國立交通大學

電機學院 IC 設計產業研發碩士班

碩　士　論　文

JPEG2000 編碼器之加速 和 TI DSP 系統平台上之實現

Acceleration and Implementation of JPEG2000 Encoder on TI DSP Platform

研 究 生：劉建志

指導教授：杭學鳴　博士

中 華 民 國 九 十 五 年 十 二 月

JPEG2000 編碼器之加速和 TI DSP 系統平台上之實現

Acceleration and Implementation of JPEG2000 Encoder on TI DSP Platform

研究生：劉建志　　Student: Chien-Chih Liu

指導教授：杭學鳴　　Advisor: Dr. Hsueh-Ming Hang

國 立 交 通 大 學

電機學院 IC 設計產業研發碩士班

碩 士 論 文

A Thesis

Submitted to College of Electrical and Computer Engineering

National Chiao Tung University

in Partial Fulfillment of the Requirements

for the Degree of Master

in

Industrial Technology R & D Master Program on

IC Design

December 2006

HsinChu, Taiwan, Republic of China

中華民國九十五年十二月

JPEG2000 編碼器之加速
和 TI DSP 系統平台上之實現

研究生：劉建志　　　　　　　　　　　　指導教授：杭學鳴 博士

國立交通大學
電機學院 碩士班

摘要

由於數位影像應用的逐漸普及，為了提供更有壓縮效率以及支援更多功能的影像處理，一個新一代的靜態影像壓縮標準 JPEG2000 於是產生。它在高壓縮率下也能夠提供相當好的主觀品質，此外，它在壓縮效能和傳送位元流時提供了更細緻的調整功能。然而，JPEG2000 在計算上的複雜度相當的高，在本論文中，我們將 JPEG200 編碼器實現在 TI DSP 平台上。我們根據 JPEG2000 中最複雜的 Tier 部份，提出兩種改善方法，並且加上 TI DSP 最佳化的各種相關工具來進行加速。

我們的參考軟體採用了 openJPEG ver.1.0，因為這套軟體的小波轉換模組已經使用一維補償式結構(lifting scheme)來進行加速，所以針對佔了整個編碼器九成運算量的 Tier1 模組，我們先探討常見的改善方式，並實際在我們所使用的平台上做測試，然後我們提出了兩種改進方法，一種稱為 VGOSS(Variable group of sample skip)，另外一種則是修改 VGOSS 的方式，來達成減少運算量的目的。這個方式是將需要編碼的資料紀錄起來，減少對不需要的編碼的資料所浪費的檢查時間。另外，我們改變了原來編碼的順序，提供更快的運算架構。當我們對影像使用無失真編碼時，除了採用所提供的加速方法，還有使用 DSP 的編譯程序最佳化、及程式碼的加速技術、還有快取記憶體的重新配置等功能，在最後的在 DSP 系統上的實驗數據顯示，我們使用以上所有技術後，可以比最原始的效能還要快 32 倍，如果比較在同樣的 DSP 最佳化設定還有記憶體配置下，我們的快速演算法仍然可以減少 45%的運算量。

關鍵字：

JPEG2000、TI DSP、DSP 系統加速、EBCOT

Acceleration and Implementation of JPEG2000 Encoder on TI DSP Platform

Student: Chien-Chih Liu Advisor: Dr. Hsueh-Ming Hang

College of Electrical and Computer Engineering

National Chiao Tung University

Abstract

Because the usage for digital imagery gets increasingly popular, to enhance the compressed image efficiency and features, a new still image coding standard called JPEG2000 was proposed. It provides an excellent subjective quality at low bit rates. It also offers fine granularity scalability in compression efficiency and transmitting compressed bit stream. However, JPEG2000 is also very complicated in computational complexity. In this thesis, we implement a JPEG2000 encoder on the TI DSP platform. We propose two speed-up methods and use the TI DSP optimization tools to accelerate the Tier1 module, which is the most complex part in the JPEG2000 standard.

We start with the ver.1.0 OpenJPEG reference software, which has adopted the 1-D lifting scheme to accelerate the DWT module. Thus we focus on the Tier1 module, which takes about 90% of total computing time. We study the previous methods first and examine their effectiveness on our DSP platform. Then, we propose two improved methods, one is called VGOSS (Variable Group Of Sample Skip), and the other is a modified VGOSS method. We eliminate the unnecessary checking cycles by recording the NBC (Need-to-Be-Coded) samples on a list. Furthermore, the sample index is reordered to facilitate fast execution. In the DSP implementation of the proposed methods, we use code acceleration techniques and DSP compiler-level optimization. We also tune the cache allocation to reduce memory access time. The experimental results show that the best performance is up to 32 times faster than the original program without any optimization on the DSP platform. If the original program is compiled with the DSP optimization tools and proper cache assignment, our fast algorithm can still reduce the computation by 45%.

Key words:

JPEG2000、TI DSP、DSP platform acceleration、EBCOT

誌謝

在這兩年的研究生涯中，經歷了許多挑戰和挫折，感謝神能讓我在這裡學習和成長，我要感謝我的指導教授杭學鳴老師，他的細心指導和豐富的學識經歷，讓我獲益良多，當我研究遇到瓶頸時，老師總是能以關心體諒的方式，讓我重新拾起信心，在這短暫的兩年，獲得的比我想像中要多更多。

我還要感謝通訊電子與訊號處理實驗室的夥伴們，在這裡的研究和討論讓我解決許多課業和研究上的問題，VK 學長總是給了我相當棒的提醒和建議，還要感謝峰誠學長、俊榮學長、崑健學長、繼大學長、雄哥、思浩、Osban，大師、Mark、Stan、小新、John、家賢、阿竹、蔡蟲、Geni、旻弘…和實驗室的所有人，你們讓我的研究生生活變得更多彩多姿，還有許多幫助和支持我的朋友們，謝謝你們。

最後要感謝家人和女友的支持和關心，讓我能專注在自己的研究上，不論在生活上或是求學上也給了我最大的鼓勵和支持。

謝謝所有陪我走過這一段歲月的師長、家人和朋友!

誌於 2006.12 風城交大

建志

Contents

List of Figures

List of Tables

Chapter 1 Introduction

1.1 Introduction

Digital image is an essential part of our daily information in the world today. The standards for the efficient representation and interchange of digital images are important. JPEG2000 is a well-known image algorithm for its excellent coding performance especially in low bit-rate. It is the most recent addition to a family of international standards developed by the Joint Photographic Experts Group (JPEG). This group operates under the auspices of Joint Technical Committee 1, Subcommittee 29, Working Group 1 (JTC 1/SC 29/WG 1), a collaborative effort between the International Organization for Standardization (ISO) and International Electro technical Commission (IEC). The JPEG committee has already released the JPEG and JPEG-LS standards. The JPEG standard is the most popular image compression in recent years. However, the JPEG committee intends to create a new image coding system for different types of still images (bi-level, gray-level, color, multi-component), with different characteristics (natural images, scientific, medical, remote sensing, text, rendered graphics, and etc.). The targets of the JPEG2000 coding system are expected to be the low bit-rate operation with a rate-distortion and subjective image quality performance superior to the existing image standards.

The JPEG2000 standard implements an entirely new way of compressing images based on the wavelet transform, in contrast to the discrete cosine transform (DCT) used in the JPEG standard. It also supports lossy and lossless compression of single-component (gray level) images and multi-component (color) images. In addition to this basic compression functionality, a number of other features are provided, including progressive recovery of an image by fidelity or resolution, region of interest coding, random accessing and so on. However, the complexity of JPEG2000 algorithm is the most critical issue in the implementation on an embedded system. Typically, the Embedded Block Coding with Optimized Truncation (EBCOT) is the major part and computationally intensive in the JPEG2000 algorithm. The EBCOT employs a post-compression Rate-Distortion Optimization (RDO) tool, which truncates the bit-stream at the target bit-rate providing optimal image quality. Because of these tools, the JPEG2000 algorithm has a much higher computation than the JPEG algorithm. In order to reduce the cost and power consumption, we analyze and accelerate the JPEG2000 algorithm in this study.

1.2 Overview of the Thesis

In this thesis, the JPEG2000 encoder is implemented on an embedded system-a TIDSP platform. A few speed-up methods are adopted in our encoder. In the Chapter 2, the concepts of the JPEG2000 algorithm are introduced and all coding modules are presented in the following sections. Chapter 3 introduces the implementation environment including the DSP platform, coding development tools, and some typical optimization methods. In Chapter 4, the JPEG2000 encoder is profiled and analyzed. Some previous accelerating methods are reviewed and modified in our DSP platform. Then, we propose our improved methods to accelerate the JPEG2000 encoder in Chapter 5 and extensive experiments using different methods are also presented in Chapter 5. Finally, we give a summary of this project and also discuss the future possible work in Chapter 6.

Chapter 2 Conspectus of JPEG2000 Algorithm

The JPEG standard has been in use for almost a decade now. It provides a valuable tool during all these years, but it cannot fulfill the advanced requirements for image coding of today. The JPEG2000 standard provides a set of features that are important to many high-end and emerging applications by adopting new technologies. This chapter introduces the feature set and provides an overview of the Part1 of JPEG2000 standard Part 1. It is the core of the JPEG2000 for image coding system. The details of JPEG2000 Part 1 can be found in [1].

2.1 Introduction to JPEG2000

Starting from March 1997, a new call for contributions was launched for the development of a new standard for the compression of still images, the JPEG2000 standard [1], [2]. The requesting compression technologies had been submitted to an evaluation during the November 1997 WG1 meeting in Sydney, Australia. The JPEG2000 standard has been achieved many desired features including different types of still image, different characteristics, and different imaging models within a unified system. The most important features [7] of JPEG2000 algorithm are listed as below.

Superior low bit-rate performance:

While superior performance at all bit-rates was considered desirable, improved performance at low bit-rate (e.g. below 0.25 bpp), with respect to JPEG, was considered to be an important requirement for JPEG2000. JPEG2000 has a compression advantage over JPEG of roughly 20% and a subjective quality benefit.

Continuous-tone and bi-level compression:

Seamless compression of image components (e.g., R, G, or B), each from 1 to 16 bits deep, was desired from one unified compression architecture.

Progressive transmission by pixel accuracy and resolution:

Progressive transmission that allows images to be reconstructed with increasing pixel accuracy or spatial resolution is essential for many applications. For examples, World Wide Web, image archival and printers, are common applications.

Lossless and lossy compression:

JPEG2000 provides both lossless and lossy compression, again from single compression architecture. It is desired to provide lossless compression in the natural course of progressive decoding.

Region-of-Interest Coding:

Some parts of an image are more important than others, and would like to be transmitted with better quality and less distortion than the rest of the image. Users can define certain ROI's in the image to be coded and transmitted first.

Random code-stream access and processing:

This feature allows users to define certain ROI's in the image to be coded and transmitted with less distortion than the rest of the image. Besides, rotation, filtering, translation, scaling and feature extraction are supported.

Robustness to bit-errors:

It is desirable to consider robustness to bit-errors while designing the codestream. In the noisy communication channels (e.g., wireless), proper design of the codestream can aid subsequent error correction systems in alleviating catastrophic decoding failures.

Open architecture:

It is desirable to allow open architecture to optimize the system for different image types and applications. A decoder is only to implement the core tool set and a parser that understands the codestream. Furthermore, unknown tools could sent from the source and be adopted by the decoder.

Content-based description:

Image archival, indexing and searching is an important in image processing. Content-based description of images might be available as part of the compression system.

Side channel spatial information (transparency):

Side channel spatial information such as alpha planes and transparency planes are useful for transmitting information for processing the image for display, printing or editing.

Protective image security:

Protection of a digital image can be achieved by means of watermarking, stamping, encryption, and labeling. The SPIFF has implemented labeling method, and JPEG2000 must be easy to achieve the target.

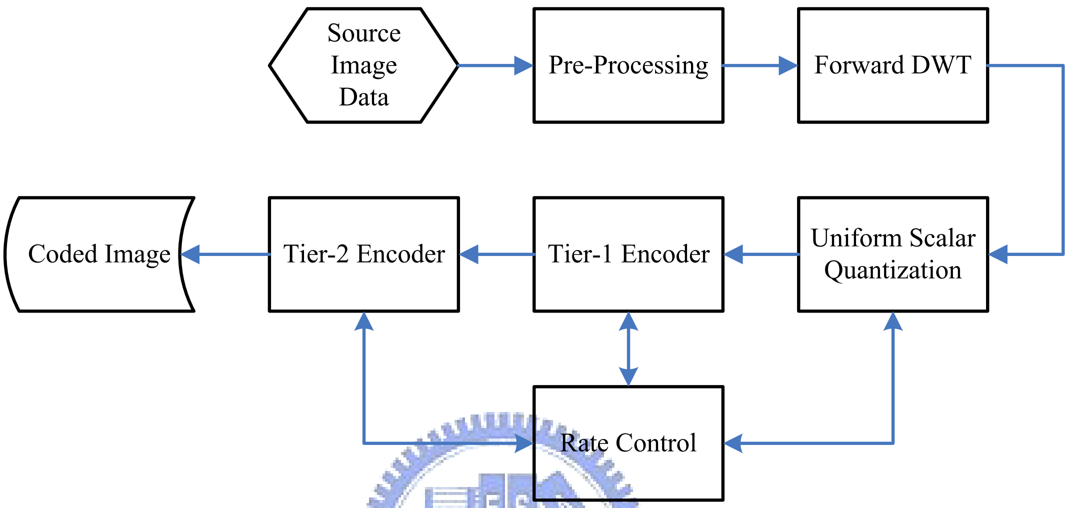


Figure 2-1 General block diagram of JPEG2000 encoder [1]

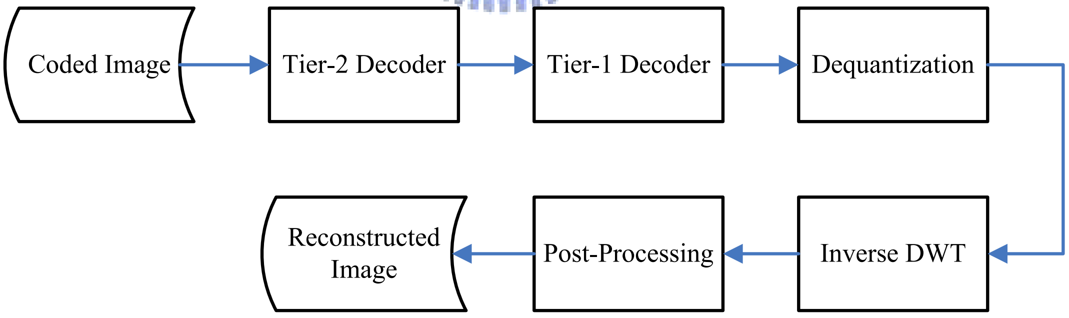


Figure 2-2 General block diagram of JPEG2000 decoder [1]

Due to above-mentioned attractive features, JPEG2000 has a very large potential application base. Some possible application areas include: document imaging, digital photography, desktop publishing, Internet, image archiving, medical imaging, remote sensing, and web browsing. The JPEG2000 standard compression engine (Encoder and Decoder) is illustrated in block diagrams in Figure 2-1 and Figure 2-2. It is comprised of numerous parts, several of which are listed in Table 2-1. Part 2 [3] and Part 3 [4] describe extensions to the baseline codec that are useful for certain specific applications such as intraframe-style video compression. For convenience, we will refer to the codec defined in Part 1 of the standard as the baseline codec. Before introducing the major block of the codec, we should know that the most parts of the JPEG2000 standard are written from the point of view of the decoder. Besides, the decoder is the reverse of the encoder. We will only describe the JPEG2000 encoding tools in the following sections.

Part	**Title**	**Purpose**
1	Core coding system	Specifies the core codec for the JPEG2000 family of standard
2	Extensions[3]	Specifies additional functionalities that are useful in some applications but need not be supported by all codec
3	Motion JPEG2000[4]	Specifies extensions to JPEG2000 for intraframe-style video compression
4	Conformance testing[5]	Specifies the procedure to be employed for compliance testing
5	Reference software[6]	Provides sample software implementations of the standard to serve as a guide for implementations

Table 2-1 Part of the JPEG2000 standard

2.2 Pre-Processing

The Pre-Processing block includes three types of processes, which are "Image Tiling", "DC Level Shifting", and "Component transformations". We will describe these terms as follows.

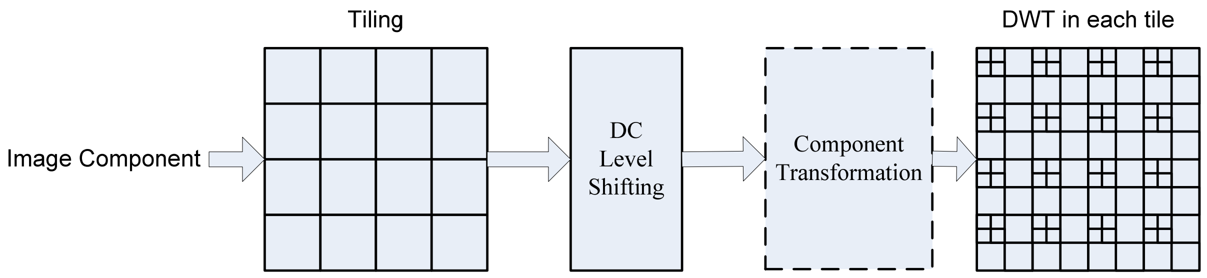


Figure 2-3 Tiling, DC-Level shifting, and Component transformation (optional)

2.2.1 Image Tiling

The standard operations, including component mixing, wavelet transform, quantization and entropy coding, works on image tiles which are the partition of the original image. The image tiles are rectangular non-overlapping blocks which are compressed independently. Tiling reduces memory requirements, and since they are reconstructed independently, they can be used for decoding specific parts of the image instead of the whole image.

2.2.2 DC Level Shifting

After tiling image, all samples of the each tiles are dc level shifted by subtracting the same quantity 2^{P-1}, where P is the component's precision. DC level shifting is performed on samples of components that are unsigned only.

2.2.3 Component Transformation

The followed stage is an optional inter-component transformation. It reduces the correlation between components, and lead to improved coding efficiency [8]. The JPEG2000 supports multiple-component image, and different bit depths. For the reversible (i.e. lossless) systems, the only requirement is that the bit depth of each output image component must be identical to the bit depth of the corresponding input image component. The JPEG2000 supports two different component transforms, irreversible component transformation (ICT) for lossy coding and reversible component transformation (RCT) for lossless or lossy coding. All image component samples $I_0(x, y), I_1(x, y), I_2(x, y)$, corresponding to the first, second, and third components, produce transform samples $Y_0(x, y), Y_1(x, y), Y_2(x, y)$. The forward and inverse RCT are achieved by means of (2.2-1) and (2.2-2). The other one, ICT, refers to (2.2-3) and (2.2-4).

$$\begin{pmatrix} Y_0 \\ Y_1 \\ Y_2 \end{pmatrix} = \begin{pmatrix} \left\lfloor \dfrac{I_0 + 2I_1 + I_2}{4} \right\rfloor \\ I_2 - I_1 \\ I_0 - I_1 \end{pmatrix}$$

Forward RCT

(2.2-1)

$$\begin{pmatrix} I_1 \\ I_0 \\ I_2 \end{pmatrix} = \begin{pmatrix} Y_0 - \left\lfloor \dfrac{Y_1 + Y_2}{4} \right\rfloor \\ Y_2 + I_1 \\ Y_1 + I_1 \end{pmatrix}$$

Inverse RCT

(2.2-2)

$$\begin{pmatrix} Y_0 \\ Y_1 \\ Y_2 \end{pmatrix} = \begin{pmatrix} 0.299 & 0.587 & 0.114 \\ -0.16875 & -0.33126 & 0.5 \\ 0.5 & -0.41869 & -0.08131 \end{pmatrix} \begin{pmatrix} I_0 \\ I_1 \\ I_2 \end{pmatrix}$$

Forward ICT

(2.2-3)

$$\begin{pmatrix} I_0 \\ I_1 \\ I_2 \end{pmatrix} = \begin{pmatrix} 1.0 & 0 & 1.402 \\ 1.0 & -0.34413 & -0.71414 \\ 1.0 & 1.772 & 0 \end{pmatrix} \begin{pmatrix} Y_0 \\ Y_1 \\ Y_2 \end{pmatrix}$$

Inverse ICT

(2.2-4)

2.3 Discrete Wavelet Transform and Quantization

The wavelet transform is used for analysis of the tile components into different decomposition levels. These decomposition levels contain a number of subbands, which consist of coefficients that describe the horizontal and vertical spatial frequency characteristics of the original tile component. Due to the statistical properties of these subband signals, the transformed data can usually be coded more efficiently than the original untransformed data.

In JPEG2000 system, two wavelet transform kernels are provided. The DWT can be irreversible or reversible. The default reversible transformation is implemented by means of the Le Gall 5-3 filter, the analysis and the corresponding synthesis filter coefficients are given in Table 2-2. The other one, default irreversible transform, is implemented by means of the Daubechies 9-7 filter, and the corresponding coefficients are given in Table 2-3.

	Analysis Filter Coefficients		Synthesis Filter Coefficients	
i	Low-Pass Filter $h_L(i)$	High-Pass Filter $h_H(i)$	Low-Pass Filter $g_L(i)$	High-Pass Filter $g_H(i)$
0	6/8	1	1	6/8
±1	2/8	-1/2	1/2	-2/8
±2	-1/8			-1/8

Table 2-2 Le Gall 5-3 analysis and synthesis filter coefficients

	Analysis Filter Coefficients		Synthesis Filter Coefficients	
i	Low-Pass Filter $h_L(i)$	High-Pass Filter $h_H(i)$	Low-Pass Filter $g_L(i)$	High-Pass Filter $g_H(i)$
0	0.6029490182363579	1.115087052456994	1.115087052456994	0.6029490182363579
±1	0.2668641184428723	-0.5912717631142470	0.5912717631142470	-0.2668641184428723
±2	-0.07822326652898785	-0.05754352622849957	-0.05754352622849957	-0.07822326652898785
±3	-0.01686411844287495	0.09127176311424948	-0.09127176311424948	0.01686411844287495
±4	0.02674875741080976			0.02674875741080976

Table 2-3 Daubechies 9-7 analysis and synthesis filter coefficients

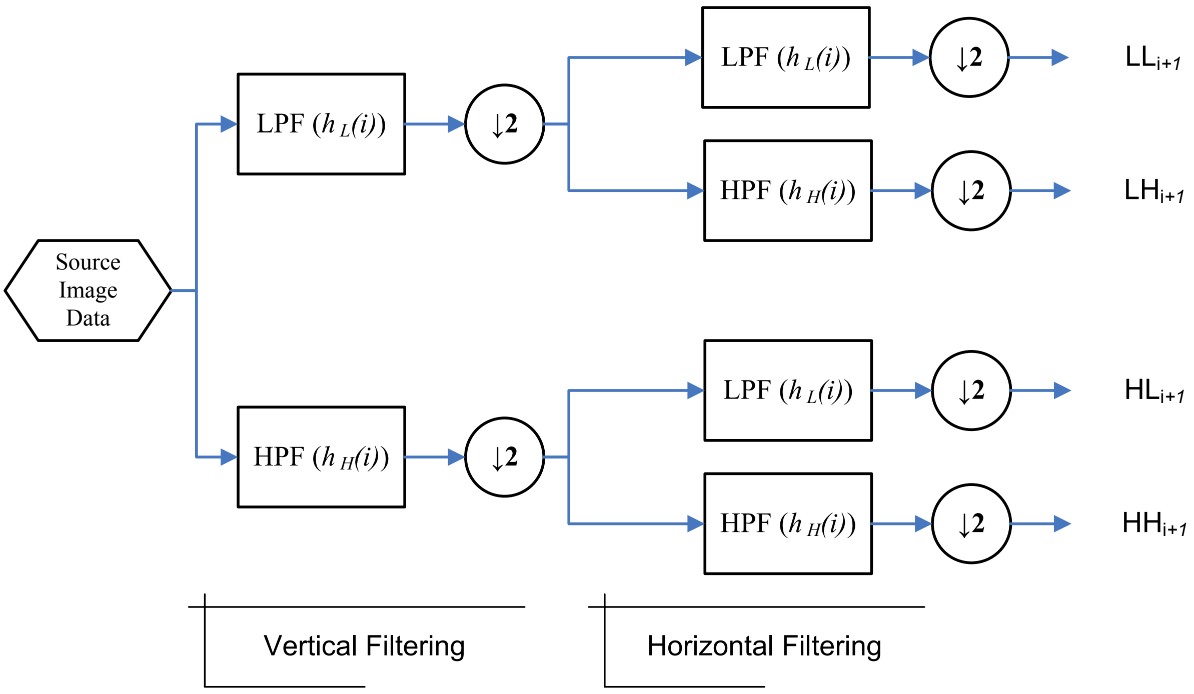


Figure 2-4 2-D forward discrete wavelet transform

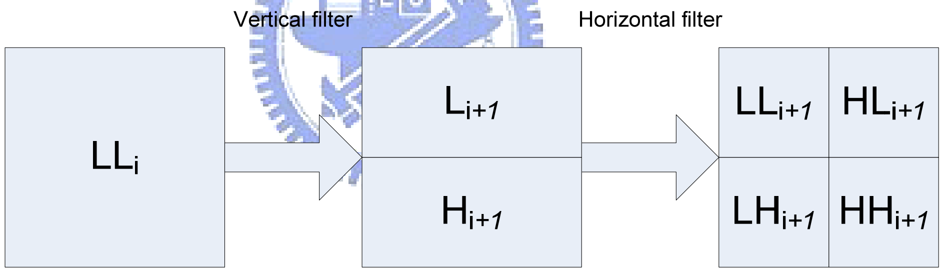


Figure 2-5 2-D DWT decomposition

Usually, the two-dimensional (2-D) discrete wavelet transform is accomplished by cascading two one-dimensional (1-D) discrete wavelet transform. It is decomposed by one-dimensional discrete wavelet transform with 2-channel in horizontal and vertical directions respectively, as shown in Figure 2-4. After one-dimensional vertical discrete wavelet, two subbands are formed. The low-pass samples represent a downsampled low-resolution version of the original set. The high-pass samples represent a downsampled residual version of the original set. And then the subbands pass through the other horizontal filter. The four higher-level subbands are all composed of quarter original image size such as Figure 2-5.

Power of 2 decompositions is allowed in the form of dyadic decomposition (in Part I) as shown in Figure 2-6. For a N by N image through the M-level two-dimensional discrete wavelet transform decomposition, the size of each subband is $N/2^M$ by $N/2^M$. An example of a dyadic decomposition into subbands of the image 'Lena' is illustrated in Figure 2-7.

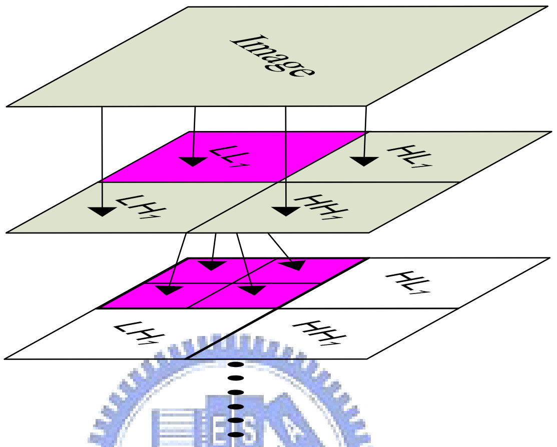


Figure 2-6 Hierarchical of multi-level 2-D DWT

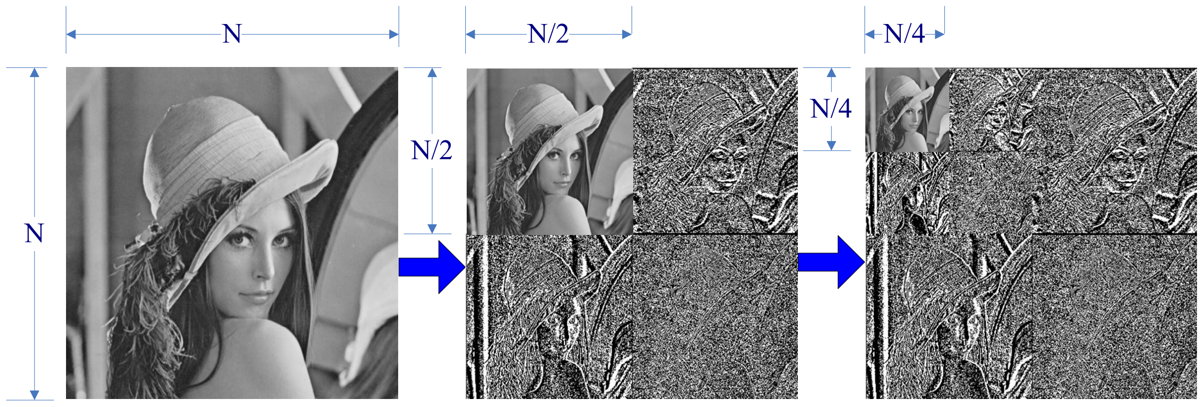


Figure 2-7 An example of Lena image for multi-level 2-D DWT

After transformation, all coefficients are quantized. Sever quantization options are provided in JPEG2000 standard. Only the uniform scalar quantization which is the default quantization method in JPEG2000 standard Part 1 would be introduced here.

In integer mode, the quantizer step sizes are always fixed at one, effectively bypassing quantization and forcing the quantizer indices and transform coefficients to be one and the same. In this case, lossy coding is still possible, but rate control is achieved by other mechanism. In the case of real mode, the quantizer step sizes are chosen in conjunction with rate control. Each of the transform coefficients $a_b(u,v)$ of the subband b is quantized to the value $q_b(u,v)$ according to the formula (2.3-1). Since the step size Δ_b is represented relative to the dynamic range R_b of the subband b, it is defined in (2.3-2). The exponent/mantissa pairs (ε_b, μ_b) are either explicitly signaled in the bit stream syntax for every sub-band.

$$q_b(u,v) = sign(a_b(u,v))\left\lfloor \frac{|a_b(u,v)|}{\Delta_b} \right\rfloor \qquad (2.3\text{-}1)$$

$$\Delta_b = 2^{R_b-\varepsilon_b}(1+\frac{\mu_b}{2^{11}}) \qquad (2.3\text{-}2)$$

2.4 Embedded Block Coding with Optimized Truncation

Embedded block coding with optimized truncation (EBCOT) [9] is adopted for the entropy coding of JPEG2000.The EBCOT consists of two major coding step, tier-1 and tier-2, as shown in Figure 2-8. The tier-1 part is the embedded block coding (EBC) which is composed of the context formation (CF) and the arithmetic encoder (AE). The tier-1 coder divides each subband coefficient into code-blocks and all code-blocks are coded separately into a block-based embedded bit-stream. The coding is performed using the bit-plane coder described later in next section. For each code-block, an embedded code is produced, comprised of numerous coding passes and the output of the tier-1, block-based embedded bit-stream, is a collection of coding passes for the various code-blocks. After that, the tier-2 truncates the embedded bit-stream to minimize the overall distortion. We will introduce the two tiers in following sections.

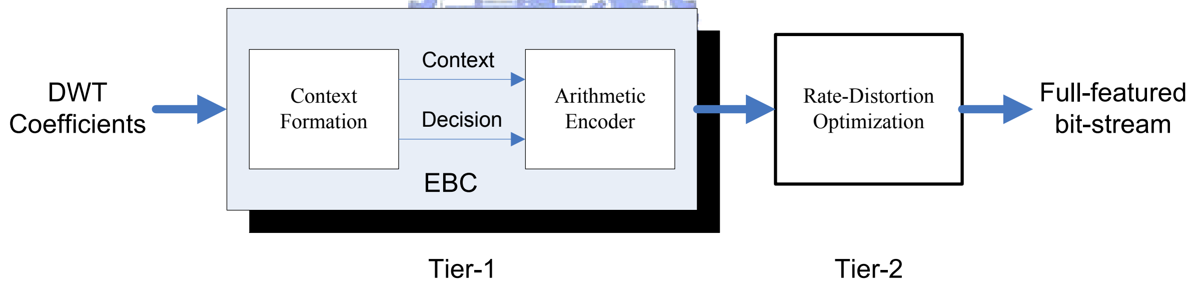


Figure 2-8 Two tiers of EBCOT algorithm

2.4.1 Tier-1 Coding

The tier-1 coding is also a known as the embedded block coding (EBC). It includes the context formation (CF) and the arithmetic encoder (AE) and its basic coding unit is a code-block. The EBC is a bit-level processing algorithm, and the code-block is coded in a bit-plane by bit-plane manner which is from the most significant bit (MSB) bit-plane to the least significant bit (LSB) bit-plane in a code-block. Every bit-plane takes three passes and it is scanned in a stripe-based method, as presented in Figure 2-9.

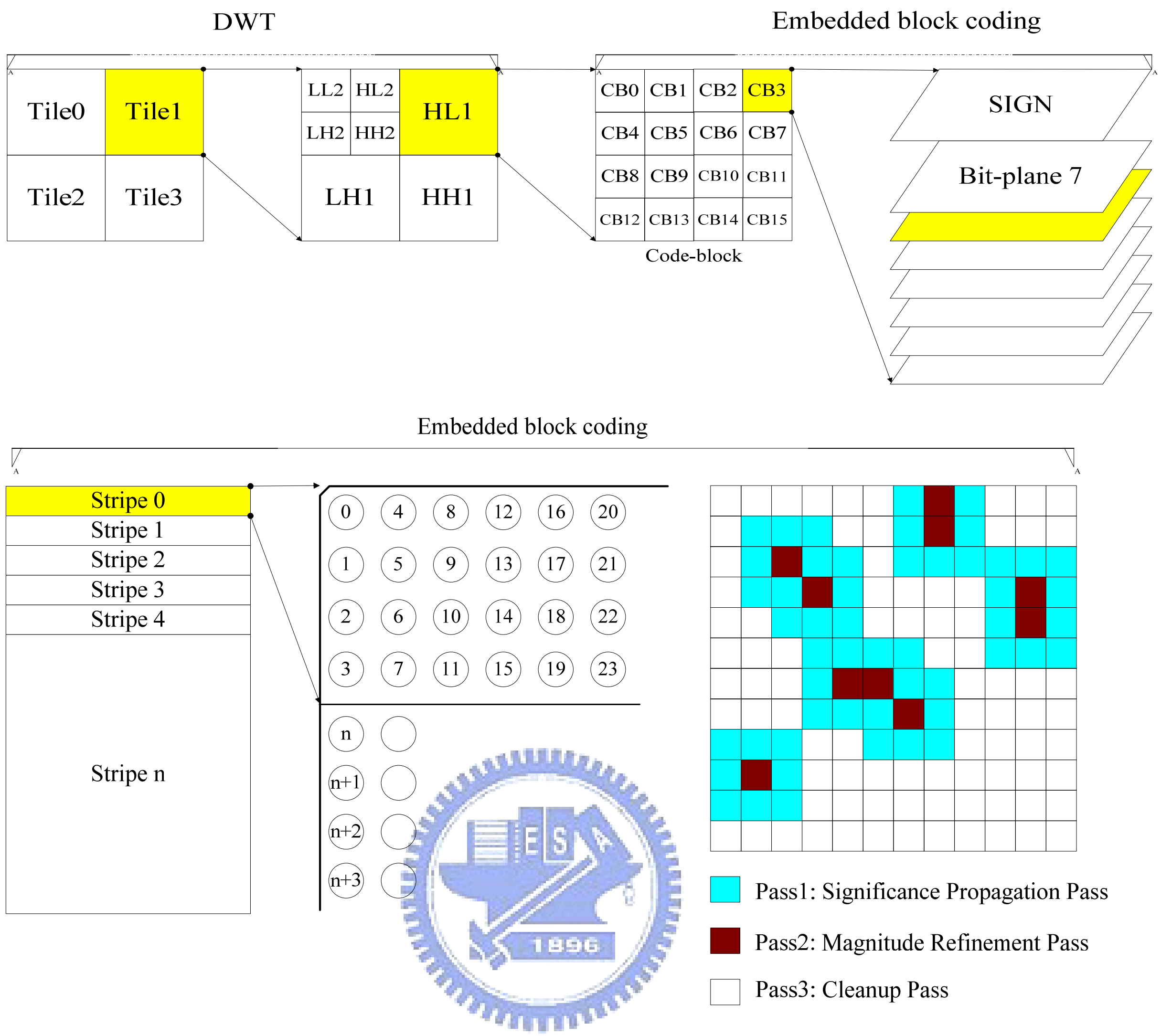


Figure 2-9 Diagram of tile, code-block, bit-plane, stripe and coding pass

2.4.1.1 Context Formation (CF)

The embedded block coding is essentially a context-adaptive arithmetic encoder as shown in Figure 2-8. The context formation (CF) generates context-decision pairs for the arithmetic encoder (AE). The context is adopted to adapt the probability of the decision by the AE. In context modeling, all code-blocks are coded a bit-plane at a time starting from the MSB bit-plane with a non-zero element to the LSB bit-plane. For each bit-plane in a code-block, a special scan pattern is use for each of three coding passes. The three coding passes are coded in order as Pass1 (significance propagation pass), Pass2 (magnitude refinement pass), and then Pass3 (cleanup pass). Each coefficient bit from DWT is coded in only one of the three coding passes, and the coding condition is shown in Table 2-4.

Coding Pass	***Coding Condition***
Pass1 (Significance Propagation Pass)	Insignificant sample with at least one significant neighbor
Pass2 (Magnitude Refinement Pass)	Significant sample
Pass3 (Cleanup Pass)	Insignificant sample with all

Table 2-4 Coding Pass Classification

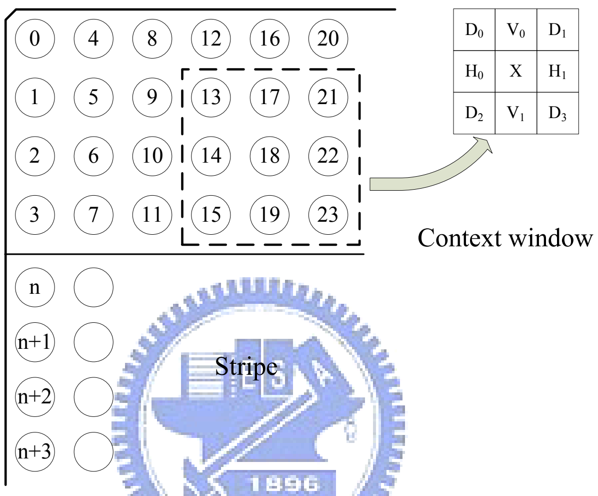


Figure 2-10 Context window and Neighbors states

Since the context-based arithmetic coding is employed, a means to select context selection is necessary. Figure 2-10 shows the context window and the 4-connected or 8-connected neighbors of a sample is selected that is performed by examining state information.

The first coding pass (Pass1) for each bit plane is the significance propagation pass. During the significance propagation pass, a bit is coded if its location is not significant, but at lease one of its 8-connected neighbors is significant. Nine context labels (Table 2-5) are created based on how many and which ones are significant. The significance propagation pass includes only bits of coefficients that were insignificant and have a non-zero context. All other coefficients are skipped. If the value of this bit then the significance state is set to 1 and then the sign coding must be performed. The sign coding is determined using another context table.

Only four neighbors are considered, and each neighbor may have one of three states: significant positive, significant negative, or insignificant. Both vertical and horizontal give the different contribution for the context table. The nine permutations of the vertical and horizontal contributions are reduced into five context labels as shown in Table 2-6. The decision of sign coding can be obtained by performing the logic XOR operation with the XOR bit of the sign context table.

The second coding pass (Pass2) for each bit plane is the magnitude refinement pass. This pass signals subsequent bits after the most significant bit for each sample. If a sample was found to be significant in a previous bit plane (except those that have just become significant in the immediately proceeding significance propagation pass), the next most significant bit of that sample is conveyed using a single binary symbol. The context used in magnitude refinement coding is determined by the summation of the significance state of the horizontal, vertical, and diagonal neighbors as shown in Table 2-7.

All the remaining coefficients in the bit-plane are insignificant and have the context value of zero during the significance propagation pass. These are all included in the cleanup pass (Pass3). The cleanup coding not only uses the neighbor context, like that of the significant coding from Table 2-5, but also a run-length coding. If the four contiguous samples in a column and the context labels of the four samples are all zeros, the run-length coding is performed.

LL and LH sub-bands (vertical high-pass)			HL sub-band (horizontal high-pass)			HH sub-band (diagonally high-pass)		Context Label
ΣH	ΣV	ΣD	ΣH	ΣV	ΣD	Σ(H+V)	ΣD	
2	X^b	X	X	2	X	X	≧3	8
1	≧1	X	≧1	1	X	≧1	2	7
1	0	≧1	0	1	≧1	0	2	6
1	0	0	0	1	0	≧2	1	5
0	2	X	2	0	X	1	1	4
0	1	X	1	0	X	0	1	3
0	0	≧2	0	0	≧2	≧2	0	2
0	0	1	0	0	1	1	0	1
0	0	0	0	0	0	0	0	0

Table 2-5 Contexts for the significance propagation pass and cleanup coding passes

Horizontal contribution	Vertical contribution	Context Label	XOR bit
1	1	13	0
1	0	12	0
1	-1	11	0
0	1	10	0
0	0	9	0
0	-1	10	1
-1	1	11	1
-1	0	12	1
-1	-1	13	1

Table 2-6 Contributions of the vertical (and the horizontal) neighbors to the sign context

$\Sigma H + \Sigma V + \Sigma D$	First refinement for this sample	Context Label
X^b	False	16
$\geqq 1$	True	15
0	True	14

Table 2-7 Contexts for the magnitude refinement coding pass

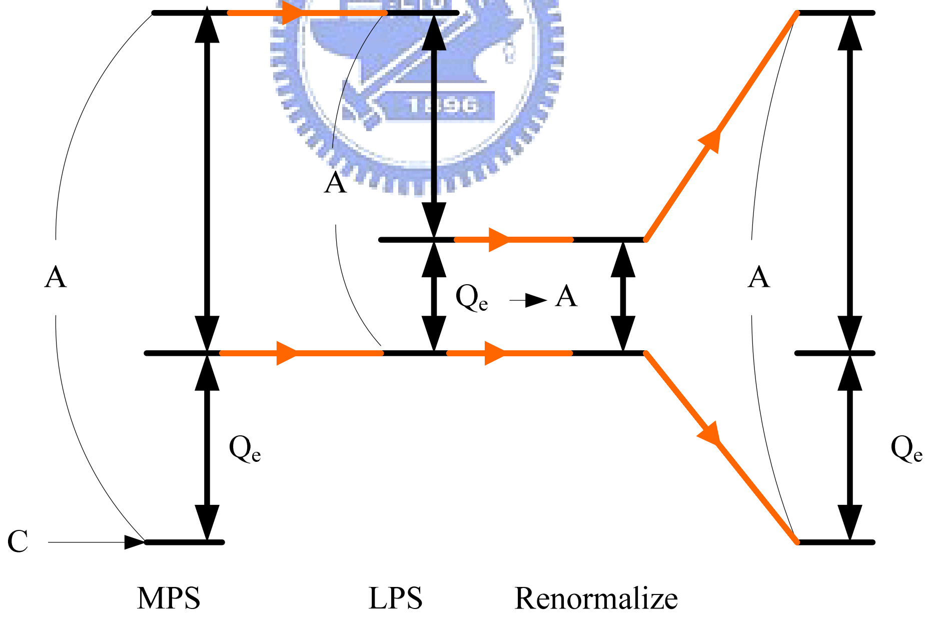


Figure 2-11 Basic operation of the AE

(Most Probable Symbol, Least Probable Symbol, and Renormalization)

2.4.1.2 Arithmetic Encoder (AE)

The decision which is produced by the CF is coded during arithmetic encoder. The AE is an adaptive, binary MQ-coder [10]. The basis of the binary arithmetic coding process is the recursive probability interval subdivision of Elias coding. Since it is a binary AE, there are only two sub-intervals. With each binary decision, the current probability interval is subdivided into two sub-intervals, and the codestream is modified (if necessary) so that points to the base (lower bound) of the probability sub-interval assigned to the symbol as shown in Figure 2-11. Besides, a lazy coding mode is used to reduce the number of symbols that are arithmetically coded. According to this mode, after the fourth bitplane is coded, the first and second pass are included as raw, while only the third coding pass of each bitplane employs arithmetic coding.

2.4.2 Tier-2 Coding

The tier-2 encoding follows the tier-1 encoding, and the input of the tier-2 encoding process is the set of bit-plane coding passes generated during tier-1 encoding. Each coding pass is a candidate of truncation point of a code-block and the coding pass information is packaged into data units called packets in tier-2 coding. For meeting a target bit-rate or transmission time, the packaging process imposes a particular organization of coding pass data in the output codestream. Thus rate control assures that the desired number of bytes is used by the codestream while assuring the highest image quality possible. We will review the RDO algorithm in following section.

In the encoder, rate control can be achieved through two distinct mechanisms, the choice of quantization step size and the selection of the subset of coding passes to include in the codestream. When lossless coding is employed, only the first mechanism may be used. The quantization step sizes must be fixed to one. In lossy coding mode, both of the two mechanisms may be employed. If the quantization step sizes are changed, the tier-1 encoding must be performed again. Since tier-1 coding requires a lot of computation, changing step sizes may not be practical in the encoder. The encoder can elect to discard coding passes in order to control the rate. The contribution of each coding pass makes to rate, and calculates the distortion reduction. Using this information, the encoder can include the coding passes in order of decreasing distortion reduction until the bit budget has been exhausted.

The goal of rate control is to minimize the distortion while keeping the rate smaller than the target rate, R_T. The problem is mapped into Lagrange optimization problem [11] as (2.4-1).

$$\min(D+\lambda R)=\min\left(\sum_i (D_i^{z_i}+\lambda R_i^{z_i})\right) \qquad (2.4\text{-}1)$$

The D means total distortion, and R means total bit rate. The Lagrange multiplier(λ) is used to minimize J = D+R λ ,and thus the derivative of J is set to zero. The candidate corresponding pass m of the bit-plane k in the code-block i (B_i) is represented as Z_i. Then the optimal λ ,(* λ), and the slop of R-D curve can be obtained as (2.4-2).

$$*\lambda=-\frac{\partial D}{\partial R} \qquad (2.4\text{-}2)$$

For each code-block B_i, the slop of R-D curve is corresponding to the number of Z_i as (2.4-3). The S_i^{Zi} means the reduction speed when B_i is truncated at Z_i. The optimal solution proved in [11] is constrained as below (2.4-4). The $*Z_i$ is the optimal truncation point of B_i, and the rate-distortion optimization can be achieved when Z_i is sufficiently closed to $*Z_i$.

$$S_i^{Z_i}=\frac{\Delta D_i^{Z_i}}{\Delta R_i^{Z_i}}=\frac{D_i^{Z_i}-D_i^{Z_{i+1}}}{R_i^{Z_i}-R_i^{Z_{i+1}}} \qquad (2.4\text{-}3)$$

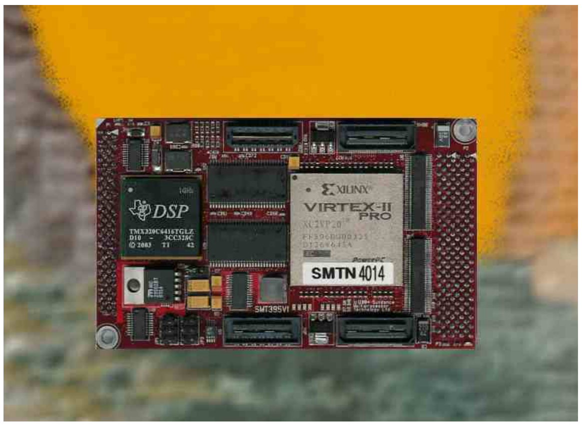
$$\begin{cases} -S_i^{Z_i}\ge *\lambda, & Z_i\ge *Z_i \\ -S_i^{Z_i}< *\lambda, & Z_i< *Z_i \end{cases}$$
$$and \qquad (2.4\text{-}4)$$
$$\sum_i R_i^{*Z_i}\le R_T$$

Chapter 3
DSP Implementation Environment

In this chapter, we will briefly introduce the DSP platform environment and some optimization methods. We use the DSP module (SMT395) made by Sundance. It houses two important chips, TMS320C6416T DSP chip made by Texas Instrument and Xilinx Virtex II Pro FPGA. As our implementation is software base system, we only focus on the DSP chip. In addition, we will introduce the software development tool, the Code Composer Studio (CCS), and bring in some efficient optimization methods by using this environment.

3.1 DSP Platform Introduction

Our DSP platform includes two major modules, SMT395 and SMT310. The DSP module, SMT395, is based on the 1GHz 64-bit TMS320C6416T DSP which is manufactured on the 90nm wafer technology. It is also supported by the T.I. Code Composer Studio and 3L Diamond RTOS to enable full multi-DSP systems with minimum efforts by the programmers. We use the TI's PCI module carrier (SMT310) to communicate between SMT395 and personal computer. Our emulation results could be passed from SMT310 PCI bus and shown on CCS windows. Figure 3-1 shows the pictures of SMT395 and SMT310. The SMT395 module can be installed on the SMT310 carrier and SMT310 can be installed on a personal computer. The block diagram of emulator system is shown in Figure 3-2. We will introduce the main DSP module (SMT395) and software environment in following sections.


SMT395

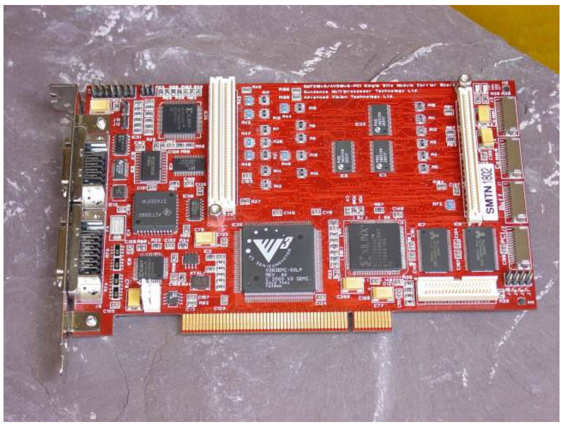

SMT310

Figure 3-1 SMT395 module and SMT310 carrier

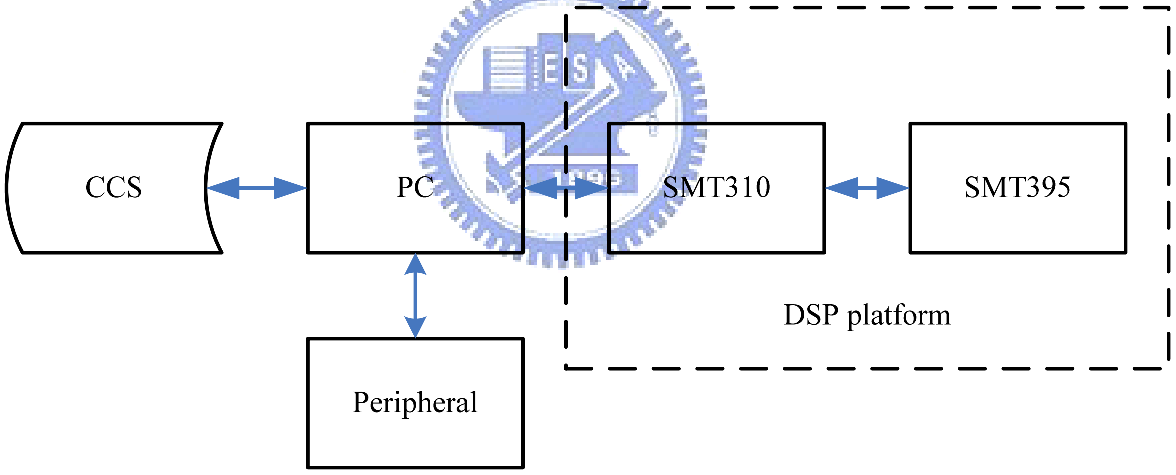


Figure 3-2 Block diagram of emulator system

3.2 Major DSP Module

In our emulator system, the DSP module (SMT395) is the most important part of this system. First, we list some important features of SMT395 module as follows [12].

- 1GHz TMS320C6416T fixed point DSP
- 8000MIPS peak performance
- Xilinx Virtex II Pro FPGA. XC2V920-6 in FF896 package
- 256 Mbytes of SDRAM @ 133MHz using k4s511632M
- Two Sundance High-speed Bus (50MHz, 100MHz or 200MHz) ports 32 bits wide
- Eight 2 Gbit/sec Rocket Serial Links (RSL) for Inter-Module communications
- Six Comports up to 20 Mbytes/sec each for Inter-DSP communication/configuration
- 8 Mbytes Flash ROM for configuration and booting
- JTAG diagnostics port

The TMS320C6416T DSP is the highest-performance fixed-point DSP generation in the TMS320C64X series of the TMS320C6000 DSP family. It is based on the second-generation high-performance, advanced VelociTI very-long-instruction-word (VLIW) architecture (Called VelociTI.2) developed by Texas Instruments [13]. The VelociTI.2 extensions in the eight functional units include new instruction to accelerate the performance in key applications and extend the parallelism of the VelociTI architecture. The functional block and DSP core diagram of TMS320C64x series is shown in Figure 3-3.

In the following sections, three major parts of TMS320C64x DSP are introduced respectively. They are central processing unit, memory, and peripherals.

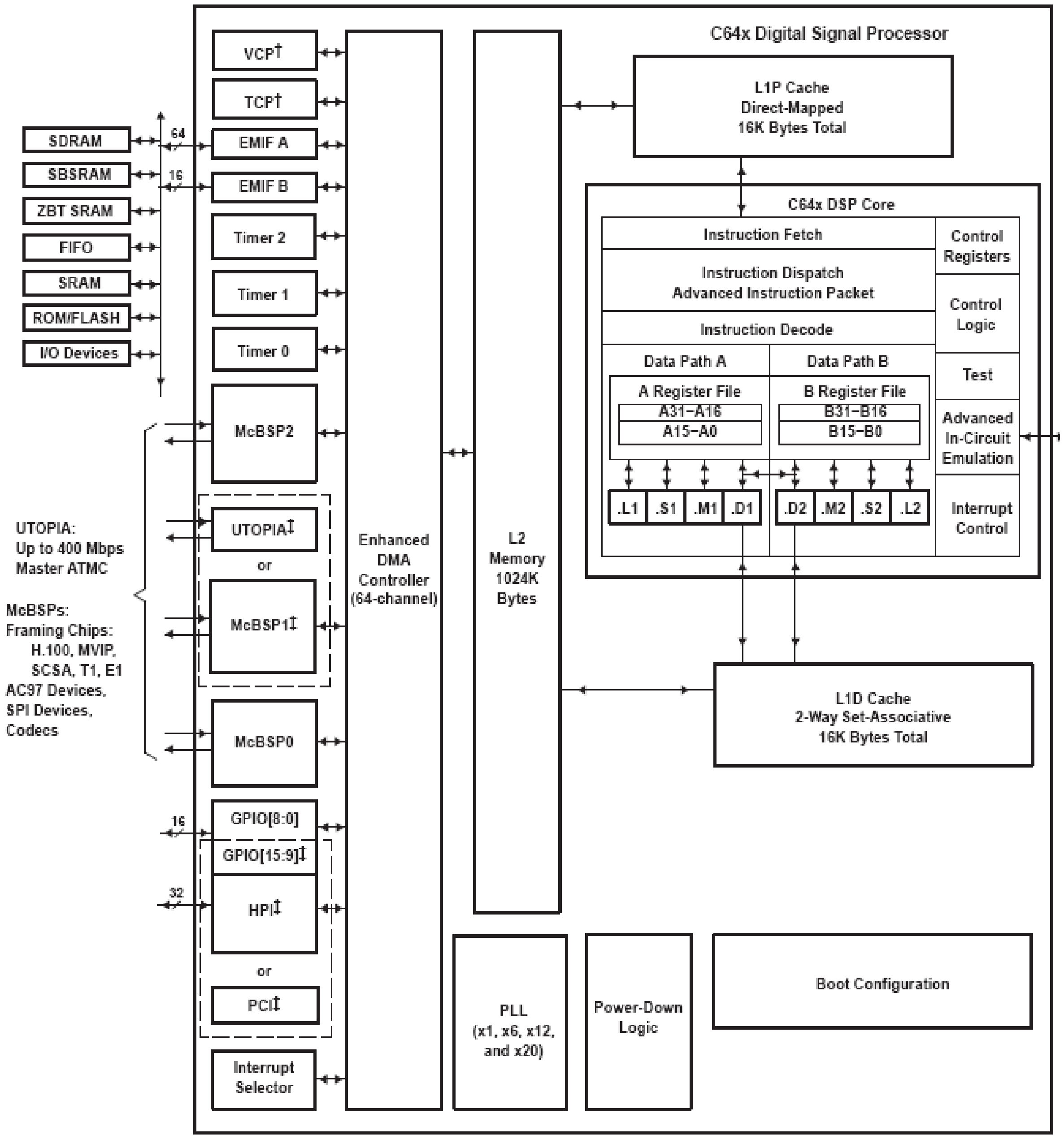


Figure 3-3 Block diagram of the TMS320C64x DSPs [13]

3.2.1 Central Processing Unit

The DSP core of C64x series consists of eight independent functional units, 64 general purpose registers, program fetch unit, instruction dispatch (attached with advanced instruction packing), instruction decode unit, two data path, test unit, emulation unit, interrupt logic, and etc. The instruction dispatch and decode units could decode and arrange the eight instructions to eight functional units respectively. The eight functional units in the C64x architecture could be further divided into two data paths, data path A and B as shown in Figure 3-3. Each path has one unit for multiplication operations *(.M)*, another one for logical and arithmetic operations *(.L)*, another one for branch, bit manipulation, and arithmetic operations *(.S)*, and another one for loading/storing, address calculation and arithmetic operations *(.D)*. The *(.S)* and *(.L)* units are for arithmetic, logical, and branch instructions. All data transfers make use of the *(.D)* units. Two cross-paths (1x and 2x) allow functional units from one data path to access a 32-bit operand from the register file on the other side. There are 32 general purpose registers, but some of them are reserved for specific addressing or used for conditional instructions. Each functional unit has its own 32-bit bus for writing into a general-purpose register file. All functional units which end in 1 (for example, *(.L1)*) write to register file A while all functional units which end in 2 (for example, *(.L2)*) write to register file B.

3.2.2 Memory and Peripherals

The C64x uses a two-level cache-based architecture and has a powerful and diverse set of peripherals. The level 1 program cache (L1P) is a 128 Kbit direct mapped cache and the level 1 data cache (L1D) is a 128 Kbit 2-way set-associative cache. The level 2 memory/cache (L2) consists of an 8 Mbit memory space or combinations of cache (up to 256 Kbytes) and mapped memory. Besides, the TMS320C6416T uses two external memory interfaces (EMIF) to access asynchronous memories (SRAM and EPROM) and synchronous memories (SDRAM, SBSRAM, ZBT SRAM, and FIFO).

The C64x contains some peripherals such as enhanced direct memory access (EDMA) controller, host-port interface (HPI), external memory interface (EMIF), PCI, and etc. The EDMA supports up to 64 EDMA channels which service peripheral devices and external memory. For the C64x device, the association of an event to a channel is fixed, and each of the EDMA channels has one specific event associated with it. These specific events are captured in the EDMA event registers even if the events are disabled by the EDMA event enable registers. The HPI is a parallel port through which a host processor can directly access the CPU’s memory space. The host can direct access to memory-mapped peripherals and has ease of access. The PCI module supports connection of the C6000 device to a PCI host via the integrated PCI master/slave bus interface.

3.3 Coding Development Environment

In this Section, we will give a briefly introduction about the coding development environment in this project. The code composer studio (CCS) and the coding development flow are illustrated. The tutorial [14] introduces the key features of CCS and the programmer's guide [15] gives a reference for programming TMS320C6000 digital signal processor (DSP) devices. A programmer needs to be familiar with coding development flow and CCS for building a new project on the DSP platform efficiently.

3.3.1 Code Composer Studio

Code Composer Studio (CCS) speeds and enhances the development process for programmers who create and test real-time, embedded signal processing applications. The CCS extends the basic code generation tools with a set of debugging and real-time analysis capabilities which is described as Figure 3-4. In addition, the CCS includes the following components which are listed below and all of these work together as shown in FIG..

- TMS320C6000 code generation tools
- Code Composer Studio Integrated Development Environment (IDE)
- DSP/BIOS plug-ins and API
- RTDX plug-in, host interface, and API

The code generation tools provide the foundation for the development environment provided by the CCS such as C compiler, assembler, assembler optimizer, linker, archiver and etc. The code composer studio integrated development environment is designed for editing, building, and debugging DSP target programs. During the analysis phase of the software development cycle, traditional debugging features are ineffective for diagnosing subtle problems that arise from time-dependent interactions. Therefore the DSP/BIOS plug-ins provides real-time analysis such as program tracing, performance monitoring, and file streaming. In addition, the real-time data exchange (RTDX) provides real-time, continuous visibility into the way DSP applications operate in the real world. It allows system developers to transfer data for bi-directional real-time communications between a host computer and the DSP devices without stopping their target application.

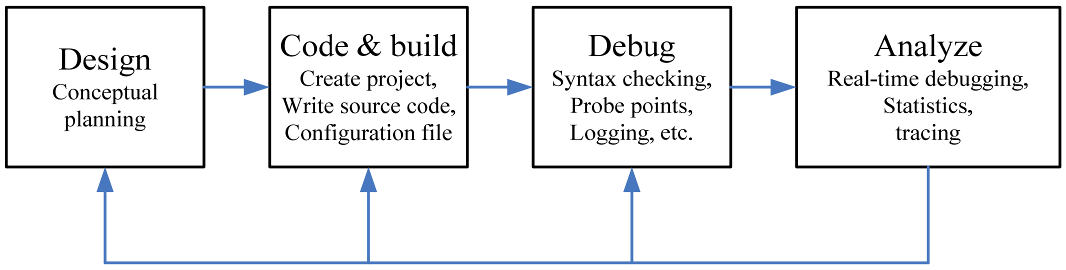

Design
Conceptual planning
Code & build
Create project, Write source code, Configuration file
Debug
Syntax checking, Probe points, Logging, etc.
Analyze
Real-time debugging, Statistics, tracing

Figure 3-4 Development cycle

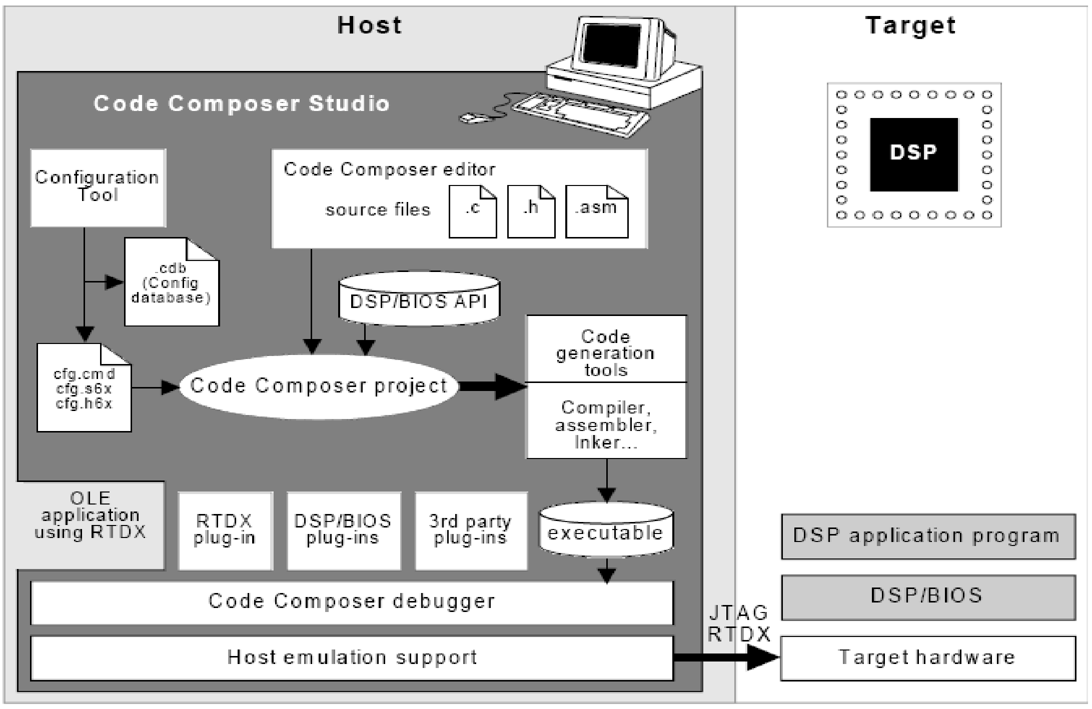


Figure 3-5 Code composer studio development

3.3.2 Code Development Flow

Traditional development flows in the DSP industry have involved validating a C model for correctness on a host PC or UNIX workstation and then painstakingly porting that C code to hand coded DSP assembly language. But this is both time consuming and error prone, the recommended code development flow involves utilizing the C6000 code generation tools to aid in optimization rather than forcing the programmer to code by hand in assembly. These advantages allow the compiler to do all the laborious work of instruction selection, parallelizing, pipelining, and register allocation. The phases of recommended code development flow are described as Figure 3-6. In phase 3, writing linear assembly code is not adopted unless the software pipelining efficiency is hardly achieved or the unbalanced resource allocation is hardly solved by the compiler with C code.

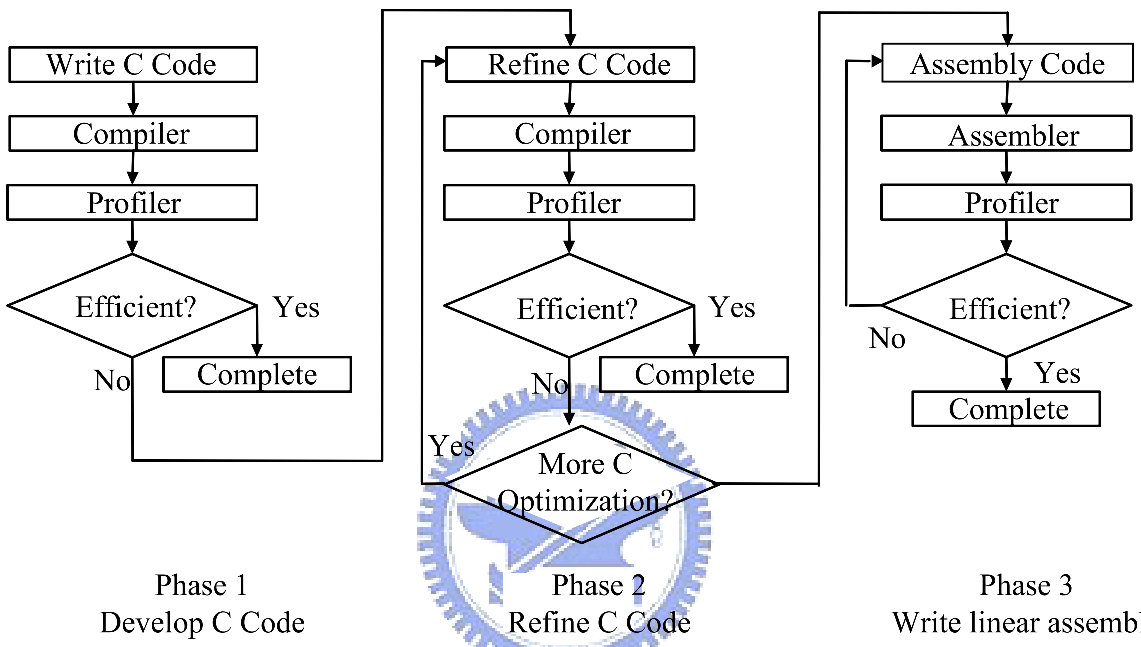


Figure 3-6 Code develop flow

3.3.3 Simulation Tools

In the code develop flow mentioned in Figure 3-6 we know that profiling is an essential step for analyzing coding efficiency. We use the C64xx CPU cycle accurate simulator to simulate the core of the C64xx processor with cycle accuracy. This is faster than the device cycle accurate simulators but does not simulate peripherals and cache system (use a flat memory system). In addition, we use another simulator called C6416 device cycle accurate simulator to simulate the C64xx XDS510 emulator. It simulates the C6416 processor and supports L1D, L1P, L2 cache, EDMA, QDMA, Interrupt Selector, McBSP(3), Timer(3), TCP, VCP and EMIF. It also supports interfacing with Async, SDRAM and Generic sync RAM Memory models. Finally, we use C64xx XDS510 emulator with the hardware board to verify our project. The TMS3206416T which is connected via the XDS510 that emulator sets the I/O ports on our DSP platform. In the following sections, the profiling results of all the simulators are presented. But we like to remind that the C64xx XDS510 emulator cannot profile the CPU cycles.

3.4 Optimization on TI DSP Platform

As Figure 3-6 indicates, the optimization tools increase execution performance. In the following sections, several optimization technologies using VelociTI architecture and software technologies are introduced and adopted in this project.

3.4.1 Architecture of TI TMSC6000 Family

The TMS320C6000 series use the VelociTI architecture which is a high-performance, advanced very-long-instruction-word (VLIW) architecture. The architecture contains multiple execution units running in parallel, which allow them to perform multiple instructions in a single clock cycle. This makes an excellent choice for multi-channel, multi-function, and performance-driven applications. In addition, the C6000 pipeline can dispatch eight parallel instructions every cycle and parallel instructions proceed simultaneously through the same pipeline phases. It eliminates traditional architectural bottlenecks in program fetch, data access, and multiple operations. More detail features about this architecture are introduced in [16].

The TMS320C621x, TMS320C671x, and TMS320C64x DSPs of the TMS320C6000 DSP family have the two-level memory architecture for program and data. The first-level program cache is designated L1P, and the first-level data cache is designated L1D. Both the program and data memory share the second-level memory, designated L2. The L2 is configurable allowing for various amounts of cache and SRAM. Figure 3-3 shows the block diagram of the C64x DSP. The L1P and L1D provide a fast on-chip memory. Accesses by the CPU to these first level caches can complete without CPU pipeline stalls. If the data requested by the CPU is not contained in cache, it is fetched from the next lower memory level. However, over the past years the performance of processors has improved at a much faster pace than that of memory. As a result, there is a performance gap between CPU and memory speed. High-speed memory is available but consumes much more size and is more expensive compared with slow memory.

Hierarchical memory architecture is commonly adopted in the embedded system as Figure 3-7. A fast but small memory is placed close to the CPU that can be accessed without stalls. The next lower memory levels are increasingly larger but also slower the further away from the CPU. Addresses are mapped from a larger memory to a smaller but faster memory higher in the hierarchy. Typically, the higher-level memories are cache memories that are automatically managed by a cache controller. L2 memory is configurable and can be split into L2 SRAM (addressable on-chip memory) and L2 cache for caching external memory locations. The L2 cache is a 4-way set associative cache whose capacity varies between 32 Kbytes and 256 Kbytes depending on its mode. It services cache misses from both L1P and L1D as well as DMA accesses using the EDMA controller. On a C6416T DSP for instance, the sizes of L1D and L1P are 16 Kbytes respectively. The size of L2 is 1 Mbytes and external memory can be several Mbytes large. Although the L2 memory can operate as SRAM, as cache, or as both, the L2 SRAM and L2 cache act with little difference. For example, a single L1D read miss takes 6 cycles when serviced from L2 SRAM, and 8 cycles when serviced from L2 cache. The detailed specifications are described in [17].

In practical implementation, image program usually takes lots of memory space for instant processing. Although there is a L2 memory configured as a SRAM after a reset, it is not enough for all the program instructions and the data. The L1 cache controller fetches most data from external memory with lots of CPU stall. In order to exploit all of the L2 SRAM, programmers must specify the relative data in the linker command file and modify the data structure. This expands time and affects program structure. Because the L1 cache is not large enough, L2 cache is a convenient way to decrease CPU stalls.

There are two ways to configure L2 cache on DSP platform. If the DSP/BIOS is used, L2 cache is enabled automatically. Otherwise, L2 cache can be enabled in the program code by issuing the appropriate chip support library (CSL) commands. Additionally, in the linker command file the memory to be used as L2 SRAM has to be specified. Since L2 cache cannot be used for code or data placement by the linker, all sections must be linked into L2 SRAM or external memory. Further, external memory addresses are optional for cacheable or non-cacheable in the setting of program codes. The real effect on DSP platform is going to be presented in following chapters.

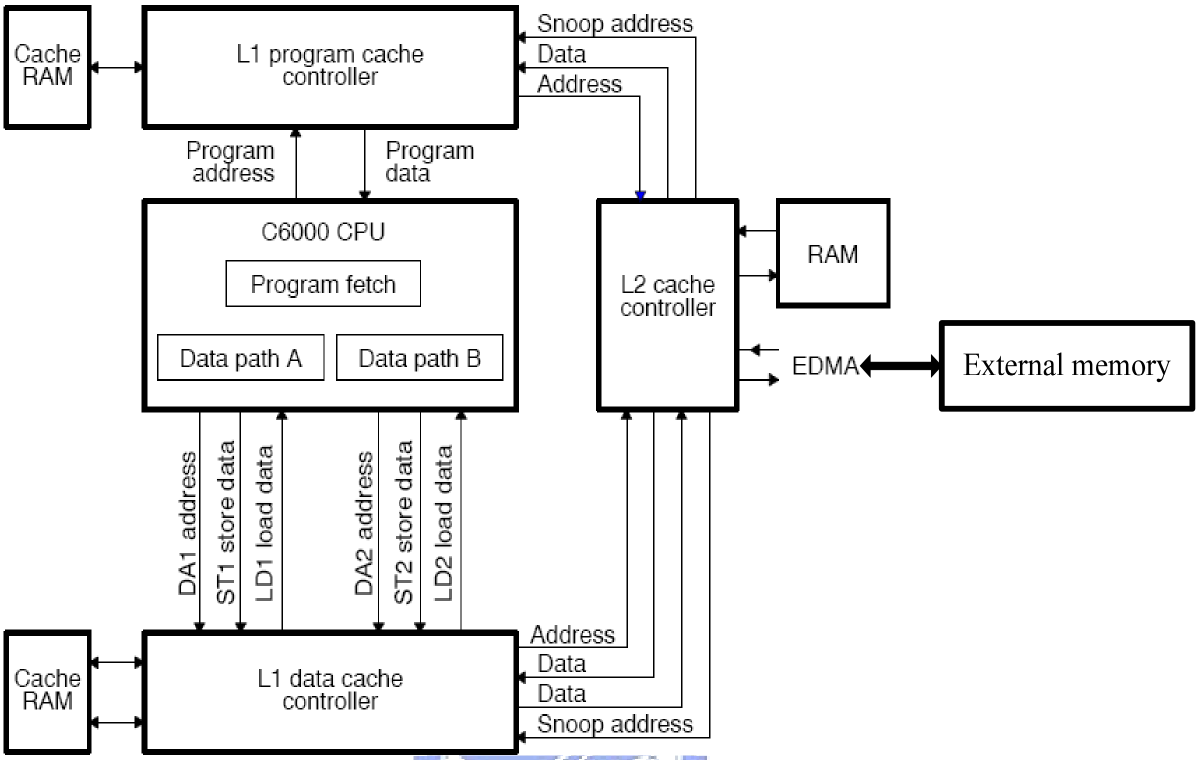


Figure 3-7 TMS320C64x hierarchical memory

3.4.2 Compiler-Level Optimization

The compiler, which includes the parser and optimizer, accepts C/C++ source code and produces C6x assembly language source code. The Figure 3-8 gives a description of the C/C++ compiler. The optimizer can reduce code size and improve executing time by using compiler options. There are four optimization levels which are register (-o0), local (-o1), function (-o2), and file (-o3).

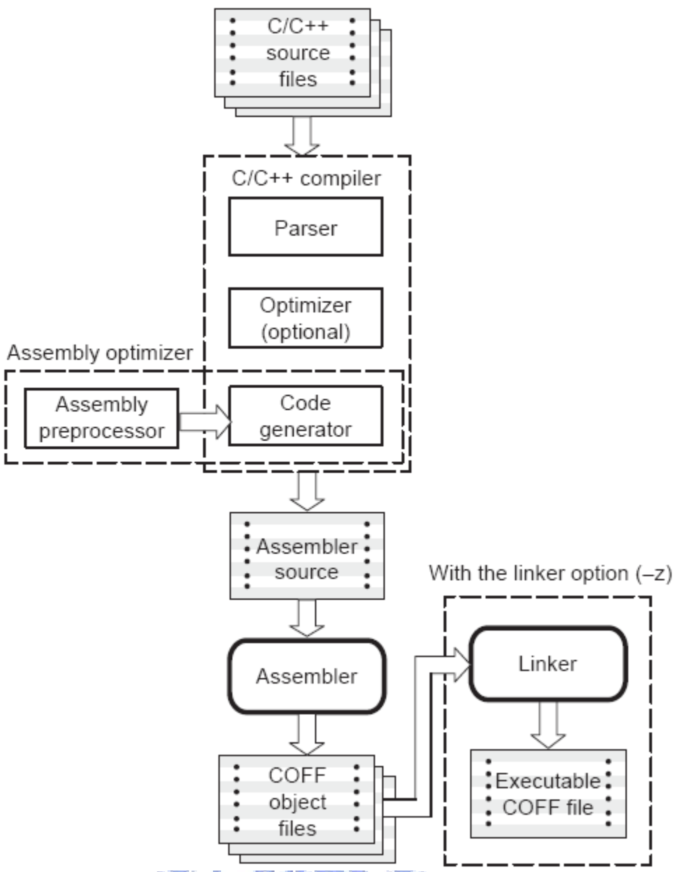


Figure 3-8 C/C++ compiler

The register level (-o0) performs optimizations with control-flow-graph simplification, allocating variables to registers, loop rotation, eliminating unused code, simplifying expressions and statements, and expanding calls to functions declared inline. Next, the local level (-o1) performs all –o0 optimizations, plus local copy/constant propagation, removing unused assignments, and eliminating local common expressions. The function level (-o2) performs all –o1 optimizations, plus software pipelining, loop optimizations, eliminating global common sub-expressions and unused assignments, converting array references in loops to incremented pointer form, and loop unrolling. Finally, the highest level, file level (-o3), performs all –o2 optimizations, plus removing all functions never called, simplifying functions with return values never used, inline calls, reordering function declarations, propagates arguments into function bodies, and identifying file-level variable characteristics. In general, using the –o2 or –o3 level is necessary for performance and code size. The option is also used with the assembly optimizer. Some key optimizations such as software pipelining and loop unrolling are specified with these options.

3.4.3 Program-Level Optimization

Except the optimizations as mentioned in previous sections, there are several methods to speed up the program. First, the linker command file allocates the data sections in different memory. The data which are accessed frequently should be allocated in the higher and fast memory level such as SRAM or cache. Programmer need to analyze the frequency of data accessing for better performance. Although the L2 cache provides an easy way to access external memory, exploiting the SRAM sometimes gets better performance than using the L2 cache. Besides, the missing cycles are

Second, the C6000 C/C++ compiler supports such pragmas like CODE_SECTION, DATA_SECTION, MUST_ITERATE, UNROLL and etc. We know that branch prediction takes lots of cycles when it failed. Through the pragma, such as MUST_ITERATE, the information is provided to aid the compiler in choosing the best loops and loop transformations which means software pipelining and nested loop transformations. There are three methods to unroll the loop. First, you can use the compiler to unroll the loop automatically. Second, you can suggest that the compiler unroll the loop using these DSP pragmas. The last one is that you can unroll the code yourself. Sometimes it also helps the compiler reduce code size and sometimes unrolling by the compiler generates some redundant loops. The detailed specifications are described in [18]. Some of these pragmas are adopted in this project, and the test results are shown in following sections.

Third, the C64x DSPs are fixed-point processors, so they do not directly support floating-point data types. C64x DSPs can simulate floating-point operations, but it takes lots of extra clock. Decreasing floating-point operations is another way to speed up the system. Table 3-1 shows the different data types supported in CCS and take note of the data type "Long" is 40 bits width. Besides, use the short date type for fixed-point multiplication inputs whenever possible because this data type provides the most efficient use of the 16-bit multiplier in the C6000. It is about one cycle for "short × short" versus five cycles for "int × int". But use int or unsigned int data types for loop counters, rather than short or unsigned short data type, to avoid unnecessary sign-extension instructions.

Data Type	char	short	int	float	long	long long	double
Size (bits)	8	16	32	32	40	64	64

Table 3-1 Different data types

The loop unrolling is an efficient method to improve compiler performance. The compiler tries to reschedule the assembly with a full pipeline. Mostly, more instructions without dependence make better parallelism and decrease stalls. Compiler level optimizations and pragmas facilitate loop unrolling. The TMS320C6416T uses the Very-Long-Instruction-Word (VLIW) structure called VelociTI.2. It works efficiently with the loop unrolling to make optimal scheduling. Besides, unrolling loop by programmer is efficacious too. Sometimes compiler level optimizations are restricted to some compiler rules so that loop unrolling by hand is a manual work. In order to make VLIW structure efficiently, we use loop unrolling to fill up the function unit slots. The code size expanded with number of unrolling loops is the major shortcoming. However, the C6000 software pipelining mentioned before is a technique to reorganize loops. It interleaves instructions from different iterations without unrolling the loop. Both of two techniques can be applied simultaneously on the platform for H/W and S/W optimization, and the overhead of a loop and the time issues have eased.

Finally, there are some special functions, called intrinsics, provided by the C6000 compiler. These functions map directly to inlined C64x instructions to optimize the C/C++ code quickly. All instructions that are not easily expressed in C/C++ code are supported as intrinsics. The trick is that intrinsics use a single load or store instruction to access multiple data (SIMD). For example, it can combine four 8-bit data (char) or two 16-bit data (short) to a 32-bit data type, and then it executes one operation instead of four (char) or two (short) operations. If the SIMD method is employed, the code efficiency is improved substantially. Figure 3-9 shows an example of using SIMD method. Other intrinsics enhance the efficiency in the similar way and are described in [19].

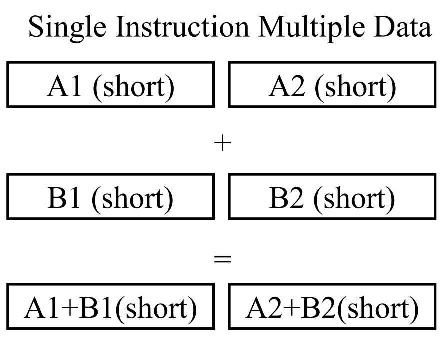


Figure 3-9 SIMD example for using word access for adding short data

Chapter 4

Analysis of Embedded Block Coding and Speed-Improving Methods

In this chapter, we introduce the JPEG2000 software environment and its configuration. JPEG2000 configurations could have an impact on the performance. Some features improve the coding performance but spend lots of memory or complexity. Then we analyze the JPEG2000 encoder and identify the most complex elements in JPEG2000 algorithm. The goal of this chapter is to find algorithms to reduce the JPEG2000 implementation complexity on the DSP platform. Several speed-up methods are presented and compared each other.

4.1 Parameters and Software Environment

4.1.1 Jasper and OpenJPEG Reference Software

In the JPEG2000 standard part 5 [6], it provides two standard reference softwares : JasPer and JJ2000. The JJ2000 is a Java implementation of ISO/IEC 15444-1 (i.e. JPEG2000 image coding standard part1) and the JasPer software is written in the C programming language for the codec specified in ISO/IEC 15444-1. The JasPer is an open-source initiative to provide a free software-based reference implementation of the JPEG2000 codec. All the related documents and software of JasPer could be downloaded from [20]. Now, the latest version 1.701 of the JasPer software is available. We have tested the JasPer reference software and the results are shown as Table 4-1. The main configurations are using 64 by 64 code-block size, 5 decomposition levels, and 1 tile. We use the 512 by 512 gray images, Goldhill, Barb, Lena, and Baboon, which are shown in Figure 4-1.

	5-3 filter				9-7 filter			
BPP	Goldhill	Barb	Lena	Baboon	Goldhill	Barb	Lena	Baboon
0.04	25.1	21.9	25.8	20.0	25.3	22.3	26.2	20.1
0.05	25.7	22.4	26.7	20.2	25.9	22.8	27.0	20.3
0.0625	26.2	22.7	27.4	20.4	26.4	23.1	27.9	20.6
0.125	28.1	24.6	30.2	21.3	28.4	25.2	30.9	21.6
0.25	30.1	27.3	33.2	22.8	30.5	28.3	34.1	23.2
0.5	32.7	30.9	36.3	25.1	33.2	32.1	37.3	25.5
1	35.9	35.8	39.3	28.6	36.5	37.2	40.4	29.1
2	40.7	41.3	43.4	34.1	41.9	43.1	44.6	34.8
3	44.8	45.2	47.5	39.0	46.6	46.8	47.1	40.0
4	49.2	49.5	53.7	43.9	46.9	47.0	47.1	45.4
5	66.5	66.5	66.7	48.2	46.9	47.0	47.1	46.6
6	66.5	66.5	66.7	58.3	46.9	47.0	47.1	46.6
7	66.5	66.5	66.7	66.9	46.9	47.0	47.1	46.6
8	66.5	66.5	66.7	66.9	46.9	47.0	47.1	46.6

Table 4-1 PSNR (dB) of different images using JasPer Ver.1.701 encoder

	5-3 filter				9-7 filter			
BPP	Goldhill	Barb	Lena	Baboon	Goldhill	Barb	Lena	Baboon
0.04	25.3	22.1	26.1	20.1	25.4	22.3	26.5	20.2
0.05	25.8	22.5	26.9	20.3	26.0	22.8	27.2	20.4
0.0625	26.3	22.9	27.5	20.5	26.5	23.4	28.0	20.7
0.125	28.1	24.6	30.2	21.3	28.5	25.4	31.0	21.7
0.25	30.1	27.3	33.1	22.8	30.5	28.4	34.2	23.2
0.5	32.7	30.9	36.3	25.1	33.2	32.3	37.3	25.6
1	35.9	35.8	39.4	28.6	36.6	37.2	40.4	29.1
2	40.9	41.4	43.7	34.2	41.9	43.2	44.9	34.8
3	49.5	49.8	54.0	44.0	49.5	49.3	49.0	45.6
4	49.5	49.8	54.0	44.0	49.5	49.3	49.0	45.6
5	Infinite	Infinite	92.3	Infinite	49.5	49.3	49.0	50.5
6	Infinite	Infinite	92.3	Infinite	49.5	49.3	49.0	50.5
7	Infinite	Infinite	92.3	Infinite	49.5	49.3	49.0	50.5
8	Infinite	Infinite	92.3	Infinite	49.5	49.3	49.0	50.5

Table 4-2 PSNR (dB) of different images using OpenJPEG Ver.1.0 encoder

Figure 4-1 Gray level test images

Because the C language is now mostly convenient implementation language on many platforms, we use the C version of JPEG2000. We have surveyed the other implementations in C language. Our preferred package is OpenJPEG which is an open-source JPEG2000 codec written in C language. It is developed by the Communications and Remote Sensing Lab (TELE), in the University Catholique de Louvain (UCL). The reference software is downloaded from [21] and tested in the same conditions as JasPer software. The test results are presented in Table 4-2.

Compare JasPer with OpenJPEG, and we see that the performance of the OpenJPEG is slightly better than JasPer's. When the decomposition tool uses 5-3 filter, the OpenJPEG software achieves the lossless image encoding. However, the JasPer can not reach this target in the same conditions. On the TI CCS, OpenJPEG works well, but JasPer fails in the release mode with compiler optimization file level (-o3). As a result, we choose OpenJPEG to implement and accelerate on our DSP platform.

4.1.2 Parameter Configuration

In this section, we analyze the coding effect of various coding parameters, including the filter type, decomposition level and tile size. The coding performance on the Rate-Distortion (R-D) curves is measured by the Peak Signal-to-Noise Ratio (PSNR) in dB. The default parameter settings are 1 tile, 5 decomposition levels and 64 by 64 code-block size. First, Figure 4-2 shows the performance of two recommended filter, the 5-3 filter and the 9-7 filter. It shows that the 9-7 filter outperforms the 5-3 filter. This is because the 9-7 filter provides a better capability on energy compaction. Because the 9-7 filter is a floating point transform, the PSNR increase of the 9-7 filter terminates at about 50dB as shown in Table 4-2. In JPEG2000, multilevel discrete wavelet transform decomposition is used to provide better coding efficiency as well as the resolution scalability. The number of decomposition levels affects the coding efficiency. Figure 4-3 shows the results using different decomposition levels and the test image is "Goldhill". Typically, two-level decomposition is sufficient for most natural images but we use three levels as the default setting in the later chapters. We use a larger image, bike, whose resolution is 2048 by 2056 as shown in Figure 4-4, to test different tile sizes. The results show that a larger tile size makes better coding efficiency. In our default setting, we choose one tile for all the test images in the following tests. Also, we use the 64 by 64 code-block. Unlike the previous coding parameters, the impact of the code-block size is usually not noticeable. It varies with different images and other parameters. In general, the coding performance of the 64 by 64 code-block is better than that of the 32 by 32 code-block.

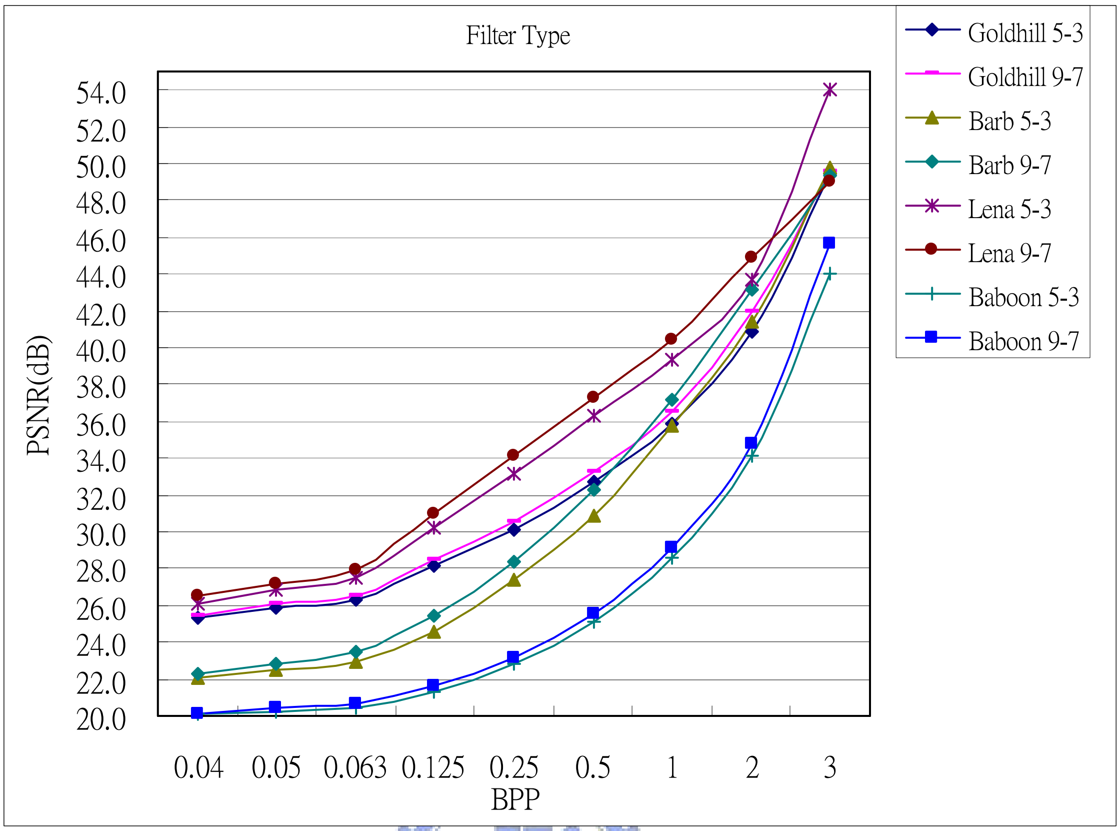


Figure 4-2 Comparison of the 5-3 filter and the 9-7 filter

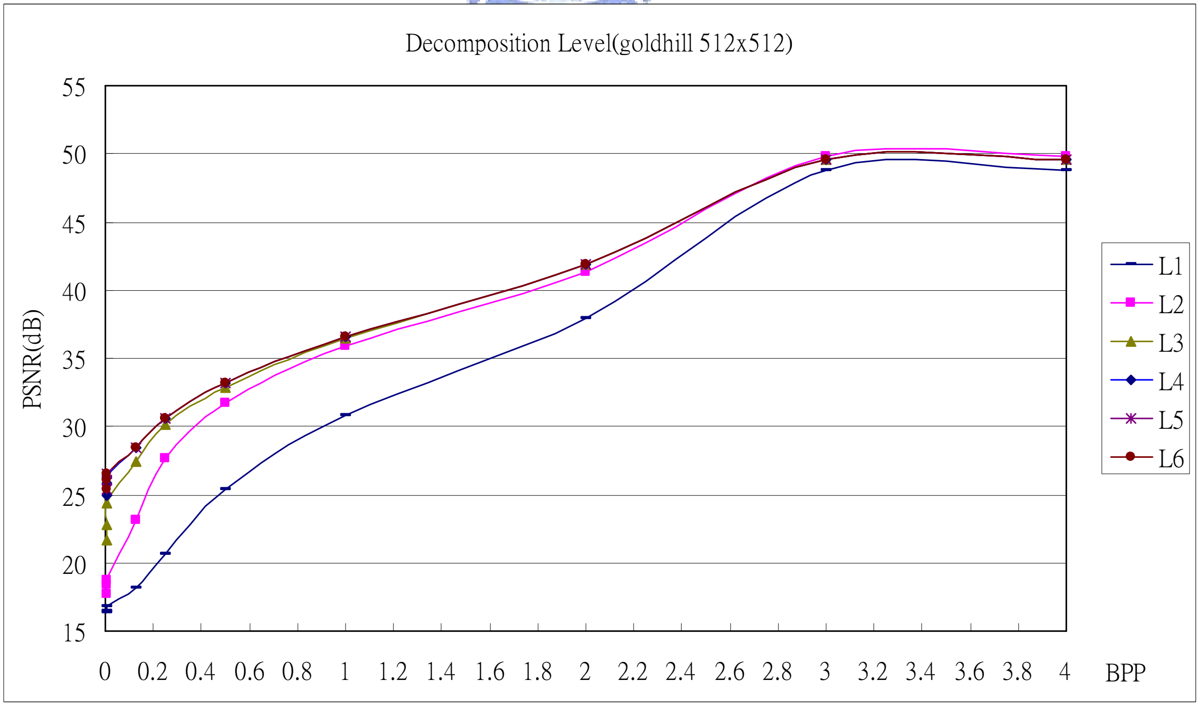


Figure 4-3 Comparison of different decomposition levels (Goldhill)

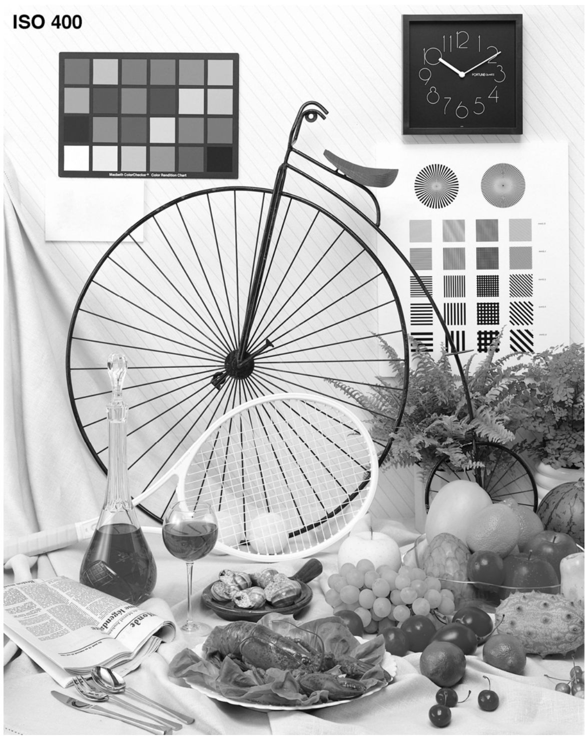


Figure 4-4 Bike 2048x2560

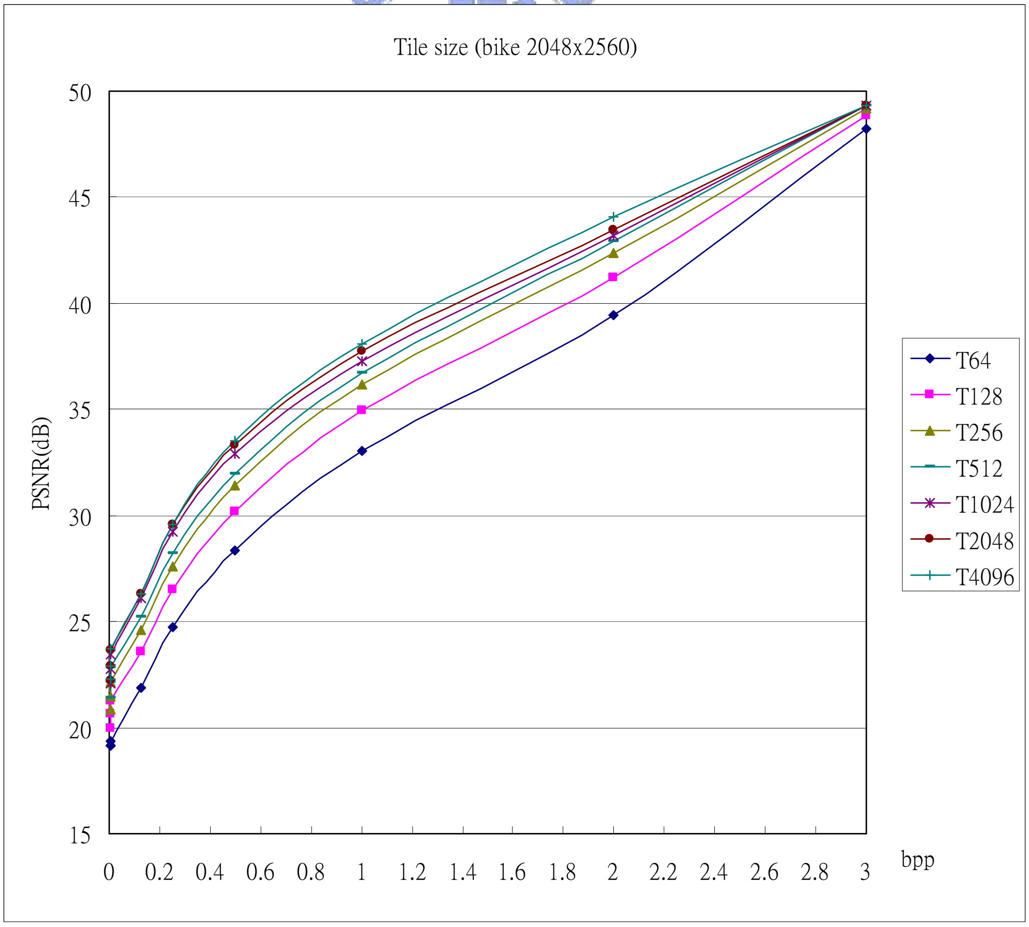


Figure 4-5 Impact of tile size on coding performance

4.2 JPEG2000 Encoder Complexity Analysis

We profile the JPEG2000 encoder to find which part takes the most computation time. As mentioned in section 3.3.3 we have two methods in taking the profiles. One is using C64xx simulator and the other is using C6416 simulator. We will concentrate on the most critical area and try to accelerate these modules. The profiling results using the two methods are shown in Figure 4-6 and Figure 4-7. The test image is 512x512 "Goldhill". The settings are lossless, 1 tile, 64 by 64 code-block, and without any optimization. The profiling results show that Tier1 is the most critical module in the encoder. The profiling result of C64xx means the accurate cycles of the C64xx core processor with flat memory system and the other means the actual cycles of the C64xx XDS510 emulator. The cycles are shown in Table 4-3. The total cycles of the C6416 simulator are approximately nine times of that of the C64xx simulator. It means that the JPEG2000 encoder takes about 8 seconds for encoding the "Goldhill" on the 1GHz DSP. Indeed, we run the encoder on the hardware platform, and it takes about 9 seconds, too. We first like to find the bottleneck on the C6416 emulator which includes the memory access time.

Simulator	C64xx	C6416	Ratio
DWT cycles	73,327,701	552,674,115	13 %
Tier1 cycles	846,100,912	7,509,481,733	11%
Tier2 cycles	1,550,147	15,933,932	9%
Others (cycle)	9,475,720	103,399,183	9%
Total cycles	930,454,480	8,181,488,963	11%

Table 4-3 Cycles on different simulators

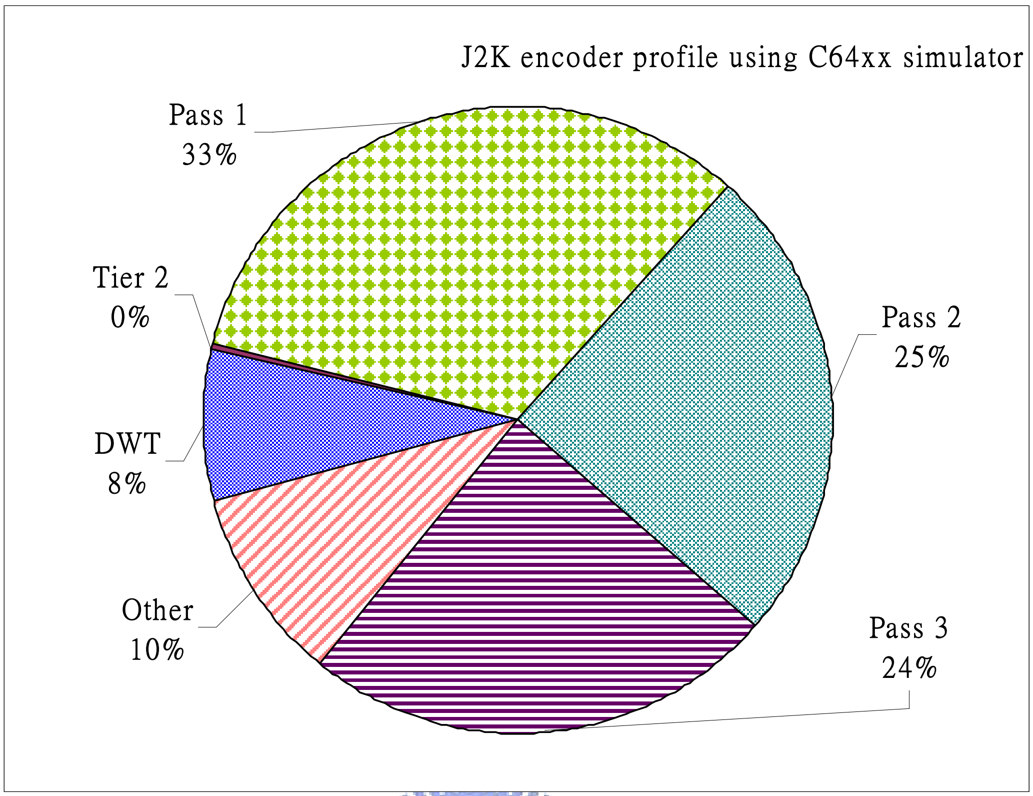


Figure 4-6 Complexity profiling of the JPEG2000 encoder on the C64xx simulator

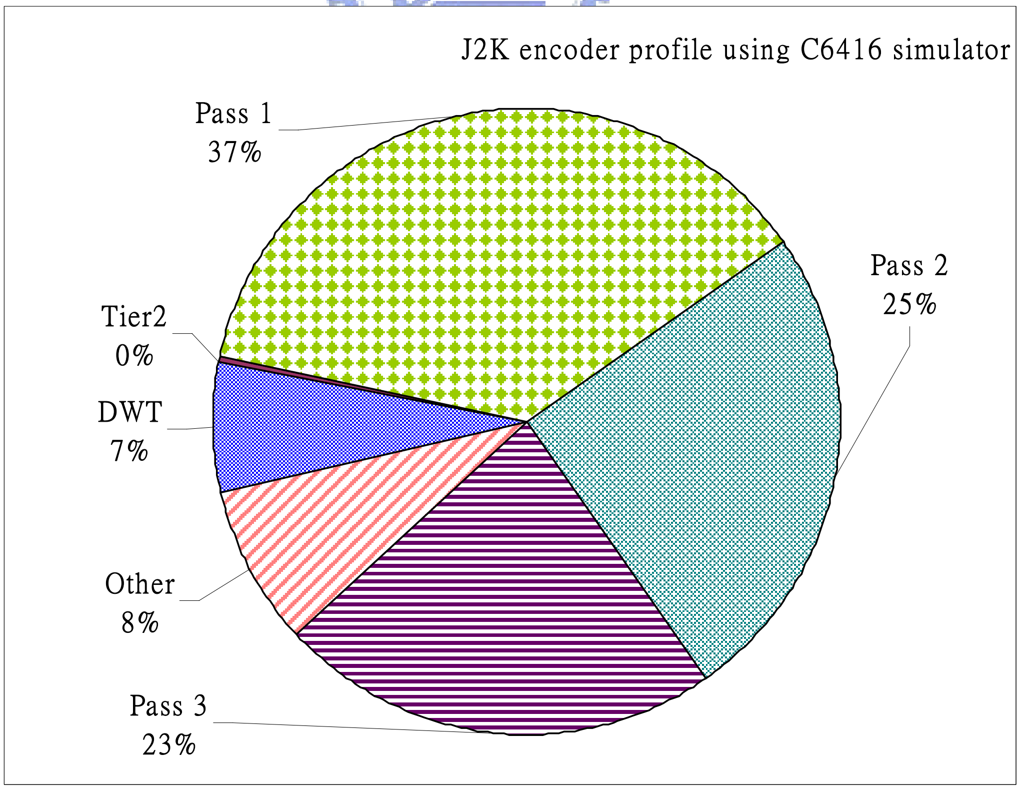


Figure 4-7 Complexity profiling of the JPEG2000 encoder on the C6416 simulator

4.3 Major Encumbrances

In last section, we identify two major bottlenecks in running JPEG2000 on a DSP system. One is that the actual cycles on the DSP platform are more than the CPU cycles. The other is that the Tier1 is the most complexity module in the JPEG2000 encoder. We will look into the problems and try to improve the JPEG2000 encoder on the DSP platform.

4.3.1 Memory System

The C64xx CPU cycle accurate simulator uses the flat memory system. It ignores the locality of the instructions and data. But on the real DSP platform nine times of cycles are required. Table 4-4 shows the cycle distributions generated by the C6416 simulator. The core processing cycles are only 12 % of the total cycles. The stall cycles are the most critical part in the total cycles. In the memory hierarchy of our DSP platform, the L1D is too small so that the data miss frequently occurs. The large main memory has a long access time although it is cheap in cost. We know that the speed gap between CPU and memory speed is large. The numerous stall cycles means that the system wastes a lot time in transferring data. If the most data are in the cache, the stall cycles will decrease.

Our DSP platform has 16 Kbytes L1D cache and 16 Kbytes L1P cache. In the section allocation map, the small sections such as text (instructions), stack, and const occupy about one fifth of the SRAM. Usually, the heap size is inevitable large in an image encoder and the heap data must locate in the external memory. We can modify the data structure in the OpenJPEG software and try to improve the SRAM utilization rate. In order to test this method, we modify some dynamic data that are used frequently, such as code-block and flags variables, and set them to the static variables located in the SRAM. As shown in Table 4-4, the stall cycles decrease to 66 % of original one and the data cache hit rate arises from 77% to 84%. This method is a common method to improve performance but it is not a convenient way. The architecture of TI TMSC6000 family mentioned in section 3.4.1 provides two-levels of cache memory. If we also use the L2 cache, it brings in a great improvement as shown in Table 4-4. The data cache hit rate has turned up to 99% so that the stall cycles decrease to 3 % of original one. The percentage of the core cycles also arises to 82 %. Because the four fifths of the SRAM are available, we set the L2 cache size to its maximum located in the SRAM. The larger cache reduces the memory read misses, but it uses a large area (higher cost) and longer hit time. In this project, the two-level cache configuration leads the most benefit than one-level cache.

Now again, we profile the JPEG2000 encoder with the L2 cache system. Also, we adopt the optimization methods described in section 3.4.2 . Remember that the C64xx simulator uses the flat memory system. The results are shown in Figure 4-8 and Figure 4-9. The most parts are 69% and 97% of total cycles respectively. In flat memory system, the cycles of DWT module are only 6% of the reality system. It means that the DWT module takes much time in fetching data for wavelet transform. However, we still focus on the Tier-1 which is the most cycles of the entire system. We will discuss the code-block coding in next section.

C6416 simulator	Original		Common		L2 cache	
Event	Cycles	Percentage	Cycles	Percentage	Cycles	Percentage
Total Cycles	8,392,238,361	N/A	5,915,074,610	N/A	1,186,156,486	N/A
Core cycles(excl. stalls)	967,163,032	12	964,032,493	16	967,163,800	82
NOP cycles	397,863,892	41	410,684,804	42	397,864,176	41
Stall Cycles	7,425,076,475	88	4,951,043,649	84	218,992,734	18
Cross Path Stalls	3,333,928	0	4,395,347	0	3,333,928	0
L1P Stall Cycles	29,706,277	0	27,351,604	0	26,902,533	2
L1D Stall Cycles	7,392,048,013	88	4,919,312,165	83	188,644,902	16
Instruction cache hits	212,197,536	95	211,727,096	94	213,184,547	95
Instruction cache misses	12,255,209	5	12,605,583	6	11,268,436	5
Data cache references	314,088,122	N/A	304,776,872	N/A	314,088,293	N/A
Data cache reads	209,003,568	67	199,697,704	66	209,003,658	67
Data cache writes	105,084,554	33	105,079,175	34	105,084,630	33
Data cache hits	243,394,864	77	256,297,355	84	310,833,373	99
Data cache read hits	156,975,567	75	163,526,475	82	207,879,556	99
Data cache write hits	86,419,297	82	92,770,880	88	102,953,817	98
Data cache misses	70,693,258	23	48,479,517	16	3,254,920	1
Data cache read misses	52,028,000	25	36,171,222	18	1,124,106	1
Data cache write misses	18,665,258	18	12,308,295	12	2,130,814	2

Table 4-4 The effect of using L2 cache memory

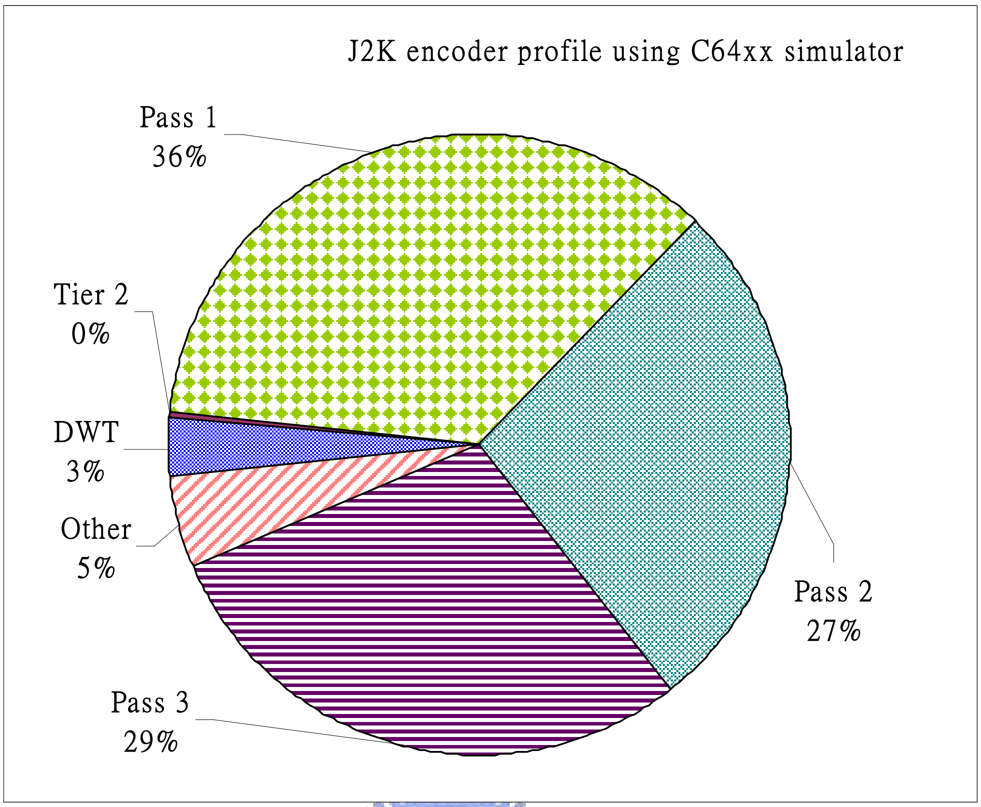


Figure 4-8 Profile using file level optimization (-o3) on C64xx simulator

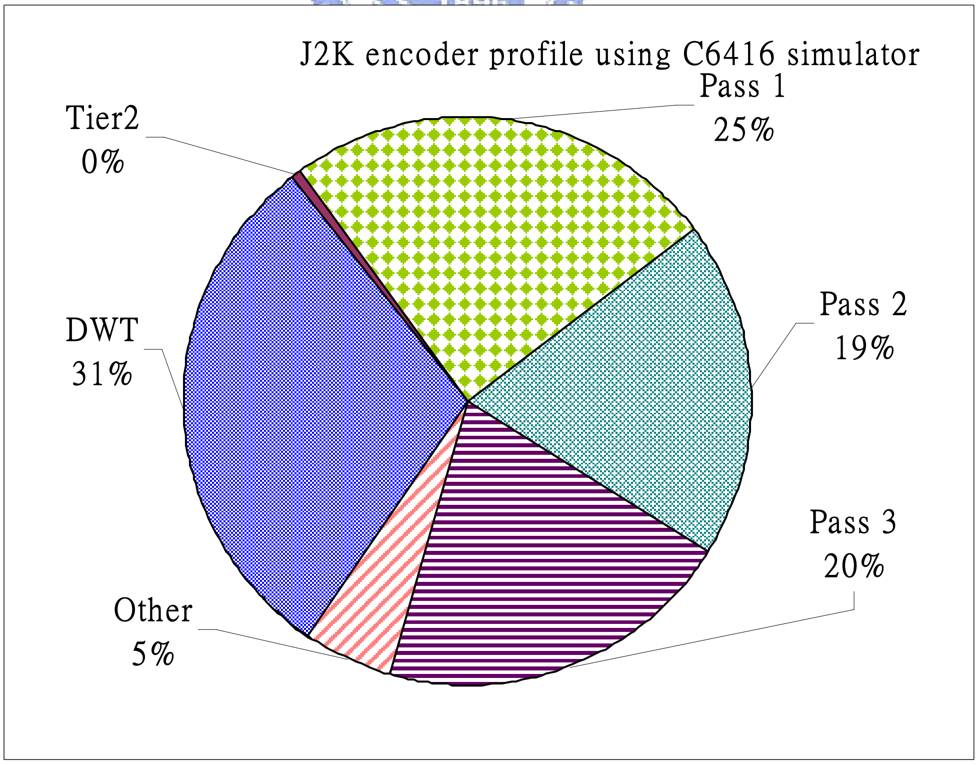


Figure 4-9 Profile using L2 cache and file level optimization (-o3) on C6416 simulator

4.3.2 Analysis of Bit-Plane Coding

In this section, we will discuss another obstruction, which is the most critical module in the JPEG2000 encoding flow. The tier-1 module is the most complex part in the encoder. In our DSP platform, it takes 97% of total cycles in the JPEG2000 algorithm. As discussed in section 2.4.1 , the bit-plane coding is about the main part in Tier1 module. In the bit-plane coding, an n by n dimension image takes about $n \times n \times (3 \times bn - 2)$ clocks to complete. The "bn" means the number of bit-planes. The procedure of the bit-plane coding is shown in Figure 4-10. There are many branch conditions in the flowchart and the MQ arithmetic encoder also takes a number of cycles in the bit-plane coding. However, the MQ coder is already a mature technique. Therefore, we focus on the Pass operations.

At the beginning, all the samples are in an insignificant state. The Pass3 process must be performed at the first most significant bit-plane. The decision of the Pass3 process loops until all samples are checked by the Pass3 process. Because similar states of samples often cluster in the bit-plane, the Pass3 process encodes nearly all samples continuously in the higher bit-planes. That is, most samples are insignificant and the samples of their neighborhood are also insignificant. The Run-Length coding provides an efficient way to encode these samples and produces a better compression ratio.

Starting from the next bit-plane, the Pass1 process scans the states of all samples and then the Pass2 process scans the states of all samples again after Pass1. Finally, the Pass3 process ends this bit-plane. After one bit-plane process is done, the entire procedure repeats for the next bit-plane. A bit-plane process usually includes three Pass processes as described above except for the most significant bit-plane. In this way, the coding procedure continues until all bit-planes are done. In pseudo code form, the passes are described as below [22]:

Algorithm 1 Significance pass algorithm

```
1:  for each sample in code-block do
2:      if sample previously insignificant and predicted to become significant during current
          bit plane then
3:          code significance of sample /* 1 binary symbol */
4:          if sample significant then
```

```
5:            code sign of sample /* 1 binary symbol */
6:         endif
7:      endif
8:   endfor
```

Algorithm 2 Refinement pass algorithm

```
1:   for each sample in code-block do
2:      if sample found significant in previous bit-plane then
3:         code next most significant bit in sample /* 1 binary symbol */
4:      endif
5:   endfor
```

Algorithm 3 Cleanup pass algorithm

```
1:   for each vertical scan in code-block do
2:      if four samples in vertical scan and all previously insignificant and unvisited and non
          have significant 8-connected neighbor then
3:         code number of leading insignificant samples via aggregation
4:         skip over any samples indicated as insignificant by aggregation
5:      endif
6:      while more samples to process in vertical scan do
7:         if sample previously to process in vertical scan then
8:            code significance of sample if not already implied by run /* 1 binary symbol */
9:            if sample significant then
10:              code sign of sample /* 1 binary symbol */
11:           endif
12:        endif
13:     endwhile
14: endfor
```

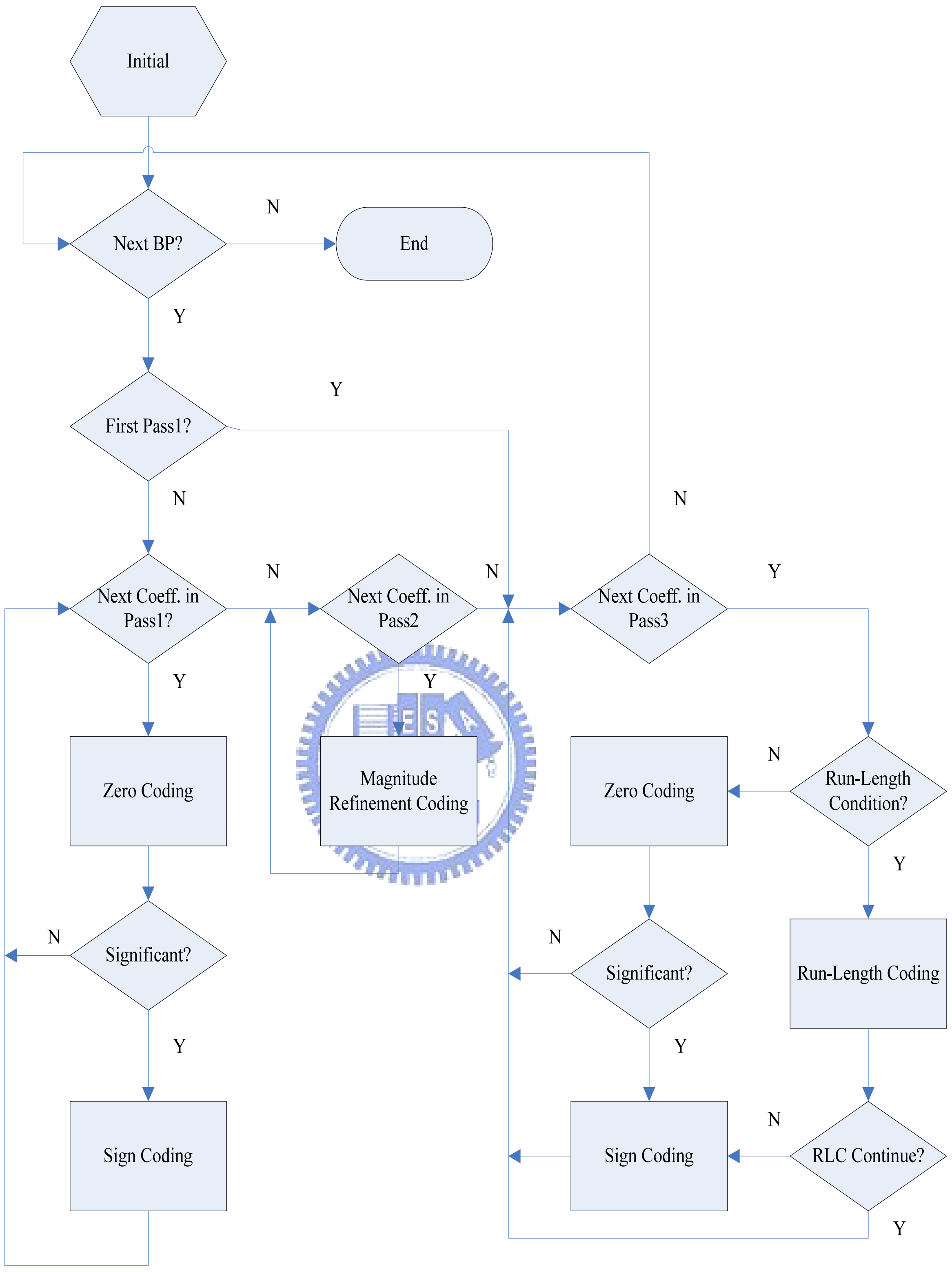
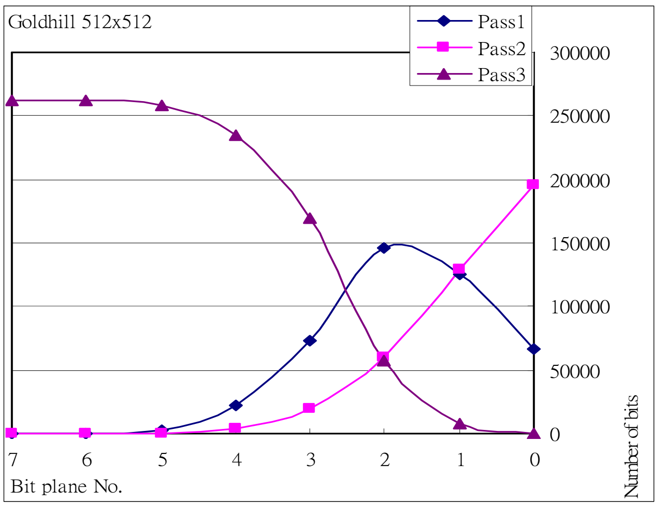
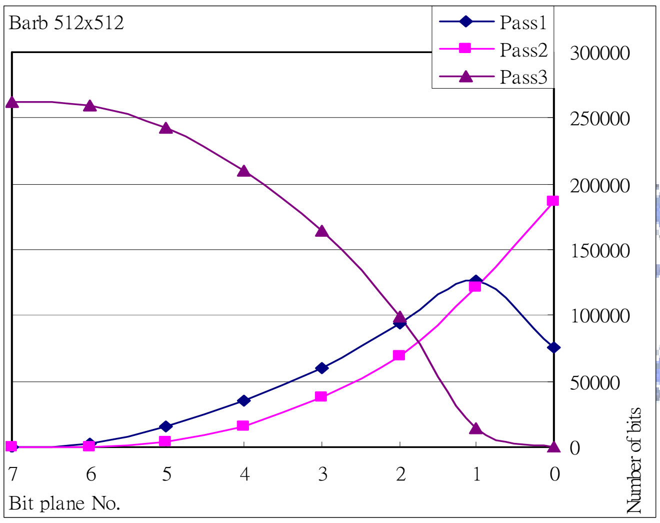


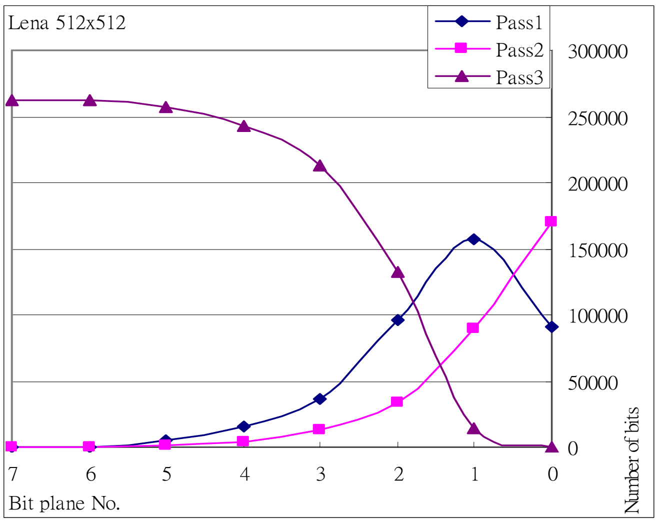
Figure 4-10 Flowchart of bit-plane coding



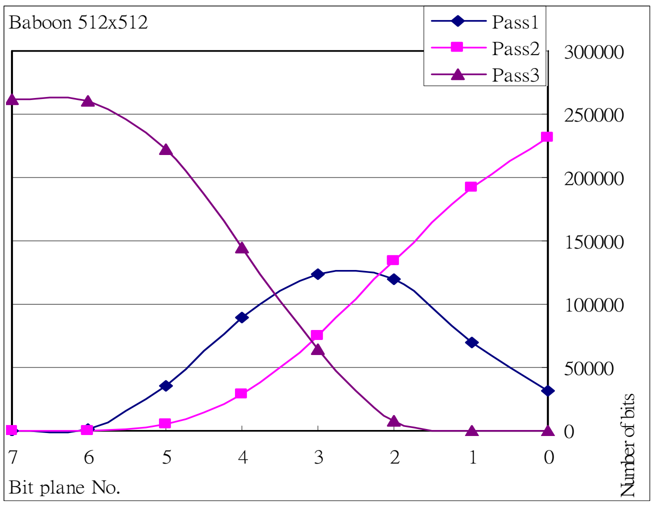
Bit-plane	Pass1	Pass2	Pass3
7	0.00%	0.00%	100.00%
6	0.01%	0.00%	99.99%
5	1.20%	0.21%	98.59%
4	8.63%	1.61%	89.76%
3	27.85%	7.25%	64.91%
2	55.65%	22.64%	21.70%
1	47.97%	49.01%	3.02%
0	25.32%	74.62%	0.06%



Bit-plane	Pass1	Pass2	Pass3
7	0.00%	0.00%	100.00%
6	1.10%	0.12%	98.77%
5	5.82%	1.61%	92.57%
4	13.62%	6.18%	80.20%
3	22.84%	14.26%	62.90%
2	35.94%	26.21%	37.85%
1	48.23%	46.07%	5.69%
0	28.73%	71.25%	0.02%



Bit-plane	Pass1	Pass2	Pass3
7	0.00%	0.00%	100.00%
6	0.12%	0.01%	99.87%
5	1.82%	0.32%	97.86%
4	5.91%	1.57%	92.52%
3	14.11%	4.77%	81.12%
2	36.79%	12.81%	50.41%
1	60.18%	34.43%	5.39%
0	34.80%	64.96%	0.24%



Bit-plane	Pass1	Pass2	Pass3
7	0.00%	0.00%	100.00%
6	0.63%	0.09%	99.27%
5	13.41%	1.94%	84.66%
4	34.08%	10.88%	55.04%
3	47.17%	28.40%	24.43%
2	45.74%	51.43%	2.83%
1	26.55%	73.43%	0.02%
0	11.89%	88.11%	0.00%

Figure 4-11 Analysis of Pass Contribution

We collect the statistics of these Pass processes and show them in Figure 4-11. We count the samples that are encoded in each pass. From the most significant bit to the least significant bit, the results are shown by curves. As Figure 4-11 shows that each coding Pass has to scan n^2 samples in a bit-plane, but not all scans are necessary. For example, in “Goldhill 512x512”, the percentage of the samples coding by Pass1 is extremely low from bit-plane 7 to bit-plane 5. They are typically lower than 10 % of the total samples for most images. Then, the maximum number of samples encoded by Pass1 are about half of total samples. The result indicates that most of the checking processes are wasted. The Pass2 process handles the samples that are significant determined by the Pass1 process or the Pass3 process. From the most significant bit to the least significant bit, the samples, which should be encoded in the Pass2 process, usually increase starting from a tiny number. This process wastes a lot of time in checking samples too. The Pass3 process handles the samples most at the higher bit-planes. Then, the samples that should be encoded in the Pass3 process decrease gradually. The statistics tell us that the Pass3 process often does not encode any samples in the least significant bit-plane or lower bit-planes. This means that the Pass3 process could be skipped and the n^2 cycles of checking samples in the Pass3 process are saved. Based on these observations, we describe several speed-improving methods in next section.

4.4 A Few Known Speed-Improving Methods

In this section, we will describe several speed-improving methods such as the CUPS (Clean Up Pass Skipping) and the PP (Pass Predicting) methods [23] [24], the SS (Sample Skipping) and the GOCS (Group Of Column Skipping) methods [25], and the PPP (Pipelined Processing of Pass) method [26] [27]. In our study, we propose a new method, which is easier to implement, and will describe it in the next chapter.

4.4.1 CUPS and PP Methods

In a bit-plane coding process, the order is Pass1, Pass2, and Pass3. The data in Figure 4-11 shows that sometimes the Pass3 process is unnecessary. The Clean Up Pass skipping (CUPS) method is adopted to terminate the Pass3 process in a bit-plane coding process. This method reduces $n \times n \times (PassZero)$ cycles of checking state. The "PassZero" means the number of Pass3 that can be skipped in the bit-plane process. This is similar to the method called "early termination" for accelerating motion estimation in video coding. In using the CUPS method, we count the numbers of coded samples in the Pass1 and the Pass2 processes in the current bit-plane. If all samples are coded in the Pass1 and the Pass2 processes, we can enable the CUPS mode to skip the following Pass3 process. The flowchart is described in Figure 4-12. We use one flag and one accumulator to perform the CUPS method. In simulations, we count the numbers of Pass3 calls in all bit-planes shown in Table 4-5. When the CUPS method is adopted, the calls decrease as shown in Table 4-5. We use the C64xx simulator to simulate the CUPS method as shown in Table 4-6. Due to additional overhead, the DSP implementation does not save as much as those in Table 4-5.

Image	Original Calls	CUPS Calls	CUPS/Original Calls
Goldhill	421	360	85 %
Barb	430	379	88 %
Lena	411	378	91 %
Baboon	476	350	73 %

Table 4-5 Calls of Pass3 process function

Image	Function	Original Cycles	CUPS Cycles	CUPS/Original
Goldhill	t1_enc_clnpass	132,353,060	117,463,739	89 %
Barb	t1_enc_clnpass	133,847,824	121,988,745	91 %
Lena	t1_enc_clnpass	129,077,570	121,369,189	94 %
Baboon	t1_enc_clnpass	147,598,193	116,370,411	79 %

Table 4-6 Comparison with the CUPS method

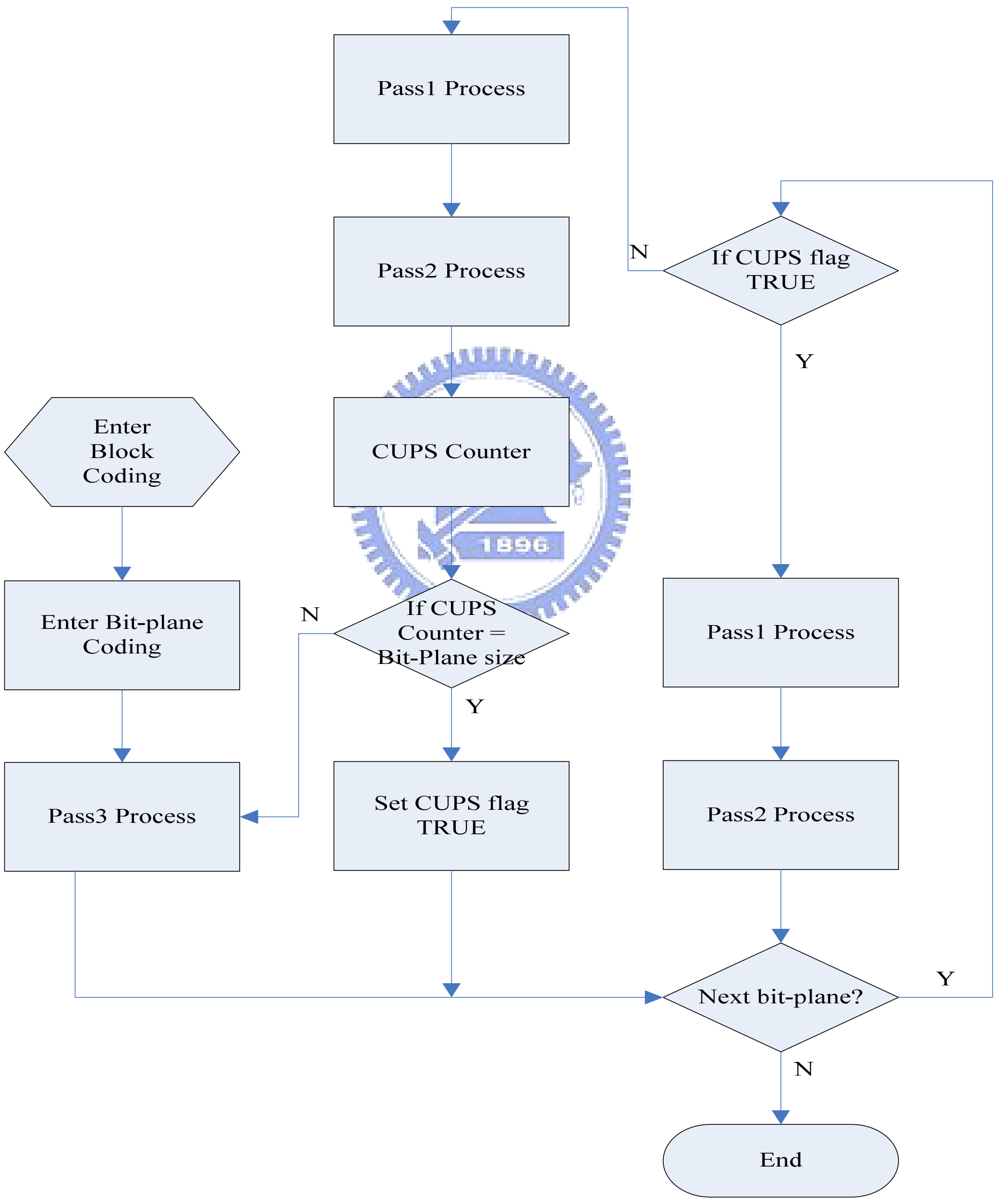


Figure 4-12 Flowchart of the CUPS method

The PP method is based on the significant sample inheritance. When the state of a sample that has just been checked in the Pass1 process becomes significant at a certain bit-plane, it affects its eight neighbors in all directions in the next lower bit-plane as shown in Figure 4-13. Thus, in the significance propagation pass, the samples which should be encoded by the Pass1 process can be predicted in the last bit-plane. Using a prediction table that records the address of these samples can reduce the clock cycles for checking states. This PP method using prediction table enables the Pass1 process to process the Pass1 samples directly without checking their states first.

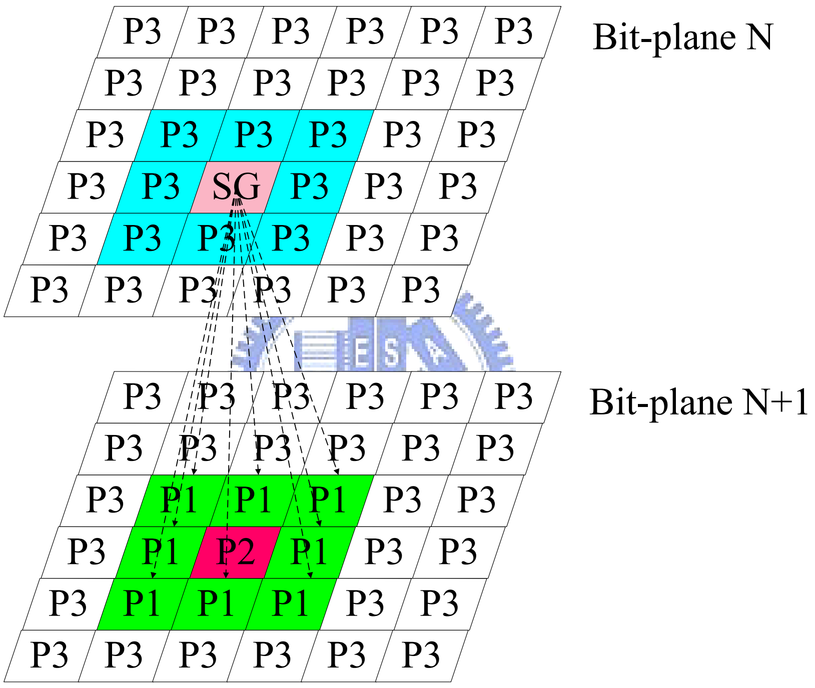


Figure 4-13 Significant sample inheritance

Although the significant samples can predict Pass1 samples on the next bit-plane, they can not predict some samples that become significant when the coding process is applied to the next bit-plane as shown in Figure 4-14. In the strip scanning step for the Pass1 process, if a sample (P1) is checked and becomes significant state (sP1), then, its neighbors (nP1) become the members of the Pass1 process in the current bit-plane. We notice that these neighbors we mention in the above are not scanned yet and they must be coded in the current Pass1 process. In other word, these samples are unpredictable in last bit-plane and can not be found on the prediction table. For this reason, there are a few strategies to include these missing samples. We will give an introduction as follows.

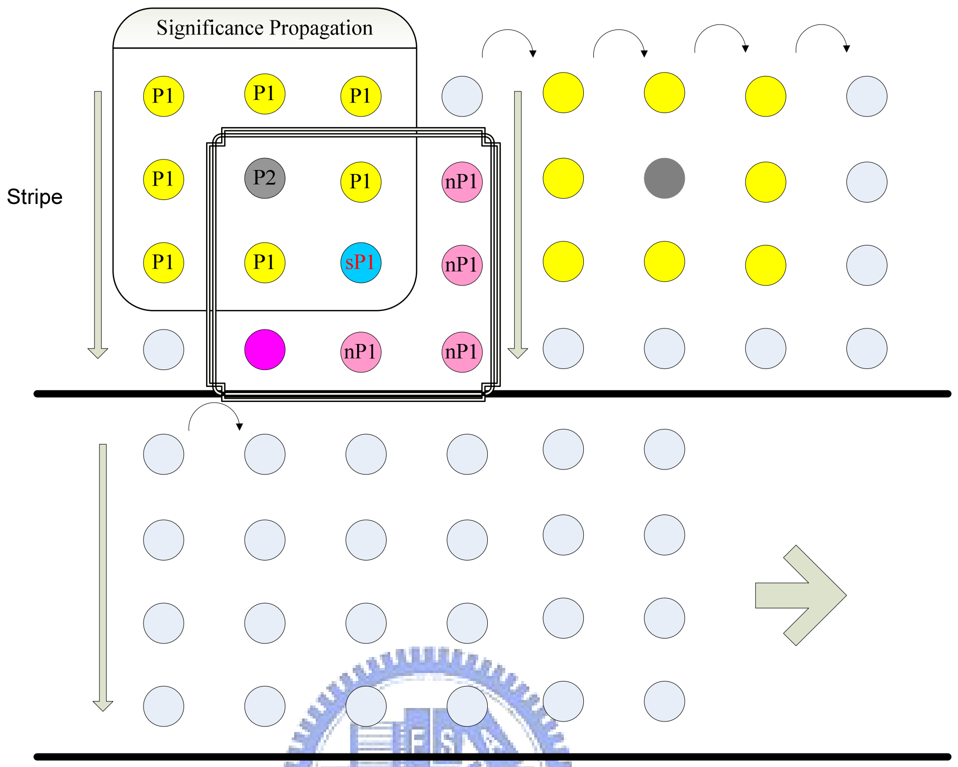


Figure 4-14 Significance Propagation

There are two methods to fix the prediction table. One is the continuous-five mode and the other is the boundary extension mode. The continuous-five mode is adopted for finding those samples that are unpredictable. Four conditions are identified in the scanning order as shown in Figure 4-15. All the addresses of the unpredictable samples are recorded in another table called CM (continuous-five mode) table. If the sP1 is located in the lower boundary of a stripe as show in Figure 4-16, the boundary extension mode must also be enabled. Three addresses, (current address＋4xcode-block width－7), (current address＋4xcode-block width －3), and (current address＋4xcode-block width＋1) are recorded in another table for the boundary extension mode. According to these two strategies, a few more samples are checked in the Pass1 process as shown in Figure 4-17. The samples (C) from continuous-five mode are recorded in the CM table and the boundary extension samples (B) are recorded in the BM table.

The pseudo code described in [23] is insufficient. A comparator is needed to compare the contents or the same index on three tables. The smallest one is chosen, and its address is fetched from the corresponding table. The Pass1 process must wait for the comparator to fetch the correct sample, which has a priority higher than the others and this method is not efficient in sequential processing software. It is easier to implement it using the hardware architecture.

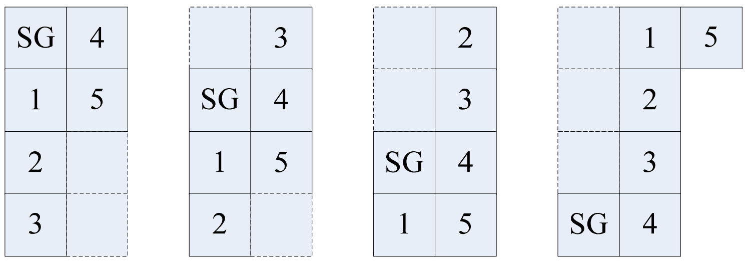


Figure 4-15 Four conditions in continuous-five mode

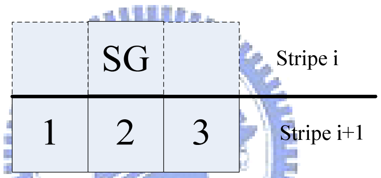
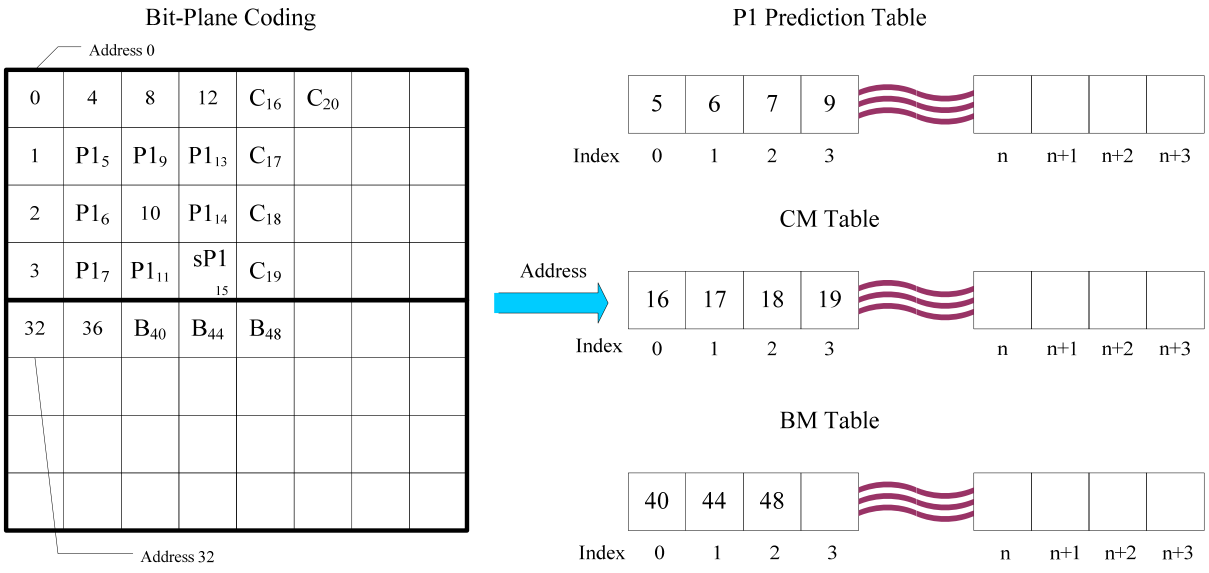


Figure 4-16 Boundary extension



Bit-Plane Coding
Address 0
0 4 8 12 C_{16} C_{20}
1 $P1_5$ $P1_9$ $P1_{13}$ C_{17}
2 $P1_6$ 10 $P1_{14}$ C_{18}
3 $P1_7$ $P1_{11}$ $sP1_{15}$ C_{19}
32 36 B_{40} B_{44} B_{48}
Address 32
P1 Prediction Table
5 6 7 9
Index 0 1 2 3 n n+1 n+2 n+3
CM Table
Address
16 17 18 19
Index 0 1 2 3 n n+1 n+2 n+3
BM Table
40 44 48
Index 0 1 2 3 n n+1 n+2 n+3

Figure 4-17 Prediction table for Pass1

4.4.2 SS and GOCS Methods

In [25], two methods are devised to reduce the computing time of bit-plane coding. Each sample is checked for three times, one for each pass, but it is only coded once in one of the three passes. Coding a 64 by 64 code-block with N magnitude bit-plane costs at least 64×64×N×3 cycles. Many bubbles (empty operations) are generated if the bit-plane coder scans and checks every sample in this manner. A column-based operation is proposed to remove bubble and reuse data instead of sample-based operation, which is the original method used in the JPEG2000 reference software.

The first method is called "Sample Skipping (SS)" method that is illustrated by Figure 4-18. The marked NBC sample is the "need-to-be-coded" sample. It means that the samples must be coded in the current Pass process. The Sample Skipping (SS) skips no–operation samples in a column. In a column checking, 0-4 NBC samples are necessary to be encoded. The serial checking architecture spends four cycles to check no matter how many NBC samples are included. The SS method is essentially a parallel checking architecture. If there are N NBC samples, N cycles are spent on the checking process and thus the other cycles (4-N) are saved. If there are no NBC samples in a certain column, only one cycle is spent for the checking process. The coding flowchart is shown in Figure 4-19. The analysis and result in the [25] are useful but the method is more adequate for a hardware architecture design. Checking 4 samples in parallel is the most critical concept in this method. Our DSP platform is a sequential software-based architecture, and thus checking 4 samples in parallel is not well supported. However the TI DSP VLIW architecture helps quite a lot to implement the SS method in our DSP platform.

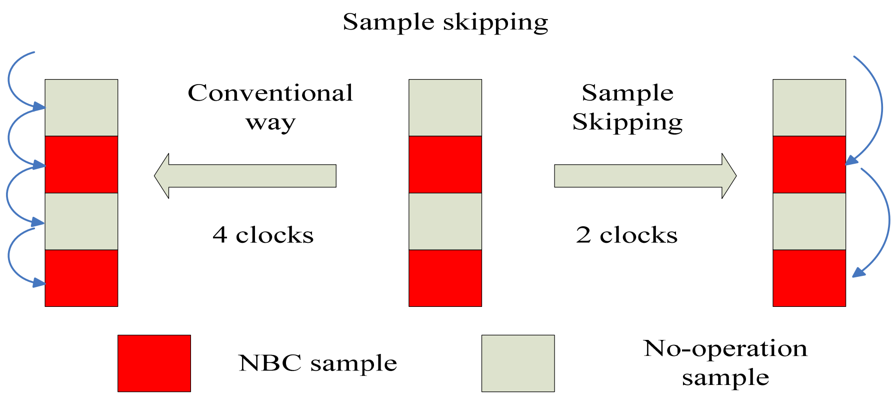


Figure 4-18 Concept of SS method

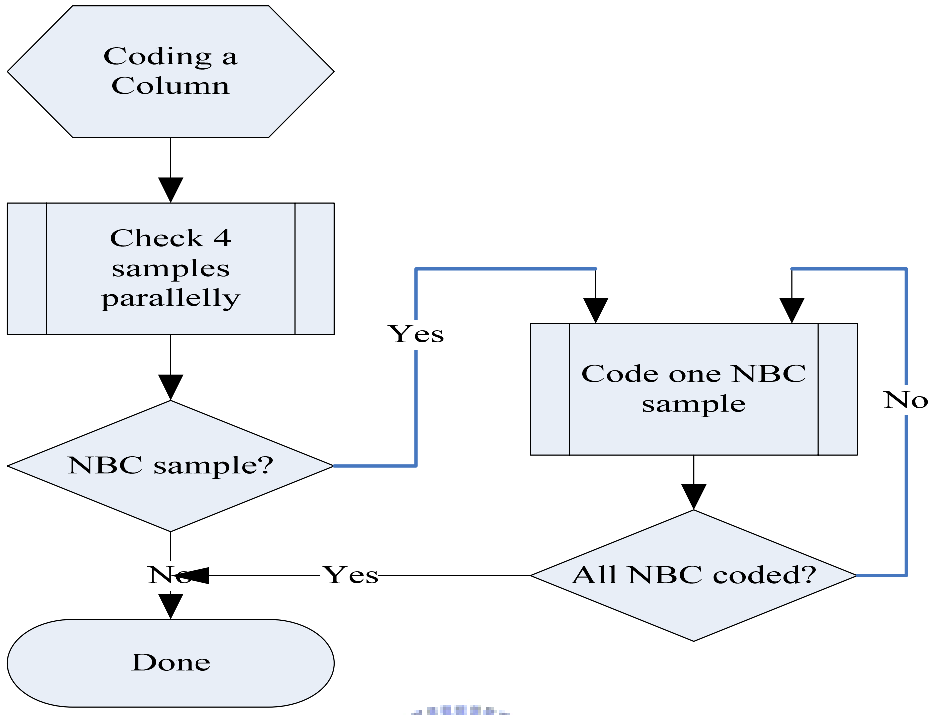


Figure 4-19 Flowchart of SS method

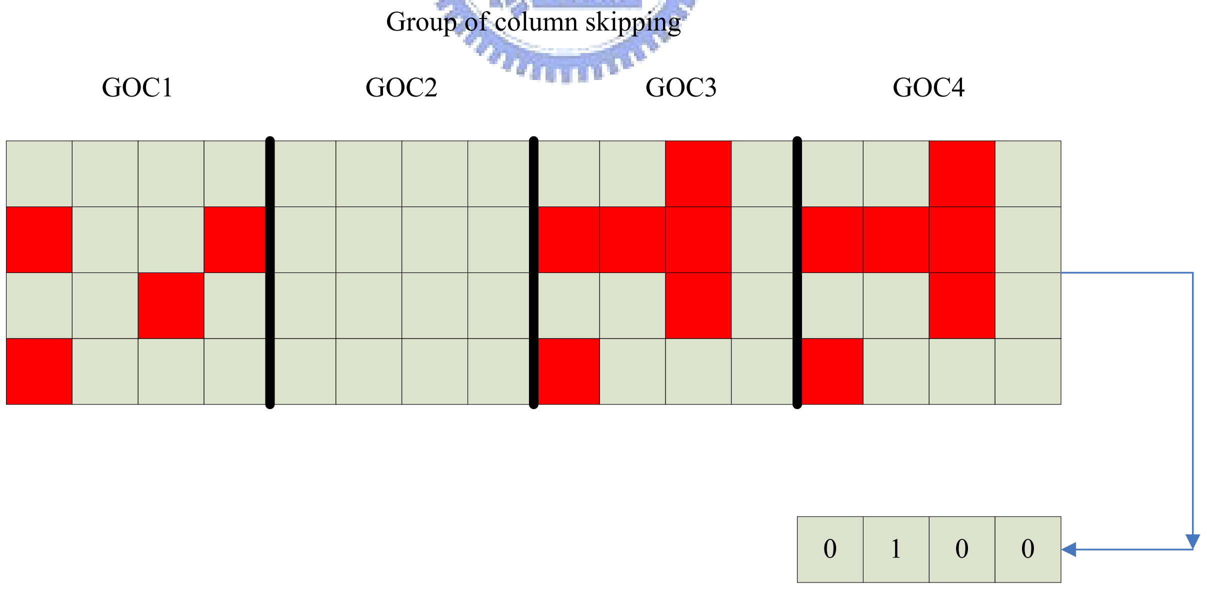


Figure 4-20 Example of the GOCS method

The second method is called the "Group-Of-Column Skipping (GOCS)" method. The concept is to skip a group of no-operation columns in one checking cycle. An example is shown in Figure 4-20, where the four columns are grouped into one group-of-column (GOC). Only 4 bits flags are used to record these 4 GOC consisting of for 16 columns. The GOC2 contains no NBC samples and a '1' bit is recorded in the GOCS table. A '0' bit means that there are some NBC samples in the GOC. The number of NBC samples in each group should be checked and recorded before the Pass1 process and all samples should be classified to each Pass processes as shown in Figure 4-21. Then, Pass1, Pass2 and Pass3 processes are executed, and the flags in GOCS tables are checked. If the flag is '1' for current coding GOC, all columns of this group can be skipped. When the '0' flag is found, the process should check all 4 columns one by one. Because the SS method usually is combined with the GOCS method together, all samples in a column will be checked at the same time. When these two methods are applied, both the processing cycles and the number of memory access can be reduced.

In [25], a run-time analysis with different number of columns as a group is presented as shown in Figure 4-22. It is a C program simulation with the SS and GOCS techniques. The result shows that eight columns as a group has the best run-time performance in all test patterns. The best one which adopts the two methods in simulation is better than the worst one by about 2 %. Here, we also simulate the different number of columns as a group by the C64xx simulator and the result is presented below.

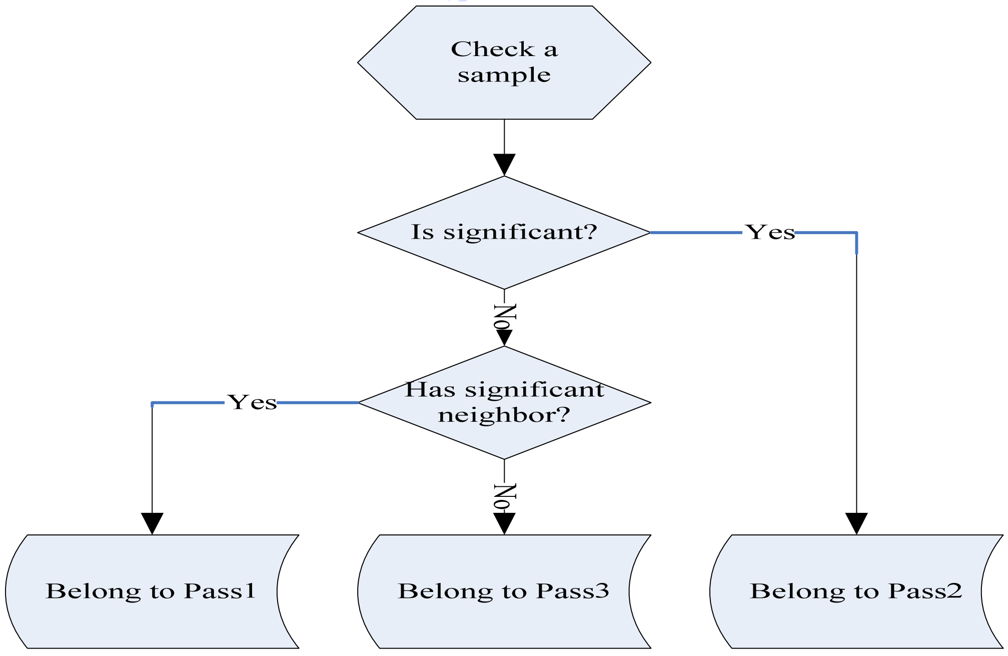


Figure 4-21 Flowchart of sample checking

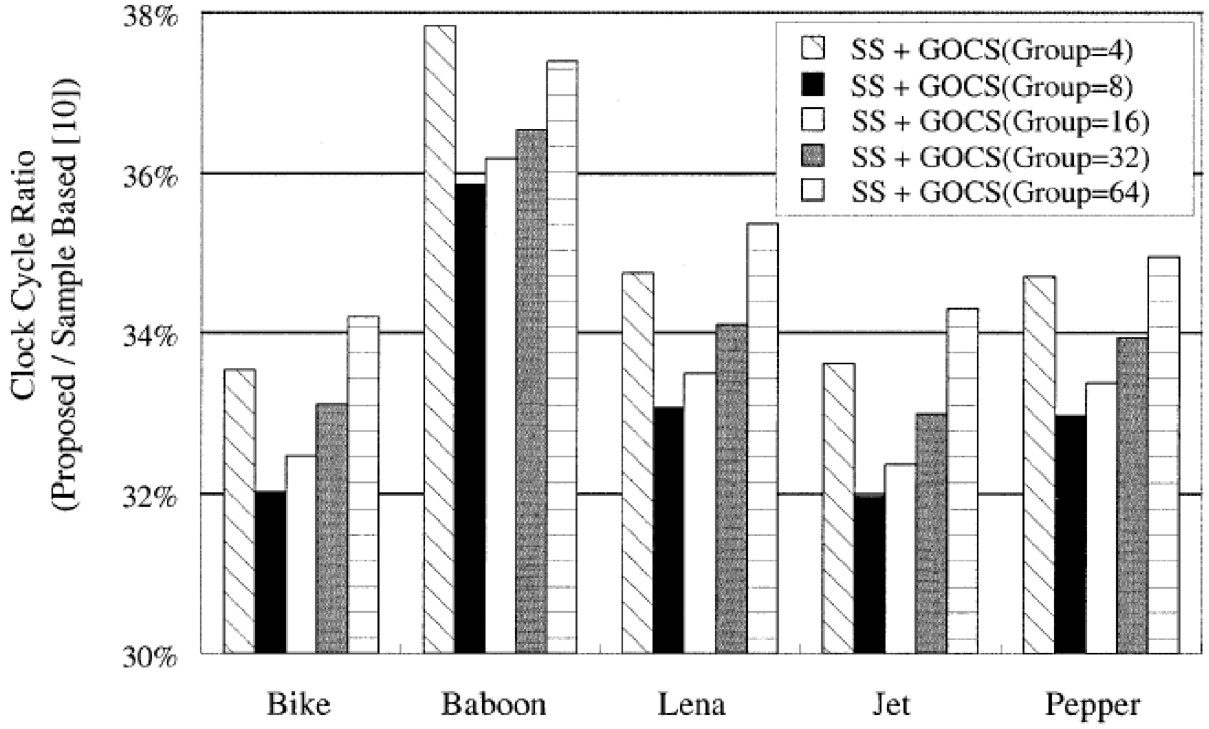


Figure 4-22 Analysis with different number of columns as a group [25]

First, we try the SS method. The data dependence causes problems and decreases the parallelism of executing instructions in the VLIW architecture. The SS method can finally be applied by reducing the data dependence and parallel programming style. Although all the instructions are scheduled by the compiler, we still have a great improvement by unrolling the four contiguous samples. We list the result in Table 4-7. The Pass1, Pass2, and Pass3 processes are the major parts in the "Tier1" operation. The improvement is about 30 % on the average for all test patterns. Moreover, the GOCS method with different GOC sizes is examined as shown in Table 4-8. The GOCS (4) method which means four columns as a group is adopted. We can see that cycles of the Pass2 and Pass3 processes in GOCS (4) are lower than the cycles of the SS method. However, the cycles of Pass1 process increase in order to apply the GOCS method. The total cycles in both GOCS (4) and SS methods are less than the cycles of SS method only. Based on the results in Figure 4-22, the cycle ratio increases as the group size increasing except for the case of four columns as a group. But the results on C64xx simulator are somewhat different. The cycle ratio goes up as the group size increases on the C64xx simulator. Anyhow, we have tested the other images, and the cost is not worth for increasing the group size.

Image	Method	Pass1	Pass2	Pass3	Tier1
Goldhill	Original	161,696,792	123,912,376	132,353,060	435,017,649
	SS	118,713,423	83,794,964	93,997,011	313,560,819
	SS/Ori. (%)	73 %	68 %	71 %	72 %
Barb	Original	158,478,206	129,282,989	133,847,824	438,746,356
	SS	113,817,756	88,715,847	95,188,336	314,859,276
	SS/Ori. (%)	72 %	69 %	71 %	72 %
Lena	Original	153,954,906	108,237,140	129,077,570	408,233,549
	SS	111,328,681	67,345,255	94,722,832	290,360,701
	SS/Ori. (%)	72 %	62 %	73 %	71 %
Baboon	Original	179,696,714	171,827,724	147,598,193	516,672,503
	SS	129,959,350	129,811,716	98,991,484	376,312,422
	SS/Ori. (%)	72 %	76 %	67 %	73 %

Table 4-7 SS method on C64xx simulator

Goldhill	Original	SS+GOCS (4)		SS+GOCS (8)		SS+GOCS (16)		SS+GOCS (32)		SS+GOCS (64)	
Pass1	161,696,792	127,248,882	79%	126,376,830	78%	125,905,854	78%	125,327,618	78%	125,218,498	77%
Pass2	123,912,376	82,816,908	67%	83,753,114	68%	84,888,984	69%	85,792,608	69%	86,954,904	70%
Pass3	132,353,060	83,662,304	63%	85,040,094	64%	86,672,842	65%	88,738,925	67%	90,905,379	69%
Tier1	435,017,649	310,822,507	71%	312,244,472	72%	314,547,950	72%	316,931,613	73%	320,147,339	74%
Barb	Original	SS+GOCS (4)		SS+GOCS (8)		SS+GOCS (16)		SS+GOCS (32)		SS+GOCS (64)	
Pass1	161,696,792	85,158,426	64%	87,028,182	65%	89,785,474	67%	93,278,418	70%	96,634,127	72%
Pass2	123,912,376	87,196,372	67%	88,314,377	68%	89,902,444	70%	91,504,222	71%	93,147,358	72%
Pass3	132,353,060	122,165,800	77%	121,296,140	77%	120,814,604	76%	120,562,892	76%	120,445,628	76%
Tier1	435,017,649	311,697,656	71%	313,795,391	72%	317,665,212	72%	322,500,270	74%	327,377,875	75%
Lena	Original	SS+GOCS (4)		SS+GOCS (8)		SS+GOCS (16)		SS+GOCS (32)		SS+GOCS (64)	
Pass1	161,696,792	88,197,594	68%	89,634,212	69%	91,548,956	71%	94,125,925	73%	96,769,658	75%
Pass2	123,912,376	64,744,920	60%	65,697,045	61%	67,063,623	62%	68,557,606	63%	70,643,014	65%
Pass3	132,353,060	120,188,969	78%	119,307,313	77%	118,851,473	77%	118,613,193	77%	118,502,145	77%
Tier1	435,017,649	290,133,598	71%	291,621,136	71%	294,452,274	72%	298,277,298	73%	302,891,567	74%
Baboon	Original	SS+GOCS (4)		SS+GOCS (8)		SS+GOCS (16)		SS+GOCS (32)		SS+GOCS (64)	
Pass1	161,696,792	80,407,186	54%	81,500,054	55%	83,344,181	56%	85,910,181	58%	89,095,497	60%
Pass2	123,912,376	130,939,612	76%	131,794,101	77%	132,910,354	77%	133,952,793	78%	135,500,769	79%
Pass3	132,353,060	137,998,154	77%	137,086,510	76%	136,537,038	76%	136,249,814	76%	136,116,590	76%
Tier1	435,017,649	366,938,271	71%	367,951,640	71%	370,369,374	72%	373,681,901	72%	378,277,625	73%

Table 4-8 SS + GOCS (different columns as a group) method on C64xx simulator

4.4.3 PPP Method

A parallel processing method is proposed in [26] and [27] called Pipelined Processing of Passes (PPP). Each bit-plane is encoded through three coding passes, called significant propagation pass (Pass1), magnitude refinement pass (Pass2) and cleanup pass (Pass3). All passes are processed sequentially. They can be arranged to process different sets of data in parallel. An example is shown in Figure 4-23. The parallel processing structure matches well the pipeline architecture. In processing the 1st stripe, in order to process Pass2, the context information of current and adjacent stripes which are updated by Pass1 is required. In the meantime, the context information of the Pass3 is updated by Pass1 and Pass2. The strategy of the PPP method is to process the three coding passes of the same bit-plane on different stripes. First, the samples of the 1st stripe in the current bit-plane are processed for Pass1. Then, the samples of the 1st and 2nd stripes in the current bit-plane are processed for Pass2 and Pass1, respectively. Third, the samples of the 1st, 2nd, and 3rd in the current bit-plane are processed by Pass3, Pass2, and Pass1, respectively. Similarly, all the other stripes are processed when all stripes in a coding block are processed, the parallel processing is done. Actually, the data flow of the context formation is not appropriate in using sequential structure software. A lot of work is necessary to apply this method by changing the bit-plane coding flowchart. Usually, the PPP method is suitable only for the hardware design or the multithread program. The TI CCS compiler for the VLIW architecture has already improved the performance by its software pipeline technology.

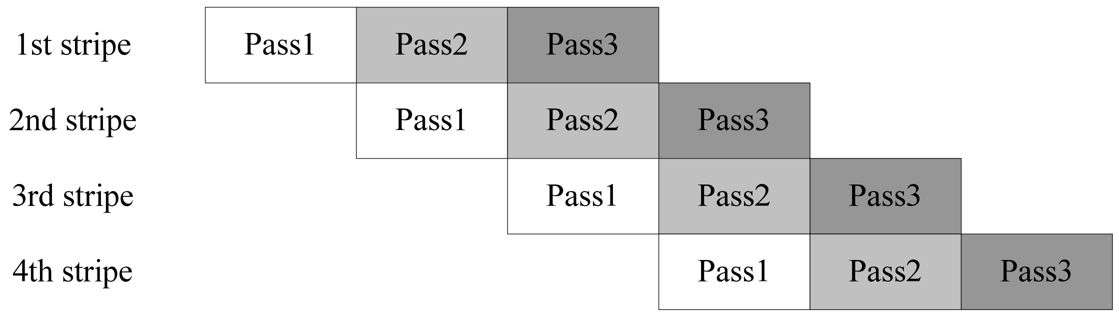


Figure 4-23 Parallel processing of passes

In reference [26], the PPP method is adopted to simulate on the TMS320C6416 (600MHz). Table 4-9 in [26] shows the performance improvement by using the PPP method for Tier-1 coding. The original mode for the three passes takes about 0.96 second for the three test images. The executing time of the Pass2 has reduced up to 41% and the reduction of the Pass3 is up to 32%. However, the executing time of the Pass 1 process is not affected by this method because there is no difference between the PPP method and the original method. The result indicates that the PPP method reduces the processing time for scanning and masking in the case of the Pass2 and Pass3 by reusing the parameter and data used in the Pass1. Although the average improvement of the Tier-1 coding is significant, the previous methods in sections 4.4.1 and 4.4.2 still has better performance than the PPP method. It seems that our DSP platform needs an efficient method to accelerate JPEG2000 algorithm. In this study, we proposed a new method to accelerate and implement JPEG2000 on our DSP platform as described in next chapter.

Image		Lena	PPP/Ori	Baboon	PPP/Ori	Peppers	PPP/Ori
Original mode	Pass1	297.8 ms	N/A	277.9 ms	N/A	269.7 ms	N/A
	Pass2	140.3 ms	N/A	156.8 ms	N/A	157.2 ms	N/A
	Pass3	522.8 ms	N/A	531.7 ms	N/A	533.9 ms	N/A
	Total	960.9 ms	N/A	966.4 ms	N/A	960.8 ms	N/A
PPP method	Pass1	298.6 ms	100 %	281.6 ms	101 %	272.8 ms	101 %
	Pass2	88.5 ms	63 %	96.3 ms	61 %	92.6 ms	59 %
	Pass3	357.4 ms	68 %	369.5 ms	69 %	378.3 ms	71 %
	Total	744.5 ms	77 %	747.4 ms	77 %	743.7 ms	77 %

Table 4-9 Comparison of processing time using PPP method [26]

Chapter 5
Acceleration of JPEG2000 Encoder on DSP Platform

In order to accelerate the JPEG2000 encoder on the DSP platform in a simple and efficient way, a new method called "Variable Group Of Samples Skipping" (VGOSS) is proposed. This method provides an easy way to implement the JPEG2000 encoder on any DSP platform. Comparing with the methods we mentioned before, this method is more suitable for the DSP platform. We will present the concept and discuss the advantages in this chapter. Besides, a modified PP method is proposed, too. Based on the procedure of the VGOSS method, we modify the PP method for our DSP platform. Then, a performance comparison between the VGOSS method and the known methods is presented in section 5.2

5.1 Proposed Acceleration Method

In this section, we describe the concept of our VGOSS method. A detail coding procedure is presented. Then, we integrate the PP method and the VGOSS method together and discuss the effects. Finally the advantages are explained as well. In addition, some other accelerating methods which are tested in our study are presented.

5.1.1 Coding Procedure of VGOSS method

We first need to rearrange the block before the pass coding. In Figure 5-1, all the pass processes in the block-coding must comply with the stripe style scanning. Four vertical continuous columns are grouped in a stripe. The stripe scanning follows the index 0, 1, 2, 3, 4, 5, and so on in order. But the addresses of the stripe data are not continuously stored in a memory.

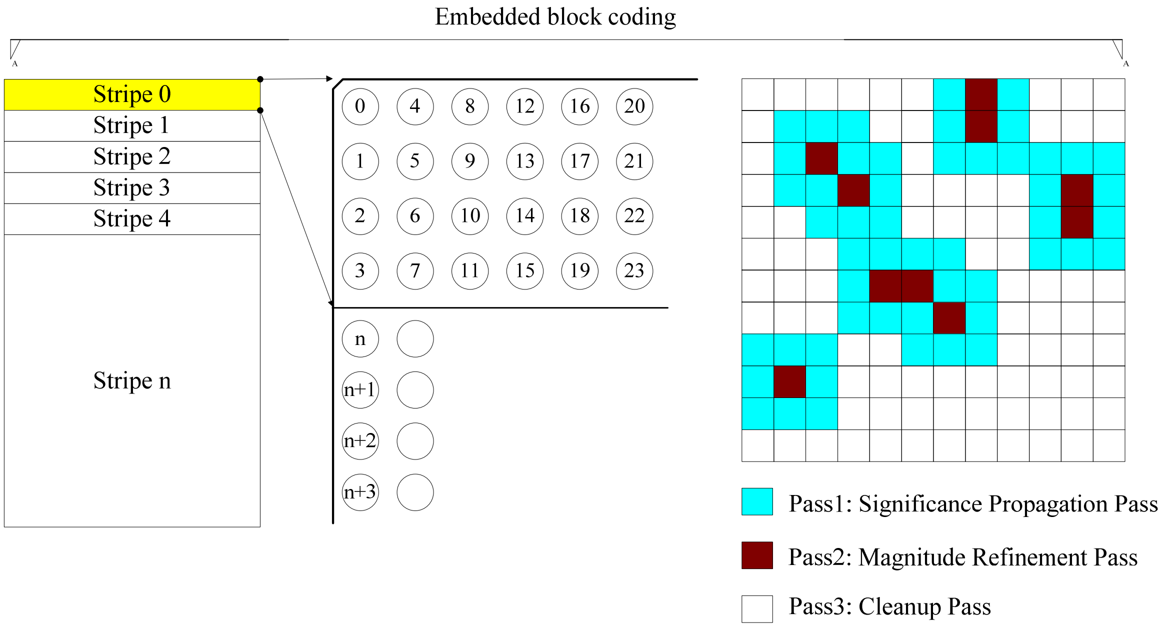


Figure 5-1 Diagram of the stripe and coding pass

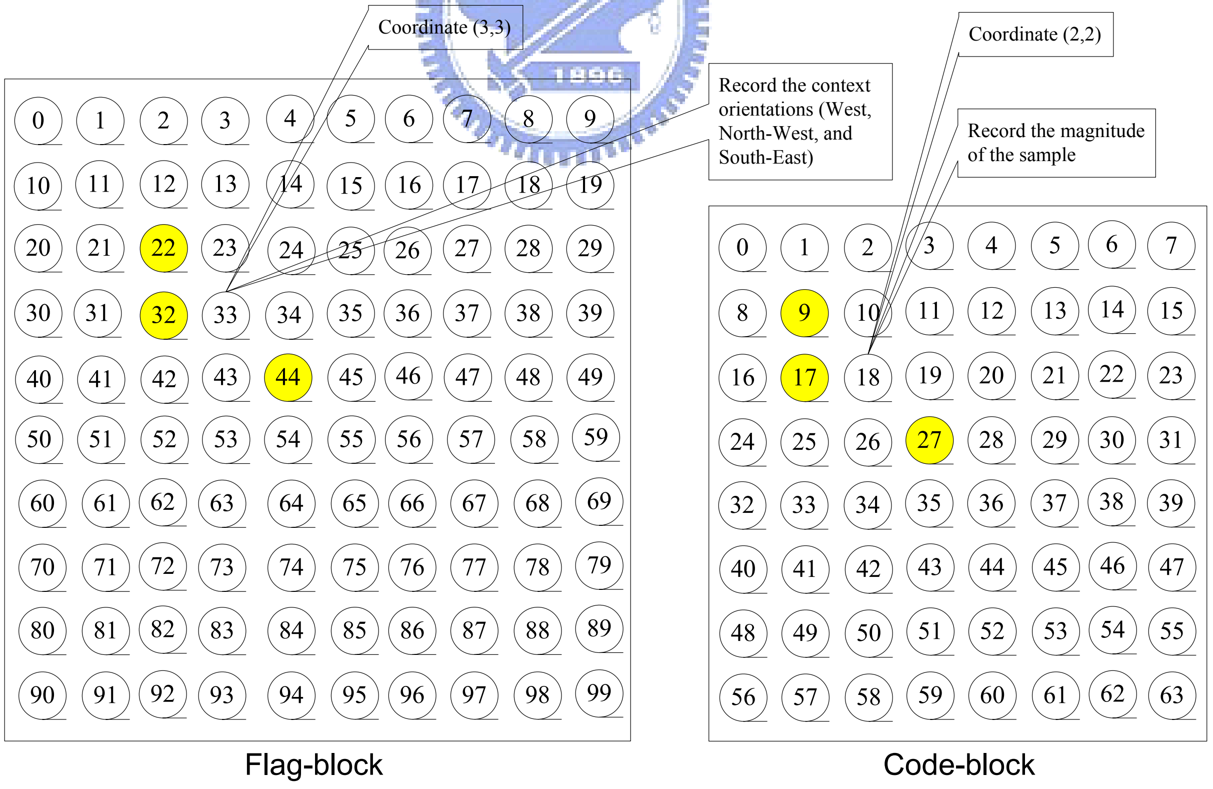

Coordinate (3,3)
Record the context orientations (West, North-West, and South-East)
Coordinate (2,2)
Record the magnitude of the sample
Flag-block
Code-block

Figure 5-2 Flag-block and code-block

We first describe the flag-block and the code-block. For example, an 8 by 8 code-block has 64 samples and each sample has its associated state. The state records the context orientations (defined below) and the significance information in each bit-plane and all states are stored in a flag-block. The states in a bit-plane are covered with the new context orientations in the next bit-plane. The context orientations include North, East, South, West, North-East, South-East, South-West, and North-West. A sample can include all the context orientations and record them in its associated flag in the flag-block. The significance information describes the visited and the significant samples. The visited sample is the coded sample in other pass in the current bit-plane. The significant sample means that a sample has the significant state recorded in its flag. The eight context orientations are specified to indicate the significant neighbors. Any of the three pass processes decide to code this sample by checking its own state. In this case, an 8 by 8 flag-block is necessary to record the sates of all samples. But the flag-block is padded to 10 by 10 for the boundary blocks.

An example of the flag-block and the code-block is shown in Figure 5-2. The shaded samples are significant ones. The coordinate (3, 3) sample in the flag-block records the context orientations (West, North-West, South-East). This means that the coordinate (2, 2) sample of the code-block is next to three significant samples. In this example, the coordinate (x, y) of the code-block sample corresponds to the coordinate (x+1, y+1) sample due to padding. However, in implementation, we convert the 2-D index into 1-D index. The 1-D (33) sample in the flag-block records the states that associate with the 1-D index 18 sample in the code-block. The 1-D index 18 is calculated by the equation, y×(code-block width)+x, and (x, y) = (2,2). All samples and flags are accessed by the one-dimensional index for using the VGOSS method.

Because stored samples of the code-block are rearranged, the flag-block samples are also rearranged. We first describe the rearrangement of the code-block. For example, an eight by eight code-block is shown in Figure 5-3. The stripe scanning addresses are 0, 8, 16, 24, 1, 9, 17, 25, 2, 10, 18, 26, and so on. The shaded samples (9, 17, 27, 4, 41, 51, and 38) have to be encoded in the current pass process. In the JPEG2000 standard, all samples are scanned and coded in the bit-plane but the VGOSS method can encode only the samples that are going to be encoded in the current pass. In order to use the VGOSS method, the order of all addresses must be a continuous sequence. In other words, the code-block should be rearranged in the bit-plane coding. For an N by N code-block, we arrange it to an N/4 by 4N code-block as shown in Figure 5-3.

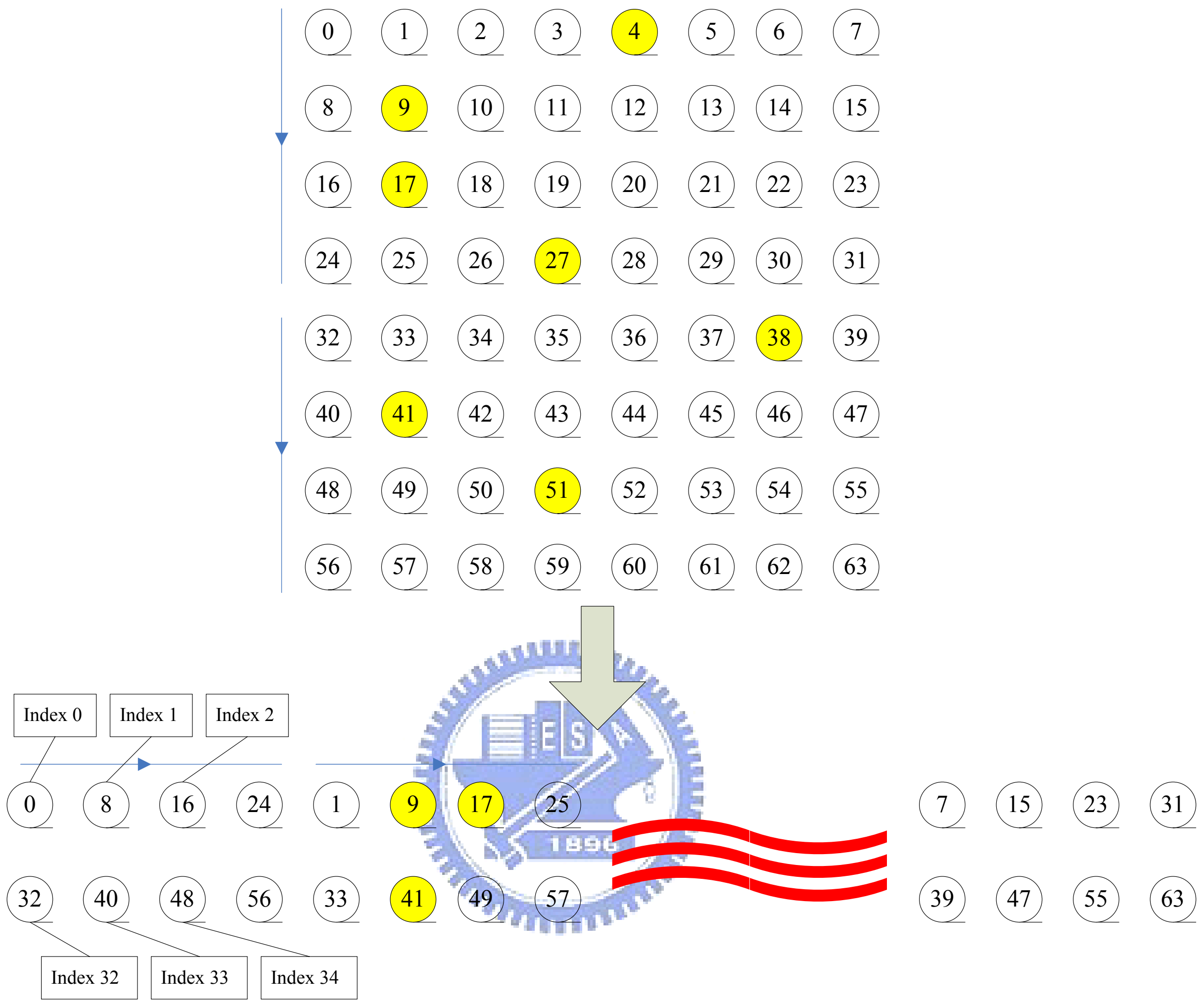
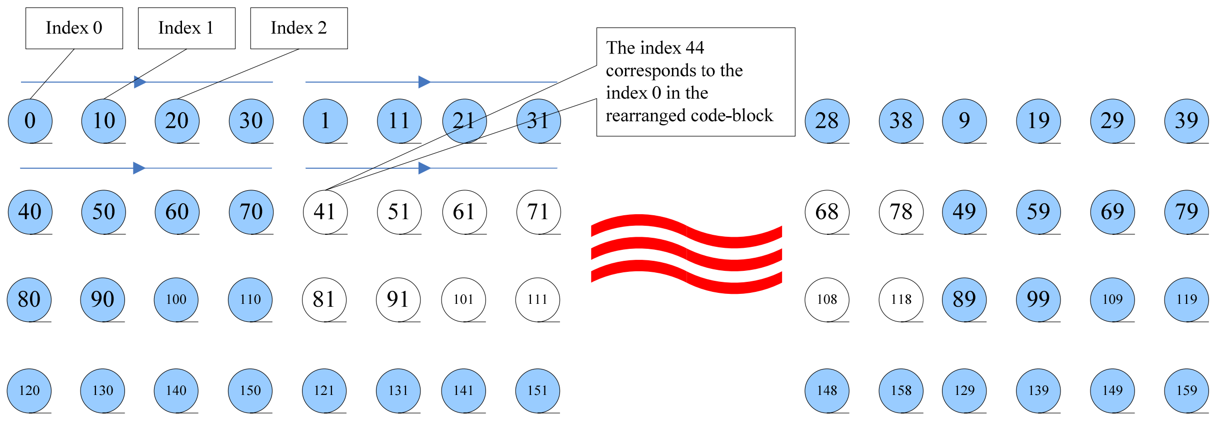


Figure 5-3 Address order of the stripe in the rearranged code-block



Index 0
Index 1
Index 2
The index 44 corresponds to the index 0 in the rearranged code-block

Figure 5-4 Rearranged flag-block with paddings

Also, the flag-block is similarly rearranged. The padded flag-block records the context orientation. The rearranged flag-block shown in Figure 5-4 corresponds to the rearranged code-block shown in Figure 5-3. The padded flag-block has a size of (N/4+2) by (4N+8) as shown in Figure 5-4. The shaded samples (0, 10, 20, 30, 1, 11, 21, 31, and ….) are the padding samples. They are used for the boundary extensions. The rearranged code-block index and the rearranged flag-block index are calculated by equation (5.1.1-1). For example, the index 44 shown in Figure 5-4 corresponds to the index 0 of the rearranged code-block shown in Figure 5-3.

$$\begin{aligned}
&If \quad rearranged \quad codeblock \quad coordinate = (x', y') \\
&codeblock \quad width = (original \quad codeblock \quad width) << 2 \\
&\\
&rearranged \quad codeblock \quad index \\
&= y' \times codeblock \quad width + x' \\
&rearranged \quad flagblock \quad index \\
&= (y'+1) \times (codeblock \quad width + 8) + x' + 4
\end{aligned} \tag{5.1.1-1}$$

The rearrangement does not change the coding performance of on the JPEG2000. The samples of a code-block are fetched from the tile data image and all cycles needed for this new ordering are included in our test results. The cycles of arranging a code-block do not increase on the C64xx simulator (flat memory system) or the C6416 simulator when the compiler-level optimization is not used. Table 5-1 shows the percentages of the increased cycles under different conditions in real tests. In the following experiments, the increase cycles for the rearrangement are included for fair comparison.

Goldhill	C64xx simulator		C6416 simulator with L2 cache	
	Tier1	Increase	Tier1	Increase
Non-opt.	846,100,895	33	878,528,763	400
O3 (file level)	435,017,649	265,828	441,174,161	271,149

Table 5-1 Effect of the data rearrangement

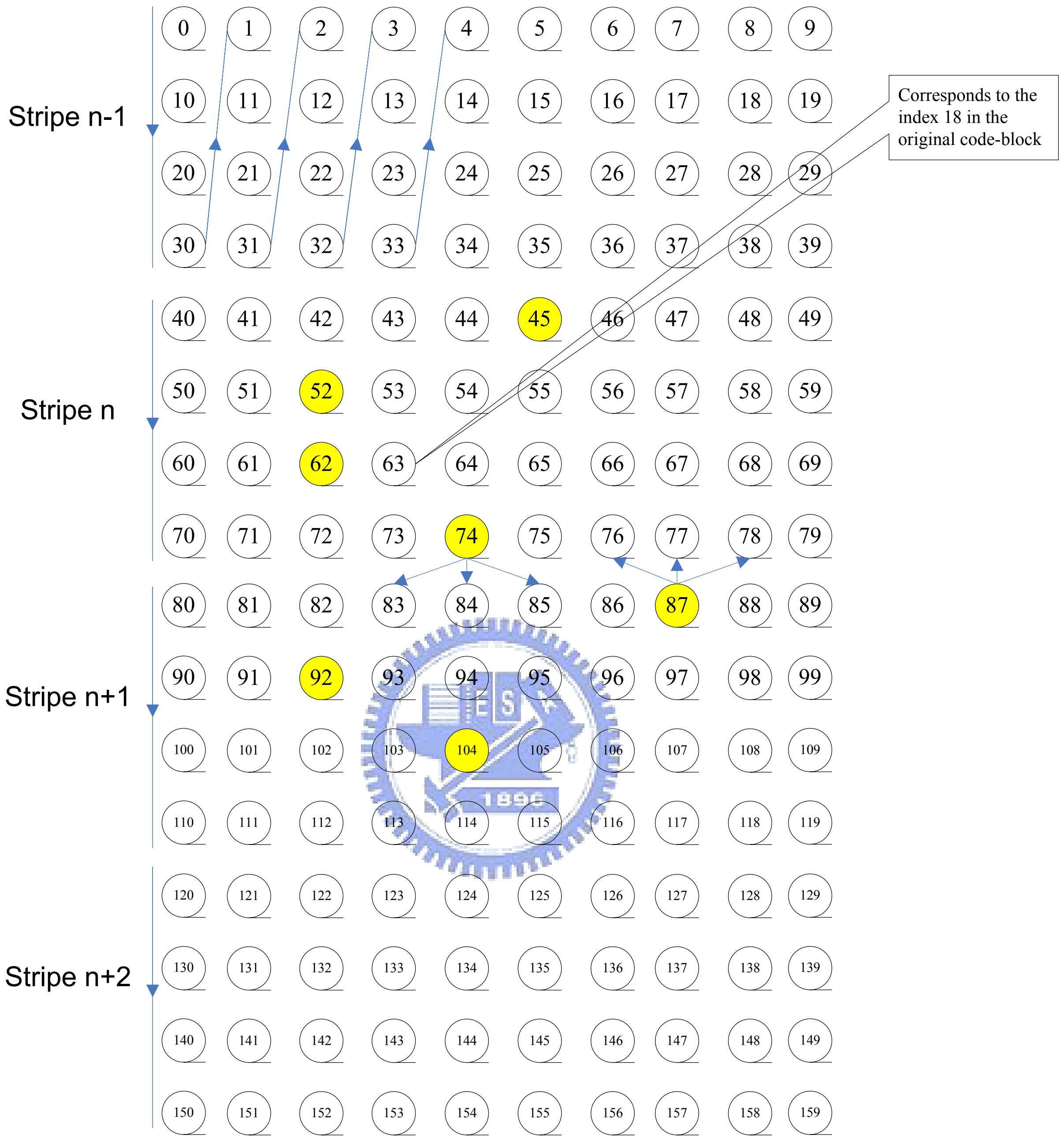


Figure 5-5 Restored flag-block with paddings

Then, the updating flag procedure needs to be modified. We describe the original updating flag procedure in the OpenJPEG reference software first. The flag-block in Figure 5-5 is transferred from the rearranged flag-block in Figure 5-4 and the shaded samples are significant. Typically, a significant sample affects its neighbor eight samples. The context orientations and the significant information are recorded in the flag-block. Each sample has its own flag to record the context orientations and the significant information. A new significant sample may lead to a change of eight flags (of its neighbors). In general, the updating flag procedure updates eight neighbor flags. The pseudo code is shown below.

Original flag updating procedure algorithm: [OpenJPEG ver.1.0]

```
*fp = current flag pointer
s = current significance
*np = fp - (code-block width + 2)
*sp = fp + (code-block width + 2)

np[-1] |= significant South-East
np[1] |= significant South-West
*np |= significant North
sp[-1] |= significant North-East
sp[1] |= significant North-West
*sp |= significant South
if (s = TRUE) then
   *np |= sign North
   *sp |= sign South
   fp[-1] |= sign East
   fp[1] |= sign West
endif
```

Typically, each sample in the strip affects the eight neighbor flags. The first sample in a stripe affects the same bit-plane flags in the previous stripe but these samples may not be coded depending on the visited information. The last sample in a stripe affects the sample bit-plane flags in the next stripe and the next stripe is not visited as shown in Figure 5-5. For example, the index ‘74’ flag in the ‘n’ stripe is significant. It affects three flags (83, 84, and 85) in the ‘n+1’ stripe. The sample with index ‘84’ in the ‘n+1’ stripe affects the three flags (76, 77, and 78) in the ‘n’ stripe. The middle two in the four continuous samples affect their corresponding eight neighbor flags. However, the rearranged flag-block has new set 1-D indices. We have to modify the updating flag procedure to update the correct neighbor flags as below:

Modified flag updating procedure algorithm:

```
hint = 0x3&(current coordinate x)
*fp = current flag pointer
s = current significance
*np = fp - (code-block width + 5)
*sp = fp + (code-block width + 5)

switch (hint)
    case 0x01:
    case 0x02:
        *(fp-5) |= significant South-East
        *(fp-3) |= significant North-East
        *(fp+3) |= significant South-West
        *(fp+5) |= significant North-West
        *(fp-4) |= significant East
        *(fp-1) |= significant South
        *(fp+1) |= significant North
        *(fp+4) |= significant West
        if (s is TRUE) then
            *(fp-4) |= sign East
            *(fp-1) |= sign South
            *(fp+1) |= sign North
            *(fp+4) |= sign West
        endif
    case 0x00:
        *(np-4) |= significant South-East
        *(np+4) |= significant South-West
        *(fp-3) |= significant North-East
        *(fp+5) |= significant North-West
        *np |= significant South
        *(fp-4) |= significant East
        *(fp+1) |= significant North
        *(fp+4) |= significant West
        if (s = TRUE) then
            *np |= sign South
            *(fp-1) |= sign East
            *(fp+1) |= sign North
            *(fp+4) |= sign West
        endif
    case 0x03
        *(sp-4) |= significant North-East
        *(sp+4) |= significant North-West
        *(fp-5) |= significant South-East
        *(fp+3) |= significant South-West
        *sp |= significant North
        *(fp-4) |= significant East
        *(fp-1) |= significant South
        *(fp+4) |= significant West
        if (s = TRUE) then
            *sp |= sign North
            *(fp-4) |= sign East
            *(fp-1) |= sign South
            *(fp+4) |= sign West
        endif
endswitch
```

In practice, the flag updating procedure is decoupled into three sub-procedures by the three cases and the total cycles of the flag updating procedure are reduced. We test the flag updating procedures on the C6416 simulator when the L2 cache is enabled. The results are shown in the Table 5-2. For these test images, the modified flag updating procedure takes about 62 % in calculation of the original one. Our proposed method does not increase processor cycles in the flag updating procedure.

	Goldhill	Barb	Lena	Baboon
Original updating flag procedure	28,994,661	28,365,064	27,499,705	30,648,603
Modified updating flag procedure	17,951,047	17,420,942	16,972,554	19,142,144
Ratio	62%	61%	62%	62%

Table 5-2 Comparison between original and modified updating flag procedures

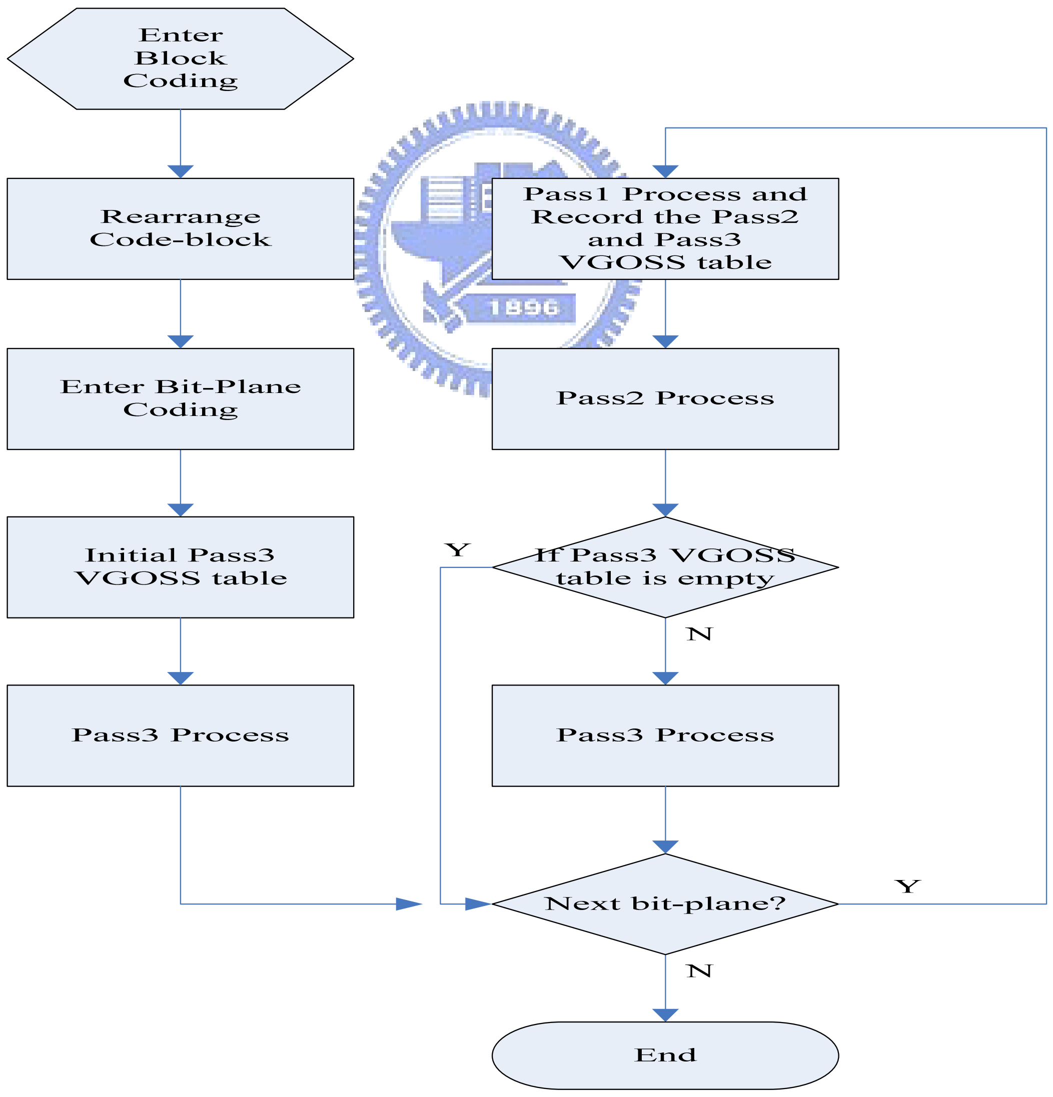


Figure 5-6 Flowchart of the bit-plane coding

After rearranging a code-block, the bit-plane coding is performed. The flowchart of the bit-plane coding is shown in Figure 5-6. The rearrangement of the code-block is first executed. The flag-block is also rearranged and ready. At the beginning, the Pass3 VGOSS table is initialized and the Pass3 process is executed in the first bit-plane coding. There are some significant samples in the Pass3 process because the bit-plane coding is started from the first nonzero most-significant-bit plane. Then, the Pass1 process is executed, and each sample can be classified according to its state which is recorded in the flag-block. The Pass2 and the Pass3 VGOSS tables are completed in this process. Then, all flags of the samples are checked in the Pass1 process and some insignificant samples which have significant neighbors are coded. The following step is the Pass2 process and then the Pass3 process. If the Pass3 VGOSS table is empty, the Pass3 process is skipped. The next bit-plane coding is executed until all bit-planes are done.

The flowcharts of the three pass processes are shown in Figure 5-7, Figure 5-8, and Figure 5-9. There are two VGOSS tables for the Pass2 and the Pass3 processes, respectively. In Figure 5-7, the Pass3 process is described. At the beginning, all samples are in the insignificant state and the Pass3 process is performed at the most significant bit-plan. The VGOSS table of Pass3 process is set to all NBC samples. It means that each offset is 1 in the Pass3 VGOSS table and all samples must be scanned in the first Pass3 process. The offset means the distance between the current index and the next index. After the next index is obtained, the run-length condition is checked each time. The run-length coding is executed if the four continuous samples are insignificant and do not have any significant neighbors. If the run-length coding is performed, the VGOSS table skips the number of the run-length in the following offsets. These offsets must equal one because they represent continuous samples that are coded in the run-length coding. Otherwise, the zero coding is executed. The sign coding is also executed when the sample becomes significant in the current bit-plane. Then, the next offset is read to calculate the next index. The first Pass3 process is completed until all offsets are used in the Pass3 VGOSS table. Otherwise, if the Pass3 process is not first-time running, the Pass3 process uses the Pass3 VGOSS table that is updated in the Pass1 process. If the VGOSS table is empty, it means that all samples are coded in the Pass1 and the Pass2 processes in the current bit-plane. The Pass3 coding is skipped in the current bit-plane.

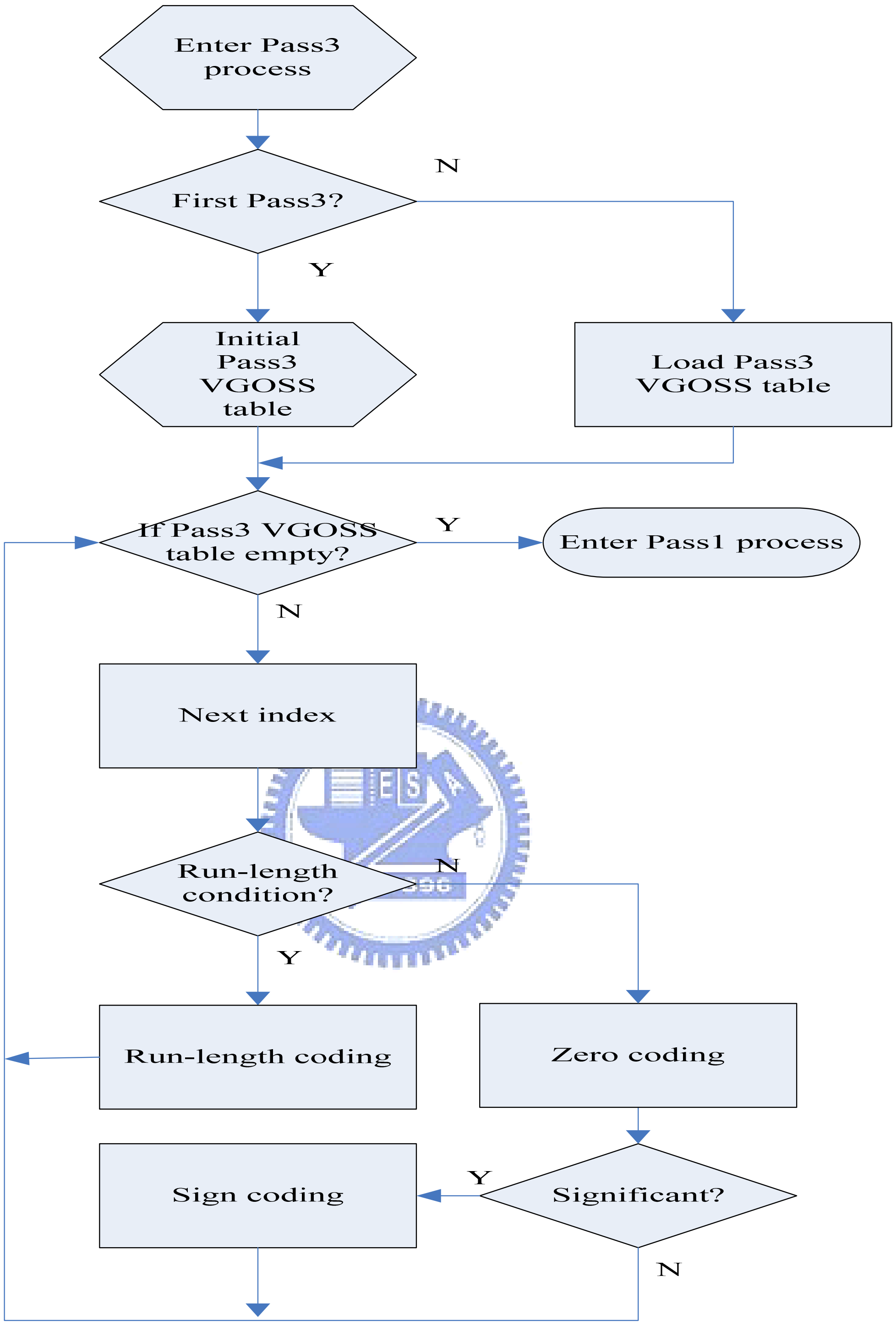


Figure 5-7 Flowchart of the Pass3 process

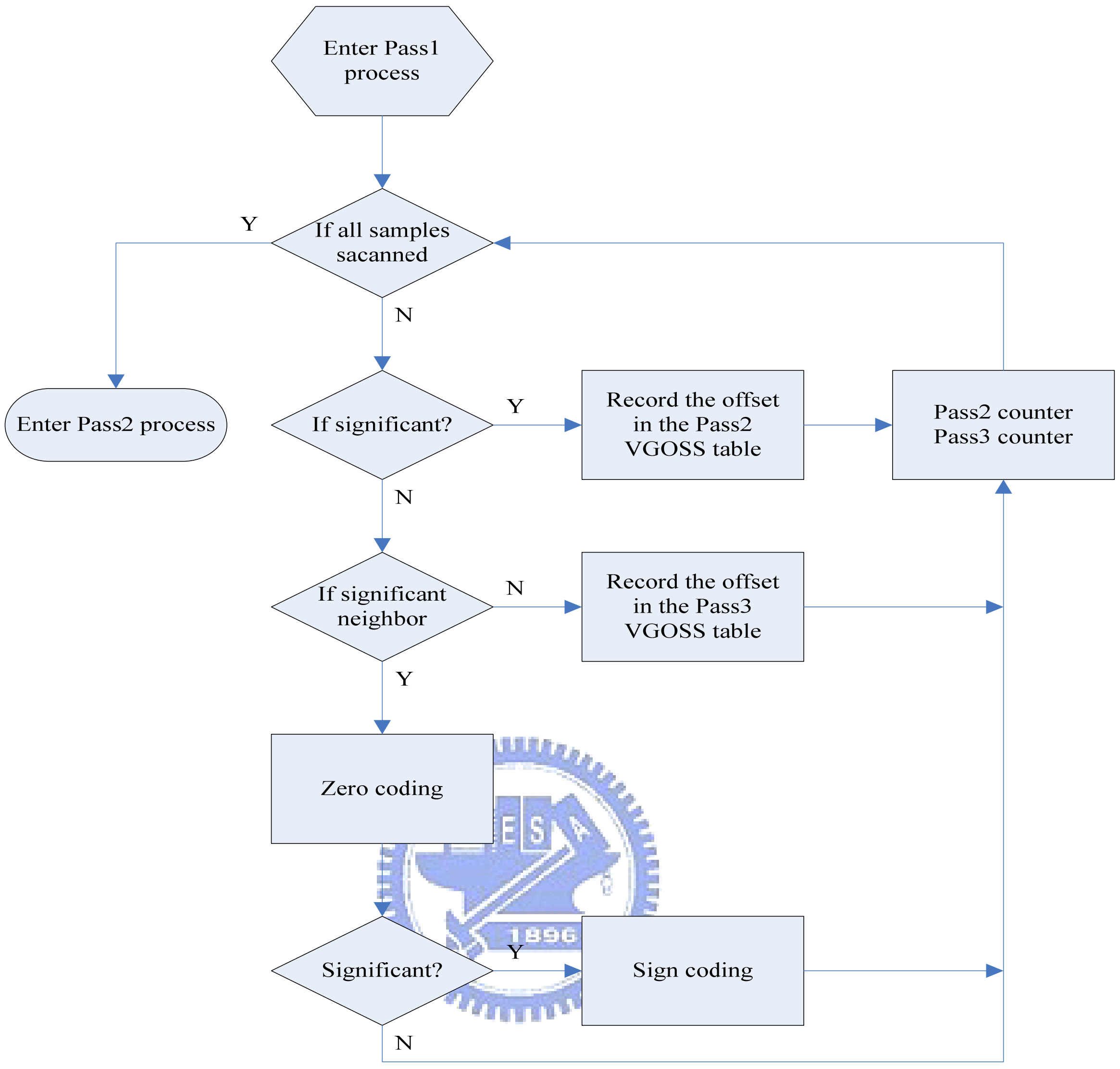


Figure 5-8 Flowchart of the Pass1 process

After the Pass3 is completed in the current bit-plane, the next bit-plane is coded. The Pass1 process checks every flag and encodes the NBC samples in the current bit-plane. The Pass1 process is described in Figure 5-8. All the samples are distinguished from two branches. The samples that should be coded in the Pass2 process are significant. If the current sample is in the significant state, the offset is recorded in the Pass2 VGOSS table. The Pass2 VGOSS counter is reset to zero and then updated for the next counting. But the Pass3 counter is updated only. If the sample is insignificant and does not have any significant neighbors, the offset is recorded in the Pass3 VGOSS table. The Pass3 counter is reset to zero and then updated for the next counting. But the Pass2 counter is updated only. Otherwise, the sample belongs to the Pass1 coding. Then, the zero coding is executed. If the sample is significant in the current bit-plane, the sign coding is also executed. Also, the Pass2 and Pass3 counters are updated, if the sample belongs to the Pass1 coding. When all samples are scanned, the Pass1 process is completed in the current bit-plane. These VGOSS tables are completed for the Pass2 and the Pass3 processes in the current bit-plane.

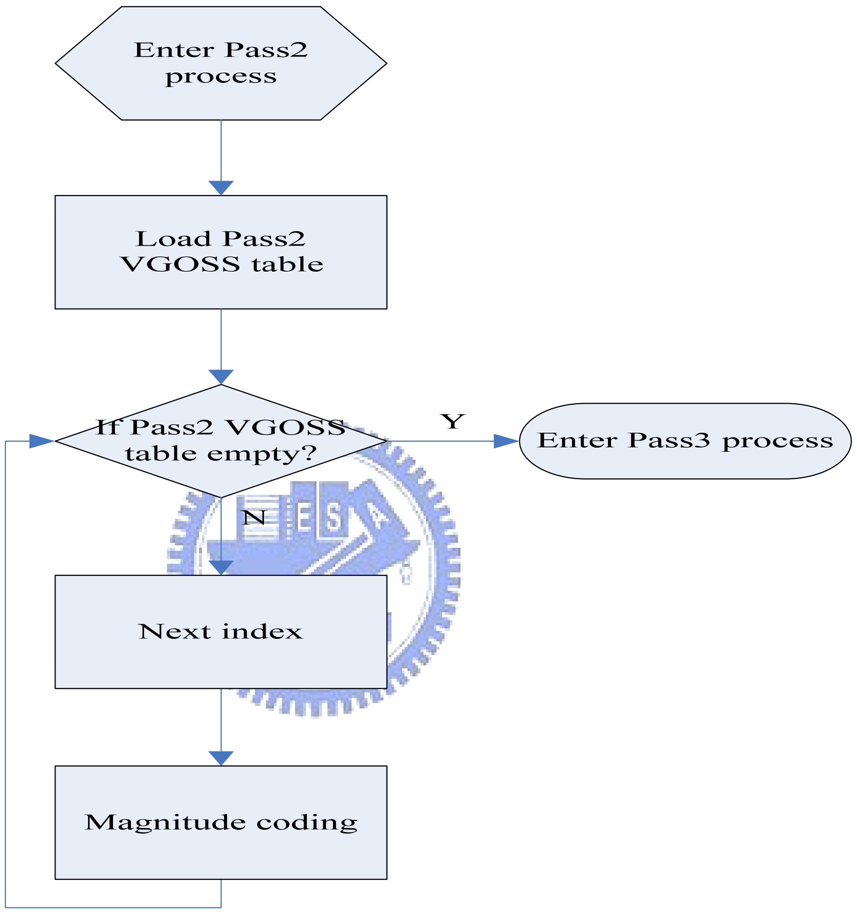


Figure 5-9 Flowchart of the Pass2 process

The Pass2 process is executed after the Pass1 process and the Pass2 table is updated in the Pass1 process in the current bit-plane. The flowchart of the Pass2 process is shown in Figure 5-9. If all the offset are used in the Pass2 VGOSS, the Pass2 process is completed. The following step is executing the Pass3 process and the pass processes is done in the current bit-plane coding. Afterward the Pass1, Pass2, and the Pass3 processes are executed until all the bit-planes are coded.

In practice, the modified flag updating procedure is split into three procedures. Under three different conditions, three updating flag procedure are executed separately and also the bit-plane scanning is modified. According to the VGOSS method, the pseudo code is listed below.

Appendix A: Pseudo code of the VGOSS method

Algorithm 1 Significance pass algorithm

```
1:  for (k=0; k<rearranged code-block height; k++) do
2:    for (i=0; i<rearranged code-block width; i+4) do
3:      fp0 (flag pointer) = (k+1)*(rearranged code-block width) + i + 4
4:      if fp0 is insignificant then
5:        if fp0 has significant neighbor then
6:          code significance of the sample
7:          if sample is a new significance then
8:            code sign of the sample
9:            updating flag procedure case 0x00
10:         else
11:           count the Pass2 and Pass3 offsets
12:         endif
13:       else
14:         Record the Pass3 offsets in the Pass3 VGOSS table
15:         count the Pass2 and Pass3 offsets
16:       endif
17:     else
18:         Record the Pass2 offsets in the Pass2 VGOSS table
19:         count the Pass2 and Pass3 offsets
20:     endif
21:     fp1 (flag pointer) = (k+1)*(rearranged code-block width) + i + 5
22:      if fp1 is insignificant then
23:        if fp1 has significant neighbor then
24:         code significance of the sample
```

```
        if sample is a new significance then
            code sign of the sample
            updating flag procedure case 0x01
        else
            count the Pass2 and Pass3 offsets
        endif
    else
        Record the Pass3 offsets in the Pass3 VGOSS table
        count the Pass2 and Pass3 offsets
    endif
else
        Record the Pass2 offsets in the Pass2 VGOSS table
        count the Pass2 and Pass3 offsets
endif
fp2 (current flag pointer) = (k+1)*(rearranged code-block width) + i + 6
if fp2 is insignificant then
    if fp2 has significant neighbor then
        code significance of the sample
        if sample is a new significance then
            code sign of the sample
            updating flag procedure case 0x01
        else
            count the Pass2 and Pass3 offsets
        endif
    else
        Record the Pass3 offsets in the Pass3 VGOSS table
        count the Pass2 and Pass3 offsets
    endif
else
        Record the Pass2 offsets in the Pass2 VGOSS table
        count the Pass2 and Pass3 offsets
endif
```

```
57:         fp3 (current flag pointer) = (k+1)*(rearranged code-block width) + i + 7
58:         if fp3 is insignificant then
59:             if fp3 has significant neighbor then
60:                 code significance of the sample
61:                 if sample is a new significance then
62:                     code sign of the sample
63:                     updating flag procedure case 0x01
64:                 else
65:                     count the Pass2 and Pass3 offsets
66:                 endif
67:             else
68:                 Record the Pass3 offsets in the Pass3 VGOSS table
69:                 count the Pass2 and Pass3 offsets
70:             endif
71:         else
72:                 Record the Pass2 offsets in the Pass2 VGOSS table
73:                 count the Pass2 and Pass3 offsets
74:         endif
75:     endfor
76: endfor
```

Algorithm 2 Refinement pass algorithm

```
1:  while Pass2 VGOSS table is not empty do
2:      code magnitude of the sample
3:  endwhile
```

Algorithm 3 Cleanup pass algorithm

```
1:  while Pass3 VGOSS table is not empty do
2:      if ((current address & 0x03)==0 and run-length is not zero) then
/*first sample in a stripe*/
```

```
3:          **if** (run-length = 4) **then**
4:             skip the following 4 offset in the Pass3 VGOSS table
5:             **continue**
6:          **else**
7:             code significant sample
8:             skip the following number of run-length offset in the Pass3 VGOSS table
9:             **continue**
10:         **endif**
11:      **else**
12:         run-length = 0
13:      **endif**
14:      cleanup coding
15:      **if** sample significant **then**
16:         code sign of sample
17:         updating flag procedure case (0x03&current data point)
18:      **endif**
19:   **endwhile**
```

5.1.2 Modified VGOSS method

According to the PP method, the absolute coordinates are recorded in the prediction table. It seems more efficient if the missing samples and the sorting problems can be solved. We modify the VGOSS method to record the absolute address in the code-block. Basically, all the pass procedures are similar to the original VGOSS pass procedure. Only the Pass1 procedure is modified, and all pass processes use the absolute index in the VGOSS tables. The modified Pass1 procedure is described in Figure 5-10. The difference is that the VGOSS method records the offset and the modified VGOSS method records the absolute index. The counters are not necessary, and the Pass1 process seems to be more efficient. However, the experimental results show that the VGOSS method is slightly better than the modified VGOSS method. The experimental results will be shown in section 5.2 and we will compare these methods in the next section.

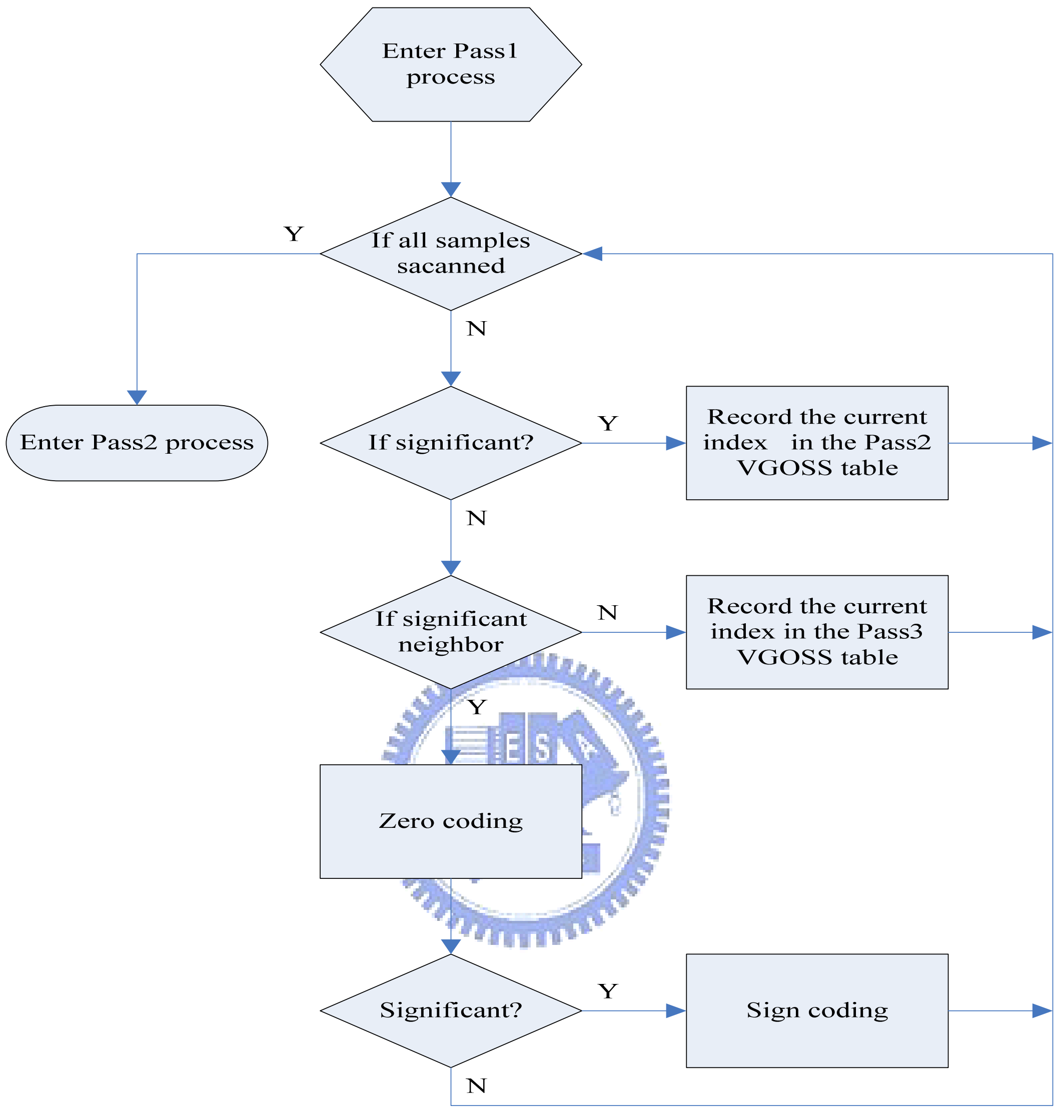


Figure 5-10 Flowchart of the Pass1 process

The VGOSS tables are now recording the address information instead of the address offsets. The flowchart of the modified VGOSS method is almost the same as that of the VGOSS method except that the absolute address is recorded. Although the flowchart is the same, there are some differences in the Pass1 process and the Pass3 process. In the traditional PP method, all tables are prepared in the previous bit-plane. All addresses which are to be encoded in a certain bit-plane are recorded in the previous bit-plane. In the VGOSS method, all the offsets are recorded when the Pass1 process is performed in the current bit-plane. This method avoids the missing sample problem and the comparator of three tables. Recording address is also efficient to accelerate the bit-plane coding and it records those samples that are need-to-be-coded in the Pass1 process.

Before coding a code-block, address rearrangement is performed in advance. Because the original address arrangement cannot be used in the new method easily and updating flag procedure is also complicated. The flag table records the state of the samples and each flag must associate with relative samples. The updating flag procedure should be modified as described in section 5.1.1 It seems that the Pass1 process simply records the NBC samples but there are some problems using this method. Because the code-block size in the higher decomposition level may be smaller than the specified code-block size. It affect the loop branches. It should be noticed in the Pass3 process. Typically, we think the performance of the modified VGOSS method is fast than the one of the VGOSS but the experimental results show that the VGOSS method is faster.

5.1.3 Advantages of the Proposed Methods

The methods we describe in the previous sections are efficient in accelerating the JPEG2000 algorithm. Each has its own advantages in reducing the cycles. In this sub-section, we will discuss the major advantages using our proposed methods.

Image	Encoded samples in Pass1 process	Encoded samples in Pass2 process	Encoded samples in Pass3 process
Goldhill	24 %	22 %	54 %
Barb	22 %	24 %	54 %
Lena	22 %	17 %	61 %
Baboon	26 %	36 %	38 %

Table 5-3 Percentage of encoding samples in each pass process

Each pass process encodes about one-fourth of samples in coding a block except for the Pass3 process. Table 5-3 shows that the Pass1 process encodes about 24% of total checked samples from the most significant bit-plane to the least significant bit-plane. The PP method uses the prediction table to record those NBC (need-to-be-coded) samples in the next bit-plane. Reducing to check the samples saves lots of processing cycles. Our proposed method saves the checking cycles in the Pass2 and Pass3 processes as well.

The VGOSS method is different from the PP method. The PP method predicts the NBC samples in the last bit-plane, and some missing samples are fixed by the continuous-five and boundary extension modes, which have been described in section 4.4.1 . The PP method has to predict the NBC samples in the Pass3 process or after the Pass3 process. Typically, the prediction tables are completed in the Pass3 process. If the prediction tables are completed after the Pass3 process, the extra N^2 checking cycles are required. In our approach, the VGOSS tables include all NBC samples. The VGOSS method records the NBC samples in the current bit-plane and it avoids the missing samples in the significant propagation as described in Figure 4-14. No samples miss and no sorting repair list or comparison index problems are produced in the VGOSS method. The checking loop of each pass process depends on the number of the offset in the VGOSS table. The number of the NBC samples is obtained before the Tier2 coding. We can calculate the distortion before coding the entire code-block. Besides, the PP method records the coordinates of the sample because the stripe scanning is not a sequential order. The VGOSS method only records the offsets, and most offsets are small numbers. The table size is smaller than the one of the PP method.

The CUPS is a known speed-improving method. We know that the scanning hierarchy for a code-block is ordered from the lower to the upper level, which is pixel, column, stripe, pass, and bit-plane. To skip from the upper levels represents to save more operation cycles. The CUPS method is to skip all samples which are not necessarily checked in the clean-up pass process. Usually, the CUPS method is applied to the Pass3 process with the PP method. When the CUPS method is applied, it skips the Pass3 process in the bit-plane coding. The prediction tables in the PP method are affected by the CUPS method. The N^2 checking cycles is still required to complete the prediction tables. However, the VGOSS method can skip the Pass3 process without extra N^2 cycles and the Pass1 and the Pass3 processes are the candidates to skip all samples in the checking process. If all samples become significant, they are coded in the Pass2 process in all following bit-planes. Thus, the Pass1 and the Pass3 process can be skipped entirely. In our proposed method, the skipping pass method is already adopted without additional effort. If the Pass1 or the Pass3 VGOSS table is empty, the corresponding pass process will be skipped, similar to the CUPS method.

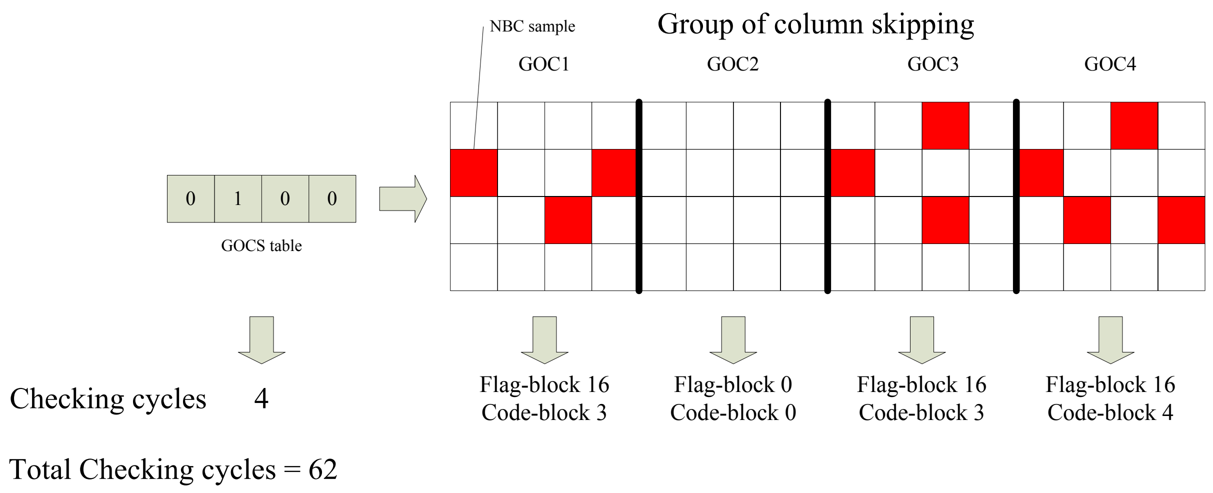


Figure 5-11 Checking cycles of the GOCS method

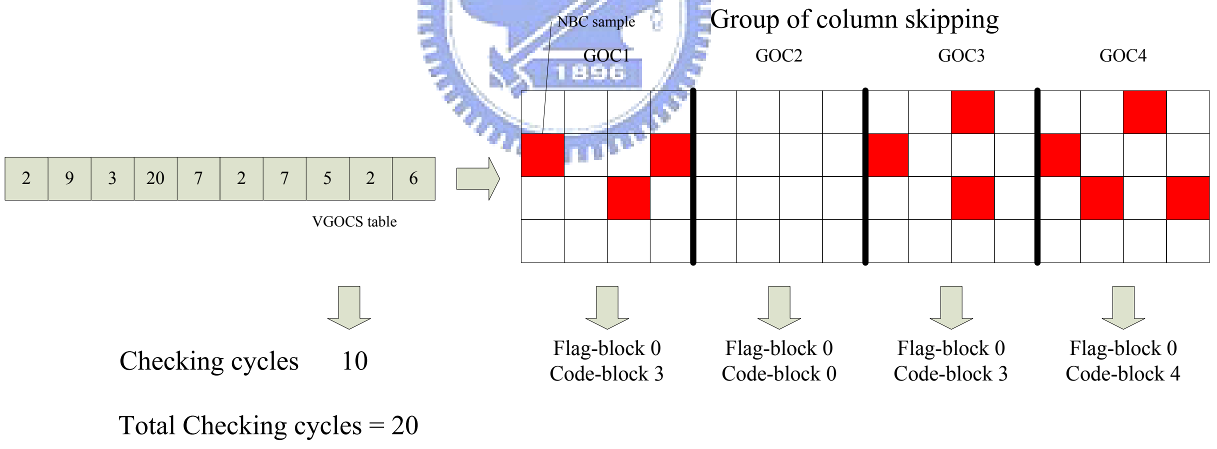


Figure 5-12 Checking cycles of the VGOSS method

The SS method and the GOSS methods described in section 4.4.2 are often used in accelerating the JPEG2000 encoder but it is not convenient to implement on a sequential processing platform. Based on the bit-plane coding statistics, most checking cycles are wasted in the three pass processes. The significant samples usually propagate on a bit-plane. The GOCS method makes use of this property but the most appropriate column size of a group varies for different test images. In the experimental results of using different column sizes on the C64xx simulator, we find that the GOSS method with a fixed group size is not efficient in skipping the un-coded samples. The variable column size is a better way to handle all different test images. The GOSS method results show that skipping a group of samples is still a good idea to accelerate the block coding. In practice, skipping a variable-size group of samples that are not necessarily coded is achieved already by our proposed method.

Figure 5-11 shows the total checking cycles when applying the GOCS method. The group size is set to 4 columns. The 96 samples are grouped into 4 groups called GOC1, GOC2, GOC3, and GOC4. The checking step includes checking the flag-block and the code-block. We assume that checking the GOCS table takes 4 cycles. In this example, the GOCS table skips GOC2 and uses 0 cycles. The GOC1, GOC3 and GOC4 are checked because there are NBC samples in each group. Each flag in the flag-block is checked in the GOC, and the state of the NBC sample is decided. If the sample is need-to-be-coded, its value recorded in the code-block is checked. If the value of the current sample is 1, the state of the current sample becomes significant. The sign coding is also executed. For example, the GOC1 takes 16 cycles in checking the state that is recorded in the flag-block. There are three NBC samples, and 3 cycles is taken in checking the sample values. The total checking cycles is 62 of the example in Figure 5-11.

In Figure 5-12, the VGOSS table takes 10 cycles to identify the NBC samples. The VGOSS method takes zero cycles in checking the state of each sample. The value of each NBC sample has to be checked. The total checking cycles in a code-block are same as that of the GOCSS method.

The SS method is also a fast process for the hardware implementation but it is not suitable for the sequential program. In our DSP system, the VLIW architecture provides the parallel executing equivalence to take the advantage of the SS method. To a certain extent, the compiler level can use its parallelism. The other way to improve utility of the DSP functional units is to modify the assembly code.

Finally, we compare the modified VGOSS method with the original one. The modified VGOSS method records the absolutely index only, so the counters are unnecessary. However, the counters are necessary when using the offset in the pass process. Consequently, the computation of using the absolute address is less than using the offset. Theoretically, the modified VGOSS method is more efficient than the VGOSS method. But the experimental results show that the original VGOSS method is slightly more efficient than the modified one. We will give the comparison results in section 5.2 .

5.1.4 Software Speed-up Techniques

The arithmetic coding algorithm contains sequential processing steps, nested conditional operations, and inner while loops. They decrease the efficiency of the software pipelined scheduling. The JPEG2000 binary arithmetic encoder is characterized by four functions, Code MPS, Code LPS, RENORME, and BYTEOUT. These functions are executed based on the context state of the arithmetic encoder, its interval width (A), and codeword value (C). The encoder must decide if a Most Probable Symbol (MPS) or Least Probable Symbol (LPS) is encoded, whether to renormalize (RENORME) the interval width and codeword, and determine if a compressed byte needs to be sent to the bit-stream (BYTEOUT).

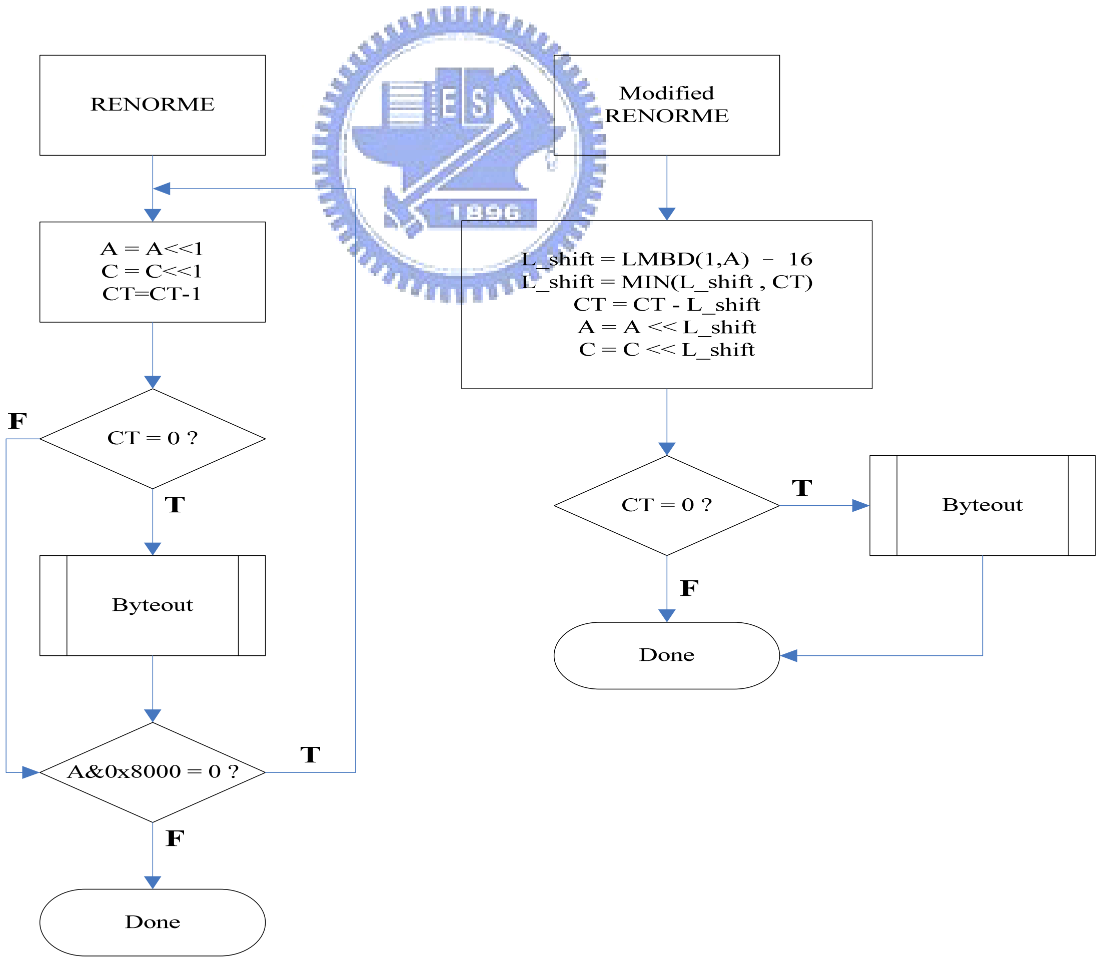


Figure 5-13 RENORME and modified procedure

In [28], several optimizing techniques for the JPEG2000 binary arithmetic encoder on a VLIW architecture are proposed. The first technique is decoupling the coefficient bit modeler from the arithmetic encoder. To rewrite the arithmetic encoder in an efficient loop form, the bit modeling and the arithmetic encoding processes are decoupled. The decoupling probably provides the better compiler scheduling. Thus, the compiled code may be more efficiently. The second technique is eliminating the loops in the RENORME function. An intrinsic function of the C64x is used to eliminate the loop condition as shown in Figure 5-13. We revise the modified RENORME function in [28]. The third technique is decoupling the BYTEOUT function from the MQ encoder. Due to the average 5% of the total encoding number is called by BYTEOUT function, the decoupling may have the additional benefit of making the encoding loop more efficient. Finally, we modified the MQ coder according to the concept that is described in [28]. We enable the L2 cache on the C6416 simulator but do not use the compiler-level optimization. The experimental results are obtained and shown in Table 5-4.

	Goldhill	Barb	Lena	Baboon
Original method	254,471,438	253,084,608	228,484,653	317,505,663
Modified method	222,464,585	220,992,114	200,095,451	276,482,317
Ratio	87%	87%	88%	87%

Table 5-4 Cycles of the MQ coder on the C6416 simulator

	Goldhill	Barb	Lena	Baboon
DWT encode	552,674,115	552,674,115	552,674,110	552,674,110
Speed-up	196,657,558	196,657,558	196,657,558	196,657,558
Ratio	36%	36%	36%	36%

Table 5-5 DWT module on C6416 simulator

In addition, a few well known techniques such as unrolling, packeted data processing and pragma instructions could be used for accelerating the program as described in section 3.4.3 .

But some techniques are not suitable for this program. For example, the packet date processing is suitable for the 8-bit or the 16-bit data type. However, the most data types in the JPEG2000 software are 32-bit data type. The pragma directives tell the compiler how to treat a certain function, object, or section of code. This method improves the performance slightly on the C6416 simulator. The L2 cache is an efficient way to reduce the accessing time of the external memory, and the pragma directives provide the SRAM to store the data that are often used. Although the L2 cache occupies a portion of the SRAM, the two methods are different in the produced missing cycles. The DATA_SECTION pragma decides the memory allocation without setting DSP BIOS. It is useful to define the use of memory. The MUST_ITERATE pragma instruction helps that the loop call executes a certain number of times. These speed-up techniques are useful for the DWT module. We apply the compiler-level optimization and the software speed-up methods. The reduction is about 64% as shown in Table 5-5. The modified procedure shown in Figure 5-13 also uses the LMBD intrinsic function. All of them may help the running speed on the DSP platform, and they are the program level optimization. The experimental results are presented in the next section.

5.2 Experimental Results

In this section, we are going to present the experimental results using the test images described in section 4.1 The notation "M1" means the scheme uses the SS and GOSS methods described in section 4.4.2 . "M2" represents the proposed VGOSS method described in section 5.1.1 and "M3" represents the modified VGOSS method described in section 5.1.2 . The "Ori" means the original program and the Tier1 module includes the Pass1 module, the Pass2 module, and the Pass3 module. The proposed M2+ and M3+ methods are accelerated with program level optimization described in section 3.4.3 and 5.1.4 . Otherwise, the compiler-level optimization uses the file level optimization and the non-level optimization. All the ratios in these tables are the cycles of the proposed method divided by the original cycles and are expressed in percentage.

On Table 5-6, we compare different methods using the C64xx simulator. The programs do not use compiler-level optimization. The Tier1 function takes the most cycles of the JPEG2000 encoder. We can see that our proposed methods have achieved about 30%

reduction. In other words, our proposed method has a better performance without enabling any hardware or software optimizations. The comparison of the M2 and M3 will be discussed later.

		Pass1	ratio	Pass2	ratio	Pass3	ratio	Tier1	ratio
Goldhill	Ori	305,253,023	N/A	230,409,367	N/A	226,859,815	N/A	846,100,895	N/A
	M1	282,191,607	92%	161,154,959	69%	153,603,555	67%	691,653,125	81%
	M2	293,537,043	96%	147,094,303	64%	140,270,753	62%	617,477,074	73%
	M3	268,073,463	87%	146,687,443	63%	154,029,099	67%	610,316,669	72%
Barb	Ori	298,583,742	N/A	242,601,594	N/A	229,189,368	N/A	854,930,539	N/A
	M1	271,438,557	91%	171,781,735	71%	155,859,497	68%	686,567,790	80%
	M2	285,719,070	96%	157,152,430	65%	142,649,768	62%	622,141,054	73%
	M3	260,057,725	87%	156,718,367	65%	156,544,656	68%	614,892,043	72%
Lena	Ori	290,392,605	N/A	199,107,794	N/A	221,486,542	N/A	793,191,469	N/A
	M1	266,220,518	92%	126,029,839	63%	160,976,617	73%	638,251,568	80%
	M2	278,136,726	96%	111,841,674	56%	148,145,016	67%	574,660,595	72%
	M3	253,699,774	87%	111,530,410	56%	162,578,666	73%	569,297,918	72%
Baboon	Ori	340,504,870	N/A	329,348,990	N/A	251,280,448	N/A	1,011,860,852	N/A
	M1	307,918,629	90%	260,281,541	79%	148,146,791	59%	810,277,071	80%
	M2	325,038,946	95%	242,780,578	74%	135,044,067	54%	739,661,816	73%
	M3	295,676,364	87%	242,114,405	74%	148,261,575	59%	727,801,158	72%

Table 5-6 Comparison using C64xx simulator without compiler-level optimization

Then, we use the compiler-level optimization to accelerate the encoder. The exact cycles are shown in Table 5-7 and we can see that the cycles are reduced to half of their counterpart in Table 5-6. This means the encoder has been accelerated about two times faster. The original encoder takes about 0.8 sec to encode a 512 by 512 image on the C64xx simulator. The running time is estimated based on 1GHz DSP processor. The encoding cycles are thus divided by the 10^9. The compiler level optimization can reduce the encoding time to about 0.4 sec. However, the C64xx simulator uses the flat memory system, and the real time on the C6416 emulator is related to the result of the C6416 simulator. On this table, our proposed M2 method has a reduction of up to 37% in the Tier1 module. The proposed M2 method seems to have a better performance than the proposed M3 method and the additional program level optimization helps the M2 method to achieve about 46% reduction of computation cycles as shown in the M2+ results. And, the M2+ results are still slightly better than the M3+ results.

		Pass1	ratio	Pass2	ratio	Pass3	ratio	Tier1	ratio
Goldhill	Ori	161,696,792	N/A	123,912,376	N/A	132,353,060	N/A	435,017,649	N/A
	M1	127,578,760	78%	81,792,398	66%	83,544,436	63%	310,108,162	71%
	M2	125,231,272	77%	65,696,264	53%	71,670,151	54%	278,049,520	63%
	M2+	111,479,250	69%	52,262,769	42%	64,940,416	49%	236,919,268	54%
	M3	134,596,715	83%	64,882,190	52%	72,759,624	54%	287,959,889	66%
	M3+	120,941,561	75%	51,041,835	41%	68,490,347	52%	246,919,263	57%
Barb	Ori	158,478,206	N/A	129,282,989	N/A	133,847,824	N/A	438,746,356	N/A
	M1	122,166,076	77%	87,196,372	67%	85,158,426	64%	311,697,932	71%
	M2	120,764,739	76%	70,177,352	54%	72,977,347	55%	279,411,207	64%
	M2+	107,855,946	68%	55,856,088	43%	66,049,340	49%	238,014,402	54%
	M3	130,671,579	82%	69,308,863	54%	74,122,298	55%	289,863,349	66%
	M3+	117,796,112	74%	54,551,358	42%	75,907,089	57%	248,421,913	57%
Lena	Ori	153,954,906	N/A	108,237,140	N/A	129,077,570	N/A	408,233,549	N/A
	M1	120,130,825	78%	64,744,920	60%	88,197,594	68%	290,075,454	71%
	M2	118,381,074	77%	49,993,868	46%	75,995,528	59%	259,792,581	64%
	M2+	105,516,600	69%	39,695,355	37%	68,746,351	53%	222,169,225	54%
	M3	126,991,911	82%	49,370,996	46%	77,238,351	60%	269,292,266	66%
	M3+	114,192,081	74%	38,759,155	36%	78,765,768	61%	231,705,425	57%
Baboon	Ori	179,696,714	N/A	171,827,724	N/A	147,598,193	N/A	516,672,503	N/A
	M1	138,079,206	77%	130,939,612	76%	80,407,186	54%	367,019,323	71%
	M2	136,630,164	76%	108,313,813	63%	68,628,442	46%	329,208,921	64%
	M2+	122,000,811	68%	86,310,345	50%	62,199,750	42%	278,849,859	54%
	M3	149,193,658	83%	106,981,058	62%	69,600,789	47%	341,680,709	66%
	M3+	134,442,629	75%	84,308,963	49%	78,911,706	53%	291,039,854	56%

Table 5-7 Comparison using C64xx simulator with file level optimization

In order to analyze the actual performance on the DSP emulator, we take profiling of the encoder on the C6416 simulator. First, we compare all the methods on the C6416 simulator without any compiler level optimization or program level optimization. The results are presented in Table 5-8. It shows that all methods have little improvement on the C6416 simulator. The Pass1 process checks all samples in the code-block but the other processes encode the NBC samples only. It seems the Pass1 process does not gain any acceleration as compared with Table 5-6.

		Pass1	ratio	Pass2	ratio	Pass3	ratio	Tier1	ratio
Goldhill	Ori	3,017,620,410	N/A	2,042,494,971	N/A	1,878,381,119	N/A	7,509,481,733	N/A
	M1	3,025,667,367	100%	1,741,809,460	85%	1,421,319,656	75%	6,770,907,062	90%
	M2	3,136,395,302	103%	1,705,536,220	83%	1,401,228,658	74%	6,559,372,183	87%
	M3	3,109,291,411	103%	1,705,523,470	83%	1,413,508,168	75%	6,549,455,756	87%
Barb	Ori	2,896,173,480	N/A	2,158,293,928	N/A	1,893,826,534	N/A	7,524,305,067	N/A
	M1	2,878,730,141	99%	1,876,066,087	87%	1,452,988,453	77%	6,786,729,765	90%
	M2	2,997,467,962	103%	1,822,898,565	84%	1,418,383,687	75%	6,555,043,830	87%
	M3	2,970,457,703	103%	1,822,891,700	84%	1,430,763,867	76%	6,545,326,635	87%
Lena	Ori	2,836,766,126	N/A	1,664,108,463	N/A	1,858,239,288	N/A	6,922,931,525	N/A
	M1	2,839,909,398	100%	1,353,038,108	81%	1,494,062,471	80%	6,253,649,422	90%
	M2	2,954,013,613	104%	1,295,049,729	78%	1,461,410,941	79%	6,026,615,579	87%
	M3	2,928,402,594	103%	1,295,042,543	78%	1,474,297,106	79%	6,018,804,632	87%
Baboon	Ori	3,307,931,711	N/A	3,113,296,427	N/A	2,018,577,407	N/A	9,047,999,627	N/A
	M1	3,264,497,611	99%	2,875,062,110	92%	1,401,263,282	69%	8,152,221,728	90%
	M2	3,395,751,005	103%	2,819,811,057	91%	1,365,494,753	68%	7,897,700,868	87%
	M3	3,363,909,744	102%	2,819,796,302	91%	1,377,280,810	68%	7,882,548,295	87%

Table 5-8 Comparison using C6416 simulator without compiler-level optimization

Then we use the file level optimization which has been described in section 3.4.2 and the results are shown in Table 5-9. We can see that the cycle of the Tie1 has about 30% reduction as compared to Table 5-8. The encoder still needs 5,896,744,104 (including all modules) cycles to encode a Goldhill image. All the accelerating methods are encumbered by the memory system as discussed in section 4.3.1 . Until now, the memory system dominates the performance on the C6416 simulator. Although we have a great performance on the C64xx simulator as shown in Table 5-7, the final judgments on the performance is on the real time system, i.e., the C6416 emulator. Because the C6416 emulator cannot profile the cycle information of the encoder on the board, we assume that the experimental results of the C6416 simulator can approximately represent the results of the C6416 emulator.

		Pass1	ratio	Pass2	ratio	Pass3	ratio	Tier1	ratio
Goldhill	Ori	2,112,041,634	N/A	1,378,876,280	N/A	1,247,956,850	N/A	5,249,595,284	N/A
	M1	1,909,390,616	90%	1,170,834,214	84%	1,017,962,461	81%	4,609,052,664	87%
	M2	2,117,760,455	100%	1,219,529,873	88%	1,008,254,857	80%	4,613,282,518	87%
	M3	2,107,525,133	99%	1,220,058,177	88%	999,638,429	80%	4,595,193,586	87%
Barb	Ori	2,039,284,465	N/A	1,452,630,437	N/A	1,262,285,138	N/A	5,269,580,191	N/A
	M1	1,823,491,911	89%	1,265,470,505	87%	1,039,524,522	82%	4,643,920,352	88%
	M2	2,024,787,896	99%	1,301,723,052	90%	1,023,242,623	81%	4,617,566,683	88%
	M3	2,016,239,162	99%	1,302,308,385	90%	1,014,583,518	80%	4,601,177,484	87%
Lena	Ori	1,988,997,365	N/A	1,138,002,909	N/A	1,247,101,515	N/A	4,878,183,463	N/A
	M1	1,788,189,759	90%	922,499,516	81%	1,069,658,356	86%	4,284,479,896	88%
	M2	1,985,936,734	100%	929,742,870	82%	1,054,472,108	85%	4,237,821,331	87%
	M3	1,976,123,575	99%	930,163,782	82%	1,045,640,767	84%	4,219,831,865	87%
Baboon	Ori	2,319,243,443	N/A	2,064,641,254	N/A	1,324,106,786	N/A	6,253,178,570	N/A
	M1	2,073,048,478	89%	1,925,065,285	93%	996,385,327	75%	5,539,745,533	89%
	M2	2,297,572,925	99%	2,005,493,307	97%	979,527,354	74%	5,550,726,038	89%
	M3	2,289,349,786	99%	2,006,364,071	97%	970,994,914	73%	5,535,075,645	89%

Table 5-9 Comparison using C6416 simulator with file level optimization

We know that enabling the L2 cache system can improve the total encoding time on the C6416 emulator. We compare Table 5-8 with Table 5-10. The encoding time of the encoder using the L2 cache system is eight times faster than the former one. As for these speed-up methods, the M1 method could just have about 20 % reduction of the computation cycles in the Tier1 module and our proposed methods have better performance at about 30 % reduction in the same module.

		Pass1	ratio	Pass2	ratio	Pass3	ratio	Tier1	ratio
Goldhill	Ori	308,284,632	N/A	251,097,478	N/A	232,841,458	N/A	878,528,789	N/A
	M1	283,514,595	91%	164,102,018	65%	170,341,982	73%	715,481,512	81%
	M2	310,759,475	100%	150,799,659	60%	142,705,474	61%	643,385,635	73%
	M3	286,373,249	92%	151,103,318	60%	156,405,272	67%	637,677,407	72%
Barb	Ori	301,567,183	N/A	262,931,785	N/A	235,251,179	N/A	887,081,291	N/A
	M1	272,633,925	90%	174,226,431	66%	172,958,920	74%	710,121,902	80%
	M2	301,927,080	100%	161,098,804	61%	145,148,671	62%	647,343,159	73%
	M3	277,311,304	92%	161,388,576	61%	158,987,469	68%	641,531,078	72%
Lena	Ori	293,270,703	N/A	213,746,035	N/A	227,449,758	N/A	819,425,979	N/A
	M1	267,314,454	91%	127,929,966	60%	178,827,591	79%	661,884,222	81%
	M2	293,726,038	100%	114,665,949	54%	150,681,664	66%	598,149,363	73%
	M3	270,410,167	92%	114,862,234	54%	165,059,091	73%	594,083,833	72%
Baboon	Ori	344,015,246	N/A	360,498,318	N/A	257,274,679	N/A	1,055,301,462	N/A
	M1	309,308,487	90%	263,822,006	73%	164,094,852	64%	834,062,055	79%
	M2	344,603,091	100%	248,835,280	69%	137,393,177	53%	770,203,559	73%
	M3	316,054,573	92%	249,336,149	69%	150,554,300	59%	759,984,175	72%

Table 5-10 Comparison by C6416 simulator using L2 cache

without compiler-level optimization

At the end, we enable the file level optimization and all results are shown in Table 5-11. Our proposed methods, M2 and M3, have a reduction of up to 35 % of computation cycles and, furthermore, the M2+ and M3+ use the program level optimization. It helps our proposed methods (M2+ and M3+) to reduce up to 45 % of computation cycles in the Tier1 module. These results now can achieve approximately those results of Table 5-7, which are generated by the C64xx simulator (using the flat memory system).

		Pass1	ratio	Pass2	ratio	Pass3	ratio	Tier1	ratio
Goldhill	Ori	163,080,554	N/A	124,821,177	N/A	133,472,282	N/A	441,174,161	N/A
	M1	128,930,478	79%	83,076,184	66%	101,499,569	76%	333,581,675	75%
	M2	128,567,039	78%	67,302,409	53%	73,328,088	54%	287,411,966	65%
	M2+	112,848,128	69%	53,775,561	43%	66,497,047	49%	243,772,802	55%
	M3	135,518,273	83%	66,348,752	53%	77,189,981	57%	297,154,656	67%
	M3+	122,246,007	75%	52,620,095	42%	67,810,545	51%	253,842,164	58%
Barb	Ori	159,898,842	N/A	130,222,929	N/A	135,022,022	N/A	445,058,085	N/A
	M1	125,607,586	79%	88,384,942	68%	100,587,687	74%	334,635,666	75%
	M2	123,971,510	78%	71,805,581	55%	74,693,375	55%	288,726,161	65%
	M2+	109,214,696	68%	57,459,450	44%	67,696,874	50%	245,428,000	55%
	M3	131,593,386	82%	70,838,058	54%	78,625,116	58%	299,196,088	67%
	M3+	119,108,218	74%	56,192,138	43%	69,091,530	51%	255,512,543	57%
Lena	Ori	155,250,384	N/A	109,054,998	N/A	130,199,119	N/A	414,230,692	N/A
	M1	123,411,310	79%	65,712,808	60%	104,118,479	80%	313,098,474	76%
	M2	121,457,646	78%	51,269,739	47%	77,789,117	60%	268,694,555	65%
	M2+	106,797,331	69%	40,955,168	38%	70,440,728	54%	229,200,081	55%
	M3	127,855,746	82%	50,567,223	46%	81,937,980	63%	278,421,068	67%
	M3+	115,431,429	74%	40,045,260	37%	71,836,295	55%	238,400,634	58%
Baboon	Ori	181,379,029	N/A	172,997,973	N/A	148,789,097	N/A	523,492,124	N/A
	M1	141,906,202	78%	132,573,813	77%	95,167,853	64%	390,208,533	75%
	M2	140,240,468	77%	110,727,881	64%	70,188,995	47%	339,579,061	65%
	M2+	123,588,182	68%	88,581,223	51%	63,678,663	43%	287,012,673	55%
	M3	150,294,582	83%	109,189,994	63%	73,777,013	50%	351,565,266	67%
	M3+	135,935,289	75%	86,669,837	50%	64,985,784	44%	298,879,302	57%

Table 5-11 Comparison using C6416 simulator with L2 cache and file level optimization

Comparing the proposed M2 and M3 methods, the M2 method records the offset but it has a better performance than the M3 method on the DSP simulators or emulator. In general, the M3 method records the address without counting the offset which is adopted in the M2 method and thus it should have better performance than the M2 method. However, all results show that the M2 method is more efficient than the M3 method. This is because the computation cycles of the M3 method in the Pass1 process are tinier than that of the M2 method. According to the assembly code, the M2 method use more counters to compute the offset, but those counters could lead to better parallelism instead.

Now, we summarize the best performance as shown in Table 5-12 and Table 5-13. When the bottleneck of the memory system is ignored, the results (Ori+), which are profiled using the file level optimization on the C64xx simulator, can achieve 1.9 times faster than the original one (Ori) as shown in Table 5-12. Furthermore, our best solution (M2+) can accelerate up to 3.6 times faster than the original one. On the C6416 simulator, the results (Ori+) are measured with the L2 cache and file level optimization and the proposed method (M2+) can accelerate the Tier1 module nearly 2 times faster than (Ori) as the results of the C6416 simulator.

		Pass1	Mul.	Pass2	Mul.	Pass3	Mul.	Tier1	Mul.
Goldhill	Ori	305,253,023	N/A	230,409,367	N/A	226,859,815	N/A	846,100,895	N/A
	Ori+	161,696,792	1.9x	123,912,376	1.9x	132,353,060	1.7x	435,017,649	1.9x
	M2+	111,479,250	2.7x	52,262,769	4.4x	64,940,416	3.5x	236,919,268	3.6x
Barb	Ori	298,583,742	N/A	242,601,594	N/A	229,189,368	N/A	854,930,539	N/A
	Ori+	158,478,206	1.9x	129,282,989	1.9x	133,847,824	1.7x	438,746,356	1.9x
	M2+	107,855,946	2.8x	55,856,088	4.3x	66,049,340	3.5x	238,014,402	3.6x
Lena	Ori	290,392,605	N/A	199,107,794	N/A	221,486,542	N/A	793,191,469	N/A
	Ori+	153,954,906	1.9x	108,237,140	1.8x	129,077,570	1.7x	408,233,549	1.9x
	M2+	105,516,600	2.8x	39,695,355	5.0x	68,746,351	3.2x	222,169,225	3.6x
Baboon	Ori	340,504,870	N/A	329,348,990	N/A	251,280,448	N/A	1,011,860,852	N/A
	Ori+	179,696,714	1.9x	171,827,724	1.9x	147,598,193	1.7x	516,672,503	2.0x
	M2+	122,000,811	2.8x	86,310,345	3.8x	62,199,750	4.0x	278,849,859	3.6x

Table 5-12 Comparison using C64xx simulator (Best solution) with file level optimization

		Pass1	Mul.	Pass2	Mul.	Pass3	Mul.	Tier1	Mul.
Goldhill	Ori	3,017,620,410	N/A	2,042,494,971	N/A	1,878,381,119	N/A	7,509,481,733	N/A
	Ori+	163,080,554	19x	124,821,177	16x	133,472,282	14x	441,174,161	17x
	M2+	112,848,128	27x	53,775,561	38x	66,497,047	28x	243,772,802	31x
Barb	Ori	2,896,173,480	N/A	2,158,293,928	N/A	1,893,826,534	N/A	7,524,305,067	N/A
	Ori+	159,898,842	18x	130,222,929	17x	135,022,022	14x	445,058,085	17x
	M2+	109,214,696	27x	57,459,450	38x	67,696,874	28x	245,428,000	31x
Lena	Ori	2,836,766,126	N/A	1,664,108,463	N/A	1,858,239,288	N/A	6,922,931,525	N/A
	Ori+	155,250,384	18x	109,054,998	15x	130,199,119	14x	414,230,692	17x
	M2+	106,797,331	27x	40,955,168	41x	70,440,728	26x	229,200,081	30x
Baboon	Ori	3,307,931,711	N/A	3,113,296,427	N/A	2,018,577,407	N/A	9,047,999,627	N/A
	Ori+	181,379,029	18x	172,997,973	18x	148,789,097	14x	523,492,124	17x
	M2+	123,588,182	27x	88,581,223	35x	63,678,663	32x	287,012,673	32x

Table 5-13 Comparison using C6416 simulator (Best solution) with file level optimization

Afterward, we record the real time execution on the C6416 emulator, i.e. the hardware platform. We use the timer on the DSP platform to count the executing time. We compare the 'ori', 'M1', 'M2+', and 'M3+' and all results are shown in Table 5-14. The 'M2+' result can achieve about 45% reduction of the computation time. It is similar to the result of the C6416 simulator. Comparing these reduction ratios in Table 5-11 and Table 5-14, all results are consistent. The simulation results on the C64xx simulator also achieve approximately those results on the hardware platform or the C6416 simulator.

Finally, we compare the difference in executing time between the C6416 simulator and the C6416 emulator (i.e. the hardware platform) shown in Table 5-15. We convert the cycles of the C6416 simulator to seconds by dividing 10^9 (1GHz DSP). The results show that the original executing time on the C6416 emulator is slower than the one on the C6416 simulator by about 11~14 %. When the L2 cache is adopted, the results show that the executing time on the hardware platform (C6416 emulator) is similar to that on the C6416 simulator. In summary, our proposed method can achieve an better performance and can implement on the DSP platform in an efficient and simple manner. The experimental results on different simulators or emulator show consistent results. It means that our proposed method can achieve about half reduction on the real system.

Tier1		Non compiler-level optimization (sec)	Ratio	L2 cache without compiler-level optimization (sec)	Ratio	L2 cache with file level optimization (sec)	Ratio
Goldhill	Ori	8.741983	N/A	0.898467	N/A	0.447415	N/A
	M1	7.666878	88%	0.717021	80%	0.346838	78%
	M2+	7.646112	87%	0.579569	65%	0.250016	56%
	M3+	7.643923	87%	0.580308	65%	0.259049	58%
Barb	Ori	8.752465	N/A	0.904697	N/A	0.451394	N/A
	M1	7.654667	87%	0.719655	80%	0.347562	77%
	M2+	7.711458	88%	0.584394	65%	0.251254	56%
	M3+	7.701879	88%	0.584885	65%	0.260636	58%
Lena	Ori	7.786148	N/A	0.835078	N/A	0.420186	N/A
	M1	7.052966	91%	0.670789	80%	0.325005	77%
	M2+	7.067648	91%	0.54088	65%	0.23467	56%
	M3+	7.056214	91%	0.542619	65%	0.2434	58%
Baboon	Ori	10.2021	N/A	1.077518	N/A	0.53121	N/A
	M1	9.19398	90%	0.84524	78%	0.405664	76%
	M2+	9.306895	91%	0.69134	64%	0.294029	55%
	M3+	9.328353	91%	0.689299	64%	0.304899	57%

Table 5-14 Comparison of the executing time on the C6416 emulator

		C6416 simulator (cycles)	Conversion (sec)	C6416 emulator (sec)	Ratio (simulator/emulator)
Goldhill	Ori	7,509,481,733	7.509482	8.741983	86%
	Ori+	441,174,161	0.441174	0.447415	99%
	M2+	243,772,802	0.243773	0.250016	98%
Barb	Ori	7,524,305,067	7.524305	8.752465	86%
	Ori+	445,058,085	0.445058	0.451394	99%
	M2+	245,428,000	0.245428	0.251254	98%
Lena	Ori	6,922,931,525	6.922932	7.786148	89%
	Ori+	414,230,692	0.414231	0.420186	99%
	M2+	229,200,081	0.2292	0.23467	98%
Baboon	Ori	9,047,999,627	9.048	10.2021	89%
	Ori+	523,492,124	0.523492	0.53121	99%
	M2+	287,012,673	0.287013	0.294029	98%

Table 5-15 Comparison between the C6416 simulator and the C6416 emulator

(The executing time of the Tier1 module)

Chapter 6
Conclusions and Future Work

6.1 Conclusion

The main target of this thesis is to accelerate the JPEG2000 encoder and implement the encoder on the TI C6416T DSP platform. We have presented three known methods and proposed two improved methods to accelerate the Tier1 module, which is the major part in the JPEG2000 encoder. We first presented the previous speed-up methods and discussed their advantages and drawbacks. The advantages of these methods may not match the sequential processing environment. Therefore, we proposed the VGOSS and the modified VGOSS methods. Also, the codes are modified to allow program level optimizations as discussed in Section 3.4.3 and 5.1.4 .

. The proposed VGOSS method is constructed on the *re-ordered* code-block samples and the extension of the GOSS and PP methods. It encodes only the NBC (need-to-be-coded) samples but reduces the checking cycles in the original three pass processes. It is easy and simple to implement and has a good performance on the DSP platform. It can run about 3.6 times faster than the original software on the C64xx simulator, which uses the flat memory system. It means a reduction of the computation cycles up to 72 %. However, the real DSP system may spend extra cycles on accessing external memory. It is profiled by the C6416 simulator explained in Section 3.3.3 . And our best performance is up to 32 times faster than the original one without DSP optimization. If the DSP compiler-level optimization technique is applied to the original codes, the speed-up is about 45%. The improvement is due to two factors, (1) our proposed VGOSS method and (2) the program level optimizations. Finally, we compare the results between the simulator and the emulator (i.e. hardware platform). The modified VGOSS method has about the same execution performance as VGOSS method on DSP although we expect it would be faster. All the results are consistent.

In summary, our proposed acceleration methods have improved the JPEG2000 encoder quite significantly. However, the JPEG2000 algorithm is still complicated in hardware implementation. The memory bottleneck is still a challenge to the embedded system. Because our DSP platform supports the L2 cache memory system, we have a great improvement by using this feature. There is still room for improvement to accelerate the JPEG2000 algorithm on different types of the embedded systems.

6.2 Future Works

The block coding takes the major part of the computational cycles in the JPEG2000 encoding process. The experimental results given in this thesis are the lossless image encoding and our proposed methods reduce checking cycles of the Pass2 and Pass3 processes. The acceleration methods can be used for encoding a lossy image also. It takes some effort to improve the rate control module using our proposed method in lossy image coding. In each bit-plane coding, our proposed methods have counted the number of NBC samples in the first pass process. Also, it means that we can obtain the distortion information before the Pass2 and Pass3 processes in the current bit-plane. Also, our proposed method match DSP hardware well and can be more efficient than the other known methods. In the reference software, the DWT module has been accelerated by a lifting scheme and is about 2 times faster than the previous vision of the reference software. We did not focus on accelerating the DWT module yet, but it becomes an important part in the JPEG2000 encoder after our acceleration. The speed-up of the DWT module can be another topic for research.

References

[1] JPEG2000 Part I Final Draft International Standard (ISO/IEC FDIS15444-1). ISO/IEC JTC1/SC29/WG11 N1855, Aug. 2000.

[2] JPEG2000 Requirements and Profiles, ISO/IEC JTC1/SC29/WG1 N1271, Mar.1999

[3] Information technology, JPEG2000 image coding system Part 2 : Extensions (ISO/IEC 15444-2), 2002.

[4] Information technology, JPEG2000 image coding system Part 3 : Motion JPEG2000 (ISO/IEC 15444-3), 2002.

[5] Information technology, JPEG2000 image coding system Part 4 : Compliance testing (ISO/IEC 15444-4), 2002.

[6] Information technology, JPEG2000 image coding system Part 5 : Reference software (ISO/IEC 15444-5), 2002.

[7] C. Christopoulos, A. Skodras, and T. Ebrahimi, "The JPEG2000 Still Image Coding System: An Overview", IEEE Transactions on Consumer Electronics, Vol.46, No. 4, pp. 1103-1127, Nov. 2000.

[8] M.J. Nadenau and J. Reichel, "Opponent Color, Human Vision and Wavelets for Image Compression", in Proc. 7th Color Image Conf., Scottsdale, AZ, 16-19 Nov. 1999, pp.237-242.

[9] D. Taubman and et al, "Embedded Block Coding in JPEG2000", in Proceedings of IEEE International Conference on Image Processing, vol. 2, Vancouver, Canada, Sept. 2000, pp.33-36.

[10] J.L. Mitchell and W.B. Pennebaker, "Software Implementation of the Q-coder", IBM J. of Res. Develop., Vol.32, No.6, pp.753-774, Nov. 1988.

[11] D. Taubman, "High Performance Scalable Image Compression with EBCOT", IEEE Transaction on Image Processing, Vol. 9, No. 7, pp. 1158-1170, July 2000.

[12] Sundance home page : http://www.sundance.com

[13] Texas Instruments, "TMS320C6414T, TMS320C6415T, TMS320C6416T fixed-point Digital Signal Processors", Literature number SPRS226, NOV. 2003.

[14] Texas Instruments, "TMS320C6000 Code Composer Studio Tutorial", Literature number SPRU301CI, Feb. 2000.

[15] Texas Instruments, "TMS320C6000 Programmer's Guide", Literature number SPRU198I, Mar. 2006.

[16] Texas Instruments, "TMS320C64x Technical Overview", Literature number SPRU395B, Jan. 2001.

[17] Texas Instruments, "TMS320C64x DSP Two-Level Internal Memory Reference Guide", Literature number SPRU610B, Aug. 2004.

[18] Texas Instruments, "TMS320C6000 Optimizing Compiler User's Guide", Literature number SPRU187L, May. 2004.

[19] Texas Instruments, "TMS320C6000 CPU and Instruction Set Reference Guide", Literature number SPRU189F, Jan. 2000.

[20] The JasPer Project Home Page : http://www.ece.uvic.ca/~mdadams/jasper/

[21] The OpenJPEG Home Page : http://www.openjpeg.org/index.php?menu=main

[22] The JPEG2000 still image compression standard, ISO/IEC JTC 1/SC 29/WG 1 N 2412 (ITU-T SG 16), Dec. 2002

[23] K.L. Lin, "Analysis and Architecture Design for JPEG2000 Still Image Encoding System", M.S. thesis, Department of Electrical Engineering, National Central University, Chung-Li, Taiwan, ROC, 2002.

[24] T.H. Tsai and L.T. Tsai, "JPEG2000 Encoder Architecture Design with Fast EBCOT

Algorithm", IEEE VLSI-TSA International Symposium, page 279-282, April 2005.

[25] C.J. Lian and et al, "Analysis and Architecture Design of Block-coding Engine for EBCOT in JPEG2000", IEEE Transactions on Circuits and Systems for Video Technology, Vol. 13, No. 3, March 2003.

[26] B.D. Choi, and et al, "DSP Implementation of Real-time JPEG2000 Encoder Using Overlapped Block Transferring and Pipelined Processing", International Conference on High Performance Computing (HiPC) 2004, Vol. 3296, page 333-341.

[27] J.K. Cho and et al, "Fast DSP implementation of JPEG2000", TENCON 2004, Vol. A, page 231-234 Vol. 1, Nov. 2004.

[28] B. Valentine and O. Sohm, "Optimizing the JPEG2000 Binary Arithmetic Encoder for VLIW Architectures", IEEE International Conference on International Conference on Acoustics, Speech, and Signal Processing (ICASSP), Vol. 5, P.5, May 2004.

自　　傳

劉建志, 西元 1978 年生於台中市。西元 2000 年畢業於私立元智大學電機工程學系，之後於金門服役，退伍後於華邦電子公司擔任應用工程師一職長達三年，2005 年進入交通大學攻讀碩士學位，研究方向為數位訊號處理，影像和視訊壓縮等。

Chien-Chih Liu, born in 1978 and Taichung city. He received the BEng degree in Yuan-Ze University Department of Electrical Engineering in 2000. Once completing my two-year military service in Kinmen, I had a position (field application engineer) in Winbond company for three years. After that, I enrolled in Chiao Tung University and my major course is the digital signal processing, image, and video compression.