Published in IET Science, Measurement and Technology
Received on 29th July 2011
Revised on 26th December 2011
doi: 10.1049/iet-smt.2011.0158

ISSN 1751-8822

Novel binary tree Huffman decoding algorithm and field programmable gate array implementation for terrestrial-digital multimedia broadcasting mobile handheld

S. Beak[1] *B. Van Hieu*[1] *H. Lee*[1] *S. Choi*[1] *I. Kim*[1] *K. Lee*[2] *Y. Lee*[2] *T. Jeong*[1]

[1]*Department of Electronic Engineering, Myongji University, Korea*
[2]*Korea Electric Telecommunication Institute, Korea*
E-mail: sbeak@mju.ac.kr

Abstract: Recent mobile devices, which adopted Eureka-147, terrestrial-digital multimedia broadcasting (T-DMB) systems, are developed as integrated circuit. As a result, the space of memory expands hardly on mobile handheld. Therefore most mobile handheld must operate a lot of application on limited memory. To solve the problem, most of the mobile devices use some kind of compression algorithms to overcome the memory shortage. Among such algorithms, Huffman algorithm is most widely used. In this study, the authors present a novel binary tree expression of the Huffman decoding algorithm which reduces the memory use approximately by 50% and increases the decoding speed up to 30%. The authors experiment the decoding speed on an evaluation kit (SMDK 6400), which is a T-DMB mobile handheld with an advanced risk machine processor. Later to enhance the decoding speed, the authors present an optimum Huffman decoder based on hardware implementation.

1 Introduction

Terrestrial-digital multimedia broadcasting (T-DMB), based on Eureka-147 digital audio broadcasting, is now employed as a marvellous broadcasting system for mobile handheld. It is able to provide a multimedia broadcasting public service under a fixed, portable mobile environment [1]. However, the limited memory of a mobile device limits the operation of various multimedia applications on the mobile device. To overcome such a problem, Huffman code is widely used on mobile devices

Since Huffman encoding scheme was proposed by D.A. Huffman in 1952, Huffman code has been adopted to mobile platform in text, image and video compression [2]. An advantage of this compression algorithm is an efficient utilisation of channel bandwidth and storage size [3]. Among Huffman decoding algorithms, the binary tree method is the simplest method. The binary tree method uses pointers to represent the tree structure. It is difficult to achieve memory efficiency on mobile device. In addition, the decoding speed of this method is slow.

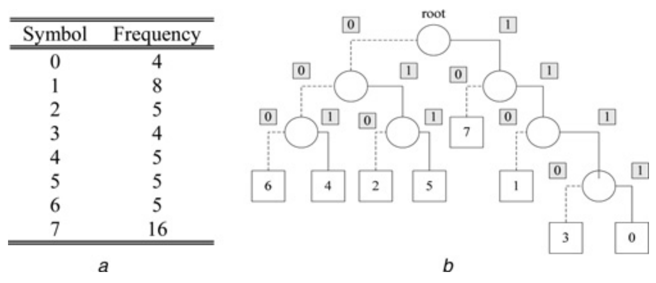
Therefore many works try to improve the memory utilisation or the decoding speed. However, to the best of our knowledge, there has not been a single work that satisfies the two factors simultaneously – fast decoding speed while using the minimal memory.

Hence, we propose a novel Huffman decoding algorithm based on binary tree search that satisfies both factors at the same time. In addition, using a proposed algorithm, we designed a Huffman decoder on a field programmable gate array (FPGA), which is suitable for T-DMB.

This paper is organised as follows. Section 1 introduces the T-DMB and Huffman coding. Sections 2 and 3 explain the conventional Huffman encoding and decoding, respectively. Section 4 presents the proposed Huffman decoding algorithm with a reduced memory size and a fast decoding speed. In Section 5, the performance of the conventional and the proposed algorithm are compared on T-DMB platform using the two factors as the comparison metric. Section 6 shows the implementation of the algorithm in hardware. Section 7 shows the result of the implementation using a private company's FPGA. Finally, concluding remarks are given in Section 8.

1.1 Huffman encoding algorithm

Huffman code is a variable length code, which means that symbols are mapped into code words with different number of bits [3–5]. Giving a set of symbols, Huffman code of the set is generated by constructing a binary tree. The binary tree is created based on probabilities of symbols. Once the frequency data have been determined [6–8] like Fig. 1*a*, the two parentless nodes with the lowest probabilities are selected. After that, a new node called the parent of the two lowest probability nodes is created. The probability of the new node is equal to the sum of its children's probabilities. This process is repeated until there



Symbol	Frequency
0	4
1	8
2	5
3	4
4	5
5	5
6	5
7	16

Fig. 1 *Procedure of Huffman encoding*

a Number of frequency times
b Huffman binary tree

Table 1 Encoded data by Huffman algorithm

Symbol	Codeword	Length	Symbol	Codeword	Length
0	1111	4	4	001	3
1	110	3	5	011	3
2	010	3	6	000	3
3	1110	4	7	10	2

is only one parentless node left. Fig. 1*b* is the binary tree corresponding with data in Fig. 1*a*.

Huffman code is obtained from the binary tree. The left and right branches of a node are assigned 0 and 1, respectively. The path from the top parent node to the lowest node is the code word used to encode the symbol. Table 1 is the Huffman code generated from binary tree in Fig. 1*b*.

2 Conventional Huffman decoding algorithm

Giving a Huffman code, conventional Huffman decoding method regenerates the binary tree, called decoding tree, to decode data. For example, Fig. 1*b* is the decoding tree corresponding with the Huffman code in Table 1. In order to visualise the difference between leaf nodes and intermediate nodes of the tree, we use squares for leaf nodes and circles for intermediate node. The numbers inside the squares are leaf node values, which correspond to the symbol values. A path from the root node (the top of circles in Fig. 1*b*) to a leaf node (any square in Fig. 1*b*) is the codeword corresponding to that leaf node. For example, the path from the root node to the leaf node 6 is 0-0-0, which is the codeword of the symbol 6 as in row seven of the Table 1.

Using the decoding tree, algorithm may be accomplished by reading the encoded data one bit at a time. The process starts at the root node. If the received bit is 0 or 1, the left or right child node will be considered, respectively. If the child node is a leaf node, a new symbol whose value is the leaf node's value is decoded and the process restart at the root node. In contrast, the process continues until it reaches a leaf node.

There are two methods to implement the binary tree for Huffman tree, array data structure and linked list. The advantages of the linked list data structure are low memory usage, insertion and deletion of a node. Although the linked list data structure has above merits, it is not used in reality because of a major drawback, difficulty in accessing the parent node of the tree. On the other hand, the array data structure allows accessing the intermediate node. Therefore it has been adopted for decoding Huffman tree. But, the major disadvantage is the memory spent on storing such a complete binary tree [9–11].

Lastly, the calculation about need of memory spread widely. Formula (1) shows the memory used (in bytes) by a conventional Huffman binary tree

$$\mu = 2(n-1) \tag{1}$$

where 'μ' is the Memory Usage, 'n' is the number of leaf node.

2.1 Proposed Huffman decoding algorithm

The advantage of our method for representing the above Huffman binary tree is the low memory use while maintaining an easy accessibility to the original date – combining the advantages of the two data structures (linked list and array data structure). From the decoding tree, our method generates a tabular representation of the decoding tree. And the decoding process is performed on this tabular representation. The following describes the proposed algorithm in more detail.

Firstly, each intermediate node is assigned a unique number. If a decoding tree has n leaf nodes, it will have $n-1$ intermediate nodes. As a consequence, intermediate nodes' index number is from 0 to $n-2$. Fig. 2 is the decoding tree in which all intermediate nodes are indexed. The numbers inside circles are index numbers of intermediate nodes. In Fig. 2, we use depth-first-search to assign index number. The index number is from 0 to 6.

After indexing the decoding tree, it can be represented in a tabular representation like Table 2. Each entry of the representation is a transition of the decoding tree. Because all intermediate nodes always have two child nodes, each intermediate node always occupy two entries of the representation. In our method, the transitions to left and right nodes of the intermediate node K occupy position 2K and 2$K+1$ of the tabular representation, respectively. Each entry of the representation includes two fields called Classify code and Value. Classify code specifies whether a child node is a leaf node or intermediate node. If a child node is an intermediate node, Value specifies the index number of that node. If a child node is a leaf node, Value specifies the symbol value of that node.

We visualise the value of each entry of the tabular representation in Fig. 2. A pair of number in rectangle is an entry. The number in grey is Classify code and the other is

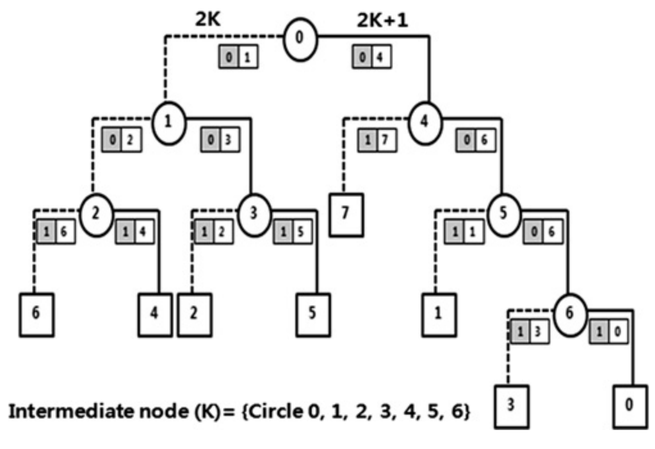


Fig. 2 *Proposed Huffman decoding binary tree*

Table 2 Memory storage technique of proposed decoder

Intermediate node, K	Memory address	Classify code	Value
0	0	0	1
	1	0	4
1	2	0	2
	3	0	3
2	4	1	6
	5	1	4
3	6	1	2
	7	1	5
4	8	1	7
	9	0	5
5	10	1	1
	11	0	6
6	12	1	3
	13	1	0

Value. Table 2 is the tabular representation corresponding to the decoding tree in Fig. 2.

Using the tabular representation, the decoding process is as follows. The decoding process is a table lookup process. At the beginning, an index is assigned zero. When a new data bit is received, this bit is added with the index to generate the new value of the index. Then, the entry whose position is specified by index is fetched. If its Classify code is 1, a new symbol whose value is entry's Value is decoded and the index is reset to zero. In contrast, if Classify code is 0, the index is set to 2 × Value.

At last, Formula (2) shows proposed algorithm memory calculation based on bits.

$$\mu = 2(n-1)(d+1) \tag{2}$$

where 'μ' is Memory Usage, 'n' is the number of leaf node and 'd' is the symbol length.

3 Comparison of the conventional and the proposed algorithm

Table 3 compares the amount of memory used by the conventional and the proposed Huffman decoding algorithm. As mentioned above, the formula for the proposed Huffman decoding algorithm is expressed in bits; thus, the value of the calculation needs to be divided by eight. As shown in Table 3, the proposed algorithm only uses half of the memory.

To prove that the proposed algorithm is faster than the conventional algorithm, we first experiment the two algorithms, conventional and proposed Huffman algorithm, on a T-DMB platform. The T-DMB platform is based on WinCE OS, and consists of 16/32 bit RISC microcontroller,

Table 3 Compare the quantity of memory

	Conventional Huffman decoding algorithm	Proposed Huffman decoding algorithm
calculation	$2(n-1) = 2(8-1)$	$2(n-1)(d+1) =$ $[2(8-1)(3+1)]/8$
total memory	14 byte	7 byte

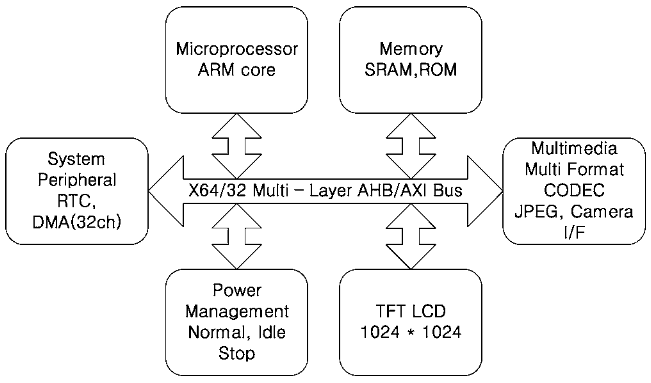
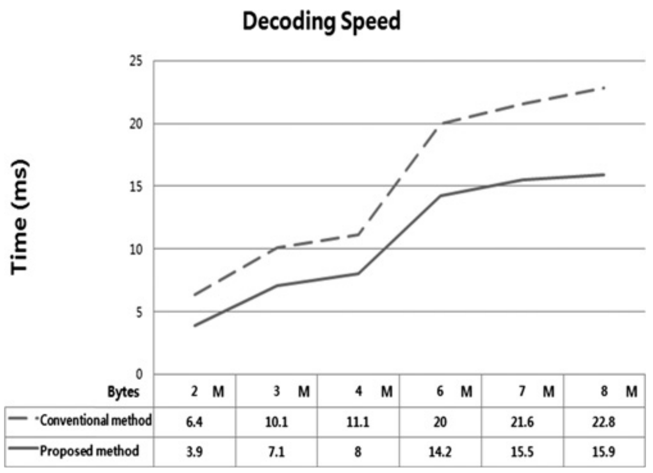

Microprocessor ARM core
Memory SRAM,ROM
System Peripheral RTC, DMA(32ch)
X64/32 Multi – Layer AHB/AXI Bus
Multimedia Multi Format CODEC JPEG, Camera I/F
Power Management Normal, Idle Stop
TFT LCD 1024 * 1024

Fig. 3 *Block Diagram of T-DMB platform*



Bytes	2 M	3 M	4 M	6 M	7 M	8 M
Conventional method	6.4	10.1	11.1	20	21.6	22.8
Proposed method	3.9	7.1	8	14.2	15.5	15.9

Fig. 4 *Test result on the T-DMB mobile platform*

SDRAM, TFT, LCD etc. as shown in Fig. 3. The Microcontroller has 16 KB I-Cache, 16 KB D-Cache and 64 and 128 MB SDRAMs. We run the two decoding algorithms on six files of various lengths. To get an accurate result, each file is tested four times, and those results are averaged. The results of the test are shown in Fig. 4. The proposed algorithm is faster than the conventional decoding tree search method as shown in Fig. 4.

4 Implementation on hardware

To validate the hardware implementation of the decoder, the Test Environment shown in Fig. 5 is used. The software component, the encoder and its peripherals, is written in C; the hardware component, the decoder and its peripherals, is written in Verilog. The following two sections describe the implementation of the software and hardware components in more detail, respectively.

4.1 Implementation of algorithm in software

Original data (Software part shown in Fig. 5): Huffman encoding scheme has been used to compress data in the diverse digital field. Hence, Huffman encoder deals with some data types. To prove our decoding algorithm, the text file with 64 Kbyte is chosen.

4.1.1 Encoder: The encoder performs Huffman encoding from the original data. After encoding, it generates two Huffman Rom Data and Encoded Rom Data. Huffman Rom

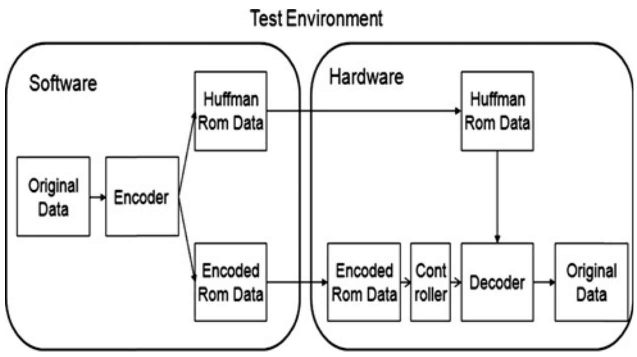


Fig. 5 *Data flow structure of test environment*

Data are the tabular representation of the Huffman decoding tree and Encoded Rom Data are the content of encoded data. Both of them are stored to ROM so that the hardware can read and perform decoding.

4.2 Implementation of algorithm in hardware

4.2.1 Decoder: The decoder is based on the flowchart shown in Fig. 6. The decoder access data from ROM, which stores Tabular representation of Huffman decoding tree, based on the new data bit from inData signal. At the beginning, romAddr is set to 0. When there is a new bit of data, it is added with current romAddr form address to access ROM. Since each memory word of the ROM is a entry of the tabular representation, the first bit or romData (romData[0]) is the IsSymbol field. If it is 1, a new symbol is decoded. So a signal called hasData is set to 1 to announce, the new symbol is loaded to a register called outData, and the romAddr is resetted to 0. If it is 0, the new ROM address is calculated as romData[8:1].

Using the flowchart in Fig. 6, the structure of the Huffman decoder is shown in Fig. 7. All registers connect Clk, nRst signals for synchronous and asynchronous operation and link EN signal for test. The function of pins shown in Fig. 7 is described as follows. The decoding circuit is very simple with only some registers.

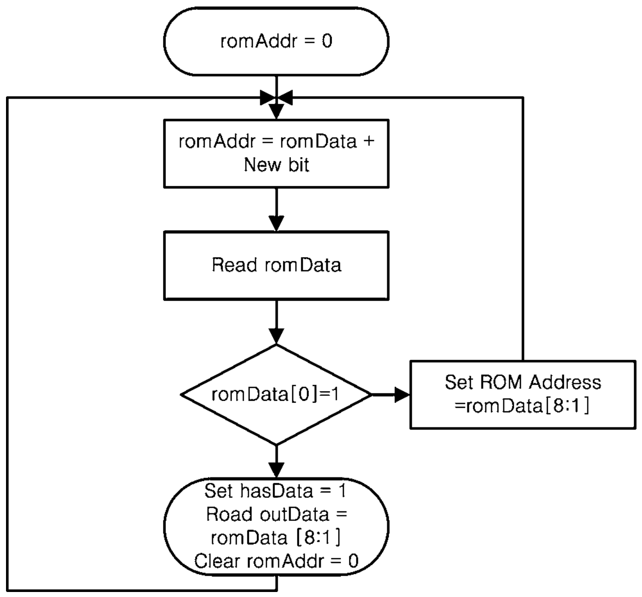

romAddr = 0
romAddr = romData + New bit
Read romData
romData[0]=1
Set ROM Address =romData[8:1]
Set hasData = 1 Road outData = romData [8:1] Clear romAddr = 0

Fig. 6 *Flowchart of Huffman decoder*

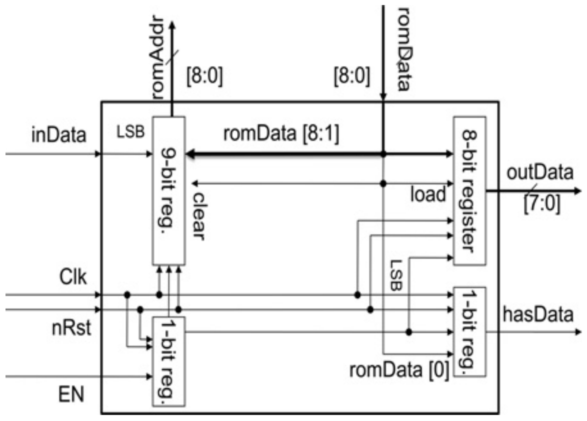


Fig. 7 *Structure of the Huffman decoder*

```
always @( negedge nRst or posedge Clk) begin
  if(nRst == 1’b0) begin
  // when nRst take place “0”, not operation.
    hasData <= 1’b0;
    romAddr <= 9’b000000000;
    delay1ClockEN <= 1b’0;
  end
  else begin
    delay1ClockEN <= EN;
    if (delay1ClockEN <= 1’0) begin
    //first clock after enabled of finish
      romAddr <= {8’b00000000,inData};
      hasData <= 1’b0;
    end
    else begin // Second clock to last clock
    if (romData[0] == 1’b1) begin
      romAddr <= {8’b00000000, inData};
      outData <= romData [8:1];
    end
    else begin
      romAddr <= {romData[8:1],inData}
    end
    hasData <=RomData[0];
  end
end
```

Fig. 8 *Proposed decoder design in RTL code*

- inData: input encoded data (1 bit/clock);
- Clk: system clock;
- EN: decoder enable (synchronous);
- nRst: decoder reset (asynchronous);
- outData: (8 bits) decoded data;
- hasData: indicates that decoder is outputting data;
- romAddr: (9 bits) address signals for accessing rom;
- romData: (9 bits) data from ROM.

Hence, the performance of hardware depends on users’ FPGA shown in Fig. 8. So we derive the formula about the speed.

$$S = N \times T \tag{3}$$

where S is speed result and N is data size (bit) and T is clock speed.

4.2.2 Controller: The data in ROM are stored in byte (8 bits) unit. However, our decoding circuit decode one bit after one clock cycle. Controller will read data from ROM, then serial shift it to the decoding circuit.

Fig. 9 shows a register transfer level (RTL) code which is a part of the proposed 1-bit controller.

```
//Decoder Enable
if (counter != bitLength) begin
//Read memory
  if (counter[2:0] == 3'h0) begin
    romEN  <= 1'b1;
    nRomRD <= 1'b0;
  end
  else begin
    romEN  <= 1'b0;
    nRomRD <= 1'b1;
  end
  //Decoder enable
  if (counter[2:0] == 3'h1) begin
    decoderEN      <= 1'b1;
  end
  //Shift data
  if (counter[2:0] == 3'h1) begin
    shiftData        <= romData;
  end
  else begin
    shiftData <= {1'b0, shiftData[7:1]};
  end
  counter <= counter + 1;
end
else begin
  romEN     <= 1'b0;
  nRomRD    <= 1'b1;
  decoderEN <= 1'b0;
end
```

Fig. 9 *Proposed controller design in RTL code*

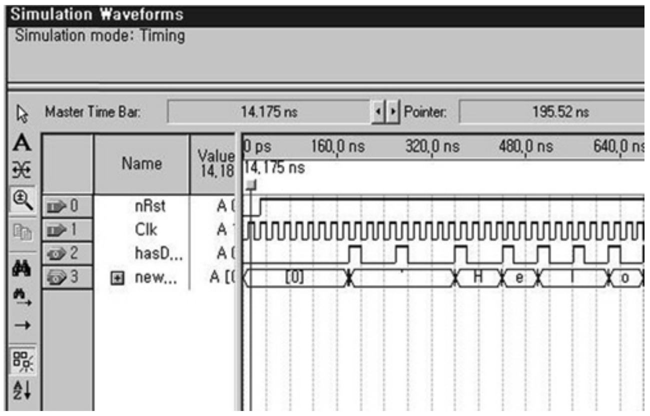


Fig. 10 *Waveform verification result using Huffman decoder*

4.2.3 Original data (Hardware part shown in Fig. 5):

To compare decoded result against the original data, we simulate the proposed Huffman decoder using a private company's software. The Huffman hardware decoder correctly encoded data having a character, that is, 'HELLO' message as shown in Fig. 10.

5 Test and results

Since the proposed algorithm can decode any type of digital data, MP3 data were used for testing. To test decoding speed, we use 65.3 Kbyte MP3 data.

The performance is measured on five different series of a same vender. We express five different series as A to E. Among each series, the proposed decoder was tested on multiple models and the best performance result was recorded. Among each series, E series is superior to the other series. The complete simulation result is shown in Fig. 11.

Fig. 12 shows decoding speed of the best performance result on various FPGA series. The decoding speed of '*E*' model is

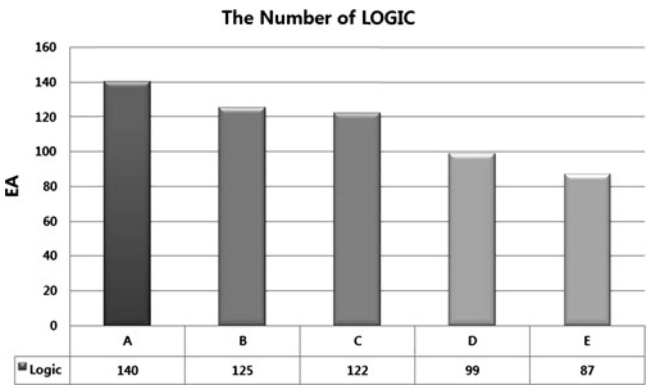


Fig. 11 *Logic elements result of a private company's FPGA*

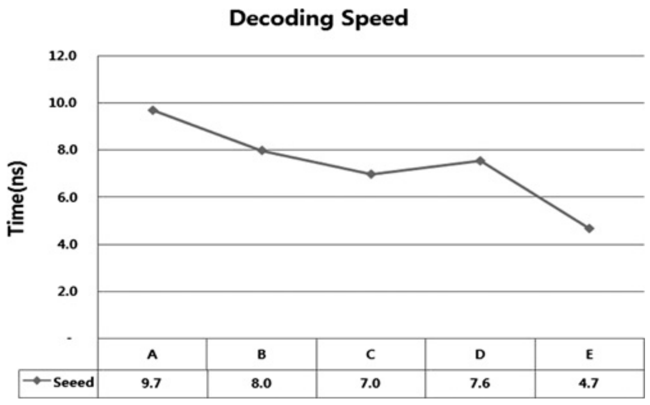
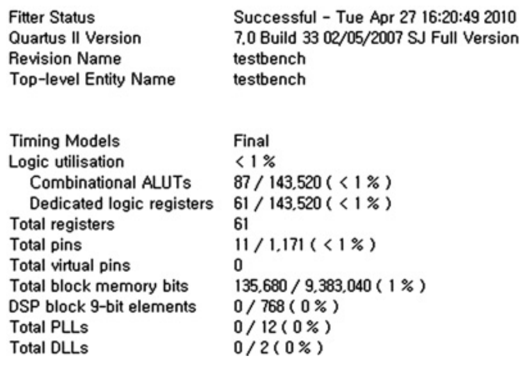


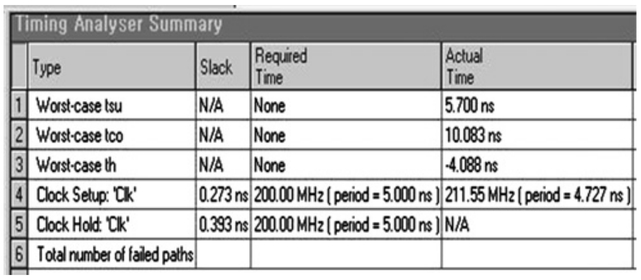
Fig. 12 *Decoding speed result of a private company's FPGA*

4.7 ns and total logic elements are 87. In the case of '*D*', decoding speed 7.6 ns and total logic elements are 99. As mentioned above, E series is superior to the other series. The complete simulation result is shown in Fig. 12. Finally, Figs. 13 and 14 show the synthesis result for verification.



Fitter Status	Successful - Tue Apr 27 16:20:49 2010
Quartus II Version	7.0 Build 33 02/05/2007 SJ Full Version
Revision Name	testbench
Top-level Entity Name	testbench
Timing Models	Final
Logic utilisation	< 1 %
Combinational ALUTs	87 / 143,520 (< 1 %)
Dedicated logic registers	61 / 143,520 (< 1 %)
Total registers	61
Total pins	11 / 1,171 (< 1 %)
Total virtual pins	0
Total block memory bits	135,680 / 9,383,040 (1 %)
DSP block 9-bit elements	0 / 768 (0 %)
Total PLLs	0 / 12 (0 %)
Total DLLs	0 / 2 (0 %)

Fig. 13 *Logic elements verification of B series on FPGA*



Timing Analyser Summary

	Type	Slack	Required Time	Actual Time
1	Worst-case tsu	N/A	None	5.700 ns
2	Worst-case tco	N/A	None	10.083 ns
3	Worst-case th	N/A	None	-4.088 ns
4	Clock Setup: 'Clk'	0.273 ns	200.00 MHz (period = 5.000 ns)	211.55 MHz (period = 4.727 ns)
5	Clock Hold: 'Clk'	0.393 ns	200.00 MHz (period = 5.000 ns)	N/A
6	Total number of failed paths			

Fig. 14 *Frequency verification of B series on FPGA*

6 Conclusion

In this paper, the implementation of the proposed Huffman decoder on an FPGA has been shown. We prove the advantage of our algorithm from the experimental results. Hence, we expect the proposed method to be applied to T-DMB systems. The speed of proposed hardware decoder is limited by the sequential processing. Hence, a customised IC design using VLSI [12] technology can further improve the speed. A new Huffman hardware decoder using pipeline technology [9, 13], which is our next research topic, may contribute to improve speed.

7 Acknowledgments

This work was supported by National Science Foundation (NRF) grant funded by the Korea government (MEST) (No. 20090069991). This work was supported by the Korea Government Ministry of Knowledge and Economics (MKE) under the grant No. I-2010-1-012 of the Electric Power Industry Tech. Evaluation and Planning Center (ETEP). This work was supported by the IT R&D program of The MKE/KEIT. 10040191, the development of Automotive Synchronous Ethernet combined IVN/OVN and safety control system for 1 Gbps class.

A preliminary version of this paper is presented at IEEE ICCE 2010 in LV, US in January 12, 2010 and edited with an honourable request by the program committee of IEEE ICCE 2010 for a journal publication (corresponding author T. Jeong).

8 References

1 Kim, S.-H., Kim, Y.-S., Lim, J.-S., Ahn, C., Choi, U.-R., Soe, B.-S.: 'Design of the channel estimation algorithm for advanced terrestrial DMB system', *IEEE Trans. Broadcast.*, 2008, **54**, pp. 816–820

2 Wang, P.-C., Yang, Y.-R., Lee, C.-L., Change, H.-Y.: 'A memory-efficient Huffman decoding algorithm'. 19th Int Conf on, Advanced Information Networking and Applications, AINA 2005, March 2005, vol. 2, pp. 475–479

3 Aspar, Z., Mohd Yusof, Z., Suleiman, I.: 'Parallel Huffman decoder with an optimize look up table option on FPGA'. TENCON 2000. Proc. of TESCON, September 2000, vol. 1, pp. 73–76

4 Rigler, S., Bishop, W., Kennings, A.: 'FPGA-based lossless data compression using Huffman and LZ77 algorithms'. Electrical and Computer Engineering, CCECE 2007. Canadian Conf., April 2007, pp. 1235–1238

5 Ezhilarasan, M., Thambidurai, P., Praveena, K., Srinivasan, S, Sumathi, N.: 'A new entropy encoding technique for multimedia data compression'. Int. Conf. on Computational Intelligence and Multimedia Applications, December 2007, vol. 4, pp. 157–161

6 Huffman, D.A.: 'A method for the construction of minimum redundancy codes'. Proc. IRE40, 1952, pp. 1098–1101

7 Roman, S.: 'Coding and information theory' (Springer, New York, 1992)

8 Chung, K.L., Wu, J.G.: 'Level-compressed Huffman decoding', *IEEE Trans. Commun.*, 1999, **47**, (10), pp. 1445–1447

9 Parhi, K.K.: 'High-speed Huffman decoder architectures'. Signals, Systems and Computer, 1991. Conf. Record of the 25th Asilomar Conf., November 1991, vol. 1, pp. 64–68

10 Oh, H.-Y., Erturk, S., Chang, T.-G.: 'Low complexity video encoding with one-bit transform based network-driven motion estimation', *IEEE Trans. Consum. Electron.*, 2007, **53**, (2), pp. 601–608

11 Jeong, T., Hieu, B., Beak, S.: 'Memory efficient multimedia Huffman decoding method and apparatus for adapting Huffman table based on symbol from probability table', *South Korea Patent File No.10-1030726*, November 26. 2009

12 Chang, S.-F., Messerschmitt, D.G.: 'VLSI designs for high-speed Huffman decoder'. Proc. IEEE Int. Conf. on Computer Design: VLSI in computers and Processors, ICCD'91, October 1991, pp. 500–503

13 Parhi, K.K.: 'High-speed VLSI architectures for Huffman and Viterbi decoders', *IEEE Trans. Circuits Syst. II: Analog Digit. Signal Process.*, 1992, **39**, pp. 385–391