A Hardware Architecture of a Counter-Based Entropy Coder

Armein Z. R. Langi [1,2]

[1]Research Center on Information and Communication Technology
[2]Information Technology RG, School of Electrical Engineering and Informatics
Institut Teknologi Bandung, Bandung 40116, Indonesia

Abstract. This paper describes a hardware architectural design of a real-time counter based entropy coder at a register transfer level (RTL) computing model. The architecture is based on a lossless compression algorithm called Rice coding, which is optimal for an entropy range of $1.5 < H < 2.5$ bits per sample. The architecture incorporates a word-splitting scheme to extend the entropy coverage into a range of $1.5 < H < 7.5$ bits per sample. We have designed a data structure in a form of independent code blocks, allowing more robust compressed bitstream. The design focuses on an RTL computing model and architecture, utilizing 8-bit buffers, adders, registers, loader-shifters, select-logics, down-counters, up-counters, and multiplexers. We have validated the architecture (both the encoder and the decoder) in a coprocessor for 8 bits/sample data on an FPGA Xilinx XC4005, utilizing 61% of F&G-CLBs, 34% H-CLBs, 32% FF-CLBs, and 68% IO resources. On this FPGA implementation, the encoder and decoder can achieve 1.74 Mbits/s and 2.91 Mbits/s throughputs, respectively. The architecture allows pipelining, resulting in potentially maximum encoding throughput of 200 Mbit/s on typical real-time TTL implementations. In addition, it uses a minimum number of register elements. As a result, this architecture can result in low cost, low energy consumption and reduced silicon area realizations.

Keywords: *counter-based coder; lossless compression; hardware architecture; RTL.*

1 Introductions

Real-time lossless compression hardware will be a part of next generation computing chips (very large scale integrated circuits, VLSI) to sustain higher data rates over on-board limited channel and storage capacities. They include processor, network, and storage chips. With a reduced number of bits, limited capacity of transmission channels or storage can be used effectively. Such a reduction has a direct impact on complexity reductions, cost reductions, as well as overall system reliability improvements [1]. For example, space explorations by National Aeronautics and Space Administration (NASA) missions generate huge science data, requiring real-time compression [2]. Consequently,

international bodies such as Committee for Space Data Systems (CCDS) have defined compression standards for real-time systems [3].

A compression scheme consists of an encoder and a decoder. As shown in Figure 1, an encoder receives input data in a form of 8 bit samples (parallel), and produces output bitstream (serial) using an encoding algorithm. The number of bits to represent the bitstream is fewer than that of the input samples, hence it achieves compression. When the original samples are needed, a decoder shown in Figure 2 receives the bitstream and converts it back using a decoding algorithm into 8-bit samples losslessly to be used as intended. The scheme performs correctly if the decoder output data are identical to the encoder input data.
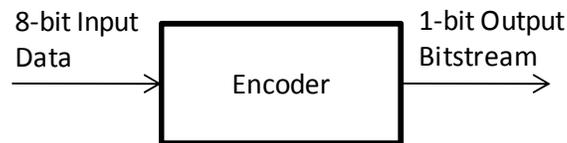
Figure 1 An encoder receives 8-bit input data and converts them into 1-bit output bitstream, with total number of bits in the bitstream is fewer than that of the input data.
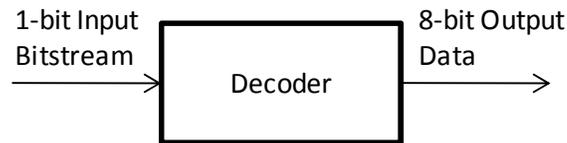
Figure 2 A decoder receives the bitstream from the encoder and converts it back to 8-bit output data, which are identical to the encoder's 8-bit input data.

Compression hardware requires high performance lossless compression schemes, with fast processing time, implemented using minimum hardware resources, as well as low energy consumptions. As a result, compression hardware is still expensive to implement. Typical lossless compression schemes, such as Huffman and arithmetic coding, require sophisticated data ordering as well as multiplicative computation [1]. In other words, compression algorithms usually demand processor-based computing models. Consequently a valid architecture must incorporate processors as well as multipliers and memory. They always demand expensive silicon areas, prohibiting them to be used widely in intended computing chips.

Recently we have developed a counter-based compression scheme [4] utilizing Rice coding [2]. We have shown that its performance is comparable to that of Huffman coding while its complexity is promisingly much lower due to its use of simple counters. Furthermore, the counter coder can be designed to be optimal on an entropy range of $1.5 < H < 7.5$ bits per sample, an expected and reasonable range of data entropy encountered in typical 8-bit (or its multiple) computing chips. In rare cases when the chips must deal with entropy ranges below 1.5 bits per sample, simple reversible preprocessing schemes can be added to bring the sample entropy range into a $1.5 < H < 7.5$ range. Hence we propose the use of a counter coder for compression hardware such as VLSI.

To implement the counter codes into hardware or VLSI chips, we need an architectural design. A valid architecture must always perform an input-output relation correctly. However the main features of good hardware architecture are in its optimization of speed performance, efficient area usages, and efficient power consumptions. It must also be easy to implement, meaning the architecture should be described in enough details for straightforward implementations. A register transfer level (RTL) computing model provides such implementable details, while still abstract enough to allow understanding of the inner working of the implementation.

This paper presents a hardware architectural design of a real-time lossless compression scheme for a purpose of hardware implementations in computing chips. We have designed an RTL computing model of Rice coding utilizing simple counters in an entropy coding scheme. The RTL computing model allows us to use registers, simple counters and standard logic gates to ensure low energy and silicon area requirements. The scheme uses a pipelined architecture to increase data throughput.

The paper first describes the counter coder algorithm as a requirement for architectural design. It then proposes and describes a novel data structure to allow robust bitstream, pipelining processing, and simple hardware implementation. Using the algorithm and the data structure, this paper presents the hardware architecture at an RTL for both the encoder and decoder, and shows that the architecture can be implemented using standard logics without any use of any processors, memory, or multipliers. We have validated the architecture in a coprocessor design, implemented using an FPGA platform. Finally the paper discusses the resulting architecture, and provides concluding remarks.

2   A Counter Code Algorithm

To design the architecture, we must first understand the counter code algorithm and then use it as a design requirement. A basic counter coder called Ψ_1 (or sometimes also called PSI–1) works as follows [2]:

1. Given a block of data samples (for $j = 1, ..., J$), coder Ψ_1 assumes that each sample takes a symbol s_i, for $i = 1, ..., 256$, as shown in Table 1.

2. Coder Ψ_1 treats each sample symbol s_i having sample data d_i as a non-negative integer number x_i. The average length of sample data is R = 8 bits per sample.

3. For every sample in the block (having a symbol s_i, thus having sample data d_i), a Ψ_1 encoder converts d_i into a codeword w_i of a length $l_i = x_i + 1$, consisting of x_i consecutive zero bits '0' followed by a closing one bit '1' (see again Table 1). For example, if a sample happens to be d_i = 0000 0011, it must have x_i = 3, and the Ψ_1 then encodes it using 4 bits, i.e., 3 zeros followed by a one. Hence, the encoding algorithm (converting d_i into w_i) is simply down counting, which is summarized in Table 1.

4. The reconstruction is obviously simple counting too. Given a codeword w_i, a Ψ_1 decoder just counts the number of zero bits until a one appears. The counting result is the sample value x_i. Using Table 1, it determines that the codeword belongs to a symbol s_i, hence it produces the sample data d_i as its output.

Table 1   A codeword table of Ψ_1 Code.

i	Symbols s_i	Sample Data d_i	Numbers x_i	Codewords w_i	Length l_i
1	s_1	0000 0000	0	1	1
2	s_2	0000 0001	1	01	2
3	s_3	0000 0010	2	001	3
4	s_4	0000 0011	3	0001	4
…	…	…	…	…	…
256	s_{256}	1111 1111	255	0000…00001	256

It has been studied elsewhere [2] that this Ψ_1 code in Table 1 is optimal for a monotonically decreasing distribution source with a first-order entropy around 2, i.e., $1.5 < H < 2.5$. Optimal means the number of bits to represent a block of J samples $\{d_{i(1)}, d_{i(2)}, \cdots, d_{i(J)}\}$, which is

$$L = J + \sum_{j=1}^{J} x_{i(j)} \tag{1}$$

is comparable to the total entropy of the data in the block. Such a distribution ensures that shorter length codewords occure more frequently, achieving an effective compression, i.e., $L/J < R$. We say that the optimal range is the natural entropy range for Ψ_1. For input samples within this natural entropy, Ψ_1 performance is comparable to that of a Huffman coder [4].

For input samples with entropies outside that range, Rice introduced a concept of word splitting. If $H > 2.5$, it is safe to assume that the least significant bits (LSBs) of sample data $d_{i(j)}$ are completely random. In this case there is no need to perform any compression on those LSBs. An encoder can then split $d_{i(j)}$ into two portions: k LSBs and $(8\text{-}k)$ most significant bits (MSBs). The MSBs are coded using Ψ_1 encoder before being sent to bitstream, while the LSBs are sent uncoded. A decoder must first recovers the MSBs using Ψ_1 decoder, and then concatenates the results with the uncoded LSBs, resulting in the desired $d_{i(j)}$.

This counter compression of word splitting code (into k LSBs and $(8\text{-}k)$ MSBs) is called $\Psi_{1,k}$. It has been shown in [3] that $\Psi_{1,k}$ has a natural entropy range of $1.5 + k < H < 2.5 + k$. It should be noted, computing data of an 8-bit resolution usually has an entropy range of $1 < H < 8$. For $H = 8$, there is no need for any lossless compression. For $H < 1$, one can employ simple reversible preprocessing, such as zero run length code, to bring the data entropy into the $1 < H < 8$ range. Hence, for our purpose of general 8-bit computing we can limit $\Psi_{1,k}$ for $0 \le k \le 7$.

Given a block of data samples (for $j = 1, ..., J$), the $\Psi_{1,k}$ coder must then have a mechanism to estimate the entropy of the block to ensure it uses the optimal k. Rice has come with an estimation rule of thumb based on sum of x_i values in the block [2]. Since we assume that the data samples has a source according to

Table 1 with statistics of a monotonically decreasing distribution, x_i with low values are likely to occur. The lower the entropy, the more the distribution is skewed toward lower value x_i (i.e., lower L). As a result, a block with lower entropy will have a smaller sum of x_i values in the block. In other words, average entropy in a block can be estimated from L/J. This rule of thumb is shown in Table 2. A small value of L/J is reflected in a smaller sum of x_i, corresponding to a low entropy value, hence a small selected k.

As mentioned before if the samples do not have such a characteristic, i.e., a non-negative and monotonically decreasing distribution, there are several simple preprocessing schemes available to preprocess the samples to comply with the characteristics.

Table 2   A rule of thumb to estimate block entropy for $J = 8$ and $n = 8$ for 8 options of coders (adapted from [2]).

Decision range	Sum of x_i values	Entropy	k
$L/J \leq 5/2$	0 – 12	1.5	0
$5/2 < L/J \leq 9/2$	13 – 28	2.5	1
$9/2 < L/J \leq 17/2$	29 – 60	3.5	2
$17/2 < L/J \leq 33/2$	61 – 125	4.5	3
$33/2 < L/J \leq 65/2$	126 – 252	5.5	4
$65/2 < L/J \leq 129/2$	251 – 508	6.5	5
$129/2 < L/J \leq (128n–831)/2$	509 – 764	7.5	6
$(128n–831)/2 < L/J$	765 – 1023	8	8

3   A Design of a Robust Data Structure and an RTL Computing Model

Having described the algorithms to be implemented, we must now design the data structure for the hardware architecture. Given original data of N consecutive samples (Figure 3(a)), we preprocess the samples into N non negative numbers having a monotonically decreasing distribution (Figure 3(b)) to ensure entropy compliance of the input data. Furthermore, we split the data into M consecutive blocks of J samples shown in Figure 3(c). If N is not divisible by J, the last block M will not have J samples. Zero samples are then added to the last block to ensure that the last block will have J samples.

The encoder then works on an individual block consecutively to produce bitstream. A bitstream consists of M individual bitstream blocks called CW-1, CW-2, to CW-M, corresponding to M sample blocks (see Figure 3(d)). Each

bitstream block (CW-i) consists of a codeword indicating the coder $\Psi_{1,k}$ used, and then J sample codewords representing J samples in current block correspondingly (see Figure 3(e)). Assuming there are 8 possible coders $\Psi_{1,k}$, the codeword indicating which coder used requires three bits, with its corresponding splitting k as in Table 2. A sample then consists of k bits of LSB, followed by encoded MSB (see Figure 3(f)). The encoded MSB are codewords w_i in Table 1, consisting of several bits '0' and a closing bit '1'.
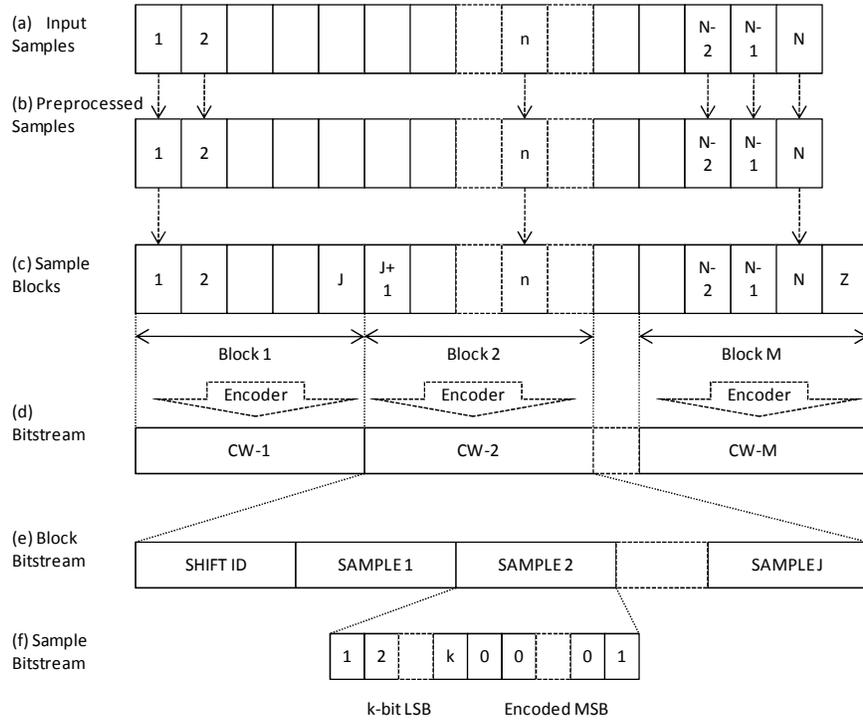
Figure 3   A simple data structure for counter coder.

Such a data structure has desired robustness due to the following features and benefits:

1. Although the code becomes variable length, it is uniquely decodable. All data components in the bitstream are either of fixed lengths or having closing bits.

2. All samples are coded independently, thus can also be recovered independently. This allows random accesses of individual compressed samples.

3. Encoding errors of one block can be isolated and do not propagate to the next block.

We can now have an RTL computing model working on the data structure to implement the counter code algorithms. An encoder can use a *buffer* to capture a block of *J* samples. The encoder then estimates the block entropy by accumulating the values of all samples within the block buffer. An *adder* and a *register* can perform as an *accumulator*. Using Table 2, a *combinatorial logic* can estimate the block entropy, especially in a form of the shift factor *k*. Using the shift factor *k*, a *parallel-to-serial loader-and-shifter* can split a sample into MSBs and LSBs. The *loader-and-shifter* shifts *k* LSBs into a bitstream *multiplexer*. The encoder also needs to produce '0' bits as many as the sample MSB value. A *down counter* can accomplish this process. The *multiplexer* can concatenate the LSB bits with the encoded MSB into bitstream.

An RTL computing model for the decoder is even simpler. A *demultiplexer* identifies the shift factor *k* in the bitstream and uses it to extract correct LSBs from the bitstream. The encoded MSBs triggers an *up-counter*, resulting in MSB sample values. A *bit-concatenator* can then combine all MSBs and LSBs into a decoded sample.

4   A Hardware Architecture Based on the RTL Model

Having designed the data structure and its corresponding computing model, we can now present the hardware architecture, consisting of an encoder and a decoder. Figure 4 shows architecture of an encoder at an RTL level. The encoder consists of an 8 bit FIFO BUFFER of size *J*, an 8-bit ADDER, an 8-bit REGISTER, a parallel-serial (P-S) LOADER SHIFTER, a combinatorial circuit SELECT LOGIC, a DOWN COUNTER, and a MULTIPLEXER. It accepts *J* samples of one block input data and produces a bitstream block CW-*i*.

With references to Figure 3 and 4, the encoder works as follows. Preprocessed samples enter the FIFO BUFFER and 8-bit ADDER in parallel. The BUFFER accommodates *J* samples (with a typical *J* = 8), while the ADDER accumulates the sum of those *J* numbers into REGISTER. After all *J* samples have been accumulated, the SELECT LOGIC reads the content of REGISTER to decide SHIFT ID and SHIFT COUNT using a rule described in Table 3. The encoder then sends SHIFT ID to MULTIPLEXER as a start of CW-*i* bitstream block. Figure 3(e) shows their positions in the bitstream.

Error! Not a valid link.

Figure 4   An architecture of the encoder.

Table 3   A rule table of SELECT LOGIC.

REGISTER Contents	SHIFT ID	SHIFT COUNT
00 0000 0101 – 00 0000 1100	000	0
00 0000 1101 – 00 0001 1100	001	10
00 0001 1101 – 00 0011 1100	010	110
00 0011 1101 – 00 0111 1100	011	1110
00 0111 1101 – 00 1111 1100	100	11110
00 1111 1101 – 01 1111 1100	101	111110
01 1111 1101 – 10 1111 1100	110	1111110
10 1111 1101 – 11 1111 1111	111	111111110

Having decided on which $\Psi_{1,k}$ coder to use, the encoder now ready to produce bitstream of each sample, using the following steps:

1. It first loads one sample from FIFO BUFFER into P-S LOADER SHIFTER. Using the SHIFT COUNT information, the P-S LOADER SHIFTER starts to shift out LSB BITS of the P-S LOADER SHIFTER into MULTIPLEXER.

2. The remaining MSB BITS in the P-S LOADER SHIFTER are then loaded in parallel into DOWN COUNTER. This DOWN COUNTER starts down counting until it reaches zero. For each count, the COUNTER produces bit '0' into the multiplexer. When the counter is empty, it produces a bit '1' as a closing mark, to complete the processing of one sample.

The encoder repeats these steps for the remaining samples in the block. This then completes the conversion of one block of samples into bitstream CW-*i*.

To reconstruct the samples from the bitstream, Figure 5 shows an RTL architecture of decoder, consisting simply of DEMULTIPLEXER, SELECT ID LOGIC, UP COUNTER, and CONCATENATOR. For each block CW-I received by the decoder, DEMULTIPLEXER splits the block to obtain SHIFT ID. This is used to determine SHIFT COUNTS using Table 1, and then consequently the number of bits for LSB BITS.

Error! Not a valid link.

Figure 5   Architecture of the decoder.

The decoder is then ready to recover each sample for 1 to *J*. It first recovers the LSB bits and puts them into CONCATENATOR. The decoder then uses each bit '0' of the MSB stream to upcount the UP COUNTER, and stops counting when it encounters the closing bit '1'. The MSB bits are then available in the UP COUNTER and sent to CONCATENATOR, which together with the LSB bits become the decoded sample.

The decoder repeats the process until it recovers all *J* samples in a given block.

5   Simulation Results of the Hardware Architecture

Having defined the algorithm and architecture of the coder, we can now validate the architecture. We have implemented and validated the architecture on a C++ simulation model. Using C++ simulations, we can study the validity of the architecture with image compression data. Here we use data samples from a wavelet image compression scheme [6]. To achieve monotonically decreasing distribution, the data samples are preprocessed using wavelet scalar quantization (WSQ) described in [7], with a slight modification to obtain non-negative sample data with an entropy range of 1 to 8 bits per sample. Later the simulations also provide verification data for an FPGA implementation.

From the simulation results, we observe several advantages of this architecture. First, the architecture is optimal. A compression scheme is optimal if it can achieve a compression result with an efficient bits-per-sample performance down to a level of its entropy. Using the WSQ scheme, we can vary prescribed bitrates of the quantization, resulting in preprocessed sample data having corresponding sample entropy values, ranging from 1 to 8 bits per sample. We then apply the coder to those sample data, and measure the bitrates of the bitstream. The results are compared to entropy values of the sample data, shown in Figure 6. The bitrates coincide with sample entropy values, confirming that the coder is optimal and the architecture performs an optimal compression.
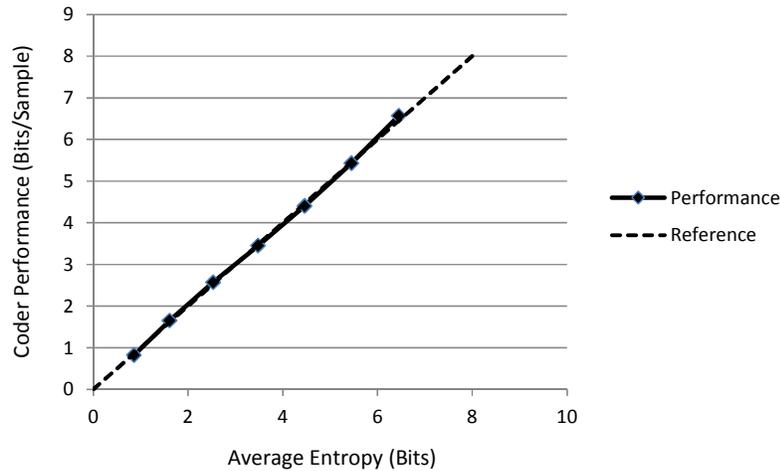
Figure 6 The coder performance is optimal because it produces bitstream having bits per sample as efficient as sample entropy.

Second, the architecture allows localized error handling. Samples are grouped into independent blocks (see Figure 3 (e)). As a result, errors in one block do not propagate into another block. This is desirable as channels are not always free from noise.

Third, the block size can be controlled and preconfigured optimally to suit various applications having different block sizes.

6   An FPGA Implementation, Results, and Discussions

Having validated the algorithm and architecture, we can now design a system to implement the coder on an FPGA platform as an example of the architecture implementation. First we define design assumptions, especially its external operating environment. We then choose a basic coprocessor computing model. The model consists of a control unit (CU) and a data path unit (DPU). The encoder and decoder DPUs implement our architectures in Figures 4 and 5, respectively. Later we describe input-output assignments to facilitate interactions of the coprocessor with its external environment (see also [5]).

6.1   Design Assumptions

We assume in a real environment, the coder consists of an encoder and a decoder separated physically by a long distance communication channel (see Figure 7). The channel is bit oriented, meaning data are transferred one way

from encoder to decoder bit-by-bit serially. Data input of the encoder are available locally, stored in memory. Encoder can access the memory through a bus oriented channel. Similarly, output data of the decoder must be stored to memory for further use.
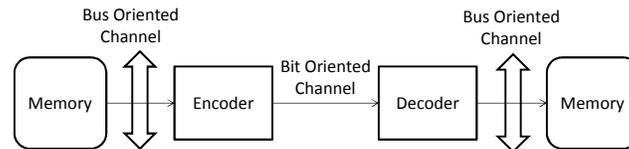
Figure 7 The encoder accepts input data from parallel bus memory and producing bitstream to serial channel, and decoder does the other way around.

6.2 A Basic Coprocessor Computing Model

We use a basic coprocessor computing model, shown in Figure 8, to satisfy the above environmental requirements. The coprocessor interacts with three major external subsystems: Host Processor, Memory, and Channel I/O. The coprocessor consists of a CU and a DPU. A host processor ultimately controls the coprocessor, giving commands to the coprocessor to perform its functions.
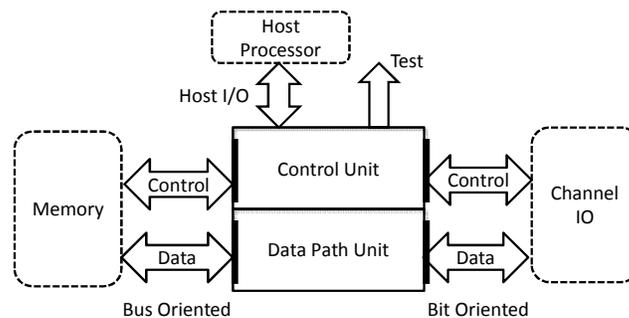
Figure 8 The coprocessor interacts with control oriented host processor, bus oriented memory, and bit oriented channel I/O.

Both the encoder and decoder use the same computing model. An encoder DPU gets input data from memory, performs the actual data compression, and sends data to channel I/O. Conversely, a decoder DPU gets input from channel, performs decoding processes, and stores the results into memory.

The CU manages and controls the DPU to ensure synchronized interactions with the external subsystems. The CU accepts commands and giving status

signals to the host processor through Host I/O. Interactions with memory are controlled through bus control signals. Channel control signals manage interactions with the transmission channel. Optionally, we provide test signals for testability purposes.

To ensure synchronized interactions with external subsystems, we design the encoder and decoder coprocessor to interact with various signals. Through the signals the coprocessor interacts with host processor, memory, and channel I/O. Optionally, we can observe internal working of the coprocessor through test pins.

6.3   Results and Discussions

We have implemented the Rice coder (both encoder and decoder) as a coprocessor for 8 bit/sample data on an FPGA Xilinx XC4005. One XC4005 contains 196 combinatorial logic units (CLU) and 112 user I/O pins. In our implementation, the encoder uses 30% CLB F&G, 15% CLB H, 16% CLB FF, and 34% I/O pins. The decoder uses 31% CLB F&G, 19% CLB H, 16% CLB FF, and 34% I/O pin. Hence, an X4005 is sufficient to implement both encoder and decoder. Furthermore in this particular implementation, the encoder and decoder can achieve 11.6 MHz and 19.4 MHz clock rates, respectively. Since a 10 MHz clock rate corresponds to a 1.5 Mbits/s throughput, the FPGA implementation achieves 1.74 Mbit/s and 2.91 Mbits/s for the encoder and the decoder, respectively.

From the FPGA implementation, we observe further advantages of this architecture. First, the RTL descriptions show that the architecture is easy to implement in terms of resources requirements. This architecture can be implemented easily using standard registers and counters, shown in Figures 4 and 5. A complete encoder and decoder system use less than 196 CLUs.

In contrast, many compression schemes require either high numbers of CLUs or processor level components. FPGA implementations of a lossless coder called LZW reported in [8] and [9] use more than 3000 slices. Since one slice implements 32 CLUs, they use an order of 100.000 CLUs. Other compression FPGA implementations reported in [10] and [11] requires an additional DSP processor and more than one high-capacity FPGA unit.

Second, it is possible to implement pipelining using double buffers. When the pipeline critical path is fully operational, the system can run at a maximum speed. This speed is limited only by counter delay as a bottleneck. Fast counters can be implemented to achieve a level of 5 ns processing delay (excluding 2 buffer delays) on TTL platforms. This results in ability to process real-time data

up to 200 Mbit/s. A faster platform would result in a proportionally faster throughput.

It should be noted that a VLSI chip set implementation of Rice coder on 1.0-μm CMOS process by J. Venbrux, P.-S. Yeh, and M. N. Liu has been reported earlier in [12]. They reported that both the encoder and the decoder require 71000 transistors. The chip sets use data samples at resolutions of 4 to 14 bits, achieving operating rates of 50 Msamples/s and 25 Msamples/s for the encoder and decoder, respectively.

7   Concluding Remarks

This paper has described hardware architecture of a counter-based lossless compression scheme using an RTL computing model. The system consists of an encoder and a decoder. The encoder and decoder use a buffer, an adder, registers, logics, combinatorial logics, as well as counters, removing any need to use area expensive processors, memory and multipliers. The architecture is suitable and optimal for non-negative samples having monotonically decreasing statistics, with an entropy range between 1 to 8 bits per samples. It uses a table to estimate data entropy quickly. We have validated and verified the architecture using C++ simulations and an FPGA implementation. We demonstrated that an XC4005 FPGA is more than sufficient to implement the architecture of both the encoder and decoder, reaching throughputs of 1.74 Mbit/s and 2.91 Mbit/s for the encoder and the decoder, respectively. The resulting architecture is suitable for real-time hardware implementation, potentially up to a 200 Mbit/s throughput through double buffer pipelining. Future works include modifying the architecture to suit system on chip (SoC) applications and implementations.

Acknowledgements

Nomenclature

FPGA	=	Field Programmable Gate Arrays
RTL	=	Register transfer level
VLSI	=	Very large scale integrated circuits
MSB	=	Most significant bits
LSB	=	Least significant bits

References

[1]   Langi, A.Z.R., *Review of Data Compression Methods and Algorithms*, Technical Report, DSP-RTG–2010–9, Institut Teknologi Bandung, September 2010.

[2]   Rice, R.F., *Some Practical Universal Noiseless Coding Coding Techniques, Part III, Module PSI–14,k+*, JPL Publication 91–3, NASA, JPL California Institute of Technology, p. 124, November 1991.

[3]   CCSDS, *Image Data Compression*, Recommended Standard CCSDS 122.0-B-1, Consultative Committee for Space Data Systems, November 2005. (available at http://public.ccsds.org, accessed 4 March 2011)

[4]   Langi, A.Z.R., *Lossless Compression Performance of a Simple Counter-Based Entropy Coder*, ITB Journal of Information and Communication Technology, **5**(3), 2011.

[5]   Langi, A.Z.R., *An FPGA Implementation of a Simple Lossless Data Compression Coprocessor*, Proc. International Conference on Electrical Engineering and Informatics (ICEEI) 2011, Bandung, July 2011.

[6]   Langi, A.Z.R. & Kinsner, W., *Wavelet Compression for Image Transmission through Bandlimited Channels*, ARRL QEX Experimenters's Echange, (ISSN: 0886–8093, USPS 011–424), **151**, pp. 12–21, September 1994.

[7]   Bradley, J.N. & Brislawn, C.N., *The Wavelet/Scalar Quantization Compression Standard for Digital Fingerprint Images*, Proc. IEEE Int. Symp. Circuits and Systems, London, May 3–June 2, 1994.

[8]   Cui, W., *New LZW data compression algorithm and its FPGA implementation*, Proc. 26th Picture Coding Symposium (PCS 2007), November 2007.

[9]   Heliontech.com, *Compression Systems*, (available at http://www. heliontech.com/comp_sys.htm, accessed August 24, 2011).

[10] Jilani, S.A.K. & Sattar, S.A., *JPEG Image Compression Using FPGA With Artificial Neural Networks*, IACSIT International Journal of Engineering and Technology, **2**(3), p 252-257, June 2010.

[11] Kingh, S.N., Kumar, J., Rajan, R. & Panigrahi, S., *Hardware Image Compression with FPGA*", ACEEE International Journal of Recent Trends in Engineering, Academy Publisher, **12**(8), pp. 33-35, November 2009.

[12] Venbrux, J., Yeh, P.-S. & Liu, M.N., *A VLSI Chip Set for High-Speed Lossless Data Compression*, IEEE Transactions on Circuits and Systems for Video Technology, **2**(4), pp. 381-391, December 1992,.