Design of a Synthesisable Reed-Solomon ECC Core

David Banks
Publishing Systems and Solutions Laboratory
HP Laboratories Bristol
HPL-2001-124

E-mail: dmb@hplb.hpl.hp.com

Reed-Solomon, error correction, verilog, synthesis, synthesisable, hardware, ECC, Galois field

In this report we describe the design of a Reed-Solomon error correction core that supports errors and erasures decoding. In a second report HPL-2001-125 we describe the verification of this core.

The core consists of separate encoder and decoder blocks that can be operated independently, each with symbol wide data paths. These blocks have sufficient throughput to handle back-to-back codewords, and the overall latency is typically less than two codewords.

The design is expressed in the Verilog hardware description language (Verilog HDL), and is fully synthesisable. The design supports a wide range of different Reed-Solomon codes, with the choice of a particular code being made at synthesis time. This approach has a number of advantages that aid shorter product design cycles, by allowing the changes in the choice of code and target technology to be made late in the design cycle. Because of its flexibility, the design could be reused across a wide range of products with differing coding requirements.

A sample design configured for a RS(160, 128,T=16) code in GF(2^8) was targeted to the Agere HL200CDE 0.20um standard cell library. This resulted in a gate count of 72K gates, an encoder latency of 2 cycles and a decoder latency of 305 cycles. The design could be clocked at 70MHz.

Approved for External Publication

1 Introduction

This document describes the designof the Reed-Solomon ECC block designed by HP Labs Bristol. For further details on the verification of the, please refer to [1]

A Reed Solomon code of the form RS(B, B - 2T, T) has the following properties:

- Codewords are blocks of B symbols, where B – 2T of these symbols are information symbols, and the remaining 2T are check symbols.
- Symbols are sequences of W bits.
- It can correct up to T symbol errors; an error is a corruption whose location and magnitude are unknowns.
- It can correct up to 2T symbol erasures; an erasure is a corruption whose location is known, but whose magnitude is unknown.
- It can correct any combination of E symbol errors and J symbol erasures, as long as $2E + J \leq 2T$.
- If $B = 2^W-1$, the code is said to be a full-length code.
- If $B < 2^W-1$, the code is said to be a shortened code.

Reed-Solomon codes operate in Galois fields; for an introduction to Galois field arithmetic, see[2].

Codewords are treated as polynomials by using the codeword symbol values as the coefficients of the polynomial. The standard convention in Reed-Solomon codes is that the value of the first symbol (i.e. the first information symbol) in the codeword is used as the coefficient for the x^{B-1} term, and the value of the last symbol in the codeword (i.e. the last check symbol) is used at the coefficient for the x^0 term.

A valid codeword has the property that, when viewed as a polynomial, it is exactly divisible by the code generator polynomial. The code generator polynomial takes the form:

$$g(x) = (x + a^{L})(x + a^{L+1})(x + a^{L+2})\ldots(x + a^{L+2T-1})$$
$$= g_0 + g_1 x + g_2 x^2 + \ldots + g_{2T-1} x^{2T-1} + x^{2T}$$

The choice of L is somewhat arbitrary, in that all values of L result in valid Reed-Solomon codes. Some simplifications of the decoder are possible if L=1. Some simplifications of the encoder are possible for values of L that give rise to palindromic generator polynomials. We have used the case where L=1.

The 2T check symbols are calculated from: $b(x) = x^{2T}\, a(x) \bmod g(x)$, where the coefficients of $a(x)$ are the information symbols, and the coefficients of $b(x)$ are the check symbols.

The codeword $c(x)$ is calculated from: $c(x) = x^{2T}\, a(x) + b(x)$. This is simply a concatenation of a(x) and b(x). The codeword $c(x)$ is now exactly divisible by $g(x)$.

The minimum distance between different codewords is 2T + 1. Any two codewords will differ by at least 2T + 1 symbols; we can add T errors, and still be closer to the original codeword that to any other.

2 Galois field operations

2.1 Field generation

The Galois field $GF(2^W)$ is a finite field consisting of 2^W elements, generated from a primitive field generator polynomial.

For example, the Galois field $GF(2^4)$ can be generated from the primitive polynomial $p(x) = x^4 + x + 1$.

$GF(2^4)$ consist of the following elements:

$\{ 0, 1, \alpha, \alpha^2, \alpha^3, \alpha^4, \alpha^5, \alpha^6, \alpha^7, \alpha^8, \alpha^9, \alpha^{10}, \alpha^{11}, \alpha^{12}, \alpha^{13}, \alpha^{14} \}$

where α is the root of $p(x)$, i.e. $p(\alpha) = 0$.

2.2 Representations

2.2.1 Power representation

Power representation of field elements is in the form α^i where $0 \le i \le 2^W - 2$.

2.2.2 Tupple representation

Tupple representation of field elements uses the fact that $p(\alpha) = 0$.

For example, in $GF(2^4)$ generated from $p(x) = x^4 + x + 1$:

$$p(x) = x^4 + x + 1$$
$$p(\alpha) = \alpha^4 + \alpha + 1 = 0$$
$$\therefore \alpha^4 = \alpha + 1$$

Similarly,

$$\alpha^5 = \alpha\,\alpha^4$$
$$= \alpha\,(\alpha + 1)$$
$$= \alpha^2 + \alpha$$

In general, each field element may be represented as a linear combination of $\alpha^{W-1} \ldots \alpha^2\ \alpha^1$ and α^0.

For $GF(2^4)$ generated from $p(x) = x^4 + x + 1$, the complete field is:

power representation	*tupple representation*
0	0
1	1
α	α
α^2	α^2
α^3	α^3
α^4	$\alpha + 1$
α^5	$\alpha^2 + \alpha$
α^6	$\alpha^3 + \alpha^2$
α^7	$\alpha^3 + \alpha + 1$
α^8	$\alpha^2 + 1$

α^9	$\alpha^3 + \alpha$
α^{10}	$\alpha^2 + \alpha + 1$
α^{11}	$\alpha^3 + \alpha^2 + \alpha$
α^{12}	$\alpha^3 + \alpha^2 + \alpha + 1$
α^{13}	$\alpha^3 + \alpha^2 + 1$
α^{14}	$\alpha^3 + 1$

2.2.3 Binary representation

Binary representation simply uses the coefficient of α^i as the i^{th} bit in a W-bit number.

For GF(2^4) generated from $p(x) = x^4 + x + 1$, this looks like:

power representation	*tupple representation*	*binary representation*
0	0	0000
1	1	0001
α	α	0010
α^2	α^2	0100
α^3	α^3	1000
α^4	$\alpha + 1$	0011
α^5	$\alpha^2 + \alpha$	0110
α^6	$\alpha^3 + \alpha^2$	1100
α^7	$\alpha^3 + \alpha + 1$	1011
α^8	$\alpha^2 + 1$	0101
α^9	$\alpha^3 + \alpha$	1010
α^{10}	$\alpha^2 + \alpha + 1$	0111
α^{11}	$\alpha^3 + \alpha^2 + \alpha$	1110
α^{12}	$\alpha^3 + \alpha^2 + \alpha + 1$	1111
α^{13}	$\alpha^3 + \alpha^2 + 1$	1101
α^{14}	$\alpha^3 + 1$	1001

It is quite straightforward to show that if a linear feedback shift register is constructed from p(x) and seeded with 1, then the sequence generated will correspond to the binary representation sequence shown above.

2.3 Arithmetic operations

When implementing Galois field arithmetic operations, we assume the data is in binary representation.

2.3.1 Addition

Addition in GF(2):

0 + 0 = 0
0 + 1 = 1
1 + 0 = 1
1 + 1 = 0

Addition in GF(2^4):

$b = b_0 + b_1 a + b_2 a^2 + b_3 a^3$ and $g = c_0 + c_1 a + c_2 a^2 + c_3 a^3$

where $b, g \in$ GF(2^4) and $b_i, c_i \in$ GF(2)

$b + g = (b_0 + c_0) + (b_1 + c_1)a + (b_2 + c_2)a^2 + (b_3 + c_3)a^3$

Thus, addition of two field elements is achieved by XORing the binary representation of two field elements together.

The following parameterised verilog function generates the hardware for Galois field addition:

```
function [WIDTH - 1 : 0] GFAdd_fn;
  input [WIDTH - 1 : 0] a;
  input [WIDTH - 1 : 0] b;
begin
  GFAdd_fn = a ^ b;
end
endfunction
```

2.3.2 Full multiplication

Multiplication in GF(2):

0 . 0 = 0
0 . 1 = 0
1 . 0 = 0
1 . 1 = 1

Multiplication in GF(2^4):

$b = b_0 + b_1 a + b_2 a^2 + b_3 a^3$ and $g = c_0 + c_1 a + c_2 a^2 + c_3 a^3$

where $b, g \in$ GF(2^4) and $b_i, c_i \in$ GF(2) and $p(x) = x^4 + x + 1$

$$\begin{aligned} b \,.\, g &= p_0 + p_1 a + p_2 a^2 + p_3 a^3 + p_4 a^4 + p_5 a^5 + p_6 a^6 \\ &= p_0 + p_1 a + p_2 a^2 + p_3 a^3 + p_4(a + 1) + p_5(a + a^2) + p_6(a^2 + a^3) \\ &= (p_0 + p_4) + (p_1 + p_4 + p_5)a + (p_2 + p_5 + p_6)a^2 + (p_3 + p_6)a^3 \\ &= d_0 + d_1 a + d_2 a^2 + d_3 a^3 \end{aligned}$$

where

$p_0 = b_0 c_0$
$p_1 = b_0 c_1 + b_1 c_0$
$p_2 = b_0 c_2 + b_1 c_1 + b_2 c_0$
$p_3 = b_0 c_3 + b_1 c_2 + b_2 c_1 + b_3 c_0$
$p_4 = b_1 c_3 + b_2 c_2 + b_3 c_1$
$p_5 = b_2 c_3 + b_3 c_2$
$p_6 = b_3 c_3$

The hardware to implement this is illustrated in Figure 1.

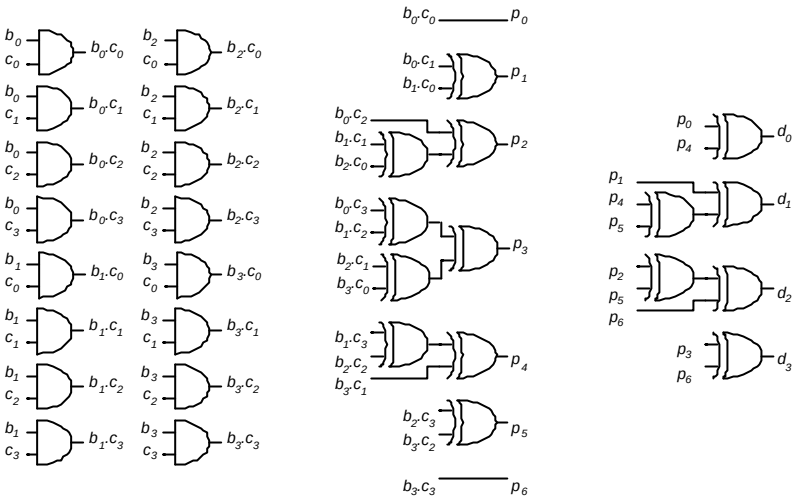


Figure 1 - GF(2^4) Multiplier

The following parameterised verilog function generates the hardware for Galois field multiplication:

```
function [WIDTH - 1 : 0] GFMult_fn;
  input [WIDTH - 1 : 0] a;
  input [WIDTH - 1 : 0] b;
  reg [WIDTH * WIDTH - 1 : 0]
    andarray;
  reg [WIDTH * 2 - 2 : 0]
    product;
  reg [WIDTH - 1 : 0]
    tmp;
  integer
    i,
    j,
    prbs;
begin
  andarray = 0;
  product = 0;
  tmp = 0;
  for (i = 0; i < WIDTH; i = i + 1)
    for (j = 0; j < WIDTH; j = j + 1)
      andarray[i * WIDTH + j] = a[i] & b[j];
  for (i = 0; i < WIDTH * 2 - 1; i = i + 1)
    begin
      product[i] = 0;
      for (j = ((i < WIDTH) ? 0 : i - WIDTH + 1);
               j <= ((i < WIDTH) ? i : WIDTH - 1);
               j = j + 1)
        product[i] = product[i] ^ andarray[WIDTH * j + i - j];
    end
  for (i = 0; i < WIDTH; i = i + 1)
    begin
      tmp[i] = 0;
      prbs = 1;
      for (j = 0; j < WIDTH * 2 - 1; j = j + 1)
        begin
          if (prbs & (1 << i))
            tmp[i] = tmp[i] ^ product[j];
          prbs = prbs << 1;
          if (prbs & (1 << WIDTH))
            prbs = prbs ^ PRIMITIVE;
        end
    end
  GFMult_fn = tmp;
end
endfunction
```

The three loops in the verilog correspond directly to the three stages in the Figure 1.

2.3.3 Constant multiplication

The verilog function for full multiplication can also be used when one of the operands is a constant. If this constant is allowed to propagate through the logic, the logic can be considerably optimised. The specific gates that are removed will depend on the value of the constant. Synopsys can perform this optimisation automatically.

As an example, assume $b = \alpha^9 = \alpha^3 + \alpha = 1010$:

$b = a + a^3$ and $g = c_0 + c_1 a + c_2 a^2 + c_3 a^3$

where $b, g \in$ GF(2^4) and $b_i, c_i \in$ GF(2) and $p(x) = x^4 + x + 1$

$$\begin{aligned} b \cdot g &= p_0 + p_1 a + p_2 a^2 + p_3 a^3 + p_4 a^4 + p_5 a^5 + p_6 a^6 \\ &= p_0 + p_1 a + p_2 a^2 + p_3 a^3 + p_4(a + 1) + p_5(a + a^2) + p_6(a^2 + a^3) \\ &= (p_0 + p_4) + (p_1 + p_4 + p_5)a + (p_2 + p_5 + p_6)a^2 + (p_3 + p_6)a^3 \end{aligned}$$

where

$p_0 = b_0 c_0$
$p_1 = b_0 c_1 + b_1 c_0$
$p_2 = b_0 c_2 + b_1 c_1 + b_2 c_0$
$p_3 = b_0 c_3 + b_1 c_2 + b_2 c_1 + b_3 c_0$
$p_4 = b_1 c_3 + b_2 c_2 + b_3 c_1$
$p_5 = b_2 c_3 + b_3 c_2$
$p_6 = b_3 c_3$

Now, substituting $b_0 = 0$, $b_1 = 1$, $b_2 = 0$ $b_3 = 1$

$p_0 = 0$
$p_1 = c_0$
$p_2 = c_1$
$p_3 = c_2 + c_0$
$p_4 = c_3 + c_1$
$p_5 = c_2$
$p_6 = c_3$

$$\begin{aligned} b \cdot g &= (c_3 + c_1) + (c_0 + c_3 + c_1 + c_2)a + (c_1 + c_2 + c_3)a^2 + (c_2 + c_0 + c_3)a^3 \\ &= d_0 + d_1 a + d_2 a^2 + d_3 a \end{aligned}$$

From this example it can be seen that the array of AND gates can be optimised away, and that the result could be formed by XORing together bit combinations from the variable input. In this case, eight 2-input XOR gates would be required.

In general, the silicon area of a constant multiplier is about 25% of that of a full multiplier.

2.3.4 Inversion

Inversion of a field element is best done as a lookup table for small fields. The following verilog function can be used to construct such a lookup table:

```
function [WIDTH - 1 : 0] GFInverse_fn;
  input [WIDTH - 1 : 0]
```

```
    a;
  reg [WIDTH - 1 : 0]
    res,
    prbstable [0 : (1 << WIDTH) - 1];
  integer
    i,
    prbs;
begin
  prbs = 1;
  for (i = 0; i < (1 << WIDTH); i = i + 1) begin
    prbstable[i] = prbs;
    prbs = prbs << 1;
    if (prbs & (1 << WIDTH))
      prbs = prbs ^ PRIMITIVE;
  end
  res = 0;
  for (i = 0; i < (1 << WIDTH) - 1; i = i + 1)
    if (a == prbstable[i])
      res = prbstable[(1 << WIDTH) - 1 - i];
  GFInverse_fn = res;
end
endfunction
```

The input to the function, *a*, is in binary representation.

The first for loop is used to fill the *prbstable* memory with the binary representation of successive field elements. The binary representation of a^i is stored in *prbstable* entry *i*.

The second for loop iterates through the table until an entry equal to *a* is found. The index of that entry is i, and so the element *a* in power representation is a^i.

The inverse of a^i is $a^{2\text{^}W - 1 - i}$.

The element at entry $2^W - 1 - i$ is the inverse of *a*, in binary representation, and so this value is returned by the function.

It turns out that Synopsys does a reasonable job of expanding these loops and optimising the resultant logic, at least for the field GF(2^8).

2.4 Representation conversion

2.4.1 Power to tupple conversion

The following verilog implements power to tupple conversion. In our implementation this is used to generate constants, and so no logic is actually generated:

```
function [WIDTH - 1 : 0] GFPtoT_fn;
  input [31:0]
    power;
  integer
    i,
    tupple;
begin
  tupple = 1;
  if (power % ((1 << WIDTH) - 1) != 0)
    for (i = 0; i < power % ((1 << WIDTH) - 1) ; i = i + 1)
      begin
        tupple = tupple << 1;
          if (tupple & (1 << WIDTH))
            tupple = tupple ^ PRIMITIVE;
      end
  GFPtoT_fn = tupple;
end
endfunction
```

2.4.2 Tupple to power conversion

Tupple to power conversion is not needed.

3 Encoder

3.1 Algorithm

The encoder needs to calculate the 2T check symbols from the B – 2T information symbols, such that the resultant codeword is exactly divisible by the generator polynomial.

The 2T check symbols are calculated from: $b(x) = x^{2T}\, a(x) \bmod g(x)$, where the coefficients of $a(x)$ are the information symbols, and the coefficients of $b(x)$ are the check symbols.

The codeword $c(x)$ is calculated from: $c(x) = x^{2T}\, a(x) + b(x)$. This is simply a concatenation of $a(x)$ and $b(x)$. The codeword $c(x)$ is now exactly divisible by $g(x)$.

3.2 Interface

```
module encoder (
                // INPUTS
    clock,
    clocken,    // an active high clock enable
    reset,
    load,       // Must be asserted to mark the first symbol of the message.
    din,        // The symbols of the message to be encoded.

                // OUTPUTS

    active,     // Asserted (high) during the output of the codeword.
    sob,        // Asserted (high) to mark the first symbol of the codeword.
    eob,        // Asserted (high) to mark the last symbol of the codeword.
    dout        // The symbols of the codeword.
);
```

The system clock has the rising edge as the active edge.

The reset is an active high *synchronous* reset, and must be asserted for a minimum of one clock period.

To load a B-2T symbol message into the encoder, the load signal must be asserted with the first symbol of the message, and then de-asserted as the remaining B-2T-1 symbols of the message are clocked in over successive clock cycles. There must be a minimum of 2T idle cycles between the B-2T symbol messages to allow time for computation of the check symbols.

Stall cycles may be inserted at any point by taking clocken low. This freezes the state of the whole encoder, and is equivalent to gating the clock.

There is minimal buffering within the encoder, so the first symbol of the B symbol codeword will be output on dout a few clock cycles after load is asserted. The remaining symbols of the codeword are output on dout over successive clock cycles.

The actual latency of the encoder (load active to sob active) is 2 clock cycles (i.e. the equivalent of two pipeline register stages), assuming clocken is held high. Any stall cycles add directly to the latency.

Active is asserted (high) for the B cycles during which the codeword is being output. Sob is asserted (high) with the first symbol of the codeword. Eob is asserted (high) with the last symbol of the codeword.

The throughput of the encoder is such that it can output back-to-back codewords without any idle cycles in between.

The timing diagram for a RS(160,128,T=16) code is shown below in Figure 2.

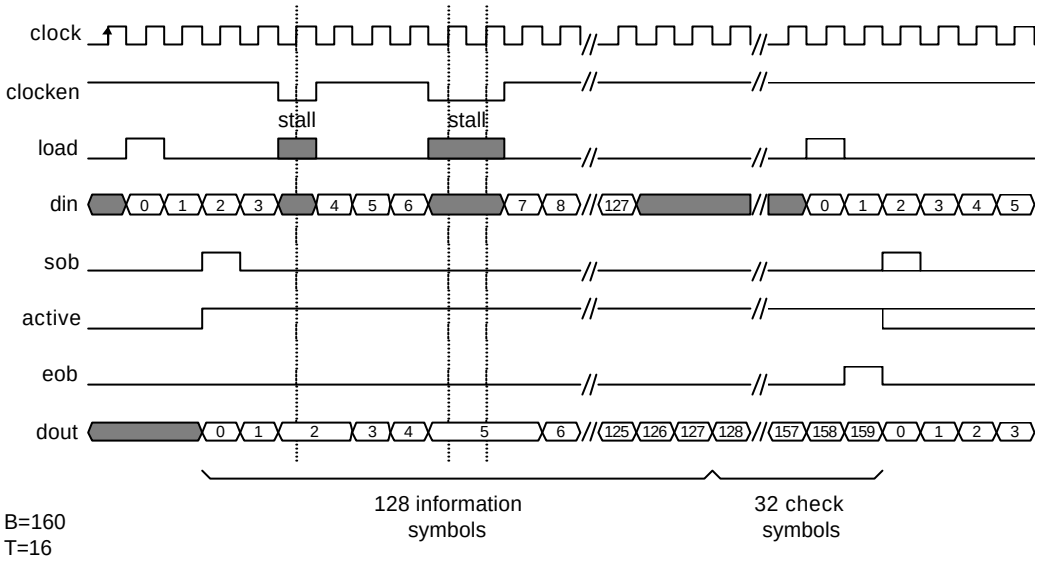


Figure 2 - Encoder timing diagram

This example shows one codeword, possibly followed immediately by a second one. The diagram also shows some stall cycles. During a stall cycle the values applied to the inputs are immaterial, and the outputs hold their previous values.

3.3 Block diagram

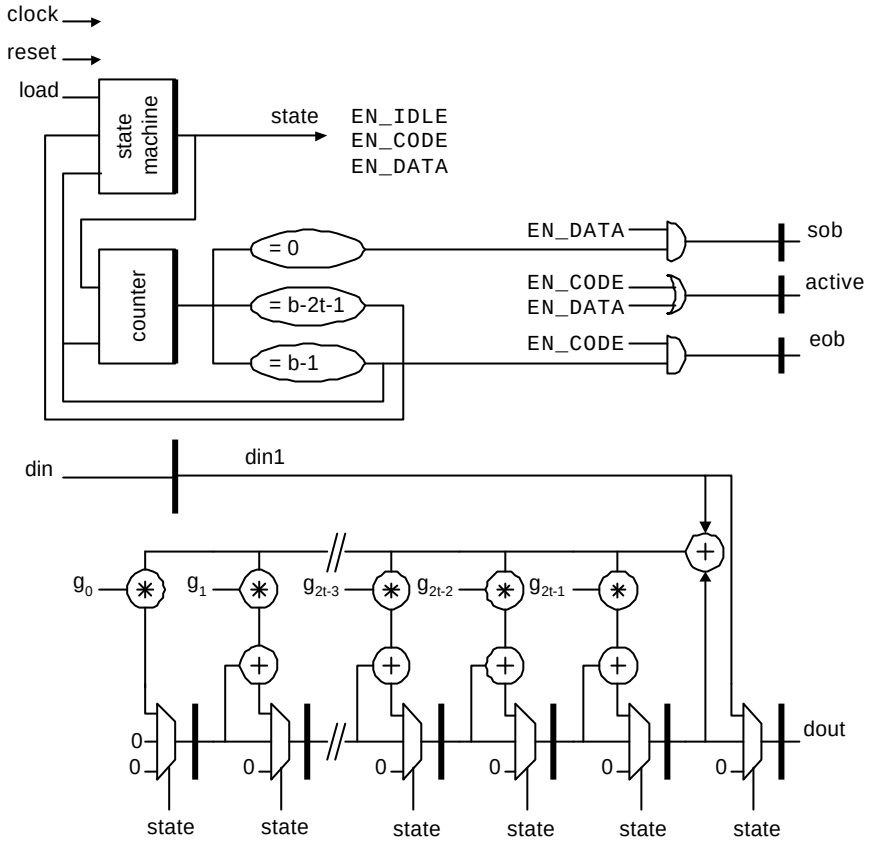


Figure 3 - Encoder block diagram

3.4 Operation

The classic linear-feedback shift register (LFSR) structure can be used to perform polynomial division. For a detailed description of how this works, see [3] page 172. The polynomial formed from the B – 2T information symbols is divided by the generator polynomial, and the remainder of this division is used as the 2T check symbols.

The lower half of Figure 3 implements a 2T tap LFSR. The multiplier coefficients are the coefficients of the generator polynomial in its expanded form (i.e. g_i is the coefficient of the x^i term).

The state table for the encoder block is shown below:

state	*count*	*comment*
EN_IDLE	0	Stay in this state until load asserted.
EN_DATA	0	Stay in this state for B – 2T cycles while information symbols are consumed and check symbols calculated.
EN_DATA	1	
…	…	
EN_DATA	B - 2T - 1	
EN_CODE	B - 2T	

EN_CODE	B - 2T + 1	Stay in the state for 2T cycles while check symbols are clocked out. If generating back-to-back codewords, move straight back to EN_DATA, else EN_IDLE.
…	…	
EN_CODE	B - 1	
EN_DATA or EN_IDLE	0	etc

Table 1 - State table for the encoder block

Initially the state machine is in the EN_IDLE state.

As soon as load is asserted, the state machine moves to the EN_DATA state and the counter starts incrementing. For the next B – 2T cycles the multiplexors connect the adder outputs to the register inputs, thus forming an LFSR. At the end of B – 2T cycles, the LFSR registers hold the remainder.

When the count reaches B – 2T – 1 the state machine moves to the EN_DATA state. The multiplexors now simply connect the registers into a conventional shift register, allowing the check symbols to be shifted out, most significant first. For the next 2T cycles the check symbols are shifted out.

When the counter reaches B – 1, the state machine will move back to the EN_IDLE state (if load is zero) or move back to the EN_DATA state (if there is another data block to encode).

There is an optimisation that could be done to remove the 2T multiplexors. A gate could be inserted in the feedback path, allowing the feedback term to be forced to zero. This would have the same effect as switching over the multiplexors. If this were done, the reset behaviour would change slightly. On reset, the state machine would have to stay in the idle state for 2T cycles, allowing zeros to propagate through the registers.

4 Decoder

4.1 Algorithm

Almost all practical decoders reported in the literature follow the syndrome based decoding approach. This involves the following steps:

1. Calculate the syndromes
2. Calculate the error locator polynomial σ(x) from the syndromes
3. Find the roots of the error locator polynomial σ(x) to determine the error locations
4. Calculate the error values

We have taken this approach, and included additional steps to support erasures decoding.

Let the original codeword be represented by the polynomial:

$$c(x) = c_{B-1}x^{B-1} + c_{B-2}x^{B-2} + \cdots + c_2x^2 + c_1x + c_0$$

Let the error pattern be represented by the polynomial:

$$e(x) = e_{B-1}x^{B-1} + e_{B-2}x^{B-2} + \cdots + e_2x^2 + e_1x + e_0$$

Let the received (corrupted) codeword be represented by the polynomial:

$$\begin{aligned} d(x) &= d_{B-1}x^{B-1} + d_{B-2}x^{B-2} + \cdots + d_2x^2 + d_1x + d_0 \\ &= c(x) + e(x) \end{aligned}$$

Calculate the syndrome polynomial:

$$S(x) = S_{2T-1}x^{2T-1} + S_{2T-2}x^{2T-2} + \cdots + S_2x^2 + S_1x + S_0$$

where $S_i = d(a^{L+i})$

Form the erasure locator polynomial:

$$\Lambda(x) = \prod_{i=0}^{J-1}(x + a^{-v_i})$$

where v_i are the symbol positions of the J erasures.

Calculate the modified syndrome polynomial:

$$T(x) = S(x) \cdot \Lambda(x) \bmod x^{2T}$$

Use the extended Euclidean Algorithm to $s(x)$ and $w(x)$ that solve the key equation:

$$s(x) \cdot T(x) \equiv w(x) \bmod x^{2T}$$

$s(x)$ is the error locator polynomial

$w(x)$ is the errata evaluator polynomial

Use the Chien search to determine the roots of $s(x)$ and $\Lambda(x)$ for $x \in \{a^{-(B-1)} \cdots a^{-(0)}\}$

If $s(x) = 0$ for some $x = a^{-i}$ an error has occurred in symbol *i*, and the error magnitude is given by:

$$e_i = \frac{w(x)}{s'(x) \cdot \Lambda(x)} \text{ where } x = a^{-i}$$

If $\Lambda(x) = 0$ for some $x = a^{-i}$ an erasure has occurred in symbol *i*, and the erasure magnitude is given by:

$$E_i = \frac{w(x)}{s(x) \cdot \Lambda'(x)} \text{ where } x = a^{-i}$$

4.2 Interface

```
module decoder (
                // INPUTS
    clock,
    clocken,    // an active high clock enable
    reset,
    load,       // Must be asserted to mark the first symbol of the codeword.
    erasurein,  // A one indicates the symbol was an erasure.
    din,        // The symbols of the codeword to be decoded.
    maxerasures,// The maximum number of erasure we will tolerate
                   before declaring uncorrectable.

                // OUTPUTS

    active,     // Asserted (high) during the output of the corrected codeword.
    sob,       // Asserted (high) to mark the first symbol of the corrected codeword.
    eob,        // Asserted (high) to mark the last symbol of the corrected codeword.
    dout,       // The symbols of the corrected codeword.
    status,     // 0 - correctable, no errors, no erasures.
                // 1 - correctable, no errors, some erasures.
                // 2 - correctable, some errors, no erasures.
                // 3 - correctable, some errors, some erasures.
                // 4 - uncorrectable, no erasures.
                // 5 - uncorrectable, some erasures.
                // 6 - uncorrectable, special case 1.
                // 7 - uncorrectable, special case 2.
    nerrors,    // The number of errors (undefined if status >= 4)
    nerasures   // The number of erasures.
);
```

The system clock has the rising edge as the active edge.

The reset is an active high *synchronous* reset, and must be asserted for a minimum of one clock period.

To load a B symbol codeword into the decoder, the load signal must be asserted with the first symbol of the codeword, and then de-asserted as the remaining B-1 symbols of the codeword are clocked in over successive clock cycles.

Stall cycles may be inserted at any point by taking clocken low. This freezes the state of the whole decoder, and is equivalent to gating the clock.

The latency within the decoder is approximately two codewords, and some time later the first symbol of the corrected codeword will be output on dout. The remaining symbols of the corrected codeword are output on dout over successive clock cycles.

Active is asserted (high) for the B cycles during which the corrected codeword is being output. Sob is asserted (high) with the first symbol of the corrected codeword. Eob is asserted (high) with the last symbol of the corrected codeword.

The throughput of the decoder is such that it can output back-to-back corrected codewords without any idle cycles in between.

The timing diagram for a RS(160,128,T=16) code is shown below in Figure 4.

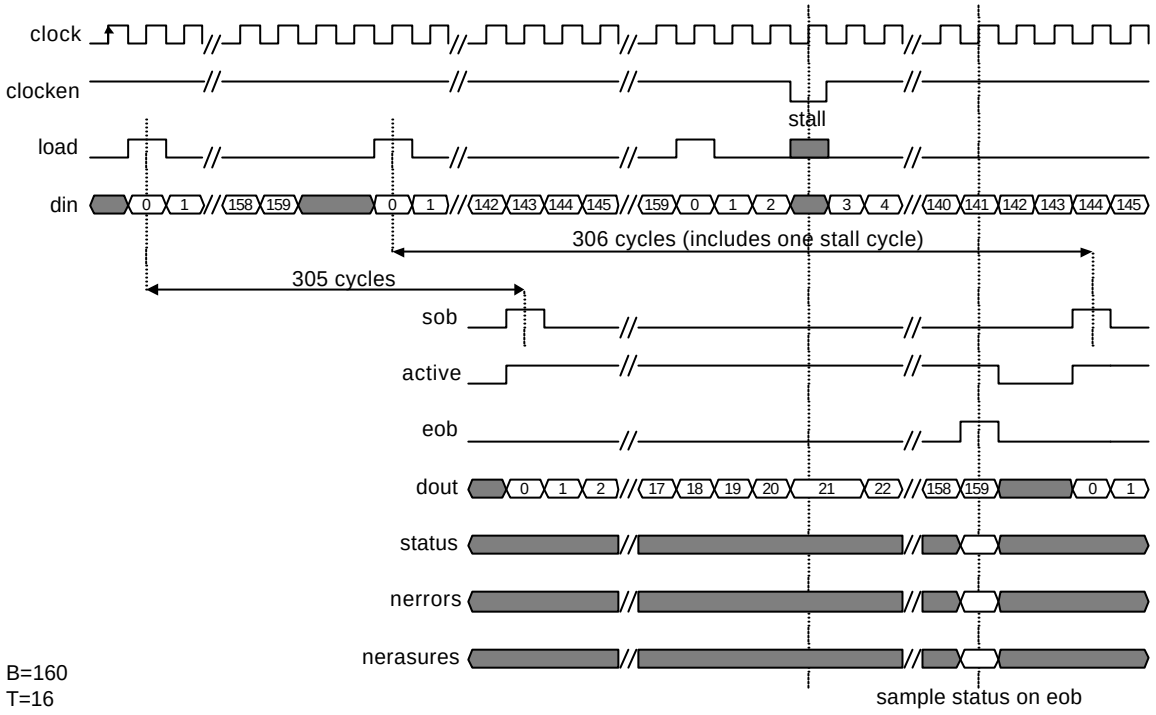


Figure 4 - Decoder timing diagram

For a RS(160, 128, T=16) code the actual latency of the encoder (load active to sob active) is 305 clock cycles (i.e. the equivalent of 305 pipeline register stages), assuming clocken is held high. Any stall cycles add directly to the latency

The status signals indicate whether or not the decoder was able to correct the codeword.

In the case where the codeword was correctable (status 0, 1, 2, 3) nerrors and nerasures indicate the numbers of errors and erasures present.

In the case where the codeword was uncorrectable (status 4, 5, 6, 7) nerrors is undefined and nerasures indicates the number of erasures present. The codeword (just) output in this case is not guaranteed even to be a valid codeword, and should be treated as undefined.

The values of status, nerrors and nerasures should be sampled on the clock edge that occurs when eob is asserted.

The *maxerasures* input allows the maximum number of erasures that will be tolerated to be reduced (below the theoretical maximum of 2T). The rational for doing this is that it reduces the probability of a corrupted codeword with a large number of erasures miscorrecting.

The decoder uses the following basic status codes:

0 Correctable, no errors, no erasures.
1 Correctable, no errors, some erasures.
2 Correctable, some errors, no erasures.
3 Correctable, some errors, some erasures.
4 Uncorrectable, no erasures.
5 Uncorrectable, some erasures.

Due to the internal architecture of the decoder, the status code is generated after the decoder has attempted to correct the corrupted codeword, and is only available when the final symbol of the corrected codeword is being output. This minimises latency. Thus, regardless of whether the error pattern is correctable, or not, the decoder will always attempt to correct it.

As an additional check, the decoder re-calculates the syndromes over each sequence of symbols output by the decoder. This check is performed by the final pipeline stage within the decoder, called the *monitor* block. Two additional status codes are introduced by this block, both of which should be treated as uncorrectable:

6 Uncorrectable, special case 1. This represents the case where the status code going in to the monitor block was 0 to 3 (i.e. correctable), yet for some reason the syndrome of the sequence of symbols output by the decoder was non-zero, indicating an invalid codeword. This could indicate a design error in the decoder. It could also indicate that hardware is not operating reliably, say due to incorrect power supply voltages, or excessive system noise.

7 Uncorrectable, special case 2. This represents the case where the status code going in to the monitor block was 4 or 5 (i.e. uncorrectable), yet for some reason the syndrome of the sequence of symbols output by the decoder was zero, indicating a valid codeword. This event does occur in practise, particularly if the weight of the error pattern is 2T + 1 (i.e. just above what is correctable). Usually the codeword, whilst valid, is the wrong one. The only reason we expose this behaviour externally is because it may help us to design more effective decoders in the future.

4.3 Block diagram

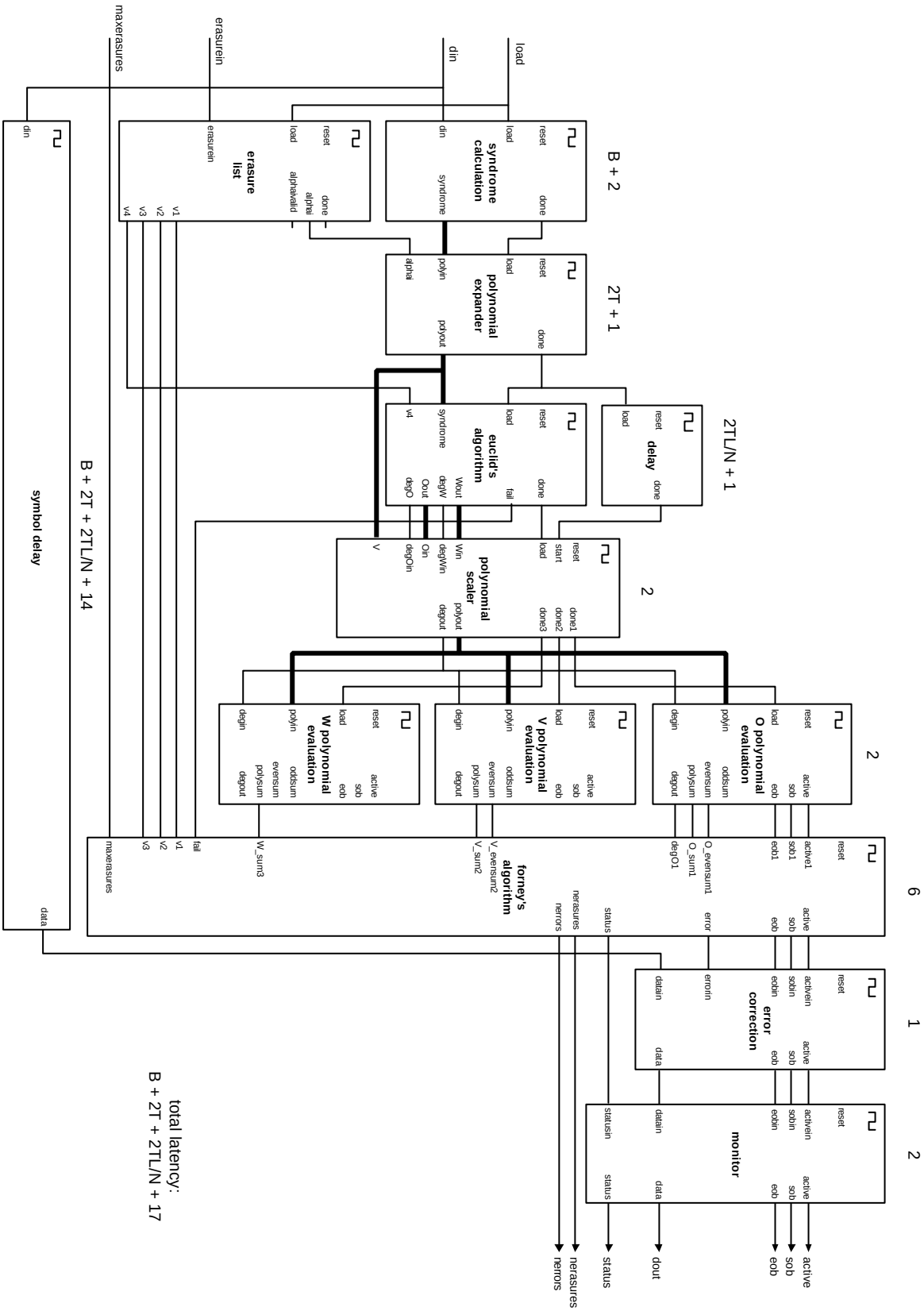


Figure 5 - Decoder block diagram

4.4 Syndrome block

4.4.1 Algorithm

The syndrome block calculates the syndromes of the codeword to be decoded.

The symbols of the codeword form the coefficients of a polynomial, where the first symbol received is d_{B-1} and the last symbol received is d_0:

$$d(x) = d_0 + d_1 x + d_2 x^2 + d_3 x^3 + \cdots + d_{B-1} x^{B-1}$$

The syndromes are obtained by evaluating this polynomial at the roots of the generator polynomial. The generator polynomial has 2T distinct roots (α, α^2, α^3, …, α^{2T}), therefore there are 2T syndromes to be calculated.

The calculation of each syndrome is performed recursively, using Horner's rule:

$$\begin{aligned} syndrome_i &= d(a^i) \\ &= d_0 + d_1 a^i + d_2 a^{2i} + \cdots + d_{B-1} a^{(B-1)i} \\ &= d_0 + a^i (d_1 + a^i (d_2 + \cdots a^i (d_{B-1}) \cdots)) \end{aligned}$$

The order of evaluation of this recursive calculation requires the coefficients to be available in the order d_{B-1} first, through to d_0 last. This matches perfectly the transmission order of symbols into the decoder.

4.4.2 Block diagram

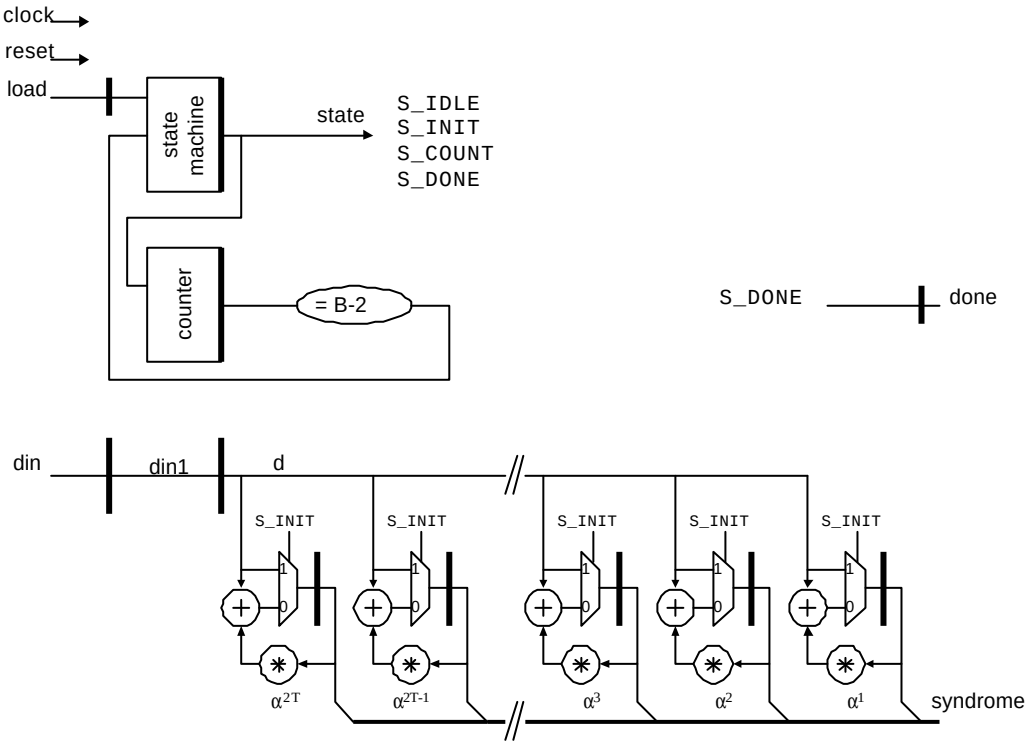


Figure 6 - Syndrome block diagram

4.4.3 Operation

The state table for the syndrome block is shown below:

state	*count*	*comment*
S_IDLE	0	Stay in this state until load asserted.
S_INIT	0	In this state, load d_{B-1} into each syndrome register.
S_COUNT	1	Stay in this state for B – 2 cycles consuming symbols d_{B-2} to d_1 and performing the recursive calculation.
S_COUNT	2	
…	…	
S_COUNT	B – 2	
S_DONE	B – 1	Consume the last symbol, d_0, and perform the last iteration of the recursive calculation. If decoding back-to-back codewords, move straight back to S_INIT, else S_IDLE.
S_INIT or S_IDLE	0	etc

Table 2 - State table for syndrome block

The latency of this block (assuming no stall cycles) is B + 2 cycles.

4.5 Erasurelist block

4.5.1 Algorithm

The erasurelist block contains a FIFO like structure to maintain a list of up to 2T erasure locations.

If the first symbol of the codeword is as an erasure, a value of $\alpha^{-(B-1)}$ is queued in the FIFO. If the next symbol is an erasure, $\alpha^{-(B-2)}$ queued in the FIFO etc.

In addition to storing the erasure locations, the erasurelist block pre-computes the following values. Let the total number of erasures in a codeword be J, then:

$$v1 = J$$

$$v2 = \alpha^{J}$$

$$v3 = \alpha^{-(B-1)J}$$

$$v4 = (2T - J)\,L/N \text{ if } J \le 2T$$

$$= 0 \text{ otherwise}$$

The v1, v2 and v3 values are used for the Forney block, the v4 value is used by the Euclid block.

For details of L and N, see section 4.7.3.

4.5.2 Block diagram

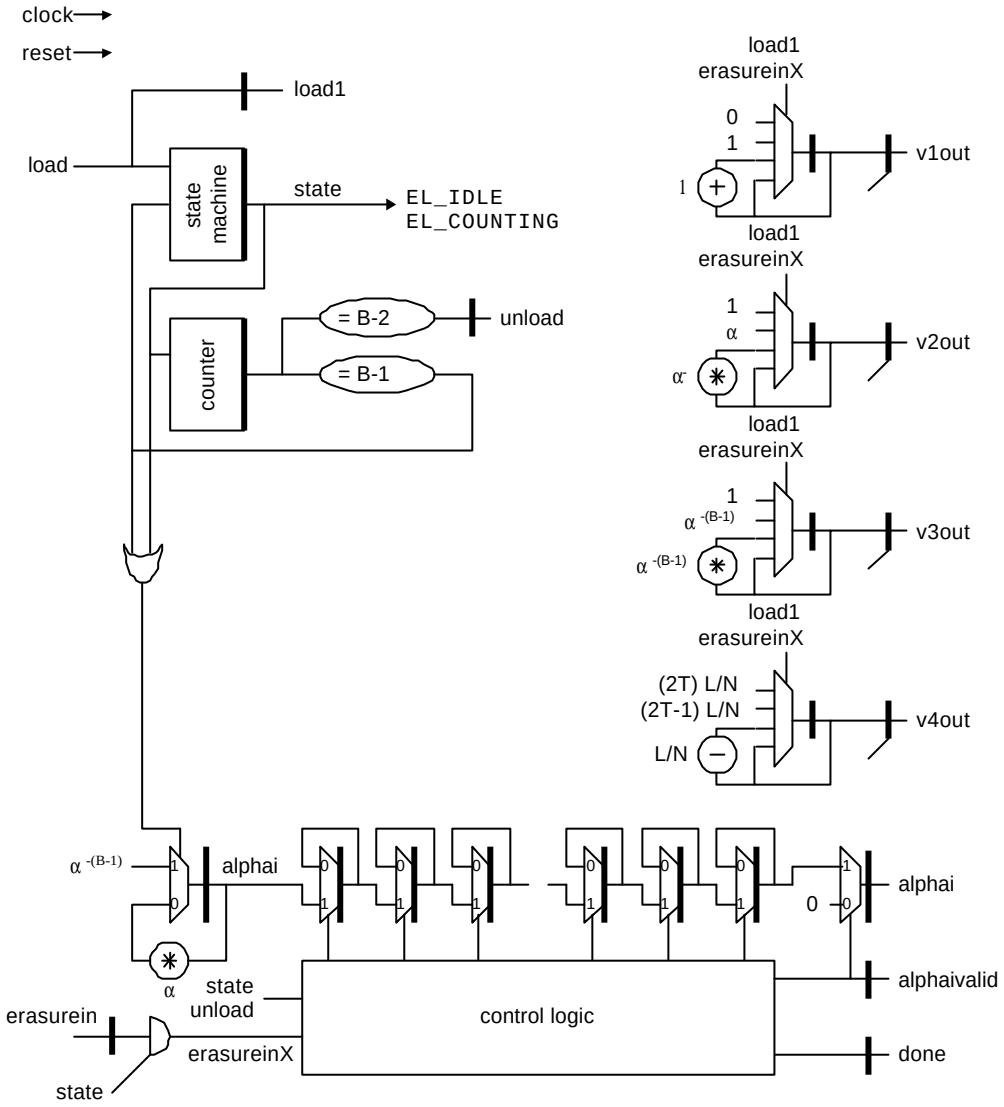


Figure 7 - Erasurelist block diagram

4.5.3 Operation

The state table for the syndrome block is shown below:

state	*count*	*alphai*	*comment*
EL_IDLE	0	$\alpha^{-(B-1)}$	Stay in this state until load asserted.
EL_COUNTING	0	$\alpha^{-(B-1)}$	Stay in this state for B cycles consuming erasure flags e_{B-1} to e_0.
EL_COUNTING	1	$\alpha^{-(B-2)}$	
…	…	…	
EL_COUNTING	B – 2	$\alpha^{-(1)}$	
EL_COUNTING	B – 1	$\alpha^{-(0)}$	
EL_IDLE or EL_COUNTING	0	$\alpha^{-(B-1)}$	etc

Table 3 - State table for erasurelist block

The latency of this block (assuming no stall cycles) is B + 2 cycles.

4.6 Expander block

4.6.1 Algorithm

The purpose of this block is to calculate:

a) The erasure locator polynomial:

$$\Lambda(x) = (x + a^{-v_0})(x + a^{-v_1})(x + a^{-v_2}) \cdots (x + a^{-v_{J-1}})$$

where the set of a^{-v_i} represents the locations of J erasures where $0 \le J < 2T$.

b) The modified syndrome polynomial:

$$T(x) = S(x) \cdot \Lambda(x)$$

where S(x) is the syndrome polynomial.

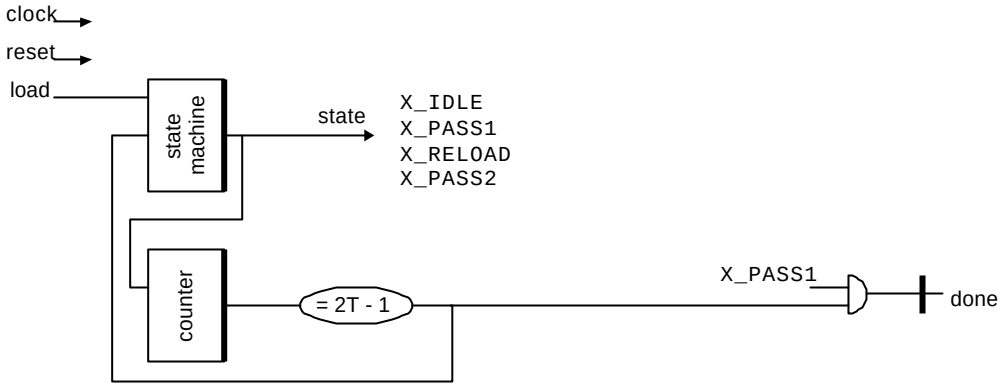
In both cases, the same basic operation is used:

$$polyout(x) = polyin(x) \cdot (x + a^{-v_0})(x + a^{-v_1})(x + a^{-v_2}) \cdots (x + a^{-v_{J-1}})$$

To calculate *T(x)* the initial value loaded into *polyin(x)* is *S(x)*.

To calculate *S(x)* the initial value loaded into *polyin(x)* is 1.

4.6.2 Block diagram



sel:	condition:
0:	X_IDLE && load
1:	X_RELOAD
2:	(X_PASS1 || X_PASS2) && (alphai != 0)
3:	(X_PASS1 || X_PASS2) && (alphai == 0)

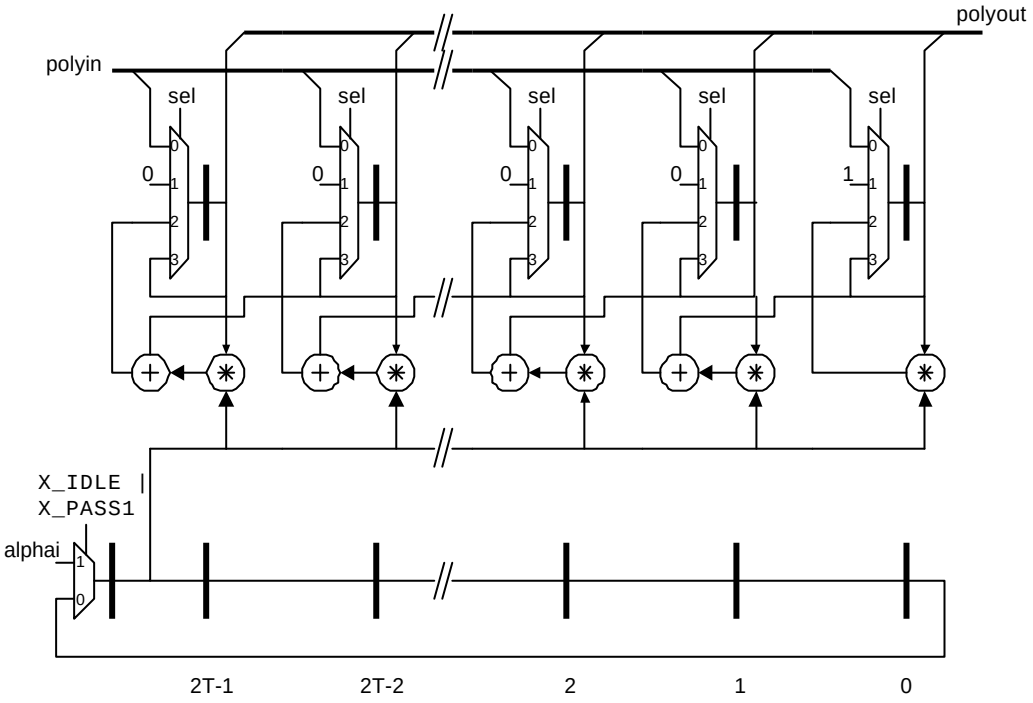


Figure 8 - Expander block diagram

4.6.3 Operation

The same hardware is used (time multiplexed) to generate both $T(x)$ and $L(x)$.

The state table for the expander block is shown below:

state	*count*	*comment*
X_IDLE	0	Load the polynomial register with polyin when load asserted, them move on.
X_PASS1	0	

X_PASS1	1	Stay in this state for 2T cycles consuming up to 2T erasure locations.
…	…	
X_PASS1	2T – 1	
X_RELOAD	0	Reload polynomial register with 1.
X_PASS2	0	Stay in this state for 2T cycles consuming up to 2T erasure locations.
X_PASS2	1	
…	…	
X_PASS2	2T - 1	
X_IDLE	0	etc

Table 4 - State table for expander block

In the first pass the polynomial register is initialised with $S(x)$, and over the next 2T clock cycles the erasure locations a^{v_i} are consumed. At the end of 2T cycles, the polynomial register holds $T(x)$. Done is asserted at this point to indicate the cycle in which $T(x)$ is available.

In the second pass the polynomial register is initialised with 1 and over the next 2T clock cycles the erasure locations a^{v_i} are consumed. At the end of 2T cycles, the polynomial register holds $L(x)$. This value is held until load is asserted again.

The erasure locations are stored in a shift register for re-use in the second pass, so that they only need inputting once into the block. A value of zero is an invalid erasure location, and so this is used as padding if there J < 2T.

The latency of this block (assuming no stall cycles) is 2T+1 cycles.

4.7 Euclid block

4.7.1 Algorithm

The design of this block is heavily leveraged from Gadiel Seroussi's work. For further details see [4] pages 205-241. Some of the diagrams are reproduced here to aid understanding of our implementation.

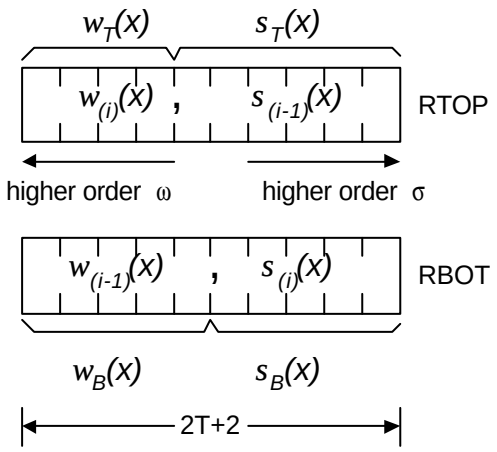


Figure 9 - Data structure for Euclidean computation

This data structure efficiently holds four polynomials. The maximum degree of each polynomial is 2T (so it takes up 2T+1 register slots). However, the algorithm is such

that as the degree of $s_i(x)$ increases, so the degree of $w_i(x)$ decreases. Thus it is possible to pack both polynomials into 2T+2 register slots.

The following procedure describes the computation performed. Again, in [4] there are many pages of mathematical proof and explanation.

Procedure E2: Extended Euclidean algorithm (modified version)

1. **Initialize**

$$w_T(x) := x^{2T} \qquad s_T(x) := 1$$

$$w_B(x) := T(x) \qquad s_B(x) := 0$$

At all times maintain $d = \deg w_T(x) - \deg w_B(x)$.

Initially $d = 1$

2. Repeat $2T - J$ times (where J is the number of erasures):

   a. **set**
   m_T := left most (leading) coefficient of $w_T(x)$
   m_B:= left most (leading) coefficient of $w_B(x)$

   b. **if** $m_B \neq 0$ and $\delta > 0$ (i.e. the bottom comma is to the left of the top comma), **then**
   swap RTOP and RBOT
   swap m_T and m_B

   c. **if** $m_B \neq 0$, **then set**

$$w_B(x) := m_T w_B(x) - x^d m_B w_T(x)$$

$$s_T(x) := m_T s_T(x) - x^d m_B s_B(x)$$

   d. **shift** RBOT (and its comma) one position to the left**.**

3. **output** $w_B(x)$ as $w(x)$ and $s_T(x)$ as $s(x)$.

This differs from Gadiel's procedure in the following aspects:

i. We initialise with $T(x)$ rather than $S(x)$
ii. The number of iterations is reduced from $2T$ to $2T - J$.
iii. The sign of δ is reversed (this is purely cosmetic).

In Gadiel's implementation, the data structure for the Euclidean computation is exactly $2T+2$ slots wide and there are O(2T) functional units (referred to as ST cells). Each ST cell contains an adder and a multiplier. A basic iteration of procedure E2 can be started every three cycles. The multiplier is busy in two of these cycles, and the third is overhead. A new Euclidean computation can be started every O(6T) cycles, and the overall latency is O(12T) cycles

4.7.2 Basic cell design

The goal for our basic cell design is to maximize throughput and minimize latency. Consequently, our basic cell design contains an adder and two multipliers. With O(2T) of these basic cells, an iteration of procedure E2 can be started every cycle, which results in a throughput and latency of O(2T) cycles. Our basic cell is illustrated in Figure 10.

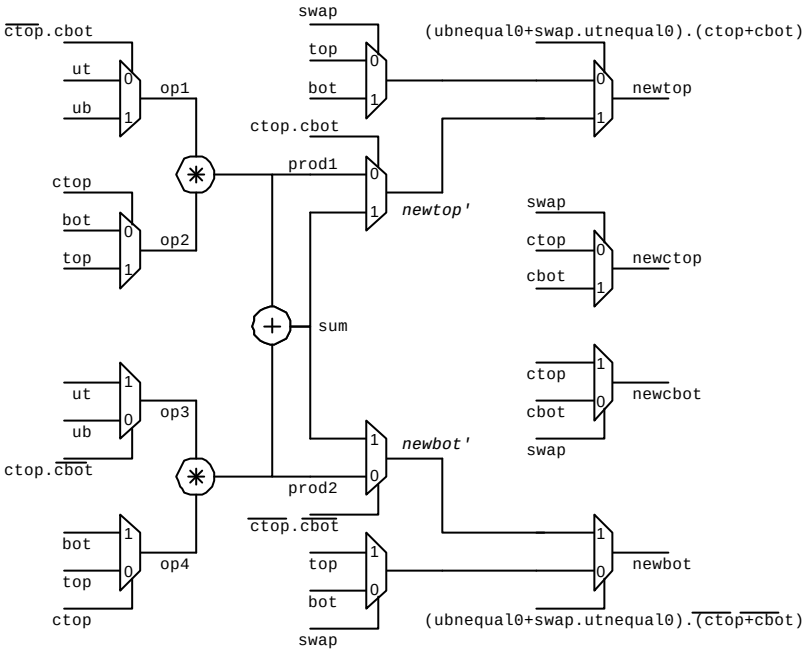


Figure 10 - Euclid cell design

This purely combinatorial logic handles steps (a), (b) and (c) in the procedure E2 simultaneously.

The following signals are passed into this logic:

- top – the value of the top polynomial coefficient
- ctop –flag to indicate whether the coefficient is left (0) or right (1) of the comma
- bot – the value of the bottom polynomial coefficient
- cbot –flag to indicate whether the coefficient is left (0) or right (1) of the comma
- ut – corresponds to m_T at the start of step (a), prior to any swapping.
- ub – corresponds to m_B at the start of step (a), prior to any swapping.
- swap – the condition tested in step (b), set to one if $m_B \neq 0$ and $\delta > 0$ prior to any swapping.
- ubnequal0 – set to one if $m_B \neq 0$, prior to any swapping
- utnequal0 – set to one if $m_T \neq 0$, prior to any swapping

The main function of the cell is to implement the computation of step (c):

$$w_B(x) := m_T w_B(x) - x^d m_B w_T(x)$$

$$s_T(x) := m_T s_T(x) - x^d m_B s_B(x)$$

Let *i* represent the index number the slot in RTOP and RBOT in Figure 9 (with an index of 0 being at the far right hand side).

Let top_i be the value stored in slot *i* of RTOP.

Let bot_i be the value stored in slot *i* of RBOT.

We define the following shorthand notation:

$$a_i = m_T \cdot bot_i$$
$$b_i = m_B \cdot top_i$$
$$c_i = m_T \cdot top_i$$
$$d_i = m_B \cdot bot_i$$

Given the layout of the polynomials in the registers, multiplication factor x^d is obtained trivially due to the alignment of the polynomials. Although procedure E2 specifies subtraction, this the same as addition in a Galois field, and so $(a_i - b_i)$ is the same as $(b_i - a_i)$.

Consider the case where no swap occurs:

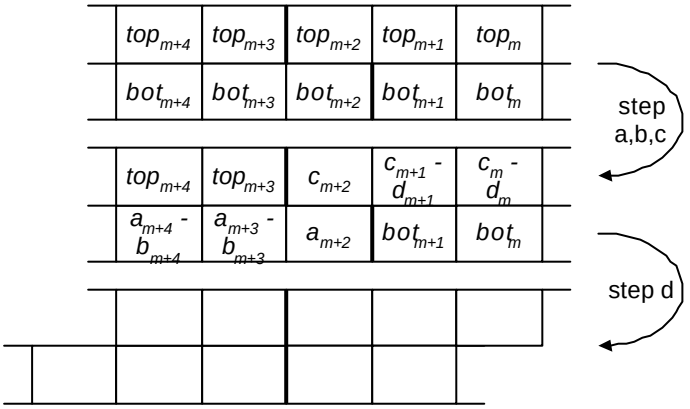


Figure 11 - Euclid cell operation - no swap occurs

From this diagram it is possible to verify that the logic in Figure 10 does indeed implement the correct logic:

position	*m+4 and m+3*	*m+2*	*m + 1 and m*
ctop	*0*	*1*	*1*
cbot	*0*	*0*	*1*
swap	*0*	*0*	*0*
op1	*ut*	*ut*	*ut*
op2	*bot*	*top*	*top*
op3	*ub*	*ut*	*ub*
op4	*top*	*bot*	*bot*
prod1	*ut . bot = a*	*ut . top = c*	*ut . top = c*
prod2	*ub . top = b*	*ut . bot = a*	*ub . bot = d*
sum	*a + b*	*a + c*	*c + d*
newtop	*top*	*prod1 = c*	*sum = c + d*
newbot	*sum = a + b*	*prod2 = a*	*bot*

Now consider the case where a swap occurs:

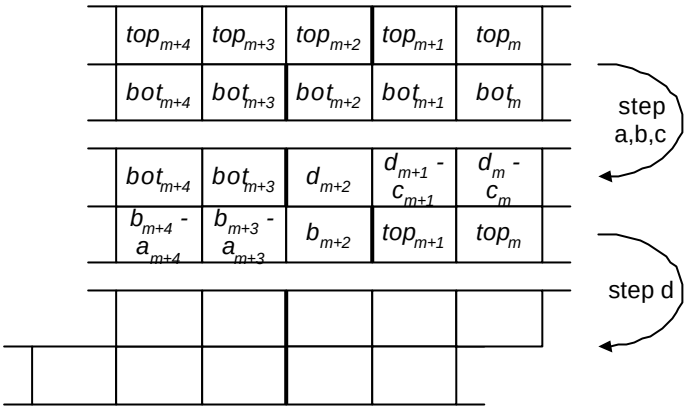


Figure 12 - Euclid cell operation – swap occurs

From this diagram it is possible to verify that the logic in Figure 10 does indeed implement the correct logic:

position	*m+4 and m+3*	*m+2*	*m + 1 and m*
ctop	*0*	*0*	*1*
cbot	*0*	*1*	*1*
swap	*1*	*1*	*1*
op1	*ut*	*ub*	*ut*
op2	*bot*	*bot*	*top*
op3	*ub*	*ub*	*ub*
op4	*top*	*top*	*bot*
prod1	*ut . bot = a*	*ub . bot = d*	*ut . top = c*
prod2	*ub . top = b*	*ub . top = b*	*ub . bot = d*
sum	*a + b*	*b + d*	*c + d*
newtop	*bot*	*prod1 = d*	*sum = c + d*
newbot	*sum = a + b*	*prod2 = b*	*top*

One final complication is that the computation in step (c) of procedure E2 is conditional on *(m_B ¹ 0)* after the swap. This is achieved by adding an additional *load* condition to the final multiplexor used to generate *newtop* and *newtop*. In our design the combinatorial cell handles steps (a), (b) and (c) simultaneously, and so this *load* condition (logically) expands to:

$$load = \overline{swap} \cdot (m_B \neq 0) + swap \cdot (m_T \neq 0)$$

For convenience, define the following:

$$a = (m_T \neq 0)$$
$$b = (m_B \neq 0)$$
$$c = (d > 0)$$

Then,

$$swap = b \cdot c$$
$$\overline{swap} = \bar{b} + \bar{c}$$
$$load = (\bar{b} + \bar{c}) \cdot b + (b \cdot c) \cdot a$$
$$= b \cdot \bar{b} + b \cdot \bar{c} + b \cdot c \cdot a$$
$$= b \cdot (\bar{c} + c \cdot a)$$

This can be simplified further by noting that m_T is never zero. The initial value for m_T is 1 because $w_T(x) = x^{2T}$. The only time the value of m_T changes is following a swap, when it is updated from m_B, which will be non-zero or the swap would not happen. Thus, substituting *a = 1:*

$$load = b \cdot (\bar{c} + c \cdot 1)$$
$$= b$$

For historic reasons[1] the condition we actually use for *load* is:

$$load = (m_B \neq 0) + swap \cdot (m_T \neq 0)$$
$$= b + b \cdot c \cdot a$$
$$= b$$

The second part of this expression is actually redundant, but does not affect the logical operation of the system. As the design is now frozen, and there is no logical problem, we have not changed this.

4.7.3 Cell sharing

If the throughput of the Euclidean computation O(2T) is less than the code length B, then the decoder throughput will not be limited by the Euclid stage. In certain applications, where overall latency is not critical, it may be advantageous to allow the Euclidean computation be spread over additional cycles, if this reduces the implementation size. This can be achieved by allowing a basic cell to be shared between multiple slots in the data structure. For example, by sharing a cell between two slots, the total number of cells required is halved, and the computation time and latency will increase to O(4T).

In general, procedure E2 requires $O(2T \times 2T \times 2) = O(8T^2)$ multiplications, regardless of how it is implemented. Since our basic cell contains two multipliers, a configuration with *N* basic cells will complete the computation in $O(4T^2/N)$ cycles.

The number of computation cells can be configured at synthesis time by the synthesis parameter *N*. There are several constraints on *N*, but the main one is that it must be a factor of the width of the data structure.

The width of the data structure is controlled by a second synthesis parameter, *L*, where $L \geq 2T+2$. If $L = 2T+2$, the data structure is identical to that shown in Figure 9. If $L > 2T+2$, then some redundant cells are added. The configuration used for the RS(160, 128, T=16) code is shown in Figure 13.

[1] An oversight that only came to light as I was writing this documentation.

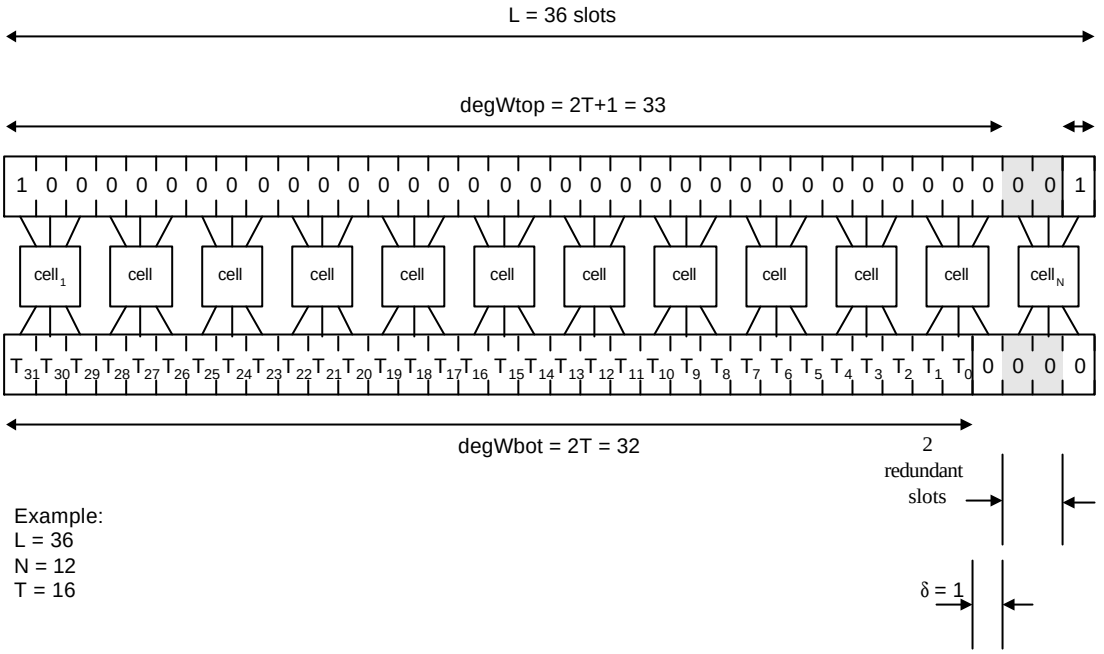


Figure 13 - Example of the modified Euclidean data structure

It turns out that it is convenient to locate the redundant cells between the polynomials, rather than at either end. This means their locations move as the computation progresses. However, this does not affect the results.

The initialisation values for the top and bottom registers are also shown in Figure 13.

During the computation following variables are maintained:

degWtop – the space occupied by $w_T(x)$ (initially 2T+1)

degWbot – the space occupied by $w_B(x)$ (initially 2T)

At the end of the computation, $w_B(x)$ is output as $w(x)$ and $s_T(x)$ is output as $s(x)$. The degrees of these polynomials are calculated as follows:

$degW = degWbot$

$degO = 2T + 2 - degWtop$

At a register-transfer level, things get more involved. There have been examples in the literature of similar hardware sharing schemes, but all of these required an additional overhead in terms of multiplexors to route the data values appropriately. In our implementation we have managed to eliminate the overhead, by forming ringlets of registers. Data values circulate around these ringlets, and the computation cell is connected to a fixed point. This is illustrated in Figure 14.

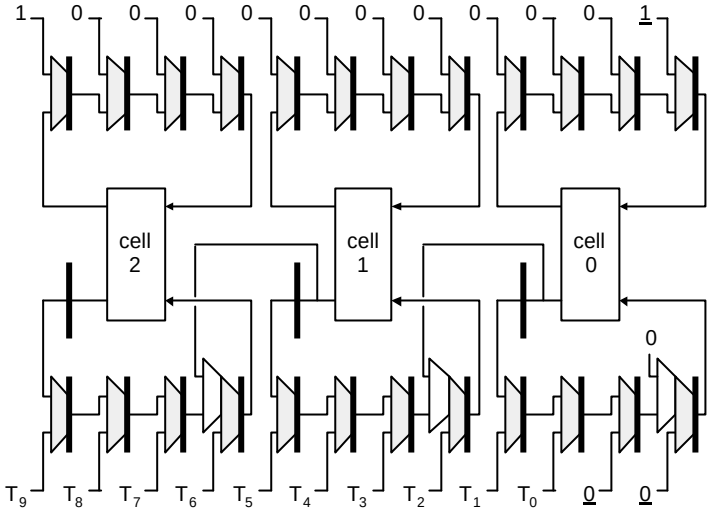


Figure 14 - Implementation of euclidean computation

This example is for L=12 and N=3; each cell is shared between four slots. The shaded multiplexors are required to load the initialisation values into the registers. Where the initialisation values are constants, these multiplexors will be optimised. The other three multiplexors and three additional registers implement step (d) of procedure E2, effectively shifting the bottom registers on position left at the end of an iteration.

4.7.4 Operation

Procedure E2 requires basic *2T-J* iterations, where *J* indicates the number of erasures. In our implementation each iteration is spread over L/N clock cycles to reduce the number of basic cells from L to N. Thus, the number of cycles to complete the calculation is $(2T - J) \times L/N$.

The state sequencing of the computation varies depending on *L, N, T* and *J*, but there are two basic cases to consider:

i. The trivial case, where there are (*J=2T)* erasures:

state	*count1*	*count2*	*done*	*degO*	*degW*	*fail*
E_IDLE	0	0	0	-	-	-
E_IDLE	0	0	1	1	2T	0

Table 5 - State table for euclid block (trivial case)

ii. The non-trivial case, where there are (J < 2T) erasures:

(this example uses *L=36, N=12, T=16* and *J=7*)

state	*count1*	*count2*	*done*	*degO*	*degW*	*fail*
E_IDLE	0	0	0	-	-	-
E_CALCING	0	0	0	-	-	-
E_CALCING	1	1	0	-	-	-
E_CALCING	2	2	0	-	-	-
E_CALCING	3	0	0	-	-	-
E_CALCING	4	1	0	-	-	-

E_CALCING	5	2	0	-	-	-
…	…	…	…	…	…	…
E_CALCING	72	0	0	-	-	-
E_CALCING	73	1	0	-	-	-
E_CALCING	74	2	0	-	-	-
E_IDLE	0	0	1	2T+2-degWtop	degWbot	(degWbot >= degWtop)

Table 6 - State table for euclid block (non-trivial case)

4.8 Delay block

The latency through the Euclid block is *(2T-J)L/N* cycles, which varies depending on the number of erasures. To ensure the decoder as a whole has constant latency, the next block (the scaler) is triggered $(2T)L/N+1$ cycles after the Euclidean computation starts, rather than on its completion. This achieved using the delay block.

4.9 Scaler block

4.9.1 Algorithm

The roots of the error locator polynomial $s(x)$ indicate the error locations. An exhaustive search is used to determine these roots. This procedure is known as the Chien search.

The convention with Reed-Solomon codes is that the on-the-wire ordering is such that the first symbol represents the coefficient of the x^{B-1} term, and the last symbol represents the coefficient of the x^0 term. It is advantageous to synchronise the Chien search with this transmission order, since this minimises buffering within the decoder and reduces overall latency.

The Chien search involves evaluating $s(x)$ for $x \in \{\alpha^{-(B-1)} \cdots \alpha^{-(0)}\}$.

For a full length code, $B = 2^W-1$, and so the first location checked is

$$x = \alpha^{-(B-1)} = \alpha^{-((2^W-1)-1)} = \alpha^{(2^W-1)}\alpha^{-((2^W-1)-1)} = \alpha$$

The next location would be:

$$x = \alpha^{-(B-2)} = \alpha^{-((2^W-1)-2)} = \alpha^{(2^W-1)}\alpha^{-((2^W-1)-2)} = \alpha^2$$

and so on. The classic approach to implementing the Chien search uses the following:

$$s(x) = s_0 + s_1 x + s_2 x^2 + \ldots + s_{2T} x^{2T}$$
$$s(\alpha) = s_0 + s_1 \alpha + s_2 \alpha^2 + \ldots + s_{2T} \alpha^{2T}$$
$$s(\alpha^2) = s_0 + s_1 \alpha^2 + s_2 \alpha^4 + \ldots + s_{2T} \alpha^{4T}$$
$$s(\alpha^3) = s_0 + s_1 \alpha^3 + s_2 \alpha^6 + \ldots + s_{2T} \alpha^{6T}$$
$$etc$$

Strictly speaking, $s(x)$ can be of degree at most T, and so there is some redundancy here. However, the other polynomials $w(x)$ and $\Lambda(x)$ can be of degree 2T, and since the scaler block is shared, we assume any of the polynomials can be of degree 2T.

This computation can be implemented by $2T+1$ stages, where each stage includes a register, a constant multiplier and an adder, connected as shown in Figure 15.

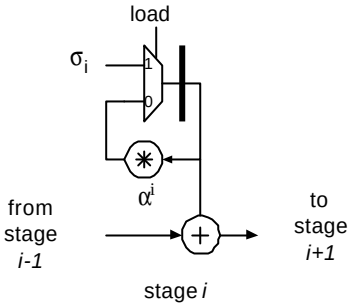


Figure 15 - Polynomial evaluation cell

The registers are initialised with the coefficients of $s(x)$. Over successive clock cycles, the i^{th} coefficient is repeatedly multiplied by a^i, and the results summed.

One clock cycle after loading, the sum will be $s(a)$; this will be zero if there is an error in the first symbol of the codeword. In general, after N clocks, the sum will be $s(a^N)$; this will be zero if there is an error in the N^{th} symbol of the codeword.

So far we have described the operation for full-length codes, where $B = 2^w - 1$. In general, we also need to handle shortened codes, where $B < 2^w - 1$.

The above hardware still works in this case, but needs $2^W - B$ clock cycles following initialisation before the first useful result $s(a^{-(B-1)})$ is obtained. This is effectively dead time, and limits the overall throughput of the decoder, preventing it from decoding back-to-back codewords. Note that even in the ideal case of a full-length code, there is one cycle of dead time. This is because the initialisation value is $s(a^0)$ which does not correspond to a location within the codeword.

For a shortened-code, the first location checked should be

$$x = a^{-(B-1)} = a^{(2^W-1)}a^{-(B-1)} = a^{2^W-B}$$

The next location would be:

$$x = a^{-(B-2)} = a^{(2^W-1)}a^{-(B-2)} = a^{2^W-B+1}$$

and so on.

To eliminate the dead time, we need to scale the coefficients of $s(x)$ to effectively allow the Chien search to start immediately at position $B-1$. This scaling is straightforward: the i^{th} coefficient of $s(x)$ needs scaling by $a^{(2^W-B)i}$. A bank of 2T constant multipliers can achieve this in one cycle.

4.9.2 Block diagram

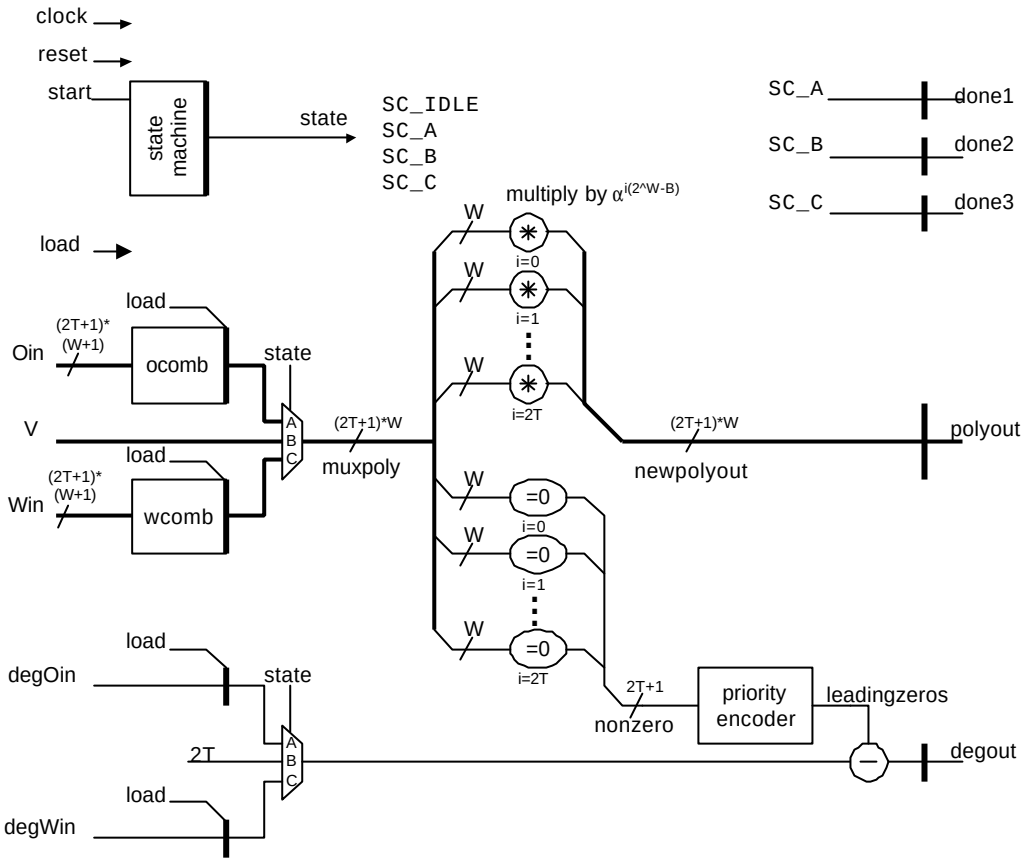


Figure 16 - Polynomial scaler block diagram

4.9.3 Operation

The scaler block implements the polynomial scaling described above. There are three polynomials that need scaling. These are the error locator polynomial $s(x)$ and the errata evaluator polynomial $w(x)$ from the Euclid block, and the erasure locator polynomial $\Lambda(x)$ from the Polynomial Expander block. Once *start* is asserted, these are processed over successive clock cycles. The three done signals indicate to successive blocks when each scaled polynomial (and its true degree) is available. These signals will be skew by one cycle with respect to each other.

The format of the polynomials $s(x)$ and $w(x)$ from the Euclid block is somewhat strange, due to the layout of registers within that block. The *Ocomb* and *Wcomb* functions in the scaler serve to map $s(x)$ and $w(x)$ to a standard format, illustrated in Figure 17. Note that the $\Lambda(x)$ polynomial does not require any reformatting.

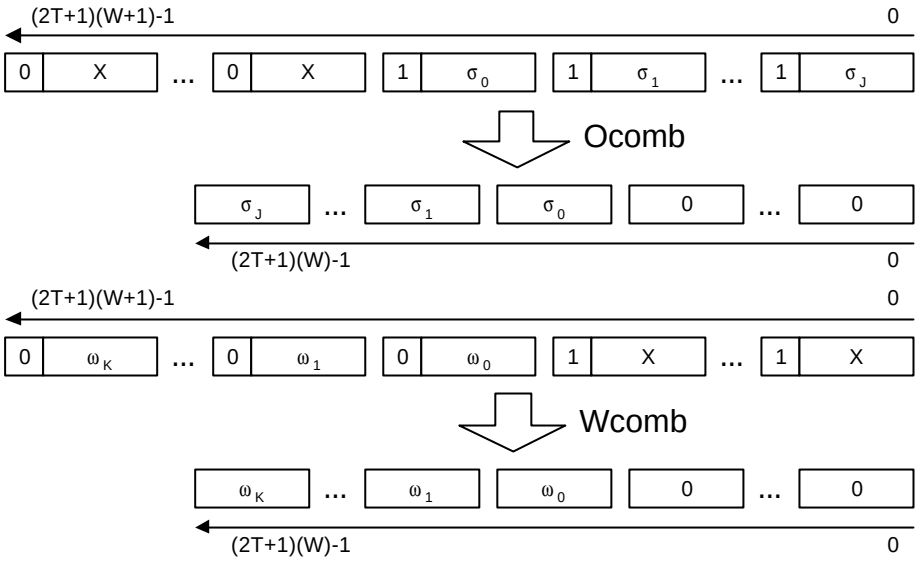


Figure 17 - Reformatting of polynomials in scaler

Note that the alignment of $s(x)$ and $w(x)$ is such that the polynomials are scaled. More specifically, if $s(x)$ is of degree J, then the scale factor is x^{2T-J}. Similarly, the scale factor for $w(x)$ is x^{2T-K}. These scale factors are compensated for in the Forney block.

Since the result of the Euclid block is only valid for one cycle, the scaler block includes registers to capture the result when available. These are loaded when the load signal is asserted. The load signal is driven from the done signal from the Euclid block.

The last function performed by the scaler block is to calculate the true degrees of the polynomials, since the results of the Euclid block may include leading zeros. The number of leading zeros is counted, using a priority encoder, and this value is subtracted from the degree output by the Euclid block. The true degrees of the polynomials are used by the Forney block to detect uncorrectable error patterns.

The latency of this block (assuming no stall cycles) is 2 cycles.

4.10 Polynomial evaluation block

4.10.1 Algorithm

Polynomial evaluation is part of the Chien search, as described in section 4.9.1.

We sum the odd and even terms separately, for reasons described in section 4.11.

4.10.2 Block diagram

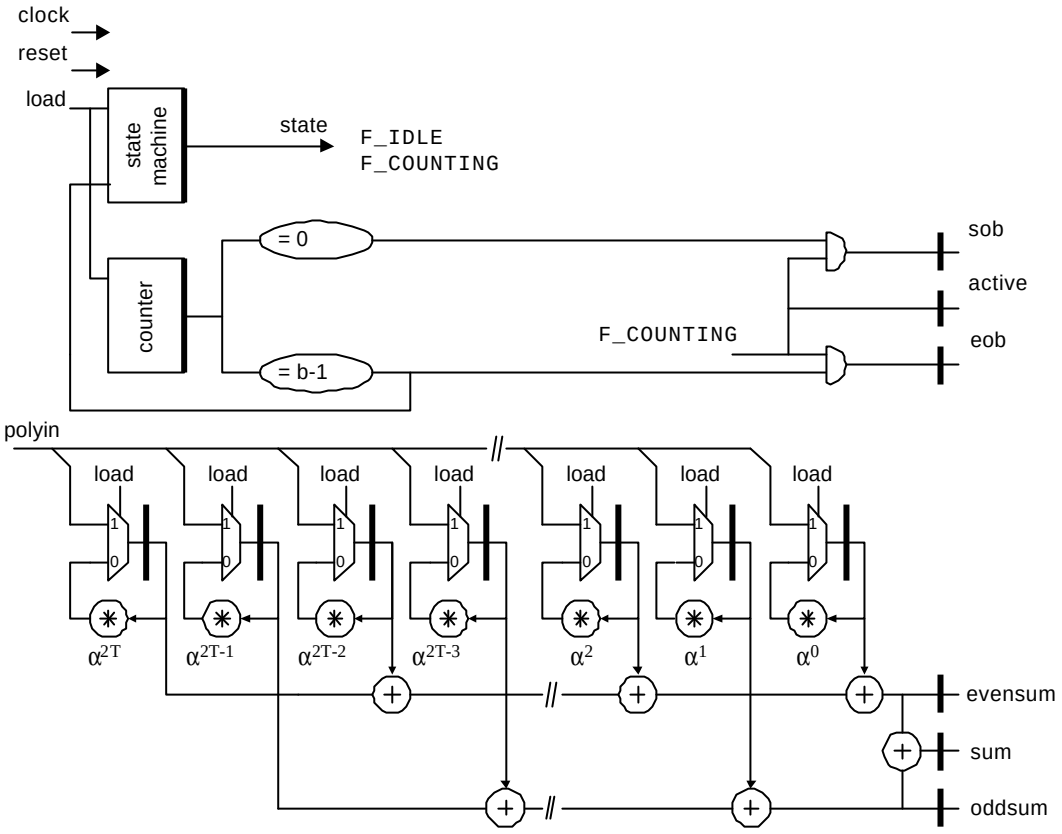


Figure 18 - Polynomial evaluation block diagram

4.10.3 Operation

The state table for the polyeval block is shown below:

state	*count*	*comment*
`F_IDLE`	x	Stay in this state until load asserted.
`F_COUNTING`	0	Stay in this state for B cycles while Chien search is being performed. Polynomial is evaluated at B different values.
`F_COUNTING`	1	
…	…	
`F_COUNTING`	B - 1	
`F_IDLE` or `F_COUNTING`	x or 0	Loop back to `f_counting` if load asserted immediately, else return to `f_idle`.

Table 7 - State table for polynomial evaluation block

The latency of this block (assuming no stall cycles) is 2 cycles.

4.11 Forney block

4.11.1 Algorithm

The Chien search involves simply evaluating $s(x)$, $w(x)$ and $L(x)$ for $x \in \{a^{-(B-1)} \cdots a^{-(0)}\}$. From these values, the Forney equations are used to actually calculate the error magnitudes.

The general form of the Forney equations is:

If $s(x) = 0$ for some $x = a^{-i}$ an error has occurred in symbol i, and the error magnitude is given by:

$$e_i = \frac{w(x)}{s'(x) \cdot \Lambda(x)} \text{ for } x = a^{-i}$$

If $\Lambda(x) = 0$ for some $x = a^{-i}$ an erasure has occurred in symbol i, and the erasure magnitude is given by:

$$E_i = \frac{w(x)}{s(x) \cdot \Lambda'(x)} \text{ for } x = a^{-i}$$

LEMMA 1: If $s(x) = 0$ then it is possible to obtain $xs'(x)$ by summing either the odd or even power terms of $s(x)$.

PROOF:

We can write $s(x)$ as:

$$s(x) = s_{2T}x^{2T} + s_{2T-1}x^{2T-1} + \cdots + s_3x^3 + s_2x^2 + s_1x + s_0$$

The derivative of $s(x)$ is:

$$s'(x) = 2Ts_{2T}x^{2T-1} + (2T-1)s_{2T-1}x^{2T-2} + \cdots + 3s_3x^2 + 2s_2x + s_1$$

Because we are working in a Galois field, the following hold true:

$$(2n)s_{i+1}x^i = (n+n)s_{i+1}x^i = (0)s_{i+1}x^i = 0$$

$$(2n+1)s_{i+1}x^i = (n+n+1)s_{i+1}x^i = (0+1)s_{i+1}x^i = s_{i+1}x^i$$

Therefore, $s'(x)$ can be simplified to:

$$s'(x) = s_{2T-1}x^{2T-2} + \cdots + s_3x^2 + s_1$$

and so,

$$xs'(x) = s_{2T-1}x^{2T-1} + \cdots + s_3x^3 + s_1x$$

Thus, we can obtain $xs'(x)$ by simply summing the odd terms of $s(x)$.

Observe also that we are only interested in $s'(x)$ where $s(x) = 0$. This means that the sum of the odd terms must equal the sum of the even terms.

Hence, we can also obtain $xs'(x)$ by simply summing the even terms of $s(x)$.

LEMMA 2: The scale factors resulting from the misalignment the polynomials *s(x)* and *w(x)* when loaded into the polynomial evaluation block can be easily determined:

PROOF:

At the start of the Euclidean computation:

$$w_T(x) = x^{2T}$$
$$\therefore \deg w_T(x) = 2T$$
$$w_B(x) = T(x)$$
$$\therefore \deg w_B(x) = 2T - 1$$

At the end of the Euclidean computation, the final degrees of the polynomials are:

$$\deg w_T(x) = 2T - d_T$$
$$\deg w_B(x) = 2T - 1 - d_B$$
$$\deg s_T(x) = d_T$$
$$\deg s_B(x) = 1 + d_B$$

Each iteration of the computation can only increase d_T or d_B by one, it follows that after *2t - J* iterations:

$$d_T + d_B = 2T - J$$
$$d_B = 2T - J - d_T \quad .$$

Thus, at the end of the computation:

$$\begin{aligned}\deg w(x) &= \deg w_B(x) \\ &= 2T - 1 - d_B \\ &= 2T - 1 - (2T - J - d_T) \\ &= d_T + J - 1\end{aligned}$$
$$\begin{aligned}\deg s(x) &= \deg s_T(x) \\ &= d_T\end{aligned}$$

When polynomials *s(x)* and *w(x)* are evaluated, we avoid shifting them to the correct position, hence a scale factor is included. More specifically, if the degree of the polynomial is d, then the scale factor is x^{2T-d}. Therefore,

$$\vec{w}(x) = x^{2T-(d_T+J-1)} w(x)$$

$$\therefore w(x) = \frac{\vec{w}(x)}{x^{2T-(d_T+J-1)}}$$

$$\vec{s}(x) = x^{2T-d_T} s(x)$$

$$\therefore s(x) = \frac{\vec{s}(x)}{x^{2T-d_T}}$$

also

$$xs'(x) = \frac{\vec{s}_{ODD\_OR\_EVEN}(x)}{x^{2T-d_T}}$$

$$\therefore s'(x) = \frac{\vec{s}_{ODD\_OR\_EVEN}(x)}{x^{2T+1-d_T}}$$

Also, as

Using the results of Lemma 1 and Lemma 2 we can re-write the Forney equations:

$$e_i = \frac{w(x)}{s'(x) \cdot \Lambda(x)}$$

$$= \frac{\vec{w}(x)}{x^{2T-(d_T+J-1)}} \cdot \frac{x^{2T+1-d_T}}{\vec{s}_{ODD\_OR\_EVEN}(x)} \cdot \frac{1}{\Lambda(x)}$$

$$= \frac{x^J \cdot \vec{w}(x)}{\vec{s}_{ODD\_OR\_EVEN}(x) \cdot \Lambda(x)}$$

and similarly:

$$E_i = \frac{w(x)}{s(x) \cdot \Lambda'(x)}$$

$$= \frac{\vec{w}(x)}{x^{2T-(d_T+J-1)}} \cdot \frac{x^{2T-d_T}}{\vec{s}(x)} \cdot \frac{x}{\Lambda_{ODD\_OR\_EVEN}(x)}$$

$$= \frac{xJ \cdot \vec{w}(x^{-i})}{\vec{s}(x) \cdot \Lambda_{ODD\_OR\_EVEL}(x)}$$

These equations are directly implemented.

The other function implemented in the Forney block is the detection of uncorrectable error patterns.

Let *nerasures* be the number of symbols declared as erasures.

Let *nerrors* be the number of distinct roots of $s(x)$ for $x \in \{a^{-(B-1)} \cdots a^{-(0)}\}$

If any of the following conditions arise, then the error pattern is declared uncorrectable:

- The Euclidean computation terminated mid-division (i.e. bottom comma is aligned with, or to the right of, the top comma).
- *nerasures* exceeds the decoder *maxerasures* input.
- *nerrors* differs from the true degree of $s(x)$.
- *nerasures* + 2 * *nerrors* > 2T
- A root of $s(x)$ co-incides with a root of $\Lambda(x)$ (i.e. the same location is both an error and an erasure)

4.11.2 Block diagram

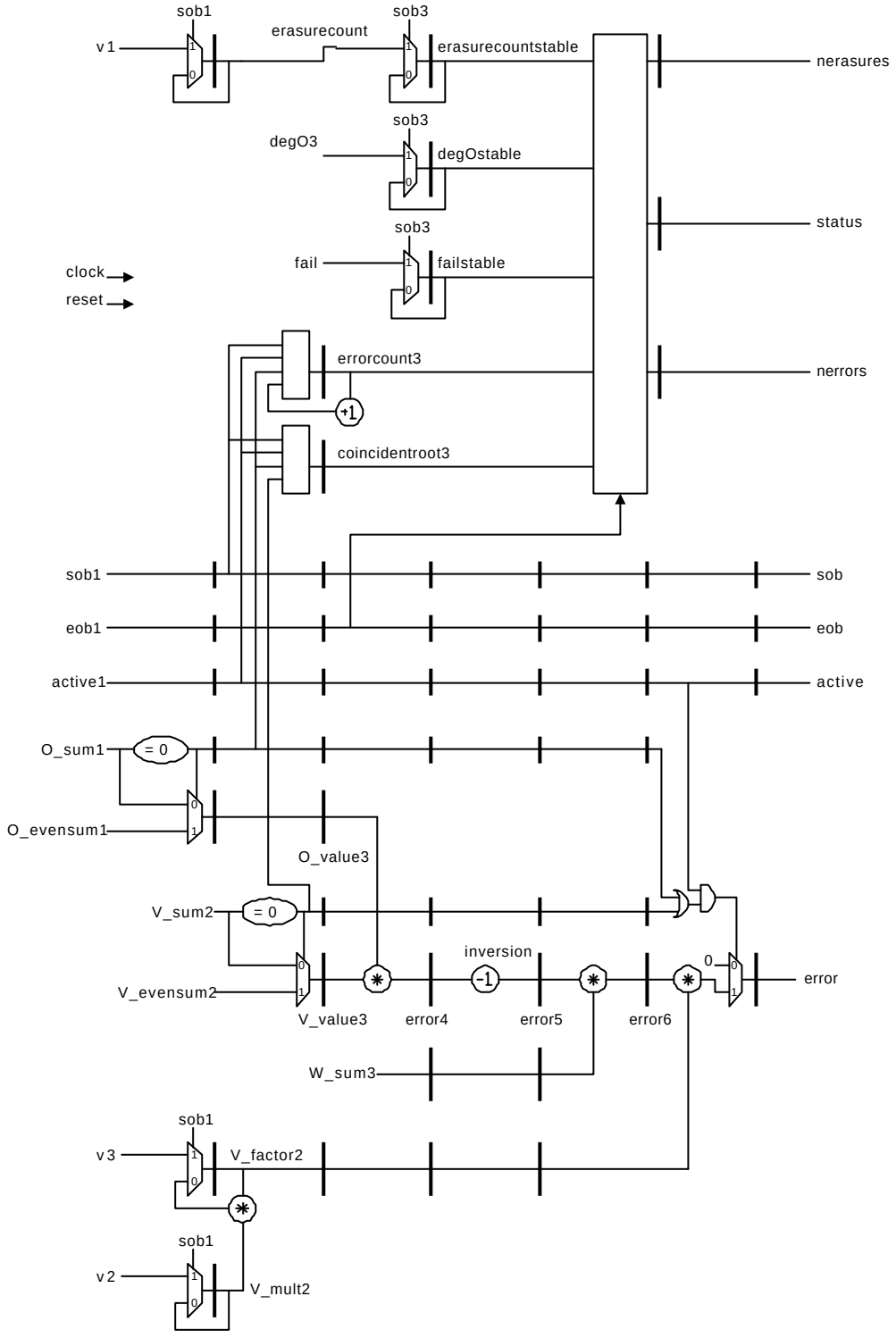


Figure 19 - Forney block diagram

4.11.3 Operation

This block is implemented as a heavily pipelined datapath, driven by the three polynomial evaluation blocks for $s(x)$, $L(x)$ and $w(x)$. There is one cycle skew between each of these blocks, due to the scaler, thus the results feed into the datapath at different stages.

The first multiplier corresponds to the multiplication on the denominator of the Fourney equations. The arguments are selected according to whether this symbol is an error or an erasure (it cannot be both). This is followed by an inversion and then two further multiplications, one to multiply in *w(x)* and the other to multiply in the correction factor x^J. The final multiplexor ensures that a zero error value is output when there is no error or erasure.

The erasurelist block pre-computes the following values (for *J* erasures)

$$v1 = J$$

$$v2 = \alpha^{J}$$

$$v3 = \alpha^{-(B-1)J}$$

The correction factor v_factor2 is calculated recursively, the sequence being:

$$\alpha^{-(B-1)J}, \alpha^{-(B-2)J}, \alpha^{-(B-3)J}, \cdots \alpha^{-2J}, \alpha^{-J}, 1$$

It can be seen by inspection that this corresponds to x^J for $x \in \{\alpha^{-(B-1)} \cdots \alpha^{-(0)}\}$.

Some brief comments on the timing constraints:

The signals v1, v2, v3 are generated by the erasurelist block and change when it's done signal is asserted. They are then held for a minimum of B cycles. The Forney block samples them on sob1, thus:

Constraint 1: erasurelist.done ⇒ forney.sob1 ≤ B cycles
(2T + 1) + (2TL/N + 1) + 4 ≤ B cycles

The signals degO3, fail are generated by the Euclid block and change when it's done signal is asserted. They are them held for a minimum of B cycles. The Forney block samples them on sob3, thus:

Constraint 2: Euclid.done ⇒ forney.sob3 ≤ B cycles.
(2TL/N + 1) + 6 ≤ B cycles

Generally constraint 1 will be the limiting factor.

The nerasures, nerrors and status outputs of the Forney block change on eob3, and thus are valid 3 cycles prior to the eob output. They are then held for a minimum of B cycles.

The latency of this block (assuming no stall cycles) is 6 cycles.

4.12 Symbol delay block

This block is trivial – it introduces a delay on the symbol data, to compensate for the delay through the previous blocks in the decoder. It is implemented as a symbol-wide shift register.

This needs to include:

- B+2 stages to compensate for the syndrome block
- 2T+1 stages to compensate for the polynomial expander block

- (2TL/N)+1 stages to compensate for the Euclid/delay blocks
- 2 stages to compensate for the scaler block
- 2 stages to compensate for the polynomial evaluation block
- 6 stages to compensate for the Forney block

Totalling these up yields B+2T+(2TL/N)+14 stages.

For B=160, T=16. L=36 and N=12 this works out at 302 stages.

4.13 Error correction block

This block is trivial – error correction is done by XORing the delayed input data with the error output of the Forney block.

The latency of this block (assuming no stall cycles) is 1 cycle.

4.14 Monitor block

4.14.1 Algorithm

As an additional check, the decoder re-calculates the syndromes over each sequence of symbols output by the decoder. This check is performed by the final pipeline stage within the decoder, called the *monitor* block.

If the status code was 0 to 3, the sequence of symbols output by the decoder should always correspond to a valid codeword (i.e. the syndromes will be zero). If this is not the case, the status code is replaced with 6.

If the status code was 4 or 5, the sequence of symbols output by the decoder is unlikely to be a valid codeword (i.e. one or more of the syndromes should be non-zero). If this is not the case, the status code is replaced with 7.

The status codes 6 and 7 should always be treated as uncorrectable.

4.14.2 Block diagram

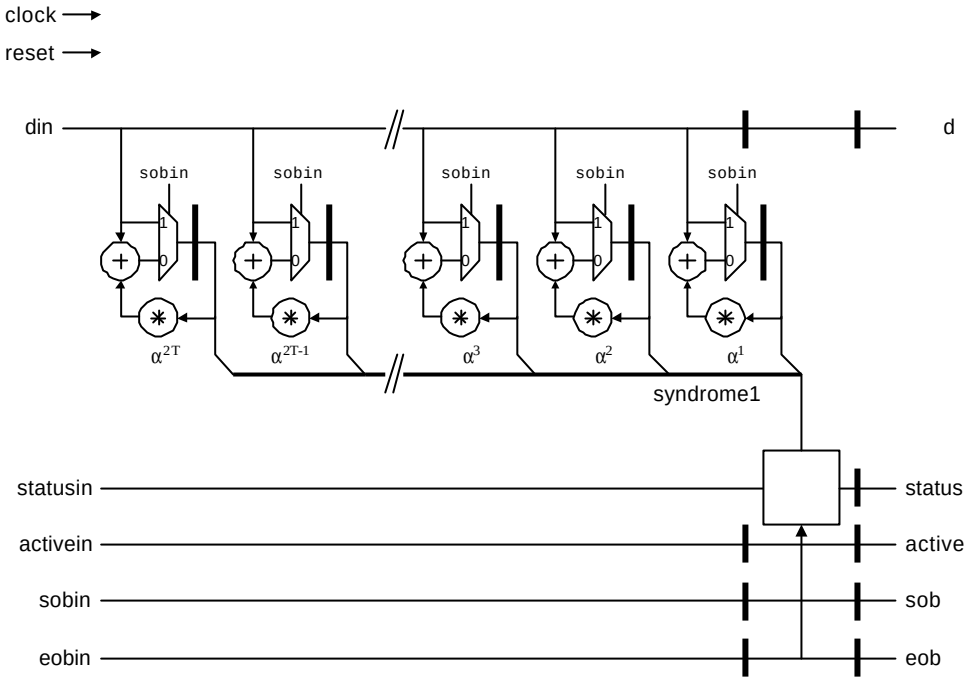


Figure 20 - Monitor block diagram

4.14.3 Operation

This block is implemented as a pipelined datapath. The status code from the Forney block is modified as follows:

```
if (eob1 == 1)
   if ((statusin < 4) && (syndrome1 != 0))
      status <= 6;
   else if ((statusin >= 4) && (syndrome1 == 0))
      status <= 7;
   else
      status <= statusin;
```

The latency of this block (assuming no stall cycles) is 2 cycles.

5 Synthesis

5.1 Source file layout

ReadMe

```
rs/ReadMe
```

Parameter configuration

```
rs/params.v
```

Verilog source files

```
rs/encoder.v
rs/decoder.v
rs/delay.v
rs/erasurelist.v
rs/euclid.v
rs/expander.v
rs/fourney.v
rs/messagedata
rs/monitor.v
rs/polyeval.v
rs/scaler.v
rs/symboldelay.v
rs/syndrome.v
rs/EuclidCell.v
rs/EuclidCell_fn.v
rs/EuclidCell_fn_body.v
rs/GFAdd.v
rs/GFAdd_fn.v
rs/GFAdd_fn_body.v
rs/GFInverse.v
rs/GFInverse_fn.v
rs/GFInverse_fn_body.v
rs/GFMult.v
rs/GFMult_fn.v
rs/GFMult_fn_body.v
rs/GFPtoT_fn.v
```

Synthesis control scripts

```
rs/RUNSYN
rs/reedsolomon.script
```

Galois arithmetic synthetic library

```
galois/GALOIS_GFAdd_mod.v
galois/GALOIS_GFMult_mod.v
galois/analyze.script
galois/galois.sl
galois/galois.sldb
```

5.2 Configuring the design

The design configuration is contained in the `params.v` file.

5.2.1 Parameters

The following parameters define the specific Reed-Solomon code:

- T - The code error correction capability.

- B - The code length.
- WIDTH - The width (in bits) of a code symbol.
- PRIMITIVE - The primitive field generator polynomial for the code. The binary representation of this number is used to form the field generator polynomial.
- GENERATOR - The code generator polynomial whose first root must be α. This value can be calculated using the `Generate.c` program.

The following parameters configure the layout of the registers in the Euclid block, as described in section 4.7.3.

- L - The number of logical stages in the Euclid block.
- N - The number of physical stages in the Euclid block.

For example, for an RS(160, 128, T=16) code over the galois field $GF(2^8)$ generated from $p(x) = x^8 + x^4 + x^3 + x^2 + 1 = 0$ the correct values are:

```
L = 36
N = 12
T = 16
WIDTH = 8
PRIMITIVE = 285
B = 160
GENERATOR = 256'he81dbd328ef6e80f2b52a4ee019e0d77
                9ee086e3d2a3326b281b68fd18efd82d
```

There are several constraints on L and N:

- L must be even
- L must be ≥ 2 T + 2 (the size of Gadiels array)
- N must be less than L
- N must be a factor of L
- (2T + 1) + (2TL/N + 1) + 4 ≤ B cycles (see constraint 1 in section 4.11.3)
- (2TL/N + 1) + 6 ≤ B cycles (see constraint 2 in section 4.11.3)

Example1: B = 160, T = 16, L = 36, N = 12

⇨ 134 ≤ 160

⇨ this is acceptable

Example2: B = 160, T = 16, L = 36, N = 9

⇨ 165 > 160

⇨ this is unacceptable (breaks constraint 1)

5.2.2 Clock enable

To configure the design with a synchronous clock enable, define the following macro in the `params.v` file:

```
`define ALWAYS_AT_POSEDGE_CLOCK always @(posedge clock) if (clocken == 1)
```

To configure the design without a synchronous clock enable, define the following macro in the `params.v` file:

```
`define ALWAYS_AT_POSEDGE_CLOCK always @(posedge clock)
```

De-asserting the clock enable essentially freezes the state of the whole design, rather like a gated clock.

Note that including a synchronous clock enable can add considerably (10%-20%) to the overall area, since a 2-input multiplexor needs to be added to the front of each flip-flop. This overhead might be reduced if the target ASIC library includes flip-flops with a built-in clock enable.

5.2.3 Synthetic libraries

The Galois field addition and multiplication operator implementations supplied from a used defined synthetic library. This has two advantages:

i. A level of hierarchy is created automatically for each synthetic operator, thus reducing the number of gates at any one level. This has a dramatic (approximately 80%) reduction in synthesis time.
ii. Constants are automatically propagated into these operators, allowing constant multipliers to be optimised automatically (as described in section 2.3.3)

A verilog function is mapped to a synthetic operator using the Synopsys `map_to_operator` directive:

```
function [WIDTH - 1 : 0] GFAdd_fn;
// synopsys map_to_operator gfadd_op
// synopsys return_port_name x
`include "GFAdd_fn_body.v"

function [WIDTH - 1 : 0] GFMult_fn;
// synopsys map_to_operator gfmult_op
// synopsys return_port_name x
`include "GFMult_fn_body.v"
```

An alternative is to compile the adder and multiplier as standalone modules, and then use the Synopsys `map_to_module` directive. The `-boundary_optimization` flag to the Synopsys compile command should be used, as in this case constant propagation does not occur automatically. The results achieved are similar, but with an increased compile time.

5.3 Synthesising the design

5.3.1 Build script

To synthesise the design, make sure you have a valid `.synopsys_dc.setup` file in your home directory. Then execute the following:

```
cd galois
./dc_shell –f analyze.script
cd rs
mkdir WORK
./RUNSYN
cd run_<date>
cat errors.txt (there should be none)
cat warnings.txt (there will be a few)
```

For reference, here is the current `RUNSYN` file:

```
#!/bin/csh

# create a results directory
```

```
set dir=run_`date +"%d%h%y_%H%M"`

# run synopsys
dc_shell -f reedsolomon.script | tee build.log

# move results files to results directory
mkdir $dir
mv build.log command.log $dir
mv *.area *.timing *.routing *.cells *.db *.vg $dir

# perform some post processing of results
cd $dir

echo Checking for errors:
grep "Error" build.log | tee errors.log

echo Checking for warnings:
grep Warning build.log | tee warnings.log

printf "%-12s %10s %10s %10s %10s %12s\n" "module" "comb area" "reg area" "net
area" "total area" "timing" | tee summary.log

foreach file ( `/bin/ls *.area` )

   set i=`echo $file | cut -d'.' -f1`
   set name=`echo $i | cut -d'_' -f1`

   set a1=`cat ${i}.area | grep "Combinational area" | cut -d':' -f2 | cut -
d'.' -f1 | awk '{print $1}'`
   set a2=`cat ${i}.area | grep "Noncombinational area" | cut -d':' -f2 | cut -
d'.' -f1 | awk '{print $1}'`
   set a3=`cat ${i}.area | grep "Net Interconnect area" | cut -d':' -f2 | cut -
d'.' -f1 | awk '{print $1}'`
   set a4=`cat ${i}.area | grep "Total cell area" | cut -d':' -f2 | cut -d'.' -
f1 | awk '{print $1}'`
   set cp=`cat ${i}.timing | grep "data arrival time" | head -1 | awk '{print
$4}'`

   printf "%-12s %10s %10s %10s %10s %12s\n" $name $a1 $a2 $a3 $a4 $cp | tee -a
summary.log

end
```

For reference, here is the current `reedsolomon.script` file:

```
/*
 * EQN-10 - warning: Defining new variable
 * VAL-3  - warning: Parameter/generic value exceeds the threshold length 20
 */
suppress_errors = { EQN-10 VAL-3 }
high_fanout_net_threshold = 0
search_path = search_path + ../galois
define_design_lib GALOIS -path ../galois
synthetic_library = synthetic_library + "galois.sldb"
link_library = link_library  + "galois.sldb"
define_design_lib WORK -path ./WORK
hlo_resource_allocation = none
hlo_resource_implementation = area_only

foreach (DESIGN, { GFAdd, GFMult, GFInverse } ) {
   analyze -format verilog DESIGN + ".v"
   elaborate DESIGN
   set_max_area 0
   set_fix_multiple_port_nets -all
   check_design
   compile
}
foreach (DESIGN, { EuclidCell } ) {
   analyze -format verilog DESIGN + ".v"
   elaborate DESIGN
   set_max_area 0
   set_fix_multiple_port_nets -all
   uniquify
   check_design
   compile
```

```
   set_dont_touch current_design
}
foreach (DESIGN, {  delay polyeval expander erasurelist scaler fourney monitor
syndrome euclid symboldelay decoder encoder }) {
    analyze -format verilog DESIGN + ".v"
}
foreach (DESIGN, { encoder decoder } ) {
   elaborate DESIGN
   uniquify
   create_clock -period 1000 clock
   set_max_area 0
   set_fix_multiple_port_nets -all
   check_design
   compile
   write -format db -hierarchy -output DESIGN + ".db"
   write -format verilog -hierarchy -output DESIGN + ".vg"
   report_timing -nets > DESIGN + ".timing"
   report_area > DESIGN + ".area"
   report_routability > DESIGN + ".routing"
   report_cell > DESIGN + ".cells"
}
d1 = "syndrome"
d2 = "erasurelist"
d3 = "expander"
d4 = "euclid"
d5 = "delay"
d6 = "euclid"
d7 = "scaler"
d8 = "polyeval_0"
d9 = "polyeval_1"
d10 = "polyeval_2"
d11 = "fourney"
d12 = "symboldelay"
d13 = "monitor"
foreach (DESIGN, { d1 d2 d3 d4 d5 d6 d7 d8 d9 d10 d11 d12 d13 }) {
   echo DESIGN
   current_design DESIGN
   report_timing -nets > DESIGN + ".timing"
   report_area > DESIGN + ".area"
   report_routability > DESIGN + ".routing"
   report_cell > DESIGN + ".cells"
}
quit
```

5.3.2 Results

The following results were obtained for the RS(160, 128, T=16) code, targeting the Agere MACO libraries:

submodule	*comb area*	*reg area*	*net area*	*total area*	*timing*
delay	175	160	11	335	0.34
erasurelist	3619	6624	352	10243	0.39
euclid	31158	13128	1794	44286	0.68
expander	30819	8625	1390	39444	1.08
fourney	7235	3360	422	10595	0.39
monitor	11253	4496	487	15749	0.39
polyeval	11360	4993	500	16353	0.34
polyeval	11360	4993	500	16353	0.34
polyeval	11360	4993	500	16353	0.34
scaler	17154	13040	1006	30194	0.34
symboldelay	16912	38656	1329	55568	0.34
syndrome	10948	4552	481	15500	1.38

module	*comb area*	*reg area*	*net area*	*total area*	*timing*
encoder	13299	4560	538	17859	6.78
decoder	163531	107796	8615	271327	13.78

The area figures are in grids (an Agere metric). For the MACO process, the gate density is quite low (2.55 gates/grid) and so the apparent gate counts are quite high. The encoder comes out at 7.0K gates and the decoder comes out at 106K gates.

Moving to their HL200CDE standard cell library, with a gate density of 3.23 gates/grid improves matters. The encoder comes out at 5.5K gates and the decoder comes out at 84K gates.

If the synchronous clock enable is removed, and the standard cell library used, the area is significantly less. The encoder comes out at 5.0K gates and the decoder comes out at 73K gates.

6 References

[1] Verification of a Synthesisable Reed-Solomon ECC Core, HPL Technical Report HPL-2001-125, David Banks, May 2001.
[2] *Introduction to finite fields and their applications,* Rudolf Lidl and Harald Niederreiter, Cambridge University Press, 1994.
[3] *Error Control Coding*, Shu Lin and Daniel J Costello, Jr., Prentice Hall, 1983.
[4] *Reed-Solomon codes and their applications*, edited by Stephen B Wicker and Vijay K Bhargava, IEEE Press, 1994.