A CODING METHOD FOR UVLC TARGETING EFFICIENT DECODER ARCHITECTURE

Shang Xue and Bengt Oelmann

Department of Information Technology and Media, Mid Sweden University
SE-851 70 Sundsvall, Sweden
xue.shang@mh.se

ABSTRACT

Variable length code (VLC) is used in a large variety of lossless compression applications. A specially designed VLC, called "Universal Variable Length Code" (UVLC), is utilized in the latest video coding standard H.26L under development. In this work we propose a coding method that we call "Alternating Coding" (ALT coding) for UVLC. And an efficient UVLC decoder is designed (ALT decoder) on the basis of it. ALT coding facilitates a much easier decoding scheme for UVLC. It frees the decoder from codeword tables and the sizes of the barrel shifters are reduced. The codeword tables and barrel shifters usually occupy the largest portion of the area in the traditional VLC decoders and they are also performance limiting in terms of speed and power consumption. We compare the ALT decoder with one of the most efficient VLC decoders called "VLC decoder using plane separation" (PLS). Our results show that the ALT decoder increases 25% in speed, decreases 41% in size, and consumes 45% power of the PLS decoder.

1. INTRODUCTION

Image and video coding standards all utilize entropy coding in the form of variable length codes (VLCs) for their efficient compression. The video coding standard H.26L utilizes a unique VLC pattern which is called Universal Variable Length Code (UVLC) to perform entropy coding [4]. UVLC was first proposed in [1]. In [1,2,3] it is suggested to be used in the coding of motion vectors as well as DCT coefficients for H.26L. UVLC is claimed to be able to provide good performance in terms of coding efficiency, configurability to various applications and error resiliency. Although UVLC is efficient in compression, the variable code length also limits the decoding throughput. The decoding process needs to identify the codeword boundaries, each of which depends recursively on the previous codeword boundary. In this paper we propose a new coding method, called "Alternating Coding" (ALT coding), which is applied to UVLC. It enables the extraction of the special properties of UVLC and facilitates it with ability of immediate codeword boundary detection. In addition, by ALT coding method, the decoding procedure of UVLC can be simplified to computational operations instead of codeword table searching. Therefore the decoder can be greatly simplified.

VLC decoders are usually implemented by using look-up tables and a shifting scheme [6,7]. All possible codewords and codeword lengths need to be stored in look-up tables so that they can be matched out according to the input data. The shifting scheme shifts the input data according to the codeword lengths in order to perform decoding continuously. The codeword tables can be implemented with ROM or PLA and the shifting scheme is usually implemented with barrel shifters. These two parts in a VLC decoder occupy the largest portion of the area and as they are the two crucial parts in determining the codeword boundaries, they are both included in the critical timing path of the decoder. Look-up tables and barrel shifters are therefore the performance limiting components in a VLC decoder. UVLC belongs to the VLC family. Therefore it is straight-forward to implement a UVLC decoder by using the existing architectures for decoding general VLCS, i.e. using look-up tables and a shifting scheme.

In this paper we present a new type of UVLC decoder based on the ALT coding method. It does not contain look-up tables, and the sizes of barrel shifters are greatly reduced. Therefore it is faster, much smaller and less power-consuming. With the development in mobile video communications, the construction of smaller, faster, and less power-consuming video CODECs becomes increasingly important. In the paper, we compare the performances of the proposed UVLC decoder with a decoder developed by Jae Ho Jeon et al. [8], under the name of "Fast Variable-Length Decoder Using Plane Separation" (PLS), which was claimed to be one of the most effective VLC decoders. We compare the ALT decoder to the PLS decoder in delay, area and power consumption. Our results show that the ALT decoder is 1.34 times faster, 1.7 times smaller, and consumes 45% power in comparison to the PLS decoder.

The outline of this paper is as follows. First the coding method, "Alternating Coding", for UVLC is described. Then the ALT decoder is presented. After that we present a comparison of performance of the ALT decoder to the PLS decoder. Finally we draw some conclusions.

2.ALTERNATING CODING

Table 1 gives an example of UVLC [1].

Table 1. An example of UVLC

Class	Coarse	Additional	UVLC		Value to be
k	code	code	Codeword	Length	expressed
1	1	None	1	1	1
2	00	x_0	$0x_00$	3	‘x_0’+ 2[2:3]
3	010	x_1x_0	$0x_11x_00$	5	‘x_1x_0’+ 4[4:7]
4	0110	$x_2x_1x_0$	$0x_21x_11x_00$	7	‘$x_2x_1x_0$’+ 8[8:15]
5	01110	$x_3x_2x_1x_0$	$0x_31x_21x_11x_00$	9	‘$x_3x_2x_1x_0$’+ 16[16:31]
6	011110	$x_4x_3x_2x_1x_0$	$0x_41x_31x_21x_11x_00$	11	‘$x_4x_3x_2x_1x_0$’+ 32[32:63]

For each codeword we can look at the odd-indexed bits (OIB) and the even-indexed bits (EIB) separately. The OIB form a fixed pattern “011..10”. The EIB, as indicated by x_n in Table 1, is an arbitrary binary code.

The OIBs of the UVLCs can be looked on as a set of unary codes whose length represents the code class in Table 1. The EIBs are a set of binary codes whose length can be determined by their corresponding odd-indexed parts, i.e. one bit shorter than OIB. The idea of alternating coding is to split the unary parts and the binary parts of a codeword, to encode and transmit them separately. In coding the unary OIBs, two different sets of codes are applied, one is {0,00,000, ..., 0000...0}, another is {1,11,111, ..., 1111...1}. They are used alternatingly in the coding procedure. The binary EIBs are kept the same. As the length of each EIB can be calculated by its corresponding odd-indexed part, it can be simply deemed as fixed-length codes and can be easily decoded once the OIB is decoded. For example, if we have a UVLC series 001 00011 011 01011 , after applying the alternating coding, we will get one OIB sequence and one EIB sequence. The OIB sequence will be 11 000 11 000 , and the EIB sequence will be 0 01 1 10 . The ALT coded series will be transmitted separately as: 11 000 11 000 0 01 1 10 . Figure 1 shows how ALT coding is applied for UVLC.

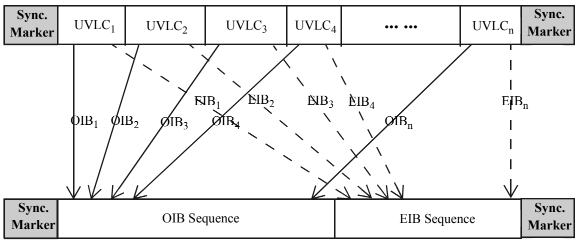

Fig. 1: ALT coding for UVLC

Separating the OIB sequence and the EIB sequence and applying two codeword tables for the OIBs provide two advantages in the decoding procedure:

1. The codeword boundaries can be easily determined by detecting the value changes in an OIB series.

2. The coded value can be calculated as:

$$code\ number = 2^{length(OIB)-1} + decimal(EIB) \qquad (1)$$

Here *length(OIB)* means the number of bits an OIB has.

These advantages will enable simplification of the decoding procedure.

Tu-Chih Wang et al. proposed an efficient UVLC encoder in [5]. In their design, the encoding procedure is simplified by modifying the code values first. Table 2 shows the modified code numbers and the corresponding codewords of UVLC. It can be seen that the EIBs, i.e. the underlined bits in UVLC, and the underlined bits in the binary expressions of the modified code numbers are exactly the same. Encoding is then done by inserting the fixed OIB pattern in between the bits of the binary code. Their encoder design [5] uses a “Code Splitter” to achieve the insertion of the OIB. The “Code Splitter” is simply a wiring box and does not contain any gates. The encoder using the ALT coding method can then be easily modified from the existing encoder by eliminating the “Code Splitter”, and outputting the two parts separately though some buffering is needed before the whole UVLC sequence is generated.

Table 2. Modified code table

Code Number	Modified Code Number	Binary	UVLC
0	1	00001	1
1	2	$0001\underline{0}$	$0\ \underline{0}\ 1$
2	3	$0001\underline{1}$	$0\ \underline{1}\ 1$
3	4	$001\underline{00}$	$0\ \underline{0}\ 0\ \underline{0}\ 1$
4	5	$001\underline{01}$	$0\ \underline{0}\ 0\ \underline{1}\ 1$
5	6	$001\underline{10}$	$0\ \underline{1}\ 0\ \underline{0}\ 1$
6	7	$001\underline{11}$	$0\ \underline{1}\ 0\ \underline{1}\ 1$
7	8	$01\underline{000}$	$0\ \underline{0}\ 0\ \underline{0}\ 0\ \underline{0}\ 1$
8	9	$01\underline{001}$	$0\ \underline{0}\ 0\ \underline{0}\ 0\ \underline{1}\ 1$

3.ALT DECODER

To fully exploit the advantages facilitated by ALT coding, the decoding of an ALT coded UVLC packet demands processing the OIB sequence and the EIB sequence separately. This requires a separation of the OIB and EIB sequences upon receiving the whole UVLC packet. Therefore buffering the UVLC packet before decoding is needed. However, as buffering is normally needed in image/video CODECs, this will not bring extra cost to the decoder.

When the number of source symbols contained in a packet is known to be *N* (such as for motion vectors and DCT coefficients in H.263), the packet can be easily separated into an OIB sequence and an EIB sequence.

Let *N* be the number of codewords in the packet, and *L* be the packet length. Let l_{OIB} represent the length of the OIB sequence and l_{EIB} represents the length of the EIB sequence. We have:

$l_{EIB} = l_{OIB} - N$, and

$L = l_{EIB} + l_{OIB} = 2l_{OIB} - N$.

Therefore, we have:

$l_{OIB} = \frac{(L+N)}{2}$ and $l_{EIB} = \frac{(L-N)}{2}$.

In order to make the ALT decoder comparable with general VLC decoders, in this paper, we ignore the peripheral architectures such as the input buffers or the OIB-EIB separation logic into consideration. This is still fair as when general VLC CODECs are applied in image/ video decoding, different peripheral architectures need to be implemented anyway.

The ALT decoder proposed in this paper is based on the ALT coding. The maximum codeword length is set to be 31 bits in order to cover adequate number of codewords.

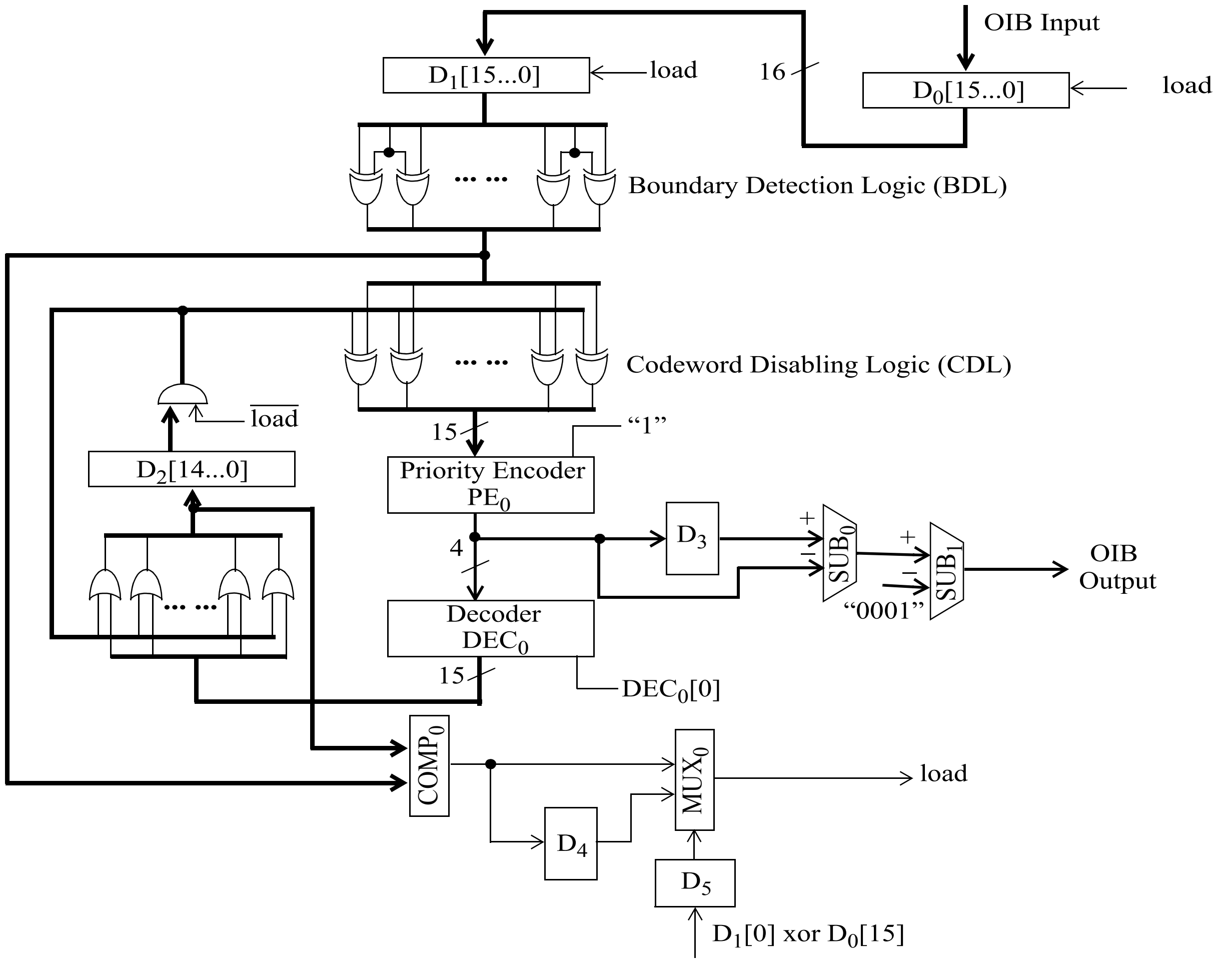

Fig. 2: OIB decoder

The ALT decoder consists of two decoders, one is the OIB decoder and the other is the EIB decoder. The architecture of the OIB decoder is described in Figure 2. Its function is to generate the length of each OIB, offset it to get the length of each EIB and provide it as a reference in decoding the actual UVLC. Having in mind that the maximum codeword length is 31 bits, the maximum length for the OIB is then set to be 16 bits. The decoder consists of one 16-to-4 priority encoder (PE_0), one 4-to-16 decoder (DEC_0), two 16-bit buffers (D_0 and D_1), one 15-bit register D_2, one 4-bit register D_3, one 15-bit comparator ($COMP_0$), two 4-bit subtractors (SUB_0 and SUB_1), one 1-bit 2:1 multiplexer (MUX_0), and two 1-bit registers (D_4 and D_5).

The OIB input of the decoder is put into the two buffers D_0 and D_1, the first two bytes in D_1 and the second two bytes in D_0. The first two-byte OIB series is then fed to the xor-gates in the "Boundary Detection Logic" (BDL) where two consecutive bits are xored with each other. As the OIBs are now denoted in alternating all-one and all-zero codes, only at each OIB boundary a “1” will be generated by the xor operations. Therefore, each “1” indicates an OIB boundary. The output after the BDL is then fed into the priority encoder PE_0 in order to generate the position of the first OIB boundary. Register D_3 is originally loaded with the number 16 (that is “0000” in a 4-bit binary code). The length of the first OIB is then calculated by SUB_0 and at the same time D_3 is updated with the position of the first OIB boundary. The 4-to-16 bit decoder DEC_0 generates the position of the first OIB boundary and disables the first “1” of the input of the priority encoder by using the or-gates and the “Codeword Disabling Logic” (CDL). In the next clock cycle, the second OIB boundary is encoded by PE_0. Again the second OIB boundary is put to D_3 and its position is decoded by DEC_0. The same operations are then repeated. Thus the length of each OIB is generated. By using another subtractor SUB_1, the output of SUB_0 is offset by one. The length of the corresponding EIB is then generated and put to the OIB output.

Assume we have a series of values to be coded using the UVLC in Table 1 and the values are {3, 9, 1, 1, 1, 4, 1, 26, 15, 1, 1, 1, 1, 1, 2, 12, ...}. Then the OIB sequence will be {00, 1111, 0, 1, 0, 111, 0, 11111, 0000, 1, 0, 1, 0, 1, 00, 1111, ...} and the EIB sequence will be {1, 001, 00, 1010, 111, 0, 100, ...} respectively.

Table 3 illustrates how the OIB decoder decodes the above OIB sequence. Suppose the OIB decoder was initialized to all zeros before D_1 is loaded with data. When “load” is set to high, D_1 and D_0 are loaded with “0011110101110111” and “1100001010100111” respectively. Then $D_1[0]$xor$D_0[15]$ is set to low, which indicates the last codeword in D_1 continues in D_0.

Table 3. Example of the decoding procedure of OIB decoder

Clock cycle 0	D_1_out	0011110101110111	PE_0_out	1110
	BDL_out	010001111001100	D_3_out	0000
	CDL_out	010001111001100	SUB_0_out	0010
	DEC_0_out	0100000000000000	OIB_out	0001
	D_2_in	010000000000000	$COMP_0$_out	0
	D_2_out	000000000000000	load	0
Clock cycle 1	D_1_out	0011110101110111	PE_0_out	1010
	BDL_out	010001111001100	D_3_out	1110
	CDL_out	000001111001100	SUB_0_out	0100
	DEC_0_out	0000010000000000	OIB_out	0011
	D_2_in	010001000000000	$COMP_0$_out	0
	D_2_out	010000000000000	load	0
Clock cycle 2	D_1_out	0011110101110111	PE_0_out	1001
	BDL_out	010001111001100	D_3_out	1010
	CDL_out	000000111001100	SUB_0_out	0001
	DEC_0_out	0000001000000000	OIB_out	0000
	D_2_in	010001100000000	$COMP_0$_out	0
	D_2_out	010001000000000	load	0
...	... ...			
Clock cycle 6	D_1_out	0011110101110111	PE_0_out	0011
	BDL_out	010001111001100	D_3_out	0100
	CDL_out	000000000000100	SUB_0_out	0001
	DEC_0_out	0000000000001000	OIB_out	0000
	D_2_in	010001111001100	$COMP_0$_out	1
	D_2_out	010001111001000	load	1
Clock cycle 7	D_1_out	1100001010100111	PE_0_out	1110
	BDL_out	010001111110100	D_3_out	0011
	CDL_out	010001111110100	SUB_0_out	0101
	DEC_0_out	0100000000000000	OIB_out	0100
	D_2_in	010000000000000	$COMP_0$_out	0
	D_2_out	000000000000000	load	0
...	Decoding continues			

Table 3. Example of the decoding procedure of OIB decoder

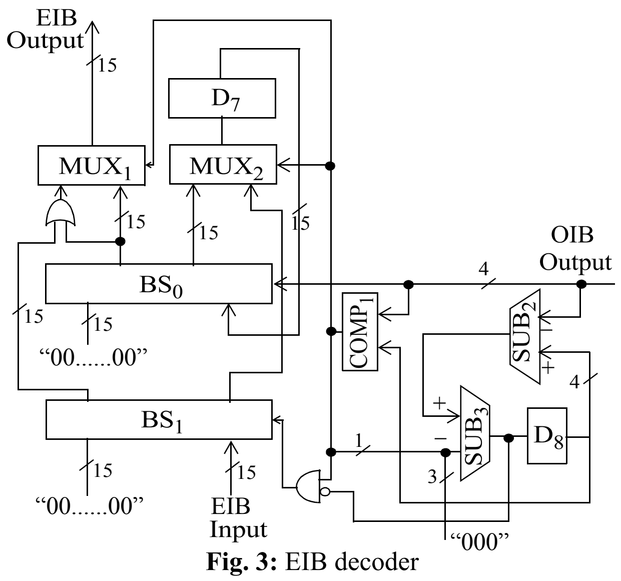

Fig. 3: EIB decoder

The EIB decoder is illustrated in Figure 3. It contains one 15-bit registers (D_7), one 4-bit register D_8, two 15-bit 2:1 multiplexers (MUX_1 and MUX_2), two 30-bit barrel shifters (BS_0 and BS_1), two 4-bit subtractors (SUB_1 and SUB_2), and a 4-bit greater than & equality comparator ($COMP_1$). The EIB series is first loaded in the lower half of the two barrel shifters, the first 15 bits in BS_0 and the following 15 bits in BS_1. The upper half of the barrel shifters are both loaded with 15 bits zeros. D_8 is originally loaded with 15 (“1111” in binary), which is the maximum EIB length. BS_0 shifts the EIB series to the upper half of it according to the first EIB length generated from the OIB decoder. The first EIB is then generated from the upper half of BS_0. At the same time, SUB_2 outputs the length of the rest of the EIB series after the first EIB has been shifted out. This length is stored in D_8, to be used for the decoding of the next EIB. In the next clock cycle, the lower half of BS_0 is loaded with the shifted EIB series and the upper half is cleared into all zeros. Therefore the decoding of the next EIB can be performed. The same operations are then repeated. When EIB decoding is performed till the end of the first 15 bits, the length of the EIB series left in BS_0 will be equal to or smaller than the length of the next EIB. This will make the output of $COMP_1$ become "1". A new 15-bit EIB sequence will then be loaded to the EIB input. The contents of BS_0 and BS_1 are both shifted according to the length of the next EIB. The two separated parts of the last EIB in BS_0 can be merged by the or-gates and MUX_1 so that the complete EIB can be generated. MUX_2 is used to load new data into BS_0. Decoding can then be performed continuously.

Take the same example we used for illustrating the OIB decoder. Now we have an EIB sequence {10010010101110100......} to decode. Table 4 shows how decoding is performed in the EIB decoder.

Table 4. Example of the decoding procedure of EIB decoder

Clock cycle 1	BS_1_out	000000000000000$\mathbf{X_{14}X_{13}}$ **… …** $\mathbf{X_0}$	OIB_out	0001
	BS_0_out	000000000000000**100100101011101**0	$COMP_1$_out	0
	D_7_out	000000000000000	SUB_2_out	1110
	EIB_out	000000000000001	SUB_3_out	1110
Clock cycle 2	BS_1_out	000000000000000$\mathbf{X_{14}X_{13}}$ **… …** $\mathbf{X_0}$	OIB_out	0011
	BS_0_out	0000000000000**00100101011101**0000	$COMP_1$_out	0
	D_7_out	001001010111010	SUB_2_out	1100
	EIB_out	000000000000001	SUB_3_out	1011
Clock cycle 3	BS_1_out	000000000000000$\mathbf{X_{14}X_{13}}$ **… …** $\mathbf{X_0}$	OIB_out	0000
	BS_0_out	0000000000000000**00101011101**0000	$COMP_1$_out	0
	D_7_out	001010111010000	SUB_2_out	1100
	EIB_out	000000000000000	SUB_3_out	1011
…	**… …**			
Clock cycle 15	BS_1_out	000000000000000$\mathbf{X_{14}X_{13}}$ **… …** $\mathbf{X_0}$	OIB_out	0001
	BS_0_out	000000000000000**01**00000000000000	$COMP_1$_out	0
	D_7_out	010000000000000	SUB_2_out	0010
	EIB_out	000000000000000	SUB_3_out	0001
Clock cycle 16	BS_1_out	00000000000$\mathbf{X_{14}X_{13}}$ **… …** $\mathbf{X_0}$00	OIB_out	0011
	BS_0_out	000000000000100$\mathbf{X_{12}X_{11}}$ **… …** $\mathbf{X_0}$00	$COMP_1$_out	1
	D_7_out	100000000000000	SUB_2_out	1100
	EIB_out	0000000000001$X_{14}X_{13}$	SUB_3_out	0010
…	Decoding continues … …			

In Table 4, X_{14}...X_0 indicates the new data coming after the first 15 bits. In our example, $X_{14}X_{13}$ actually are "00".

The complete architecture of the ALT decoder is shown in Figure 4. The decoded EIB length, i.e. the OIB Output, is put to a code converter. The truth table of the code converter is given in Table 5.

Table 5. Truth table of the Code Converter

Input (4 bit)	Output (16 bits)	Output in decimal
0000	0000000000000000	2^0
0001	0000000000000001	2^1
0010	0000000000000011	2^2
0011	0000000000000111	2^3
0100	0000000000001111	2^4
0101	0000000000011111	2^5
0110	0000000000111111	2^6
0111	0000000001111111	2^7
1000	0000000011111111	2^8
1001	0000000111111111	2^9
1010	0000001111111111	2^{10}
1011	0000011111111111	2^{11}
1100	0000111111111111	2^{12}
1101	0001111111111111	2^{13}
1110	0011111111111111	2^{14}
1111	0111111111111111	2^{15}

The function of this code converter is to change the length of the EIB (i.e. *length(OIB)-1*) into $2^{length(EIB)}$ so that the code number can be calculated by a simple addition operation using the result of the code converter, as mentioned in relation (1). Therefore, the output of the code converter is added to the output of the decoded EIB using the 16-bit adder (ADD) to generate the actual code number. The decoding of a UVLC is then completed.

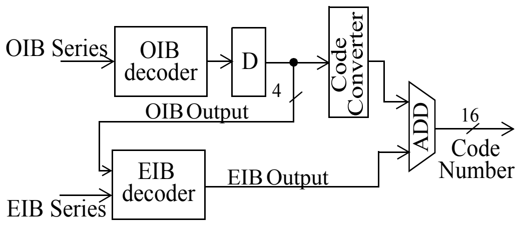

Fig. 4: ALT decoder

In this ALT decoder, no look-up tables are needed and the size of the shifting scheme is greatly reduced. It is capable of decoding one codeword per clock cycle.

4.COMPARISON OF PERFORMANCE

The ALT decoder is compared with the PLS decoder developed by Jae Ho Jeon et al.[8] shown in Figure 5.

The original structure in [8] was designed for VLC that has a maximum codeword length of 16 bits. Here we reconfigure it for UVLC whose maximum codeword length is 31 bits. The decoder consists of two separate planes. For UVLC, each plane consists of a 62-bit barrel shifter, a 62-bit 2:1 multiplexer, and a 62-bit output register. The codeword table in this case is loaded with a UVLC codeword table and so is the code length table. This decoder is capable of decoding one codeword per clock cycle and the design makes the coding process parallel by using an "or plane". However, feeding the codeword length from the look-up tables back to the barrel shifters still limits the decoding throughput. All the possible codewords, codeword lengths and decoded code numbers need to be implemented in the look-up tables, and two types of barrel shifters are included, both of them are large in size. These all limit the efficiency of the PLS decoder. According to our synthesis results, look-up tables and barrel shifters take as much as 75% of the total area of the PLS decoder.

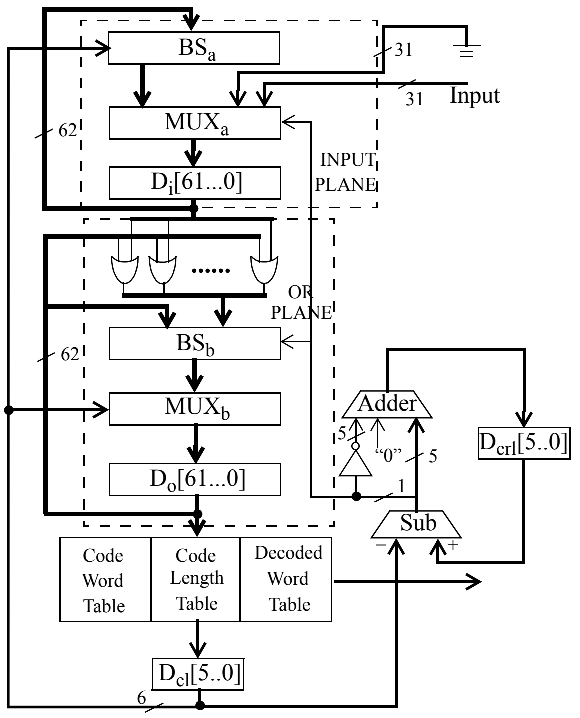

Fig. 5: PLS decoder

We compare the delay, area and power consumption of the ALT decoder to those of the PLS decoder. Both types of decoders are implemented in VHDL and synthesized using *Design Compiler* from Synopsys. The delay has been obtained from static timing analysis and the figures for power consumption from Synopsys' *Power Compiler.* A standard cell library in a 0.5µm CMOS process has been used. The results are shown in Table 6.

The exceeding performance of the ALT decoder is evident compared to that of the PLS decoder in all factors: speed, area and power. The reduction of size and power consumption of the ALT decoder is due to the elimination of huge codeword tables and code length tables and the reduction of the size of the shifting scheme in the conventional VLC decoders. This is also part of the reason why the ALT decoder increases in speed. Another factor for the speed increase of the ALT decoder is because the coding procedure is parellelized by separating the decoding of OIBs and EIBs.

Table 6. Comparison of performance

	ALT	PLS	Ratio (ALT/PLS)
Delay (ns)	8.96	12.0	75%
Area (gates)	1855	3146	59%
Power (mW)	6.74	15.0	45%

5.CONCLUSIONS

We propose the ALT decoder for decoding UVLC. This decoder is based on a coding method that we call "Alternating Coding". The ALT coding provides conveniences in the decoding of UVLC which enables efficient decoder design. It can be seen that the ALT decoder increases 25% in speed, decreases 41% in size, and consumes 45% power compare to the PLS decoder, while the PLS decoder is declared to be one of the best decoders for variable length codes. This makes it a very strong motivation for UVLCs to be encoded and decoded using the "Alternating Coding" method.

6. REFERENCES

[1] Y. Itoh, "Bi-directional motion vector coding using universal VLC," *Signal Processing: Image Communication*, vol. 14, pp. 541-557, May 1999.

[2] Y. Itoh, Ngai-Man Cheung, "Universal variable length code for DCT coding," in *International Conference on Image Processing*, vol. 1, pp. 940-943, 2000.

[3] N.-M. Cheung, Itoh, Y., "Configurable variable length code for video coding," in *International Conference on Acoustics, Speech, and Signal Processing*, vol. 3, pp. 1805-1808, 2001.

[4] ITU-T, *H.26L TML8 Document from http://standard.pictel.com*, Sep., 2001.

[5] Tu-Chih Wang, Hung-Chi Fang, Wei-Min Chao, Hong-Hui Chen, Liang-Gee Chen, "An UVLC encoder architecture for H.26L," in *IEEE International Symposium on Circuits and Systems*, vol. 2, pp. 308-311, 2002.

[6] M. T. Lei, M. T. Sun, "An entropy coding system for digital HDTV applications," in *IEEE Trans. Circuits Syst. Video Technol.*, vol. 1, no. 1, pp. 147-155, March 1991.

[7] H. D. Lin, D. G. Messerchmitt, "Designing high-throughput VLC decoder Part II-Parallel decoding methods", in *IEEE Trans. Circuits Syst. Video Technol.*, vol. 2, pp. 197-206 June 1992.

[8] Jae Ho Jeon et al, "A fast variable-length decoder using plane separation," in *IEEE. Trans. Circuits Syst. Video Technol.*, vol. 10, pp. 806-812, Aug. 2000.