

When Noise Becomes the Signal: GRAND on FPGA

Laura Landon
llandon9@mit.edu

Abstract—This project implements a hardware demonstration of Guessing Random Additive Noise Decoding (GRAND) using an FPGA. The system exposes the process of error correction to the user by accepting text input over UART, applying a selectable forward error correction code, simulating a noisy channel, and decoding the corrupted output for comparison. Two coding families are supported, Hamming and repetition codes, with code selection controlled via hardware switches. GRAND decoding is integrated as both a universal hard-decision decoder and soft-reliability GRAND (SRGRAND), demonstrating its potential as a flexible alternative to traditional code-specific decoders.

I. INTRODUCTION

In modern digital communication systems, reliable data transmission hinges on error correction. While most practical decoders are code-specific, these traditional approaches require a dedicated decoder for each encoder. The recently proposed Guessing Random Additive Noise Decoding (GRAND) algorithm [1] departs from this paradigm by focusing on the noise rather than the codebook, allowing for universal decoding of any block code, given a way to verify a sequence's presence in the codebook.

The core function of GRAND is generating codeword guesses and checking their validity. Once implemented, different codes can be swapped out using the same GRAND implementation. This one-time implementation requirement is useful for hardware because it allows an existing decoding chip to be used with a variety of codes. Additionally, the process of checking codeword guesses is highly parallelizable. Motivated by ongoing research in my lab into GRAND and its variants, I wanted to explore the challenges and benefits of GRAND in hardware as well as design a tool which could aid me in future research.

The technical challenge of this project lies in the tradeoffs of flexibility and performance. I wanted my implementation of GRAND to be flexible (supporting multiple code types) as well as performant (able to decode in real-time with variable-length noise pattern searches). GRAND decoding requires generating many candidate noise patterns, ordered by likelihood, and checking the validity of each result against the codebook. The process of decoding is variable and unpredictable, requiring careful handshaking and buffering to decouple timing at each step of the encoding-channel-decoding pipeline.

II. SYSTEM OVERVIEW

A block diagram of the full system is provided in two parts, 1 per page in the interest of size and readability, at the end of this report (see Fig. 3 and 4). All modules are green,

indicating their presence in the final synthesized design, except for the Reed-Muller encoder which was synthesized in order to analyze the resource usage of a lookup-based encoder but not included in the final synthesized design. As seen in those figures, the system consists of about four primary categories and one key subcategory.

a) *User I/O*: This takes the form of a UART RX/TX connection, taking in user-entered text and returning the original, corrupted, and corrected version of each byte to be displayed for the user. Bytes received from the UART RX are buffered in `bytes_fifo` for encoding. The bytes passed to the UART TX are pulled from the buffers `orig_fifo`, `corrupt_fifo`, and `decoded_fifo`. A state machine in my `top_level.sv` manages which buffer to pull bytes from and sends them according to a simple order-based protocol. Not included in the block diagram are the python and bash scripts which use this order-based protocol to take the sequential UART bytes and display them for the user as three different lines of text and report the error correction statistics.

b) *Encoding*: The final synthesized design allows the user to test decoders on two types of codes: Hamming (12,8) and Triple Modular Redundancy (24,8). Encoders pull bytes from the bytes FIFO and encode them, converting 8-bit inputs into larger outputs. The user is able to control the encoder selection via the lowest two FPGA switches. Because each encoder outputs a different number of bytes, codeword length is also tracked.

c) *Channel*: In a wireless channel, the target scenario for this project (although at present this project implements a simulated channel), codewords are further segmented and transmitted over the physical layer in smaller units determined by the modulation scheme. This project implements a binary symmetric channel (BSC) operating on individual bits. This requires a serializer to convert from codeword to bits along with a deserializer to compose the corrupted bits into a codeword again. The BSC uses an ADC combined with LFSR to generate 8-bit pseudorandom noise values and checks whether the value is less than a threshold corresponding to the desired bit error rate (BER). The channel also outputs a noisy 8-bit reliability value centered around a high mean for correct bits and a low mean for flipped bits.

d) *Decoders*: The project implements four decoders: Hamming, TMR, GRAND, and SRGRAND. As the GRAND decoder is the focus of this project, it will be given its own paragraph of explanation. However, GRAND or traditional decoder alike are plugged into the system in the same way:

they take corrupted codewords from the deserializer and output a single byte. The decoder is selected using switches 2 and 3. Because GRAND is an unpredictable, variable latency decoder, deserialized data is buffered and read once the previous decoder has raised its ready flag.

e) *GRAND*: A more detailed block diagram explaining GRAND is given in Fig. 1. When a GRAND module receives the start signal, it uses the noise pattern generator to generate noise patterns in order of likelihood. Each noise pattern is added to the corrupted codeword to form a guess, and that guess is validated using the appropriate `is_codeword` module. GRAND stops and returns valid when `is_codeword` returns 1. If GRAND does not find a codeword before the number of checks reaches its maximum (parameterized) value, it sets `abandoned` to 1 and returns the corrupted byte. Thus GRAND is significantly more complex than the other decoders, with dependence on multiple other modules and an unpredictable latency which can be as high as the maximum check value.

The difference between hard-decision GRAND and SR-GRAND is the way they determine the order of likelihood of noise patterns. Hard-decision GRAND assumes that lower weight noise sequences are more likely than higher weight noise sequences, but it treats all bits as equally likely to be flipped. SRGRAND is named for its exploitation of soft reliability information from the channel. It thresholds each reliability value to form a reliability mask and then generates noise patterns only for those bits which are marked unreliable by the mask. This offers the dual benefit of providing better accuracy while decreasing the search space, allowing for faster convergence of the decoder.

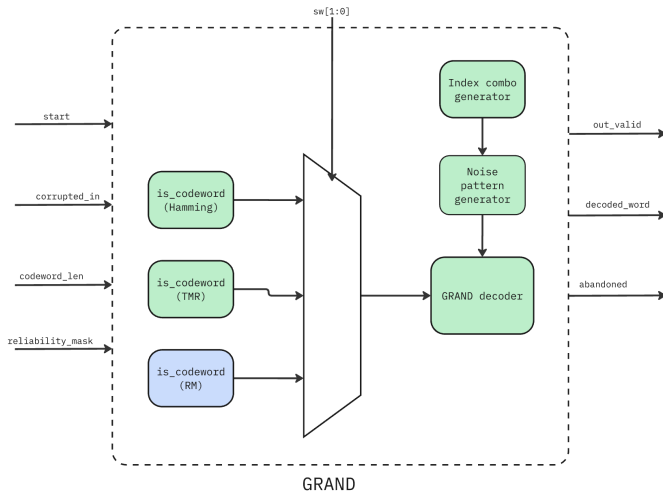


Fig. 1. Block diagram of GRAND implementation

III. DESIGN DETAILS

A. User I/O

The user I/O system comprises the entry and exit points of the system. This means the UART RX and TX modules,

which are unchanged from Lab 3, as well as the system which buffers original, corrupted, and decoded values and keeps them aligned when sending them off the FPGA board to the host PC. In the `top_level` module, there is a dedicated buffer each for original, corrupted, and decoded bytes. A state machine in the module waits for a decoded byte, at which point it sends data off-board using a simple time division multiplexing protocol:

[`TAG_O`][`orig_byte`][`TAG_C`][`corr_byte`][`TAG_F`][`fixed_byte`]

Each synchronization tag is a single character.

On the PC side, a bash script allows the user to type bytes to send to the FPGA manually or to send a predetermined message automatically. A python script handles the bytes received from the FPGA and sorts them according to tag, then outputs the separated streams to the terminal as seen in 2.

[illegible]

Fig. 2. Real-time coding output displayed in terminal.

B. Encoder Subsystem

The encoder subsystem allows the user to select between encoders using `sw[0:1]`. The encoders available are Hamming (12,8) and TMR (24,8). A lookup table-based Reed-Muller decoder was also implemented and synthesized, although not included in the final design.

a) *Hamming Encoder*: Hamming codes are widely known and used for scenarios with low bit error rates. For simplicity, encoder inputs were kept to single bytes, leading to a smallest possible code length of 12. The generator matrix G was passed to the encoder as a parameter, which uses combinational for loop and XOR logic to perform binary matrix multiplication and output a codeword. The module has single cycle latency and gated output.

b) *TMR Encoder*: Triple modular redundancy duplicates each bit three times, a straightforward operation using a for loop and gated output and resulting in a code rate of 8/24 or 1/3.

c) *Reed Muller Encoder*: This encoder was implemented specifically to validate the code-agnostic nature of the GRAND decoder core. GRAND requires only a means of checking if a received word is a valid codeword. The Reed Muller encoder was chosen for this slot because lookup-based encoding is fundamentally different from the combinational matrix multiplication used by Hamming, thus providing a comprehensive test for GRAND’s universality. It was implemented as a lookup table where the address is the 8-bit data byte and the value is

the codeword. The codebook was pre-generated using a python script and initialized into a xilinx BRAM as a .mem file. Out of a need to allot time to more core areas of the project, integration of this specific encoder was paused. The analytical resource usage (BRAM consumption) from its synthesis is used in the evaluation section of this paper to quantify the overhead of supporting lookup-based codes.

C. Channel Model

The channel subsystem comprises the serializer, deserializer, and channel model. The serializer and deserializer use the code length of the current encoder to determine the number of bits per codeword and output bits or codewords, respectively.

The channel model simulates a Binary Symmetric Channel (BSC) in which each bit has independent probability of being flipped. To determine whether to flip a bit, it relies on an ADC, an LFSR, and a desired bit error rate (BER). A floating ADC pin is read in as noise and XOR'ed with an LFSR value for stability. The resulting 8-bit noise value is compared with the BER value read off the upper 8 switches of the FPGA, and if below the BER, the channel bit is flipped. The LFSR alone is too predictable, but the ADC alone has either too uneven a distribution or too consistently low of values. A version of this design was tested where the ADC was connected to a microphone chip as well, but the noise from the microphone chip was very clearly not zero-mean and was too inconsistent to use in conjunction with a desired BER, hence the floating pin ADC.

The channel also outputs soft reliabilities for each bit. While a real wireless system would obtain soft reliability during the demodulation step, in this system a pseudo-soft reliability is calculated using two different uniform distributions centered around a low or high mean. If a bit is flipped, the mean is low, and if a bit is not flipped, the mean is high. The lowest bits of the LFSR are used as the jitter and added to or subtracted from the reliability mean to generate a soft reliability. This reliability is thresholded before being passed to the `reliabilities_deserializer`.

A note regarding the byte labeled as “corrupted” in the output stream: when the Hamming encoder is selected, the byte fed into the Corrupted FIFO is the first 8 bits of the deserialized output, corresponding to the identity columns of the Hamming generator matrix (which is in systematic form). When the TMR encoder is selected, the byte fed into the Corrupted FIFO is arbitrarily composed of every third bit from the deserialized output.

D. Decoder Subsystem

a) Hamming Encoder: Similar to the Hamming encoder, the Hamming decoder is parametrized with the appropriate parity matrix H , performs binary matrix multiplication using XOR operations, and has a gated output and single cycle latency.

b) TMR Encoder: The TMR decoder condenses each set of three bits via majority vote using addition. This could be done without an adder if necessary but the logic depth is not a bottleneck.

c) GRAND: The GRAND decoder is more general and also significantly more complex than the two specialized decoders. The overarching `grand_decoder` module depends on the modules `get_next_idx`, `get_noise_pattern`, `is_codeword_hamming`, and `is_codeword_tmr`.

get_next_idx. This module requires a significant amount of combinational depth and was eventually split across two clock cycles of latency in order to meet timing. Each time this module is used, it iterates across the list of input indices to check if any index can be incremented, and sets `found` high if so. Then if `found` is high, it increments that index as well as all following indices as needed. This requires multiple iterations across the indices as well as 8-bit adders. When complete, this module outputs the new list of indices along with a `valid` flag and a `done` flag indicating whether this is the last set of indices possible for the given list length.

get_noise_pattern. This module is essentially an FSM keeping track of the state of the `get_next_idx` module. When restarted using `restart`, it emits first the noise sequence of all zeros. After that, it continuously feeds the previously outputted list of indexes back into `get_next_idx` and outputs the noise pattern with 1's at the corresponding indices. Once it reaches the last set of indices for a given weight, it increases the weight and starts at the next set of lowest indices. This process continues until the weight reaches the maximum weight, set in the parameters. In my implementation, that maximum weight is 1 for hard-decision GRAND because the codewords used are small enough that flipping more bits often results in false positives. Once the last possible index list and weight are reached, it raises the flag `last_noise_pattern`.

is_codeword. These two modules use the Hamming parity matrix and a check for continuous triplets to verify whether a codeword is in the codebook. This requires for loops and XOR operations.

The overall GRAND module is launched by the input `start`, at which point it XORs the noise guesses output by the `get_noise_pattern` module with the corrupted input over and over until either `is_codeword` or `last_noise_pattern` goes high. This stacks the latencies of `get_next_idx` and `get_noise_pattern` back to back for however many noise patterns it takes to get a valid codeword. This is where the high latency and unpredictable variability of GRAND comes from.

d) SRGRAND: SRGRAND is an extension of GRAND and relies on all the same modules. The information from the soft reliabilities is passed to SRGRAND in the form of a reliability mask, a list of 0s and 1s. SRGRAND calculates the number of 1s in the mask and passes that number to `get_noise_pattern` as the length of the full word.

The key point of this project is that SRGRAND has much better performance because it is able to reasonably correct multiple bit errors in a single codeword. Since the length of the sub-noise pattern is shorter, SRGRAND benefits from both the fact that the search space for noise patterns is significantly reduced and that the noise patterns tested are much less

TABLE I
SUMMARY OF ERROR CORRECTION PERFORMANCE OVER 1000 BYTES

Encoder → Decoder	Meas. BLER (%)	Decoder Accuracy (%)	Efficiency (%)
Hamming → Hamming	10.7	93.8	42.06
Hamming → GRAND	11.1	94.3	48.65
Hamming → SRGRAND	9.6	97.3	71.88
TMR → TMR	13.6	93.6	52.94
TMR → GRAND	13.4	89.1	18.66
TMR → SRGRAND	13.1	97.1	77.86

likely to result in false positives. The Hamming (12,8) code used prior in this project can correct maximum one error per codeword, so this is an area where SRGRAND clearly performs better as long as the soft reliabilities are more often accurate than they are misleading.

IV. EVALUATION

A. Latency and Throughput

The project meets timing requirements. Total slack is 1.935ns and the longest path is my_srgrand/my_is_codeword_tmr/is_codeword_reg_0 with 13.01ns. This was after splitting the get_next_idx across two cycles - if that module is run in one clock cycle, it has a combinational depth of 20 and requires 15.105ns resulting in a slack of -0.017ns and failing to meet timing. In both cases, these are module which GRAND relies on each time it makes and checks a guess, and therefore there is incentive to keep these submodules to as few cycles as can be accommodated by the timing.

B. Resource Utilization

Although GRAND presents challenges with latency, it is highly efficient from a resource usage perspective. The overall utilization figures on the FPGA are very low, providing substantial margin for future implementation of more complex versions of GRAND such as ORBGRAND [2] or SOGRAND [3]. Utilization metrics include:

- Slice LUTs: 1219/32600 (3.74 %)
- Slice Registers: 836/65200 (1.28 %)
- DSP Blocks: 0/120 (0.00 %)
- Block RAM Tiles: 0/75 (0.00 %)

The current 0% utilization of DSP Blocks and Block RAM Tiles is intentional, confirming that the hard-decision design relies purely on combinational and sequential logic. This leaves all 120 available DSPs for the fixed-point multiplication and complex noise modeling required by the soft-decision ORBGRAND channel, and all 75 BRAM tiles available for storing the reliability buffers and the full lookup table for an arbitrary code. The synthesis of the project version which included a Reed-Muller (2,4) encoder (i.e. all 256 possible bytes mapped to 16 bit codewords) has the following BRAM utilization:

- Block RAM Tiles: 0.5/75 (0.67 %)

which demonstrates that resource utilization for small codes is not an issue. The system is thus very scalable.

C. Experimental Data

To validate the theory and intuitions built up from writing these decoders, accuracy tests were run for each combination of encoder and decoder. These results are reported in Table I. The measured block error rate (BLER) is the number of bytes which are corrupted before decoding, and the fact that it is higher for TMR than for Hamming is likely inherent to the code length; TMR codewords are longer and therefore have more chances to have an error occur. The decoder accuracy is the accuracy rate of the decoded output. The efficiency is calculated as Errors Corrected / Total Errors and therefore is an indication of the effectiveness of coding independent of BLER.

The primary trend in these results is to show that while the accuracy of GRAND is sometimes unpredictable, SRGRAND emerges as the clear winner for the decoders tested. In each case, SRGRAND has the highest accuracy as well as an improvements in efficiency of 20-30%. This shows that when hard-decision GRAND is augmented with soft information, it becomes a genuinely interesting area for research.

D. Project Deliverables

This project achieves the design goals stated in the preliminary report of a real-time, interactive error correction learning and research tool exploring error correction and GRAND variants in particular. A few of the stretch goals have become future work, such as a greater variety of encoders (including a lookup-based encoder) and implementation of more complicated GRAND variants such as ORBGRAND or SOGRAND. Because the system has been designed to be modular, addition of these components would be very smooth, and in fact the Reed-Muller and Hamming SECDED encoders have already been written but not integrated.

V. CONCLUSION

This project demonstrates a modular FPGA-based system for real-time error correction, featuring multiple encoder/decoder pairs and a working implementation of hard-decision GRAND and SRGRAND. The system supports live text input over UART and visualizes corrupted versus corrected messages to highlight the effects of channel noise and coding. As mentioned previously, future work includes more encoder types and more recent and complex GRAND variants. Another major improvement to realism would be the addition of an actual RF wireless channel in order to use more realistic

soft-reliabilities. This system provides a flexible foundation for exploring modern decoding strategies in hardware.

VI. CONTRIBUTIONS

All work on this project was done by Laura Landon (solo project). My colleagues Lukas Rapp and Joe Griffin helped me to better understand some of the GRAND variants that form the basis of their research.

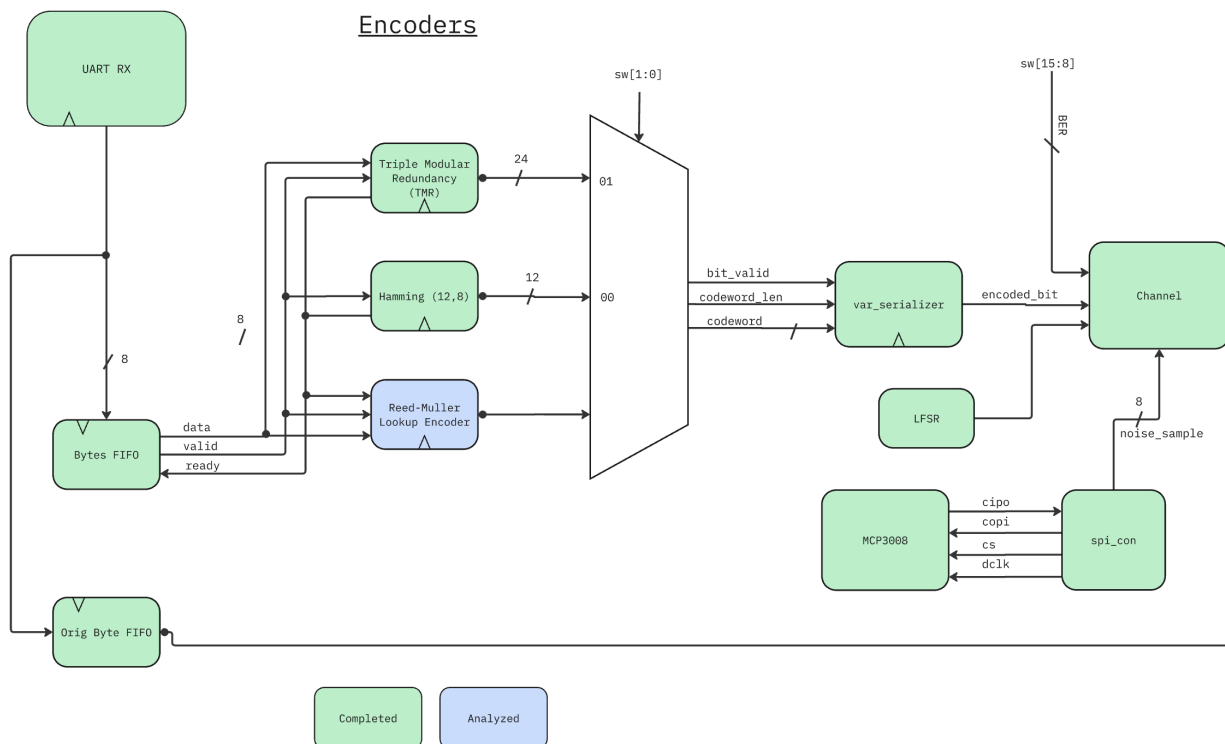


Fig. 3. System block diagram (Part 1): Encoding plus channel.

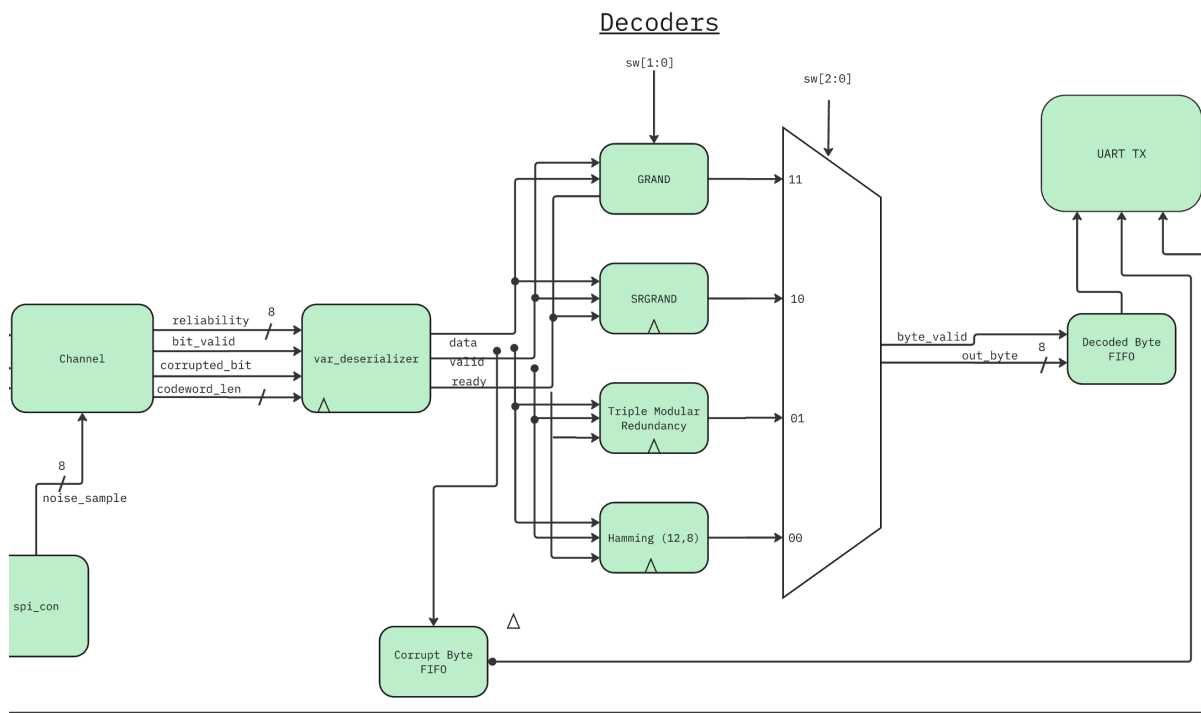


Fig. 4. System block diagram (Part 2): Channel plus decoding.

REFERENCES

- [1] K. R. Duffy, J. Li, and M. Médard, “Capacity-achieving guessing random additive noise decoding,” *IEEE Transactions on Information Theory*, vol. 65, no. 7, pp. 4023–4040, 2019.
- [2] K. R. Duffy, W. An, and M. Médard, “Ordered reliability bits guessing random additive noise decoding,” *IEEE Transactions on Signal Processing*, vol. 70, pp. 4528–4542, 2022.
- [3] P. Yuan, M. Medard, K. Galligan, and K. R. Duffy, “Soft-output (so) grand and iterative decoding to outperform ldpcs,” *arXiv preprint arXiv:2310.10737*, 2023.