Report 5
Final Report
17th May 2007

Abhinav Agarwal
Grant Elliott
S. R. K. Branavan

The objective of this project is to design & implement a Reed-Solomon decoder for GF(256) with the primitive polynomial given as a parameter, supporting a minimum data rate sufficient for integration into a IEEE 802.16 receiver.

IEEE 802.16 uses a Reed-Solomon code over a Galois Field of 256 [GF(256)]. The standard also specifies the primitive (field-generator) polynomial for the Galois Field as $x^8 + x^4 + x^3 + x^2 + 1$. The highest data rate given in the standard is 134.4 Mbps.

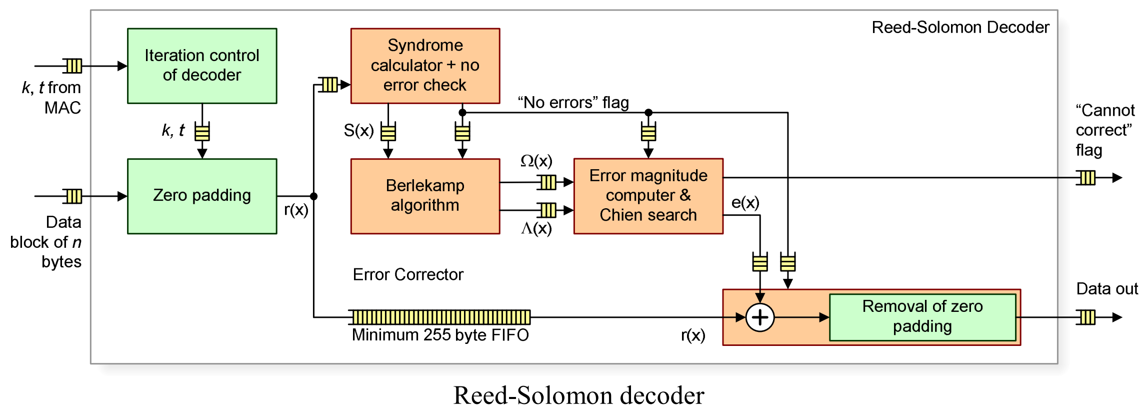
1. The Decoding Process

The Reed-Solomon decoder goes through a set of 4 main steps in decoding the message. These are:

1. Calculate the *syndrome polynomial*.
2. Compute the *error locator polynomial* and the *error evaluator polynomial* from the *syndrome*.
3. Find the *error locations* & *error values* from the locator & evaluator polynomials.
4. Use the error locations & values to correct the received message.

2. Decoder block diagram

The diagram below shows the high level architecture of the Reed-Solomon decoder. It also indicates communication between modules using FIFOs. The blocks highlighted in orange are the primary modules of the algorithm, and are thus also the most complex. The primary modules receive *t*, and the error corrector received *k* from the *iteration control module* through FIFO decoupled paths. This detail is not shown in the diagram below for clarity.



Reed-Solomon decoder

Zero padding / removal handles shortened code requirement of the 802.16 protocol. This requirement states that the incoming data block can have the following parameter values:

k : 6 – 255
t : 0 – 16

where

n	Number of overall bytes after encoding
k	Number of data bytes before encoding
$2t = n - k$	Number of parity bytes

The values of *k* and *t* are given to the decoder by the *Medium Access Control* (MAC) layer for each *burst profile*. A burst profile can consist of multiple data blocks. The *Iteration control* block maintains the *k* and

t values received from the MAC and passes it on to the Syndrome calculator for each received data block. These values are then passed on from module to module sequentially, in parallel to the data. This approach has been taken since it will potentially simplify control & physical layout.

3. Design Constraints

To be of any use, the Reed-Solomon decoder needs to support the sustained throughput requirements of the protocol in which it is used. This introduces minimum data-rate constraints on each of the primary modules within the decoder. The table below lists the *input & output rates* for a decoder throughput of B bytes per second. The table also lists *iterations / second*, *clock rate*, and *number of sequential arithmetic/shift operations* based on our current micro-architectural design.

Blocks	Input rate	Output rate	Iterations / s	Clock rate	Seq ops / s
Syndrome calculator	B	B/8	32B	32B	64B
Berlekamp algorithm	B/8	B/16	B/8	2.5B	38/8 B
Error magnitude calculation & Chien search	B/16	B	1.5B	1.5B	3B

4. Design Considerations

Reed-Solomon decoding is a computationally intensive process involving Galois field arithmetic. It also operates on fairly large blocks of data. This introduces speed / area tradeoff issues – particularly in the primary modules & the Galois field multiplier circuits, of which eight instances occur in our architecture. The multiplier and the primary modules have various potential degrees of folding. There is thus considerable room for architectural exploration.

5. High-level Design

In this section we discuss only the design of the primary modules of the Reed-Solomon decoder - those that implement the actual Reed-Solomon decoding algorithm.

5. 1 Notation

The IEEE 802.16 standard defines the Reed-Solomon codec to operate on the Galois field of 2^8 ($GF(2^8)$). $GF(2^8)$ has 256 unique elements, and therefore a byte can represent each unique element.

The elements of the Galois field $GF(2^8)$ are the ordered set of 256 symbols $\{0, \alpha^0, \alpha^1, \alpha^2, \ldots, \alpha^{254}\}$ where the exponentiation is done using Galois field arithmetic defined by the primitive polynomial of field. Here α is the root of the primitive polynomial, and in this case is 2.

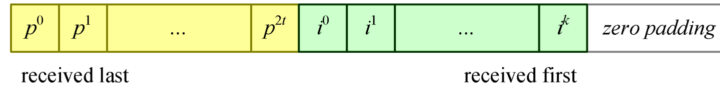
In the following discussion, a fixed length stream of n symbols is represented as a polynomial of degree n in a variable x where coefficients correspond to the symbols. Thus a message block of 255 bytes would be represented as the following polynomial:

$$R(x) = r_{254}x^{254} + r_{253}x^{253} + \ldots + r_i x^i + \ldots + r_0$$

where r_i is the ith byte of the message block.

It is assumed that the received block is sequenced in a manner such that the information bytes are received before the parity bytes, and that if zero padding was necessary in the calculations of the the parity bytes,

they were inserted before the information bytes. This assumption is shown below with the information & parity bytes in polynomial form.



5.2 Galois Field Arithmetic.

The rules for addition & multiplication in a Galois field are obtained by adding & multiplying in the usual manner, and then reducing the result modulo the primitive polynomial of the field.
i.e. If p is the primitive polynomial, a and b are elements of the Galois field, addition $\oplus$, and multiplication $\otimes$ are defined as :

$$a \oplus b = (a + b) \text{ modulo } p$$
$$a \otimes b = (a \times b) \text{ modulo } p$$

For GF(2^8), addition translates into a straightforward bitwise XOR operation on bytes a, b:

$$a \oplus b = (a \text{ xor } b)$$

Multiplication on the other hand requires the modulo operation to be performed, and is thus more complex.

For example:

For simplicity, consider a Galois field of GF(2^4) with the primitive polynomial $p(x) = x^4 + x + 1$.
If $a = (1 + \alpha + \alpha^3)$ and $b = (\alpha + \alpha^2)$,

$$\begin{aligned} a \otimes b &= (1 + \alpha + \alpha^3) \times (\alpha + \alpha^2) \\ &= \alpha + \alpha^3 + \alpha^4 + \alpha^5 \end{aligned}$$

Since α is a root of $p(x)$,

$$\alpha^4 + \alpha + 1 = 0$$
$$\Rightarrow \quad \alpha^4 = \alpha + 1$$
$$\text{and} \quad \alpha^5 = \alpha^2 + \alpha$$

Substituting these values for α^4 and α^5 gives us:

$$\begin{aligned} a \otimes b &= \alpha + \alpha^3 \oplus (\alpha + 1) \oplus (\alpha^2 + \alpha) \\ &= 1 + (\alpha \oplus \alpha \oplus \alpha) + \alpha^2 + \alpha^3 \\ &= 1 + \alpha + \alpha^2 + \alpha^3 \end{aligned}$$

As can be seen in this example, the first step of polynomial multiplication is simply a matter of shifting (multiplication by powers of α) & adding (which itself is the XOR operation):

$$\begin{aligned} (1 + \alpha + \alpha^3)(\alpha + \alpha^2) &= (1 + \alpha + \alpha^3)\alpha \oplus (1 + \alpha + \alpha^3)\alpha^2 \\ &= (\alpha + \alpha^2 + \alpha^4) \oplus (\alpha^2 + \alpha^3 + \alpha^5) \end{aligned}$$

The second step is calculating the result of step 1 modulo the primitive polynomial. A GF of order 2^n can have symbols of at most 2^{n-1}. Thus any terms produced by step 1 with exponents greater than $n - 1$ will need to be reduced back into the Galois field. Thus, the reduction of any higher degree term can be done as follows:

$$\alpha^m = (p(x) - \alpha^n)\ \alpha^{m-n}$$

GF Multiplication pseudocode:

p[8 :0] – primitive polynomials
a[7:0], b[7:0] – values being multiplied.

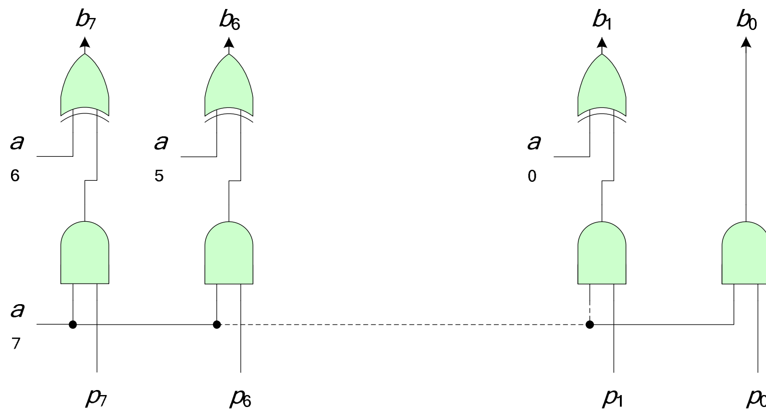
```
for i = 0:7
   for j = 0:7
      result [i+j] ^= a[j] & b[i]

for i = 15:8
   if result [i] == 1
      result [15:0] ^= (p[7:0] << (i - 8))

return result [7:0]
```

As is obvious, this directly lends itself to a BlueSpec implementation.

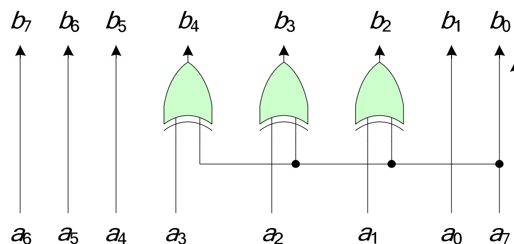
Multiplication by a root of the primitive polynomial is much simpler. Consider a general polynomial a

$$
\begin{aligned}
A &= \Sigma\, a_i\, \alpha^I \\
\alpha A &= \Sigma\, a_i\, \alpha^I \bmod p \\
&= \Sigma\, a_i\, \alpha^{i+1} \bmod p \\
&= (\Sigma\, a_{i-1}\, \alpha^i) - a_7 p \\
&= \Sigma\, (a_{i-1}\, \alpha^i \oplus a_7 p_i)
\end{aligned}
$$

This solution suggests a combinational circuit with 8 AND gates and 7 XOR gates. If the primitive polynomial is a constant parameter, this circuit reduces exclusively to as many XOR gates as there are non-zero coefficients in the primitive polynomial, neglecting the highest and lowest order terms (which, in fact, have unity coefficients for all primitive polynomials). For the GF(256) primitive polynomial used in 802.16, multiplication by a root of the primitive polynomial requires only three XOR gates.



Combinational circuit for multiplication by α for an arbitrary primitive polynomial.



Combinational circuit for multiplication by α for the primitive polynomial $x^8 + x^4 + x^3 + x^2 + 1$

A constant power of alpha may be enumerated by combining these circuits, yielding only a constant value after optimization (assuming again that *p* is a constant parameter). Multiplication by a power of alpha may be computed iteratively, requiring only a few XOR gates instead of a full multiplier. Using these techniques, we may eliminate the vast majority of general purpose multipliers that would otherwise have been necessary.

Finally, division consists of multiplication by an inverse. Provided the polynomial defining the field is primitive, a unique multiplicative inverse *b* satisfying *ab=1 mod p* exists for all *a*. Constructing it for a general primitive polynomial is non-trivial, however. Following considerable math, it is possible to express each bit of the inverse as a 7x7 determinant resulting from eliminating a column in a 7x8 matrix, itself determined with 70 XORs and 56 ANDs. Unfortunately, finding the eight determinants requires an extreme amount of logic. Instead, we simply build a 256 element lookup table for the specific polynomial at compile time and continue to investigate optimized combinational circuits for calculating inverse.

5. 3 Syndrome Calculator

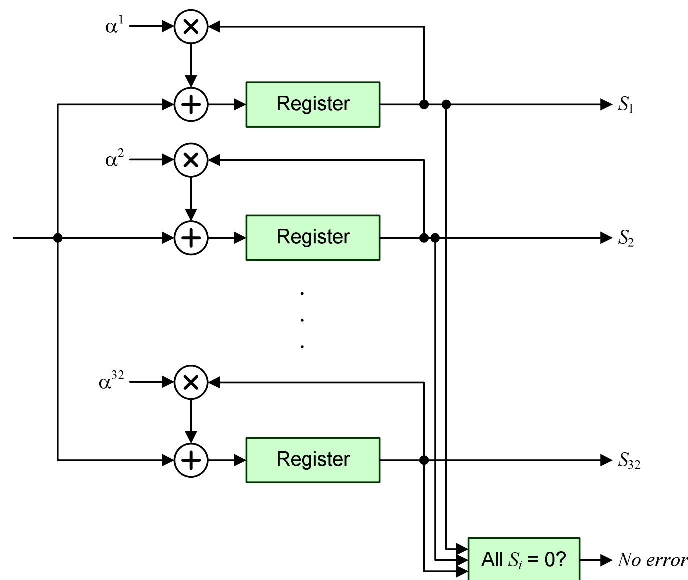
The *syndrome* is a series of *2t* bytes, which contain all the information that can possibly be extracted from the received message about any errors that may have been introduced into it. This set of *2t* bytes is then used by the rest of the Reed-Solomon algorithm to find & correct the errors if possible.

The syndromes, S_j, are calculated by evaluating $R(x)$, the polynomial representation of the received codeword, at powers of α up to $2t$:

$$S_j = R(\alpha^j) = r_{n-1}\alpha^{(n-1)j} + r_{n-2}\alpha^{(n-2)j} + \ldots + r_0 \qquad \forall\, j \in \{1, 2, \ldots, 2t\}$$

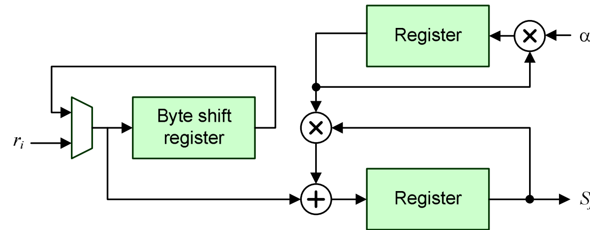
For an uncorrupted codeword, all syndromes will be zero. Calculation of syndromes thus serves as a test for corruption.

The following hardware computes 32 syndromes, the first *2t* of which are meaningful. Powers of α are calculated using the multiplication by α hardware and optimize to mere constants if the primitive polynomial is a constant.



Parallel implementation of Syndrome computation.

This operation may be serialized. However, since the received bytes r_i are needed for the calculation of each S_j, they must be stored in a re-circulation buffer until all S_j's are computed. This buffer occupies more space than the 31 parallel syndrome calculators which were removed. As such, the serial implementation is actually larger as well as slower.

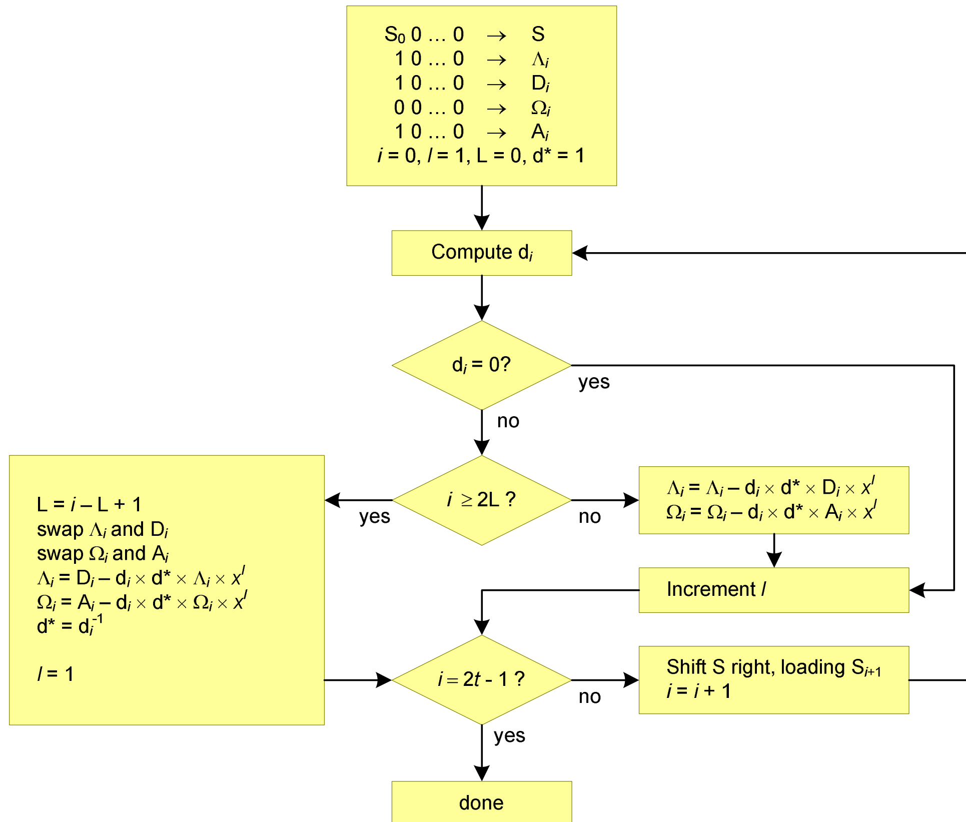


Serial implementation of Syndrome computation.

5. 4 Berlekamp Algorithm

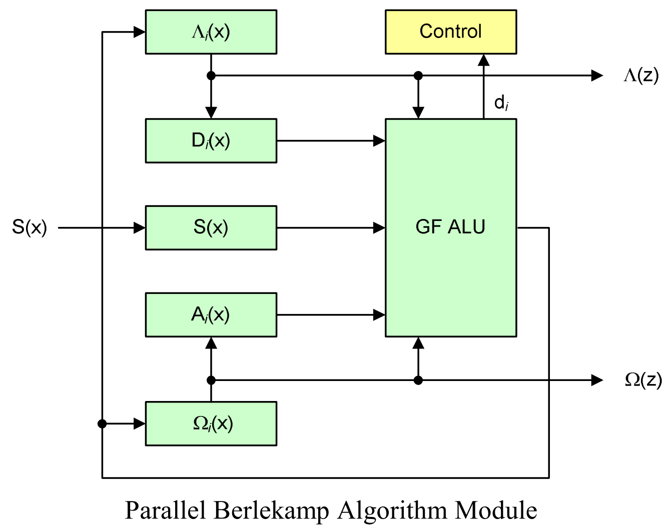
This module uses the syndrome to compute the *error locator polynomial* $\Lambda(x)$ and the *error evaluator polynomial* $\Omega(x)$ through the Berlekamp algorithm. These polynomials are later used to compute the actual errors and their locations. Effectively, the Berlekamp algorithm efficiently solves t simultaneous equations. Its existence makes Reed-Solomon decoding tractable.

The module's block diagram and its iterative flow chart are shown below.



The control logic flow for the Berlekamp module

Here, d_i is result of the convolution of the current Λ polynomial with the first i symbols of the Syndrome. As can be seen from the flow chart above, the Λ and Ω polynomials are computed iteratively.



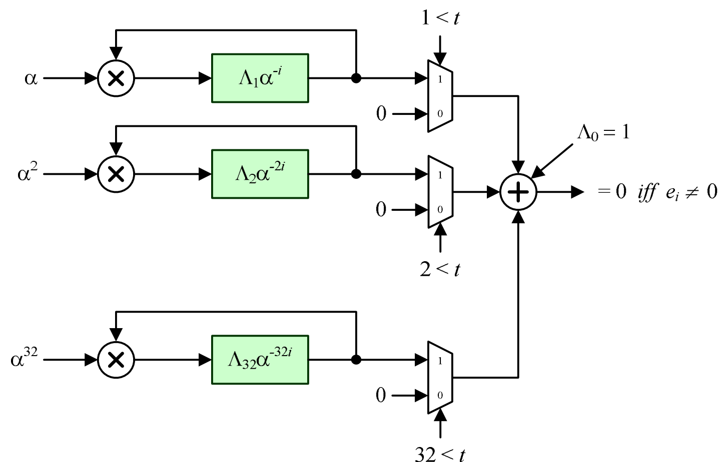
Parallel Berlekamp Algorithm Module

5.5 Chien Search

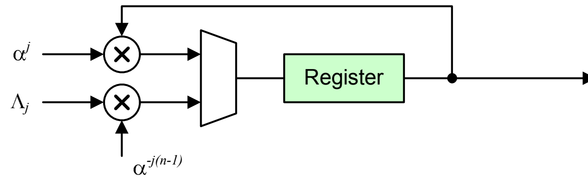
The error locations are given by the inverse roots of the error locator polynomial. The Chien Search algorithm finds these roots by performing an exhaustive search over the Galois field. An error has occurred in symbol i of the received data if and only if $\Lambda(\alpha^{-i}) = 0$.

i.e. $$\sum_{k=0}^{t} \Lambda_k(\alpha^{-ik}) = 0$$

A hardware implementation of an iterative version of this algorithm is shown below. In the above circuit, initially i is set to $n - 1$ (i.e. 254). Over subsequent iterations, the multiplications by α iterate i down to 0, resulting in all the potential error locations e_i being checked. In the parallel implementation shown in the diagram, all 32 terms of $\Lambda(\alpha^{-i})$ are always computed, but only the first t terms are added together to check if the location i has an error.



Each "row" in the diagram above is a circuit of the form shown below, and the multiplexer loads the register with $\Lambda_j\alpha^{-j(n-1)}$ for the initial cycle. It can be shown that $\alpha^{-j(n-1)} = \alpha^{j}$, and therefore, this value is actually used to initially load the register. Successive powers of α are calculated iteratively, requiring only a few XOR gates.



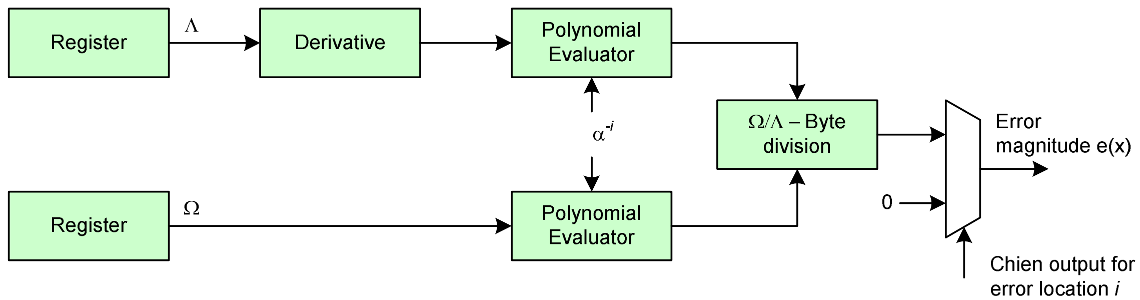
5.6 Error Magnitude computation

The error magnitude at a location i which has been identified as being in error is given by:

$$e_i = \frac{\Omega(\alpha^{-i})}{\Lambda'(\alpha^{-i})}$$

Where the derivative Λ' is given by $\Lambda'(x) = \Lambda_1 + \Lambda_3 x^2 + \Lambda_5 x^4 + \ldots$, which requires no logic - only dropping the even terms and shifting the odd terms by one position.

The diagram below shows the hardware implementation of this module.



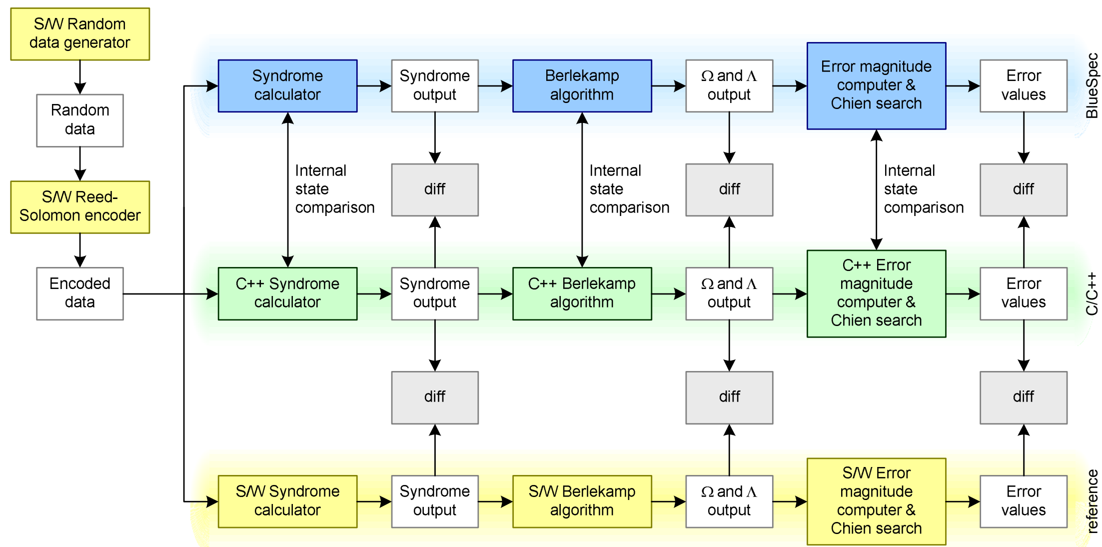
As an optimization on the circuit above, the actual implementation computes the error magnitude values only for the locations containing errors. If the Chien error location search does not find any locations to be in error, this indicates that the data block had more error than could be corrected. i.e. there were more than t errors in a data block encoded with FEC information capable of correcting up to t errors. This therefore is used to generate the “cannot correct” flag. This flag is generated $2t$ cycles after all the information bytes in the block have been output by the decoder. The delay is due to the 2t parity bytes that the decoder needs to processes. The “cannot correct” flag is streamed out through a FIFO of its own.

6. Design Verification / Testing

6.1 Functional testing

Testing will be done by comparing the outputs of the BlueSpec modules against the outputs of sample software implementations of the Reed-Solomon decoder from external sources. To make debugging & verification easier, we also code our design of the Reed-Solomon algorithm in C/C++ and verify its output against reference sample code output. This allows us to confirm the correct operation of the design prior to implementing it in BlueSpec. Since this software implementation is a value-accurate reference of the design, it allows us to debug & verify any future version of the BlueSpec implementation.

The diagram below shows how the sample implementations & our C++ implementations are used to test the BlueSpec modules. All BlueSpec modules are shown in blue, our C++ implementations in green, and external source modules in yellow.



Test Framework Overview (BlueSpec modules highlighted in blue)

Top-level testing of the final design was done using a reference Reed-Solomon decoder implementation *rsdec* in the Communications toolbox of MATLAB 7.3. We first generated large sequences of encoded messages, each with variable number of information bytes *k*, parity bytes *2t*, and total length *n*. These sequences were then, randomly corrupted and were used as inputs for a test-file of the MATLAB implementation and the bluesim executable testbench of our hardware design. The outputs of both decoders were then compared to identify differences. The final design functionally matches exactly with the MATLAB reference.

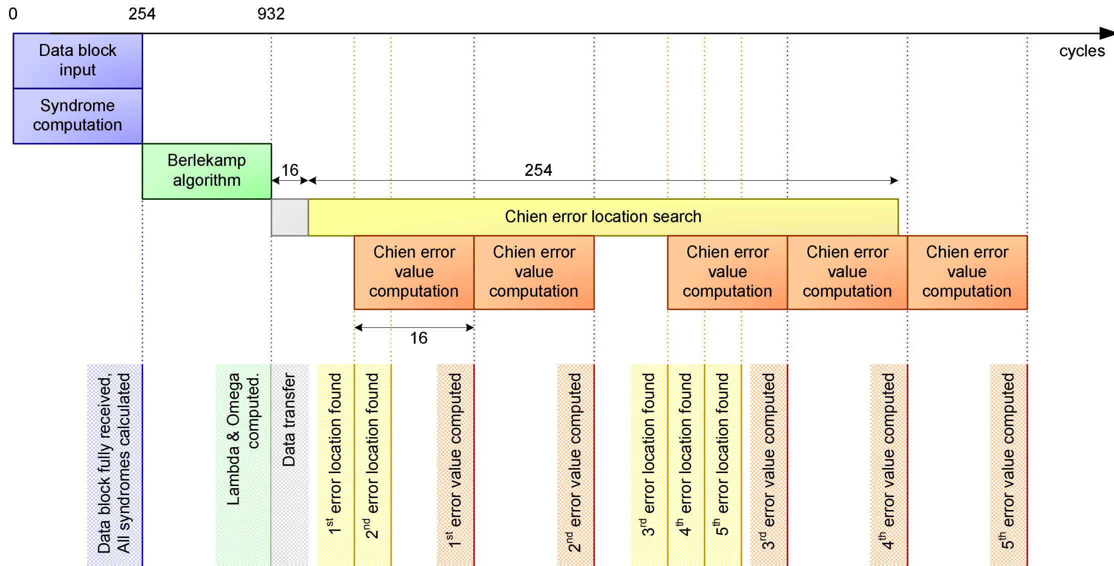
To ensure the proper function of the ReedSolomon decoder, an automated test framework was setup. This framework does the following:

1. Generate a pre-specified number of messages each containing multiple blocks of data
2. Encode the messages using a reference Reed Solomon codec.
3. Corrupt the messages with a number of errors drawn from a uniform distribution between 1 and 16.
4. Decode the corrupted message using the BlueSpec Reed Solomon decoder.
5. Verify correct operation by comparing the decoded messages with the uncorrupted originals.

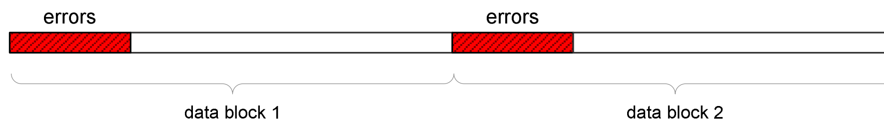
Using this setup, the decoder was tested on test sets of 100 messages each with 100 data blocks over multiple test cycles.

6.2 Performance testing

The diagram below shows the process of Reed-Solomon decoding along the *clock cycle* axis.



As can be seen, the end-to-end cycle time depends heavily on the distribution of the errors in the data block due to the time taken to compute the error magnitudes. Understandably, the worst case throughput occurs when a large number of correctable errors occur as a contiguous block – in particular when the first16 bytes of a data block are in error. Note that 16 is the largest number of errors that can be corrected by the Reed-Solomon decoder as implemented here. Under the current decoding method, if the number of errors are more than what can be corrected, the throughput is not affected as the error magnitudes are not calculated for any of the bytes in that block.



Worst case sequence of errors

The above error pattern results in a cycle time per block of 905 at steady state (i.e. without startup times / shutdown times) for the current implementation.

The current implementation has been tested with combinations of *blocks without errors*, *with correctable errors*, and *with non-correctable errors*, and has been found to operate correctly.

7. Design Exploration

Buffer Sizing

The design used here is essentially pipelined in that the Syndrome Calculator, Berlekamp Algorithm and Chien Error Calculator can operate on 3 different data blocks at a given time. Therefore, the FIFO buffer feeding the Error Corrector needs to be able to maintain multiple data blocks for the design to be able to operate at peek through-put.

However, it's not simply a matter of sizing the buffer at 3 times the maximum block size. The Syndrome Calculator starts calculating from the time it receives the first byte in the data block, and the Chien Error Calculator puts out error values for the received bytes in their order of arrival. This means that if we get three blocks B0, B1 & B2, with B0 arriving first, as B2 is being received, its syndrome is calculated. At the same time, the Berlekamp Algorithm is operating on B1, and the Chien Error Calculator is putting out error values of B0. As error values are computed, B0 is corrected, and streamed out.

t = 16 (32 parity bytes)

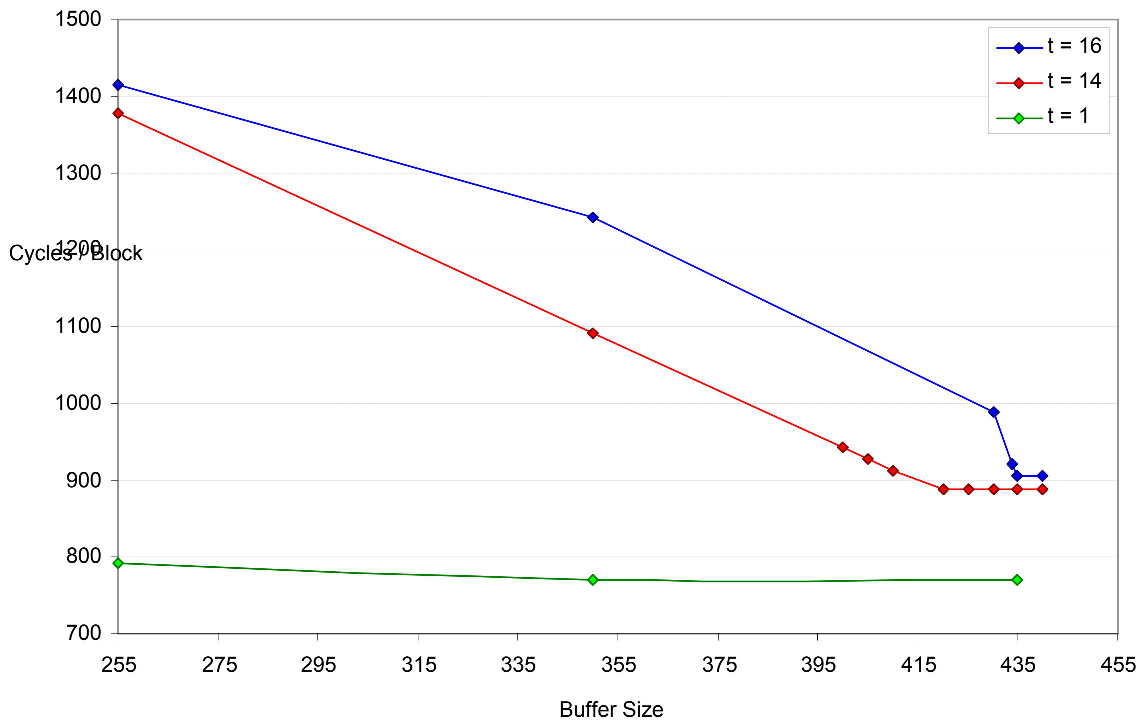
Buffer Size	Cycle Count
255	1414
350	1242
430	988
434	920
435	**905**
1020	905

t = 14 (28 parity bytes)

Buffer Size	Cycle Count
255	1377
350	1092
400	942
410	912
420	**887**
425	887

t = 1 (2 parity bytes)

Buffer Size	Cycle Count
255	792
300	**770**
350	770
435	770



Therefore, the buffer needs to be able to hold at least 1 block and at most 3. The optimal size is a little less than twice the number of data bytes in a block as can be seen below. Note that the number of data blocks *d*,

is 255 - 2t where t is the number of correctable errors. The tables below show the cycles per block at steady state through-put for the case where each block has t errors (the maximum correctable) at the beginning. This is the worst case scenario for the sizing of this buffer as the calculation of the error values takes (t + 1) cycles per error – therefore the rest of the data block remains in the buffer until the t errors can be corrected, which takes t (t + 1) cycles. I.e. for t = 16, if all the 16 errors are at the beginning of the block, at least (255 – 16) bytes will remain in the buffer for 272 cycles.

The area, clock timing and power at the optimal through-put and minimum area cases are given below. These figures are for the fastest possible clock timing. Also note that the power figures are from the report produced by dc-synth as we were not able to get the proper power estimation tool-flow working. The Cycles / Block gives the number of clock cycles needed to decode a single block in the worst case.

Buffer Size	Area (μm^2)	Power (mW)	Clock Timing (ns)	Cycles / Block
255	846089.7	214.76	4.849	1414
435	1030593.1	272.31	4.728	905

8. Synthesis

For 134 Mbps rate, block throughput required is 905 cycles * 14 ns. *Given the current clock timing, a data rate of 396.8 Mbps can be achieved.*

For the maximum 802.16 Reed-Solomon data rate of 29.1 Mbps referred to in Ng et al[5], a clock speed of 64ns (15.6 MHz) is sufficient. At this clock speed our design achieves a power estimate of 13.94 mW. This compares very favorably with the 21.028 mW quoted in Ng et al for their Verterbi decoder at the same throughput.

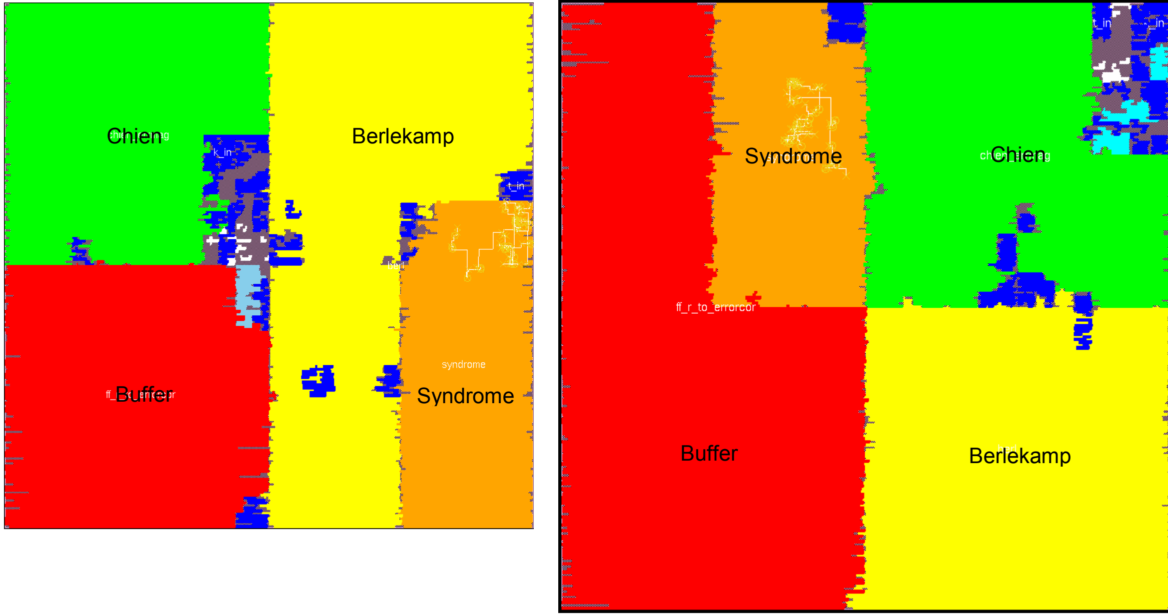
As shown in the table below, the throughput achieved by our optimal design exceeds all requirements quoted in literature. The average case below is computed using randomly generated messages with the number of errors drawn from a uniform distribution between 1 and 16. As such it is a conservative estimate, and the actual average case is expected to be higher given the low processing time associated with error-free packets.

Highest throughput quoted in Ng et al	29.1 Mbps
Highest throughput quoted in 802.16 spec (IEEE 802.16-2004)	134 Mbps
Maximum empirical throughput achieved by optimal design for the worst case burst error.	**393 Mbps**
Maximum empirical throughput achieved by optimal design for the average case.	**550 Mbps**

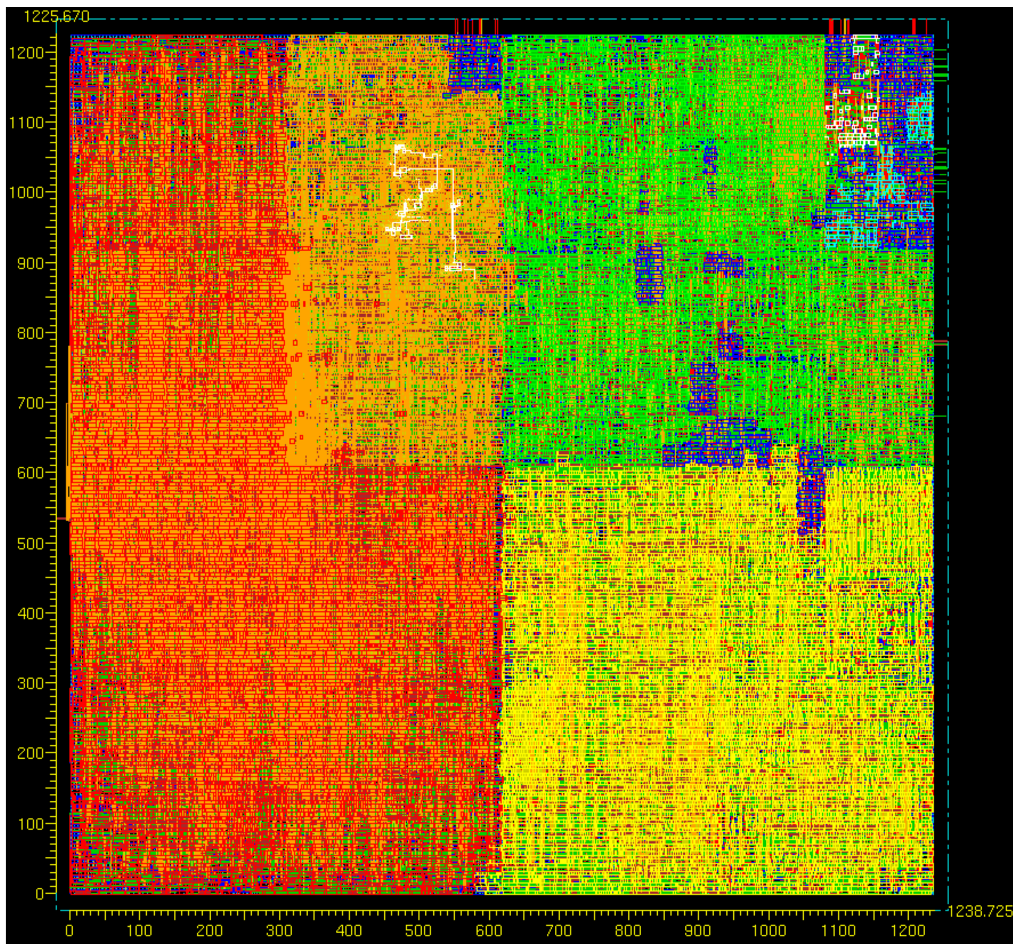
The screenshot below shows the post-route layout with the following highlighted:

- Red – Reed-Solomon main buffer.
- Orange – Syndrome Calculator.
- Yellow – Berlekamp Module.
- Green – Chien Error Calculator.
- Cyan – Error Corrector.
- Blue – Input & Inter-module FIFOs
- White – Miscellaneous arithmetic logic.

The critical path is shown in white.



Comparison of die sizes. The layout on the left uses a buffer size of 255,
and the one on the right uses the optimal buffer size of 435.



A view of the fast throughput design's layout showing die scale.

Module	Area (μm^2)	Percentage of die
Syndrome	140474	13.6
Berlekamp	242773	23.6
Chien	175672	17.0
Error Corrector	5246	0.5
Buffer	417586	40.5
Total	**1030593**	**100**

A breakdown of die area by module for the optimal design.

The critical path with both of the buffer sizes lies in the Syndrome calculator where the *no error* check is performed. This check involves the following operations:

1. Check that all the message bytes have been received.
2. Check that each of the first 2*t* syndrome bytes is 0. *t* is a dynamic value received from the Iteration Control module.

A folded AND operation is done to combine the zero checks of the 2*t* syndromes into the output value.

Note that all of the layout diagrams shown here have been synthesized with synthesize boundaries around the modules in the Reed Solomon design. However, with the design fully parameterized (where the primitive polynomial is passed to the mkReedSolomon module as a static elaboration parameter) BlueSpec is unable to complete the compilation if the synthesize boundaries are present. On the other hand, having synthesize boundaries results in better layout and significantly better critical path times. It is therefore suggested that unless the compilation issue can be resolved, the mkReedSolomon module be used with the primitive polynomial declared as a global variable rather than as a static parameter.

9. Implementation

The Reed-Solomon decoder has been implemented & tested with multiple contiguous data blocks of varying information byte lengths & parity byte lengths. This involves the following modules:

1. Galois field addition & multiplication.
2. Syndrome Calculator.
3. Berlekamp algorithm
4. Chien search
5. Error magnitude computation
6. No error & too many errors check
7. Error correction
8. Zero padding

In addition:

1. Design exploration has been carried out on the Syndrome calculator, Chien Error Corrector, and the main buffer in the Reed-Solomon module. The current implementation uses the optimal designs from this exploration.
2. Worst case cycle counts per block have been estimated & empirically confirmed.
3. The design has been placed & routed to obtain clock rates.
4. The design has been tested with 10,000+ test messages using an automated test framework.

Because we were unable to implement a general case Galois field divider in BlueSpec, a lookup table approach is used. To allow for the parameterization of the ReedSolomon module, a preprocessor was written in C++ to automatically generate the BlueSpec code lookup table. This preprocessor parses the BlueSpec code in which mkReedSolomon is instantiated.

10. Contibutions

- Reed Solomon decoder parameterized by the primitive polynomial.
- The decoder meets the functional specification for 802.16, but easily adaptable to other applications, such as CD.
- 802.16 throughput requirements are exceeded by a factor of 10.
- Decoder designed for integration with the OFDM framework.
- Developed generalized GF arithmetic functions for Bluespec.

11. References

1. Error-Correction Coding for Digital Communications. George C. Clark, Jr & J. Bibb Cain.
2. Error Correction Coding. Mathematical Methods and Algorithms. Todd K. Moon.
3. From WiFi to WiMAX: Techniques for IP Reuse across Different OFDM Protocols. Man C. Ng, Murali Vijayaraghavan, Gopal Raghavan, Nirav Dave, Jamey Hicks, Arvind.
4. http://www.eccpage.com (Non-commercial Reed-Solomon codec implementation by Simon Rockliff, University of Adelaide)
5. IEEE Std 802.16-2004 and IEEE Std 802.16e-2005 http://standards.ieee.org/getieee802/donwload/802.16-2004.pdf
6. Reed Solomon Decoder documentation, Communications Toolbox, MATLAB 7.3