Flat Earth Project Final Report

1st Nathan Jones
Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
Cambridge, MA, USA
nathandj@mit.edu

2nd Samuel Calvert
Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
Cambridge, MA, USA
samue100@mit.edu

3rd Sawyer Sands
Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
Cambridge, MA, USA
szsands@mit.edu

***Abstract*—We present a low-rate picture transmission (LRPT) decoder to receive and process signals from a Meteor-M series weather satellite for the purpose of viewing images of Earth transmitted directly from space. This system will demodulate and decode the transmission then perform error correction and packet parsing on the bitstream to extract the image data. Then an image of the Earth will be displayed to a screen via HDMI.**

***Index Terms*—Interleaved codes, low earth orbit satellites, phase shift keying, Reed-Solomon codes, satellite broadcasting, Viterbi algorithm**

I. Introduction

The Meteor-M series weather satellites openly broadcast images taken of Earth in the form of JPEGs. However, the image data is embedded among the various analytics of the satellite and the information of several other instruments. Furthermore, all this data goes through several layers of error correction to safeguard from various kinds of noise it could experience while traveling to Earth. Lastly, the data is modulated into a radio transmission and broadcast to Earth. The challenge this system handles is removing the various layers separating the received radio transmission from the image data.

A bitstream is first recovered by demodulating the received transmission's offset quadrature phase-shift keying (OQPSK) modulation. A synchronization word is then found to identify the start of a packet. Each packet is then convolutionally de-interleaved and decoded, the latter using a Viterbi decoder. Pseudo random noise is then removed from each packet. Afterward, each packet's block interleaving is undone and Reed-Solomon decoding is performed. The packet is then re-interleaved and manipulation of the bitstream is complete. The JPEG data is then extracted from the packets and displayed to a screen via HDMI.

II. Physical Construction and RF Receiver

The Meteor M2-3 satellite transmits an OQPSK signal at 137.1MHz, with a baseband transmission rate of 80Kbps. The receiver consists of a v-dipole antenna, a low noise amplifier, and an RTL-SDR using the RTL2832U chipset. The SDR should be tuned to the center frequency of 137.1MHz and have bandwidth large enough to capture all components of the modulated signal, which has at most 150KHz of bandwidth [1]. The recommended sample rate on this chipset is 0.9Msps. After mixing and filtering, complex samples are streamed to demodulation processes onboard a personal computer.

III. Demodulation

At this point, the I and Q samples have been sampled by an ADC driven by an external sampling clock. The tasks of timing and carrier recovery are left to a discrete receiver architecture on the host computer. To accomplish the first task of symbol synchronization, the recovery system needs to determine the symbol period as well as the sampling phase. This allows for sampling at the center of the symbol period, thereby reducing intersymbol interference. Note that since the signal has already been discretized, an interpolating filter is used to adjust the sampling phase and frequency. The interpolated samples are passed through a matched filter and then a Mueller and Mueller detector estimates the timing error. The error is then fed back through a second-order loop filter to the interpolator.

Since there is likely a frequency offset between the local and transmitter carrier clocks, the received signals on the constellation diagram will undergo continuous rotation. To correct this offset, the exact carrier frequency and phase must be determined. Carrier recovery is accomplished via a digital phase locked loop with an IIR loop filter and numerically controller oscillator. A decision directed phase detector is used to determine the phase error and the symbol decisions [2]. The OQPSK demodulator is essentially the QPSK demodulator described above, but with the Q-channel delayed by half a symbol period. Existing implementations for this OQPSK demodulator have been found online [3]. Based on this implementation, the I and Q symbols are demodulated as 8-bit two's complement non-return-to-zero integers, which are then transmitted over UART at 100MHz to the Urbana Spartan-7 FPGA.

IV. Decoding

According to the Meteorlogical Operational satellite(METOP) standard, the physical transmission layer for LRPT specifies that the transmitted packets are convolutionally encoded using generator polynomials $G_1 = 1111001$ and $G_2 = 1011011$ to generate two outputs of parity bits, forming the I and Q channels respectively. To further reduce the effects of burst errors, the convolutionally encoded packets are interleaved using a Forney convolutional interleaver. An 8-bit unique word(UW) is inserted at the start of any 72 bit packet to assist in synchronization of the deinterleaver at the receiver.

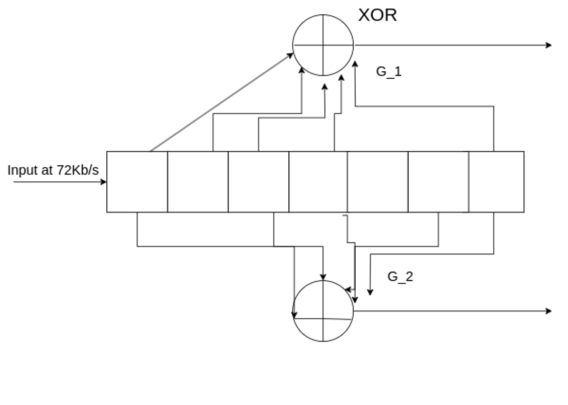

Fig. 1. Convolutional encoding using a length 7 shift register with $r = 1/2$

Thus, the first step in decoding is to deinterleave the serialized 80 bit packet. Deinterleaving requires finding the most likely position of the 8 bit UW(0x27) within interleaved frames, so that the frames can be aligned for the deinterleaver. This implementation performs a correlation operation on sets of 32 frames, utilizing four 18Kbit BRAMs. Then, based on the rotation of the heighest weighted UW, this set of frames is then derotated to resolve the phase ambiguity that may result if the carrier synchronizer locks onto a phase other than the reference phase.

The derotated symbols are then sent to the Forney convolutional deinterleaver. The deinterleaver consists of 36 branches with an elementary delay of $M = 2048$. Symbols are sent and read sequentially over the branches. The delay in branches increases linearly, so the deinterleaver stores at most $M \cdot 8 \cdot \sum_{i=0}^{35} i \approx$ 10Mb. Therefore, storing the symbols within the deinterleaver requires use of Urbana's DDR3 SDRAM. Calculating the address requires large modulo operations to calculate the address as well as crossing clock domains. The DDR3 can output about 128 bits per 250-300 ns, meaning that the deinterleaver is a bottleneck in the LRPT pipeline [13].

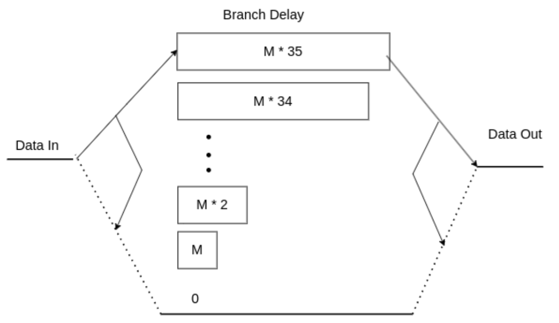

Fig. 2. Forney deinterleaver with elementary branch delay $M = 2048$ and 36 branches

The signed 8-bit symbols from the deinterleaver are differentially decoded and then passed through the Viterbi decoder. The two's complement soft decision data is first converted to binary offset and then parallelized into I and Q inputs. To recover the intended message, the Viterbi decoder uses the received encoder outputs to determine the maximum a posteriori estimate of the state sequence. In this problem, the $2^{k-1} = 64$ states are given by the bits in the shift register. The Viterbi algorithm is best understood as a process on a two-dimensional array of vertices, with each column containing 64 vertices corresponding to the states. On each time step, the encoder receives a bit as input and generates two output bits. Each state in the trellis column has an out-degree of two, with each transition corresponding to receiving either a one or zero as the input bit.

To determine the sequence of states, and thus the sequence of input bits, the decoder assigns a state metric(SM) to each vertex. This metric is a measure of how likely the decoder is to be in that state on the current time step. Another metric, known as the branch metric(BM) is assigned to each transition, which provides a measure of the difference of actual and expected output values for that transition.

To compute the SM for each node at timestep t, we have the following recursive definition for state i with ancestor nodes j and k:

$$SM[i]_t = \min\{SM[j]_{t-1} + BM[(j,i)]_t, SM[k]_{t-1} + BM[(k,i)]_t\}$$

The BM unit calculates squared Euclidean distance between the expected and received output values. The noisy 8-bit output values are converted to offset binary, where the Euclidean norm is found for each of the expected binary output pairs: $\{11, 10, 01, 11\}$ to the noisy values.

Calculating the current state metric falls to the Add Compare Select(ACS) module. The ACS associated with a given state sums the branch metric and previous state metric for each incoming path. The sums are compared, and the updated state metric and the encoders likely input are output. Because the state metrics are liable to overflow, some form of normalization of the metrics is required. Additionally, since the distance between any two state metrics is not necessarily bounded, popular techniques like modulo normalization are not possible. [12] As a result, this paper's implementation is more rudimentary. If m is the maximum state metric value, then when the smallest state metric is greater than $\lfloor m/2 \rfloor$, a value of $\lfloor m/2 \rfloor$ is subtracted from each state metric. The expensive comparison and subtraction operations must be done in a separate cycle, requiring stall logic within this part of the pipeline.

The decisions from each module as well as the most likely previous states are fed into a Traceback Unit(TBU). The maximum likelihood estimate of the sequence of encoded values can be found be tracing back the decisions starting from the current state with the smallest metric. The decisions from this traceback operation are the reversed sequence of encoded values. Because comparing all the state metrics is infeasible and the traceback memory is limited, an alternate scheme for traceback implementation is required.

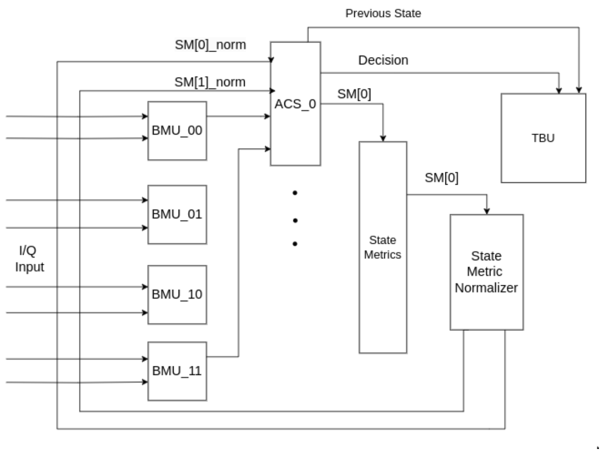

Fig. 3. Viterbi Decoder Block Diagram with single ACS module

This implementation uses the remarkable property that all survivor paths merge to a single state. That is, if every path is traced back X stages, then all paths will converge to a common state. For our encoder with 64 states and rate $1/2$, X turns out to be approximately 28. This property allows for a traceback scheme involving a memory block and read and write pointers. The survivor memory, storing previous states and descision bits for the nodes at each timestep, is $S = 120$ entries deep. A write pointer loops through the memory in order of decreasing address. There are 2 traceback pointers which move in the other direction through the survivor memory. Each traceback pointer traces back $X = 30$ elements before reading $B = 30$ decision bits out. When a traceback pointer encounters the write pointer, the traceback pointer begins its traceback and read cycle. Since B was chosen precisely to be 30, when the traceback pointer finishes its read cycle, it will collide with the write pointer on that same clock cycle. Note also that the write pointer is initialized at the 0th index at the first timestep [11].

Because there can be at most two simultaneous read or write operations on any column or row in the survivor memory, two identical survivor BRAMs both with dual read and write ports must be instantiated. The traceback read results are then fed into a BRAM acting as a stack, off of which the Viterbi decisions are read.

We then use these Viterbi decisions to find the start of a Channel Access Data Unit(CADU). Each CADU is 1024 bytes in length. Another parameterized instance of the UW synchronization module correlates the stream in 8 CADU size frames against the word 0x1ACFFC1D to align the packet. Because the input stream was derotated before heading into the Viterbi decoder, the Viterbi decisions only need to be correlated against this word. The CADUs are then sent to the desrambler.

A. Descrambler

To avoid long sequences of the same bit state, the satellite has added pseudo-random noise generated by a Fibonacci linear feedback shift register (LFSR) described by the following polynomial [1]:

$$h(x) = x^8 + x^7 + x^5 + x^3 + 1 \tag{1}$$

To remove the noise, the 255 byte sequence generated by the LFSR is XOR'd with each of the four 255 byte sequences within a CVCDU. The descrambler module takes in a data byte and XOR's it with its respective pseudo-noise byte generated by the LFSR contained within a separate module. The LFSR sequence restarts every 255 bytes or when a signal indicating a new CVCDU is received.

Modules	Slices	Slice LUTs	BRAMs	Power (W)
Viterbi	24.47%	16.88%	21.33%	0.273
CADU Correlation	4.99%	3.75%	3.33%	0.102
UW Deinterleave Correlation	5.77%	4.35%	2.67%	0.105

TABLE I
MODULE UTILIZATION ON SPARTAN-7

V. REED-SOLOMON

Reed-Solomon error correction appends 32 parity bytes to every 223 bytes of data. This type of Reed-Solomon is termed RS(255, 223) given the 223 byte message word and 255 byte code word. RS(255, 223) can correct up to 16 corrupted bytes of the message word.

The 8-bit symbols of the code and message words are elements of GF(2^8) where GF stands for Galois Field. The elements are generated using the following field generator polynomial over GF(2) [4]:

$$F(x) = x^8 + x^7 + x^2 + x + 1 \tag{2}$$

where $F(\alpha) = 0$ and $\alpha = 0b00000010$.

A. Encoding

On the transmitter side, a code generator polynomial $g(x)$ is used to encode the data. The equation g(x) is defined as [4]

$$g(x) = \prod_{j=128-16}^{127+16} (x - \alpha^{11j}) = \sum_{i=0}^{32} G_i x^i. \tag{3}$$

The 32 roots of $g(x)$ will be referred to as α^z for the remainder $\alpha^{11(112+j)}$ of the paper.

The 223 symbols of the message word are treated as the coefficients of a polynomial

$$m(x) = m_0 + m_1 x + m_2 x^2 + ... + m_{222} x^{222} \tag{4}$$

where m_0 is the last symbol received and m_{254} is the first. The polynomial $m(x)$ is multiplied by x^{32} to get

$$m(x)x^{32} = m_0 x^{32} + m_1 x^{33} + m_2 x^{34} + ... + m_{222} x^{254} \tag{5}$$

The polynomial $m(x)$ is then divided by (3). The 32 symbols of the remainder are the parity symbols. They are appended to the end of message word to form the 255 symbol code word that will be transmitted. The code word is therefore represented as polynomial

$$c(x) = c_0 + c_1 x + c_2 x^2 + ... + c_{254} x^{254}. \tag{6}$$

B. De-Interleaver

To reduce the effect of burst noise on sequential symbols of data by spreading the noise across multiple code words, the transmitter interleaves the symbols of each code word with a depth of four. The de-interleaver module sends each received symbol to one of four Reed-Solomon instances, incrementing through each sequentially using a counter. Reed-Solomon is then performed on each 255 symbol long section of a CVCDU. The process of de-interleaving is shown in Fig. 7.

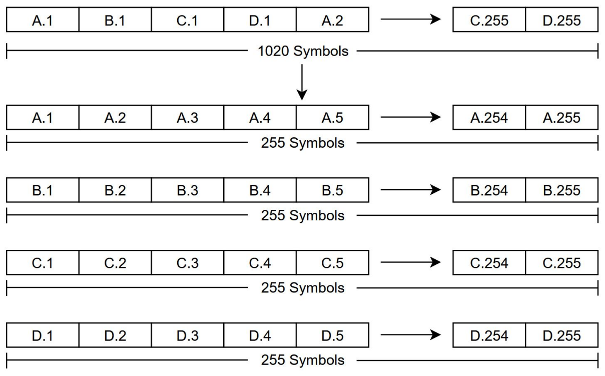

Fig. 4. Process of de-interleaving a CVCDU into Reed-Solomon code words A, B, C, and D, each 255 symbols long.

C. Syndromes

The symbols of the incoming 255 symbol long code word will be treated as the coefficients of the received polynomial

$$r(x) = r_0 + r_1 x + r_2 x^2 + ... + r_{254} x^{254} \tag{7}$$

where r_{254} is the first symbol received and r_0 is the last. The received polynomial is equivalent to

$$r(x) = c(x) + e(x) \tag{8}$$

where $e(x)$ is the error polynomial.

Equation (7) is evaluated at the roots of (3). Since $c(\alpha^z) = 0$ for the $g(x)$ roots α^z, $r(\alpha^z) = e(\alpha^z)$. Evaluating (7) at each zero produces syndromes $S_0, S_2, ...S_{31}$. The syndromes are

$$\begin{aligned} S_0 &= r(\alpha^{212}) = e(\alpha^{212}) = \sum_{i=0}^{254} r_i (\alpha^{212})^i \\ S_1 &= r(\alpha^{223}) = e(\alpha^{223}) = \sum_{i=0}^{254} r_i (\alpha^{223})^i \\ S_2 &= ... \end{aligned} \tag{9}$$

In the syndrome calculation module, the syndromes are evaluated using Horner's rule [7]:

$$\begin{aligned} S_0 &= r_0 + \alpha^{212}(r_1 + ... + \alpha^{212}(r_{253} + \alpha^{212} r_{254})...)) \\ S_1 &= r_0 + \alpha^{223}(r_1 + ... + \alpha^{223}(r_{253} + \alpha^{223} r_{254})...)) \\ S_2 &= ... \end{aligned} \tag{10}$$

This method allows the syndromes to be evaluated as symbols are received. Each syndrome calculation is implemented as an adder and a multiplier. Received symbols are first added to the running sum then multiplied by the respective α^z. There are 32 multipliers for the 32 roots. They have been calculated to be a series of XOR gates for each bit. Fig. 5

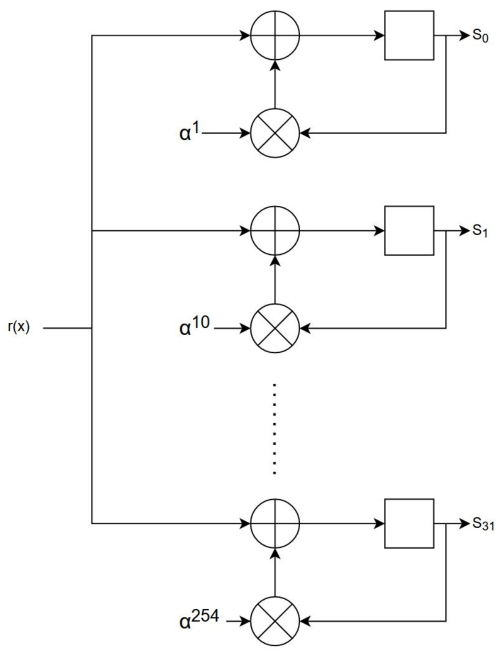

Fig. 5. Block diagram of the syndrome calculation.

The syndromes are the coefficients of the syndrome polynomial

$$S(x) = S_0 + S_1 x + S_2 x^2 + ... + S_{31} x^{31}. \tag{11}$$

If $S(x) = 0$ no error has occurred. If it is non-zero, however, the rest of Reed-Solomon will be performed.

The error polynomial is defined as

$$e(x) = \sum_{i=0}^{254} e_i x^i \tag{12}$$

where $v \leq 16$ coefficients are non-zero, since RS(255, 223) can fix up to 16 errors.

Since each syndrome S_m is equal to $e(x)$. The coefficient e_{i_k} and corresponding power α^{i_k} are equivalent to the magnitude Y_k and location X_k of the kth error. So the syndromes can be rewritten as

$$S_j = \sum_{k=1}^{v} e_{i_k} (\alpha^{11(112+j)})^{i_k} = \sum_{k=1}^{v} Y_k X_k^{11(112+j)} = e(\alpha^{11(112+j)}) \tag{13}$$

For $j \in [0, 32]$ the syndromes are then

$$S_0 = \sum_{k=1}^{v} Y_k X_k^{212}$$
$$S_1 = \sum_{k=1}^{v} Y_k X_k^{223}$$
$$. \tag{14}$$
$$.$$
$$.$$
$$S_{31} = \sum_{k=1}^{v} Y_k X_k^{43}$$

D. Error Location Polynomial

The key equation

$$\Omega(x) = \Lambda(x)S(x)mod x^{32}. \tag{15}$$

describes the relationship between the syndrome polynomial $S(x)$, the error location polynomial $\Lambda(x)$, and the error evaluator polynomial $\Omega(x)$.

The error location polynomial $\Lambda(x)$ is defined by

$$\Lambda(x) = 1 + \sum_{j=1}^{v} \lambda_j x^j \tag{16}$$

where v is the number of errors present in the code word. The roots of the error location polynomial are the inverses of the error locations $X_1, X_2, ..., X_v$. The error location polynomial can therefore be rewritten as

$$\Lambda(x) = (1 - X_1 x)(1 - X_2 x)...(1 - X_v x). \tag{17}$$

The error locations can be found from the roots of $\Lambda(x)$, but the coefficients $\lambda_1, ..., \lambda_v$ must be found first.

To begin it is acknowledged $\Lambda(X_k^{-1}) = 0$. And assuming $n = 11(112 + j)$ such that $j \in [0, 32]$, (16) can be multiplied by $Y_k X_k^{n+v}$ to obtain

$$\begin{aligned}
&Y_k X_k^{n+v} \Lambda(X_k^{-1}) = 0\\
&Y_k X_k^{n+v}(1 + \lambda_1 X_k^{-1} + \lambda_1 X_k^{-2} + ... + \lambda_v X_k^{-v}) = 0\\
&Y_k X_k^{n+v} + \lambda_1 X_k^{-1} Y_k X_k^{n+v} + ... + \lambda_v X_k^{-v} Y_k X_k^{n+v} = 0\\
&Y_k X_k^{n+v} + \lambda_1 Y_k X_k^{n+v-1} + ... + \lambda_v Y_k X_k^{n} = 0
\end{aligned} \tag{18}$$

Sum from $k = 1$ to v:

$$\sum_{k=1}^{v} (Y_k X_k^{n+v} + \lambda_1 Y_k X_k^{n+v-1} + ... + \lambda_v Y_k X_k^{n}) = 0. \tag{19}$$

Separate each term and move the λ outside the summation term:

$$\begin{aligned}
&\sum_{k=1}^{v} Y_k X_k^{n+v} + \sum_{k=1}^{v} \lambda_1 Y_k X_k^{n+v-1} + ... + \sum_{k=1}^{v} \lambda_v Y_k X_k^{n} = 0\\
&\sum_{k=1}^{v} Y_k X_k^{n+v} + \lambda_1 \sum_{k=1}^{v} Y_k X_k^{n+v-1} + ... + \lambda_v \sum_{k=1}^{v} Y_k X_k^{j} = 0
\end{aligned} \tag{20}$$

Each summation term is now equal to a syndrome:

$$S_{j+v} + \lambda_1 S_{j+v-1} + \lambda_2 S_{j+v-2} + ... + \lambda_v S_j = 0 \tag{21}$$

Rearrange:

$$\lambda_v S_j + \lambda_{v+1} S_{j+1} + ... + \lambda_2 S_{j+v-2} + \lambda_1 S_{j+v-1} = -S_{j+v} \tag{22}$$

Expand by replacing j:

$$\begin{aligned}
&\lambda_v S_1 + \lambda_{v+1} S_2 + ... + \lambda_2 S_{v-1} + \lambda_1 S_v = -S_{v+1}\\
&\lambda_v S_2 + \lambda_{v+1} S_3 + ... + \lambda_2 S_v + \lambda_1 S_{v+1} = -S_{v+2}\\
&.\\
&.\\
&.\\
&\lambda_v S_v + \lambda_{v+1} S_{v+1} + ... + \lambda_2 S_{2v-2} + \lambda_1 S_{2v-1} = -S_{2v}
\end{aligned} \tag{23}$$

There are now v equations and v unknowns, so the coefficients of $\Lambda(x)$ can be solved for. The coefficients of $\Lambda(x)$ will be found using the Berlekamp-Massey algorithm. This approach will generate an LFSR whose first 32 outputs are the the syndromes $S_0, S_1, ..., S_{31}$. The taps of the LFSR are the coefficients of $\Lambda(x)$. Fig. 6 shows a finite state machine representing the implemented process of Berlekamp-Massey to find the error location polynomial [9].

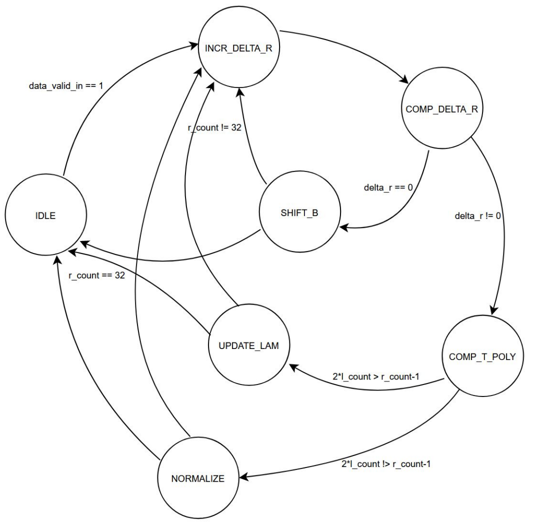

Fig. 6. Finite State Machine of the Berlekamp-Massey algorithm.

E. Error Evaluator Polynomial

The error evaluator polynomial defined as

$$\Omega(x) = \sum_{j=0}^{v-1} \omega_j x^j \tag{24}$$

will be used later to find the error magnitudes. The polynomials relationship with $S(x)$ and $\Lambda(x)$ is defined in (15). The order of each $S_i * \lambda_j$ term is $i + j = n$, so each ω_n term is implemented as a sum of $S_i * \lambda_j$ multiplications for each (i, j) pair that sums to n. The multipliers are implemented as a series of XOR gates similar to the α^z multipliers, but both numbers are unknown here. Each bit of the ω_n term is calculated by XORing certain S_i and λ_j bits.

F. Error Locator

The roots of the error location polynomial $\Lambda(x)$ are found using Chien's search algorithm. The roots of $\Lambda(x)$ are the error locations $X_1^{-1}, X_2^{-1}, ..., X_v^{-1}$. Chien's algorithm evaluates $\Lambda(x)$ at each possible value in GF(2^8). A value α^i in which $\Lambda(\alpha^i) = 0$ is a root and α^{-i} the location of an error [6].

Chien's search is implemented by multiplying each coefficent $\lambda_1, \lambda_2, \lambda_3, ...$ by it's corresponding alpha value $\alpha^1, \alpha^2, \alpha^3 ...$ then adding the products together. On the first cycle, the coefficients are added together which is equivalent to $\Lambda(alpha^0)$. Then each coefficient is multiplied by its corresponding α and the products are added. This is equivalent to $\Lambda(\alpha)$. This is repeated for 254 clock cycles. A counter increments each step and the inverse is calculated when $\Lambda(x) = 0$. The locations of the errors are now found.

G. Error Evaluator

The error evaluator uses Forney's algorithm to calculate the error magnitudes $Y_1, Y_2, ..., Y_v$ at error locations $X_1, X_2, ..., X_v$. Using Forney's algorithm, an error magnitude Y_l at X_l is given by

$$Y_l = -\frac{X_l^{1-112}\Omega(X_l^{-1})}{\Lambda'(X_l^{-1})}. \tag{25}$$

The formal derivative of $\Lambda'(x)$ is given by [5]

$$\Lambda'(x) = \sum_{i=1}^{v} i * \lambda_i x^{i-1} \tag{26}$$

and is derived using $\Lambda(x)$ and a multiplier.

H. Error Corrector

The error corrector will subtract the error polynomial $e(x)$ from the received polynomial $r(x)$ to get the original code word polynomial $c(x)$. It does this by simply subtracting each $e(x)$ coefficient from the respective $r(x)$ coefficient. The original 223 symbol long message word is output without the 32 parity symbols.

I. Re-Interleaver

The re-interleaver takes in the four 223 symbol message words from the four RS instances running in parallel. It then re-interleaves them in the order with which they were de-interleaved.

Without the 32 parity symbols, the CVCDU that was input to the de-interleaver leaves the re-interleaver as a 892 symbol long Virtual Channel Data Unit (VCDU).

VI. Packet Parsing

Receiving in VCDUs we parse them into Multi Packet Data Unit (MPDU) of 14 Minimum Code Units (MCUs) where each packet represents a portion of each line of the image. Since the physical VCDU packets are constant in length but the information they contain varies, we check for the pointer to the first header of the first MPDU in each VCDU. Identifying the offset of each allows us to find the header and read in the data bits to our MPDU we send to the next module.

Specifically, our defragmentation algorithm works by reading in bits using a simple FSM to isolate complete MCUs. First, it identifies the location of the next VCDU header as the end of our current packet. Then, it reads the data from each MPDU until it arrives at an MPDU who's data exceeds the end of the VCDU. In this case, we must save the start of the MPDU and combine it with the remaining information in the next VCDU.

VII. Image Display

Each MCU is an 8x8 image compressed using the JPEG algorithm which we must decompress, then display as RGB. This is accomplished by first Huffman decoding using standardized Huffman lookup tables to get our AC and DC values. With these, we are able to de-zigzag then dequantize our matrix storing the image data. From there the inverse discrete cosine transform is applied to the entire 8x8 and we receive yCrCb values for each individual pixel. Finally, we convert this to RGB values and display.

VIII. Python FPGA Serial Communicator

In order to verify each step in the process we are using a Python script to send and receive data in communication with our FPGA. To access this data we we will be using serial communication via UART over USB (In a similar fashion to Manta). Since our FPGA is running at a 100MHz clock cycle we use a maximum UART baud rate of 115200. Still, this is much slower than our FPGA clock rate and not easily generated modifying by a factor of 2 we will use an approximate ratio very close to our desired 100000000/115200 = 868.05555, explicitly, $2^{17}/151 = 868.02649$. With this ratio, we can use an accumulator to generate our baud ticks within an acceptable error range and then sample 16 times per tick. This breaks down into two separate modules: receiver and transmitter.

For the receiver, we will use a state machine that begins in an idle state listening for the falling edge of the rx bit to signal a start bit has been received. From there it will wait for 8 sample ticks to position in the middle of the signal before double checking the signal is still low and we received a valid start bit. It then moves to the data state where it will sample every 16 sample ticks and store the sampled value in our 8 bit register until all 8 data bits have been received. Upon receiving the 8th bit the FSM moves to the stop state where we sample once more to verify the signal is high before returning to the idle state. If the sample is not high we have received invalid data and return nothing. Otherwise, the data has been correctly received and we signal the higher level module via a one-cycle burst of the data valid signal and store our received byte in the data out signal.

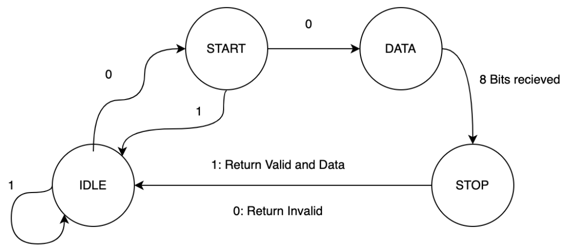

Fig. 7. UART Receiver FSM

For the transmitter, it works nearly the same in reverse fashion. We start at an idle state waiting for a signal to begin the transmission of data stored in the input register. Once a signal to transmit is received, the transmitter sends a low signal for one baud tick before sending a bit of data per baud tick. Finally, once all eight bits have been sent the transmitter sends the one baud tick of high signal representing the stop bit before returning to the idle state and possibly beginning another byte transmission.

To achieve this UART serial communication on the computer side we are using the pyserial library to read and write data over our connection. Since we are only working with higher latency modules a basic ability to read and write one at a time to the FPGA is sufficient. This is simplified by having the python end constantly listening for the data transfer or constantly sending the data transfer. UART communication uses a start bit of idle high state to a low state immediately preceding the data transfer that triggers the beginning of the data read at each end.

For our project, we chose to use a simple 10-bit UART protocol with 1 start bit, 8 data bits, and 1 stop bit. In testing, this achieved no discernible errors that would necessitate a parity bit or additional stop bits. This is likely due to our higher sample rate of 16 times per baud that led to accurate results.

IX. References

The source repository can be found at: ”https://github.com/theory789/lrpt_decoder”. Our modifications of an existing LRPT implmentation in C: ”https://github.com/theory789/meteor_decode”

References

[1] Coordination Group for Meteorlogical Satellites, “LRPT/AHRPT Global Specification”, CGMS doc. no. 04, October 1998
[2] JM Cioffi, ”Fundamentals of Synchronization”, https://cioffi-group.stanford.edu/doc/book/chap6.pdf
[3] https://github.com/dbdexter-dev/meteor_demod
[4] CCSDS: “Telemetry Channel Coding”, CCSDS recommendation 101.0-B-3, May 1992
[5] https://en.wikipedia.org/wiki/Forney_algorithm
[6] https://en.wikipedia.org/wiki/Chien_search
[7] https://en.wikipedia.org/wiki/Horners´_method
[8] https://en.wikipedia.org/wiki/ReedSolomon_error_correction
[9] https://en.wikipedia.org
[10] https://scholarworks.rit.edu/cgi/viewcontent.cgi?article=8953&context=theses /wiki/BerlekampMassey_algorithm
[11] R. Cypher and C. B. Shung, ”Generalized trace back techniques for survivor memory management in the Viterbi algorithm,” [Proceedings] GLOBECOM ’90: IEEE Global Telecommunications Conference and Exhibition, San Diego, CA, USA, 1990, pp. 1318-1322 vol.2, doi: 10.1109/GLOCOM.1990.116708.
[12] HUI-LING LOU, ”Implementing the Viterbi Algorithm”, SEPTEMBER 1995, IEEE SIGNAL PROCESSING MAGAZINE, pp.42-51
[13] Andrew Weinfeld MiG Generator