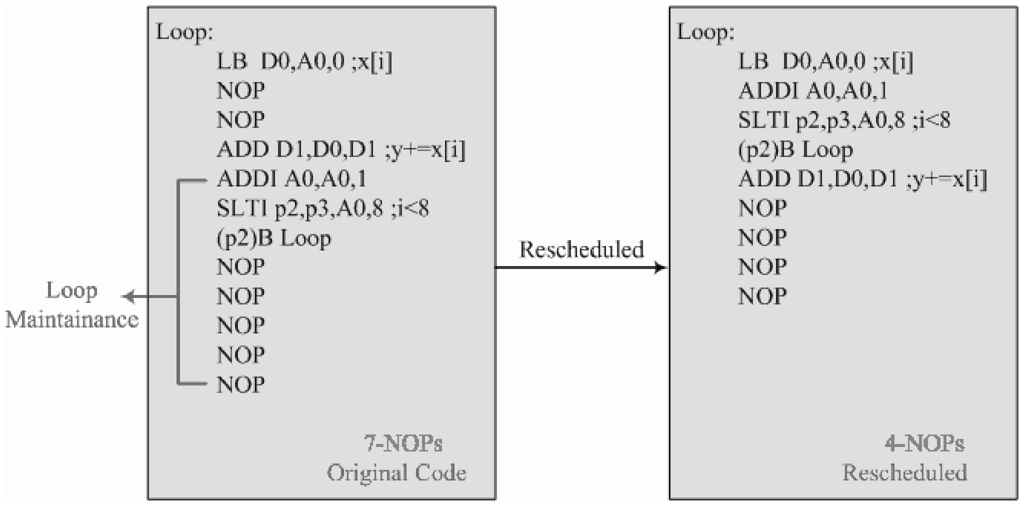
In the left half of Fig. 5.2, we simply translate the C program in Fig. 5.1 to the PACDSP assembly code. We can see that two stalls after the “LB” instruction are resulted from the dependency of the register D0, because data loading from memory requires two cycles to be valid in PACDSP.

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

We can utilize the independence of instructions to eliminate the stalls as much as

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

Figure 5.1: Example of vector addition.



```
Loop:
      LB  D0,A0,0 ;x[i]
      NOP
      NOP
      ADD D1,D0,D1 ;y+=x[i]
      ADDI A0,A0,1
      SLTI p2,p3,A0,8 ;i<8
      (p2)B Loop
      NOP
      NOP
      NOP
      NOP
      NOP
```

```
Loop:
      LB  D0,A0,0 ;x[i]
      ADDI A0,A0,1
      SLTI p2,p3,A0,8 ;i<8
      (p2)B Loop
      ADD D1,D0,D1 ;y+=x[i]
      NOP
      NOP
      NOP
      NOP
```

Figure 5.2: Example of static rescheduling technique.

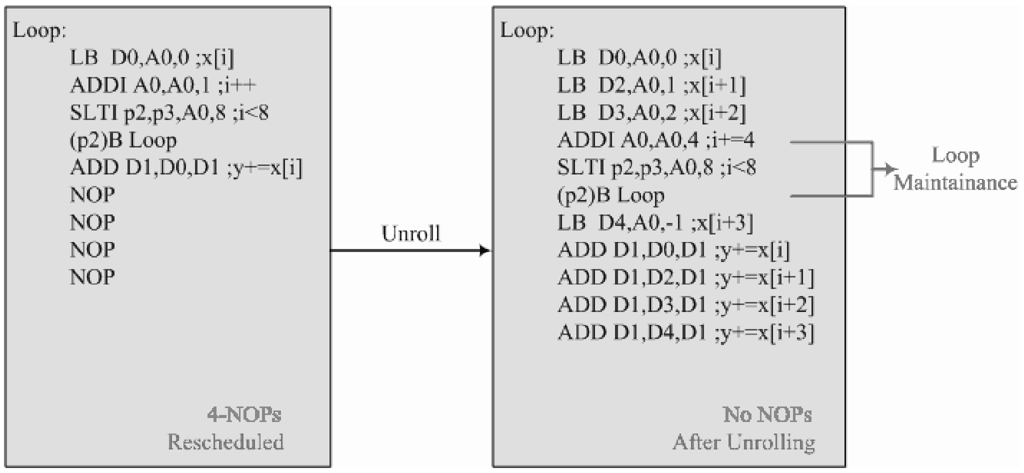
possible. In the right half of Fig. 5.2, we change the order of the assembly code, which reduce the stalls from seven to four. However, since the computation is not very complex, we cannot further reduce the number of stalls simply through rescheduling.

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 5.3 shows the assembly code before and after loop unrolling.

we see that in Fig. 5.3, 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, 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.



```
Loop:
        LB  D0,A0,0 ;x[i]
        ADDI A0,A0,1 ;i++
        SLTI p2,p3,A0,8 ;i<8
        (p2)B Loop
        ADD D1,D0,D1 ;y+=x[i]
        NOP
        NOP
        NOP
        NOP
```

```
Loop:
        LB  D0,A0,0 ;x[i]
        LB  D2,A0,1 ;x[i+1]
        LB  D3,A0,2 ;x[i+2]
        ADDI A0,A0,4 ;i+=4
        SLTI p2,p3,A0,8 ;i<8
        (p2)B Loop
        LB  D4,A0,-1 ;x[i+3]
        ADD D1,D0,D1 ;y+=x[i]
        ADD D1,D2,D1 ;y+=x[i+1]
        ADD D1,D3,D1 ;y+=x[i+2]
        ADD D1,D4,D1 ;y+=x[i+3]
```

Figure 5.3: Example of loop unrolling technique.

Software Pipelining

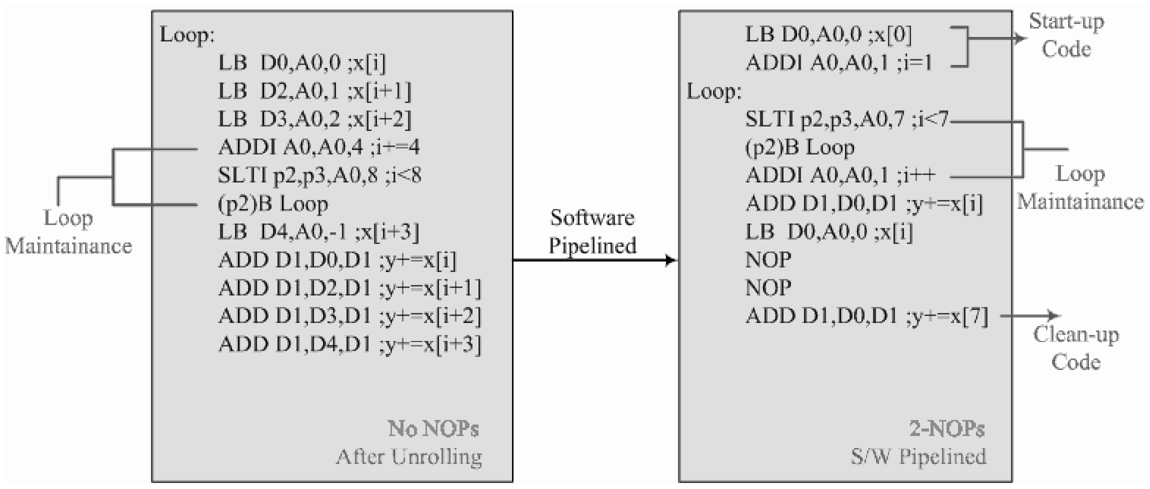
The concept of software pipelining is to reorganize the loop and to interleave dependent instructions from different loop iterations to separate dependent instructions within the original loop. Different from loop unrolling, we just reschedule the loop, so the stalls may not be entirely eliminated. An example of software pipelining is illustrated in Fig. 5.4.

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

5.1.2 Features of PACDSP

To effect an efficient implementation on PACDSP, we should utilize the parallelism in the VLIW architecture and SIMD instructions. However, not all the computations can be distributed to both clusters, so we have to check if the features of the implemented algorithm can make use of the parallelism in PACDSP.

For example, since the branch instructions affect the program execution sequence of both clusters, it is better to put two regular and independent parts of the overall algorithm in different clusters. For this, an iterative computation can be separated into two parts if



```
Loop:
      LB  D0,A0,0 ;x[i]
      LB  D2,A0,1 ;x[i+1]
      LB  D3,A0,2 ;x[i+2]
      ADDI A0,A0,4 ;i+=4
      SLTI p2,p3,A0,8 ;i<8
      (p2)B Loop
      LB  D4,A0,-1 ;x[i+3]
      ADD D1,D0,D1 ;y+=x[i]
      ADD D1,D2,D1 ;y+=x[i+1]
      ADD D1,D3,D1 ;y+=x[i+2]
      ADD D1,D4,D1 ;y+=x[i+3]
```

```
      LB D0,A0,0 ;x[0]
      ADDI A0,A0,1 ;i=1
Loop:
      SLTI p2,p3,A0,7 ;i<7
      (p2)B Loop
      ADDI A0,A0,1 ;i++
      ADD D1,D0,D1 ;y+=x[i]
      LB  D0,A0,0 ;x[i]
      NOP
      NOP
      ADD D1,D0,D1 ;y+=x[7]
```

Figure 5.4: Example of software pipelining technique.

the computations are independent in different iterations. In MPEG-4 object-based video encoder, the functions for motion estimation, DCT, IDCT, and quantization, and inverse quantization are very regular computations. We will discuss these functions in the following sections. However, we usually use only one cluster to implement sequential code sequence. In some complicated functions, such as CAE, we can put some independent parts of the computation in another cluster and use the broadcast instruction to fetch the desired results back. That is another way to improve the performance, but we should pay attention to the stalls caused by data communication so that they would not outweigh the gain from parallel computation.

SIMD instructions are also very helpful for optimization. The data length in motion estimation is equal to a byte per pixel. Thus it is useful to use the special SIMD instructions available on PACDSP to calculate SAD. We will show the SIMD example below.

5.2 Fixed-Point DCT and IDCT

We have seen previously that efficient and accurate fixed-point DCT and IDCT are essential in our implementation on PACDSP. In this section, we discuss the fixed-point design which takes into account the PACDSP architecture. Since the optimized techniques are similar for DCT and IDCT, our discussion only focuses on IDCT only.

DCT and IDCT Algorithm

The DCT and IDCT in MPEG-4 are defined as

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

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

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

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

To implement DCT and IDCT on PACDSP, there are two critical issues, namely, efficiency and accuracy, which are discussed below.

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 [16]. One fast method reduces the required 1-D IDCTs from 16 to 8 [16]. 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 [17]. A comparison of computational complexity of different algorithm is listed in Table 5.1.

Note that the computational complexity is estimated for floating-point computation. Since the transform coefficients used in [17] are reciprocals of cosine values, the error increases because of limited accuracy in the fixed-point approximation on PACDSP. In

Table 5.1: Comparison of Computational Complexity for 8-point IDCT

	Direct Form	FCT [17]	MoMuSys	Even_Odd FCT [18]
Multiplications	64	12	16	20
Additions	56	29	26	28

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.

Accuracy of IDCT

Since the PACDSP is not capable of floating-point computations, we have to convert the IDCT algorithm to fixed-point computation. There are also many approximation algorithms to floating-point IDCT. There are integer reversible algorithms for DCT/IDCT [19],[20], but they consist of several matrix computations, and the computational complexity should be much higher. Therefore, we do not implement a reversible transform.

Since the native wordlength 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 products to the nearest integers.

The 1-D IDCT algorithm used in MoMuSys has the signal flow shown in Fig. 5.5. We need to check if the implementation is accurate enough. The modefied IEEE Std. 1180-1190, which is currently withdrawn, is usually used to test for the compliance of the implementation of IDCT algorithms. The compliance test requires five statistical measurements, which are as follows [5]:

- 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 results of MoMuSys algorithm is shown in Table 5.2. We see that the simple rounding method introduces significant errors, so this algorithm does not comply with the IEEE 1180-1190 standard after converting to fixed-point computation. However, we see that the odd-indexed coefficients are rounded twice in this algorithm, yielding

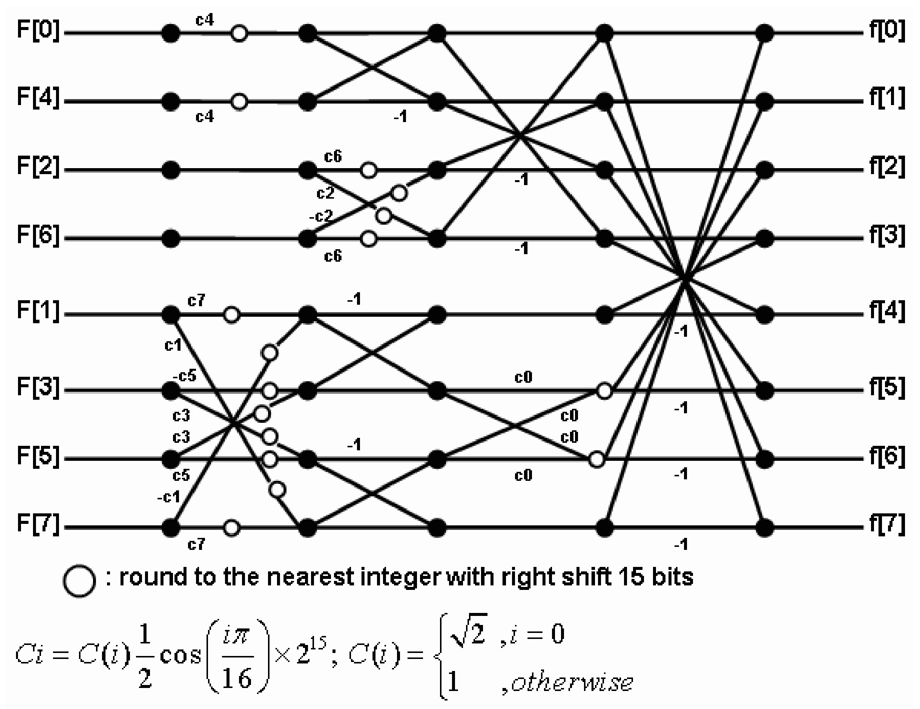

F[0]
F[4]
F[2]
F[6]
F[1]
F[3]
F[5]
F[7]
f[0]
f[1]
f[2]
f[3]
f[4]
f[5]
f[6]
f[7]
c4
c4
c6
c2
-c2
c6
c7
c1
-c5
c3
c3
c5
-c1
c7
c0
c0
c0
c0
-1
: round to the nearest integer with right shift 15 bits
$Ci = C(i)\frac{1}{2}\cos\left(\frac{i\pi}{16}\right)\times 2^{15};\ C(i) = \begin{cases}\sqrt{2} & ,i = 0\\ 1 & ,otherwise\end{cases}$

Figure 5.5: The IDCT algorithm used in MoMuSys [9].

serious rounding errors. Therefore, we try to use the even-odd decomposition algorithm [18] whose signal flow is shown in Fig. 5.6. In this algorithm, each coefficient is rounded once, which can reduce the rounding error. Moreover, we use the following rounding rules.

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

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 5.2. We can see that our fixed-point IDCT has enough accuracy to pass the test.

Table 5.2: Test of Compliance for Modified IEEE Std. 1180-1190 in MPEG-4

Item	Modified IEEE 1180–1190	MoMuSys	Our Algorithm
ppe	≤ 2	>2 (X)	≤ 2 (○)
$pmse$	≤ 0.06	137.8279 (X)	0.0081 (○)
$omse$	≤ 0.02	5.2222 (X)	0.0056 (○)
pme	≤ 0.015	10.8429 (X)	0.0019 (○)
ome	≤ 0.0015	0.5742 (X)	0.0001 (○)
all zero input	all zero output	○	○

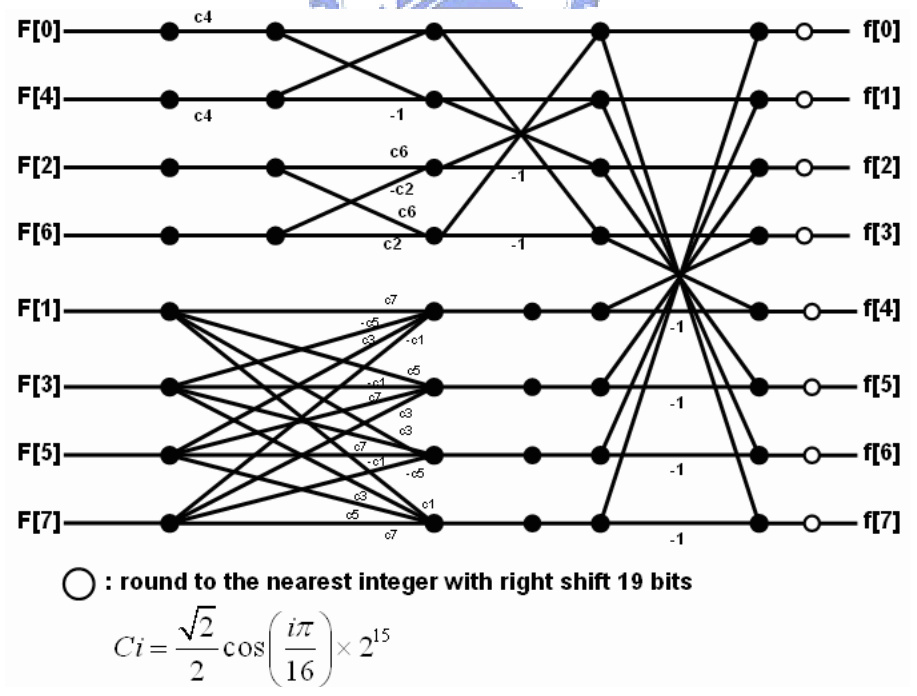


Figure 5.6: The even-odd decomposition IDCT algorithm [13].

Table 5.3: Comparison of IDCT on Different Platforms

Designs	Processing Units	Clock (MHz)	2-D Fast Algo.	Cycles	Equivalent Instruction Counts
TI C62x [21]	2 MUL, 6 ALU	200	row-column	230	1840
TI C64x [22]	2 MUL, 6 ALU	600	row-column	154	1232
IDCT Core [21]	1 ALU	33	direct 2-D	1208	1208
PACDSP v3.0 (ours)*	2 AU, 2 L/S	200	even-odd	293	1172

*Note: If considered having 5 processing units, then equivalent instruction counts = 1465.

Optimization of IDCT on PACDSP

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

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

The performance of various IDCT implementation are listed in Table 5.3. We see that the implementation on PACDSP is competitive, because of less arithmetic units required.

Implementation of DCT on PACDSP

Similar to the optimization of IDCT, we can utilize the same way to implement DCT on PACDSP. Figure 5.7 shows the signal flow of even-odd decomposition DCT algorithm. However, the performance of various DCT implementation are listed in Table 5.4.

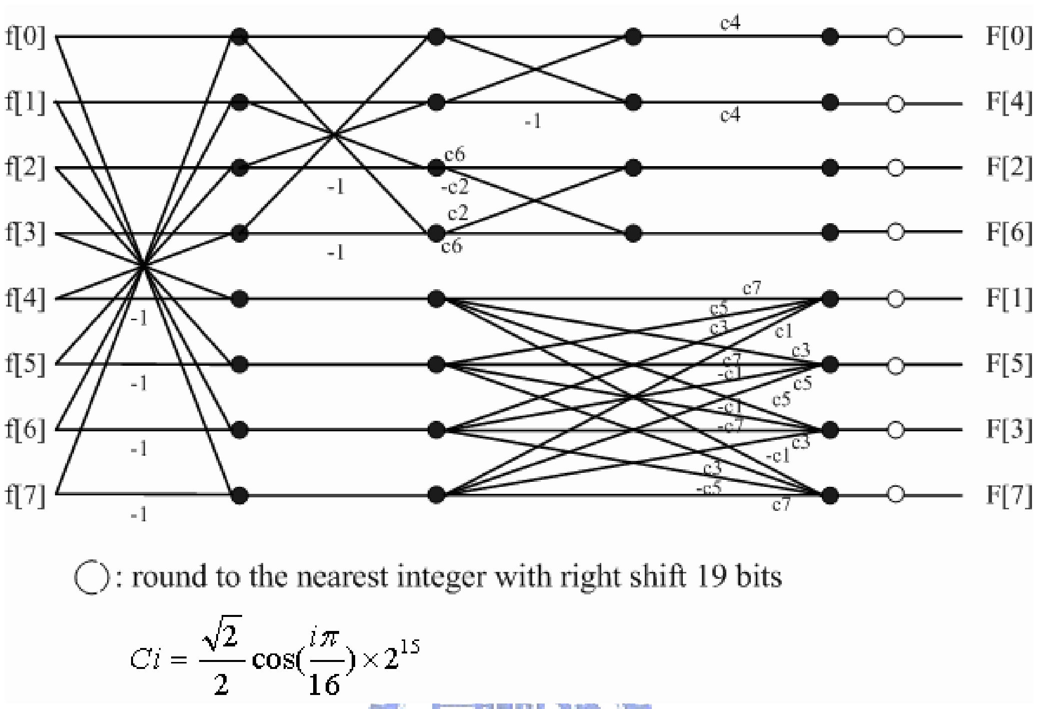

f[0]
f[1]
f[2]
f[3]
f[4]
f[5]
f[6]
f[7]
F[0]
F[4]
F[2]
F[6]
F[1]
F[5]
F[3]
F[7]
c4
-1
c6
-c2
c2
c6
c7
c5
c3
c1
-c7
-c1
-c5
c3
-c1
c7
○: round to the nearest integer with right shift 19 bits
$Ci = \frac{\sqrt{2}}{2}\cos(\frac{i\pi}{16}) \times 2^{15}$

Figure 5.7: The even-odd decomposition DCT algorithm [13].

Table 5.4: Comparison of DCT on Different Platforms

Designs	Processing Units	Clock (MHz)	2-D Fast Algo.	Cycles	Equivalent Instruction Counts
TI C62x [21]	2 MUL, 6 ALU	200	row-column	208	1664
TI C64x [22]	2 MUL, 6 ALU	600	row-column	116	928
PACDSP v3.0 (ours)*	2 AU, 2 L/S	200	even-odd	321	1284

*Note: If considered having 5 processing units, then equivalent instruction counts = 1605.

5.3 Fixed-Point Quantization

5.3.1 H.263 Quantization Method

We only consider the H.263 quantization method in our implementation. The quantization method is defined as follows:

- Intra coded block

$$QF[v][u] = \begin{cases} \dfrac{F[0][0] + \frac{dc\_scaler}{2}}{dc\_scaler}, \text{ if } v, u\text{=0 (DC component)}, \\ \\ \dfrac{|F[v][u]|}{2 \times QP} \times \text{SIGN(F[v][u]), otherwise (AC component).} \end{cases} \tag{5.3}$$

- Inter coded block

$$QF[v][u] = \frac{|F[v][u]| - \frac{QP}{2}}{2 \times QP} \times SIGN(F[v][u]) \tag{5.4}$$

where dc_scaler is a nonlinear scaling factor introduced in chapter 2. The division operation is unavailable in PACDSP. We should find a fixed-point method to replace the division operation, and the accuracy is an important issue.

5.3.2 Lossless Fixed-Point Quantization Method

If floating-point division were available, the quantizations defined above could be achieved by floating-point division and rounding or truncation. However, in our case, a more efficient way is to replace it by fixed-point multiplication. That is, an approximate inverse of the divisor is multiplied to the dividend following by a right shift of the result. A key issue is how many bits should be used to represent the divisor's fixed-point approximate inverse to achieve a lossless substitution.

Since the quantizer parameter (QP) is in the range from 1 to 31, the divisor, dc_scaler is from 8 to 46 for luminance blocks, and from 8 to 25 for chrominance blocks. That means, among all the possible divisor value, i.e.,$\{2\times QP, dc\_scaler\}$, the maximum value is 62. If the precision of the fixed-point approximation can distinguish the minimum difference between the non-linear scaling factors $\frac{1}{61} - \frac{1}{62} = \frac{1}{3782}$, then we can achieve the lossless substitution is possibly achieved. Therefore, it needs at least 13 bits (Q1.12 representation) to represent the divisor's inverse in fixed-point approximation. Table 5.5 list all the possible values of the inverse of divisor in Q1.15 format.

The memory space required for the table is 248 bytes. In our implementation of quantization, we can get the dc_scaler and the associated inverse of divisor by looking up the table. Then the division operation can be achieved by multiplication and right shift without any precision loss.

5.4 Implementation of SAD Calculation Using SIMD

Calculating the sum of absolute difference (SAD) is the most critical function in motion estimation. In this section, we optimize the SAD calculation by using SIMD instructions on PACDSP.

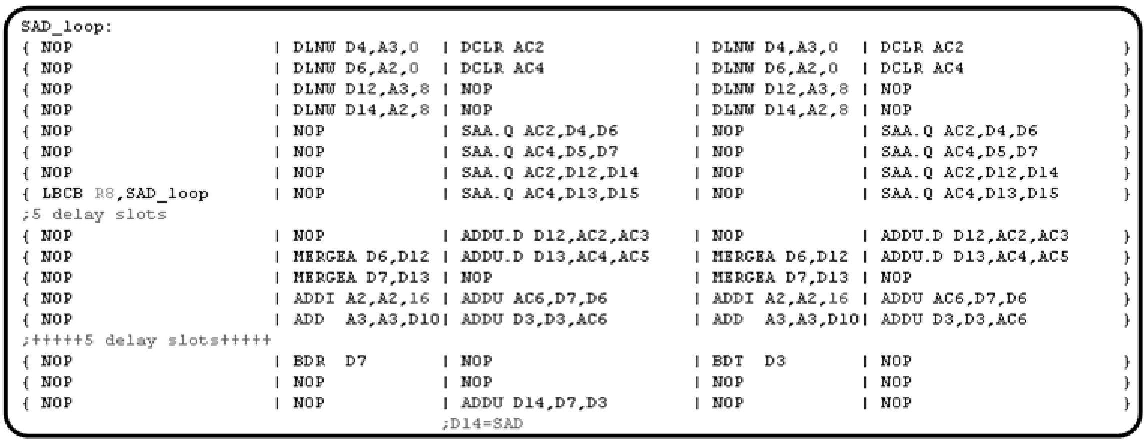
Since the luminance data only contain 8 bits per pixel, we can use 32-bit SIMD instructions to handle 4 pixels in a single instruction. In addition, we can carry out the 16×16 or 8×8 SAD calculation into two clusters. The optimization techniques described in previous sections can be used. Figure 5.8 shows an example code for 16×16 SAD calculation in PACDSP.

In the example code, we use double-loads to load 8 pixels in one instruction. Then, a special SIMD instruction, namely "SAA.Q," is used. Fig. 5.9 shows the syntax and operation of "SAA.Q." It subtracts four pairs of 8-bit values, takes the absolute values and accumulates them individually. Finally, we use the instructions "ADDU.D" and "MERGEA" to sum up the results. It takes 114 cycles to implement a 16×16 SAD calculation and 32 cycles to implement an 8×8 SAD calculation on PACDSP. Table 5.6 shows the performance of various SAD implementations. We can see from the last column of Table 5.6 that the implementation on PACDSP is competitive.

In object-based video encoder, the SAD calculation is only applied to the pixels be-

Table 5.5: Fixed-Point Quantization Table

QP	DC_Scaler		$\frac{1}{QP}$	$\frac{1}{DC\_Scaler}$		QP	DC_Scaler		$\frac{1}{QP}$	$\frac{1}{DC\_Scaler}$	
	Luma	Chroma		Luma	Chroma		Luma	Chroma		Luma	Chroma
1	8	8	32768	4096	4096	17	25	15	1928	1311	2185
2	8	8	16384	4096	4096	18	26	15	1820	1260	2158
3	8	8	10923	4096	4096	19	27	16	1725	1214	2048
4	8	8	8192	4096	4096	20	28	16	1638	1170	2048
5	10	9	6554	3277	3641	21	29	17	1560	1130	1928
6	12	9	5461	2731	3641	22	30	17	1489	1092	1928
7	14	10	4681	2341	3277	23	31	18	1425	1057	1820
8	16	10	4096	2048	3277	24	32	18	1365	1024	1820
9	17	11	3641	1928	2979	25	34	19	1311	964	1725
10	18	11	3277	1820	2979	26	36	20	1260	910	1638
11	19	12	2979	1725	2731	27	38	21	1214	862	1560
12	20	12	2731	1638	2731	28	40	22	1170	819	1489
13	21	13	2521	1560	2521	29	42	23	1130	780	1425
14	22	13	2341	1489	2521	30	44	24	1092	745	1365
15	23	14	2185	1425	2341	31	46	25	1057	712	1311
16	24	14	2048	1365	2341						



```
SAD_loop:
{ NOP                 | DLNW D4,A3,0  | DCLR AC2             | DLNW D4,A3,0  | DCLR AC2             }
{ NOP                 | DLNW D6,A2,0  | DCLR AC4             | DLNW D6,A2,0  | DCLR AC4             }
{ NOP                 | DLNW D12,A3,8 | NOP                  | DLNW D12,A3,8 | NOP                  }
{ NOP                 | DLNW D14,A2,8 | NOP                  | DLNW D14,A2,8 | NOP                  }
{ NOP                 | NOP           | SAA.Q AC2,D4,D6      | NOP           | SAA.Q AC2,D4,D6      }
{ NOP                 | NOP           | SAA.Q AC4,D5,D7      | NOP           | SAA.Q AC4,D5,D7      }
{ NOP                 | NOP           | SAA.Q AC2,D12,D14    | NOP           | SAA.Q AC2,D12,D14    }
{ LBCB R8,SAD_loop    | NOP           | SAA.Q AC4,D13,D15    | NOP           | SAA.Q AC4,D13,D15    }
;5 delay slots
{ NOP                 | NOP           | ADDU.D D12,AC2,AC3   | NOP           | ADDU.D D12,AC2,AC3   }
{ NOP                 | MERGEA D6,D12 | ADDU.D D13,AC4,AC5   | MERGEA D6,D12 | ADDU.D D13,AC4,AC5   }
{ NOP                 | MERGEA D7,D13 | NOP                  | MERGEA D7,D13 | NOP                  }
{ NOP                 | ADDI A2,A2,16 | ADDU AC6,D7,D6       | ADDI A2,A2,16 | ADDU AC6,D7,D6       }
{ NOP                 | ADD  A3,A3,D10| ADDU D3,D3,AC6       | ADD  A3,A3,D10| ADDU D3,D3,AC6       }
;+++++5 delay slots+++++
{ NOP                 | BDR  D7       | NOP                  | BDT  D3       | NOP                  }
{ NOP                 | NOP           | NOP                  | NOP           | NOP                  }
{ NOP                 | NOP           | ADDU D14,D7,D3       | NOP           | NOP                  }
                                        ;D14=SAD
```

Figure 5.8: An example code for 16×16 SAD calculation in PACDSP.

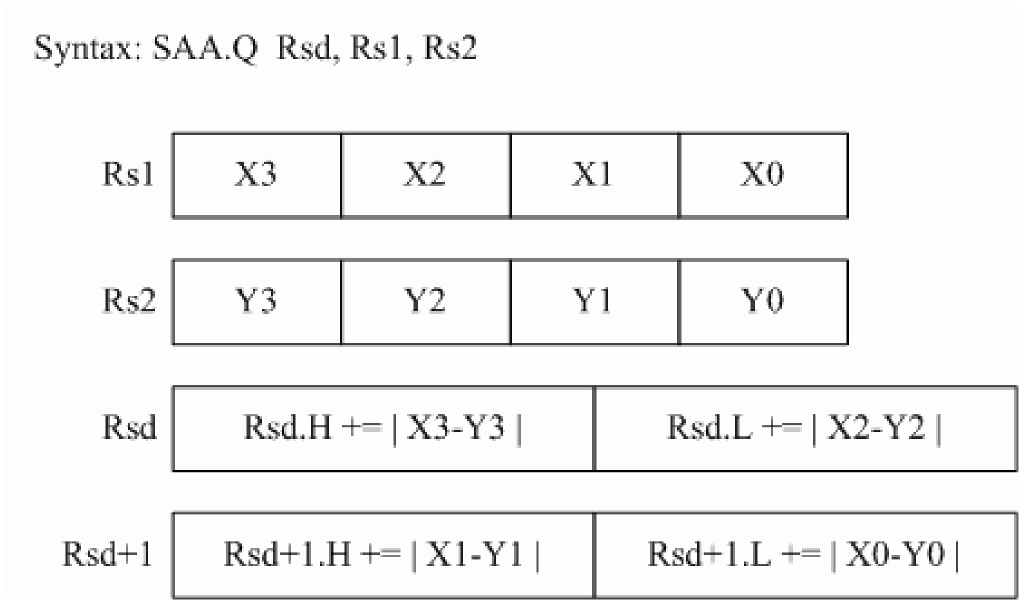


Figure 5.9: The syntax and operation of SAA.Q instruction.

Table 5.6: Comparison of SAD Implementation on Different Platforms

Block					Equivalent
Size	Designs	Processing units	Clock (MHz)	Cycles	Instruction Counts
16×16	TI C62x [21]	2 MUL, 6 ALU	200	272	2176
	TI C64x [22]	2 MUL, 6 ALU	600	67	536
	PACDSP v3.0 (ours)*	(1 Scalar), 2 AU, 2 L/S	200	114	456
8×8	TI C62x [21]	2 MUL, 6 ALU	200	80	640
	TI C64x [22]	2 MUL, 6 ALU	600	31	248
	PACDSP v3.0 (ours)**	(1 Scalar), 2 AU, 2 L/S	200	32	128

*Note: If considered having 5 processing units, then equivalent instruction counts = 570.

**Note: If considered having 5 processing units, then equivalent instruction counts = 160.

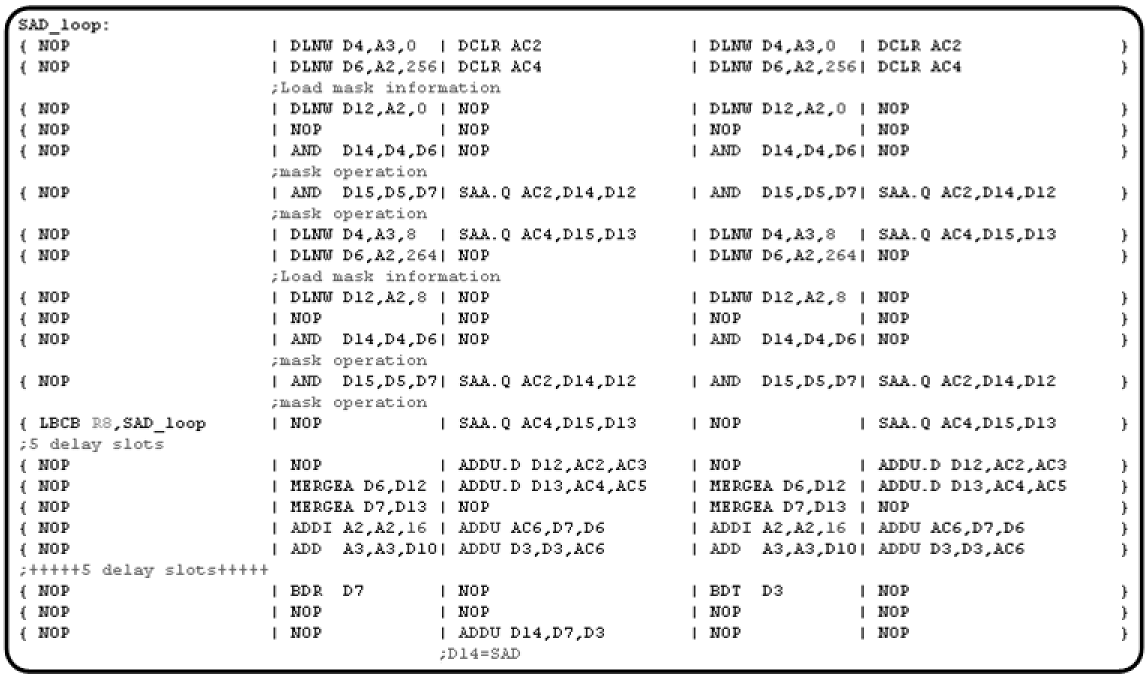
longing to the object. For this, a conditional operation is used in the reference code. However, in order to utilize the advantages of SIMD instructions, we use a masking method in place of conditional operation. That means we use the shape information to mask the reference data before the subtraction operation. The assembly code for masked SAD calculation in our implementation is shown in Fig. 5.10.

5.5 Simulation Results on PACDSP Instruction Set Simulator (ISS)

Before the dual-core implementation of object-based video encoder on the hardware system, we test and verify our assembly code for PACDSP on the instruction set simulator (ISS). The ISS is developed by the SoC Technology Center (STC) of Industrial Technology Research Institute in Chutung of Taiwan. The input file of the simulator is split through a parsing tool, "as2tic," which parses the assembly code into two parts, data and instructions. We can configure the ISS to decide which kinds of information we want to print out to files.

5.5.1 Statistics of Motion Estimation on ISS

The "tier_para" of motion estimation is set to 5. Table 5.7 shows the execution time obtained by performing the motion estimation for 1 P-VOP on ISS. The information about object size is listed in the second column, "MB Number," which means how many macroblocks containing object pixels are there within the VOP. The average cycles for each MB and breakdown for integer-pixel search and half-pixel search are also shown. The average cycles for integer-pixel search are related to the motion characteristics of the sequences. However, since the search points of half-pixel motion estimation for each MB is fixed, the average execution times for half-pixel searches are almost the same.



```
SAD_loop:
{ NOP                   | DLNW D4,A3,0  | DCLR AC2              | DLNW D4,A3,0  | DCLR AC2                 }
{ NOP                   | DLNW D6,A2,256| DCLR AC4              | DLNW D6,A2,256| DCLR AC4                 }
                        ;Load mask information
{ NOP                   | DLNW D12,A2,0 | NOP                   | DLNW D12,A2,0 | NOP                      }
{ NOP                   | NOP           | NOP                   | NOP           | NOP                      }
{ NOP                   | AND  D14,D4,D6| NOP                   | AND  D14,D4,D6| NOP                      }
                        ;mask operation
{ NOP                   | AND  D15,D5,D7| SAA.Q AC2,D14,D12     | AND  D15,D5,D7| SAA.Q AC2,D14,D12        }
                        ;mask operation
{ NOP                   | DLNW D4,A3,8  | SAA.Q AC4,D15,D13     | DLNW D4,A3,8  | SAA.Q AC4,D15,D13        }
{ NOP                   | DLNW D6,A2,264| NOP                   | DLNW D6,A2,264| NOP                      }
                        ;Load mask information
{ NOP                   | DLNW D12,A2,8 | NOP                   | DLNW D12,A2,8 | NOP                      }
{ NOP                   | NOP           | NOP                   | NOP           | NOP                      }
{ NOP                   | AND  D14,D4,D6| NOP                   | AND  D14,D4,D6| NOP                      }
                        ;mask operation
{ NOP                   | AND  D15,D5,D7| SAA.Q AC2,D14,D12     | AND  D15,D5,D7| SAA.Q AC2,D14,D12        }
                        ;mask operation
{ LBCB R8,SAD_loop      | NOP           | SAA.Q AC4,D15,D13     | NOP           | SAA.Q AC4,D15,D13        }
;5 delay slots
{ NOP                   | NOP           | ADDU.D D12,AC2,AC3    | NOP           | ADDU.D D12,AC2,AC3       }
{ NOP                   | MERGEA D6,D12 | ADDU.D D13,AC4,AC5    | MERGEA D6,D12 | ADDU.D D13,AC4,AC5       }
{ NOP                   | MERGEA D7,D13 | NOP                   | MERGEA D7,D13 | NOP                      }
{ NOP                   | ADDI A2,A2,16 | ADDU AC6,D7,D6        | ADDI A2,A2,16 | ADDU AC6,D7,D6           }
{ NOP                   | ADD  A3,A3,D10| ADDU D3,D3,AC6        | ADD  A3,A3,D10| ADDU D3,D3,AC6           }
;+++++5 delay slots+++++
{ NOP                   | BDR  D7       | NOP                   | BDT  D3       | NOP                      }
{ NOP                   | NOP           | NOP                   | NOP           | NOP                      }
{ NOP                   | NOP           | ADDU D14,D7,D3        | NOP           | NOP                      }
                                        ;D14=SAD
```

Figure 5.10: Assembly code of masked 16×16 SAD calculation in our implementation.

5.5.2 Statistics of Shape Coding on ISS

The execution time statistics of shape coding are shown in Table 5.8, which are obtained by implementing the shape coding for 1 P-VOP on ISS. The information about object size and the percentage of boundary MBs over total MBs is given in the second column. All the MBs call the function "ShapeInterMB" but only the boundary MBs would do motion search on the alpha plane. That means, the execution time of "ShapeInterMB" is dependent on the percentage of boundary MBs over total MBs. Another fact affecting the

Table 5.7: Execution Time of Motion Estimation for 1 P-VOP of QCIF on ISS

Test seq.	MB	Execution Time (cycles)			Execution Time/MB (cycles)		
(QCIF)	Number	Total	Integer-Pel	Half-Pel	Total	Integer-Pel	Half-Pel
foreman	47	2,175,977	1,637,881	512,038	46,297	34,849	10,894
akiyo	48	1,336,459	820,576	491,370	27,843	17,095	10,237
stefan	15	684,103	514,735	156,792	45,607	34,316	10,453

execution time is the motion characteristics. Take the almost stationary sequence, akiyo, for example. About half of the boundary MBs find an identical BAB over its search range. It can not only terminate the search procedure but also skip the CAE operation. That is why the execution time of the sequence akiyo is much less than the other two sequences. In addition, the skip ratio of mode section mentioned in chapter 4 would also affect the execution time of CAE_MB.

5.6 Conclusion

We used several optimization techniques to improve our implementation of the MPEG-4 video encoder on PACDSP. We first discussed some algorithm optimization techniques in the previous chapter. Then we rescheduled a dual-core implementation on the PAC system and tried to eliminate all the unnecessary stalls in our assembly code on the DSP. We further distributed the regular and independent computations into two clusters as much as possible. If there were any consecutive loads or stores, we replaced the original program with double-loads or stores. In addition, we also applied the general code optimization techniques discussed in this chapter. Now we show the speed-ups of these optimization methods for shape coding and motion coding. Since we only implement the transformer part of texture coder on DSP, the performance of texture coder in our implementation will be left to the next chapter.

Table 5.9 shows the performance of algorithm optimization on ARM926EJ-S. We can find that more than 60% of computation is saved in the motion coder and more than 55%

Table 5.8: Execution Time of Shape Coding for 1 P-VOP of QCIF on ISS

Test seq.	Boundary MBs	Execution Time (cycles)				
(QCIF)	/Total MBs	Total	ShapeInterMB	%	CAE_MB	%
foreman	23/47	1,516,959	750,482	49.47	732,613	48.29
akiyo	23/48	570,231	327,368	57.41	207,889	36.46
stefan	15/15	900,265	381,647	42.39	506,918	56.31

of computation is saved in the shape coder.

Table 5.10 shows the results from implementing the optimized coder on PACDSP. We can find that the implementation on PACDSP is much faster than on ARM926EJ-S. The first reason is that we utilized the DSP architecture to optimize our implementation. In addition, we have a well-scheduled hand code on PACDSP, while C-level coding is used on the ARM926EJ-S platform.

We have placed the most computation intensive parts of the MPEG-4 object-based video encoder on PACDSP. We will discuss the dual-core implementation and performance in the next chapter.

Table 5.9: Execution Time of P-VOP Motion Estimation and Shape Coding after Algorithm Optimization on ARM926EJ-S

Coder	Test Seq. (QCIF)	Execution Time (cycles)		
		Original	Algorithm Optimized	% of reduction
Motion	foreman	79,675,422	28,433,273	64.31
	akiyo	48,952,190	17,296,872	64.67
	stefan	24,251,478	7,361,425	69.65
Shape	foreman	35,191,526	12,984,353	63.10
	akiyo	12,907,962	5,342,844	58.61
	stefan	18,419,663	6,751,031	63.35

Table 5.10: Execution Time of P-VOP Motion Estimation and Shape Coding after Optimization on PACDSP

Coder	Test Seq. (QCIF)	Execution Time (cycles)		
		Original†	Architecture Optimized	% of reduction
Motion	foreman	28,433,273	2,175,977	92.35
	akiyo	17,296,872	1,336,459	92.27
	stefan	7,361,425	684,103	90.71
Shape	foreman	12,984,353	1,516,959	88.32
	akiyo	5,342,844	570,231	89.33
	stefan	6,751,031	900,265	86.66

†Original means the execution time after algorithm optimization on ARM926EJ-S.

Chapter 6

Performance Analysis and Implementation Results

In this chapter, we analysis the performance of our implementation of MPEG-4 object-based video encoder, including the program size, the encoding frame rate, and the coding quality.

6.1 Performance Analysis

In this section, we discuss the code size and data size of our implementation. In order to prevent the problem of cache miss, we should ensure that the total size of program and data are smaller than 32 kB and 64 kB, respectively, which are provided by PACDSP. Then, we estimate the frame rate for our implementation of MPEG-4 object-based video encoder.

6.1.1 Code Size

Table 6.1 shows the code size of the three coders and the major functions of each coder in MPEG-4 object-based video encoder on PACDSP. The program size of "ShapeCodingIntraCAE" and "ShapeCodingInterCAE" are the biggest, whose purpose is to encode the binary shape information. The second are "ShapeInterMB" and "16x16FullPel_ME," which do the motion estimation over alpha plane and luminance plane, respectively. Since the instruction memory of PACDSP is limited to 32 kB, we need to care the total size of our program. The total program size is 26,904 bytes in our implementation, and it is smaller than the instruction cache size. Therefore, no cache miss will happen in our implementation.

6.1.2 Data Size

The data memory of PACDSP is limited to 64 kB, and Table 6.2 shows the data memory used of each coder in our implementation. Since the coded unit is one VOP per coding in our implementation, we should ensure the designed memory space is large enough to load the VOP of any size. In the worst case, the VOP size is equal to the frame size, 176×144. We decide to allocate a frame size (176×144 bytes) to put the input image plane, and Table 6.2 shows the data size profile in the worst case.

In addition, because the number of registers is limited, we may need some memory space for storing the calculated results in the encoding procedure. And such memory space is designated "Temporary".

In ShapeCoder, current alpha plane (176×144) and reference alpha plane (unextended, 176×144) are needed to be the input data. We allocate 3 kB of memory to store the output bitstream of the coded shape information.

In MotionCoder, current luminance plane (176×144) and reference luminance (extended, 208×176) are needed to be the input data. The first output data are the motion vectors and the second is the compensated luminance plane (176×144), which is saved in the same memory space as current luminance plane.

In Transformer, current luminance and chrominance plane are the input data. Since the residual data is in the range [-255,255], it need 2 bytes to save one pixel. The memory is unavailable to load the luminance and chrominance data in frame size ($176\times144\times1.5\times2$ = 76032 bytes). However, the data of each MB are is independent in the transformer, and we can only input the non-transparent MB. Therefore, we allocate the memory space that can hold 80 residual macroblocks ($80\times6\times64\times2$ = 61440 bytes). The output data are quantized coefficients and reconstructed data, and they are both saved in the same

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

Coder Category	Function Name	Code Size (Bytes)	%
ShapeCoder	ShapeInterMB	2824	10.50
	ShapeCodingIntraCAE	3268	12.15
	ShapeCodingInterCAE	3256	12.10
	Others	2908	10.81
MotionCoder	16x16FullPel_ME	2764	10.27
	8x8FullPel_ME	1700	6.32
	InterPolate_SubPel	420	1.56
	Inter16_SubPelME	1036	3.85
	Inter8_SubPelME	1400	5.20
	MC_Luma	896	3.33
	Others	1012	3.76
Transformer	BlockDCT	772	2.87
	BlockQuantH263	492	1.83
	BlockDequantH263	344	1.28
	BlockIDCT	672	2.50
	Others	1068	3.97
Total		26904	100.00

memory space as input data.

6.1.3 Frame Rate Estimation

After our optimization, we now estimate the frame rate of our implementation. First, the frame rate of single-core implementation (only ARM) is shown in Table 6.3. It is noted that the cycles are obtained by encoding 1 I-VOP or P-VOP for each test sequence with a fixed QP, 4.

Before we estimate the frame rate of dual-core implementation on PAC system, we introduce the operating frequency of two cores and the transmitting frequency of the bus as follows.

- ARM core: 200 MHz
- PACDSP core: 200 MHz (real chip)
- Bus: 35 MHz (32 bits width)
- Write data: 2 cycles
- Read data: 1 cycle

There are three major parts in Tables 6.4 and 6.5, which are ARM core, PACDSP core and the data transmitted on bus. The cycles of the ARM and PACDSP means the

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

Coder	Memory Usage (Bytes)			Total Size
Category	Input	Output	Temporary	(Bytes)
ShapeCoder	55,728	3,072	2,597	61,397
MotionCoder	62,459	26,235[†]	1,743	65,093
Transformer	62,162	61,528[††]	608	62,858

[†]It has 25,344 bytes that uses the same memory space as input data.

[††]It has 61,440 bytes that uses the same memory space as input data.

Table 6.3: Frame Rate Estimation of Single-Core Implementation

I or P-VOP	Test Seq.(QCIF)		foreman	akiyo	stefan
I-VOP	ARM	(cycles)	19,083,255	22,791,904	9,683,303
	Execution Time	(ms)	95.42	113.96	48.42
	Frame Time	(fps)	10.5	8.8	20.7
P-VOP	ARM	(cycles)	54,578,073	38,005,537	21,053,337
	Execution Time	(ms)	272.89	190.03	105.27
	Frame Time	(fps)	3.7	5.3	9.5

Note: Operating frequency of ARM926EJ-S is 200 MHz.

execution time of each core. We can get the expected execution time (ms) by dividing them by the operating frequency. Since our implementation is on a dual-core system, ARM part need to write data to the memory which could accessed by PACDSP. After the coding is finished by PACDSP, ARM part need to read the output data from the specific memory. Note that the clock-rate of bus is 35 MHz and the bus width is 32 bits. In addition, two cycles are taken for writing data to memory, and only one cycle for reading data from memory.

We separate the execution time into several groups. The first group, "Others," coded only by ARM core, includes the functions: reading frame data, VOP formation, output bitstream to disk, subsampling, VOP padding (only for inter coding). For intra coding, the texture padding and shape coding are coded in parallel by ARM core and PACDSP core, respectively. After the shape bitstreams are transmitted from PACDSP to ARM part and the texture data are updated from ARM to PACDSP, we start forward transform on PACDSP. Then, another parallel coding of variable length coding (VLC) and inverse transform follows the quantized coefficients are transmitted from PACDSP to ARM. However, we can see the total execution time of our implementation for intra encoding on Table 6.4, and the percentage of the total execution time for each group is shown as "%" in the table. Finally, we can estimate the frame rate of each sequence, which is shown as fps (frame per second) in the table.

Similar to intra encoding, the execution time of our implementation for inter encoding is shown on Table 6.5. Note that, for the group coded in parallel, we only need to consider the longer part when we compute the total execution. In other words, the percentage of the shorter part will be zero of the total execution time.

For the sequence of "stefan" with the smallest VOP size, we can get the best frame rate which are 33.9 and 43.0 frames per second for the intra encoding and inter encoding, respectively. For the sequence "akiyo" with the biggest VOP size, which takes a lot of cycles on VOP formation, we can get about 18-fps for both intra and inter encoding.

Compared to single-core implementation, intra encoding has a averaged speed-up ratio about 124.5%, and the averaged speed-up ratio of inter encoding is about 353.6%.

6.2 Coding Quality and Bit Rates for Different QP

In MPEG-4 video encoder, the quantization follows the DCT computation. Therefore, the value of quantization step affects the quantized coefficients, which is highly related to bit-rate and reconstructed video quality. To have a further understanding of how QP affects the two issues of video encoder, we do some analysis of different QP values in this section.

In our analysis, we encode 1 I-frame and 100 P-frame in different QP, and the averaged texture bits and PSNR are shown in Table 6.6. Since larger QP introduces more quantization distortion, the quality decreased with the QP value increased. As a result, more coefficients are quantized to zero, and the texture bit-rate decreased as well. In addition, the percentage of skipped blocks increased with larger QP value. Therefore, that makes the execution time of transformer reduced, and the analysis have been shown in previous chapter.

Table 6.4: Frame Rate Estimation for Intra Encoding of Dual-Core Implementation

Test Seq. (QCIF)		foreman	akiyo	stefan
ARM	(cycles)	4,049,467	7,250,105	3,360,106
Others	%	54.69	66.91	72.17
ARM	(cycles)	414,107	410,254	231,431
TexturePadding	%	0	0	0
PACDSP	(cycles)	565,928	510,028	321,888
ShapeCoding	%	7.64	4.71	6.91
PACDSP	(cycles)	169,818	170,404	46,190
Forward Transform	%	2.29	1.57	0.99
ARM	(cycles)	2,578,148	2,867,222	914,104
VLC	%	34.82	26.46	19.63
PACDSP	(cycles)	167,278	176,434	51,263
Inverse Transform	%	0	0	0
Bus (Write)	(bytes)	63,620	65,924	25,988
Bus (Read)	(bytes)	76,980	76,980	24,756
	%	0.49	0.35	0.29
Execution Time	(ms)	37.02	54.18	23.28
Frame Rate	(fps)	27.0	18.5	43.0

Table 6.5: Frame Rate Estimation for Inter Encoding of Dual-Core Implementation

Test Seq. (QCIF)		foreman	akiyo	stefan
ARM	(cycles)	4,640,664	7,760,793	3,737,575
Others	%	48.44	70.73	63.33
ARM	(cycles)	346,208	492,459	291,998
EncodeVOPHeader	%	0	0	0
PACDSP	(cycles)	2,175,977	1,336,459	684,103
MotionCoding	%	22.72	12.18	11.59
ARM	(cycles)	589,291	376,944	162,904
MC_Chroma & TexturePadding	%	0	0	0
PACDSP	(cycles)	1,516,959	570,231	900,265
ShapeCoding	%	15.83	5.20	15.25
PACDSP	(cycles)	174,424	177,074	46,474
Forward Transform	%	1.82	1.61	0.79
ARM	(cycles)	1,009,160	1,063,955	510,228
VLC	%	10.54	9.70	8.64
PACDSP	(cycles)	55,185	59,824	43,142
Inverse Transform	%	0	0	0
Bus (Write)	(bytes)	125,372	135,868	511,32
Bus (Read)	(bytes)	92,472	96,312	31,800
	%	0.64	0.60	0.40
Execution Time	(ms)	47.89	54.86	29.51
Frame Rate	(fps)	20.9	18.2	33.9

Table 6.6: Effects on Quality and Bit-Rate of Different QP values

Test Seq. (QCIF)	Quality and Bit-Rate	QP = 2	QP = 4	QP = 7	QP = 10	QP = 13
foreman	Texture Bits (bpv)	12289.70	4833.48	2180.42	1140.04	822.54
	PSNR_Y (dB)	42.79	37.91	34.35	32.15	30.49
	PSNR_U (dB)	44.38	40.97	37.01	35.06	32.82
	PSNR_V (dB)	44.72	40.93	36.82	34.84	32.75
akiyo	Texture Bits (bpv)	7108.33	2544.56	1309.92	722.47	610.40
	PSNR_Y (dB)	42.43	37.21	33.40	31.08	29.46
	PSNR_U (dB)	45.76	42.03	37.13	34.70	32.62
	PSNR_V (dB)	45.18	41.69	36.92	34.84	32.64
stefan	Texture Bits (bpv)	8246.52	3801.51	1785.19	943.82	589.71
	PSNR_Y (dB)	40.53	34.45	30.23	27.37	25.80
	PSNR_U (dB)	40.58	35.86	32.64	30.78	29.06
	PSNR_V (dB)	40.52	35.69	32.27	30.25	28.60

Note: bpv = bits per VOP.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

In this thesis, we considered the real-time implementation of MPEG-4 object-based video encoder on PAC system which was a dual-core platform.

We first focused on the correct of encoding the bitstream, and the coded bitstream have been verified with the reference software of MPEG-4, MoMuSys. Then, we analyzed the statistics of the MPEG-4 object-based video encoder on PC. Therefore, we had an initial understand of the encoding flow and the critical part of computation. According to the analysis, we designed our dual-core structure and implemented the DSP part on the PACDSP simulator.

After the implementation was verified, we further analyzed the encoding algorithm and coding flow to find if there was any removable computation. Based on our analysis, we optimized the program sequence to reduce the computation complexity without too much quality loss or bit-rate increased. In addition, we also utilized several general software optimization techniques, such as static rescheduling, loop-unrolling, and software-pipelining to reduce the stalls.

Finally, the optimization results were discussed. For the best case, stefan, which has the smallest VOP size, we can encode the MPEG-4 video data over 33 frames and 43 frames per second for intra and inter encoding, respectively. And the program size was about 27KB, which was smaller than the instruction cache size. In addition, the used data size of each coder was also under the limit of memory provided on PACDSP. Therefore, no cache missing problem happened in our implementation. In conclusion, the performance and quality of our implementation of MPEG-4 object-based video encoder on PAC system was competitive.

7.2 Future Work

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

- Add some popular fast motion estimation algorithm

  Motion estimation is the most computational part in MPEG-4 video encoder. However, many fast motion estimation algorithm has been proposed, and used popularly. We consider to add some fast motion estimation algorithm for flexibility.

- Data structure refinement

  The data structure is very important to the implementation on DSPs. 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 loading balance

  We can find the estimated frame rate in previous chapter, and the bottleneck is still the execution time of ARM part. If we can move more computation to PACDSP part, the performance will be improved by the advantage of dual-core implementation.

- Demonstration on PAC system

  We have done the single-core demonstration on ARM926EJ-S platform, and the major coder on DSP part have been verified on instruction set simulator (ISS) of PACDSP. Since some coding constraints are not included on the ISS, we still need to do some modification on our coding, and finally demonstrate our dual-core implementation on the PAC system.

- Add other MPEG-4 tools

  To simplify our implementation, the error-resilience tool in MPEG-4 simple profile is neglected. However, this tool is very important when the bitstream is transmitted through real channels. In the future, we need to implement the techniques of error-resilience, such as resynchronization, data partition, and reversible variable length coding (RVLC). Moreover, the other advanced profiles of MPEG-4 video compression technique can be implemented 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, May. 2006.

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

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

[6] 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.

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

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

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

[10] Kun-Bin Lee, Jih-Yiing Lin, and Chein-Wei Jen, “A Multisymbol Context-Based Arithmetic Coding Architecture for MPEG-4 Shape Coding,” *IEEE Trans. Circuits Systems Video Technology.*, vol. 15, no. 2, Feb. 2005.

[11] 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.

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

[13] 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.

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

[15] 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.

[16] 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.

[17] 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.

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

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

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

[21] 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.

[22] Texas Instuments, *TMS320C64x Image/Video Processing Library — Programmers Reference,* Literature no. SPRU023B, Oct. 2003.

[23] 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.

[24] 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.

[25] 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.

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

自傳

江政達，男，民國七十一年十月四日出生於台北縣板橋市。高中就讀於國立台灣師範大學附屬高級中學，民國九十四年六月畢業於交通大學電信工程學系，並於同年九月進入交通大學電子工程研究所碩士班就讀，於民國九十六年六月取得碩士學位，論文題目為:『MPEG-4 物件視訊編碼器在 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。