The Journal of High School Research

Hardware Acceleration of rANS Decoder

Sina Lindseth*

Submitted: 19 January 2025 **Accepted:** 27 March 2025 **Publication date:** 24 April 2025

DOI: 10.70671/y0gyk181

Abstract: The rANS (range Asymmetric Numeral System) algorithm is a popular lossless data compression technique based on probability distributions of a symbol alphabet. This class of encoders, called entropy encoders, assigns the shortest codes to the highest-probability symbols. The variety of entropy coding methods is reflected in methods such as how to compute the probabilities, construct the codes, whether to include other metadata information, and what restrictions to impose on the arithmetic or input elements. ANS is a family of entropy coders that improve upon Huffman and Arithmetic coding and are optimized for different use cases. The rANS algorithm is one of these coders and has the advantage of limiting the size of its state variable regardless of the size of the input data. In other variants, certain variables involved in the coding grow large for large input data, but they do not include extra steps to rescale those values, so there is a tradeoff. This research presents a hardware implementation of static size that realizes rANS decoding in a streaming use case. The rANS algorithm is suitable for this research because all parameters and variables involved are of known and limited size for all possible input data. The simple decoding method of the rANS algorithm allows for efficient hardware acceleration, specifically because all inputs to the algorithm are known on every clock cycle. Building a dedicated logic circuit to implement the rANS decoding equations directly allows one to update the rANS state on every clock cycle, allowing performance of one symbol decode per clock cycle. In a software solution, the rANS equations are calculated over many steps. Memory move operations to store intermediate values take even more cycles. This research introduces a simple digital logic design using a finite state machine that will decode one symbol per clock with no other latencies and no circuit elements duplicated for parallel operations. It implements one decoder without pipelining or other methods to hide parallel operations. It assumes no specific hardware or CPU architecture and measures performance in clock cycles per decoding operation.

Author keywords: Data compression; clock cycle; combination logic; decoder; state machine; numeral system

Introduction

This research implements hardware acceleration of the rANS (range Asymmetric Numeral System) decoding process. It does not consider the encoding process. The work of Giesen[1] provides several software examples of the ANS algorithm based on the original work of Duda.[2] These examples from Giesen[1] provide a code base that allows researchers to explore novel implementations, compare performance metrics, and attempt new optimizations. The author makes use of input data from a Giesen[1] software encoder that implements rANS encoding correctly, allowing the author to test the research decoder implementation against properly encoded data. The author also uses code from a Giesen[1] rANS decoder as a guide for implementing corresponding decoding operations in hardware. The final output result of this research is a simulation test bench of the logic circuit taking rANS encoded text as input. The simulator will display the system clock, and each rising edge will mark a decoded symbol appearing at the output in ASCII (see Fig. 1).

The decoded byte is output at the rising edge of each clock cycle. The radix is switched to ASCII for readability of the "DecodedByte" variable. All other variables are in hexadecimal. EncBytes is the input data in compressed form, and "init" marks the start of the decoding operations. Note how the EncBytes can stay the same for multiple DecodedByte symbols. In Fig. 1, a860bb decodes the "E" and "S." This makes sense because it is performing data decompression, so there should be fewer input bytes than output bytes, proportional to the compression ratio, since it is expanding compressed data.

This research uses a Finite State Machine as the framework for the hardware-accelerated decoder, as described in Minns et al.[3]. A single register holds the current state S, which is fed to combinational logic along with the inputs. The logic output S_{n+1} is fed to the input of the register, and the circuit updates the S state on every clock cycle. The current S state is also used to calculate S_{n+1} (see Fig. 2). The rANS algorithm fits perfectly into this finite state machine framework. It simply applies more input variables to S, and nothing prevents expanding the inputs as long as the current S state and input variables are valid on every cycle. There are no dependencies within the rANS algorithm that take multiple clock cycles to resolve. Variables are handled

*Corresponding Author: Sina Lindseth. Email: sinaswater@gmail.com
Kings High School, Seattle, WA 98133, USA

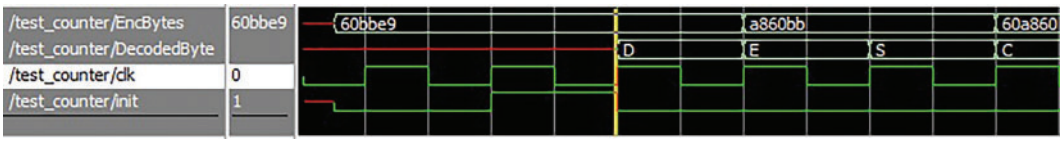

Figure 1. Simulation results showing one-symbol-per-clock decoding of an input stream starting at yellow mark

such that only one table lookup is required per step on the same table.[2] If the algorithm required multiple table lookups, multiple copies of the table would be required, each with their own lookup index.

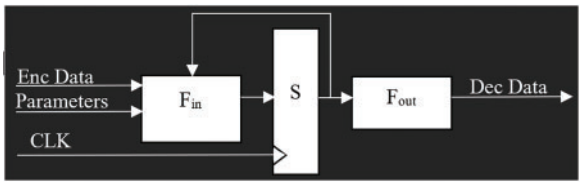

Figure 2. Decoder circuit block diagram

Methodology

The rANS decoder maintains a state variable "S" that updates on every clock cycle. The decoded byte is a function of S. The logic to calculate S_{n+1} is a function of S_n and other static parameters brought over as metadata from the encoding process. Information from the encoding process consists of the encoded data and parameters, which are tables used by the algorithm, such as probability distribution information. These tables are described in the original paper by Duda et al.[2].

This research creates a register large enough to hold S. Because rANS has a normalization feature, the size of S in bits is known and limited. As S accumulates and is about to grow beyond a maximum number of bits M, renormalization discards the excess bits. For this research, 32 bits are used, which is a number greater than M (S > M).

In Fig. 2, register S holds the current rANS state and symbol ID lookup index from the rANS algorithm. F_{in} and F_{out} are combinational logic equations. F_{in} calculates the next rANS state and symbol index, S_{n+1}. F_{out} calculates the decoded output, which is $F_{out}(S)$, the fully decoded symbol. There is zero delay in computing F_{in} and F_{out}. No clock cycles are added for these functions, and there is no pausing of the clock.

Information from Encoding Process Used during Decoding

The "Parameters" in Fig. 2 come from static lookup tables and other data brought over from the encoder. This information is unique to each encoded message. The code outputs an index that selects a value from a given table on each clock cycle. Since all input data is valid on each clock cycle and there are no dependencies on any processing, F_{in}(Enc Data, Parameters, S) is valid on every clock cycle, and F_{out} is valid on every clock cycle. Duda et al.[2] describe this information in detail, but for this research, only the validity of all inputs on every clock cycle is required.

Method to Create Logic Equations for Input and Output Functions

From Duda et al.[2] and numerous software examples, such as ryg_rans, Giesen,[1] this research constructs a bit-accurate decoder using the C programming language, as shown in Fig. 3. This program provides a set of arithmetic operations in the correct order for accurate decoding. The reference code is modified to flatten all subroutines and parallel operations. Finally, the user interface, optional features, and methods that are not part of the algorithm are removed. The referenced example has the code reduced to about 10 lines, so it has little resemblance to the original reference source material. Optimizations that target specific CPU architectures hide critical information and include items that will not port to the hardware framework in Fig. 2. The function F_{in} is formed such that all component variables and coefficients are hardware accessible and valid on every clock cycle.

```
static inline uint32_t RansDecGetAlias(RansState* r, SymbolStats* const syms, uint32_t scale_bits)
{
    RansState x = *r;

    // figure out symbol via alias table
    uint32_t mask = (1u << scale_bits) - 1; // constant for fixed scale_bits!
    uint32_t xm = x & mask;
    uint32_t bucket_id = xm >> (scale_bits - SymbolStats::LOG2NSYMS);
    uint32_t bucket2 = bucket_id * 2;
    if (xm < syms->divider[bucket_id])
        bucket2++;

    // s, x = D(x)
    *r = syms->slot_freqs[bucket2] * (x >> scale_bits) + xm - syms->slot_adjust[bucket2];
    return syms->sym_id[bucket2];
}
```

Figure 3. C code to decode one symbol given encoded input and metadata tables

This research implements function F_{in} using logic equations as in Fig. 4 with Verilog assign statements. Any number of assign statements can comprise the function. Note how the assign statements closely follow the C code in Fig. 3. There are additional index variables that are used in lookup tables. The tables are not shown in the figure, but can be viewed in the code. These are the underlying data structures in the C code, passed in as function arguments. The assign statements in Fig. 4 are combinational logic equations and are evaluated continuously, such that a change in any input propagates instantly through all the equations and updates every variable.

In Verilog, each variable must be declared with its size in bits. The two requirements are that a given variable is wide

```
assign x[31:0] = RansState[31:0];
assign mask[31:0] = (32'h1 << SCALE_BITS) - 1;
assign xm = x[31:0] & mask[31:0];
assign bucket_id[31:0] = xm[31:0] >> (SCALE_BITS - LOG2NSYMS);
assign bucket2[31:0] = bucket_id[31:0] * 2;
assign bucket2plusOne[31:0] = bucket2[31:0] + 1;
assign bucket2Final = (xm < dividerLookup[31:0]) ? bucket2plusOne[31:0] : bucket2[31:0];
assign x_mult_input[31:0] = (x >> SCALE_BITS);
assign NewRansState = multResult + xm - slot_adjustLookup;
assign slot_freqsLookupIndex[8:0] = bucket2Final[8:0];
assign slot_adjustLookupIndex[8:0] = bucket2Final[8:0];
assign sym_idLookupIndex[8:0] = bucket2Final[8:0];
assign dividerLookupIndex[7:0] = bucket_id[7:0];
assign sym_id[31:0] = sym_idLookup[31:0];
assign dec_byte[7:0] = sym_id[7:0];
```

Figure 4. Verilog code to decode one symbol given encoded input and metadata tables

enough to hold the output of its connected logic, and that all connected ports are the same size. For example, k-bit outputs are truncated if connected to inputs of j < k bits. The Verilog concatenation operator adds padding when connecting small variables to large ports.

This research predicts that possible FPGAs or ASICs will include a hardware multiplier. For one part of the equation, the model for a popular hardware multiplier is imported to simplify fitting the design into custom chipsets or an FPGA. This multiplier is not needed for simulation, but is closer to a real-world implementation.

The register S shown in Fig. 2 is an n-bit D flip-flop type register as described in Kalyan et al.[4]. On every rising edge of the clock, it updates its Q output with the current F_{in}, a subset of which is the current decoded byte index. From Fig. 4, the variable NewRansState is connected to the input of register S. It is the consolidation of all the other assign statements and table lookups and is the correct equation for F_{in}.

Concurrency in Verilog

This research takes advantage of the characteristic concurrency properties of the "assign" operator, as described in Chen et al.[5]. All the statements in Fig. 3 are calculated continuously, so any change in input produces an immediate new result at the output. This concurrency allows the entire decode of the next symbol to occur between the clock rising edges, at the instant a new encoded byte is fed to the decoder. By comparison, the statements in Fig. 2 are executed in order as instructions by a CPU, each taking one or more clock cycles.

Method to Test the Decoder

Parameter tables and encoded data are included as arrays of constants in a Verilog test bench. These arrays are accessed from the device under test, which is the decoding module. The decoding module outputs indexes used to look up values in the arrays. Note the variables in Fig. 4 that end in "LookupIndex." Any time you encode new data using a given rANS encoder, you must replace these tables and the encoded message in the test bench. For large messages, the tables may grow in size. To form arrays in Verilog, the research uses the Verilog concatenation operator and reads data using the Verilog part-select operator. When using these operators, a C program can generate these tables easily in the Verilog syntax. An example is included with the GitHub project for this research.

The test bench generates clock cycles and includes an "init" signal to mark the start of the decoding. The simulator is run at least long enough to generate a number of clock cycles equal to the length of the plain text message after init is asserted.

Method for Table Lookup

In a real chipset, tables would be stored in RAM and there would be logic to load and unload this memory. This research focuses on hardware acceleration and minimizes unrelated features to the extent possible. This implementation concatenates a given array into one large hexadecimal constant. The size of this constant is then N bytes × 8 bits/byte. The Verilog part-select operator is then used, as described in Sutherland et al.,[6] to retrieve m × 8 bits given an index. See Fig. 5 for an example.

```
Array[4 bytes]={0x00,0x01,0x02,0x03};

Concatenated Constant: table = 0x00010203

Let's extract array[1]=0x01, using part select on the variable "table" as element.

WORD_WIDTH = 8;  // width in bits of table element
Index = 1;  // Index is the variable presented to the table lookup, 0 to NSYM-1.
NSYM = 4; // number of elements in the table.

From the code:
assign element[7:0] = table[((NSYM - Index - 1)*WORD_WIDTH)+:WORD_WIDTH];

This equals table[16+:8];

This means extract 8 bits starting at bit 16 and moving to the left.
0x00010203

Therefore, element = 0x01;
```

Figure 5. Example of extracting element from array using Verilog part select

This method makes the table lookup result continuously available to the rest of the logic as long as the index is valid. When the index changes, the new result is instantly available.

Discussions and Guidelines for Applications

Applications using this decoder can leverage the fact that the number of cycles spent decoding a given text is deterministic. This determinism, combined with the one-symbol-per-cycle performance, implies that the circuit could be used in line with a network stream without the need for complex buffering, flow control mechanisms, and checks for completion of decoding.

Results

The design spends N clock cycles decoding, where N equals the length in symbols of the unencoded, or "plain text," message. The input data has a smaller size since it underwent

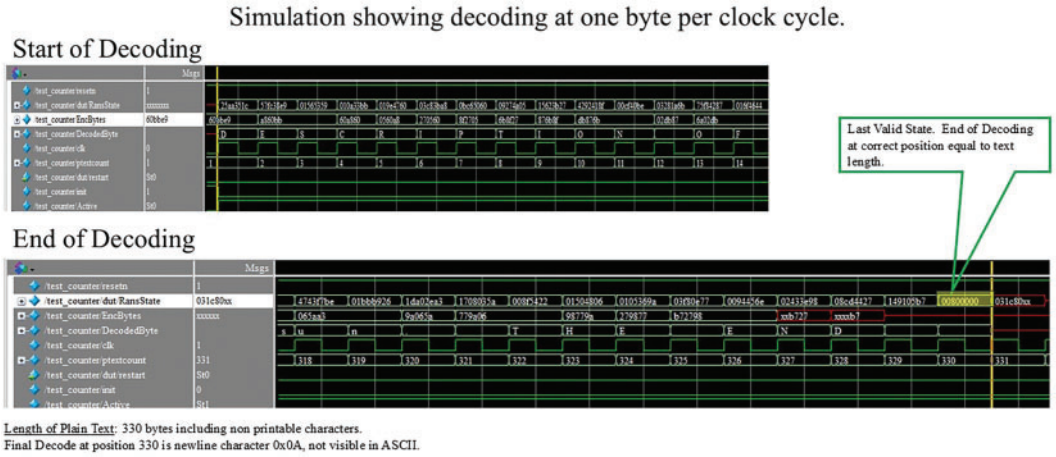

Figure 6. Simulation showing the start and end of decoding

compression in the encoding process. This level of performance was the goal of this research. See the waveforms in Fig. 6.

Discussions and Guidelines for Applications

The logic framework in this research generalizes to any input function F_{in}. A free-running counter would be a simple example. This research makes F_{in} include the entire rANS decoding algorithm. F_{in} can be infinitely complex as long as all inputs to F_{in} are valid on every clock cycle including any feedback loops within the logic.

In a real-world implementation, trade-offs would be made for parameter table storage, storage of encoded message, and multiplier delay. Memories storing tables and encoded data must be readable in one clock cycle. The delay of the multiplier and internal RAM memories will limit clock frequencies. Depending on cost, optimal trade-offs in delays and memory capacities are possible. Limiting the size of the encoded message is a good starting point to reduce required memory capacity. Using the multiplier mitigates the size of table lookups by handling a fixed operation required on every decode. However, the multiplier itself adds some delay, so a designer would evaluate whether to absorb the multiplication into a larger table lookup or keep it separate.

This research only shows the algorithm construction and does not implement the solution in real hardware. It does not estimate the clock frequencies that could be achieved in any specific ASIC or FPGA technology. As technology evolves, such timings will improve, and the tightly coupled RAM needed to present parameters and encoded data to the finite state machine will increase in capacity. Other research measures data rates and performance metrics based on logic delays in one particular chipset, but these are only valid at the time the paper is written.[1] After some years, those timings no longer make sense. This research describes a simple logic framework that can fit into an arbitrary chipset, and the performance metric is simply symbols per clock cycle, not dependent on propagation delays through the technology.

Recommendations for Future Work

The research aligns with optimizations in other research that allows for parallelism by passing intermediate states as metadata, allowing additional decoders to skip to different start points within the message, as described in Lin et al.[7]. There is nothing to prevent using multiple instances of this decoder in parallel in those implementations. Such solutions would see additional improvements beyond their fast-forwarding mechanisms by using the hardware-accelerated decoder to decode at one symbol per clock cycle from the start point obtained from their calculations. Additional research could focus on novel logic designs that provide a memory pool so that encoded data could be streamed into this memory and then distributed to multiple hardware-accelerated decoders without any errors or loss of efficiency.

Conclusions

The rANS decode operation shown in this article fulfills the stated goal of one symbol per clock cycle for efficient hardware acceleration. The heart of the algorithm is to maintain a state variable in a register, much like a counter. Since all parameters and input data are known on every clock cycle, a correctly decoded symbol is output on every clock cycle. This method achieves the advantages of both efficiency and determinism in a straightforward manner.

Acknowledegments

The author would like to express gratitude to Fabian "ryg" Giesen for developing a set of software rANS encoders and decoders, which the author used to generate encoded test data and which also served as an inspiration for this hardware-accelerated version.

Supplementary Information

The code for this project has been posted to GitHub. It includes a working example with automated scripts for simulation with ModelSim and encoded data for testing. The

project is located at https://github.com/Sinaresearch/rans_decoder/.

Journal's Disclaimer

The views and opinions expressed in this article are those of the author(s) and do not necessarily reflect the official policy or position of the *Journal of High School Research* (JHSR) or any affiliated entities. While every effort has been made to ensure the accuracy and reliability of the information presented in this article, the *Journal of High School Research* and its publishers are not responsible for any errors, omissions, or inaccuracies contained within the article. The journal does not assume any liability for the content or for the consequences of any actions taken based on the information provided herein. The author(s) of this article affirm that all necessary ethical approvals and permissions were obtained prior to conducting the research and submitting the manuscript. The responsibility for the interpretation and use of the information presented rests solely with the author(s).

References

[1] Giesen F. "ryg." Rygorous/Ryg_rans; 2014. December 16, 2024. https://github.com/rygorous/ryg_rans.

[2] Duda J. Asymmetric numeral systems: entropy coding combining speed of Huffman coding with compression rate of arithmetic coding. arXiv preprint arXiv:1311.2540. 2013. doi:10.48550/ARXIV.1311.2540.

[3] Minns P, Elliott I. *FSM-Based Digital Design Using Verilog HDL*. 1st ed. A-2 Windsor Estate, Chuna Bhatti, Bhopal 462016 India: Wiley, Genesis Global Publication; 2008:67–103. doi:10.1002/9780470987629.

[4] Kalyan YV, Vineeth Kumar M, Sreenath P. Design of D flip-flops for high performance VLSI applications using CMOS technology. *Int J Res Publ Rev*. June 2022;3(6):3029–3038.

[5] Chen Q, Zhang N, Wang J, et al. The essence of verilog: a tractable and tested operational semantics for verilog. *Proc ACM Program Lang*. October 2023;7:OOPSLA2,30. doi:10.1145/3622805.

[6] Sutherland S. *Verilog HDL Quick Reference Guide*. Sutherland HDL; 2001. https://www.sutherland-hdl.com.

[7] Lin F, Arunruangsirilert K, Sun H, Katto J. Parallel rANS decoding with decoder-adaptive scalability. *Proceedings of the 52nd International Conference on Parallel Processing*; August 7–10, 2023; Salt Lake City UT USA: Association for Computing Machinery. pp. 31–40. doi:10.1145/3605573.3605588.

About the Authors

Sina Lindseth is a Junior at King's High-school in Seattle, Washington. She enjoys figure skating and leads the Community Outreach Club at her school. Her research interests include technology and healthcare topics.