IFAC
Publications

FPGA IMPLEMENTATION OF THE DYNAMIC HUFFMAN ENCODER

Ernest Jamro, Maciej Wielgosz, Kazimierz Wiatr

AGH University of Science and Technology
al. Mickiewicz 30, 30-051, Poland
Academic Computer Center CYFRONET
ul. Nawojki 11, Kraków 30-950, Poland
email: jamro / wiatr@agh.edu.pl

Abstract: At first part of this paper, the architecture for quasi-static Huffman encoder is described which main part is Look-Up Table (LUT). In order to reduce the hardware requirements, the maximum length of the encoded word is limited. This reduces the compression ratio insignificantly which is proved in this paper. The dynamic encoding is achieved by a change of the LUT contents and hardware-software co-design approach. Consequently counting the input words statistics (histogram) and sorting the resultant histogram is implemented in hardware. The final calculation of the new LUT contents and controlling the whole system is achieved by the soft-processor MicroBlaze.

Keywords: data compression, adaptive algorithms, signal processing, hardware-software

1. INTRODUCTION

Huffman coding [1] is one of the basic and the most frequently employed lossless compression method. It takes advantage of input words statistics. Input words, which appear more frequently, are assigned to shorter codes. Conversely words, which occur fewer times, obtain longer codes. This reduces the total number of bits necessary to code the whole information. The Huffman code is based on the input signal entropy. The entropy limits the maximum compression ratio.

There are many implementations of Huffman coders in VLSI technology [2, 3] but they are not optimal for FPGAs. Many examples of FPGA static Huffman coders can be found, e.g. [4,5,6,7,8]. They assume that the signal distribution (histogram) is known prior the implementation. This kind of algorithms does not involve any modifications of the coding assignments. If statistics of input data changes in time, these types of algorithms results in coding overheads and should not be employed. This paper proposes dynamic Huffman encoder, which updates encoding scheme every (transmit) session.

The design process employs the modular structure. Several modules compatible with Embedded Development Kit (EDK) provided by Xilinx Inc. and On-chip Peripheral Bus provided by IBM [9] were designed. Besides the EDK provides soft-processor MicroBlaze which is frequently used in the system to perform program-driven algorithms, e.g. constructing the Huffman tree, summing up, construction of the final system took place in the EDK environment. In the presented system hardware-software co-design approach was adopted, i.e. the whole algorithm was divided into software and hardware part. In the case when only MicroBlaze is employed, the coding process would be significantly slowed down. Conversely, completely hardware approach would significantly complicate adaptation of Huffman coding and would increase the FPGA area.

This paper present results of 8-bit input word. Coded word can be lengthened but it would increase the size of the employed memory from 2^8 to 2^n locations (n is a bit-length of a single input word). Currently available FPGA chips support enough internal memory for n less than roughly 12. Attempt to increase n over 12 invokes external memory

utilization and consequently the significant coder rebuilding.

At the beginning of this paper the proposed architecture of the static Huffman encoder is described. The proposed architecture slightly modifies the original Huffman algorithm, therefore the compression ratio might be slightly changed, which is also studied. Then the architecture of the dynamic Huffman encoder is approached. This encoder consists of four major parts: 1) quasi-static Huffman encoder, 2) Histogram module, 3) Sorting module, 4) soft-processor MicroBlaze. All the above mentioned parts, except the MicroBlaze, are dedicated hardware for dynamic Huffman encoder.

2. STATIC HUFFMAN ENCODING

At the beginning the static Huffman encoder will be described as this encoder is also employed as a part of the dynamic Huffman coder. The main part of the static encoder is the Look-Up Table (LUT) memory, which is addressed by 8-bit input words. The output of the LUT provides the code (maximum 12-bit) and length (4-bit) assigned to each input word. Similar algorithm was employed in [6, 7, 8], but these papers did not include any information about quality of obtained compression ratio when the maximum codeword length is limited.

The Virtex FPGA contains 256x16-bit BRAMs, therefore in the adopted algorithm each of 256 input words occupy sixteen bits of the LUT. The output code is located in twelve LSBs, four MSBs determine the length of the output code. This coding format causes some limitation of the coding algorithm. Input words appearing very rarely can be coded on up to 12 bits. This results in a degradation of the compression ratio, which will be studied in the next paragraph.

2.1. Adopted coding method and compression ratio

The information I_i of coded word depends on the input word probability p_i as follows [8]:

$$I_i = -\log_2(p_i). \qquad (1)$$

The Huffman coding procedure uses the integer number of bits d_i for the coded words, and thus the following is satisfied:

$$d_i < 1 - \log_2(p_i) \qquad (2)$$

For the described algorithm, the compression ratio is maximally degraded in the case when $(m+k)$ bits are required to code almost all input symbols, where m is the maximum length of a single coded word (in our case $m = 12$); k is the number of additional bits which are skipped because of the proposed algorithm constrains. Fortunately, according to eq. 2 ($d_i = m-k$) each input symbol which cannot be efficiently coded occurs with the probability:

$$p_i < \frac{1}{2^{m+k-1}}. \qquad (3)$$

The number of redundant bits R, required to code a single n-bit input word (2^n is the number of all input words) stands as follows:

$$R = (2^n - j)\cdot k \cdot p_i < \frac{k}{2^{m+k-n-1}} \qquad (4)$$

where: j is the number of symbols which are correctly coded ($m \geq d_i$), and the assumption is made that j is a very small number (the worst case assumption).

It can be calculated from eq. 4 that the largest redundancy R is obtained for $k = 1/\ln(2) \approx 1.44$ and therefore the following is satisfied:

$$R < \frac{0.53}{2^{m-n-1}} \approx \frac{1}{2^{m-n}} \qquad (5)$$

Consequently the worst possible degradation of the compression ratio in the proposed coding format ($n=8$, $m=12$) is less than roughly 0.066 bit per coded word. This is a small value that can be accepted. It should be noted that the worst case was hereby considered and in the most cases degradation of the compression ratio is significantly lowered. Even the theoretical simplifications cause that the given maximum redundancy R in eq. 5 is greater than the practical one. Besides, the proposed coding format ($n=8$, $m=12$) in many cases improves the compression ratio because limiting maximum length of the coded word also limits size of the code table. The code table should be also sent to a receiver (at least at beginning of the transmission) together with the coded data therefore its size influences the total compression ratio. Moreover code table for rarely occurring words does not have to be sent because input data might be delivered without change to the receiver, only a specific $(m-n)$-bit preamble might be added.

2.2. Architecture of the Static Huffman Coder

The proposed static Huffman encoder denoted as *opb_huff* is presented in Fig. 1 and contains two independent OPB buses (*master* and *slave*).

- *slave* bus is used to receive input data and to modify LUT contents (used for quasi-static version)
- *master* bus is used to transfer out the coded words

The main part of the *opb_huff* is code table (LUT) implemented as dual port block RAM (BRAM). As

an input word appears on the address port of the BRAM, output code word (12 bits) and code-length (4 bits) can be read from the BRAM output. Then the code word feeds the module denoted as *barrel shifter*. The module *barrel shifter* concatenates consecutive variable-length words received from the BRAM into fixed-length (16-bit) words which can be further processed (e.g. transmitted). The concatenation requires bit-alignment of the previous and the next variable-length words, which involves a special barrel shifter. This *barrel shifter* module contains also registers and accumulators and its detail description is given in [6].

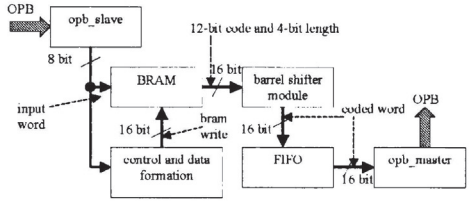

Fig. 1. *Opb_huff* block diagram – the architecture of the static Huffman coder

3. DYNAMIC HUFFMAN ENCODER

The above proposed static Huffman encoder can be also employed as a part of the dynamic Huffman encoder. Only a new code table must be written to the BRAM memory in order to change the whole coding scheme. The static Huffman encoder for which the contents of the BRAM can be changed is further denoted as a quasi-static Huffman encoder. The main problem for the adaptive Huffman encoder is calculating new BRAM contests at every data frame. The dynamic Huffman encoder should work in the real time system therefore the speed is a critical factor. The dynamic Huffman encoder consists of three different modules:

1) Histogram – calculates the input data statistics while a new image or data frame is grabbed to a buffer (external memory).
2) LUT Contents Calculation module – this module calculates and updates the new contents of the BRAM memory.
3) Quasi-static Huffman encoder – this module was described in Section 2.

It should be noted that these modules can work independently at the same time using a pipeline architecture. Consequently, while the histogram calculates the input data statistics for *frame 3*, module 2 calculates the new BRAM contents for *frame 2*, and the quasi-static Huffman encoder encodes *frame 1*, and so on.

3.1. Histogram Calculation

Histogram calculation is the first stage of code table generating process. The calculation speed is the critical criterion so two different methods: software and hardware of histogram calculation will be compared. C-language code calculating the histogram was implemented on the soft-processor MicroBlaze. This code results in twelve assembler instructions and takes about 30 clock cycles to process a single input word (single loop). Such a long calculation time is unacceptable for real time systems.

As a result, a hardware counterpart denoted as *opb_hist* was designed. This module requires only 1 clock cycle per input word and consists of two independent OPB bus interfaces.

- DOPB accepts the input data (OPB slave).
- COPB is used to read the calculated histogram and also to initialize *opb_hist* internal counters (OPB slave).

The simplified block diagram of the *opb_hist* is given in Fig. 2A. Input data *(DataIn)* address the BRAM. Every value of data feeding the *opb_hist* has its own memory cell which stores the number of occurrence of this particular data value. This number is increased by one for every new occurrence of the data and is written back to BRAM memory (*Din* port).

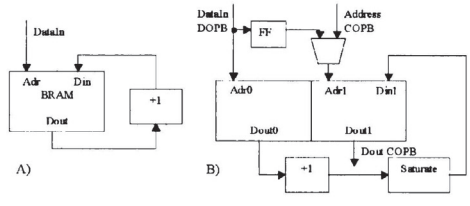

Fig.2. *opb hist* block diagram A) simplified, B) implemented

The block diagram of the implemented histogram is given in Fig 2B and is much more complicated in order to speed up the calculation process. The main problem was that BRAM read access time is a whole clock cycle. Consequently a single input data would require 2 clock cycle: the fist clock cycle for reading the previous histogram state from the BRAM and the second cycle for the incrementing previously read value and writing it back to BRAM at the same address. Such a long calculation time may be unacceptable. One of the solution presented in [13] is to use a local clock with a double frequency. This causes that some part of the logic has shorter propagation time limit. Another solution, proposed in this paper is to use both ports of the dual port BRAM, for which the first port is used to read the occurrence count (*Dout0* in Fig. 2), and the second port (*Din1*) is used to write the incremented count in

the next clock cycle. Both ports are addressed by the input data *DataIn*, however the address on the second port is delayed by one clock cycle in the module *FF*. Additional address multiplexer at *Adr1* is used to initialize and read the resultant histogram. The proposed hardware implementation has several advantages. First of all it allows fetching new input data every clock cycle. Secondly reduces the hardware requirements, MicroBlaze occupies much more logic than the *opb_hist* (see Tab. 1 for comparison).

3.2. Calculating the Huffman tree

The next step is to calculate the new BRAM contents of the quasi-static Huffman encoder. The main part of this step is to find out the Huffman tree which consists of two stages:

1. Finding two the least elements of the histogram and adding them. In this way a new root of the tree is created and the previous two are removed.
2. Repeat step one until there are at least two elements of histogram left

An example of the Huffman tree is given in Fig. 3.

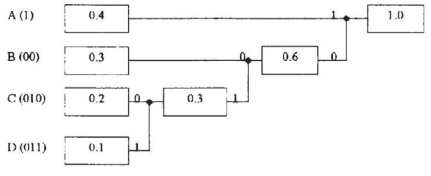

Fig. 3. Example of creating Huffman tree

3.3. Sorting the histogram

The most time consuming operation when calculating the Huffman tree is finding two least elements of the histogram. This can be achieved efficiently by at first sorting the whole histogram (table) and then just taking out the last two elements from the sorted table. The new element (sum of the least two elements) is then properly inserted to the sorted table with respect to the element value. Consequently at every stage the table is sorted and therefore finding two least elements is a trivial task.

While forming a new root of the Huffman tree, a new element is *inserted* to the table. Consequently, the insert-sort algorithm is selected. The insert-sort algorithm requires $N^2/4$ on average and $N^2/2$ on maximum comparison and shift operations; where N is the number elements in the sorted table. For 8-bit input data, $N=256$, so the average number of shift and comparison operations equals roughly 8000. If a faster algorithm is used (e.g. quick-sort) computational time will be shorten to roughly $N \cdot \log_2 N$. In the presented system it would require roughly 2000 comparisons and shifts. Unfortunately, the construction of the Huffman tree prefers the insert-sort. Besides usage of the quick-sort algorithm results in a more complicated code which might significantly reduce the algorithm speed-up. At last but not least, in the case when the sorter is implemented in hardware, the algorithm simplicity and sequential-memory access is the key issue. Summing up, the insert-sort algorithm is the final choice.

MicroBlaze processor would require a few dozens of assembler instructions to complete a single comparison and shift operation. As a result, the whole sort procedure would take too many clock cycles. Consequently, the specialized hardware module denoted as *opb_sort* has been designed. The *opb_sort* contains three independent OPB buses (two *masters* and one *slave*).

- SOPB (slave). This bus is used to control the *opb_sort*: write a new word which will be inserted to the table, write the start address of the table and the number of already sorted elements in the table.
- MSOPB (master). Fetches data from the input table.
- MDOPB (master). Sends data to the output table.

The *opb_sort* inserts properly only a single element to the already sorted table. Consequently, an external control machine (e.g. MicroBlaze) is required to control the *opb_sort* and to write sequentially new elements to the *opb_sort* when the sorter has finished the previous job. The block diagram of the sorting system in given in Fig. 4. The *opb_sort* has access to both BRAM ports. Such a solution speeds up BRAM transfers because data can be read and write at the same time and the sequential addressing is employed when accessing the memory. Summing up, both a single comparison and move operations can be accomplished in every clock cycle.

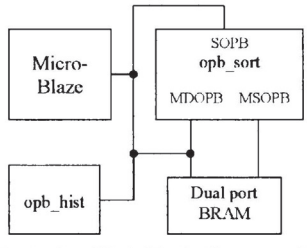

Fig. 4. A simplified block diagram of the sorting system

3.4. Calculating the LUT contents

The next step is to calculate the BRAM contents of the quasi-static Huffman encoder. This algorithm

uses the previously determined Huffman tree and requires relatively small number of cycles. Consequently it is implemented on the MicroBlaze. The most popular implementations of this algorithm are recurrent. As a result they consume large amount of heap memory that might not be available in the system as the heap memory was located in the internal FPGA memory to speed up the access time. Consequently, non-recurrent algorithm was adapted in the presented system.

4. THE WHOLE SYSTEM

In this section the whole system will be introduced with focus on interconnection and cooperation between different modules. A diagram of the system is presented Fig 5.

Fig. 5 contains the following modules (all modules were designed by the authors of this paper, otherwise the source of the module is given):

- *MicroBlaze* soft-processor (module provided by Xilinx in EDK)
- *ilmb_bram_ifctrl* – Local Memory Bus (LMB) controller for MicroBlaze program memory (module provided by Xilinx in EDK)
- *dlmb_bram_ifctrl* – Local Memory Bus (LMB) controller for MicroBlaze data memory (module provided by Xilinx in EDK)
- *bram_1* – MicroBlaze internal program and data memory (module provided by Xilinx in EDK)
- *opb2opb_mb* – OPB to OPB bridge with data width conversion and First-In First-Out (FIFO) buffer, this module is used to separate the MicroBlaze and the rest of the modules. The separation allows for local memory transfers and reduces propagation time
- *opb2opb_dma*– converts 32-bit data from SRAM to 8-bit data to feed *opb_hist* and *opb_huff* (8-bit data length format is accepted by these modules).
- *opb_dma_hist* – Direct Memory Access – transfers data from the SRAM memory to *opb_hist* module.
- *opb_dma_huff*– Direct Memory Access – transfers data from the SRAM memory to *opb_huff* module
- *opb_sort* – module which conducts sorting operation
- *opb2opb_huff_out*– converts 16-bit data from *opb_huff* to 32-bit data transferred to *opb_sram*. Additional FIFO buffer is used as the SRAM memory is shared by other modules
- *opb_epp* – interface between OPB and EPP port. During testing, the input and output data are transferred by Parallel Port and APSI system [12]
- *opb_sram* – interface between OPB and external SRAM memory
- *opb_hist* – histogram calculation module
- *bram_2* – block memory utilized by *opb_sort* module to store data being sorted. When sorting process is finished *bram_2* contains sorted data
- *opb_bram_if_ctrl_0* – OPB bus memory controller (module provided by Xilinx in EDK), enables access to sorted data (stored in *bram_2*)
- *opb_bram_if_ctrl_1* – OPB bus memory controller (module provided by Xilinx in EDK), employed to communicate between *bram_2* and *opb_sort* modules
- *opb_huff* – quasi-static Huffman encoder

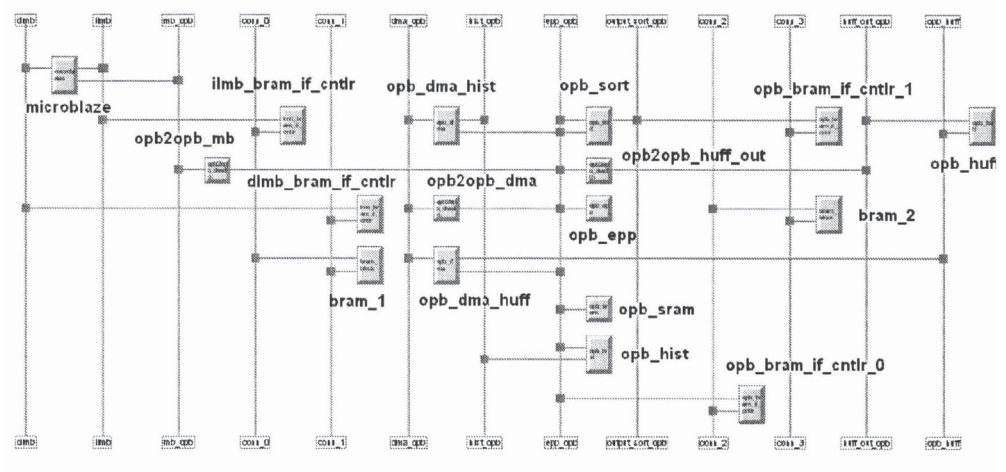

Fig. 5. EDK screen capture of the whole system

As the Microblaze controls the whole system it is worthy to enumerate all tasks it performs in sequence to explain the system operation:

1. Write the new LUT memory contents to the module *opb_huff*. The LUT memory contents was calculate in the previous time-frame.

2. Write control registers of the DMA module: *opb_dma_huff* to transfer data from external memory to Huffman encoder.
3. Read histogram data calculated in the previous time-frame and assign index to each histogram data.
4. Write control registers of the DMA module: *opb_dma_hist* to transfer data from external memory to module *opb_hist* in order to calculate the histogram.
5. Preliminary sorting (*opb_sort* employed)
6. Constructing the Huffman tree and calculating the *opb_huff* LUT contents for the next time-frame.

5. IMPLEMENTATION RESULTS

The whole Huffman dynamic encoder was built in EDK environment and contains several modules compatible with the OPB bus. The encoder was implemented in XSV board equipped with a Virtex FPGA XSV800. Input data is read from the external SRAM memory by the DMA module (*opb_dma*). The FPGA area occupied by the designed modules are given in Tab. 1. These values are approximated because almost every module is associated with additional logics which cannot be easily determined, e.g. OPB bus elements and control logics. Consequently only the whole system results are precise.

Tab.1. Implementation results

Modul	# 4-input LUT	# flip-flops	# 4-kb BRAM
opb_huff	378	157	1
opb_hist	62	10	1
opb_sort	283	98	0
MicroBlaze	1139	410	0
The whole system	3007	1042	13

6. CONCLUSIONS

Modules designed in the presented system (*opb_huff, opb_hist, opb_sort*) are compatible with On-chip Peripheral Bus (OPB) and Xilinx Embedded Development Kit (EDK). Because the modular design methodology was adopted, the proposed system can be relatively easily extended or modified and the design time was significantly shortened. Furthermore hardware-software co-design approach was adopted, which also significantly reduced the design cycle. The most time-consuming operations: counting the input words occurrences (histogram) and sorting the resultant histogram is implemented in hardware. The final calculating of the new contents of the LUT memory and controlling the whole system is achieved by the soft-processor MicroBlaze. The primary aim of this design: adapting coding schedule at every frame of the image (512x512 pixels, 25 frames/s) was achieved.

REFERENCES

[1] D.A.Huffman, A method for the construction of minimum-redundancy codes, In Proc. Inst. Radio Eng, Vol.40, No.9, pp.1098-1101, Sep. 1952

[2] A. Mukherjee, N. Ranganathan, M. Bassiouni, *Efficient VLSI designs for data transformation of tree-based codes*, IEEE Trans. Circuits and Systems, Vol.38, pp.306-314, Mar. 1991

[3] H. Park, V. K. Prasanna, *Area efficient VLSI architectures for Huffman coding*, IEEE Trans. Analog and Digital Signal Processing, Vol.40, pp.568-575, Sep. 1993

[4] Taeyeon Lee and Jaehong Park, *Design and implementation of static Huffman encoding hardware using a parallel shifting algorithm*, IEEE Transactions on Nuclear Science, Vol. 51, Issue 5, pp. 2073-2080, October 2004

[5] OpenCores Org. *Video compression systems* www.opencores.org

[6] Till Jahnke, Sven Schößler, Kolja Sulimma, Pipelined Huffman Encoder with 10 bit Input, 32 bit Output, EDA group of the Department of Computer Science at the University of Frankfurt, http://www.sulimma.de /prak/ss00/projekte/huffman/Huffman.html

[7] S. G. Mathen *Wavelet Transform based adaptive image compression on FPGA*, M.Sc. Thesis, University of Calicut, Calicut, India, 1996, http://www.itte.ku.edu/projects/ACS/documents/sarin_thesis.pdf

[8] Jari Nikara *Parallel Huffman Decoder with an OPtimize Look Up Table Option on FPGA*, Ph.D. Thesis, Tampere University of Technology, 2004

[9] IBM, *CoreConnect™ bus architecture*, http://www-3.ibm.com/chips/products/coreconnect/

[10] C. E. Shannon, *A Mathematical Theory of Communication*, Bell System Technical Journal, 27: pp. 379-423, 623-656, 1948

[11] Xess Corp. *XSV Board V1.1 Manual*, 2001, www.xess.com

[12] Jamro E. Waitr K. *Heterogeneous Hardware-Software Prototyping System for PC-controlled FPGA-based Designs*, Proc. of IFAC Workshop on Programmable Devices and Systems PDS, Cracow, Nov. 18-19 2004, pp. 186-191

[13] Garcia E. *Implementing A Histogram for Image Processing Applications*, Xcell Journal Online, Xilinx: xcell38_46.pdf.