공학석사학위논문

Cost-effective Hardware Design of a SPIHT Compression Algorithm

2015년 2월

서울대학교 대학원

전기 컴퓨터 공학부

Nguyen Xuan Truong

Cost-effective Hardware Design of a SPIHT Compression Algorithm

지도교수 김 재 하

이 논문을 공학석사 학위논문으로 제출함

2014 년 8 월

서울대학교 대학원

전기 컴퓨터 공학부

Nguyen Xuan Truong

Nguyen Xuan Truong 의 공학석사 학위논문을 인준함

2014 년 8 월

위 원 장 : 채 수 익 (인)

부위원장 : 김 재 하 (인)

위 원 : 이 혁 재 (인)

목차

표 목차

그림 목차

CHAPTER I INTRODUCTION

1.1. Wavelet Image Compression

Image compression in the wavelet domain is widely used because of its high compression efficiency and the JPEG2000 standard is one of the famous examples of the wavelet-based compression [7]. For wavelet-based compression, three popular coding algorithms are the embedded zerotree wavelet (EZW) [8], embedded block coding with optimized truncation (EBCOT) [9], and set partitioning in hierarchical trees (SPIHT) [1] algorithms. They have embedded coding property with progressive transmission of information, which allows the termination of coding operations when the target bit length (TBL) is reached. Embedded coding property can be utilized by a number of video applications that need image compression for temporary storage. For example, frame memory compression in a video compression chip requires a temporary storage that can be saved by embedded image compression [12]-[15]. A frame buffer for a large size display device is another example of temporary storage for embedded compression [16]–[17]. Among the three wavelet-based coding algorithms, EZW and EBCOT need binary arithmetic coding [8]-[9] which requires a large amount of hardware circuitry and storage. On the contrary, SPIHT can omit the binary arithmetic coder with a small drop-off of coding efficiency [1]. Therefore, it offers a cheap and fast solution. In addition, SPIHT surpasses EZW and is close to EBCOT in compression efficiency [1]-[2]. For these reasons, SPIHT has been extensively studied to achieve

an effective and low-cost image compression algorithm [2]-[6].

1.2. SPIHT-based Algorithms

The original SPIHT algorithm uses three lists to process wavelet coefficients. Thus, it requires a large of memory and circuitry to maintain three link lists. Because of its complex computation structure and data dependence between operations, the SPIHT algorithm is not easy to achieve a high throughput when it is implemented in hardware. Wheeler and Pearlman propose a modified SPIHT algorithm, called NLS (No List SPIHT), which does not require a relatively simple data structure [2]. Later Corsonello *et. al*., presents a hardware implementation of NLS [3]. As a result, NLS can process a very large block size such as 128x128 with a relatively small memory size. However, the throughput of NLS is very slow, only 0.092 bit per cycle [3]. To improve the coding speed, Chen *et. al.,* in [4] proposes a modified SPIHT that processes a 4x4 bit-plane in a single cycle. This algorithm does not exploit pixel parallelism but processes multiple sequential steps in one cycle in its hardware implementation resulting in a significant increase of the critical path delay in combinational logic circuits. Consequently, the operating clock frequency is only 10MHz and the throughput is not very high. Fry *et. al*. in [5] propose a bit-plane parallel SPIHT encoder architecture which decomposes wavelet coefficients into multiple bit-planes which are processed independently in a parallel manner. Then, the results of the multiple bit-planes are combined together to make a single bit-stream. This bit-plane-parallel approach achieves very large throughput by processing four pixels in a single cycle. However, the main drawback is that this bit-

plane-parallel approach is not applicable to a decoder. In a decoder, multiple bit-planes cannot be decoded in parallel because the decoder cannot predict the length of each bit-plane, and consequently, cannot divide the bit-stream into multiple bit-plane streams for parallel processing. As the speed of a decoder is often more important than the encoder speed, the slow decoding speed in the bit-plane parallel SPIHT severely limits the application of this algorithm. Jin *et. al*. in [6] propose a block-based pass-parallel SPIHT algorithm (BPS). BPS allows the fast processing speed for both an encoder and a decoder by coding a 4x4 block of a single bit-plane in a single cycle. The main contribution of BPS is to reorder the scanning phases, which allows the bit length pre-calculation enabling parallel execution for both an encoder and a decoder. However, BPS is designed only for 4x4 so that it cannot be applicable for different block sizes. The fixed block size results in the fixed coding speed, and consequently, it may not be fast enough to be used for very large video applications. Another limitation in BPS is that it uses an unnecessarily large buffer space because it does not take advantage of the independence among SPIHT operations.

This thesis proposes a generic partitioned SPIHT algorithm that overcomes the two drawbacks of BPS. For a given block to be compressed, the proposed algorithm partitions the block into small sub-blocks. The proposed algorithm is designed to handle various sub-block sizes unlike BPS which can handle only a 4 x 4 block. The target sub-block size depends on the decomposition level of DWT such as 2^n x 2^n if the decomposition level is n. For one-dimensional (1D) compression, 1 x 2^n is the target sub-block size. The algorithm is also applicable for any sub-block size

that is a multiple of this target size. For any sub-block size that is smaller than the target size, BPS can be used as far as it is a multiple of 4x4. To overcome the second drawback of BPS, the proposed algorithm minimizes the buffer storing intermediate results. To this end, the proposed algorithm partitions into sub-blocks which are independent of each other in coding operations. Therefore, the proposed algorithm compresses all the bit-plane in a sub-block before it moves to the next sub-block so that all the intermediate results for one sub-block are not necessary to be stored after the sub-block coding is completed. In this manner, the buffer space for the intermediate results can be shared by multiple sub-blocks and the total buffer size is significantly reduced. The proposed partitioned SPIHT is implemented in hardware achieving the throughput of 484 Mbps. The hardware requires 26.7K gates for an encoder and 34.3K gates for a decoder which is a reduction of 45.44% when compared with the non-partitioned SPIHT. The compression efficiency is evaluated with 18 test images and experimental results show that the image quality is 1.30dB less than the non-partitioned SPIHT.

This thesis is organized as follows. Section II introduces a DWT and SPIHT based compression algorithm and Section III describes the proposed block-based SPIHT algorithm. Section IV presents an improvement of the block-based algorithm by adjusting the target bit length of each block according to its complexity. In Section V, the proposed algorithm is evaluated. Its coding efficiency is evaluated with experimental results and its complexity is estimated with hardware implementation. Section VI draws the conclusion

CHAPTER II BASIC ARCHITECTURE OF THE COMPRESSION ALGORITHM

SPIHT is a fast and efficient compression algorithm for an image in the wavelet transformed domain. Fig 2.1(a) shows the data flow of the image compression based on DWT and SPIHT. A source image (pixel domain) is first transformed into wavelet domain by DWT. Generated coefficients are converted into bit-planes which are compressed by SPIHT to generate the final bit-stream. The decoder performs the reverse operations of the encoder (Fig. 2.1(b)). The bit-stream is first decoded by inverse SPIHT in order to reconstruct bit-planes. The decoded bit-planes are converted to coefficient domain. Finally, coefficients are transformed by inverse DWT, generating the reconstructed image data.

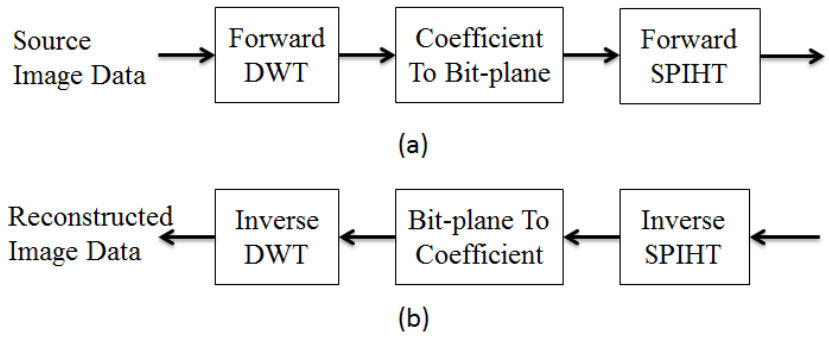


Figure 2.1: Data flow of the compression by SPIHT in the DWT domain

(a) Encoder (b) Decoder

2.1. Discrete Wavelet Transform

Discrete wavelet is the second generation of wavelet. For the hardware implementation of DWT, this thesis implements the *lifting-based filtering* which consists of a sequence of simple filtering operations. The lifting scheme consists of three stages: splitting, predicting and updating. In the first stage, the input sequence is divided into two subsets, an even-indexed sequence and an odd-indexed sequence. The even sequence is used to predict the odd one in the second stage. The difference between the odd sequence and the prediction values are calculated as the high coefficient. In the third stage, the even sequence is updated with the high-pass coefficient to compute the low-pass coefficient. Fig. 2.2 shows the block diagram of three-level lifting DWT using the bi-orthogonal Le Gall 5/3 integer filter [10]-[11] which is used in this paper (also adopted in JPEG2000 [7]). Fig. 2.2 shows the predicting and updating operations. In Fig. 2.2(a), the high-pass and low-pass coefficients ($H^j(i)$ and $L^j(i)$) at the jth level is calculated from the low-pass coefficients at the previous level by using the following equations [7]-[10]. The second stage generating the high coefficient is formulated as follows:

$$H^j(i) = L^{j-1}(2i-1) - \left\lfloor \frac{L^{j-1}(2i) + L^{j-1}(2i-2)}{2} \right\rfloor \qquad (2.1)$$

where i is the index of a coefficient for $0 \le i \le (\frac{N}{2^j} - 1)$, $\lfloor a \rfloor$ indicates the largest integer not exceeding$a$. The third stage generating the low coefficient is formulated as follows:

$$L^j(i) = L^{j-1}(2i-2) + \left\lfloor \frac{H^j(i) + H^j(i-1) + 2}{4} \right\rfloor \qquad (2.2)$$

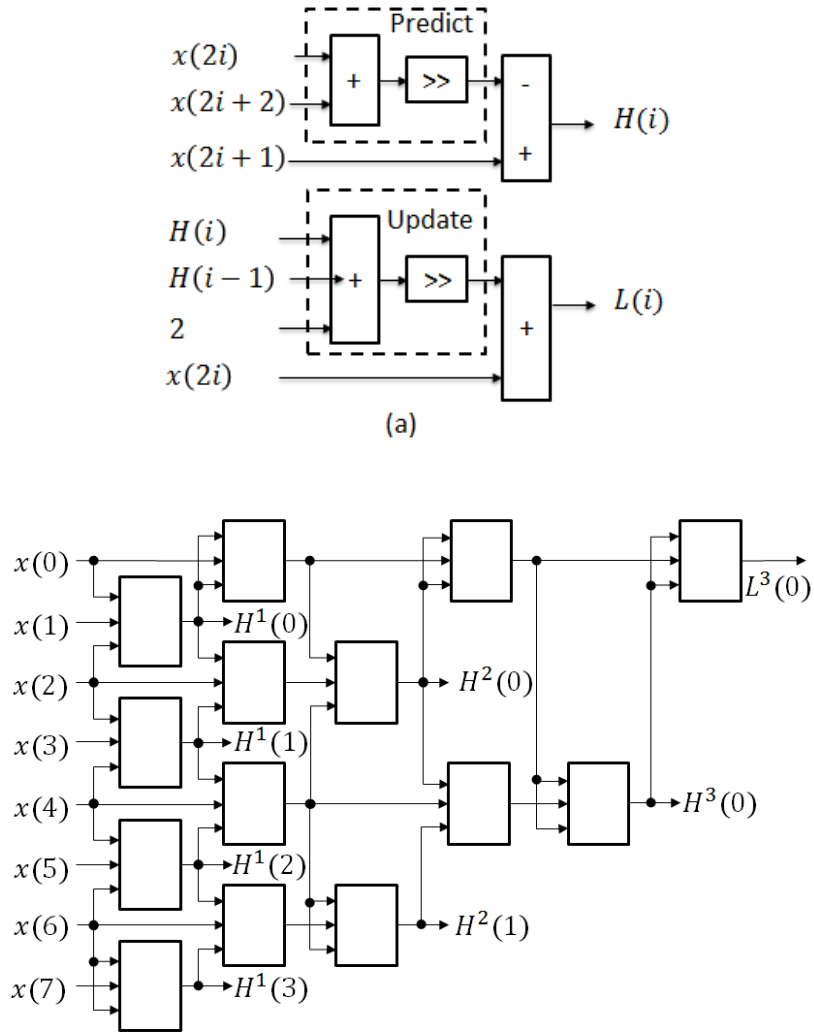


Figure 2.2: Three-level DWT for eight-point input sequences

To form $n-$level DWT, n identical filtering modules are connected as shown in Figure 2.2(b) which shows an example of 8-point DWT. After the first level analysis, the low-pass coefficient is decimated by two (4 coefficients comparing to 8 original coefficients). Four high-pass coefficients also are calculated in this level. At the second level, the number of low-pass coefficients remains by two. Finally,

the low-pass and 3th-level high-pass coefficients are calculated. Note that in order to calculate the $H^1(3)$ coefficient, an extension coefficient x_{ext} is required. In general, x_{ext} refers to $x(8)$ if $x(8)$ is used in DWT. On the other hand, if only $x(0), x(1), \dots, x(7)$ are used to calculate output coefficients, x_{ext} is set to $x(6)$. The boundary-extension method applied in the implementation is the symmetric method which is also adopted in JPEG2K ([7]).

When the processing unit of DWT is large, the output result is repeated with a given period in the sense that the level and index of the generated coefficient is same for each period. For example, the implementation of the three-level DWT with 32 input samples is presented in Table 2.1. The first column shows the execution time and the second column shows the input data which is received one data per one cycle. The third column shows the generated output sequence. The index in the parenthesis represents the output DWT coefficient index. The fourth column shows the DWT decomposition level and the index in each level. From the fifth to eighth columns also show the same as those from the first to the fourth columns where the time starts from the 18th cycle in the fifth column. As the decomposition level is 3, the output is repeated with the period of 8. For example, $H^1(0)$ is generated at time 1, 9, 17, and 25. The darkness in this table indicates the output in the same period. Due to the repeating property, the same hardware generating the first 8 coefficients can be used repeatedly generated the remaining sequence. This implies that the hardware cost does not increase even though the length of the input image sequence. Note that the repetition period depends on the DWT level. For four-level DWT, the repetition

period is 16 (=2^4). In general, n-level DWT generates the output sequence repeating in the period of 2^n.

Table 2.1: Three-level DWT operation of thirty two input and output samples

t	Input	Output	DWT	t	Input	Output	DWT
0	$x(0)$	-	-	18	$x(18)$	C(1)	$L^3(0)$
1	$x(1)$	C(16)	$H^1(0)$	19	$x(19)$	C(22)	$H^1(1)$
2	$x(2)$	-	-	20	$x(20)$	C(10)	$H^2(0)$
3	$x(3)$	C(20)	$H^1(1)$	21	$x(21)$	C(26)	$H^1(2)$
4	$x(4)$	C(8)	$H^2(0)$	22	$x(22)$	C(6)	$H^3(0)$
5	$x(5)$	C(24)	$H^1(2)$	23	$x(23)$	C(30)	$H^1(3)$
6	$x(6)$	C(4)	$H^3(0)$	24	$x(24)$	C(14)	$H^2(1)$
7	$x(7)$	C(28)	$H^1(3)$	25	$x(25)$	C(20)	$H^1(0)$
8	$x(8)$	C(12)	$H^2(1)$	26	$x(26)$	C(2)	$L^3(0)$
9	$x(9)$	C(17)	$H^1(0)$	27	$x(27)$	C(23)	$H^1(1)$
10	$x(10)$	C(0)	$L^3(0)$	28	$x(28)$	C(11)	$H^2(0)$
11	$x(11)$	C(21)	$H^1(1)$	29	$x(29)$	C(27)	$H^1(2)$
12	$x(12)$	C(9)	$H^2(0)$	30	$x(30)$	C(7)	$H^3(0)$
13	$x(13)$	C(25)	$H^1(2)$	31	$x(31)$	C(31)	$H^1(3)$
14	$x(14)$	C(5)	$H^3(0)$	32	$x(32)$	C(15)	$H^2(1)$
15	$x(15)$	C(29)	$H^1(3)$	33	-	-	-
16	$x(16)$	C(13)	$H^2(1)$	34	-	C(3)	$L^3(0)$
17	$x(17)$	C(19)	$H^1(0)$	-	-	-	-

2.2 SPIHT algorithm

2.2.1 The original SPIHT

SPIHT is a fast and effective compression algorithm used for encoding DWT coefficients [1, 2]. It processes in a bit-plane by bit-plane manner from the most significant bit-plane down to the least significant bit-plane. SPIHT performs a significance test on a set of wavelet coefficients organized in a tree structure. The

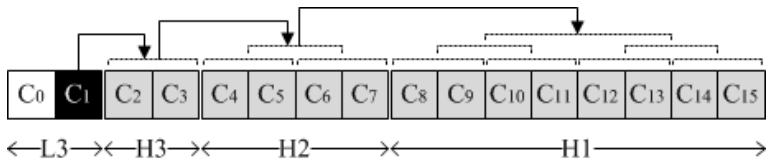
significance test is given by (2.3). For a given set of coefficients *T*, $S_n(\cdot)$ represents the result of significance test for the n*th* bit-plane:

$$S_n(T) = \begin{cases} 1, & max_{c_i \in T}\{|c_i|\} \geq 2^{\mathrm{n}} \\ 0, & otherwise \end{cases} \qquad (2.3)$$

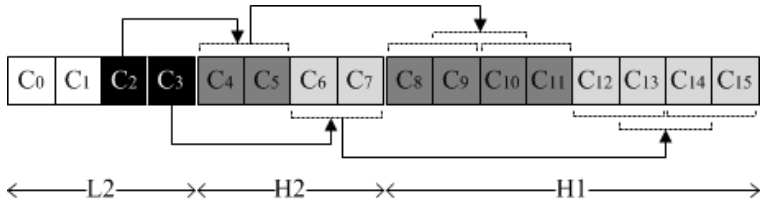
where $c(i)$ is the i*th* coefficient in T and 2^n is the threshold of the n*th* bit-plane (i.e., the significance of the bit-plane). The significance test classifies the set as a significant set in which the maximum coefficient of the set is larger than the significance of the bit-plane. Otherwise, the set is classified as an insignificant set and coded as a single bit '0'. The significant set is divided into subsets and the significance of each subset is tested again. When the subset has a single coefficient, the subset becomes a significant pixel or an insignificant pixel depending on the result of a significance test. The coding operation is terminated when all coefficients are coded or the encoded bit-stream length reaches the TBL.

SPIHT performs the significance test on the binary tree of coefficients. In this binary tree, a coefficient *c(i)* has two offspring which are the two coefficients *c(2i)*, and *c(2i+1)*. In turn, *c(2i)* has two offspring *c(4i), c(4i+1).* Therefore, *c(4i), c(4i+1), c(4i+2), c(4i+3)* are also descendants of coefficient *c(i)*. Fig. 2.3 shows the binary tree structure used in SPIHT for block size of 1 x 16. Note that the input of SPIHT is DWT coefficients and the tree structure depends on the DWT decomposition level. Fig. 3(a) and (b) show the structures for levels 3 and 2, respectively. In Fig. 3(a), coefficient *c(0)* corresponds to $L_3(0)$ which is the 0*th* low-band coefficient of level 3. Coefficient *c(1)* corresponds to $L_3(0)$ which is the 1*st* low-band coefficient of level 3. Coefficient *c(2)* corresponds to $H_3(0)$ which is the 0*th* high-pass band

coefficient of level 3. The correspondences of the other coefficients are also presented in Fig. 2.3(a). A half of the coefficients in the low-pass band with the higher index form the root(s) of all the binary trees. For example in Fig. 2.3(a), $L_3(1)$ is the only root of all the binary trees which are indicated by the arrows in the figure. In Fig. 2.3(b), $L_2(2)$ and $L_2(3)$ (corresponding to *c(2) and c(3)*) form the roots of the two binary trees.



(a)



(b)

Figure 2.3: Binary tree structure of DWT coefficients (a) Decomposition level 3 (b) Decomposition level 2

2.2.2 No-List SPIHT

NLS algorithm shown in Fig. 2.4 is a modified version of SPIHT in order to ease hardware implementation by using a marker for each DWT coefficient to store its encoding states. In NLS, every DWT coefficient is classified as one of three states, *LIS*(List of Significant Sets), *LIP* (List of Insignificant Pixels) and *LSP* (List of Significant Pixels). A coefficient is in *LIS* state if the set of itself and its

descendent coefficients is insignificant. If a coefficient is not classified as *LIS* state, then it is classified as either *LIP* or *LSP* state. A coefficient is in *LIP* state if it is insignificant. On the other hand, a coefficient is in *LSP* state if it is significant. NLS processes in a bit-plane by bit-plane manner and the algorithm in Fig. 3 presents the operation for a single bit-plane. In this algorithm, *mark[i]*, *magn[i]*, and *sign[i]* represent the state, magnitude, and sign of the *i*th coefficient *c(i)*.

NLS consists of three passes: refinement pass (RP), insignificant pixel pass (IPP), and insignificant set pass (ISP). RP processes the coefficients in *LSP* state (lines from 1 to 3). The magnitude bit of the current bit-plane for the coefficient is generated for the bit-stream (line 3). IPP processes coefficients in *LIP* state (lines from 4-9). The first step in this pass generates the magnitude bit (line 6) and then tests its significance (line 7). If it is significant, the sign bit is generated (line 8) and the state is changed to *LSP* (line 9). Lines from 14 to 23 describe the operation of ISP which processes the coefficients in *LIS* state. First, it tests the significance of the set that consists of the descendent coefficients of the current coefficient. Let *S(i)* be defined as the significance of the set consisting of the coefficient *c(i)* and all its descendant coefficients The first step of ISP tests *S(i)*. This value is one if the set is significant (i.e., the set includes at least one significant coefficient). On the contrary, this set is insignificant if all coefficients in the set are insignificant. If this set is insignificant, then only the value of *S(i)* is generated and the operation of ISP ends. If this set is significant, then its two offspring coefficients are marked as *LIS* (line 17). The next steps generate the magnitude bit, the sign bit and change the status depending on the significance of the coefficient (lines from 18 to 23). Note that the

effective compression of SPIHT is achieved by ISP which generates only a single bit for all the descendent coefficients if the significant test returns zero.

Refinement Pass		
1.	**for** $i = 0;\ i < L; i + 1$	
2.	**if** $mark[i] = LSP$ **then**	
3.	**output** $magn[i]$	
Insignificant Pixel Pass		
4.	**for** $i = 0;\ i < L; i + 1$	
5.	**if** $mark[i] = LIP$ **then**	
6.	**output** $magn[i]$	#magnitude
7	**if** $magn[i] = 1$ **then**	
8.	**output** $sign[i]$	#sign
9.	$mark[i] = LSP$	
Insignificant Set Pass		
13.	**for** $i = 0;\ i < L; i = i + 2$	
14.	**if** $mark[i] = LIS$ **then**	#Sorting
15.	**output** $S(T_i)$	#Sorting
16.	**if** $S(T_i) = 1$ **then**	
17.	$mark[2i] = LIS$ $mark[2i + 2] = LIS$	
18.	for $j = 0, 1$	
19.	**output** $magn[i + j]$	#Magnitude
20.	**if** $magn[i + j] = 1$ **then**	
21.	**output** $sign[i + j]$	#Sign
22.	$mark[i + j] = LSP$	
23.	**else** $mark[i + j] = LIP$	

Figure 2.4: The NLS algorithm for one-dimensional block

2.2.3 An example of NLS

Figure 2.5 illustrates the scanning orders and an example of bit-plane of a block. In a block, coefficients are ordered and indexed from low-pass to high-pass coefficients and from higher level coefficients to lower level coefficients, as shown in Figure 2.5(a-b). Figure 2.4(c) presents an example of coefficients in a 1x16 block

in the predefined order. Low coefficients (L3) are 56 and 57, while the third-level high ones (H3) are 5 and 1. The second-level high coefficients (H2) are -6, -2, 3 and 2, while the first-level high ones are 10, -9, 5, 0, 1, 4, 0 and -2. Bit-planes are extracted at the same orders and indexes. Figure 2.4(d, e, f) show the sign bit-plane, the 5th bit-plane and the 3rd bit-plane, respectively.

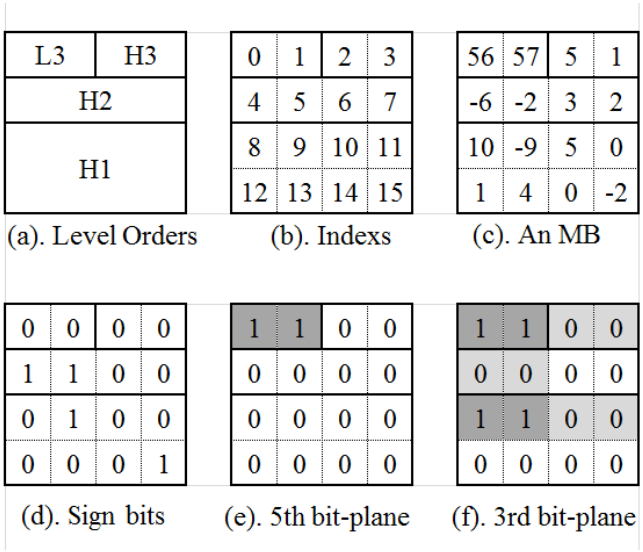


Figure 2.5: Scanning order and an example of a block

An example of the NLS algorithm for the coefficients is presented in Figure 2.6 where the 5th and 3rd bit-planes are coded. At first, two sets {55, 57} and {5, 1} are initialized as LIS sets by setting the marker for *c(0) and c(2)* as LIS. Note that L3 coefficients are not in the spatial tree. However, they can be treated as a set that doesn't have any descendants. At the 5th bit-plane, L3 coefficients have a non-zero bit, its sorting bit is set '1'. The magnitude and sign bits are immediately coded as shown in the corresponding columns. The set of coefficients {56, 57} is not in the

sorting list, and their markers are set by LSP. The remaining bit-plane bits are zero, so that only one bit ‘0’ is coded to represent to them.

At the 3rd bit-plane, the low coefficients or {56, 57} are LSP coefficients, thus only their magnitude bits are coded. The coefficients in set {5, 1} have descendants having a non-zero bit, thus their sorting bit is coded by ‘1’. Because their magnitude bits are 0 and 0, two bits of ‘0’ are coded and their sign bits are not coded at this bit-plane. Each coefficients in {5, 1} are set to LIP, and their offspring sets {-6, -2} and {3, 2} are set to LIS to perform the significance test. Similarly, the sorting bit for set {-6, -2} is set to one because its descendants have a non-zero bit at the 3rd bit-plane. The corresponding magnitude 0, 0 are coded, thus the sign bits are not coded. Each coefficients in set {6, -2} are set to LIP, and then their offspring sets {10, -9} and {5, 0} are set to LIS to perform the significance test again. The significance test for set {10, -9} is ‘1’, thus the sorting bit ‘1’ is coded. Since the bit values of {10, -9} are 1 and 1 at the 3rd bit-plane, two magnitude bits ‘1’ are coded, following by two sign bits 0 and 1. On the other hand, the sets {3, 2} and its descendants don’t have non-zero bit at this bit-plane, only bit ‘0’ is coded as the result of the significant test. After coding the 3rd bit-plane, markers’ states are listed in the last column in Figure 2.5. Two sets H2{3, 2} and H1{5, 0} are currently in the significance test. Meanwhile, L3{56, 57} and H1{10, -9} are in LSP. Two set H3{5, 1} and H2{-6, -2} are in LIP.

Bit-plane	Set	Sorting	Magnitude	Sign	Marker
5	**L3**{56, 57}	1	1, 1	0, 0	LSP, LSP
	H3{5, 1}	0			LIS
	H2{-6, -2}				
	H2{3, 2}				
	H1{10, -9}				
	H1{5, 0}				
	H1{1, 4}				
	H1{0, -2}				
3	**L3**{56, 57}	1	1, 1		LSP, LSP
	H3{5, 1}	1	0, 0		LIP, LIP
	H2{-6, -2}	1	0, 0		LIP, LIP
	H2{3, 2}	0			LIS
	H1{10, -9}	1	1, 1	0, 1	LSP, LSP
	H1{5, 0}	0			LIS
	H1{1, 4}				
	H1{0, -2}				

Figure 2.6: Example of coding bits with the NLS algorithm

Chapter 3 A PARTITIONED NLS ALGORITHM

3.1 Buffer Issues in SPIHT-based algorithms

The design challenge with a SPIHT-based algorithm is to handle with the large buffer sizes. Figure 3.1.a shows a data flow of an encoder implementing the DWT and SPIHT –based compression algorithm. It consists of four hardware modules, DWT, C2B (Coefficient to Bit-plane), SPIHT and Packer. C2B is the module that transforms the data structure generated from DWT to make it suitable for SPIHT operations. The Packer module is responsible to packetize the bit-stream generated by SPIHT for the predefined stream format. There exist three large memory buffers that temporarily store intermediate results. These buffers are represented by a shaded box in this figure.

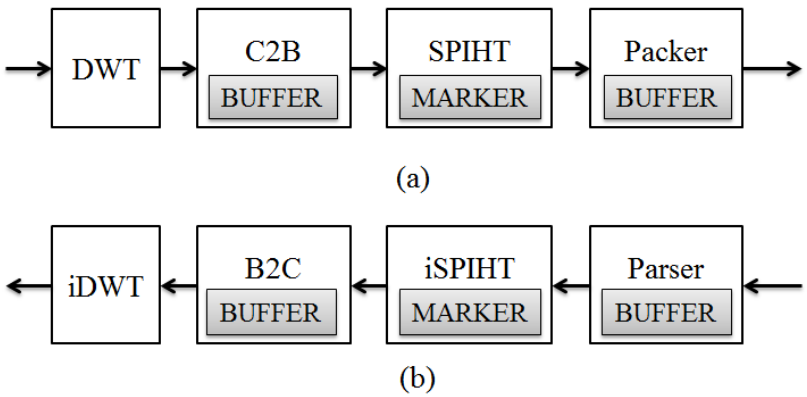


Figure 3.1: Internal Buffers in SPIHT-based algorithm

The first buffer is used by the C2B module to store the DWT coefficients. Note that DWT transforms the image inputs pixel-by-pixel, while the SPIHT encodes the

DWT coefficients bit-plane by bit-plane. The mismatch of the data structure between DWT and SPIHT makes it necessary to have a buffer storing DWT coefficients of the entire block. For example, with a 1x64 block, DWT generates 64 coefficients. In order to make a 64-bit bit-plane for SPIHT, all 64 coefficients are stored in C2B. Assume that each coefficient has 10 bits (1 sign bit and 9 magnitude bits), 640 bits are needed to be stored in the buffer. Moreover, in order speed up the processing speed and reduce latency, the pipeline manner is generally applied. This implies that two buffers are required so that when DWT writes coefficients into a buffer, SPIHT can read bit-plane from the remained one. In the pipeline manner, C2B needs to store 128 coefficients or 1280 bits. In general, the formula to calculate the buffer size is $BufSize1 = 2 \times B \times n$, where B denotes the number of coefficient per each block, n denotes the number of bits per each coefficient.

The second buffer is used by the SPIHT module to store and update the markers that store the current state of the coefficient. Recall that the state of the coefficient may be one of *NIL*, *LIS*, *LIP* and *LSP* states. Thus, each coefficient needs a 2-bit marker to store coding states. The buffer size is calculated as $BufSize2 = 2 \times B$, where B denotes the number of coefficient per each block. Note that a bit-plane a block can be divided into small pieces [15]-[17] to reduce combinational circuitry and complexity in SPIHT, but the same buffer size is still required.

The third buffer used by Packer stores the generated bit-stream. The size of this buffer may be large because of the mismatched processing speed between SPIHT and the output data rate. The SPIHT module encodes data bit-plane by bit-plane for which the output bit-stream size varies significantly depending on the characteristic

of a bit-plane. On the other hand, the output bit-stream is, in general, required to be generated in a constant or steady rate in order for a receiving device to capture the data. Assume SPIHT must perform 32-bit bit-planes, and the output bandwidth is 32. In SPIHT, 32 bits in a plane is divided into 16 sets to perform the significance test. The worst case is that all sets becomes significant at the same time, 16 significant test results, 32 magnitude bits and 32 sign bits are coded. As a result, the worst case of output coding size is 80 bits. This implies that the input size is 2.5 times larger than the output. In order to guarantee the system in worst case, the buffer size of target bit length is needed. Assume that a block size is 1x64 and the desired compression ratio is 2, the target bit length (TBL) is $\frac{64\times8}{2} = 256$ (bits). Therefore, the buffer size has 256 bits. In general, a large buffer inside Packer module is necessary to temporarily store the generated bit-stream and to transmit the data at the rate that the receiving device can handle. The buffer size is calculated by $BufSize3 = (8 \times B)/2$, where 8 refers to the 8-bit pixel format, B denotes the number of coefficient per each block, the minimum of compression is chosen as 2.

Total size of three buffers can be formulated by:

$$BufSize = BufSize1 + BufSize2 + BufSize3 = 2 \times B \times (n + 3)$$

Similarly, Figure 3.2.b shows the data flow of the decoder which processes data in the reverse order of the encoder. It consists of four hardware modules, iDWT (inverse of DWT), B2C (Bit-plane to Coefficient), iSPIHT (inverse SPIHT) and Parser. The buffers inside B2C, iSPIHT, Parser also have large size. The formula to calculate the buffer size in the decoder remains the same as in the encoder.

Recall that the hardware cost is affected by two factors: the processing speed and the processing block size. If the requirement for the processing speed is increased, the logic circuitry also needs to be increased proportionally in order to allow parallel (or pipelined) processing of hardware modules to process multiple data simultaneously. On the other hand, the increase of the block size increases proportionally the memory buffer size which stores the intermediate computation results. This is because the size of the intermediate results increases in proportion to the block size. In order to reduce the hardware cost by the internal buffer, a hardware designer may reduce the processing block size. However, the reduction of the block size causes the degradation of the compression efficiency. The remaining of this thesis attempts to reduce the buffer size by partitioning the processing block into small sub-blocks and then encoding the sub-block as the basic coding unit. In this manner, the buffer size is reduced by the number of the sub-blocks. The proposed sub-block encoding may cause the degradation of the compression efficiency. The main contribution is the partitioning scheme and the processing schedule of the sub-blocks that avoid a significant drop-off of the compression efficiency. The bit-stream generated by sub-block encoding also allows the sub-block decoding operation so that the decoder hardware cost is decreased as well.

3.2 Partitioned SPIHT for Buffer Reduction

This section proposes a modified SPIHT algorithm that partitions a processing block into multiple sub-blocks and then processes each sub-block independently to reduce the size of the buffers that store intermediate results. Although the proposed

SPIHT processes the partitioned sub-block, the DWT operation is performed with the original block size. Furthermore, the SPIHT algorithm is modified to reduce a drop-off of compression efficiency while avoiding a significant increase of the hardware logic complexity. The modified algorithm partitions the processing block into small sub-blocks and then processes each sub-block in a single cycle. Suppose that the block size is 1x32 which is partitioned into four sub-blocks P1, P2, P3, and P4, each of which consists of 1x8 coefficients. Sub-block P1 is coded at cycle t_1. When all the bit-planes of P1 are coded, the next sub-block P2 is processed. In this manner, hardware resources can be reused by the processing operation for P2. It is often the case that the dependence among coding operations makes it difficult to share the hardware resources. The reuse of buffer spaces is discussed in the next.

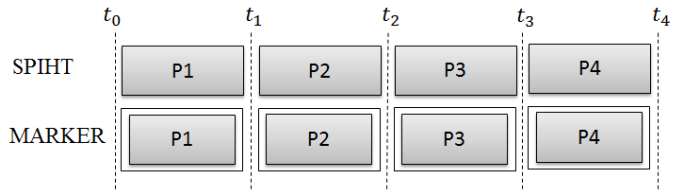


Figure 3.2: Sub-block-based SPIHT with shared maker space

To reduce the buffer that stores coefficient markers, the algorithm is modified to be processed sub-block by sub-block. It partitions a block into sub-blocks and reduces the buffer to store only one marker buffer for a sub-block instead of the entire block. Figure 3.2 shows an example assuming that the block size is 1x32 which is partitioned into four sub-blocks P1, P2, P3, and P4, each of which consists of 1x8 coefficients. Sub-block P1 is coded at time t_1. When all bit-planes of P1 are

coded, the next sub-block P2 is transformed by DWT. In this manner, markers are reset to the default values when a sub-block coding is completed. In other words, four sub-blocks share the same buffer space for marker storage. As a result, the number of markers to be maintained is reduced from 32 to 8.

The use of a single marker for all sub-blocks requires all the coding information to be reset after each sub-block coding is completed. This implies that the sub-block compression is made independently. This independent sub-block compression requires reorganization of the data structure. Figure 3.3.a shows the conventional linear data structure of the coefficients generated by DWT. This figure shows the example of 1x32 coefficients with the level 3 filtering operation. The 1x32 coefficients are partitioned into four sub-blocks of size 1x8. The first coefficient, *c(0)* is $L_0^3(0)$ of the first 8 input samples *{x(0), x(1), … , x(7)}* where the subscript 0 indicates that this coefficient is generated for the first 8 samples. The second coefficient, *c(1)* is $L_1^3(0)$ of the second 8 input samples *{x(8), x(9), … , x(15)}*. The third and fourth coefficients *c(2)* and *c(3)* are $L_2^3(0)$, $L_3^3(0)$of the third and fourth 8 input samples, respectively. The fifth coefficient *c(4)* is $H_0^3(0)$of the first 8 samples *{x(0), x(1), … , x(7)}* whereas the six, seventh and eighth coefficients *c(5), c(6)* and *c(7)* are $H_1^3(0)$, $H_2^3(0)$ and $H_3^3(0)$ of the second, third and fourth 8 input samples, respectively. As a result, the first sub-block corresponds to the DWT coefficients$\{L_0^3(0), L_1^3(0), L_2^3(0), L_3^3(0), H_0^3(0), H_1^3(0), H_2^3(0), H_3^3(0)\}$. The second sub-block corresponds to the DWT coefficients $\{H_0^2(0), H_0^2(1), H_1^2(0), H_1^2(1), H_2^2(0), H_2^2(1), H_3^2(0), H_3^2(1)\}$. Note that the marker status of the second sub-

block depends on the coding result of the first block. Therefore, it is impossible to reset the marker status between the coding operations of the first and second sub-blocks. Similarly, the dependence between the second and third sub-blocks prohibits the marker status from be reset.

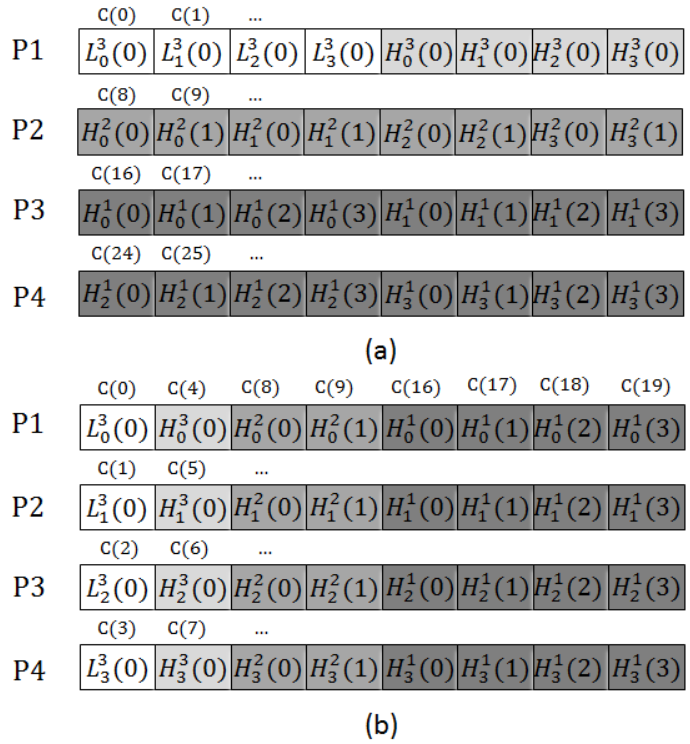


Figure 3.3: Data structures for a 1x32block

Figure 3.3(b) shows a new data structure reorganized for the independent SPIHT operation among different sub-blocks. The first sub-block is reorganized such that all the DWT coefficients for the first 8 samples consist the first sub-block, that is {c(0), c(1), … , c(7)} corresponding to $\{L_0^3(0),\ H_0^3(0),\ H_0^2(0),\ H_0^2(1),\ H_0^1(0),\ H_0^1(1),\ H_0^1(2),\ H_0^1(3)\}$. The second sub-block consists of the DWT

coefficients generated from the second 8 samples, that is {c(8), c(9), … , c(15)} corresponds to $\{L_1^3(0),\ H_1^3(0),\ H_1^2(0),\ H_1^2(1),\ H_1^1(0),\ H_1^1(1),\ H_1^1(2),\ H_1^1(3)\}$. Note that the SPIHT operations for these two sub-blocks are completely independent, and consequently, the marker can be reset after the end of the SPIHT operation for the first sub-block. The third and fourth sub-blocks are also generated from the third and fourth 8 input samples, respectively. In this manner, all four sub-blocks can be coded independently of each other and the markers can be reset between sub-block operations. As a result, the storage space for marker status is reduced to one fourth of the original space.

In order to process a sub-block by a sub-block, the SPIHT algorithm in Figure 2.3 needs to be modified because the previous SPIHT algorithm processes a pass by a pass. Figure 3.4 shows proposed partitioned SPIHT algorithm. In the modified SPIHT algorithm, the three separate loops (for three passes) are merged into a single loop. The input sample is partitioned into sub-blocks of size b which is processed independently. The operation is identical for all sub-blocks (from line 2 to 21) so that they are simply repeated as the outer-most loop (line 1).

The proposed partitioned algorithm is different from the previous block-based SPIHT such as BPS proposed in [6] because BPS can handle only a fixed sized block of 4x4 whereas the proposed algorithm can select a variable size of the sub-block. For a given DWT decomposition level of *n*, the target sub-block size of the proposed algorithm is 1 x 2^n. The proposed algorithm can handle 1D data and also be applicable for 2D data with target size 2^n x 2^n whereas BPS is applicable only for 2D data. The proposed algorithm can handle the size larger than 2^n x 2^n if it is a multiple

of the target size. On the other hand, the proposed algorithm also handles a size smaller than 2^n x 2^n because it can be combined with BPS which handles 4x4 blocks while the proposed idea is applicable. Therefore, the proposed algorithm can be applicable for general cases with a reduced hardware cost. Another important advantage of the proposed NLS is that BPS only attempts to speed up the processing time while the proposed algorithm aims to reduce the buffer size in SPIHT operations as well.

1.	$k = 0; k < L/b;\ k = k + 1$	
2.	$m = 0;\ m < b;\ m = m + 1$	
3.	$i = k * b + m$	
4.	**if** $mark[i] = LSP$ **then**	
5.	**output** $magn[i]$	#Magnitude
6.	**else if** $mark[i] = LIP$ **then**	
7.	**output** $magn[i]$	#Magnitude
8.	**if** $magn[i] = 1$ **then**	
9.	**output** $sign[i]$	#Sign
10.	$mark[i] = LSP$	
11.	**else if** $mark[i] = LIS$ **then**	
12.	**output** $S(T_i)$	#Sorting
13.	**if** $S(T_i) = 1$ **then**	
14.	$mark[2i] = LIS$ $mark[2i + 2] = LIS$	
15.	**for** $j = 0, 1$	
16.	**output** $magn[i + j]$	#Magnitude
17.	**if** $magn[i + j] = 1$ **then**	
16.	**output** $sign[i + j]$	#Sign
17.	$mark[i + j] = LSP$	
18.	**else** $mark[i + j] = LIP$	
19.	**output** $sign[i + j]$	#Sign
20.	$mark[i + j] = LSP$	
21.	**else** $mark[i + j] = LIP$	

Figure 3.4: Partitioned SPIHT algorithm

The new sub-block structure can also reduce the buffer size to store the result of DWT coefficients in B2C module in addition to the buffer in SPIHT module. This is because the DWT coefficients are generated in a periodical manner as shown in Table I which is, in fact, the same order as the proposed sub-block structure. Therefore, it is not necessary to store all the DWT operations. Instead, only the first set of the DWT coefficients are stored and forwarded to the SPIHT module because this first set corresponds to the first sub-block for SPIHT operation. This DWT and SPIHT operations are processed in a pipelined manner and the storage space in B2C module can be significantly reduced. To this end, two buffers each storing a sub-block is needed to store and forward the DWT results to SPIHT in a Ping-Pong manner. One buffer is necessary to store the DWT coefficients used as the input to SPIHT while the other buffer is used to store the output of DWT operations for the next pipeline stage.

Fig. 3.5 shows an example when the block size is 1x32 which is partitioned into four sub-blocks of size 1x8. Fig. 3.5(a) shows the conventional buffer that stores the entire 1x32 blocks in C2B module. For a pipelined execution in a Ping-Pong manner, two blocks need to be stored in the buffer of which size is 64 coefficients. The marker buffer also stores all the information for the entire 1x32 block. In Fig. 3.5(b), the space is reduced by the proposed scheme. When DWT transforms P1, 8 output coefficients are stored in the first part in the memory block. Note that those coefficients in P1 are $L^3(0)$, $H^3(0)$, $H^2(0)$, $H^2(1)$, $H^1(0)$, $H^1(1)$, $H^1(2)$, and $H^1(3)$. When the second part P2 is transformed in DWT, the next 8 output coefficients are stored in the second memory in the B2C module. At

the same time, B2C sends bit-plane by bit-plane of the first sub-block to the SPIHT block. When the third part P3 is transformed in DWT, the first part P1 in the memory block is no longer used. The coefficients of P3 are saved in the first memory. Simultaneously, B2C transfers bit-planes of the second memory to SPIHT. The process is repeated until the DWT transforms the last sub-block. As a result, the memory size in the B2C module is significantly reduced to store only two sub-blocks which consist of 16 coefficients. Thus, the buffer size is reduced only to 1/4 of the original size. Assume that the original block size is L and the sub-block size is P and the coefficient requires n-bit per a coefficient, then the reduction in memory size is $2(L-P)\times n$ bits.

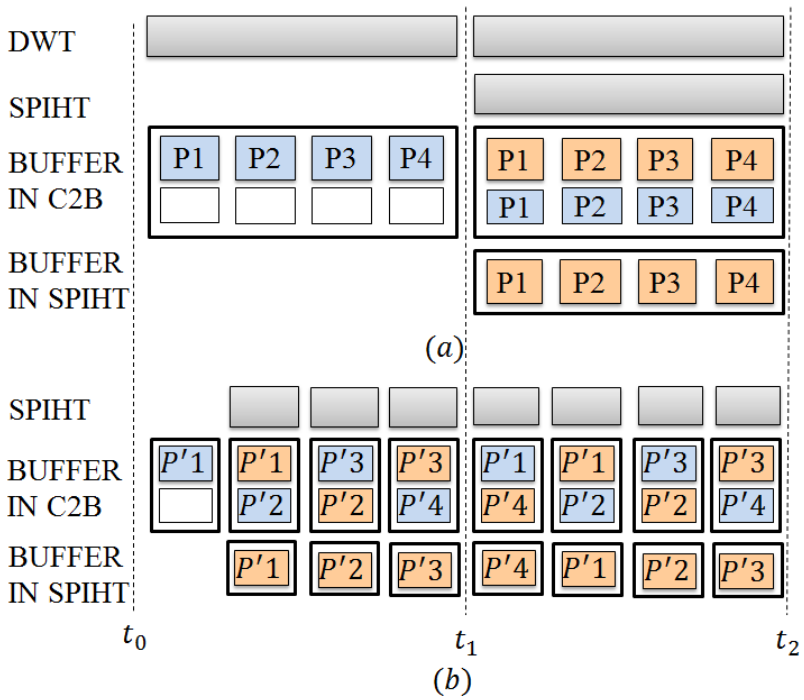


Figure 3.5: Scheduling in the partitioned algorithm

The proposed sub-block partitioning is different from the conventional DWT+SPIHT for a small block size. For example, the conventional DWT+SPIHT can be performed for 1x8 block as the basic processing block. Recall that the proposed partitioned algorithm uses 1x8 as the sub-block size while the basic processing block size is 1x32. There exists an important difference between the conventional algorithm for a 1x8 block and the proposed algorithm for a 1x8 sub-block partitioned from a 1x32 block. Fig. 3.6 shows the difference where the DWT operation of the second 1x8 block (x(8),…,x(15)) is presented. Fig. 3.6(a) shows the operations for the conventional approach. Note that the operation for the second 1x8 block is same as the first 1x8 block (x(0),…,x(7)). Therefore, Fig. 3.6(a) is exactly same as Fig. 2.2(b) except the index of the input sequence. On the other hand, the operation for the proposed algorithm is different from Fig. 2.2(b) as shown in Fig. 3.6(b). Comparing with Fig. 3.6(a), the output is same, but the input is slightly different. In Fig. 3.6(b), $\boldsymbol{x(6), x(7)}$ and $\boldsymbol{x(16)}$ are used for the operation but they are not in Fig. 3.6(a). This indicates that the conventional approach for 1x8 block does not make use of the image characteristics on the boundary pixels and degrade the compression efficiency. The sub-block compression for P2 uses 11 input samples from $\boldsymbol{x(6)}$ to $\boldsymbol{x(15)}$ for the generation of DWT coefficients of 1x8 sub-block whereas the conventional 1x8 operations use only 8 input samples from $\boldsymbol{x(8)}$ to $\boldsymbol{x(15)}$. Thus, it misses 3 out of 11 input samples, which causes a substantial degradation of the compression efficiency.

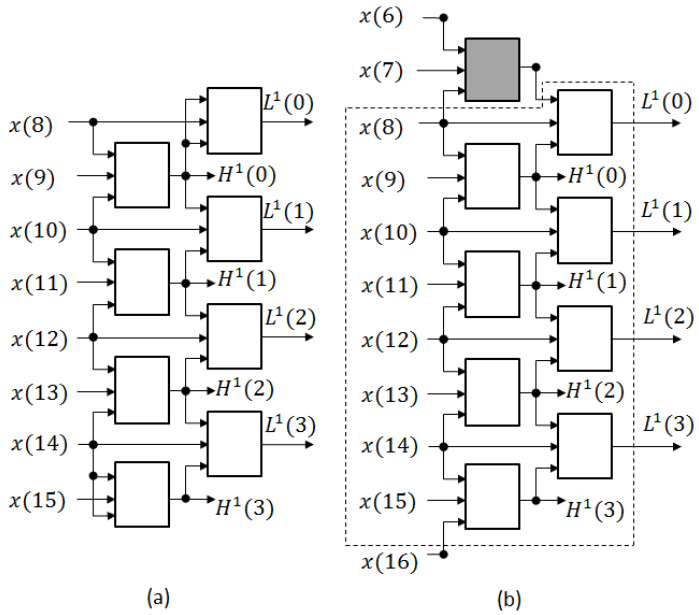


Figure 3.6: DWT analysis of the second 1x8 block (a) Conventional way (b) Boundary extension

As each sub-block is coded independently of each other, the target length of the compressed bit stream is determined independently for each sub-block. For example, assume that the target bit length of the 1x32 block is 256. Then, the target bit length of every sub-block of size 1x8 needs to be selected as 64. This independent coding of sub-blocks may decrease the compression efficiency because the degree of compression efficiency may differ among different sub-blocks. For example, one sub-block can be encoded efficiently while the other sub-block may not. In this case, the inefficient sub-block encoding may decrease the overall compression efficiency of the whole block. The next section investigates the effect of the independent sub-block coding on the compression efficiency and proposes a scheme to improve the efficiency.

Chapter IV: ADJUSTMENT OF THE TARGET BIT LENGTHS FOR INDIVIDUAL SUB-BLOCKS

4.1Motivation

Assume that a block is partitioned into four sub-blocks P1, P2, P3 and P4 (see Fig.4.1). Fig. 4.1(a) shows an example those four sub-blocks, P1, P2, P3, and P4 may have different lengths of the generated bit-stream when they are encoded in a lossless manner. If four sub-blocks are encoded independently, one fourth of the TBL is assigned to every sub-block. Fig. 4.1(b) shows an example that sub-blocks P1 and P4 are truncated by the given TBL. Meanwhile, sub-blocks P2 and P3 have wasted spaces. If the wasted spaces are used to store the truncated bit-streams of P1 and P4, then the data loss in P1 and P4 can be avoided (see Fig.4.1(c)). This re-allocation of bit-stream storage is implemented if the TBL of each sub-block is adjusted such that a relatively large TBL is assigned to a complex sub-block (P1 or P4 in this example) whereas a small TBL is assigned to a simple sub-block (P2 or P3). Fig. 4.1(d) shows an adjustment of the sub-block TBL. With the TBL adjustment, a wasted space is reduced, and consequently, the coding efficiency can be improved. The challenge in the TBL adjustment lies in that an appropriate TBL needs to be chosen for each sub-block. Let TBL_1, TBL_2, TBL_3 and TBL_4 be the target bit lengths assigned to sub-blocks, P1, P2, P3 and P4, respectively. The remaining of this section discusses how to select an appropriate values of TBL_1, TBL_2, TBL_3 and TBL_4.

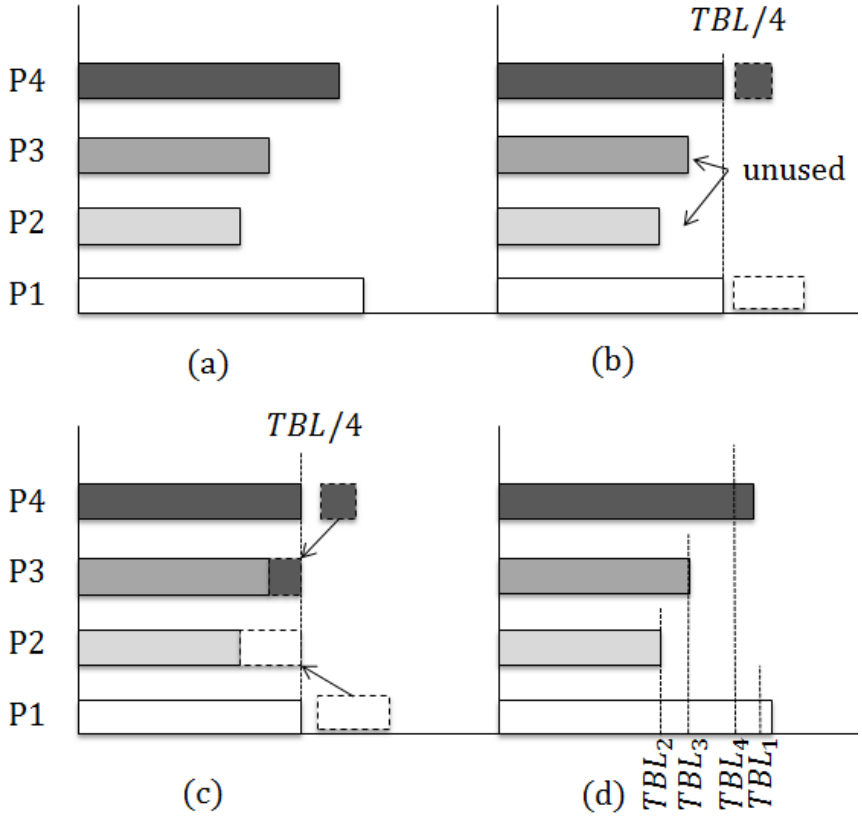


Figure 4.1: Adjustment of the target bit lengths of sub-blocks (a) generated bit-stream length for lossless compression (b) truncated bit-stream to meet the identical target bit length (c) reallocation of the truncated bit-stream to the other sub-block (d) adjusted target bit length for reallocation

4.2 Linear estimation of the loss of image quality

A SPIHT-based algorithm has an advantage in the estimation of the loss of image quality on the fly while the algorithm is running. This is because the loss can be easily estimated from the truncated bit-plane and the coefficient position where the generated bit-stream reaches the TBL. This subsection takes advantage of this

SPIHT property to estimate the loss of each sub-block and to adjust the target bit length for each sub-block. Let $loss_1, loss_2, loss_3$ and $loss_4$ denote the losses of the information by truncation for sub-blocks P_1, P_2, P_3 and P_4, respectively. These losses can be estimated from the amount of the truncated bits for sub-block coding. Suppose that SPIHT algorithm generates its bit-stream and reaches the TBL at the $Coef_i th$ coefficient of the $BP_i th$ bit-plane. Then, the estimated loss is defined as follows:

$$loss_i = BP_i \times 32/CR + Coef_i \quad (4.1)$$

where *CR* is the target compression ratio. The *CR* is used to the formula to adjust the relative importance between BP_i and $Coef_i$. Recall that a SPIHT–based algorithm processes bit-plane by bit-plane from the most significant plane to the least significant plane. In general, most significant bit-planes include many '0' bits, and consequently, their bit-stream lengths per a bit-plane are shorter than least significant bit-planes. For a small CR, the truncation takes place at one of the least significant bit-planes of which the length per a bit-plane is large. Therefore, the importance of BP_i is relatively large. On the contrary, a large CR results in the truncation position in relatively significant bit-planes of which the length per a bit-plane is relatively small. Therefore, it is reasonable to give a relatively small weight to BP_i. In summary, equation 4.1 implies that the importance BP_i relative to $Coef_i$ increases as the CR decreases. The numerator 32 is chosen from experiments.

From the estimated losses, $loss_1, loss_2, loss_3$ and $loss_4$, TBL_i is defined to make them as equal as possible.

$$TBL_i = TBL + \Delta_i \quad \text{for } i=1,2,3,4 \tag{4.2}$$

where Δ_i is defined as follows:

$$\Delta_i = loss_i - \left\lfloor \frac{loss_1+loss_2+loss_3+loss_4+2}{4} \right\rfloor \quad \text{for } i=1,2,3 \tag{4.3}$$

$$\Delta_4 = -(\Delta_1 + \Delta_2 + \Delta_3) \tag{4.4}$$

Equations (6) and (7) are formulated to make Δ_i equal for all i's, and consequently, guarantee that $TBL = TBL_1 + TBL_2 + TBL_3 + TBL_4$.

Hardware implementation can be simplified by making TBL adjustment only between P1 and P3 as well as P2 and P4. In other words, the goal of the adjustment is to make *loss1* and *loss3* equal as well as *loss2* and *loss4* equal. However, loss 1 and loss2 may not be equalized as a result of TBL adjustment. To this end, (BBB) and (CCC) are modified as follows:

$$\Delta_i = loss_i - \left\lfloor \frac{loss_i+loss_{i+2}+2}{2} \right\rfloor \quad \text{for } i=1, 3 \tag{4.5}$$

$$\Delta_{i+1} = -\Delta_i \quad \text{for } i=1, 3 \tag{4.6}$$

This simplification may slightly decrease the compression efficiency but it decreases the hardware complexity as well.

4.2 Mathematical formula for the proposed bit allocation

MSE values of a 1x64 block and its sub-blocks can be presented as follows:

$$mse_1 = \alpha \frac{loss_1^2}{16} \tag{4.7}$$

$$mse_2 = \alpha \frac{loss_2^2}{16} \tag{4.8}$$

$$mse_3 = \alpha \frac{loss_3^2}{16} \tag{4.9}$$

$$mse_4 = \alpha \frac{loss_4^2}{16} \tag{4.10}$$

$$mse = \alpha \frac{(loss_1 + loss_2 + loss_3 + loss_4)^2}{64} \tag{4.11}$$

where $mse_1, mse_2, mse_3, mse_4$ and mse are MSEs of sub-blocks P1, P2, P3, P4 and the block, respectively. A positive coefficient α represents to the method used to estimate the discarded information. Since the block and sub-blocks are applied with the same method, only one α is used. For a specific coding method and a specific block, the discarded information of a block is constant or the following sum is fixed:

$$loss_1 + loss_2 + loss_3 + loss_4 = const \tag{4.12}$$

We prove that independent sub-block coding increases the MSE, which results in the reduction in the coding efficiency. From the equation, 4.7-4.11, we obtain:

$$diff = mse_1 + mse_2 + mse_3 + mse_4 - mse =$$

$$= \alpha \frac{loss_1^2}{16} + \alpha \frac{loss_2^2}{16} + \alpha \frac{loss_3^2}{16} + \alpha \frac{loss_4^2}{16} - \alpha \frac{(w_1 + w_2 + w_3 + w_4)^2}{64}$$

$$= \frac{\alpha}{64} [2(loss_1 - loss_3)^2 + 2(loss_2 - loss_4)^2 + (loss_1 + loss_3 - loss_2 - loss_4)^2] \tag{4.13}$$

Since the last formula is always non-negative, the independent sub-block coding of a block with a specific coding method always derives a bigger MSE than the block

coding does. From the other perspective, the optimal solution for sub-block coding is to make the last formula be zero. Therefore, we obtain those equations:

$$0 = \frac{\alpha}{64}[2(loss_1 - loss_3)^2 + 2(loss_2 - loss_4)^2 + (loss_1 + loss_3 - loss_2 - loss_4)^2]$$

$$\Leftrightarrow \begin{cases} (loss_1 - loss_3)^2 = 0 \\ (loss_2 - loss_4)^2 = 0 \\ (loss_1 + loss_3 - loss_2 - loss_4)^2 = 0 \end{cases}$$

$$\Leftrightarrow loss_1 = loss_2 = loss_3 = loss_4 = \frac{loss_1 + loss_2 + loss_3 + loss_4}{4}$$

The final formula gives the optimal solution for the bit-allocation with sub-block coding. It implies that the scheme to archive the best coding efficiency is to obtain the same discarded information with all sub-blocks. Moreover, the final formula of *diff* indicates the solution to increase the coding efficiency. If we can archive *loss1* and *loss3* equal as well as *loss2* and *loss4* equal, the value of *diff* definitely decreases. Consequently, the coding efficiency is improved.

Chapter V Experimental Results

This section evaluates the compression efficiency of the proposed NLS and the cost of the hardware that implements the proposed algorithm with Verilog HDL.

5.1. Coding Efficiency

The coding efficiency of the proposed algorithm is compared with the original NLS as the reference SPIHT algorithm. For the proposed one, the block size is chosen as 1x64 which is partitioned into four sub-blocks of size 1x16. Recall that DWT is performed for a 1x64 block. For NLS, two block sizes of 1x64 and 1x16 are used and the coding efficiencies are evaluated for both block sizes. Tables 4 and 5 show the experimental results with two CRs equal to 2 and 4, respectively. For both tables, the first column shows test images for which eighteen monochrome images with 8 bits per pixel (bpp) are used (see Fig. 11). From the second to the fifth column shows the PSNR of various coding algorithm. NLS 16 and NLS 64 in the second and fifth columns refer to the NLS algorithm compressing 1x16 and 1x64 blocks, respectively. The notation 'Proposed (fixed)' in the third column represents the proposed algorithm with each target bit length fixed as the same value whereas 'Proposed (adjusted)' indicates the proposed algorithm with the adjusted TBL for an individual sub-block as presented in Section IV. The distortion is measured by the peak signal to noise ratio (PSNR):

$$PSNR = 10log_{10}\left(\frac{255^2}{MSE}\right) \quad dB \tag{5.1}$$

where *MSE* denotes the mean-squared error between the original and reconstructed images.

Figure 5.1: Eighteen monochrome test images

Comparison between NLS 16 and NLS 64 shows that the PSNR increases as the DWT block size increases. On average, the differences are 2.30dB for CR=2 and 2.07dB for CR=4. The proposed algorithm has its efficiency between NLS 16 and NLS 64. For CR=2, the proposed algorithm with the fixed TBL improves the PSNR by 0.63dB on average ranging from 0.34dB to 0.91dB when compared with the NLS 16. Meanwhile, the adjustment of the TBL improves the coding efficiency by 0.87dB on average ranging from 0.25dB to 2.47dB. For CR=4, the similar improvement is achieved by the proposed algorithm while the coding efficiency is improved by all algorithms when compared with the corresponding algorithms for CR=2. The proposed algorithm with the fixed TBL increases the PSNR by 0.47dB on average ranging from 0.31dB to 0.64dB when compared with NLS 16. Meanwhile, the TBL adjustment increases the coding efficiency by 1.49dB on average ranging from 0.70dB to 3.76dB. When the CR is small, the least significant bit planes are likely to

be included in the bit-stream. Thus, it reduces the efficiency of DWT with a large block size, and consequently, the improvement by the proposed algorithm with NLS 16 is relatively small. On the contrary, the increase of the TBL it gives an additional space to allocate bits among sub-blocks. Therefore, the efficiency of the TBL adjustment also increases.

Table 1: Quality of the compressed images at compression ratio of 4 (2bpp) (in PSNR dB)

Test Image	NLS 16	proposed (fixed)	proposed (adjusted)	NLS 64
Gold hill	29.41	30.09	30.66	31.18
Lena	29.60	30.51	31.48	32.12
Barbara	24.27	24.61	25.52	25.91
Baboon	24.12	24.58	24.83	25.24
Peppers	30.18	31.10	32.55	33.55
Cameraman	24.84	25.41	27.88	28.63
Man	29.30	30.08	30.86	31.41
Aerial	25.90	26.59	27.54	28.07
Airfield	24.97	25.61	26.00	27.00
Airfield2	27.28	27.84	28.48	29.02
Bridge	25.38	25.97	26.38	26.91
Clown	28.41	29.05	29.96	30.42
Couple	27.64	28.17	29.00	29.59
Girl face	31.82	32.58	33.87	34.38
Houses	22.72	23.23	24.13	24.53
Kiel	25.85	26.37	27.28	27.80
Lighthouse	24.63	25.11	25.50	25.80
Zelda	32.91	33.70	34.39	34.98
Average	27.18	27.81	28.68	29.25

Table 2: Quality of the compressed images at compression ratio of 2 (4bpp) (in PSNR dB)

Test Image	NLS 16	Proposed (fixed)	Proposed (Adjusted)	NLS 64
Gold hill	39.29	39.77	40.70	41.02
Lena	39.87	40.46	41.79	42.16
Barbara	33.64	33.95	36.12	36.50
Baboon	33.41	33.86	34.56	34.82
Peppers	40.56	41.22	42.74	43.13
Cameraman	35.49	35.89	39.65	40.08
Man	39.30	39.82	41.00	41.31
Aerial	36.46	37.01	38.98	39.41
Airfield	34.96	35.37	36.74	37.07
Airfield2	37.04	37.46	38.56	38.89
Bridge	35.11	35.55	36.18	36.46
Clown	38.37	38.79	40.38	40.78
Couple	37.75	38.22	39.63	40.00
Girl face	41.83	42.28	44.26	44.63
Houses	32.92	33.28	35.09	35.44
Kiel	36.00	36.46	38.23	38.57
Lighthouse	34.06	34.70	35.55	35.85
Zelda	42.41	42.85	43.62	43.83
Average	37.14	37.61	39.10	39.44

In Figure 5.2, the original and reconstructed images of the proposed method are shown. Visually, the reconstructed images look similar to the original image. However, their PNSR results are significantly different. The PSNRs at 4, 3 and 2bpp are 41.79, 36.86, and 31.48dB. The bit rate or the compression ratio plays an important role in storage and transmission. The bandwidth reduction can be archived by 50%, 62.5% or 75% when the bit rate 4, 3 or 2bpp are applied. In the contrary, the image quality are suffered the sustainably drop-off when the bit-rate is small. The following section discusses the ranges of applications of the proposed algorithm.

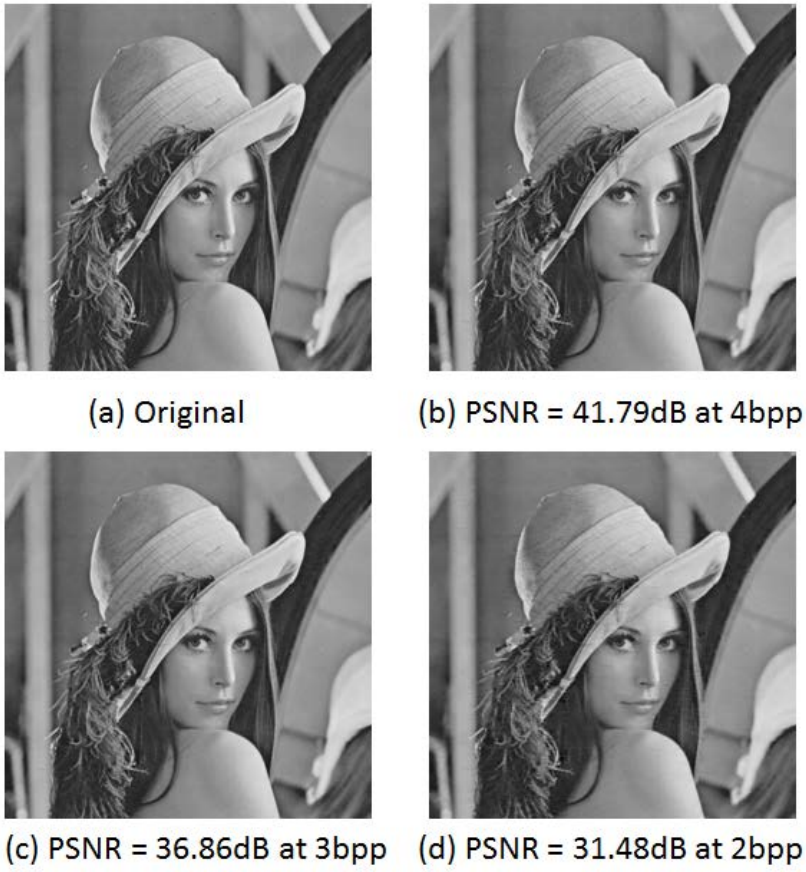


Figure 5.2: The original image and reconstructed images at 4, 3, and 2bpp

5.2 Hardware Implementation cost

5.2.1Hardware organization

This section shows the hardware organization of both implementations of the proposed and original NLS algorithms. The input and output bandwidths are 32 bits. The pixel format is YUV422 or UYVY. In Figure 5.3, we use bidirectional arrows due to the difference of data flows in encoding and decoding parts. In the proposed method or sub-block based method, only two register buffers are used in a Ping-Pong mode as shown in Figure 5.3. In the contrary, block-based method needs to store all DWT coefficient of a block, so that each buffer needs four parts each part has 2x16x10 bits. The number 2 refers to color mode. Since the color format is UYVY, a buffer is used for Y color, while another one is used by UV color. As a result, its buffer size is double times than the buffer size for the gray mode.

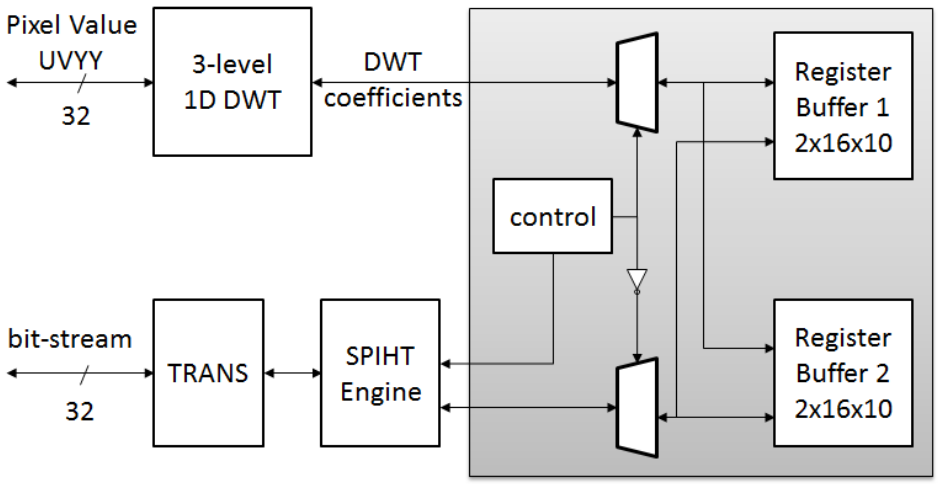


Figure 5.3: Hardware organization of the proposed algorithm

5.2.2 Processing Elements of SPIHT

SPIHT engine can be further simplified by constructing into processing elements (PE) which processes each 2 bits of two coefficients. A 16-bit bit-plane is divided by eight groups corresponding to 8 PEs connected by the Morton scanning order in Figure 5.4. The first PE is used by two bits of L3 coefficients indexed by 0 and 1, while the second PE is used by two bits of H3 coefficients indexed by 2 and 3. The third, fourth, fifth, seventh, and eighth PEs are defined at the same manner.

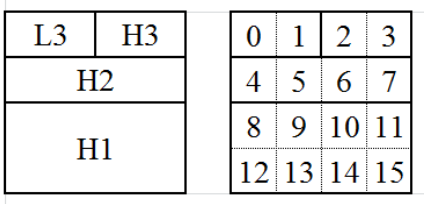


Figure 5.4: Morton-scanning order

Figure 5.5 shows two types of unified PE modules. Note that PEs used for L3 and H1 are at the same format (Type A), because they don't have to perform the significant test. On the other hand, PEs used for H3 and H2 coefficients must perform the significant test, thus they are at same format (Type B). Remind that each coefficient in SPIHT has a marker to indicate its coding state. In implementations, we construct two separate markers into a set marker and data marker. A set might be in a refinement state, a significant test state or non-coding state. In initialization, the set of L3 coefficients is in a refinement state in which no significance test is performed and only magnitude and sign bits are coded. In initialization step, only sets of L3 and H3 coefficients are initialized. The set of H3

coefficients is initialized as in the significance test state in which the significance test *S(T)* of all bit-plane of H3, H2 and H1 are examined. If the bit-plane of H3, H2 and H1 has a non-zero value, PE of H3 coefficients is moved to a refinement test, and two PEs of H2 coefficients are invoked by setting those sets in a significance test. The non-coding state refers to a set that is not coded yet or the length of bit-stream is zero. In this manner, length of a bit-stream for a PE varies from 0, 1, 2, 3, 4 and 5. Note that, the markers are constructed but the inside operation still remains. The order of coding bits for each PE is: a significance test followed by magnitude bits and sign bits. For eight PEs or 16-bit input per a bit-plane, we have eight 5-bit blocks of bit-stream and eight corresponding length values. Those values are packed and stored in 40-bit registers with a 6-bit length value.

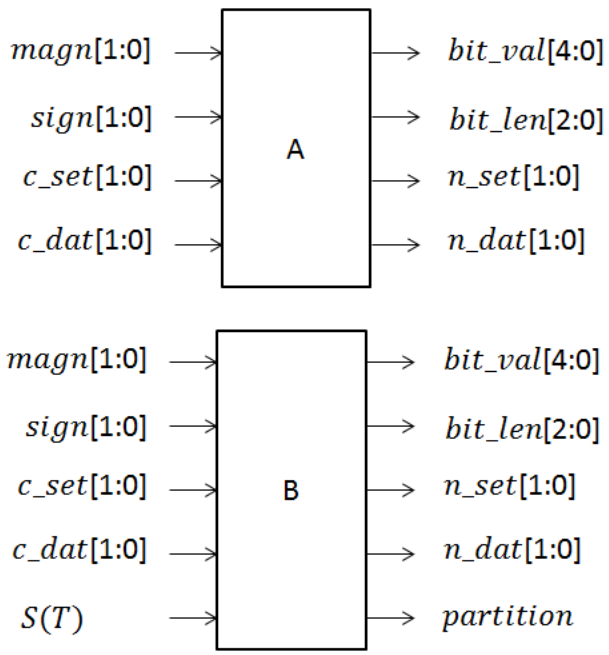


Figure 5.5: Two types of Processing Elements in Encoder

For the decoder, the processing is in reverse order, thus the similar types of processing elements are represented in Figure 5.6. Each PE is received a 5-bit input and current coding states. It decodes the magnitude and sign bits. It also calculates the next coding states and defines the decoded-bit length. Inputs for eight PE modules are 40 bits and each PE receives 5 bits as an input. The difference between decoder type A and decoder type B is that decoder type B also generates a partition flag. The flag indicates when a set is partitioned sub-sets to perform the significance test on the sub-sets. The coding states of sub-sets are updated by this flag before PE performs coding.

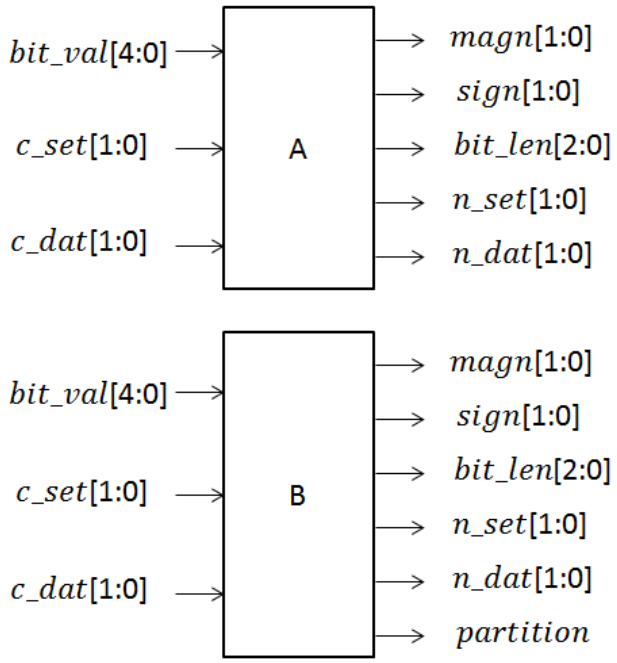


Figure 5.6: Two types of Processing Elements in Decoder

5.2.3. Gate Counts

The hardware cost for both implementations are presented in Table 5.1 (for the encoder) and 5.2 (for the decoder). The hardware model is programmed in Verilog HDL and Synopsys Design Compiler is used for netlist generation. To evaluate the cost of the proposed hardware implementation, the Verilog models are synthesized with 0.13um library for ASIC fabrication.

Table 5.1 presents the gate count comparison between two implementations in encoder. The gate count of the former NLS is 48.6K, where DWT, C2B, SPIHT and TRANS account for 15.85%, 45.66%, 11.13% and 27.35 %, respectively. The non-combinational logics inside C2B, TRANS and SPIHT account for 28.15%, 4.07% and 11.15% of the total hardware cost of encoder, respectively. In total, those non-combinational logics account for 21.1K or 43.37% of the encoder hardware cost. Note that the non-combinational logics consist of the buffer size and internal registers for temporary calculation.

The hardware costs of decoders are presented in Table 5.2. The NLS implementation requires 63.0K gate counts. Note that the gate counts of decoder are generally larger than encoder's because the shift operation is the main operation to update accumulation buffers in decoder. In decoder, bit-planes of coefficients are decoded in sequence and the input length is unknown. As a result, the shift operation is used to assign inputs for each processing element. In decoder, the gate count of non-combinational logic in B2C, iSPIHT and Parser is 18.3K which accounts for 29.10% of the total hardware cost of decoder. In the proposed design, the gate count of non-combinational logic in these modules is 4.9K.

Table 5.1: Hardware gate count of the NLS encoder

Module	Total	Registers	%
DWT	7.7K	3.6K	31.23
C2B	22.2K	13.7K	33.34
SPIHT	5.4K	2.0K	12.93
Aligner	13.3K	5.4K	22.49
Total	48.6K		100.00

Table 5.2: Hardware gate count of the NLS decoder

Module	Total	Registers	%
DWT	13.8K	4.8K	31.23
C2B	27.6K	10.3K	33.34
SPIHT	18.0K	5.7K	12.93
Aligner	3.5K	2.3K	22.49
Total	63.0K		100.00

Tables 5.3 shows the gate counts of the encoder with the fixed TBL and Table 5.4 presents those with the adjusted TBL. When compared with the hardware cost of the conventional implementation shown in Tables 2, the non-combinational logic gates of the encoder is reduced by 60.45%, In total, the hardware cost is reduced by 50.40%. For the adjusted TBL shown in Table 7, an extra amount of hardware resource is necessary. As a result, the total hardware cost of the encoder is 26.7K gates, archiving a 45.06% reduction.

Table 5.3: Hardware gate count of the proposed encoder with the fixed TBL

Module	Total	Registers	%
DWT	7.7K	3.6K	31.23
C2B	8.2K	6.0K	33.34
SPIHT	3.2K	0.8K	12.93
Aligner	5.5K	1.6K	22.49
Total	24.6K	12.0K	100.00

Table 5.4: Hardware gate count of the proposed encoder with the adjusted TBL

Module	**Total**	**Registers**	**%**
DWT	7.7K	3.6K	28.73
C2B	8.2K	6.0K	30.67
SPIHT	3.2K	0.8K	11.90
Aligner	7.7K	2.6K	28.69
Total	26.7K	13.0K	100.00

Tables 5.5 and 5.6 show the hardware costs of the proposed decoder designs with the fixed TBL and adjusted TBL, respectively. When compared with the hardware cost in Table 3, the gate count is reduced by 73.63% with the fixed TBL. Similar to the encoder, TBL adjustment requires an extra resource for parsing the bit-stream for each sub-block. As a result, the hardware cost of the decoder is 34.3K, archiving a 45.56% reduction.

Table 5.5: Hardware gate count of the proposed decoder with the fixed TBL

Module	Total	Registers	%
iDWT	13.8K	4.8K	45.10
iSPIHT	8.9K	1.5K	29.16
B2C	7.0K	2.7K	22.74
Parser	0.9K	0.6K	2.99
Total	30.5K	9.6K	100.00

Table 5.6: Hardware gate count of the proposed decoder with the adjusted TBL

Module	Total	Registers	%
iDWT	13.8K	4.8K	40.38
iSPIHT	9.1K	1.5K	26.41
B2C	7.0K	2.7K	20.36
Parser	4.4K	2.6K	12.85
Total	34.3K	11.6K	100.00

Chapter VI CONCLUSION

In this thesis, we have proposed a cost-effective algorithm based on DWT and SPIHT. In order to reduce buffer size and hardware cost, three modifications are made. The first modification is to merge three passes in the NLS algorithm into one loop. This modified algorithm generates the same length of bit stream for every bit-plane if no data is discarded. In other word, numbers of bit-stream length are same for the lossless case. For the lossy compression, it slightly degrades the PSNRs by 0.07dB (from -0.39 to 0.74 dB), as shown in experimental results. In the contrary, it simplifies the SPIHT engine and only two types of processing units are used. It also allows defining exactly the coding break position which is used in the bit-allocation scheme. The second modification is the data structure reorganization (sub-block based) to perform SPIHT coding for each sub-block independently. Instead of processing all block, it only performs each sub-block that is much smaller size. As a result, it significantly reduces buffer sizes storing temporary data in all coding modules. The experimental results show that this modification reduces hardware cost by 45.44% when a 1x64 block is partitioned into four 1x16 sub-blocks. Since DWT module is still applied for a block, the coding efficiency of independent sub-block coding improves by 0.60dB than the algorithm when DWT only performs a small block size. The third modification is the bit-allocation scheme to improve the coding efficiency. In the proposed bit-allocation, the relation of bit-stream lengths among sub-blocks is exploited. The complexity of each sub-block is presented by a weight calculated from the coding break information. Based on it, each sub-block is

assigned with an appropriate bit-stream length. The bit-allocation scheme only performs at packet module, so that it doesn't violate the independent sub-block coding. As a result, the hardware reduction still can be archived.

The hardware implementation of the partitioned SPIHT with Verilog HDL achieves the throughput of 484Mbps. The required gate count is 61.0K, archiving 45.44% reduction when compared with the conventional implementation of the NLS. To reduce a degradation of the compression efficiency by the partitioned coding scheme, the target bit length of each sub-block is adjusted according to its predicted complexity. Experimental results with 18 test images show that the PSNR is dropped by 1.30dB when compared with the conventional non-partitioned compression. The proposed algorithm is implemented in hardware which requires 26.7K gates for an encoder and 34.3K gates for a decoder.

In conclusion, three modifications make a cost-effective design of a SPIHT-based algorithm. The relations among hardware reduction, coding efficiency and bus bandwidth are carefully considered in the design. The design is applicable in both one-dimensional and two-dimensional cases. The method as bit-allocation is optionally integrated when the high coding efficiency is required. It results in the more complex circuitry in the Packer module, which limits overall hardware cost. Another drawback of the design is the limited compression ratio. However, the design is applicable with temporary storage applications when small compression ratios still significantly reduces the off-chip memory storage and bus bandwidth Overall, the proposed design makes a good tradeoff between the hardware cost reduction and the compression efficiency.

REFERENCES

[1] A. Said and W. Pearlman, “A new, fast, and efficient image codec based on set partitioning in hierarchical trees,” *IEEE trans. Circuits Syst. Video Technol*., Vol. 6, no. 3, pp. 243-250, Jun.1996

[2] F. Wheeler and W. Pearlman, “SPIHT image compression without lists,” *in. Proc. IEEE ICASSP*, Vol. 4, pp. 2047- 2050, Jun.2000

[3] P. Coronello, S. Perri, G. Staino, M. Lanuzza, and G. Cocorullo, “Low bit rate image compression core for onboard space applications,” *IEEE Trans. Circuits Syst. Video Technol*., Vol. 16, No. 1, pp. 114-128, Jan. 2006

[4] C.-C. Cheng, P.-C. Tseng, L.-G. Chen, “Multimode Embedded Compression Codec Engine for Power-Aware Video Coding System,” *IEEE Trans. Circuits and System for Video Technology,* vol. 19, No.2, pp. 141-150, Feb. 2009.

[5] T. Fry and S.Hauck, “SPIHT image compression on FPGAs,” *IEEE Trans. Circuits Syst. Video Technol.*, Vol. 15, no.9, pp. 1138-1147. Sep. 2005.

[6] Y. Jin, H.-J.Lee, “A block-based pass-parallel SPIHT algorithm,” *IEEE Trans. On Circuits Syst. Video Technol*. Vol. 22, No. 7, pp. 1064-1075, July 2012.

[7] *JPEG2000 Image Coding System,* document ISO/IEC 15444-1, 2000.

[8] J. M. Shapiro “Embedded Image Coding using Zerotrees of Wavelet Coefficients“, *IEEE Trans. Signal Processing*, Vol.41, No.12, Dec. 1993

[9] D. Taubman, “High Performance Scalable Image Compression with EBCOT”, *IEEE Trans. Image Processing* Vol. 9, No.7, pp.1158-1170, July, 2000.

[10] D. Le Gall, A. Tabatabai, “Sub-band coding of Digital Images Using

Symmetric Short Kernel Filters and Arithmetic Coding Techniques", *in Proc. Int. Conf. Acoustics, Speech, and Signal Processing, (ICASSP-88)*, 1988, Vol.2, pp. 761-764.

[11] P.Y. Chen, "VLSI Implementation for 1-D Multilevel Lifting-Based Wavelet Transform", *IEEE Transactions on computers*, VOL. 53, NO. 4, APRIL 2004

[12] T. Y. Lee, "A new frame-recompression and its hardware design for MPEG-2 video decoders", *IEEE Trans. Circuits Syt. Video Tech.*, Vol. 13, No.6, pp 529-534. Jun. 2003.

[13] Y. Jin, Y. Lee, H.-J Lee, "A new frame memory compression algorithm with DPCM and VLC in 4x4 block," *EURASIP J. Adv. Signal Process.*, Vol. 2009, No. 629285, 2009.

[14] W.Y.Chen, L.F.Ding, P.K.Tsung, and L.G.Chen, "Architecture design of high performance embedded compression for high definition video coding", *in Proceedings of the IEEE International Conference on Multimedia and Expo (ICME '08)*, pp. 825–828, Hannover, Germany, June 2008

[15] T. Song and T. Shimamoto, "Reference frame data compression method for H.264/AVC,"*IEICE Electronics Express*, vol. 4, no. 3, pp. 121–126, 2007

[16] T.B. Yng, B.-G. Lee, and H. Yoo, "A low complexity and lossless frame memory compression for display devices", IEEE Trans. Consumer Electronics, Vol.54, No.3, pp 1453-1458, Aug. 2008.

[17] Y. Jiang, Y. Li, D. Ban, Y. Xu, "Frame buffer compression without Color Information Loss", *IEEE 12th International Conf. Computer and Information Tech.*, pp.12-17, Oct. 2012.

Abstract

Set Partitioning In Hierarchical Trees (SPIHT) is one of most popular embedded coding algorithms applied for wavelet coding images. It allows progressive transmission of information and gives high coding efficiency. In addition, it can omit entropy coding of the bit stream by arithmetic code with only small loss in performance. Thus it offers a cheaper and faster hardware design. In this dissertation, a cost-effective design of a SPIHT-based algorithm is proposed. In this algorithm, an image is partitioned into 1x64 blocks, each of which is transformed by DWT to generate wavelet coefficients. The wavelet coefficients are coded by SPIHT to generate bit-stream. Due to the mismatch of the data structure between DWT and SPIHT, the large buffers are required. In order to reduce buffers, a new data structure of wavelet coefficients and partitioned SPIHT are proposed. A wavelet-based block is partitioned into small sub-blocks each of which is compressed independently. To minimize distortion due to the sub-block-based compression, a bit-allocation scheme is proposed.

The proposed design is implemented in both software and hardware. Experimental results show that the proposed design reduces the buffer size while minimizing the degradation of the rate-distortion performance. It is proved that the proposed design outperforms previous designs in hardware cost.

주요어 : **Image coding, wavelet image compression, SPIHT**

학 번 : **2012-23954**

초록

Set Partitioning In Hierarchical Trees (SPIHT) 은 웨이블릿 압축에 있어 가장 잘 알려진 임베디드 압축 알고리즘 중의 하나이다. 이 방식은 정보의 점진적 전송을 가능케 하고, 높은 압축 효율을 제공한다. 뿐만 아니라, 적은 화질 저하만으로 binary arithmetic coding 을 이용하지 않을 수 있으므로 값이 싸면서 더 빠른 하드웨어 설계를 가능케 한다. 본 논문에서는 비용 효율이 높은 SPIHT 기반의 알고리즘을 제안하도록 한다. 해당 알고리즘에서는 이미지가 1x64 블록들로 나누어지며, 각각의 블록들은 DWT 를 통해 웨이블릿 계수로 변형된다. 변형된 웨이블릿 계수들은 SPIHT 을 통해 압축되며, 이를 통해 압축 비트 스트림이 생성된다. 이때, DWT 와 SPIHT 의 자료 구조가 일치하지 않기 때문에 크기가 큰 버퍼가 필요로 하게 된다. 본 논문에서는 해당 버퍼 크기를 감소시키기 위해서 새로운 형태의 자료 구조를 제안하였다. 제안한 방식에서는 웨이블릿 기반의 블록이 더 작은 서브 블록들로 나누어진다. 그런 다음 각각의 서브 블록들은 독립적으로 압축된다. 본 논문에서 제안한 방식으로 인한 화질 감소를 최소화시키기 위하여 새로운 비트 할당에 대한 방법도 제안하였다.

제안된 방식은 소프트웨어와 하드웨어로 구현되었다. 실험 결과를 통해 제안한 구조가 화질 저하를 최소화하면서 버퍼 크기를 감소시키는 것을 확인할 수 있다. 또한 제안한 방식은 기존 방식에 비해 하드웨어 비용이 절감됨을 확인할 수 있다.

주요어 : Image coding, wavelet image compression, SPIHT

학 번 : 2012-23954