21st IEEE Int. Symp. on Field-Programmable Custom Computing Machines (FCCM)
April 28-30, 2013, Seattle, Washington

A High Throughput No-Stall Golomb-Rice Hardware Decoder

Roger Moussalli, Walid Najjar, Xi Luo, Amna Khan
Dept. of Computer Science and Engineering
University of California Riverside
Riverside, USA
Email: {rmous, najjar, luox, akhan015}@cs.ucr.edu

***Abstract*—Integer compression techniques can generally be classified as bit-wise and byte-wise approaches. Though at the cost of a larger processing time, bit-wise techniques typically result in a better compression ratio. The Golomb-Rice (GR) method is a bit-wise lossless technique applied to the compression of images, audio files and lists of inverted indices. However, since GR is a serial algorithm, decompression is regarded as a very slow process; to the best of our knowledge, all existing software and hardware native (non-modified) GR decoding engines operate bit-serially on the encoded stream. In this paper, we present (1) the first no-stall hardware architecture, capable of decompressing streams of integers compressed using the GR method, at a rate of several bytes (multiple integers) per hardware cycle; (2) a novel GR decoder based on the latter architecture is further detailed, operating at a peak rate of one integer per cycle. A thorough design space exploration study on the resulting resource utilization and throughput of the aforementioned approaches is presented. Furthermore, a performance study is provided, comparing software approaches to implementations of the novel hardware decoders. While occupying 10% of a Xilinx V6LX240T FPGA, the no-stall architecture core achieves a sustained throughput of over 7 Gbps.**

***Keywords*-FPGA; Golomb-Rice; compression; decompression; inverted index**

I. Introduction

The goal of data compression techniques is to reduce the storage space and/or increase the effective throughput from the data source (such as a storage medium). Other critical performance factors considered include code complexity and memory offloading requirements. Various compression techniques can be combined and are tailored to perform best within certain classes of applications, where assumptions on the data (format, range, occurrence, etc) hold. Examples of the latter are Run-Length Encoding (RLE) [1] as used by image compression (JPEG), and Lempel-Ziv-Welsch [2] (LZW) for text data.

Compression techniques can be mainly categorized as being lossy or lossless. Generally, lossy techniques result in a higher compression ratio, and/or a faster processing (compression/decompression) time. Lossy techniques are hence preferred when the original data does not have to be exactly retrieved from the compressed data, and differences with the original data are tolerable or non-noticeable (such is the case with audio, video, etc).

Moreover, compression techniques can be further classified as being bit-wise or byte-wise. Byte-wise (or byte-aligned) approaches typically result in a lower compression ratio due to the coarser granularity, but offer a considerably higher compression/decompression throughput.

The work presented in this paper focuses on the acceleration of the decompression of integers compressed using the lossless bit-wise Golomb-Rice [1], [3] (GR) entropy method. GR compression is designed to achieve high compression ratios on input streams with small integer ranges [4]; it is deployed in several applications, such as image compression ([4], [5], [6], [7], [8], [9]), audio compression ([10], [11]), as well as the compression of streams of inverted indices ([12], [13], [14], [15]), and ECG signals ([16], [17], [18]). Inverted indexes require very fast processing, and operate under low timing budgets as they are utilized in the querying of high-volume data, as in (web) search engines [19]; however, even though GR offers high compression ratios, other approaches are preferred due to the gap in decompression performance [14]. Similarly, with the augmented resolution standards on video processing and displays (Full-HD, Quad Full-HD), faster decompression is a must. Finally, the complex processing of the increasing amounts of ECG data can be further reduced using high-performance decoders, with decompression being a first step once data is received. In all of the aforementioned applications, inefficient decompression limits the input throughput to the computational pipelines.

We present a novel highly-parallel hardware core capable of decompressing streams of GR-coded integers at wire speed with constant throughput, operating on *raw unmodified* GR data. To the best of our knowledge, hardware and software (CPU-based) GR decoders assuming unmodified GR data operate bit-serially on the compressed stream, which highly bounds the achievable decompression speeds. On the other hand, modifications to the algorithm and assumptions on the compressed format allow the application of efficient optimizations ([5], [14], [6]), though the limiting assumptions cannot be generalized (see Section III). The proposed no-stall hardware solution is shown to outperform state-of-the-art software and hardware approaches, and achieves up to 7.8 Gbps sustained decompression throughput while occupying 10% of the available resources on a Xilinx Virtex 6 LX240T, a mid- to low-size FPGA.

The contributions of this work can be summarized as follows:

- The first hardware architecture capable of decompressing streams of raw unmodified Golomb-Rice data at a constant rate of several bytes per cycle, with no incurred stalls, and no assumptions on the compressed data.
- A novel decompression engine based on the latter architecture is further detailed, operating at a peak rate of one integer per cycle.
- An extensive design space exploration studying the

resulting resource utilization and throughput of the aforementioned approaches on a Xilinx Virtex 6 FPGA.
- A performance study comparing state-of-the-art CPU-based approaches to implementations of the novel hardware decoders.

The remainder of this paper is organized as follows: Section II provides an overview of Golomb-Rice compression; Section III describes prior art; the novel architectures are detailed in Section IV; experimental results are provided in Section V, and finally, conclusions are listed in Section VI.

II. Golomb-Rice Compression Overview

In this section, the Golomb-Rice compression technique is described, followed by the challenges imposed in parallel decompression.

A. Algorithm Description

Golomb-Rice, or simply GR, or Rice coding is a lossless bit-granularity integer compression approach, which performs best with datasets where the probability of occurrence of small numbers far exceeds that of large values. It is shown that for such input sets, GR coding has compression efficiency close to the more complex arithmetic coding, and comparable to Huffman coding, while no code tables are required, formerly a potential bottleneck in the hardware compression/decompression process [4].

In Golomb coding, given a divisor *d*, each input integer is encoded into two parts: a unary quotient *q*, and a binary remainder *r*. GR coding is a subset of Golomb coding, restricting divisors to powers of two. This implies that for a given *d*, the number of bits required to encode the remainder portion is fixed to $k = log_2(d)$ bits (otherwise variable with Golomb coding). This simple assumption/restriction has a practical negligible negative impact on compression ratio, and greatly simplifies the encoding/decoding process, by allowing the use of simple shift operations instead of the more complex division. Good choices of *d* (hence *k*) greatly affect the compression ratio, and *d* is generally picked as factor of the average of the input integer set [19], [14]; this discussion is however out of the scope of this paper.

Resulting from GR coding is the fact that integers smaller than *d* are encoded using $k + 1$ bits, being a single unary bit and the remainder bits; furthermore, the compression of (infrequent) large numbers can result in more bits than the original uncompressed number, due to the inefficient coding of the unary quotient.

B. Parallelism and Challenges

GR-coded streams offer great opportunity for parallel decompression, since integers can be decoded independently. However, finding the end of a compressed integer and the start of the next is non-trivial, and cannot be determined without knowledge of all prior stream contents.

Figure 1 shows a snapshot of a chunk of bits in a GR-encoded stream, with $k = 3$ (remainder) bits. Integers are coded as the unary quotient (variable number of 1's followed by a 0) followed by the binary remainder bits, from right to left.

The first 0 bit in Figure 1 starting from the right (underlined) could reflect the last unary quotient bit, which would result in the reconstruction of integers *i* and *i+1* as illustrated under the stream (note that unary values end with a 0 bit). On the other hand, this 0 bit could reflect the second remainder bit, leading to integers *i* and *i+1* as depicted above the stream.

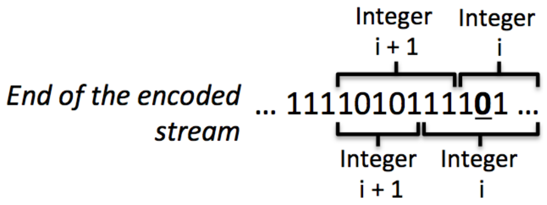


Figure 1. Snapshot of a chunk of bits in a GR-encoded stream. Assuming integers are coded as the unary quotient followed by the $k = 3$ bits binary remainder (from right to left), the above chunk can be decompressed in several ways (two of which are shown); the correct decoding cannot be determined without knowledge of all prior stream contents.

In other words, a decoder processing an N-bit chunk of compressed data per cycle cannot assume independent chunks, since compressed integers are not contained within these chunks (integers span across chunks). Furthermore, in order to process a given N-bit chunk, the decoder has to process all previous compressed data in the stream.

To the best of our knowledge, no (hardware and/or software) approach in the literature allows the processing of stream chunks in parallel. Section IV details our proposed mechanism which overcomes this challenge.

III. Related work

The FPGA-led performance boost up of compression/decompression has long been an active field of research, with the main focus on speeding up low-latency storage access [20], [21], [22]. In this section, we focus on providing an overview of the application and implementation of Golomb-Rice coding in several fields.

Inverted indexes: [12] and [15] provide an overview of inverted index querying, as well as the description and performance of several compression techniques, such as variable length integers, Elias Gamma, Delta coding and Golomb-Rice. The authors in [13], [14] provide a thorough performance and compression ratio study of several compression approaches, as applied to the inverted index problem. A novel compression technique is presented with focus on performance, combining PForDelta with GR coding. The intuition is to partition input integers into blocks, such that the compressed output of each block contains all remainder bits first, followed by the unary quotients. This allows the fast retrieval of the fixed-length remainders through simple lookups. The extraction of the contiguous variable-length unary quotient values is also achieved through smart lookup functions. However, this approach is limited by the scalability of the lookup (limited further performance enhancements), as well as the modifications to the input format required. All the implementations of this work are open source, and will be used in the performance evaluation of our proposed hardware architecture.

Image/video: [4] proposes the use of GR coding to compress the Discrete Cosine Transform (DCT) coefficients

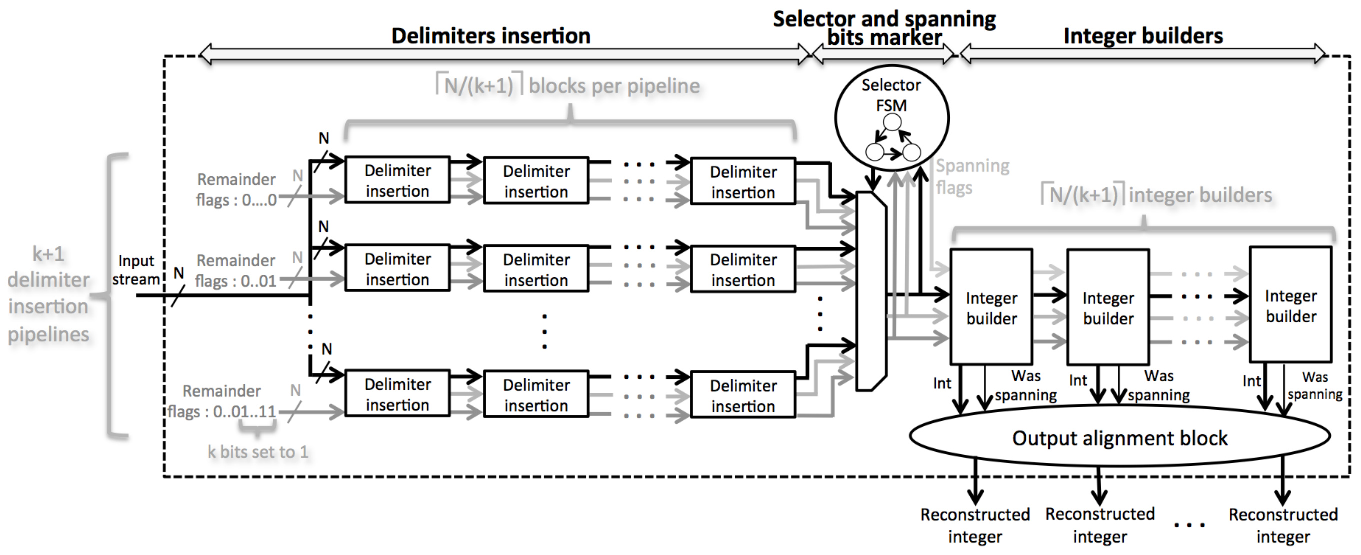


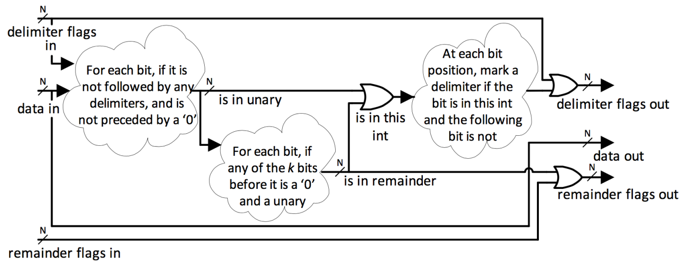
Figure 2. High-level overview of the no-stall GR decompression architecture, capable of sustaining a processing rate of N bits per cycle.

found in JPEG-LS (lossless). The authors in [6], [9] propose novel frame-recompression algorithms targeting MPEG-2 and H.264 videos, respectively. The MPEG-2 FPGA-based decoder makes use several assumptions to implement a parallel GR decoder; for instance, integers are compressed into words of fixed size 21 bits, containing exactly 7 integers each; furthermore (importantly), the boundary of compressed integers is fixed within these 21 bits, and a (small) maximum unary size is assumed. While these assumptions hold in this case, they are not characteristics of GR streams. Similarly, [5] presents a GR-based novel color image FPGA CODEC. A parallel decoder is presented, assuming modified compressed GR format, as well as small independent words containing a fixed number of compressed integers each. The authors in [7] describe the hardware implementation of a novel proposed compression codec targeting advanced-HD video, utilizing GR coding. FELICS, a lossless image compression format utilizing GR coding was introduced in [8].

Audio: [10] describes the use of GR coding in the the lossless audio MPEG-LS format; similarly, the Free Lossless Audio Codec (FLAC) [11] uses GR compression internally.

ECG signals: [16] and [17] detail the compression of DCT coefficients in Electrocardiography (ECG) signals using the lossless GR method. The work in [18] describes the FPGA implementation of a multi-bit per cycle GR compressor of ECG signals. Note that compression is orthogonal to decompression (the problem studied in this paper). The described FPGA decompressor operates in a bit-serial manner.

Miscellaneous: [23] thoroughly studies the adaptive combination of Run-Length to Golomb-Rice coding; the intuition is that unary quotients resulting from GR consist of long streams of 1's, and can be efficiently compressed with run-length encoding. The authors in [24] propose the use of GR coding in conjunction with A/D converters. A detailed CMOS-level implementation is provided, showing the ease



N
delimiter flags in
data in
For each bit, if it is not followed by any delimiters, and is not preceded by a '0'
is in unary
For each bit, if any of the k bits before it is a '0' and a unary
is in remainder
is in this int
At each bit position, mark a delimiter if the bit is in this int and the following bit is not
delimiter flags out
data out
remainder flags out
remainder flags in

Figure 3. Functionality and high-level implementation overview of a delimiter insertion block. Given a chunk of GR-encoded data, delimiter flags, and remainder flags, a delimiter insertion block is tasked with updating the input flags by marking the last bit (delimiter) and remainder bits of the next integer in the chunk, if any.

and advantages of integration of A/D converters with GR encoders.

IV. Hardware Golomb-Rice Decompression

In this section, an overview of how parallel GR decompression can be achieved is provided, followed by an in-depth description of the proposed parallel hardware GR decompression engine.

A. *Parallel Extraction of Compressed Integers*

As described in Section II-B, a decoder cannot process an N-bit chunk of compressed data without any knowledge on prior contents of the compressed stream. Hence, processing an N-bit chunk should be delayed after all previous data is decoded. In reference to Figure 1, this would indicate the property (unary or remainder) of the highlighted 0 bit.

However, given that *k=3* (number of remainder bits), the highlighted 0 bit can only one of the following: (1) a unary bit, the last to be exact; (2) the first remainder bit; (3) the second remainder bit; and (4) finally, the third and last remainder bit. Note that option (2) is trivially dismissed since the previous bit is a 1, which cannot indicate the end

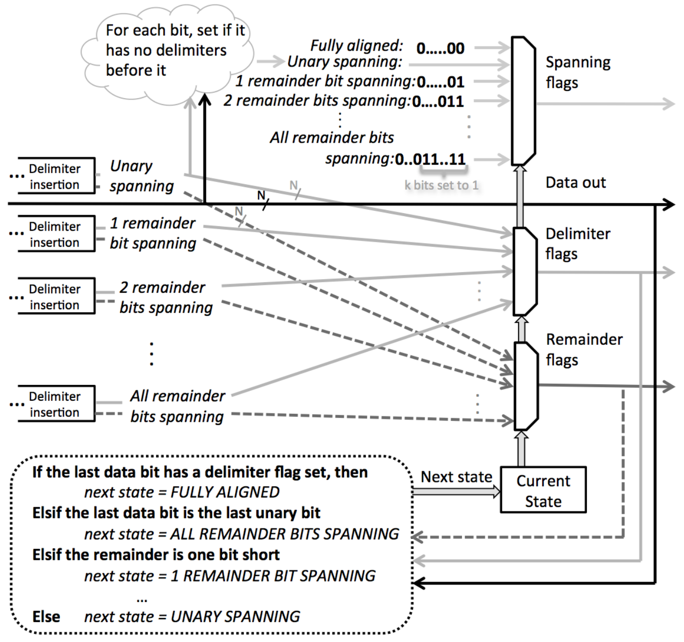

For each bit, set if it has no delimiters before it
Fully aligned: 0.....00
Unary spanning:
1 remainder bit spanning: 0.....01
2 remainder bits spanning: 0....011
All remainder bits spanning: 0..011..11
k bits set to 1
Spanning flags
Data out
Delimiter insertion
Unary spanning
N
1 remainder bit spanning
2 remainder bits spanning
All remainder bits spanning
Delimiter flags
Remainder flags
If the last data bit has a delimiter flag set, then
next state = FULLY ALIGNED
Elsif the last data bit is the last unary bit
next state = ALL REMAINDER BITS SPANNING
Elsif the remainder is one bit short
next state = 1 REMAINDER BIT SPANNING
...
Else next state = UNARY SPANNING
Next state
Current State

Figure 4. Functionality and high-level implementation overview of the *selector and spanning bits marker* stage. This stage is tasked with selecting the output of one of the delimiters insertion pipelines (left), based on knowledge of previously processed chunks (see the FSM transitions). All bits spanning into the current chunk are marked as such using the outputted *Spanning flag* bit vector (top).

of a unary (unaries end with a 0). However, for the sake of a generic example, any given (0 or 1) bit in a stream can have one of only *k+1* properties, as indicated earlier.

Assuming a decoder capable of processing a chunk of N GR-compressed bits in parallel: the i^{th} chunk of size N bits can be speculatively processed in *k+1* ways, where each would be an assumption on the property of the first bit in that chunk. Once the previous $i-1^{th}$ chunk is decoded, k computations are dismissed and one is committed. This will in turn allow the $i+1^{th}$ chunk to commit, and so on, for the remainder of the compressed stream.

Since GR coding mainly targets integers with small ranges, k is generally kept small; it is theoretically maximally less than the integer bit size (64 for double precision). Therefore, the *k+1* design space is limited. The choice of N depends on several physical constraints and performance requirements, as detailed in this section and Section V.

Encoded integers potentially span across (two or more) chunks, either because their encoding starts towards the end of a chunk, and/or because the unary portion is long. This will require some data of consecutive chunks to be combined in order to reconstruct compressed integers.

B. No-Stall Architecture Overview

The remainder of this section details a parallel no-stall hardware GR decompression architecture based on the observations described above.

1) Delimiters Insertion: Figure 2 illustrates a high level overview of the proposed architecture, of which the first portion is delimiters insertion. As an N-bit chunk of compressed GR data is received, it is forwarded to *k+1* pipelines. Each of these pipelines holds an assumption on the input chunk, and receives masks (labeled remainder flags) indicating whether each chunk bit is a remainder or a unary. This mask naturally differs from one pipeline to the next, as illustrated in Figure 2. The top pipeline assumes that the unary of the previous chunk spans into the received chunk (remainder flag is all 0's). The pipeline below it assumes that only one remainder bits spans (remainder flag of all 0's and a single 1); and so on until the bottom pipeline which assumes that all remainder bits span from the previous chunk (remainder flag of all 0's first, then k 1's). Given its respective assumption, each pipeline outputs the received data chunk, alongside a mask (flags) indicating all remainder bits in the received data chunk, as well as a mask indicating the delimiters (last bit) of each encoded integer in the chunk. These masks would allow the fast extraction of individual integers in a chunk, as detailed in later stages.

As depicted in Figure 2, each of the delimiters insertion pipeline consists of $\lceil \frac{N}{k+1} \rceil$ blocks, where the latter indicates the maximum number of encoded integers in a chunk (an encoded integer consists of at least k remainder bits and one unary bit). A ceiling notation is used to reflect the case of a compressed integer spanning into the next chunk, of which less than *k+1* bits are in this chunk.

The functionality and high-level implementation overview of each block is shown in Figure 3. Given a chunk of GR-encoded data, delimiter flags, and remainder flags, a delimiter insertion block is tasked with updating the input flags by marking the last bit (delimiter) and remainder bits of the next integer in the chunk, if any. Having $\lceil \frac{N}{k+1} \rceil$ delimiter insertion blocks per pipeline guarantees that all delimiters and remainder bits in the input chunk will be marked.

Note that for the sake of delimiters insertion, the case of a chunk with no bits spanning into it is treated as a chunk with unary bits spanning into it (top-most delimiters insertion pipeline).

2) Selector (and Spanning Bits Marker) Stage: This is the next stage in the decompression pipeline, following the delimiters insertion. As the name indicates it, it is tasked with selecting the output of one of the delimiters insertion pipelines, based on knowledge of previously processed chunks.

The first chunk received by this stage is fully aligned, meaning that there are no bits spanning into it from the previous chunk. Therefore, the (flag vectors) output of the top-most delimiters insertion pipeline is selected, where the unary was assumed to be spanning. Then, by inspecting the last delimiter flag bit, the last data bit, and the last *k-1* remainder flag bits, the selector FSM can determine the state (hence multiplexer select value) respective to the next received chunk. This process is then repeated for every chunk received.

Figure 4 details the functionality and high-level implementation of the selector stage. The left hand side represents the output of each delimiters insertion pipeline, labeled with the initial assumption of every pipeline (unary spanning, one remainder bit spanning, etc). The output of the pipelines is multiplexed using the selector FSM. The latter can be

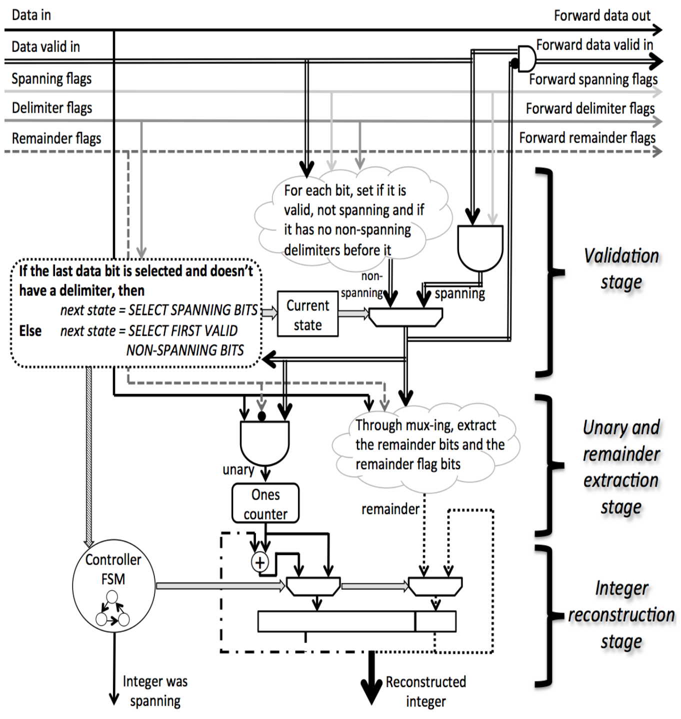

Data in
Data valid in
Spanning flags
Delimiter flags
Remainder flags
Forward data out
Forward data valid in
Forward spanning flags
Forward delimiter flags
Forward remainder flags
For each bit, set if it is valid, not spanning and if it has no non-spanning delimiters before it
If the last data bit is selected and doesn't have a delimiter, then next state = SELECT SPANNING BITS
Else next state = SELECT FIRST VALID NON-SPANNING BITS
Current state
non-spanning
spanning
Validation stage
Through mux-ing, extract the remainder bits and the remainder flag bits
unary
Ones counter
remainder
Unary and remainder extraction stage
Controller FSM
Integer reconstruction stage
Integer was spanning
Reconstructed integer

Figure 5. Functionality and high-level implementation overview of the integer builder block. In the validation stage, one compressed integer from the input chunk is selected for reconstruction, then invalidated when passed to the following integer builder (if any). The unary quotient is converted to binary using a ones counter, whereas the remainder is simply multiplexed using the remainder flags.

in one of $k+2$ states (fully aligned, unary spanning, one remainder bit spanning, two remainder bits spanning, etc), and the conditions for transitions across states are as shown in Figure 4.

The selector stage is further tasked with flagging spanning bits (bits in this chunk belonging to an integer starting in a previous chunk); the use of this flag will be clearer in the next decompression stages. Depending on the state of the current chunk, a set of spanning flags is chosen from, as shown in the top portion of Figure 4. In case the current chunk is fully aligned, then no bits span into it, and the spanning flag is set to all 0's. If remainder bits spans into the chunk, then depending on the number of remainder bits spanning, some of the least significant bits of the flag are set to 1. A constant flag exists for each of the aforementioned cases. On the other hand, when the (variable length) unary is spanning, the number of spanning bits is unknown, and has to be computed on the fly. As shown in Figure 4, a data bit is unary and spanning if there are no delimiters before it (which for every compressed data bit, is equivalent to looking for data bits with value 0 before it).

The output of the selector stage consists of a data chunk, delimiter, remainder and spanning flag vectors respective to each of the data bits. These signals are forwarded to a pipeline of integer builders.

3) Integer Builders: As shown in Figure 2 following the selector stage, a pipeline of $\lceil \frac{N}{k+1} \rceil$ integer builder blocks is deployed. The task of each integer builder is to reconstruct one of the integers in the chunk. Making use of $\lceil \frac{N}{k+1} \rceil$ blocks guarantees the handling of all potential integers in a chunk. Each integer builder selects one integer from the chunk for reconstruction, then invalidates it, and forwards the chunk with updated valid flags to the next integer builder. Invalidating an integer to be decoded ensures that no two integer builder process the same integer. Rules for choosing a compressed integer to decode are described below. Note that depending on the input stream, some integer builders are potentially idle in many cycles; that is because not all input chunks will contain bits of $\lceil \frac{N}{k+1} \rceil$ compressed integers.

Figure 5 details the implementation of an integer builder. The latter comprises of three main stages, namely the validation, unary and remainder extraction, and integer reconstruction stages.

In the validation stage, a compressed integer from the input chunk is selected, and its corresponding valid bit flags are cleared then forwarded alongside the data chunk to the next integer builder (if any). Initially, an integer builder picks the first non-spanning (valid) integer in the chunk. In case that integer is found to span (the last data bit is not delimited), then the integer builder will next select the spanning integer, in order to complete the reconstruction. This approach guarantees that no two integer builders will target the same compressed integer.

Once a compressed integer is selected (if spanning then through multiple cycles), the unary portion is converted back to binary through a one's counter. The remainder bits are selected using the remainder flag bits. The integer reconstruction stage handles the case of integers potentially spanning across multiple chunks.

The output of the integer builder block is a reconstructed integer (concatenated binary quotient and remainder), alongside a single bit flag indicating whether the compressed integer was spanning across two or more input chunks. This flag is used by the (next) output alignment block.

4) Output Alignment Block: The output alignment block adds buffers after every integer builder, such that the output of all integer builders in a given cycle reflects the processing of the same input data chunk. The number of buffers after an integer builder is simply the number of integer builders following it. Furthermore, reconstructed integers are reordered such that the integer spanning in the chunk is placed before others from that chunk. This is achieved using the *was-spanning* flag outputted with every reconstructed integer (Figures 2 and 5). Note that only one integer can be spanning in a given chunk. Also, even though the spanning integer is the first in a chunk, it is not necessarily processed by the first builder, since its processing could have started by a later builder with the previous chunk. Since some integer builders are idle in certain cycles, their output is disregarded by the output alignment block.

C. One-Integer Per Cycle Decoder Overview

Based on the no-stall architecture, a smaller one-integer per cycle architecture is presented, as depicted in Figure 6. The main difference from the no-stall approach is the use of a single (modified) integer builder, as well as the FIFOs and respective controllers highlighted in dark grey. As a

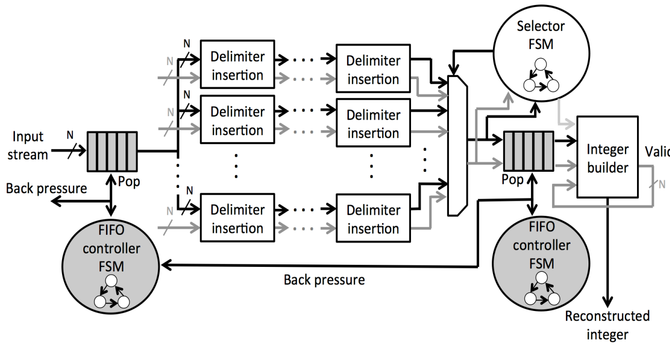


Figure 6. Overview of a decoder with a peak throughput of one integer per cycle. The main difference from the no-stall approach is the use of a single (modified) integer builder, as well as the FIFOs and respective controllers highlighted in dark grey. Back pressure is needed in between FIFOs to avoid dropping compressed data chunks.

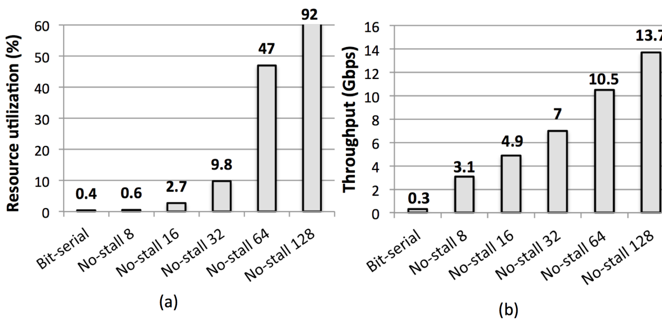


Figure 7. Resource utilization (a) and throughput (b) of the hardware decoders are shown, targeting a Xilinx V6LX240T FPGA, with *k=3*. The naive bit-serial implementation is considered for comparison purposes. The no-stall decoder processing 32 bits per cycle occupies only 10% of the (mid- to low-sized) FPGA, and achieves a 7 Gbps throughput.

data chunk is received, the number of integers it contains is computed, and that many cycles are spent processing it, prior to moving onto the next chunk. Therefore, wire speed throughput is not maintained, and a FIFO (with corresponding controller FSM) is inserted between the (modified) integer builder and the delimiter insertion pipelines. A back pressure signal is propagated from that FIFO to the controller of another FIFO at the input of the decompression block, with the goal of avoiding dropping compressed data chunks.

This architecture is presented and studied as it requires less resources than a no-stall decoder accepting similar-sized input chunks; furthermore, it outperforms the bit-serial implementation (the only implementation in the literature with no assumptions on the compressed stream).

This *one-integer-per-cycle* architecture is mainly useful when the remainder size *k* is comparable to N. For instance, the one-integer-per-cycle decoder assuming chunks of size N uses less resources than a similar no-stall due to the fewer integer builders. The minimum throughput offered by the one-integer-per-cycle decoder is comparable to that of a no-stall accepting chunks of size *k*; and depending on the dataset, the one-integer-per-cycle could provide higher effective throughput than the latter.

D. Decoder Generator

A (C++) tool has been developed to generate the HDL of the decompression pipeline, using certain parameter inputs. These include the input bit-width N (chunk size); the GR parameter *k* (number of remainder bits); whether to make use of a no-stall decoder, a single integer builder, or a bit-serial decoder (for testing purposes); the option to further pipeline certain stages; the option to deploy a multi-integer per cycle arbiter at the output, to match the bit-width of the interface of the block following the decompression pipeline (RAM, PCIe, computational core, etc); as well as other knobs useful to hardware designers. This (7000 lines of C++) tool was implemented from scratch and can be obtained by contacting the authors.

V. Experimental Evaluation

In this section, an experimental evaluation of the proposed hardware GR decoder is carried out. A performance study versus state-of-the-art software decoders is further detailed.

A. Resource Utilization Study

The hardware decoders were tested on a Pico Computing M-501 board [25], connected to a host CPU via PCIe. The M-501 board includes a Xilinx Virtex 6 LX240T FPGA, which is assumed for the remainder of this study.

Figure 7 (a) reports the resulting (post-place and route) resource utilization of the no-stall decoders, on the target V6LX240T FPGA. The Xilinx ISE v14.4 tools are used for synthesis/place and route, with the optimization goal set to speed (normal).

With the number of remainder bits *k=3*, each decoder is tailored for N, being the number of bits processed per cycle. A bit-serial decoder is included for comparison purposes.

The fully parallel no-stall architectures processing 8 and 16 bits per cycle occupy minimal FPGA resources ($< 3\%$). Generally, as N is doubled, the resulting decoder is around 4X larger, with the exception of No_stall-128. In the case of the latter, we suspect that the effort of the tools was higher for area, due to the size of the design (potentially not fitting). Furthermore, this 128-bit pipeline cannot be used on the target FPGA, though it fits; this is because any logic connecting the FPGA to peripheral devices (ethernet, DDR, PCIe, etc) would potentially require more than the remaining resources.

Figure 8 shows the resource utilization of a (32 bit) no-stall hardware decoder as the number of remainder bits k is varied. As k increases, the number of delimiter insertion pipelines (*k+1*) directly increases; conversely, the number of stages in each pipeline ($\lceil \frac{N}{k+1} \rceil$) directly decreases. Hence, the total number of delimiter insertion stages remains constant (equal to N) as k varies. On the other hand, as k increases, the number of integer builders ($\lceil \frac{N}{k+1} \rceil$) decreases, thus leading to a considerable drop in resource utilization (up to 40%). The effect of varying *k* on the operational frequency is marginal; as *k* increases, the critical paths in the delimiter insertion logic and integer builder increase (data omitted for brevity).

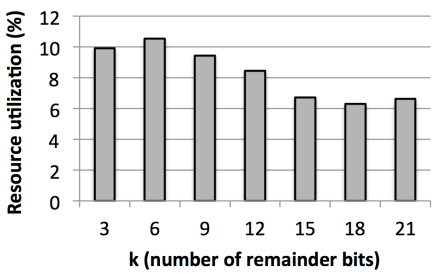


Figure 8. The resource utilization of a (32 bit) no-stall hardware decoder is studied as the number of remainder bits *k* is varied. Place and Route results are shown targeting a Xilinx V6LX240T FPGA. As *k* increases, the number of delimiter insertion pipelines directly increases, but the number of stages in each pipeline directly decreases. Hence, the total number of delimiter insertion stages remains constant as *k* varies. On the other hand, as *k* increases, the number of integer builders decreases, thus leading to a (considerable) drop in resource utilization.

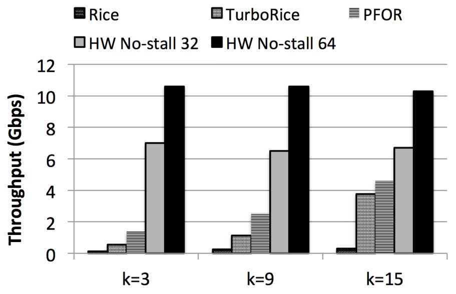


Figure 9. Throughput (Gbps) achieved by software and hardware decoders, as the number of remainder bits *k* is increased. The performance of two hardware decoders is reported here, namely (HW) No-stall 32 and No-stall 64, each processing 32 and 64 bits per hardware cycle, respectively. PFOR is considered as it has shown the best decompression performance in the literature [14]. TurboRice was introduced in [14] as a new approach combining the compression ratio of GR with the performance of PFOR.

Resource utilization of the one-integer-per-cycle decoders (Section IV-C) is comparable to that of a no-stall decoder of the same bit-width, with large *k* assumed. For instance, a 32-bit one-integer-per-cycle decoder occupies around 6.5% of the FPGA logic, comparable to a 32-bit no-stall decoder with *k=21*. On the other hand, since a single integer builder block is used, and because varying *k* has no effect on the number of delimiter insertion stages, *k* has minimal impact on the overall resource utilization of the one-integer-per-cycle decoders. Data has been omitted due to space limitations.

B. Performance Evaluation

In this section, throughput is measured at the input of the studied decoders. In other words, it is measured as a function of the time required to process a compressed document, regardless of the rate at which uncompressed integers are generated at the output. The latter has been used (in addition to the former) as s metric in some studies such as in [14].

The performance of the bit-serial and no-stall architecture hardware decompression cores is studied, as shown in Figure 7 (b), where *k=3*. Throughput is measured as a function of the operational frequency, and the number of bits processed per cycle; throughput does not increase linearly with the number of bits processed per cycle, due to the negative impact on the operational frequency of the decompression circuit.

The critical path of the no-stall decoders resides in the unary and remainder extraction stage of the integer builder. Specifically, the extraction of the remainder bits limits performance. Nonetheless, this block can be trivially pipelined further. The next long wire is found in the delimiter insertion stage; the latter can also be trivially pipelined, as it contains no control logic. Since the developed decoders achieve good performance, further pipelining is not applied here, due to the added penalty on resources.

Note that the performance of the one-integer-per-cycle decoders depends on the data set; here, the sustained throughput is bound by *k+1* bits per cycle (the minimum compressed integer size) and *N* (the number of bits read per cycle).

We next compare the performance of the proposed decoders to state-of-the-art high performance CPU-based software decoders. We make use of the open source (C++) software decoders described in [14]. Three software decoders are considered, namely (1) Rice, a highly efficient implementation of the base GR coding; (2) TurboRice, a newly proposed approach in [14], combining the compression ratio benefits of GR coding, with the performance of the PFOR method; (3) PFOR, a compression mechanism that targets blocks of integers at a time (hence neither bit- nor byte-granularity). Other CPU-based approaches were studied (variable byte, S9, S16), and their performance was within the range or Rice, TurboRice, and PFOR; hence, only the latter are reported here.

All CPU-based approaches were ran on a CentOS 5 server with an Intel Xeon processor running at 2.53 GHz, with 8MB of L3 cache, and 36 GB of RAM. Synthetic datasets containing 500 million integers each were generated, while varying the range of the integers (hence *k*). A large set of integers is assumed in order to ensure that steady-state performance is measured. Throughput is measured as a function of the wall-clock time, such that the compressed and resulting uncompressed data reside in the CPU RAM. Moreover, throughput is measured as a function of the size of the compressed data, respective to each software approach.

Figure 9 shows the throughput (Gbps) achieved by software and hardware decoders, as the number of remainder bits *k* is increased. The performance of two hardware decoders is reported here, namely (HW) No-stall 32 and No-stall 64. As discussed earlier, varying *k* has marginal impact on the performance of the hardware decoders. On the other hand, CPU-based approaches perform better as *k* increases. As *k* is increased from 3 to 15, the throughput of the CPU approaches increases by an average of 4X. This is because the implementation of the CPU-based approaches processes encoded data 4 bytes at a time (one integer). As the remainder size increases, the number of (individual) unary bits to be processed per 4 bytes decreases. Nonetheless, data to be compressed by GR is generally small, and large remainder values are not assumed.

The proposed no-stall 32 and 64 architectures provide a higher throughput than PFOR (average of 3X and 4.7X speedup respectively), as well as TurboRice (average of 6.8X and 10.4X speedup respectively). Furthermore, the only software approach that operates on unmodified GR data is Rice, where the no-stall 32 and 64 architectures are respectively up to 52X and 79X faster, as well as respectively 34X and 51.5X faster on average.

VI. Conclusions

A novel highly-parallel hardware core capable of decompressing streams of Golomb-Rice-coded integers at wire speed (no-stall) with constant throughput is presented, operating on *raw unmodified* GR data. To the best of our knowledge, hardware and software (CPU-based) GR decoders assuming unmodified GR data operate bit-serially on the compressed stream, which highly bounds the achievable decompression speeds. Hence, even though GR offers high compression ratios, other approaches are preferred due to the gap in decompression performance. The presented decoder, capable of processing several bytes per cycle, is shown to outperform an efficient GR CPU-based implementation by up to 52X, while utilizing 10% of resources available on a Xilinx V6LX240T FPGA. Furthermore, when operating on 64 bits per cycle, the presented decoder provides average speedups of 4.7X and 10.4X when respectively compared to a software implementation of the high-performance PFOR and TurboRice de/compression methods.

References

[1] S. W. Golomb, "Run-Length Encodings." *Information Theory, IEEE Trans. on*, vol. 12, no. 3, p. 399, 1966.

[2] T. Welch, "A Technique for High-Performance Data Compression," *Computers, IEEE Trans. on*, vol. 17, no. 6, pp. 8–19, 1984.

[3] M. Weinberger, G. Seroussi, and G. Sapiro, "LOCO-I: A Low Complexity, Context-Based, Lossless Image Compression Algorithm," in *Data Compression Conf., 1996. DCC'96. Proc.* IEEE, 1996, pp. 140–149.

[4] N. Memon, "Adaptive Coding of DCT Coefficients by Golomb-Rice Codes," in *Image Processing, 1998. ICIP 98. Proc. 1998 Int. Conf. on*, vol. 1. IEEE, 1998, pp. 516–520.

[5] H. Kim, J. Lee, H. Kim, S. Kang, and W. Park, "A Lossless Color Image Compression Architecture Using a Parallel Golomb-Rice Hardware CODEC," *Circuits and Systems for Video Technology, IEEE Trans. on*, vol. 21, no. 11, pp. 1581–1587, 2011.

[6] T. Lee, "A New Frame-Recompression Algorithm And Its Hardware Design For MPEG-2 Video Decoders," *Circuits and Systems for Video Technology, IEEE Trans. on*, vol. 13, no. 6, pp. 529–534, 2003.

[7] T. Tsai and Y. Lee, "A 6.4 Gbit/s Embedded Compression Codec for Memory-Efficient Applications on Advanced-HD Specification," *Circuits and Systems for Video Technology, IEEE Trans. on*, vol. 20, no. 10, pp. 1277–1291, 2010.

[8] P. Howard and J. Vitter, "Fast and Efficient Lossless Image Compression," in *Data Compression Conf., 1993. DCC'93.* IEEE, 1993, pp. 351–360.

[9] Y. Lee, C. Rhee, and H. Lee, "A New Frame Recompression Algorithm Integrated with H.264 Video Compression," in *Circuits and Systems, 2007. ISCAS 2007. IEEE Int. Symp. on.* IEEE, 2007, pp. 1621–1624.

[10] T. Liebchen and Y. Reznik, "MPEG-4 ALS: An Emerging Standard for Lossless Audio Coding," in *Data Compression Conf., 2004. Proc. DCC 2004.* IEEE, 2004, pp. 439–448.

[11] "Free Lossless Audio Codec," http://www.xiph.org/flac.

[12] A. Mahapatra and S. Biswas, "Inverted indexes: Types and techniques," *Int. Journal of Computer Science*, vol. 8.

[13] H. Yan, S. Ding, and T. Suel, "Inverted Index Compression and Query Processing with Optimized Document Ordering," in *Proc. of the 18th Int. Conf. on World Wide Web.* ACM, 2009, pp. 401–410.

[14] J. Zhang, X. Long, and T. Suel, "Performance of Compressed Inverted List Caching in Search Engines," in *Proc. of the 17th Int. Conf. on World Wide Web.* ACM, 2008, pp. 387–396.

[15] F. Scholer, H. Williams, J. Yiannis, and J. Zobel, "Compression of Inverted Indexes for Fast Query Evaluation," in *Proceedings of the 25th Annual Int. ACM SIGIR Conf. on Research and Development in Information Retrieval.* ACM, 2002, pp. 222–229.

[16] L. Batista, L. Carvalho, and E. Melcher, "Compression of ECG Signals based on Optimum Quantization Of Discrete Cosine Transform Coefficients and Golomb-Rice Coding," in *Engineering in Medicine and Biology Society, 2003. Proc. of the 25th Annual Int. Conf. of the IEEE*, vol. 3. IEEE, 2003, pp. 2647–2650.

[17] J. Chen, J. Ma, Y. Zhang, and X. Shi, "ECG Compression based on Wavelet Transform and Golomb Coding," *Electronics Letters*, vol. 42, no. 6, pp. 322–324, 2006.

[18] M. Meira, J. de Lima, and L. Batista, "An FPGA Implementation of a Lossless Electrocardiogram Compressor based on Prediction and Golomb-Rice Coding," in *Proc. V Workshop de Informática Médica*, 2005.

[19] J. Zobel and A. Moffat, "Inverted Files for Text Search Engines," *ACM Computing Surveys (CSUR)*, vol. 38, no. 2, p. 6, 2006.

[20] D. Craft, "A Fast Hardware Data Compression Algorithm and Some Algorithmic Extensions," *IBM Journal of Research and Development*, vol. 42, no. 6, pp. 733–746, 1998.

[21] B. Sukhwani, B. Abali, B. Brezzo, and S. Asaad, "High-Throughput, Lossless Data Compresion on FPGAs," in *Field-Programmable Custom Computing Machines (FCCM), 2011 IEEE 19th Annual Int. Symp. on.* IEEE, 2011, pp. 113–116.

[22] J. Núñez, C. Feregrino, S. Jones, and S. Bateman, "X-MatchPRO: A ProASIC-based 200 Mbytes/s full-duplex lossless data compressor," in *Field-Programmable Logic and Applications.* Springer, 2001, pp. 613–617.

[23] H. Malvar, "Adaptive Run-Length/Golomb-Rice Encoding of Quantized Generalized Gaussian Sources with Unknown Statistics," in *Data Compression Conf., 2006. DCC 2006. Proceedings.* IEEE, 2006, pp. 23–32.

[24] W. Leon-Salas, S. Balkir, K. Sayood, and M. Hoffman, "An Analog-to-Digital Converter with Golomb-Rice Output Codes," *Circuits and Systems II: Express Briefs, IEEE Trans. on*, vol. 53, no. 4, pp. 278–282, 2006.

[25] "Pico Computing M-Series Modules," http://www.picocomputing.com/m_series.html.