A 24 BIT DSP FOR STACK-RUN CODEC†

*P. Raffy**, *P. Nus***, *J.M. Moureaux***

* ISL - Department of Electrical Engineering / Stanford University
Stanford, CA 94305 - USA
e-mail: raffy@isl.stanford.edu

** CRAN - CNRS / Université Henri Poincaré, Nancy 1
11, Rue de l'Université, F-88100 Saint-Dié - France
e-mail: nus@cran.iutsd.u-nancy.fr - moureaux@cran.iutsd.u-nancy.fr

ABSTRACT

Stack-run is a recent lossless method of compression developped for low bit rate coding. It plays the same role as the conventional run length codec, in the sense it also exploits zeros resulted from quantization. However, stack-run coding outperforms run length coding in terms of rate-distortion trade-off, as well as low complexity. In this paper, we propose the implementation of stack-run codec using a low cost and low complexity 24 bit processor.

1. INTRODUCTION

Nowadays, the increasing demand for real-time applications requires the further development of low bit rate compression algorithms. Stack-run is a lossless method of coding [5] which appears as an efficient alternative to recent zerotrees. Among the most desirable features of the stack-run method are independent subband coding and lower addressing complexity, compared with a two-dimensional zerotree quantizer. Independency between subbands may be valued for several reasons including simplified robustness to transmission channel errors and parallel encoding and decoding. Embedded or dependent bit-streams require more complex embedded error correcting and detecting codes for efficiency. At practically low bit rates, the dependency between subbands does not appear to be very significant, allowing independence without much sacrifice in performance. In contrary, optimal bit allocation can be successfully applied and the influence of the wavelet coefficients increased or decreased at different resolutions if desired, depending on the application. This approach is shown to better preserve high frequency coefficients than zerotrees [2]. Furthermore, the use of stack-run coding to commonly used images demonstrates high performance of wavelet coders, especially for low bit rate applications (2 dB of PSNR improvement on average over JPEG [2]). It is noteworthy that there is nothing wavelet-specific about stack-run. Indeed, it can also improve JPEG or MPEG algorithms based on the block DCT.

Simulation results using an entropy estimation indicate that stack-run is PSNR competitive with classical run length using a very large alphabet. However, run length gives, in this case, poor results when followed by an adaptive entropy coder. Contrary to run length, stack-run codes amplitudes and run-lengths using a symbol alphabet of only four distinct letters so that an instantaneous and adaptive arithmetic coding can be efficiently applied. Therefore, its implementation may be of interest in many compression applications. It is the case for example of remote applications (like satellite imaging), where complexity must be as low as possible.

In this paper, we propose an original implementation of the stack-run codec based on a 24 bit DSP. This architecture presents several attractive advantages. First, it is a low cost and low complexity implementation. And second, this approach ensures flexibility in two ways:

(i) this DSP-based architecture can be easily inserted into more complex compression schemes based on a DSP design [3], and

(ii) the proposed structure can either process the encoding of all the subbands of an image in a serial manner or can be parallelized to each subband in order to increase the encoding speed.

Results show that the proposed implementation is very efficient in term of computational time.

This paper is organized as follows. In Section 2, we explain the principle of stack-run coding through the construction of the code using a toy example. In Section 3 we describe the different features of the proposed DSP-based implementation. Finally, we indicate performances in terms of computational complexity, and summarize our major contributions. Note that, the design of both encoder and decoder are addressed in Section 3.

2. STACK-RUN CODING

2.1. Principle

As runlength codec, stack-run uses two classes of coefficients, depending on whether these are clusters of zero-valued coefficients (or *runs*) or non-zero valued coefficients (referred to as significant or *levels*). Every quantized image can be

† This material is partially based upon work supported by the *Ministère des Affaires Etrangères* under LAVOISIER's grant (1997).

decomposed into *runs* and *levels*. Let us illustrate this on the following example:

Quantization stream: … 0 0 0 0 0 0 0 -1 0 0 0 0 5 10 0 0 …

This stream can be equivalently represented by:

$$\overbrace{7x0}^{run}\ \overbrace{-1}^{level}\ \overbrace{4x0}^{run}\ \overbrace{+5}^{level}\ \overbrace{+10}^{level}\ \overbrace{2x0}^{run}$$

Let us introduce stack-run coding. Each *run* or *level* can be represented by a binary stack of 0 and 1. In the first case, *runs* contain the number of consecutive zeros. In the second case, *level* is the value of the significant coefficient. The sign information is preserved thanks to symbols $\{+,-\}$. For each category, *run* or *level*, symbols can be arranged from left to right from the least significant bit (LSB) to the most significant bit (MSB). This yields:

$$\textbf{LSB} \longrightarrow \textbf{MSB:}\ \overbrace{1\,1\,1}^{run}\ \overbrace{1\ -}^{level}\ \overbrace{0\,0\,1}^{run}\ \overbrace{1\,0\,1\ +}^{level}\ \overbrace{0\,1\,0\,1\ +}^{level}\ \overbrace{0\,1}^{run}$$

Finally, we have a 4-ary symbol alphabet composed of $\{0,1,+,-\}$. Symbols "0" and "1" are affected to *levels* and represent binary values 0 and 1. Symbols "-" and "+" are kept for *levels* and also reserved for *runs* to replace binary values 0 and 1. Converted on our example, this yieds:

$$\overbrace{+\ +\ +}^{run}\ \overbrace{1\ -}^{level}\ \overbrace{-\ -\ +}^{run}\ \overbrace{1\,0\,1\ +}^{level}\ \overbrace{0\,1\,0\,1\ +}^{level}\ \overbrace{-\ +}^{run} \qquad (1)$$

2.2. Improvements of the code

According to abovementioned rule, we can establish the following *run table*:

runs	1x0	2x0	3x0	4x0	5x0	6x0	7x0	8x0
bin.wd	1	01	11	001	101	011	111	0001
τ_{run}	+	-+	++	--+	+-+	-++	+++	---+

At this stage, it is possible to improve the efficiency of this code by limiting the number of symbols to be coded. This is done for both classes, *runs* and *levels*. Indeed, we notice that it is possible to drop the MSB of the τ_{run} stream except for *runs* whose length expresses as $2^k-1,\ k \in \boldsymbol{N}$. In fact, it is obviously impossible to drop, for example, the only one bit of the "1x0" sequence. In order to distinguish some specific *runs*, it is necessary to preserve the MSB for *runs* whose length is proportional to 2^k-1. The final *run* table becomes:

runs	1x0	2x0	3x0	4x0	5x0	6x0	7x0	8x0
bin.wd	1	01	11	001	101	011	111	0001
τ_{run}	+	-	++	--	+-	-+	+++	---

Let us consider now the *level* table. According to §2.1, it can be written as follows:

levels	-4	-3	-2	-1	+1	+2	+3	+4
τ_{level}	001-	11-	01-	1-	1+	01+	11+	001+

We notice that we cannot straightforwardly replace the MSB by the sign information since *levels* "-1" and "+1" would be then indistinguishable from the *runs* "2x0" and "1x0" respectively. The solution consists first in incrementing by 1 the absolute value of each *level*, and second in dropping the MSB. Thus, the final *level* table is:

levels	-4	-3	-2	-1	+1	+2	+3	+4
τ_{level}	10-	00-	1-	0-	0+	1+	00+	10+

If we consider only one stream composed of *runs* and *levels*, this code is uniquely decodable. However, its efficiency can be improved by considering two separate streams, τ_{run} and τ_{level}. This requires some further adaptation in order to get this code be uniquely decodable.

2.3. Uniquely decodable code

Let us reconsider our example according to the modifications mentioned in section 2.2.

$$\overbrace{+\ +\ +}^{run}\ \overbrace{0\ -}^{level}\ \overbrace{-\ -}^{run}\ \overbrace{0\,1\ +}^{level}\ \overbrace{1\,1\,0\ +}^{level}\ \overbrace{-}^{run}$$

Runs and *levels* are grouped into two streams, τ_{run} and τ_{level} which are then individually entropy coded. However, decoding is impossible because we cannot distinguish the level-run transition. Indeed, symbols "+" and "-" are used by both alphabets. This problem is solved by incorporating the LSB of each *level* to the τ_{run} stream. Thus, every last symbol of τ_{level} is automatically followed by a symbol belonging to τ_{run}. On our example, this yields:

$$\overbrace{\underbrace{+\ +\ +}_{\tau_{run}}}^{run}\ \overbrace{\underbrace{0}_{\tau_{run}}\ \underbrace{-}_{\tau_{level}}}^{level}\ \overbrace{\underbrace{-\ -}_{\tau_{run}}}^{run}\ \overbrace{\underbrace{0}_{\tau_{run}}\ \underbrace{1\ +}_{\tau_{level}}}^{level}\ \overbrace{\underbrace{1}_{\tau_{run}}\ \underbrace{1\,0\ +}_{\tau_{level}}}^{level}\ \overbrace{\underbrace{-}_{\tau_{run}}}^{run}$$

Hence, the streams to be transmitted are the following:

$$\tau_{run}:\ +\ +\ +\ 0\ -\ -\ 0\,1\ -\ \ldots$$

$$\tau_{level}:\ -\ 1\ +\ 1\,0\ +\ \ldots$$

There exists a dominant sub-alphabet for each stream. τ_{run} is dominated by "+" and "-" symbols when τ_{level} has predominant "0" and "1" symbols. This property is widely exploited by the use of entropy coding (arithmetic coding) for each stream.

3. DSP IMPLEMENTATION

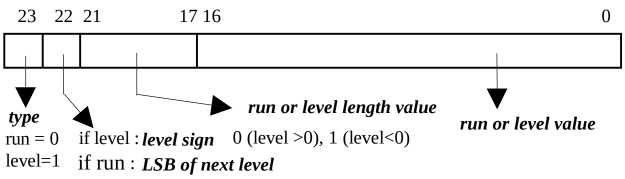
In this section, we discuss the proposed architecture for the stack-run codec. First, we give the choice of the processor and its memory organization. Then, we explain the proposed data structure for both *run* and *level* codes, as well as the general organization of the program which takes into account all the steps of the encoding process given in section 2.

3.1. Choice of the processor and memory organization

As shown in section 2, stack-run coding does not use any floating point numbers [4]. This naturally leads us to use a fixed point processor (like the 56002 from Motorola chosen here). The main advantage of this family of processors is their low cost, which authorizes high parallelism, for instance [1]. The 24-bit format of the chosen DSP is particularly well suited to *run* and *level* data structures (as we will see in the next subsection). The processor used here has three independent memory fields:
Data X: 64Kx24 bits off-chip RAM,
Data Y: 64Kx24 bits off-chip RAM,
Program P: 512x24 bits on chip RAM.
We store final *runs* in *X* and *levels* in *Y*. This structure enables the encoding of 64K *runs* and 64K *levels* which is enough for most applications, but can be easily extended if necessary by using additional external RAM.

3.2 Data structure

The proposed data structure is detailed in Figure 1. Let us call: T the field "type" of the data structure, RV (or LV) the "run or level value", RL (or LL) the "run or level length value", and RDS (or LDS) the whole *run* (or *level*) data structure (24 bits). The field "run or level value" allows *runs* of 131,071 zeros where 131,071 can be also the maximal *level* value. Note that since wavelet coefficients are distributed over the range -128 + 127, only 8 bits of this field are needed for *levels*. But for *runs*, we might be to preserve all the bits. Furthermore, the field "run or level length value" permits to decode easily the effective *run* or *level* value which is obviously variable. It is thus very



important in the whole process, since it always gives the effective length of the current *run* or *level* value. Finally, bit 21 of the structure can be used indifferently as the *level* sign or the LSB of next *level* (as explained in Section 2 for unique decodability purpose).

Figure 1: 24-bit data structure available for both *runs* and *levels*.

3.3 General organization of the program

Encoding: We assume that the initial values are stored in memory *X* in the data structure format described in Figure 1. They are represented in Table 1 and correspond to stream (1) of Section 2.1. The encoding process follows the progression described in Section 2. It consists in two steps which yield respectively table 2 and table 3:

Step1: improve the code as detailed in §2.2,

Step2: incorporate the LSB of each *level* into the previous *run* (see §2.3) and store *runs* in data *X* memory and *levels* in data *Y*.

In step 1, RL (or LL) is decremented only if RV (or LV) differs from $2^k - 1,\ k \in \mathbf{N}$. In step 2, the transfer of LSB(LV) to bit 22 of the previous RDS leads to decrease LL. Furthermore, when two *levels* are following, it is necessary to insert a new *run* (RDS) with all bits cleared except bit 22 which contains the LSB of the next *level* (see Table 3). Moreover, step 2 separates *run* streams from *level* streams, in order to be transmitted. Note that at this point, it is easy for any following stage (like arithmetic coding) to decode the effective *run* and *level* symbols of each 24 bit data structure. Indeed, τ_{run} (respectively τ_{level}) stream consists of effective bits of RV (respectively LV) and bit 23. The effective bits of RV (or LV) are easily decoded thanks to RL (or LL) which acts as a bit counter.

Decoding: Decoding is based on the same principle as encoding, so it is briefly described here. The data structure (represented in Figure 1) is unchanged. The initial data are now those represented in Table 3. Step 1 and step 2 (respectively duals of encoding step 2 and step 1) are performed on these data to transform Table 3 in Table 2, and finally in Table 1. RL and LL counters are incremented instead of being decremented (as in the encoding). Bit 22 of a *run* is transfered to LSB of the following *level* after a left shift of its value. Furthermore, a cleared RL in a *run* means two *levels* are following. In this case, bit 22 of the *run* is incorporated into the next *level* and the *run* data structure is cancelled.
Both encoding and decoding algorithms have been programmed in assembly language on a DSP56002 (from Motorola). In the next section, we show performance of the proposed implementation.

4. PERFORMANCE EVALUATION

The encoding and decoding programs require respectively only 96 x 24-bit words and 84 x 24-bit words. Thus, they can be easily stored in Program *P* memory field of the DSPs.
The computational complexity depends essentially on three parameters: *v*, *n*, and *r*, where:

- *v* is the *run* or *level* value (RV or LV),
- *n* is the number of bits of *v* equal to 1 (when $v = 2^k - 1, k \in \mathrm{N}$),
- *r* is the position of the first bit of *v* equal to 0 (when $v \neq 2^k - 1, k \in \mathrm{N}$).

Results of encoding are summarized in the following table:

	$v = 2^k - 1$	$v \neq 2^k - 1$
run	$O(4n+17)$	$O(4r+21)$
level	$O(4n+17)$	$O(4r+19)$

Table 4: Computational complexity of encoding (in cycle instructions) for step 1.

Results of decoding are summarized in the following table:

	$v = 2^k - 1$	$v \neq 2^k - 1$
run	$O(4n{+}16)$	$O\,(4r{+}24)$
	$v = 0$	$v \neq 0$
level	$O(21)$	$O\,(25)$

Table 5: Computational complexity of decoding in cycle instructions for step 2 (*runs* and *levels*). Note that the total computation time for decoding must take into account additional 20 cycle instructions for each couple (*run*, *length*) stored in Table 3 (step 1).

Let us consider now the encoding of a quantized image with *R runs* and *L levels*. The total number of cycle instructions for this image is expressed in $\boldsymbol{O}(RO_{run} + LO_{level} + 22r + 24l)$, where O_{run} and O_{level} are values read in Table 4, r is the number of *run/level* transitions and l the number of *level/level* transitions. Experiments performed on a bench of natural images such as Lena and Barbara have permitted to estimate parameters *R, L, r* and *l*. Given an image size of 512x512 pixels and 4 levels of decomposition, we found an average processing time of 5.3 ms for the larger subband (256x256 pixels). In a parallel architecture (1 DSP per subband), this time represents the total encoding computation time for the initial 512x512 pixels image.

5. CONCLUSION

An implementation based on a 24 bit DSP well suited for stack-run codec has been presented. The proposed architecture is low cost and allows low computational processing time, as well as interesting flexibility. Its efficiency is based on a well suited data structure to represent *runs* and *levels*. Its versatility leads the architecture to be adapted to the image characteristics. Finally, the proposed structure can be easily parallelized, or/and integrated into a more general design dedicated to efficient image and video coding.

REFERENCES

[1] P. Nus , "Traitement numérique du signal - Applications du processeur spécialisé DSP56002", *Publitronic - Elektor Ed.*, ISBN 2-8661-091-1, 1998.

[2] P. Raffy, M. Antonini, M. Barlaud, "A new optimal subband bit allocation procedure for very low bit rate image coding", *IEE Electronics letters*, vol. 34, issue 7, p. 647, april 1998.

[3] P. Raffy, P. Nus and JM. Moureaux, "A parallel DSP architecture for a new and high performance variable rate coder", The International Conference on Signal Processing Applications and Technology (*ICSPAT*), Toronto, september 1998.

[4] S.V. Ramaswamy and G.D. Miller, "Multiprocessor DSP architectures that implement the FCT based JPEG still picture image compression algorithm with arithmetic coding", *IEEE Transactions on Consumer Electronics*, vol. 39, 1, pp. 1-5, february 1993.

[5] M.J. Tsaï, J. Villasenor and F. Chen, "Stack-run image coding", *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 6, pp. 519-521, october 1996.

bits	23	22	21	20	19	18	17	16	15	…	3	2	1	0
R: 7X0	0	0	0	0	0	1	1	0	0	…	0	1	1	1
L: -1	1	1	0	0	0	0	1	0	0	…	0	0	0	1
R: 4X0	0	0	0	0	0	1	1	0	0	…	0	1	0	0
L: +5	1	0	0	0	0	1	1	0	0	…	0	1	0	1
L: +10	1	0	0	0	1	0	0	0	0	…	1	0	1	0
R: 2X0	0	0	0	0	0	1	0	0	0	…	0	0	1	0
L: -7	1	1	0	0	0	1	1	0	0	…	0	1	1	1

Table 1: Example of the initial stream in the data structure format detailed in Figure 1.

bits	23	22	21	20	19	18	17	16	15	…	3	2	1	0
R: 7X0	0	0	0	0	0	1	1	0	0	…	0	1	1	1
L: -1	1	1	0	0	0	0	1	0	0	…	0	0	0	**0**
R: 4X0	0	0	0	0	0	**1**	**0**	0	0	…	0	**0**	0	0
L: +5	1	0	0	0	0	**1**	**0**	0	0	…	0	**0**	**1**	**0**
L: +10	1	0	0	0	**0**	**1**	**1**	0	0	…	**0**	0	1	**1**
R: 2X0	0	0	0	0	0	**0**	**1**	0	0	…	0	0	**0**	0
L: -7	1	1	0	0	0	1	1	0	0	…	0	**0**	**0**	**0**

Table 2: Stream 1 = initial stream (see Table 1) after the first step (bits having changed are boldface typed).

bits	23	22	21	20	19	18	17	16	15	…	3	2	1	0
R: 7X0	0	**0**	0	0	0	1	1	0	0	…	0	**1**	**1**	**1**
L: -1	1	**1**	0	0	0	0	0	0	0	…	0	0	0	0
R: 4X0	0	**0**	0	0	0	1	0	0	0	…	0	0	**0**	**0**
L: +5	1	**0**	0	0	0	0	1	0	0	…	0	0	0	**1**
R	0	**1**	0	0	0	0	0	0	0	…	0	0	0	0
L: +10	1	**0**	0	0	0	1	0	0	0	…	0	0	**0**	**1**
R: 2X0	0	**0**	0	0	0	0	1	0	0	…	0	0	0	**0**
L: -7	1	**1**	0	0	0	1	0	0	0	…	0	0	**0**	**0**

Table 3: Stream 2 = stream 1 after the second step (bits of the final stream are boldface typed).