BBCH Encoder and Decoder

Programmable to t=18

Preview

Final Document

(Customized to Meet Customer’s Requirements)

Version 1.4

By

ECC Technologies, Inc. (“ECC Tek”)
4750 Coventry Road East
Minnetonka, MN 55345-3909
www.ecctek.com
Phone: 952-935-2885
Fax: 952-935-2491
Email: phil.white@ecctek.com

Notice

ECC Technologies, Inc. (ECC Tek)
4750 Coventry Road East
Minnetonka, MN 55345-3909

Phone:	952-935-2885
Fax:	952-935-2491
E-mail:	phil.white@ecctek.com
Website:	www.ecctek.com.

Revision History

02-06-07	pew	Created initial version of this document based on previous documents and drawings.
02-26-07	pew	Edited and added material.
02-27-07	pew	Edited and added material.
04-04-07	pew	Updated the EVAL block diagram.
04-05-07	pew	Edited the material.
04-06-07	pew	Updated the SYNB block diagram.
12-12-07	pew	Added Encoder and Decoder Signal Definition Tables
12-21-07	pew	Added a column to the Signal Definition Tables to show what the original signal names are.
02-16-08	pew	Corrected the INIT block diagram.
03-21-08	pew	Changed some of the heading levels.
03-21-08	pew	Corrected the K values selectable.
03-21-08	pew	Added material about when DECODERE will and will not work well.
03-23-08	pew	Added first two sections.
09-18-08	pew	Added section on Decoder Latency.

Sections

Figures

Tables

1. Introduction

ECC Tek is in the business of licensing synthesizable Verilog code which describes the design of ECC encoders and decoders at a very high level. Licensing synthesizable Verilog presents a unique set of challenges.

Since our customers do synthesis, ECC Tek has no control over what synthesis tools are used or how they are used. ECC Tek does not know what its customer's synthesis tools will do in optimizing logic or if one customer’s tools will do the same things or different things than another customer’s tools.

For example, suppose ECC Tek creates a number of Verilog modules so that the output of one module feeds the input of another. Assume we have "upstream" and "downstream" modules. If a downstream module forces a signal to a constant value, will the synthesis software, in all cases, be intelligent enough to know that certain upstream logic can be deleted because it is unneeded? ECC Tek has no way of knowing the answer to that question.

Another example is a Galois Field multiplier. If ECC Tek designs a GF multiplier to multiply two variables, will all of the synthesis software tools reduce and minimize the logic needed if one of the inputs is a constant or should ECC Tek design specific multipliers to multiply by specific constants? ECC Tek has no way of knowing the answer to this question either. Most likely, most synthesis software will properly reduce the logic, but, because of that uncertainty, ECC Tek has created many different multipliers that multiply by a constant such as the ones delivered to Customer called MULT_BY_XXXX.

2. Organization of this Document and Additional Documentation

Section 3 was added at the beginning of this document to describe the most recent items delivered to Customer rather than insert that material into the older parts of the document because ECC Tek believes that will be most useful for Customer.

Documenting the binary BCH designs is difficult because there are many implementation options.

3. Items Recently Delivered to Customer

This section summarizes the most recent items delivered to Customer and a list of issues encountered and how they were resolved. For a complete description of all of the items developed by ECC Tek and delivered to Customer, see Sections 5-32.

3.1. Encoder Versions

A block diagram of ENCODERA delivered to Customer is shown in Figure 1.

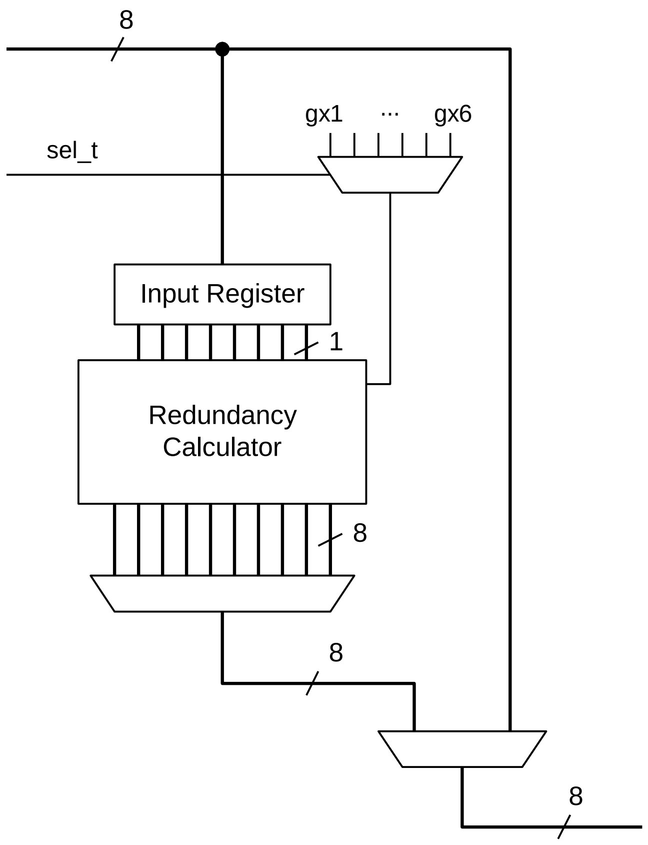


Figure 1 ENCODERA

3.2. Encoder Issues

There are no known encoder issues.

3.3. Decoder Versions

ECC Tek delivered the following six decoder versions to Customer: DECODERA, DECODERB, DECODERC, DECODERD, DECODERE and DECODERF. Customer should use one of these six versions.

3.3.1. DECODERA

A block diagram of DECODERA is shown in Figure 2.

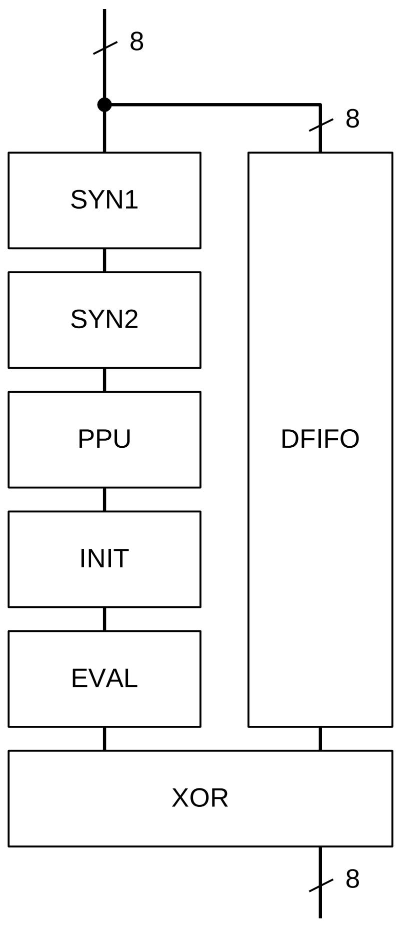


Figure 2 DECODERA, DECODERB, DECODERC and DECODERD

DECODERA uses a DFIFO model which correctly models Customer's dual-port RAM FIFO, contains changes to improve the way the pause signal is handled, and uses a k_delta input signal instead of sel_k so that 21 K values (512 to 534) are selectable rather than just 3 K values (514, 515 and 516) as in previous versions of the decoder.

The pause function was improved in DECODERA by observing that the only modules which must be clocked with the output clock and therefore need to be paused are EVAL, XOR and the reading of the DFIFO. In earlier versions of the decoder, INIT was also paused.

To improve the pause behavior, the pause signal was removed from INIT and changes were made so the PPU and INIT modules hold their outputs after setting their output go bits and stay in a WAIT state until the next section of logic clears the go bits.

Although DECODERA functions correctly with the pause signal, ECC Tek will not support the pause signal implemented inside the decoder for a number of reasons including the difficulty in specifying a precise rule as to how wide the aggregate width of pause signals can be. This difficulty and other difficulties can easily be remedied by implementing the pause function in an external PAUSE module at the output of the decoder. For future customers who require a pause signal, that is how it will be implemented, and that is the only method ECC Tek will support.

In Customer's case, Customer implemented the pause function simply by gating the output clock signal.

3.3.2. DECODERB

DECODERB contains a minor change to DECODERA to try to further improve the pause function, but it did not work. In DECODERB, the SYN2 control state machine was modified to wait for its go bit to be cleared by the PPU before continuing operation. The change had no effect on how the pause signal functioned because the decoder failed before SYN2 started to wait.

Although this change did not have the desired effect, it does no harm, so it was not removed in subsequent versions of SYN2.

3.3.3. DECODERC

DECODERC is the same as DECODERB but with the pause signal completely removed from the Verilog code. The same effect can be achieved through synthesis by fixing the pause signal in DECODERB at 0.

Because ECC Tek will not support an internal pause signal in future decoder designs, ECC Tek recommends that Customer implement either DECODERC, DECODERD, DECODERE or DECODERF.

3.3.4. DECODERD

DECODERD is the same as DECODERC but with a slower and reduced gate count PPUD installed instead of PPUC. DECODERD may be the best version to use for the t=8 applications, however, it requires a larger DFIFO than DECODERC because of the added time it takes PPUD to form the error locator polynomial.

DECODERD can only be used for t values of 15 or less if the input data is continuous because of the length of time it takes PPUD to form the error locator polynomial, L(x), if t=17 or 18.

If gate count is not a major consideration, it probably would be best to stick with the faster PPUC rather than use PPUD since there are no restrictions on t in that case.

3.3.5. DECODERE

DECODERE was created to reduce the time to calculate the syndrome which reduces the latency/delay of the decoder. DECODERE uses SYNB to replace SYN1 and SYN2 as shown in Figure 3.

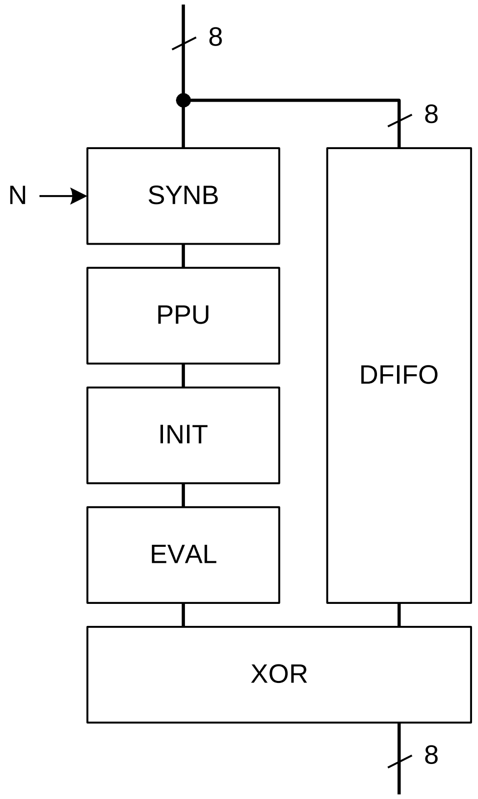


Figure 3 DECODERE

SYNB can handle any N, and has N as an input. SYNB calculates 36 syndrome values, but not all of those values are needed when t < 18.

ECC Tek believes that intelligent synthesis software will delete unneeded logic described in SYNB when t < 18 because downstream logic forces unneeded syndrome values to 0.

However, ECC Tek created a module called SYNBT8 which is SYNB customized for t=8 cases. For the t=8 cases, only 2t=16 syndrome values need to be calculated so SYNBT8 only describes logic to generate the first 16 syndrome values S1-S16. The syndrome values S17-S36 are set to all 0's in SYNBT8.

DECODERE will work with either SYNB or SYNBT8.

The latency of DECODERE with 0 errors is 546-N + 7 clock cycles.

3.3.6. DECODERF

DECODERF was created to eliminate the 546-N clock cycles it takes to initialize L(x) in previous versions. In previous versions, as N decreased, the initialization time increased. A new module called INITN was developed to initialize the error locator polynomial, L(x), as it is being passed from the PPU to EVAL so the initialization of L(x) takes 0 clock cycles as shown in Figure 4.

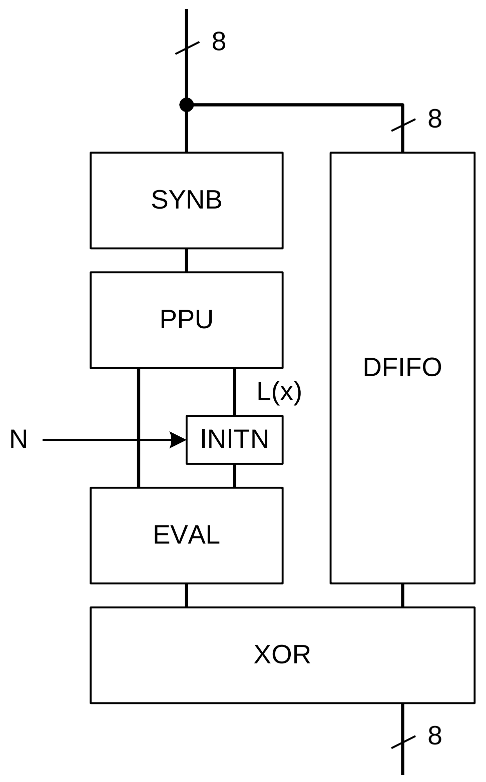


Figure 4 DECODERF

The latency of DECODERF with 0 errors is 6 clock cycles.

DECODERF should work well for those customers such as Customer who wish to synthesize the Verilog code into a circuit for one (t, K) setting because intelligent synthesis software “should” delete unneeded circuits.

DECODERF probably will not work for those customers who wish to synthesize the Verilog code into a programmable circuit so that t and K are selected by input settings because INITN will probably require too many gates in that case and the previous method of initializing L(x) will probably need to be used.

3.3.7. Future DECODER Versions

It is easy to create additional versions of the decoder by combining different versions of the subcomponent modules.

3.4. Decoder Issues

3.4.1. Pause Signal Issue

ECC Tek decided not to support the pause function implemented inside the DECODER in future versions of its designs because of the complexity of specifying a general rule as to how long the aggregate width of pauses can be, difficulties in stopping and restarting logic and because there is a much simpler method of implementing the pause function.

If a pause feature is needed in future decoders, ECC Tek will implement it as an external PAUSE module at the output of the decoder. In cases where ECC Tek's customers can gate the output clock, the pause function may also be implemented by gating the output clock as Customer is doing.

If the output of the decoder is written to a small external Pause FIFO (PFIFO), the reading of the PFIFO can be paused occasionally for different lengths of time as long as the aggregate pause width is less than the depth of the PFIFO.

3.4.2. Decoder Clocks

ECC Tek's previous decoder designs have used only one clock, but it is possible to have separate and independent clocks for the input, the PPU and the output. If the clocks are not synchronized, then logic would have to be added to resynchronize signals that crossed clock domains.

Clocking the PPU at a higher rate would reduce the decoder latency/delay and reduce the DFIFO size needed.

Clocking the output with a separate clock allows the output to be paused by gating the clock.

3.4.3. Gating the Clocks

ECC Tek normally operates under the assumption that gating the clock is not acceptable. If gating the clock is acceptable, a number of options are opened up such as not clocking either the encoder or decoder if they are not being used and pausing the output by gating the output clock.

3.4.4. Customer Maintaining the ECC Designs

It is in Customer's best interest to assume responsibility for maintaining the designs as soon as possible because then Customer can easily make any changes it desires to the designs.

3.5. PPU Versions

3.5.1. PPUA

PPUA operates on four polynomial coefficients simultaneously and has a constant decoder delay.

There may be some advantage to having a constant decoder delay in some situations. With a constant decoder delay, you can know ahead of time that, after you put in the last uncorrected byte of a page, precisely how many clock cycles later the output of the decoder will start.

Since PPUA operates on four polynomial coefficients simultaneously, it can handle every t value.

Most likely, PPUA will not be implemented by Customer. If it is implemented, we would have to make sure it was thoroughly analyzed and tested since it was not used by ECC Tek's previous customers.

3.5.2. PPUB

PPUB operates on four polynomial coefficients simultaneously and has a variable delay depending upon the number of errors that actually occurred. This is probably the version Customer will implement. It can handle all t values.

3.5.3. PPUC

PPUC is a slightly modified version of PPUB. An output signal was added to clear the go bit which starts PPU execution.

3.5.4. PPUD

PPUD is a slowed down, reduced gate count version of PPUC but does not work with t=17 or t=18 if the input is continuous. It can handle continuous input data for $t \leq 15$ only.

PPUD operates on only two polynomial coefficients simultaneously, is 1/2 the speed of PPUC and requires a little less logic and wiring. PPUD should be easier to synthesize to meet a timing target than PPUC.

3.6. Programmable ECC Design

ECC Tek has created a programmable binary BCH encoder and decoder that Customer can synthesize with one set of (t, K) values and end up with a specific circuit for that case.

This methodology is similar, but different, from what our previous binary BCH customer did. Our previous customer synthesized a very large programmable circuit from the Verilog code and used inputs to select one of 18 (t, K) sets of values.

4. Decoder Latency

Decoder latency is defined as the number of clock cycles from the last input byte received by the decoder to the first byte outputted.

Most of the decoder versions described in this document have a variable latency. For those with a fixed latency, the latency is the maximum latency required by the variable latency version.

Let T be the number of errors that actually occur and t be the maximum number of errors the decoder is configured to correct.

The decoder latency is S + P + I + O where

S = the latency of the Syndrome Generator
P = the latency of the PPU
I = the latency of the Initialization section
O = the number of overhead cycles

S = 0 or r depending upon which syndrome generator is used where r is the number of "bits" of redundancy. S = 0 if SYNB is used. S = r if SYN1+SYN2 is used. r can vary from 96 to 240 so it takes a significant amount of time to calculate the syndrome if SYN1+SYN2 is used instead of SYNB. However, SYNB will probably require more chip surface area than SYN1+SYN2.

The latency of the PPU depends upon the number of errors the decoder is configured to correct, t, and the number of errors that actually occur, T. P = tT if PPUB or PPUC is used. P = T if PPUF is used. If T = 0, the latencies are equal, but if T = 18, for example, then the PPUF latency will be 18 while the PPUB or C latency will be 324 cycles. PPUF most likely will take much more area than PPUB or C. In general, the PPU latency is 2tT/P where P is the amount of parallelism in the PPU. For PPUB and C, P=2.

I=546-N where N is the number of "bytes" in a codeword. I could be reduced to 0, but it would take many more gates.

O is the overhead which is 12 cycles.

The above formula for decoder latency agrees with what was measured except for the end cases when T=t. The latency is slightly different when T=t because they are end cases. The case when t = T = 18 is slightly different from the other end cases because it is the last end case. These minor variations are due to the way the state machine controller is implemented in the PPU.

5. Overview of BBCH Encoder and Decoder Designs

This document describes digital logic designs for encoding and decoding Binary BCH (BBCH) codes which can correct t = 7, 8, 14, 15, 17, or 18 bits in error for use with NAND Flash memory chips with data field sizes of from K = 512, 513, …, 534 bytes. The encoder adds from 12 to 30 bytes of redundancy onto each data field. The decoder handles 79 (t, K) options where the total number of bytes in a codeword, N, is limited to 546 or less as shown in Table 3. The limitation on N could easily be changed if needed.

The BCH codes being used are binary and usually encoders for binary codes input and output 1 bit a time and syndrome calculators input 1 bit at a time. For this design the encoder and decoder input and output 8 bits a time and appear, from the outside, to be a Reed-Solomon (RS) encoder and decoder operating on 8-bit symbols.

Capital letters in the following Tables are variables measured in bytes and lower case letters are variables measured in bits.

Table 1 shows selectable values for t and what the number of redundant bits, r, before adjustment is made to make codewords an even number of bytes.

t (bits)	7	8	14	15	17	18

r (bits)	91	104	182	195	221	234

Table 1 Selectable Values for t and r before Adjustment

The number of bits of redundancy, r, added to a message for each value of t as shown in Table 1 was adjusted by multiplying the binary BCH generator polynomials by factors of low degree so that the adjusted amount of redundancy is a multiple of 8 and therefore is an even number of bytes which greatly simplifies the design of the encoder and decoder. Selectable values t and R are shown in Table 2.

t (bits)	7	8	14	15	17	18
R (bytes)	12	13	23	25	28	30

Table 2 Selectable Values for t and R

6. (t, K) Options

Customer can synthesize programmable versions of the encoder and decoder while fixing t and K and end up with a specific circuit for those (t, K) values.

There are 79 possible (t, K) circuit options as shown in Table 3 on the following page.

Table 3 (t, K) Options

t (bits)	**7**	**8**	**14**	**15**	**17**	**18**	
R (bytes)	**12**	**13**	**23**	**25**	**28**	**30**	
N (bytes)	524	525	535	537	540	542	K=512 (bytes)
N (bytes)	525	526	536	538	541	543	K=513 (bytes)
N (bytes)	526	527	537	539	542	544	K=514 (bytes)
N (bytes)	527	528	538	540	543	545	K=515 (bytes)
N (bytes)	528	529	539	541	544	546	K=516 (bytes)
N (bytes)	529	530	540	542	545	-	K=517 (bytes)
N (bytes)	530	531	541	543	546	-	K=518 (bytes)
N (bytes)	531	532	542	544	-	-	K=519 (bytes)
N (bytes)	532	533	543	545	-	-	K=520 (bytes)
N (bytes)	533	534	544	546	-	-	K=521 (bytes)
N (bytes)	534	535	545	-	-	-	K=522 (bytes)
N (bytes)	535	536	546	-	-	-	K=523 (bytes)
N (bytes)	536	537	-	-	-	-	K=524 (bytes)
N (bytes)	537	538	-	-	-	-	K=525 (bytes)
N (bytes)	538	539	-	-	-	-	K=526 (bytes)
N (bytes)	539	540	-	-	-	-	K=527 (bytes)
N (bytes)	540	541	-	-	-	-	K=528 (bytes)
N (bytes)	541	542	-	-	-	-	K=529 (bytes)
N (bytes)	542	543	-	-	-	-	K=530 (bytes)
N (bytes)	543	544	-	-	-	-	K=531 (bytes)
N (bytes)	544	545	-	-	-	-	K=532 (bytes)
N (bytes)	545	546	-	-	-	-	K=533 (bytes)
N (bytes)	546	-	-	-	-	-	K=534 (bytes)

7. Generator Polynomials

Generator polynomials used before adjustment are shown in the first version of the C code. Generator polynomials used after adjustment are shown in the last version of the C code.

Generator polynomials for binary BCH codes are products of Minimal Polynomials, Mi(x), whose roots are (α^i), $(\alpha^i)^2$, $(\alpha^i)^4$, $(\alpha^i)^{16}$, $(\alpha^i)^{32}$, ... , $(\alpha^i)^n$ where $n = 2^j$ for j=0,1,2 …J. This sequence of powers of (α^i) will repeat for some J and for that J, $(\alpha^i)^n = 1$. Sometimes this sequence will repeat sooner and the degree of g(x) will be less than wt, but in this case the degrees of g(x) are always wt where w is the width of the finite field elements used to locate the position of each bit. In this case w=13 and the degrees of g(x) before adjustment are 13t as can be seen in Table 1.

The irreducible binary polynomial used to generate a finite field with 2^{13} elements is
$p(x) = 201B$ hex $= x^{13} + x^4 + x^3 + x + 1$.

In the case with w=13, all of the elements in the finite or Galois Field with 2^{13} elements are primitive because $2^{13} - 1 = 8{,}191$ which is a prime number. In order for some of the elements in a finite field with 2^w elements to be nonprimitive, 2^w-1 must be divisible.

Since all the nonzero elements in GF(2^{13}) are primitive, any of them can be used as α. However, it is conventional to use “x” or 0002 hex as the primitive element, so that is what is used in the BBCH designs which is the same as for the RS designs.

In order for an error-correcting code to correct t errors, 2t consecutive powers of α must be roots of g(x). For binary BCH codes to correct t errors, the minimal polynomials M1, M3, …, M2t-1 must be factors of g(x). That is, g(x) = M1*M3*M5*…*M2t-1. For example, to correct 18 bits, M1, M3, …, M35 must be factors of g(x) and g(x), in that case, is

M1*M3*M5*M7*M9*M11*M13*M15*M17*M19*M21*M23*M25*M27*M29*M31*M33*M35.

Minimal polynomials Mi were computed using a C program by multiplying factors $(x-(\alpha^i))*(x-(\alpha^i)^2)$ $\ldots*(x-(\alpha^i)^n)$ and are shown in Table 4.

Top Row is Powers of α
Beneath them are $(\alpha^i)^{2i}$

$8191 \equiv 1$

Row#	M1	M3	M5	M7	M9	M11	M13	M15	M17	M19	M21	M23	M25	M27	M29	M31	M33	M35
1	$\boxed{1}$	$\boxed{3}$	$\boxed{5}$	$\boxed{7}$	$\boxed{9}$	$\boxed{11}$	$\boxed{13}$	$\boxed{15}$	$\boxed{17}$	$\boxed{19}$	$\boxed{21}$	$\boxed{23}$	$\boxed{25}$	$\boxed{27}$	$\boxed{29}$	$\boxed{31}$	$\boxed{33}$	$\boxed{35}$
2	$\boxed{2}$	$\boxed{6}$	$\boxed{10}$	$\boxed{14}$	$\boxed{18}$	$\boxed{22}$	$\boxed{26}$	$\boxed{30}$	$\boxed{34}$	38	42	46	50	54	58	62	66	70
3	$\boxed{4}$	$\boxed{12}$	$\boxed{20}$	$\boxed{28}$	$\boxed{36}$	44	52	60	68	76	84	92	100	108	116	124	132	140
4	$\boxed{8}$	$\boxed{24}$	40	56	72	88	104	120	136	152	168	184	200	216	232	248	264	280
5	$\boxed{16}$	48	80	112	144	176	208	240	272	304	336	368	400	432	464	496	528	560
6	$\boxed{32}$	96	160	224	288	352	416	480	544	608	672	736	800	864	928	992	1056	1120
7	64	192	320	448	576	704	832	960	1088	1216	1344	1472	1600	1728	1856	1984	2112	2240
8	128	384	640	896	1152	1408	1664	1920	2176	2432	2688	2944	3200	3456	3712	3968	4224	4480
9	256	768	1280	1792	2304	2816	3328	3840	4352	4864	5376	5888	6400	6912	7424	7936	257	769
10	512	1536	2560	3584	4608	5632	6656	7680	513	1537	2561	3585	4609	5633	6657	7681	514	1538
11	1024	3072	5120	7168	1025	3073	5121	7169	1026	3074	5122	7170	1027	3075	5123	7171	1028	3076
12	2048	6144	2049	6145	2050	6146	2051	6147	2052	6148	2053	6149	2054	6150	2055	6151	2056	6152
13	4096	4097	4098	4099	4100	4101	4102	4103	4104	4105	4106	4107	4108	4109	4110	4111	4112	4113
14	1	3	5	7	9	11	13	15	17	19	21	23	25	27	29	31	33	35

Table 4 Roots of Minimal Polynomials Mi

The roots of Mi are indicated below each Mi by showing the powers of alpha that are roots. For example M1 has roots 1, 2, 4, 8, etc. which means M1(x) has roots of α^1, α^2, α^4, …, α^{4096}. The root powers that are inside boxes are the roots that are required to correct t errors. The rest of the roots are not needed, but necessarily must be included because of the way binary BCH generator polynomials are constructed. For example, for the t=18 case, only 36 of the 234 roots of g(x) are actually needed to correct 18 bits, but the remaining 198 roots must also be roots of g(x) because of the way g(x) is constructed.

The coefficients of Minimal Polynomials are binary. That is, all of the coefficients are either 0 or 1 so that g(x) is also binary and all of the coefficients of g(x) are also either 0 or 1 which necessarily must be the case for binary codes.

In order to increase the amount of redundancy so it is a multiple of 8, g(x) was multiplied by the following irreducible factors.

For t=7, g(x) was multiplied by $x^5 + x^2 + 1$.
For t=14, g(x) was multiplied by $x^2 + x + 1$.

For t=15, g(x) was multiplied by $x^5 + x^2 + 1$.
For t=17, g(x) was multiplied by $x^3 + x + 1$.
For t=18, g(x) was multiplied by $x^6 + x + 1$.

Much of the BBCH decoder logic operates on 13-bit finite field elements internally, but data is input and output as 8-bit bytes.

8. Versions of the C Code and Verilog Code

The simulation and testing C code and the synthesizable Verilog code have been developed in versions.

Versions of the C software have name suffixes of V1, V2, V3, etc. For example the names are Customer binary BCH All cases V1.c, Customer binary BCH All cases V2.c, etc. Initially, 6 C programs were developed – one for each t value, but later they were combined into one “All cases” program.

Versions of the Verilog modules have module names with suffixes of A, B, C, etc., to distinguish the BBCH Verilog modules from the Reed-Solomon Verilog modules which use suffixes of 1, 2, 3, etc. For example, the first version of SYN1 is called SYN1A, the second version is called SYN1B, etc.

9. What This Document Does

This document describes BBCH encoder and decoder versions delivered to Customer which correct all error patterns with t or fewer bits in error and is intended to help the reader understand the BBCH encoder and decoder hardware designs which are described in synthesizable Verilog and the C modeling software so that the designs can be easily modified and maintained.

10. What This Document Does Not Do

This document does not often repeat data that can be found in the C or Verilog code. For example, readers are referred to the C code to view the generator polynomials rather than repeating all of them in this document.

This document is not intended to be a substitute for a textbook on coding theory. It takes a lot of time, effort and study to understand algebraic coding theory. There are many good books on coding theory which should be consulted to gain an in-depth understanding of the math and theory associated with Binary BCH codes. It would be impossible for this document to cover all of the topics that a good book on coding theory covers so no attempt is made to do that.

This document does not provide mathematical proofs of mathematical results that have already been proven in textbooks or papers on coding theory. However, the C and Verilog models of the BBCH encoder and decoder designs can be used to verify that the BBCH system does, in fact, correctly correct all correctable error patterns. The C and Verilog models can also be used to determine the frequency of miscorrection.

This document presents and uses a version of the well-known Berlekamp-Massey (“BM”) algorithm to solve the key decoding equation without presenting a proof of its validity. The BM algorithm has been proven to be valid in various textbooks and papers on coding theory. A version of the BM algorithm developed by Willard Eastman is used by ECC Tek in all of ECC Tek’s designs and is referred to as the “EBM” algorithm to stand for Eastman’s version of the Berlekamp-Massey algorithm.

The term “key equation” is more correctly called the “key congruency” but the term “key equation” is used in most of the textbooks on coding theory so it will be used in this document also. For binary BCH codes, the key equation is $S(x)L(x) \equiv V(x) \bmod x^{2t}$ or, in words, S(x) times L(x) is congruent to V(x) modulo x^{2t} where S(x) is the syndrome polynomial, L(x) is the error locator polynomial, V(x) is the error evaluator polynomial and t is the number of bits being corrected.

When operations are performed on polynomials modulo x^{13}, all of the terms resulting from a polynomial operation, such as multiplication of two polynomials, of the form $c_i x^i$ are 0 if i is greater than or equal to 13 because x^{13} is a factor of all terms of the form $c_i x^i$ when i is greater than or equal to 13 and x^{13} is congruent to 0.

It is well-known by coding theorists and stated in some textbooks on coding theory that the integer variable used in the EBM algorithm which ECC Tek refers to as “degLx” will be the actual degree of L(x) when decoding is successful. This fact is used to determine when the decoder has failed, but a mathematical proof of this fact not provided.

It has been proven in many textbooks that the degree of V(x) must be strictly less than the degree of L(x) when decoding is successful, and, if L(x) = 1, then V(x) must be 0. These facts are used to simplify the decoder design but, again, are not mathematically proven.

It has been proven that calculating the syndrome in one step by evaluating the received polynomial at α, α^2, α^3, …, α^{2t} is equivalent to calculating the syndrome in two steps where the first step is to find the remainder of dividing r(x) by g(x) and the second step is evaluating the remainder at α, α^2, α^3, …, α^{2t}, but this fact is not proven in this document.

This document does not provide a detailed description of the C code because most of the C code is self-explanatory and also because the primary purpose of this document is to describe the hardware design – not the C software.

11. References

This document refers to other documents which are listed below:

1. Customer’s “*BBCH Design Requirements Document*” referred to in this document as “**Customer’s Requirements Document**”

2. A paper entitled *"Euclideanization of the Berlekamp-Massey Algorithm"* by Willard L. Eastman from Proceedings of the 1988 Tactical Communications Conference, Vol. 1 (1988) pp. 295-303. Referred to in this document as **"Eastman's Paper"**.

3. United States Patent Number 5,754,563 entitled *"Byte-Parallel System for Implementing Binary BCH Error-correcting Codes"*, May 19, 1998. Referred to in this document as **"the PRS Patent"**.

4. Reuse Methodology Manual, Third printing 1999. Referred to in this document as "**the RMM**".

5. The book entitled *"Algebraic Coding Theory"*, written in 1968 by Elwyn Berlekamp. Referred to in this document as **"Berlekamp's book"**.

If there is a conflict or contradiction between what this documents states, indicates or implies and what the Verilog code is, the Verilog code should be taken to be the final authority.

12. Notation

To simplify the notation used in this document, in the C code and in the Verilog code, English letters are used as symbols instead of Greek letters as much as possible.

A summary of the notation used by ECC Tek, Eastman and Berlekamp for the polynomials involved in solving the key decoding equation is given in Table 5.

Table 5 Comparison of Notation used by ECC Tek, Eastman and Berlekamp

	ECC Tek's Notation	**Eastman's Notation (and Notation in the PRS Patent)**	**Berlekamp's Notation**
Error Locator Polynomial	L(x)	$b^T(x)$, $\lambda(x)$ and $\Pi(x)$	$\sigma(z)$
Auxiliary Locator Polynomial	aL(x)	$b^O(x)$	$\tau(z)$
Error Evaluator Polynomial	V(x)	$p^T(x)$	$\omega(z)$
Auxiliary Evaluator Polynomial	aV(x)	$p^O(x)$	$\gamma(z)$
Last L(x) for Software Program	last_L(x)	$b^T(x)_{j-1}$	NA
Last V(x) for Software Program	last_V(x)	$p^T(x)_{j-1}$	NA
Number of Erasures	ne	μ	NA
Erasure Location i	eli	Wi	Xi
Current Discrepancy	dj	dj	$\Delta_1^{(k)}$
Last Nonzero Discrepancy	lnd	δ	NA
Number of Erasures Index	nei	λ	NA
Degree L(x) Variable	degLx	l	D(k)

Table 6 defines symbols, abbreviations and terms used in this document, in ECC Tek's C and Verilog code, in Eastman's paper, in Berlekamp's book and in other papers and books on coding theory.

Table 6 Definition of Symbols, Abbreviations and Terms

aL(x)	In ECC Tek's notation, the auxiliary error Locator polynomial.
aV(x)	In ECC Tek's notation, auxiliary error eValuator polynomial.
BM	Berlekamp-Massey – Elwyn Berlekamp and James Massey who are credited with discovering or inventing an efficient algorithm for solving the key decoding equation for BBCH codes. There are numerous versions of the BM algorithm. The version presented in Eastman's paper is used in this document and denoted as the EBM algorithm.
c[i]	In ECC Tek's C code, the codeword symbol in position i.
ci	In ECC Tek's Verilog code, the codeword symbol in position i.
dj	The current discrepancy. In this document, the current discrepancy = Vj where j is the EBM algorithm iteration number. If the current L(x) being calculated in the decoder is the correct L(x), the discrepancy will be 0 from this point until the end of the EBM algorithm is reached.
degLx	The degree of L(x). The variable degLx is the maximum degree – or an upper bound on the degree of L(x) – while L(x) is being formed from S(x) using the EBM algorithm. Once L(x) has been formed from S(x), the degLx variable is the actual degree of L(x). The degLx variable is used by the final stages of the decoder to determine if the error pattern calculated by the decoder is legitimate.
el[i]	In the C code, erasure location i.
eli	In the Verilog code, erasure location i.
g(x)	The generator polynomial for an algebraic block code such as a BBCH code. $g(x) = g_0 + g_1x+g_2x^2+\ldots+g_Rx^R$ where R is the number of redundant symbols. g_R is always 1.
gLx[]	In the C code, a global variable for storing the coefficients of L(x).
K	The number of symbols in each message or data field.

L[i]	In the C code, the coefficient of L(x) in position i.
Li	In the Verilog code, the coefficient of L(x) in position i.
Li-1	In the Verilog code, the coefficient of L(x) in position (i-1).
lnd	In the Verilog and C code, the last nonzero discrepancy calculated while executing the BM algorithm. The lnd variable is initialized to 1.
Lp[i]	In the C code, the coefficient in position i of the polynomial which is the formal derivative of the error locator polynomial. The formal derivative of L(x) is represented as L'(x) or Lpx or Lp(x).
Lpx_eval	The evaluation of Lp(x) at some specific value for x. An evaluation is the value obtained by replacing the indeterminate "x" in a polynomial with an actual finite field element and evaluating the resulting expression. The evaluation Lp(x) at "α^{-i}" is denoted as Lpx_eval or Lp(α^{-i}).
Lpxi	In the Verilog code, the coefficient in the polynomial which is the formal derivative of the error locator polynomial which is in the Th position.
Lx_eval	The evaluation of L(x) at some specific value for x. An evaluation is the value obtained by replacing the indeterminate "x" in a polynomial with an actual finite field element and evaluating the resulting expression. The evaluation L(x) at "α^{-i}" is denoted as Lx_eval or L(α^{-i}).
mlci	most likely codeword symbol in the Th position in the Verilog code.
mle[i]	In the C code, the most likely error symbol (symbol) in position i.
mlei	In the Verilog code, the most likely error pattern symbol in position i.
mlmi	In the Verilog code, the most likely message or data symbol i.
N	The number of symbols (symbols) in a codeword.
nei	The "number of erasures initiated" index.
R	The number of redundant symbols (symbols) in a codeword. For the BBCH code described in this document, R = 5.

r[i]	In the C code, the received symbol in position i.
ri	In the Verilog code, the received symbol in position i.
rs	Binary BCH – Dr. Irving Reed and Dr. Gustave Solomon who are credited with discovering rs codes in 1960.
s	The number of erasures the code can correct.
S(x)	The syndrome polynomial.
S[i]	In the C code, the coefficient of S(x) in position i.
Si	In the Verilog code, the coefficient of S(x) in position i.
t	In most books on coding theory, the number of errors a block code can correct.
Vi	In the Verilog code, the coefficient of V(x) in position i.
Vi-1	In the Verilog code, the coefficient of V(x) in position (i-1).
Vx_eval	The evaluation of V(x) at some specific value for x. An evaluation is the value obtained by replacing the indeterminate "x" in a polynomial with an actual finite field element and evaluating the resulting expression. The evaluation V(x) at "α^{-i}" is denoted as Vx_eval or V(α^{-i}).
Wj	In Eastman's paper and in some versions of the C code, the jth erasure location. The W can be thought of as standing for the W in "Where".
Π(x)	In Eastman's paper, the error locator polynomial. Π probably stands for "position".
Ω(x)	In Eastman's paper, the error evaluator or error magnitude polynomial. Omega – where "m" stands for magnitude.
α	A primitive element of a finite field. In this document $\alpha = 2$ which is one of many primitive elements in a finite field with 1024 elements.

$\gamma(z)$	In Berlekamp's book, the auxiliary error evaluator or auxiliary error magnitude polynomial which ECC Tek refers to as aV(x). Berlekamp uses the indeterminate "z" instead of "x" for polynomials with nonbinary coefficients I believe so that the reader can quickly distinguish between binary and nonbinary polynomials.
μ	In Eastman's paper, the number of erasures.
$\tau(z)$	In Berlekamp's book, the auxiliary error locator polynomial which ECC Tek refers to as aL(x). Berlekamp uses the indeterminate "z" instead of "x" for polynomials with nonbinary coefficients I believe so that the reader can quickly distinguish binary and nonbinary polynomials.
$\omega(z)$	In Berlekamp's book, the error evaluator or error magnitude polynomial which ECC Tek refers to as V(x). Berlekamp uses the indeterminate "z" instead of "x" for polynomials with nonbinary coefficients I believe so that the reader can quickly distinguish between binary and nonbinary polynomials.

13. C Code

The C code is used to test the encoder and decoder algorithms and to generate input data and expected results files for debugging the Verilog code. The C code can be used to see if the decoder correctly corrects correctable error patterns and to determine the probability of miscorrection when the number of errors in a received word exceeds the capability of the code to correct.

The WriteFiles() function in the C program writes testbench files. The other functions in the C program are used to test the encoder and decoder.

After being compiled with a standard ANSI C compiler, the C program should run on any PC or Mac with no alteration.

The C program is very basic. Printf() statements can be inserted at various points to view variables as the program executes.

What coding theorists call "symbols" are one short integer in the C code. For binary BCH codes, the symbols are "bits" and therefore each bit is a separate short integer variable in the C code. The bits are not packed into words so each bit is in a separate word.

A 13-bit finite field is used to identify each bit position in the binary BCH codewords. These 13-bit quantities are also one short integer in the C code.

Many of the variables in the C program are global variables which simplifies modification of the functions. The message array, m[i], the codeword array, c[i], and the most likely error and codeword arrays, mle[i] and mlc[i], are some of the primary global variables. Normally global variables are undesirable in software, but, in this case, they are helpful because different functions can operate on the same data without having to pass arguments from function to function. Different encoding, decoding and testing functions can be selected by calls from main() to perform various operations. Each possible encoding function assigns values to the global codeword array, c[i]. Errors can be added to the codeword c[i] array to create the received word r[i] array. Then any selected decoding function can be used to decode the r[i] array and generate the most likely error and codeword arrays, mle[i] and mlc[i].

The primary functions in the C program are the Encode() and Decode() functions. Other testing and multiplication functions allow the Encode() and Decode() functions to be tested individually or tied together and tested as a system.

There are three read-only tables in the programs. Each of the tables is initialized by main(). Although algorithms could be used instead of tables to find reciprocals and do multiplications, tables are much faster and allow many more error patterns to be tested in the same amount of time than what could be tested if algorithms were used.

Since the binary BCH code uses 13-bit bit location symbols, the multiplication table is very large. The multiplication table is a 2D array of 8,192x8,192 = 67,108,864 short integers. The computer on which the C code is run must contain much more than 67x2 = 134 MegaBytes of main memory. Modern PCs often contain more than a GigaByte of main memory so the memory requirement should not be a problem for most users.

The first table created by the C code is a list of finite field reciprocals. When a reciprocal of a finite field element is needed, this table is accessed.

The second table is a table of powers of a primitive element, α^i, where i is the index or address of a one-dimensional array and α^i is the value stored in the table or array at that address. This table is used by the decoder to quickly determine α^i given i where i = 0x0000, 0x0001,...,0x1fff.

The third table is a table of negative powers of a primitive element, $\alpha^{-i,}$ where i is the index or address of a one-dimensional array and α^{-i} is the value stored at that address or location. This table is used by the decoder to quickly determine α^{-i} given i. Received word and codeword locations are labeled as α^{-i} where i runs from 0x0000 to 0x01fff.

The Decode() function determines the most likely error word, mle[i], and the most likely codeword, mlc[i], for a particular received word, r[i].

When the Decode() function detects an abnormal condition, it sets the decode_Fail flag to TRUE (or 1) indicating that the decoding operation has failed. The decode Fail flag is used to detect error patterns that exceed the capability of the decoder to correct.

Some error patterns that exceed the correction capability of the code will either not be detected or will be miscorrected. There is no way around this fact. All decoders for algebraic block codes have this property. If a severe error pattern happens to also be a codeword, no decoder can detect it or correctly correct the received word because the error pattern will map one legitimate codeword into another legitimate codeword and the syndrome will be all zeros indicating no errors.

The probability of miscorrection can be measured and quantified by using the C testing code. Error patterns can be generated that exceed the correction capability of the code and then the Decode() function can be executed to determine the frequency of miscorrection.

13.1. C Functions for Generating Verilog Input Data and Expected Results Files

The WriteFiles() function generates Verilog input data and expected results files for use with testbenches.

Actual results generated by running Verilog simulations are compared with the contents of the expected results files generated by the WriteFiles() function. If the Verilog simulation results are correct, they will match the results generated by the WriteFiles() function.

13.2. C Functions for Testing the Encoder, Decoder and System

The testing functions are useful for verifying that the decoder can correctly correct all correctable error patterns and for determining how frequently the decoder will declare a decoding failure when error patterns exceed the correction capability of the code. For example, the testing C code can be used to determine the probability of the decoder declaring a failure conditioned on the fact that a 6-symbol error pattern has occurred.

There are a number of error pattern generator functions in the C code that can be used to generate error patterns with a particular number of errors.

Theoretically, it is possible to exhaustively test most of the decoder by assuming that an all-zero message has been encoded into an all-zero codeword because, if the syndrome generator is working correctly, the syndrome does not depend upon what message or codeword has been sent. The syndrome only depends upon the error pattern so, once it has been determined that the syndrome generator is working correctly, the remainder of the decoder can be tested by assuming an all-zero codeword was sent. Most of the C functions that generate error patterns use the error pattern generated to be the received word and assume that an all-zero codeword was sent. If the user does not feel comfortable assuming an all-zero codeword was sent, then random messages can be generated, encoded into codewords and the error pattern can be added to the codeword to form the received word.

The C functions included in the C program are listed in Table 7 with a brief description of what they do.

Table 7 C Functions Included in the C Code and What They Do

Decode()	Determines the most likely code word, mlc, for a particular received word, r.
Encode()	Generates a code word, c, from a message, m.

Initialize_alphaToThe()	Initializes the α^{i} table.
Initialize_alphaToTheMinus()	Initializes the α^{-i} table.
Initialize_Mult()	Initializes the finite field multiplication table.
Initialize_Recip()	Initializes the table of reciprocals.
Manually_Test_System()	Manually tests the encoder and decoder system.
Multiply()	Multiplies a_of_x by b_of_x to form c_of_x. a_of_x, b_of_x and c_of_x are global variables.
SystemTest2()	Tests the encoder and decoder system.
Test_Encoder()	Tests the encoder.
Test1_Decode()	Test decoder's ability to correct 1 symbol/symbol error patterns.
Test1_System()	Test encoder and decoder system.
Test2_Decode()	Test decoder's ability to correct 2 symbol/symbol error patterns.
Test3_Decode()	Test decoder's ability to correct 3 symbol/symbol error patterns.
Test4_Decode()	Test decoder's ability to correct 4 symbol/symbol error patterns.
TestR_Decode()	Test decoder's ability to correct equally probable random error patterns.
TestR5_Decode()	Test decoder's ability to correct 5 symbol/symbol random error patterns.
TestR8_Decode()	Test decoder's ability to correct 8 symbol/symbol random error patterns.
TestRn_Decode()	Test decoder's ability to correct n symbol/symbol random error patterns.
WriteFiles()	Writes Verilog input data and expected results testbench files.

The SystemTest() function tests both the encoder and decoder as a system. A message is generated at random and then encoded into a codeword. Errors are added to the codeword to form the received word and the received word is decoded to see if the decoder corrects the errors correctly. The decoder test functions are used to test the decoder against error patterns with 1 symbol in error, 2 symbols in error, 3 symbols in error, 4 symbols in error, 5 symbols in error, 8 symbols in error and a random number of bytes in error. For the BBCH code, all single, double and triple symbol error patterns have been tested.

It is not possible to exhaustive test decoders which correct many symbols when codewords are long because there are too many possible error patterns.

The purpose of the TestR() functions is to determine the probability of miscorrection when the error pattern contains a large number of random errors.

Most of the decoder test functions assume the codeword is the all-zeros codeword. A nonzero received word is created by the test function for the decoder to decode. Each received word, r[i], created by the test function has a nonzero symbol value (Value 1 = V1) in Location 1 (L1). There also may be L2, V2, L3, V3, etc. symbols.

The WriteFiles() function writes input data and expected results files for use with testbenches.

14. Introduction to the Hardware Designs Described in Verilog

ECC Tek's hardware design strategy is to use only a very small number of basic Verilog language constructs as building blocks in developing BBCH encoder and decoder designs to ensure that the designs will be synthesizable, will achieve a high level of performance, will be easy to understand and maintain and can be quickly developed.

The basic Verilog constructs used by ECC Tek are as follows:

- Constructs that ECC Tek knows will synthesize into Registers
- Constructs that ECC Tek knows will synthesize into Multiplexers
- Constructs that ECC Tek knows will synthesize into State Machines
- Constructs that ECC Tek knows will synthesize into Simple Combinatorial Logic

ECC Tek does not view Verilog as a programming language such as C, but as a way to easily simulate and synthesize circuits that have been previously laid out in block diagrams as shown in the Figures of this document.

The way ECC Tek designs digital logic is by first creating block diagrams (pictures) of the data flow required for a particular design to achieve a desired level of performance. For example, most of the block diagrams shown in the Figures in this document were created before any of the Verilog code was written, and the resulting performance of the encoder and decoder was predicted based upon previous experience in designing encoders and decoders.

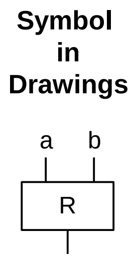
Writing and debugging the Verilog code was the last step in the design process.

By adhering to this discipline, ECC Tek is able to quickly create and debug designs that can be customized to each customer's unique requirements.

It is ECC Tek's belief that, if ECC Tek did not develop designs using only a few well-proven constructs, then the time to finish a design and the risk of failure would be much higher than it is using this design methodology.

In addition to the benefits of reducing development time and development risk, this design strategy also results in designs that are easy-to-understand, easy-to-modify and easy-to-maintain by the customer.

The symbol used in the drawings for a register, R, the associated Verilog code and the circuit ECC Tek knows will be synthesized from the Verilog code are illustrated in Figure 5.



Verilog Code

```
always @ (posedge clk or negedge reset)
      begin
      if          (reset == 1'b0)   R <= 0;
      else if  (cond1 == 1'b1)   R <= a;
      else if  (cond2 == 1'b1)   R <= b;
      else                               R <= R;
      end
```

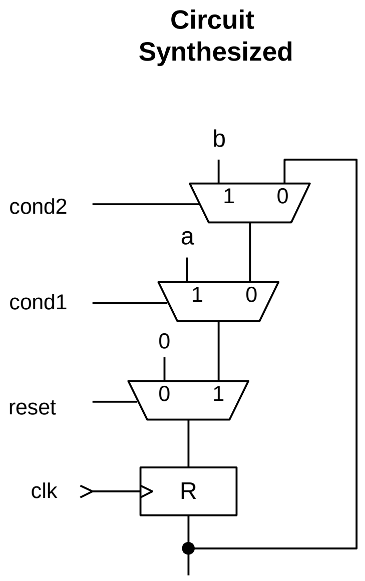
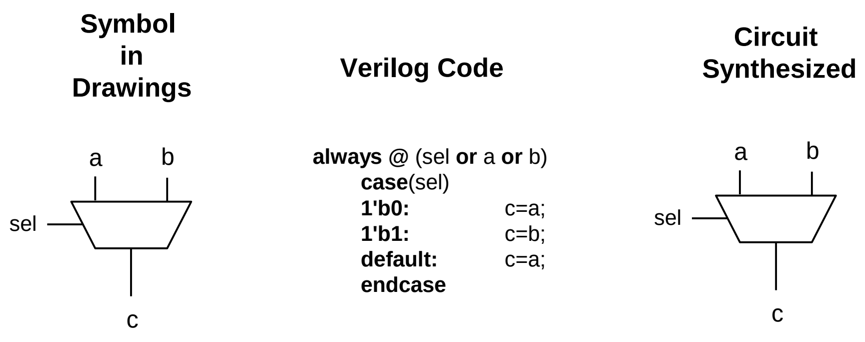


Figure 5 Registers

The symbol used in the drawings to represent a multiplexer (mux), the associated Verilog code and the circuit ECC Tek knows the Verilog code will be synthesized into are shown in Figure 6.



```
always @ (sel or a or b)
        case(sel)
        1'b0:           c=a;
        1'b1:           c=b;
        default:        c=a;
        endcase
```

Figure 6 Multiplexers (Muxes)

State Machines combine a state register with a set of muxes and ECC Tek knows the Verilog code will synthesize into a circuit of the form shown in Figure 7. A sample of the Verilog code for implementing a state machine can be seen by looking at the SM Verilog code in the PPU module, for example.

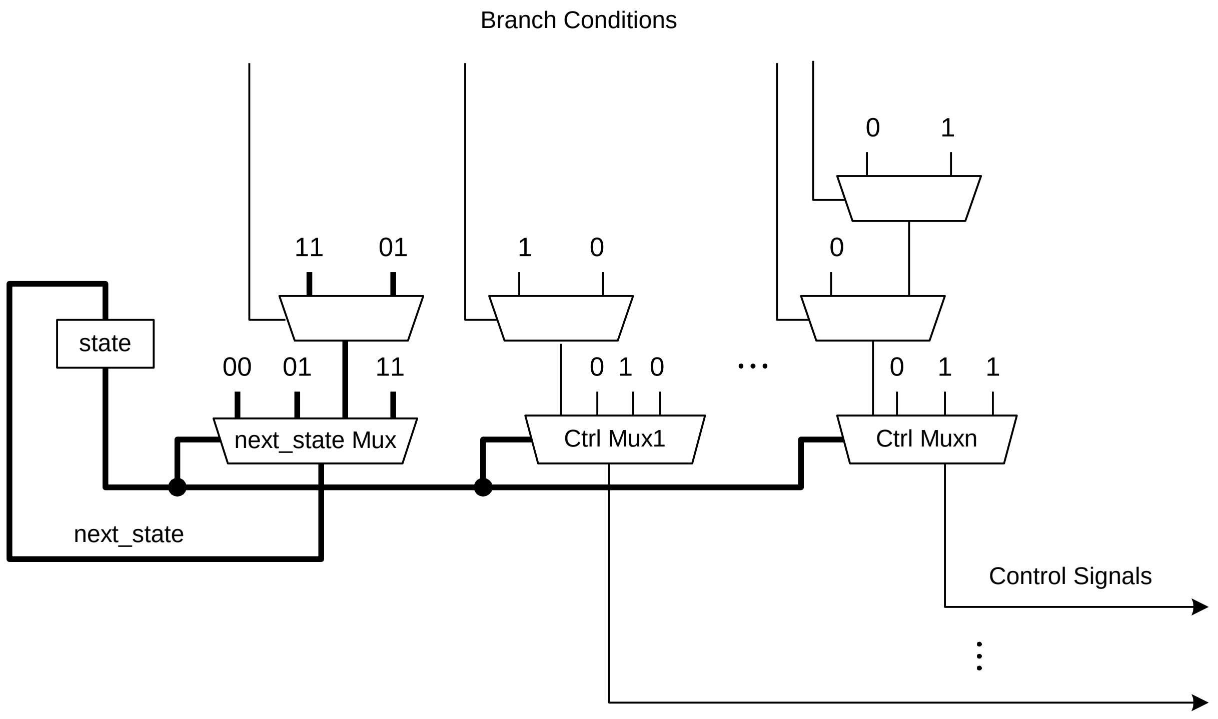


Figure 7 Generic State Machine

A generic control unit is shown in Figure 8. It consists of a number of registers (often up and down counters) used for control purposes and, usually, a state machine ("SM") to implement the control logic. In come cases the control is done completely by combinatorial logic and there is no state machine.

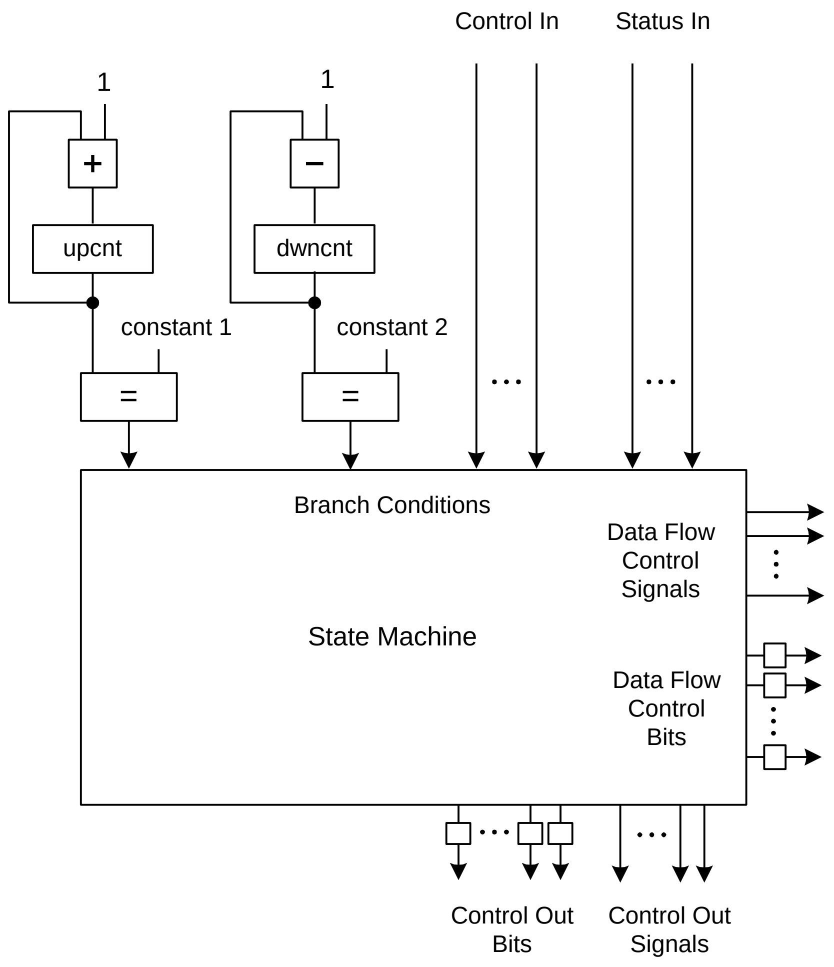


Figure 8 Generic Control Unit with a State Machine

15. Verilog Coding Style and Signal Naming Conventions

The encoder and decoder Verilog code has been written to, for the most part, comply with the Verilog coding style guidelines recommended in the RMM.

The Verilog code is written in a synthesizable form so that synthesizer software can be used to synthesize the actual circuits. This makes the BBCH designs independent of IC vendor.

The Verilog code can be synthesized to be implemented in an ASIC, a structured ASIC or in an FPGA.

Each Verilog module is in a separate file. The names of the files are the same as the names of the modules they contain with the addition of the ".v" filename extension. All filenames use primarily capital letters. For example, the file named "MULTA.v" contains the Verilog module named MULTA.

All signal or variable names in Verilog use lowercase letters.

All Verilog module inputs are prefixed with "i_" to indicate "input" and all Verilog module outputs are prefixed with "o_" for "output". For example, i_clk could be the name of a clock input and o_clk could be the name of a clock output.

The Verilog code is written in a style that makes it easy to spot errors and make changes. All input and output ports to all of the modules are on separate lines and all register and wire type variable declarations are on separate lines.

16. General Hardware Design Development Methodology

The design methodology used to develop the BBCH encoder and decoder hardware designs was to develop them in "steps" as follows:

1. Use a variation of the well-proven and efficient Berlekamp-Massey ("BM") algorithm for decoding. ECC Tek uses one of many variations of the BM algorithm for decoding as described by Willard Eastman in Eastman's paper and refers to it as the "EBM" algorithm.

2. Implement encoder and decoder algorithms in C to prove they work correctly and to generate input data and expected results files for debugging the Verilog code.

3. Thoroughly test the C models.

4. Create an initial "skeleton" version of the decoder that works, but does not have all of the final features required.

5. Add features to the "skeleton" version of the decoder to create a series of decoder versions until all of the required features have been successfully implemented. Each version is one "step". If we are unsuccessful in implementing a "next step", then we can always go back to the previously successful last step and try again.

6. Repeat the above development process for the encoder.

7. Use the C models and Verilog testbenches to debug and validate the designs.

17. Basic Logic Design Concepts Used by ECC Tek

All of the blocks of logic in the encoder and decoder can be divided into two parts – one part is the "control structure", "control unit" or just "control" and the other part is the "data flow structure", "data flow unit" or just "data flow" as illustrated in Figure 9. The control unit receives status signals from the data flow unit and sends control signals to the data flow unit to control the flow of data. Control signals are generally fed into and out of the control unit and data is fed into and out of the data flow unit.

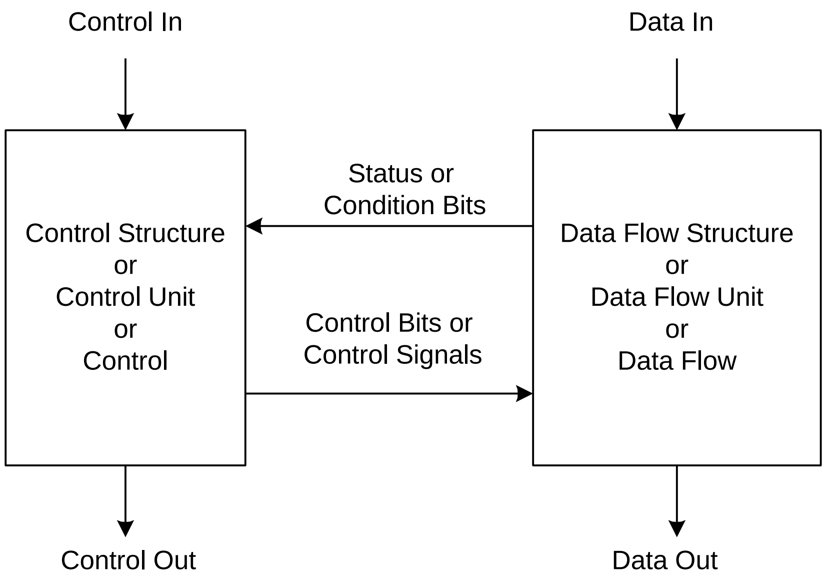

Control In
Data In
Status or Condition Bits
Control Structure or Control Unit or Control
Data Flow Structure or Data Flow Unit or Data Flow
Control Bits or Control Signals
Control Out
Data Out

Figure 9 General Form for all Logic Blocks

State machines are described in Verilog in a standard format as described in the RMM so that the Verilog code for a SM will synthesize into a logic circuit of the form shown in Figure 7.

Probably the best way to understand the SM logic is to look at the Verilog code. State diagrams can also be drawn from the Verilog code, but viewing the Verilog code is how ECC Tek creates the SM. The Verilog statements are usually simple, easy to follow and easy to understand. All of the state machines used in the encoder and decoder are very simple and usually only have a few states.

18. Drawing Conventions

In the drawings, the data flow is usually top down.

The control unit for a block of logic is usually drawn to the left of the data flow unit.

All Galois Field full multipliers are drawn with a dot (“.”) or “x” inside a box. Partial Galois Field multipliers are drawn with a star (“*”) inside a box.

19. Clocking

The encoder and decoder can be clocked at over 120 MHz using worst case maximum logic delays.

20. Implementing Finite Field Operations in Digital Logic

The encoder performs operation in a finite (or Galois) field with 2 1-bit elements or “bits”.

The decoder performs operations in a finite field with 2 1-bit elements “bits” and also in another finite field with 8,192 13-bit elements. The irreducible and primitive binary polynomial used to generate the finite field with 13-bit elements is $p(x) = 201B \text{ hex} = x^{13} + x^4 + x^3 + x + 1$.

The finite field operations used for encoding and decoding BBCH codes are as follows:

- addition of two variable finite field elements,
- multiplication of a variable finite field element by a constant finite field element,
- multiplication of two variable finite field elements, and
- finding the reciprocal (or inverse) of a finite field element.

Addition of finite field elements is a bit-wise exclusive-or (XOR) operation.

The finite field multiplier that multiplies two variable finite field elements is a Verilog module called MULTA. MULTA is a highly structured full multiplier as described in the PRS Patent and illustrated in the top part of Figure 10.

If n multipliers are used and one of the inputs is common to all n multipliers, then the common multiplier logic that would normally be implemented inside each multiplier can be pulled out and implemented outside the multiplier only one time rather than n times. ECC Tek calls the resulting multiplier with the missing piece a “partial” multiplier. Partial multipliers are illustrated in bottom part of Figure 10.

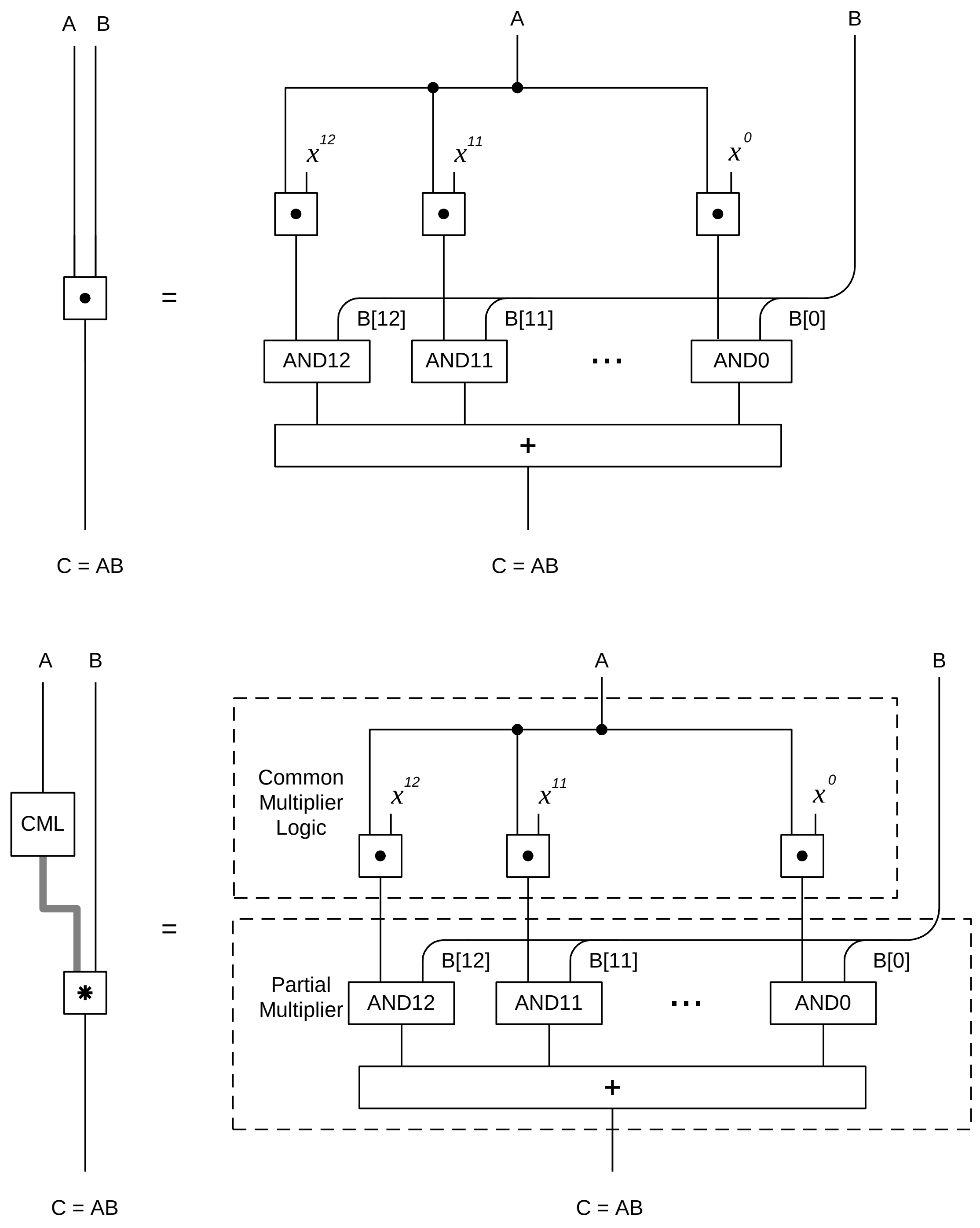


Figure 10 Full Multiplier (top) and Partial Multiplier (bottom)

If an intelligent synthesizer is used, the synthesizer should also recognize that part of the multiplier is common and the result of synthesizing full multipliers and partial multipliers should be the same. Since the synthesis result is synthesizer-dependent, ECC Tek provides both full and partial multiplier versions of the Verilog modules.

Multipliers that multiply a variable finite field element by a constant field element are implemented as shown in Figure 11. The constant, B, determines the connections in Figure 11.

Multipliers that multiply a variable by a constant are separate Verilog modules. The Verilog module that multiplies by 0ABC is called MULT_BY_0ABC.

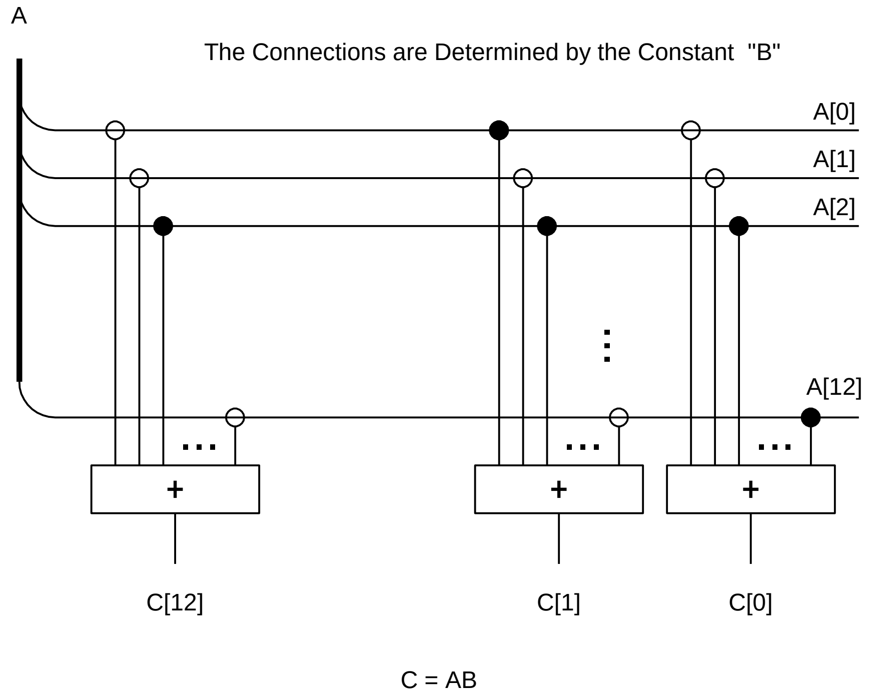


Figure 11 Multiplier that Multiplies the Variable "A" by the Constant "B"

Reciprocals of finite field elements are stored in a ROM. Only one reciprocal ROM is needed.

21. Initial Design Unknowns and Guesses

When first starting to develop a completely new encoder and decoder design such as in the binary BCH case, there are many unknowns and some initial guesses need to be made. Since ECC Tek's customers choose what circuit technology will be implemented and what synthesis software will be used, it is difficult for ECC Tek to initially know how fast the synthesized circuits will be and how much parallelism is needed in the logic so ECC Tek must make some initial guesses and do some initial synthesis runs to find out answers to those questions.

In the case of the BBCH decoder design, ECC Tek's initial guess regarding how to design the Syndrome Generator, SYNA, was too conservative and it led to a design that required too many gates so a number of iterations had to be done to develop the final Syndrome Generator modules.

22. Encoder Design Details

An overview block diagram of the encoder is shown in Figure 12.

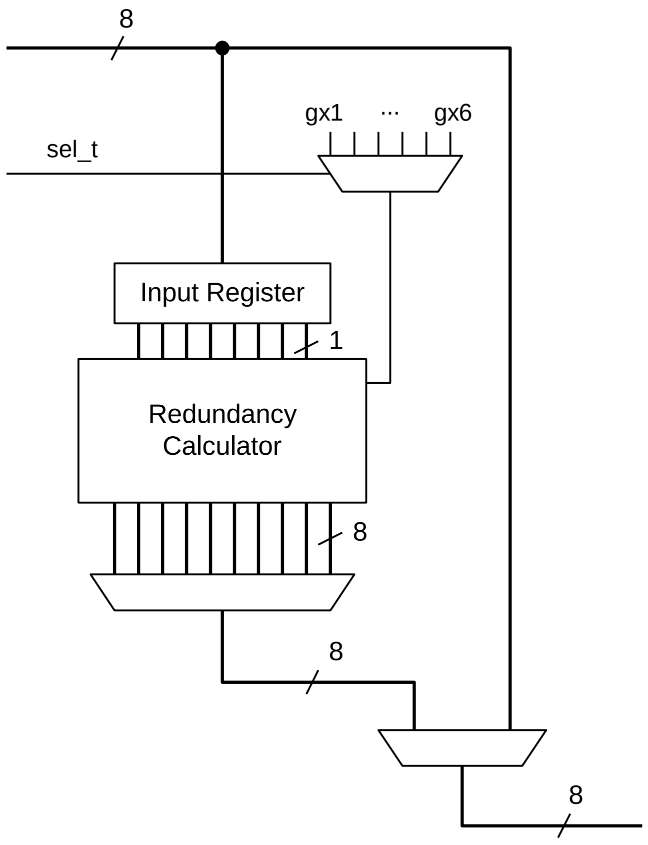


Figure 12 Encoder Block Diagram

23. Encoder Signal Definitions

Input signals have a prefix of "i_". Output signals have a prefix of "o_". For example, i_symbol_valid is an input signal and o_symbol_valid is an output signal.

Signal Name	Description
i_rst	Asynchronous reset, active low
i_clk	Synchronous clock, active on rising edge

i_abort	Abort encoding
i_k_delta[5:0]	Value added to 512 to form K
i_sel_t[2:0]	t value selected
i_byte[7:0]	Input byte
i_byte_valid	Input byte valid
i_enable_parity	Enable parity bytes to be output
o_byte[7:0]	Output byte
o_byte_valid	Output byte valid
o_first_byte	First output byte valid
o_last_byte	Last output byte valid

Table 8 Encoder Signal Definitions

24. Decoder Design Details

Details of the various Decoder versions delivered to Customer are given in Section 1.

25. Decoder Signal Definitions

Input signals have a prefix of "i_". Output signals have a prefix of "o_". For example, i_symbol_valid is an input signal and o_symbol_valid is an output signal.

Signal Name	Description
i_rst	Asynchronous reset, active low
i_clk	Synchronous clock, active on rising edge
i_abort	Abort decoding
i_pause	Pause decoding
i_k_delta[5:0]	Value added to 512 to form K
i_sel_t[2:0]	t value selected
i_byte[7:0]	Input byte
i_byte_valid	Input byte valid
o_byte_valid	Output byte valid
o_byte[7:0]	Corrected output byte
o_decode_fail	Decoder failed
o_error_count[4:0]	Number of errors corrected in a page. When o_last_byte is asserted, the o_error_count[4:0] is updated for the current page
o_error_cnt_valid	Error count valid
o_first_byte	Asserted when the first output byte is valid at the output
o_last_byte	Asserted when the last output byte is valid at the output
o_syn_not_0	Syndrome is not 0

Table 9 Decoder Signal Definitions

26. Syndrome Generator Mathematics

27. Syndrome Generator Circuits

27.1. SYNA

27.2. SYNB

27.3. SYNC

27.4. SYN1

27.5. SYN2

28. DFIFO

The DFIFO is implemented by Customer in an off-chip circuit with interfaces with the BBCH decoder logic.

The DFIFO, as implemented by ECC Tek, is configured to write the first K bytes of data to a FIFO and skip the redundant symbols.

29. PPU

30. INIT

31. EVAL

32. XOR