Adv VLSI Final Report - FT8 Encoder

Hisen Zhang <zhangz29@rpi.edu>

April 24, 2024

For the final project of the Advanced VLSI course, I designed and implemented a hardware encoder for a digital mode called FT8, which is used for radio weak signal communication. FT8 employs a structured protocol with specific timing and signal characteristics to facilitate reliable communication under challenging conditions.

Background

The FT8 mode operates within 15-second time slots, where 12.6 seconds are allocated for transmission (Tx), and the remaining time is used for other operations such as synchronization and decoding. The mode utilizes a narrow bandwidth of 50 Hz and employs Audio

Frequency-Shift Keying (AFSK) modulation with 8 tones (AFSK-8). The tone spacing is set to 5.86 Hz, resulting in a data rate of 6.25 baud.

Each FT8 transmission carries a payload of 77 bits, which undergoes forward error correction (LDPC) to enhance the reliability of the transmitted data. The protocol incorporates domain-specific encoding to enable the exchange of structured messages, including information such as call signs, received and transmitted signal strength, and maidenhead grid square locator.

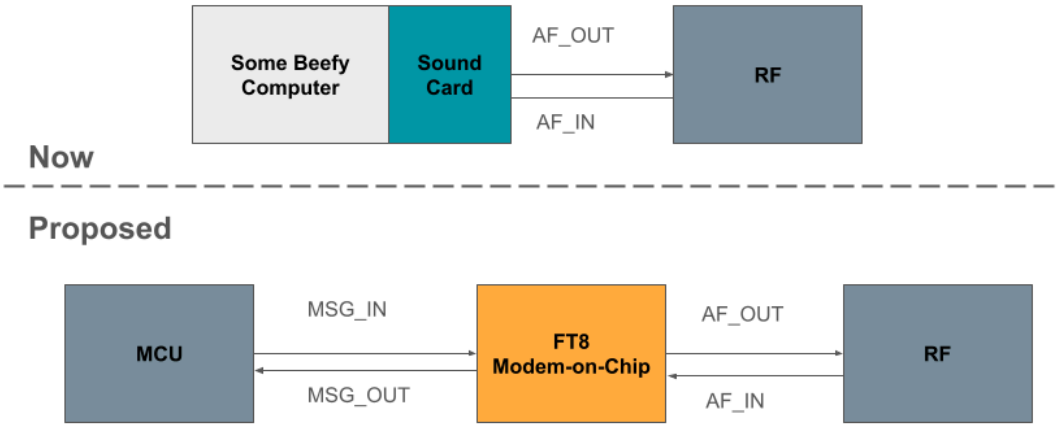
Related Work

The FT8 mode has been well-documented and extensively studied within the amateur radio community. The original specification for FT8 was proposed by Joe Taylor, K1JT, and his colleagues in a QEX article [1]. This specification laid the foundation for the FT8 protocol and provided insights into its design and implementation.

The WSJT-X software suite, developed by the WSJT development team, includes a Fortran implementation of the FT8 protocol [2]. This implementation serves as a reference for understanding the inner workings of FT8 and has been widely used by the amateur radio community.

Additionally, there have been efforts to develop open-source implementations of FT8 in various modern programming languages. One notable example is the Python FT8 demodulator by Robert Morris, AB1HL [3]. This demodulator provides a clear and concise implementation of the FT8 decoding process, which can be used as a starting point for understanding the protocol and developing hardware implementations.

Motivation



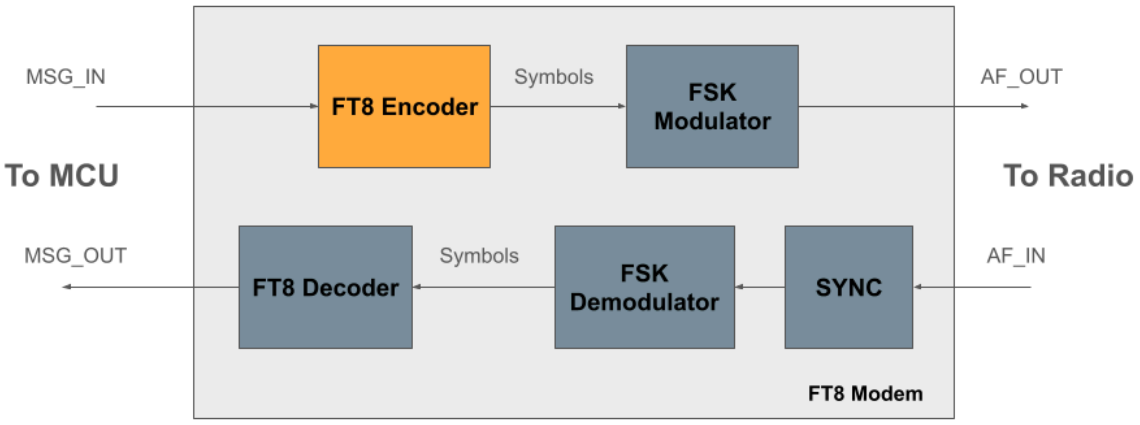
The motivation behind this project stems from the limitations of existing FT8 implementations and the potential benefits of a hardware-based solution. Currently, all existing implementations of FT8 are software-based, requiring a high-performance PC to handle the modem functionality. This dependency on a powerful computer limits the potential use cases of FT8, such as in field communication scenarios or emergency communication situations where a dedicated PC may not be available or practical.

Moreover, the FT8 protocol has reached a stable state and is not actively undergoing further development. This stability makes it an ideal candidate for hardware implementation, as the protocol specifications are well-defined and unlikely to change significantly in the near future.

Implementing the FT8 modem on a chip offers several advantages over software-based solutions. Firstly, a hardware implementation can provide significant improvements in terms of space and power efficiency. By leveraging dedicated hardware components and optimized digital signal processing (DSP) techniques, the FT8 modem can be realized in a compact and low-power form factor.

Furthermore, a hardware-based FT8 modem can benefit from the inherent parallelism and performance advantages of custom hardware architectures. By exploiting the specific characteristics of the FT8 protocol, such as its structured message format and error correction schemes, the hardware implementation can achieve higher throughput and lower latency compared to software-based solutions.

Scope of the Project

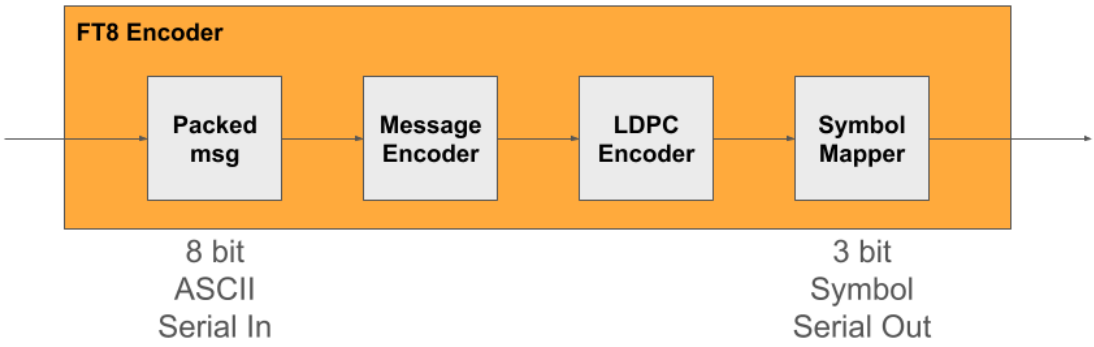


Developing a complete FT8 modem on a chip is a significant undertaking that requires substantial time and resources. Considering the scope of this project within the context of the Advanced VLSI course and the limited time available, I decided to focus specifically on the encoder component of the FT8 modem which has more relevant concepts and techniques covered in the course, such as digital design, pipelining, and error correction coding.

System Design and Implementation

The FT8 encoder hardware design consists of a top-level module named ft8_modulator, which encapsulates the entire encoding process. This top-level module instantiates three submodules: packed_message_encoder, ldpc_encoder, and symbol_mapper. Each submodule represents a distinct stage in the encoding pipeline and is responsible for a specific set of operations.

The input to the FT8 modulator is an 8-bit ASCII character, denoted as ascii_in, accompanied by a data_valid signal that indicates the validity of the input data. The output of the modulator is a 3-bit symbol, represented by symbol_out, along with a symbol_valid signal that signifies the validity of the output symbol.



Message Encoder

The encoding process begins in the packed_message_encoder submodule. This module receives the input ASCII characters and performs the necessary packing operations to generate a compact representation of the message. It concatenates the ASCII characters into a 72-bit message, appends three zero bits for padding, and calculates a 12-bit Cyclic Redundancy Check (CRC) value. The resulting output is an 87-bit packed message that includes the original message content, padding bits, and the CRC checksum.

LDPC Encoder

The packed message is then passed to the ldpc_encoder submodule, which applies Low-Density Parity Check (LDPC) encoding to the message. The ldpc_encoder takes the 87-bit packed message and encodes it into a 174-bit codeword using the LDPC(174,87) code. This code is specifically designed for the FT8 protocol and provides robust error correction capabilities.

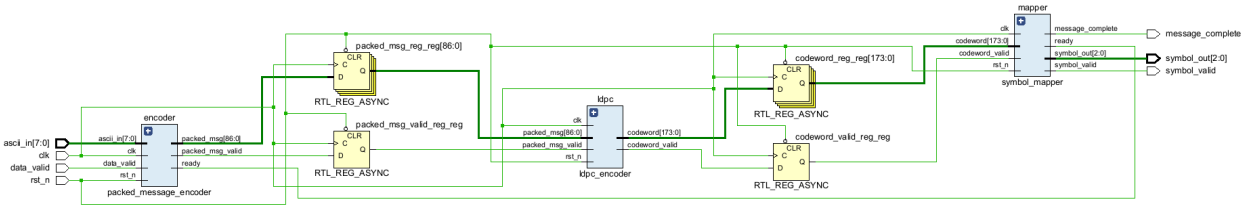
Symbol Mapper

After LDPC encoding, the 174-bit codeword is processed by the symbol_mapper submodule. This module is responsible for converting the codeword into a sequence of symbols that can be transmitted over the air. It divides the codeword into 58 groups of 3 bits each and maps each group to an 8FSK symbol. The 8FSK modulation scheme uses eight distinct frequency tones to represent different symbol values, with each tone corresponding to a specific 3-bit pattern.

In addition to the regular symbols, the symbol mapper inserts special synchronization sequences called Costas arrays into the symbol stream. Costas arrays are predefined sequences of symbols that aid in signal detection, synchronization, and timing recovery at the receiver. The symbol mapper inserts three Costas arrays at specific positions within the symbol stream, namely at the beginning, middle, and end of the transmission, per the FT8 specification.

The output of the symbol_mapper is a sequence of 79 3-bit symbols, which includes the encoded message symbols interspersed with the Costas arrays. The symbol_out signal represents the current symbol being output, and the symbol_valid signal indicates the validity of the output symbol.

Pipelining The Flow



To maximize the throughput and efficiency of the FT8 encoder, the design incorporates pipelining techniques. Pipelining allows multiple messages or symbols to be processed concurrently at different stages of the encoding process. The packed_message_encoder, ldpc_encoder, and symbol_mapper submodules each represent a pipeline stage, and registers are inserted between the stages to hold intermediate data and maintain proper synchronization.

By pipelining the encoding process, the FT8 encoder can achieve higher throughput and resource utilization. While one message is being encoded by the LDPC encoder, the packed_message_encoder can start processing the next message in parallel. Similarly, while one codeword is being mapped to symbols, the LDPC encoder can encode the next codeword concurrently. This overlapping execution of different pipeline stages allows for efficient utilization of hardware resources and reduces the overall encoding latency.

Implementation Details

CRC

The CRC calculation is a critical component of the FT8 encoder, as it provides a means to detect errors in the transmitted data. The FT8 protocol employs a 12-bit CRC checksum, which is calculated using the polynomial x^12 + x^11 + x^3 + x^2 + x + 1 (from [2]). This polynomial can be represented in binary as 1100000001111 or in hexadecimal as 0xC0F.

The CRC calculation process involves the following steps:

1. Initialize a 12-bit register, crc, to store the CRC value, and a 72-bit register, temp, to store the input data.
2. Iterate over each bit of the input data (72 bits) using a loop.
3. For each iteration:

    - Perform an XOR operation between the most significant bit (MSB) of temp and the MSB of crc.
    - If the result is 1, shift crc to the left by 1 bit (discarding the MSB) and XOR it with the CRC-12 polynomial 0xC0F.
    - If the result is 0, simply shift crc to the left by 1 bit (discarding the MSB).
    - Shift temp to the left by 1 bit (discarding the MSB) to process the next bit in the next iteration.
4. After processing all 72 bits, the final value of crc represents the calculated CRC-12 checksum.
5. Return the 12-bit CRC value.

The computed CRC-12 checksum is then appended to the 72-bit message along with 3 zero padding bits, forming the 87-bit packed message that is passed to the LDPC encoder.

LDPC

The LDPC encoding adds redundancy to the message, enabling error correction capabilities at the receiver side. The specific structure of the LDPC code used in FT8 is designed to provide robust error correction while maintaining a reasonable encoding and decoding complexity. The LDPC encoder takes the 87-bit packed message and generates a 174-bit codeword using the LDPC(174,87) code specified in the WSJT-X Fortran implementation [2].

The LDPC encoding process involves the following steps:

1. Declare a register, codeword_sys, to store the 87-bit systematic codeword, and a register, codeword, to store the final 174-bit codeword.
2. Initialize codeword_sys with the input packed message.
3. Begin the LDPC encoding process using the generator matrix, Nm. The generator matrix is used to generate the parity bits of the codeword.
4. Copy the systematic bits (the original 87 bits) from codeword_sys to the corresponding positions in codeword.
5. Iterate over each row of the generator matrix, Nm. For each non-zero entry in the matrix, XOR the corresponding systematic bit with the parity bit at the position specified by the matrix entry. This step generates the parity bits of the codeword.
6. After generating the parity bits, apply a column permutation to the codeword. The column permutation reorders the bits of the codeword according to the colorder array, which is defined based on the FT8 protocol specification.
7. Return the 174-bit LDPC-encoded codeword.

Evaluation

Testbench

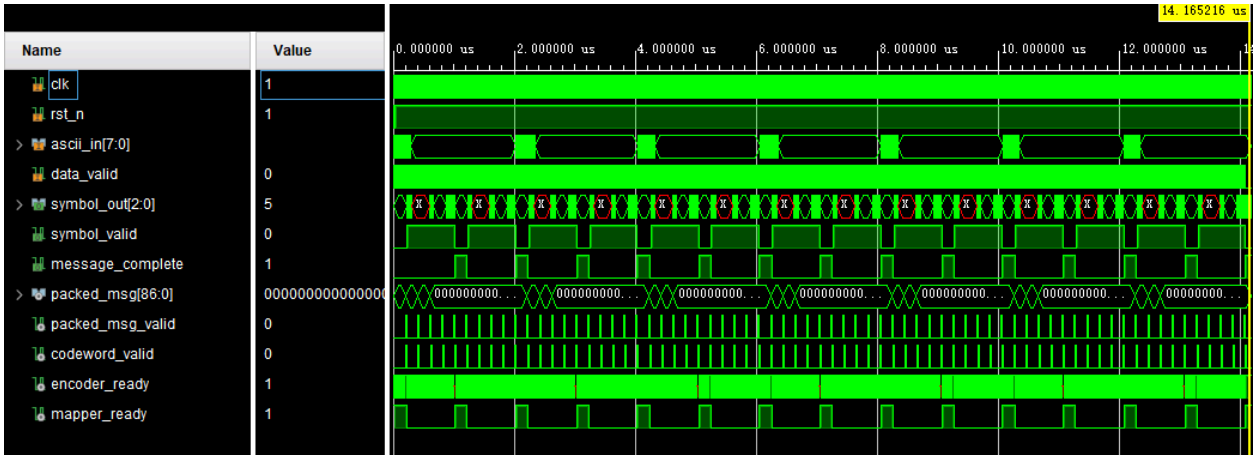
The testbench for the FT8 encoder reads a set of predefined input messages and measures the outputs. These messages are representative of typical FT8 transmissions. Below are FT8 message example:

- `CQ KI7PO DN06`
- `KI7PO IZ1M JN35`
- `IZ1M KI7PO -10`
- `KI7PO IZ1M R-12`
- `IZ1M KI7PO RRR`
- `KI7PO IZ1M 73`
- `IZ1M KI7PO 73`

The testbench processes the input messages character by character. It assigns the ASCII value of each character to the ascii_in signal and asserts the data_valid signal for one clock cycle to indicate the validity of the input character. The FT8 encoder expects the input characters to be provided sequentially, with the data_valid signal synchronized with the ascii_in signal.

The encoded symbols are output through the symbol_out signal, with the symbol_valid signal indicating the validity of each output symbol. The testbench captures these output symbols. The testbench repeats this process for each input message in the test case array, ensuring that the FT8 encoder can handle different message patterns and generate the expected output symbols consistently.

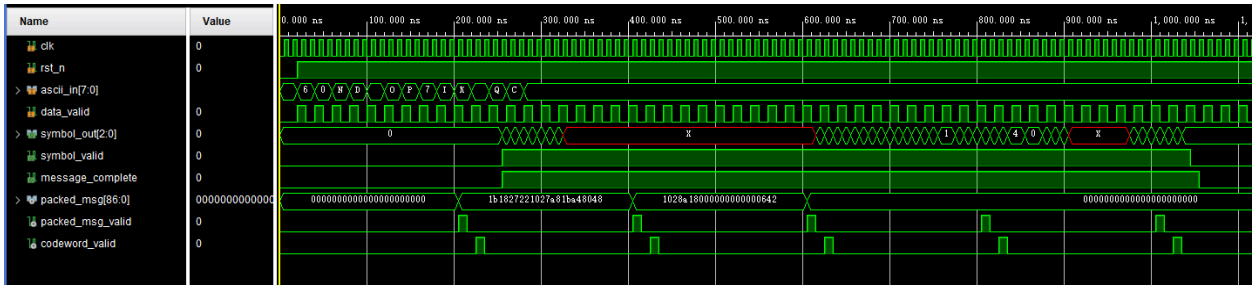
Output



The output waveforms show the symbol streams generated for each input message. The first test case demonstrates a complete FT8 encoding cycle. The ascii_in signal represents the input ASCII characters, data_valid indicates when the input data is valid, symbol_out shows the 3-bit output symbols, and symbol_valid indicates when the output symbols are valid.

The symbol_mapper submodule generates the output symbols in a sequential manner, with each symbol being output for one clock cycle. The symbol_out signal represents the current symbol being output, and the symbol_valid signal indicates the validity of the output symbol. The symbol_valid signal is asserted for each valid output symbol, aligned with the corresponding symbol_out value. As shown in the waveform, the entire sequence of 79 symbols is output consecutively, with the symbol_valid signal asserted for each symbol.

The packed_message_encoder, ldpc_encoder, and symbol_mapper submodules represent different pipeline stages. While the LDPC encoder is encoding one message, the packed_message_encoder can start processing the next message in parallel (see the valid signals; they are not fully synchronized because data are being sent serially and take additional clock cycles to reach the next stage). Similarly, while one codeword is being mapped to symbols, the LDPC encoder can encode the next codeword concurrently.

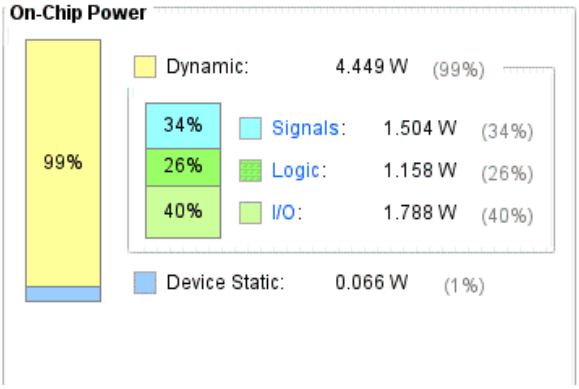


However, it was observed that the symbol_out signal contained 'X' values in certain portions of the waveform. This issue is likely caused by uninitialized or undefined values in the codeword_reg register within the symbol_mapper module. To resolve this, proper initialization and synchronization of the codeword_reg register should be ensured. Additionally, adding default assignments or conditional statements to handle undefined cases can mitigate the issue.

Power

Power estimation from Synthesized netlist. Activity derived from constraints files, simulation files or vectorless analysis. Note: these early estimates can change after implementation.

Total On-Chip Power:	**4.515 W**
Design Power Budget:	**Not Specified**
Power Budget Margin:	**N/A**
Junction Temperature:	**48.8°C**
Thermal Margin:	51.2°C (9.7 W)
Effective ϑJA:	5.3°C/W



The power performance of the FT8 encoder was evaluated using power analysis tools. The results show that the encoder consumes a relatively low amount of power, making it suitable for low-power applications with embedded controllers. The pipelined architecture and efficient design contribute to the power efficiency of the encoder.

However, further optimizations can be explored to reduce power consumption even further. The dynamic power takes 99%, and 40% of it goes into I/O, likely due to the serial I/O design. This is a trade-off, as 77 bits of packed message is probably too much for parallel I/O for a small form-factor chip. Still, techniques such as clock gating, and power gating can be employed to minimize power dissipation during idle periods or when the encoder is not actively processing data.

Limitation and Future Work

1. The current implementation only includes the FT8 encoder. To enable two-way communication, a decoder needs to be developed as well. Integration with other components of a complete FT8 transceiver system, such as the RF front-end and digital signal processing (DSP) blocks, should be explored to create a fully functional FT8 modem on a chip.

2. Each module in the encoder can be further optimized for performance and resource utilization. For example, the LDPC encoder can be optimized using techniques like parallel processing or efficient matrix multiplication algorithms.

3. The pipelining scheme can be further refined by breaking down the modules into smaller stages, allowing for finer-grained parallelism and potentially higher throughput. Also, adding clock gating features will further reduce the power consumption of the encoder, given that each transmission will be at least 15 seconds apart in the FT8 protocol.

Conclusion

In this project, a hardware encoder for the FT8 digital mode was successfully designed and implemented. The encoder incorporates techniques such as message packing, CRC calculation, LDPC encoding, and symbol mapping to generate the modulated symbol stream for transmission.

The pipelined architecture of the encoder enables improved throughput and parallel processing of messages and symbols. The encoder demonstrates power efficiency, making it suitable for low-power applications.

While the current implementation focuses on the encoder component, future work can expand the design to include the demodulator and optimize individual modules for better performance and resource utilization. Integration with other components of an FT8 transceiver system can lead to the development of a complete FT8 modem-on-chip.

Overall, this project showcases the feasibility and potential benefits of implementing the FT8 protocol in hardware, opening up possibilities for more efficient and compact weak signal communication systems using the knowledge and techniques learned from this course.

Bibliography

[1]
Alex, “alexranaldi/wsjt_fort,” *GitHub*, Aug. 04, 2021. https://github.com/alexranaldi/wsjt_fort (accessed Apr. 24, 2024).
[2]
S. Franke, B. Somerville, and J. Taylor, “The FT4 and FT8 Communication Protocols Motivation and design of the digital modes FT4 and FT8, and some details of how they are implemented in WSJT-X,” 2020. Available: https://wsjt.sourceforge.io/FT4_FT8_QEX.pdf
[3]
R. Morris, “rtmrtmrtmrtm/basicft8,” *GitHub*, Dec. 26, 2023. https://github.com/rtmrtmrtmrtm/basicft8 (accessed Apr. 24, 2024).