A Low-Complexity and High-Throughput RTL Design of a BCH (15,7) Decoder

Hendra Setiawan

Electrical Engineering Department, Islamic University of Indonesia
Jl. Kaliurang Km.14.5 Yogyakarta, Indonesia, 55583
Email: hendra.setiawan@uii.ac.id

Abstract. The Bose, Chaudhuri and Hocquenghem (BCH) codes form a large class of powerful random-error correcting cyclic codes. However, the implementation of its decoder requires high-complexity computation resources with a huge number of sequential circuits. This paper presents a low-complexity register transfer level (RTL) circuit design of a BCH decoder. In accordance with the table relationship between the syndrome and the error bit position, we propose a circuit that is mostly occupied by combinational elements without any sequential evolvement. Therefore the designed system has a low complexity and high throughput properties. The implementation of the BCH (15,7)decoder on Virtex 5 FX70TFF1136 requires 77 look-up tables (LUTs) with the maximum throughput reaching 1.7 Gbps.

Keywords: *BCH decoder; cyclic code; error correction; RTL; syndrome.*

1 Introduction

Today, error-correcting codes are used throughout digital communication systems. Satellite communications, cell-phones, compact disc players, DVDs, disk drives, two-dimensional bar code systems and many other communication devices use varying amounts of error control to achieve a certain degree of accuracy in transmitting information. The Binary Bose, Chaudhuri and Hocquenghem (BCH) codes, discovered by Hocquenghem in 1959 and independently investigated by Bose and Chaudhuri in 1960, are a remarkable generalization of the Hamming codes for multiple-error correction. BCH codes containing Reed-Solomon codes have been widely adopted in practical error-control applications, owing to their good performance against degradation and the flexibility they allow in setting appropriate parameters [1]. Digital Video Broadcasting (DVB) [2] and Worldwide Interoperability for Microwave Access (WiMAX) [3] are examples of current standards that utilize BCH in their system.

One of the well-developed algorithms to decode binary BCH code uses a Euclidean algorithm [4]. However, its process requires high computation resources due to the error-locator polynomial. Other algorithms are step-by-step

Received April 4th, 2012, Revised August 14th, 2012, Accepted for publication August 27th, 2012.
Copyright © 2012 Published by LPPM ITB, ISSN: 1978-3086, DOI: 10.5614/itbj.ict.2012.6.2.2

algorithms [5,6] that consist of procedure tests to check whether the error pattern weight falls by changing the received symbols one at a time. This decoding procedure does not terminate until the error pattern weight has been reduced to zero or all received information symbols have been tested. Hence, this is called an iterative method of decoding. Even though the hardware implementation [7] is less complex than that of the first decoding algorithm, the throughput may not be higher due to the iterative procedures.

In this paper, we propose a simple hardware implementation procedure with low complexity and high throughput properties. This simple combinational circuit was developed based on the table relationship between the syndrome and the error bit position. Thus, a low-complexity BCH decoder could be developed. Furthermore, the decoder throughput could be increased by employing pipelining and parallelization.

This paper is organized as follows. In Section 2, the architecture of the encoder and the decoder is detailed. The design complexity is explained in Section 3. In Section 4, the compilation and synthesis results are presented. Finally, conclusions are drawn in Section 5.

2 Architecture Description

2.1 Encoder Specification

The architecture of a BCH encoder using shift register has been introduced by Massy in 1969 [8]. This paper considers a BCH (15,7) encoder consisting of 7 information bits and 8 parity bits as target implementation. The sending bit (SB) of this BCH (15,7) encoder are based on the polynomial given by:

$$\begin{aligned} SB(x) = {} & u_0 \cdot x^8 + u_1 \cdot x^9 + u_2 \cdot x^{10} + u_3 \cdot x^{11} + u_4 \cdot x^{12} \\ & + u_5 \cdot x^{13} + u_6 \cdot x^{14} + r_0 + r_1 \cdot x^1 + r_2 \cdot x^2 + r_3 \cdot x^3 \\ & + r_4 \cdot x^4 + r_5 \cdot x^5 + r_6 \cdot x^6 + r_7 \cdot x^7 \end{aligned} \tag{1}$$

where u_0, u_1, u_2, u_3, u_4, u_5, u_6 represent information, and r_0, r_1, r_2, r_3, r_4, r_5, r_6, r_7 express the parity bits. This can be implemented using the remainder polynomial, based on:

$$x^8 = 1 + x^4 + x^6 + x^7 \tag{2}$$

Furthermore, Eq. (2) can be realized by a simple circuit, as shown in Figure 1, where u_6, u_5, ..., $u_0$inputted serially in signal input port (*SIN*), and r_7, r_6, ..., $r_0$generated after seven clock cycles.

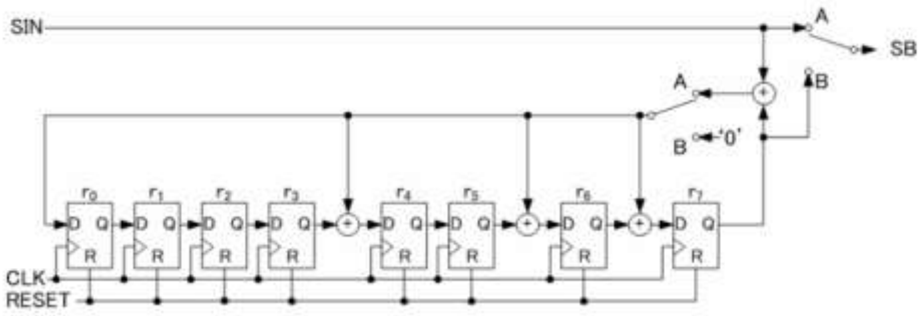


Figure 1 Circuit implementation of theBCH (15,7) encoder.

Aparallel process to get the parity bits is introduced in order to reach a higher throughput. Parallel computation is performed based on the remainder polynomial of

$$\begin{aligned} R(x) = & \left[u_0 \cdot x^8\right] + \left[u_1 \cdot x^9\right] + \left[u_2 \cdot x^{10}\right] + \left[u_3 \cdot x^{11}\right] \\ & + \left[u_4 \cdot x^{12}\right] + \left[u_5 \cdot x^{13}\right] + \left[u_6 \cdot x^{14}\right] \end{aligned} \tag{3}$$

Based on Eq. (2), we can derive the remainder polynomial x^9, x^{10}, x^{11}, x^{12}, $x^{13}$and x^{14},and then substitute to Eq. (3). Hence, we get

$$\begin{aligned} R(x) = & (u_0 + u_1 + u_3) + (u_1 + u_2 + u_4) \cdot x + (u_2 + u_3 + u_5) \cdot x^2 \\ & + (u_3 + u_4 + u_6) \cdot x^3 + (u_0 + u_1 + u_3 + u_4 + u_5) \cdot x^4 \\ & + (u_1 + u_2 + u_4 + u_5 + u_6) \cdot x^5 + (u_0 + u_1 + u_2 + u_5 + u_6) \cdot x^6 \\ & + (u_0 + u_2 + u_6) \cdot x^7 \end{aligned} \tag{4}$$

Therefore,
$r_0 = u_0 + u_1 + u_3$; $r_1 = u_1 + u_2 + u_4$;
$r_2 = u_2 + u_3 + u_5$; $r_3 = u_3 + u_4 + u_6$;
$r_4 = u_0 + u_1 + u_3 + u_4 + u_5$; $r_5 = u_1 + u_2 + u_4 + u_5 + u_6$;
$r_6 = u_0 + u_1 + u_2 + u_5 + u_6$; $r_7 = u_0 + u_2 + u_6$

Eq. (4) can be realized easily in a circuit, as shown in Figure 2, where the adder symbol is implemented using *XOR* gates.

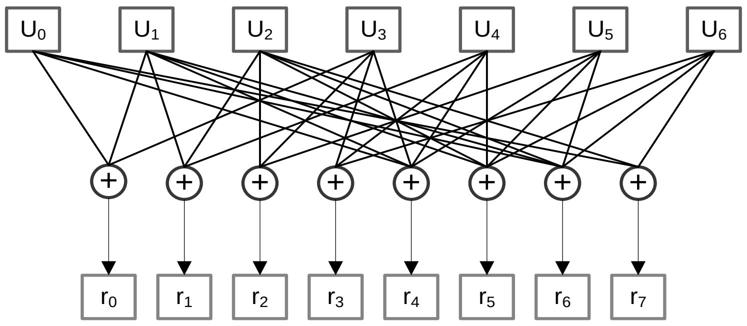


Figure 2 Parallel computation of a BCH (15,7) encoder.

2.2 Proposed BCH (15,7) Decoder

This paper proposes parallel computation for the BCH (15,7) decoder to reach a higher throughput. The proposed system consists of 7 main blocks, as shown in Figure 3.

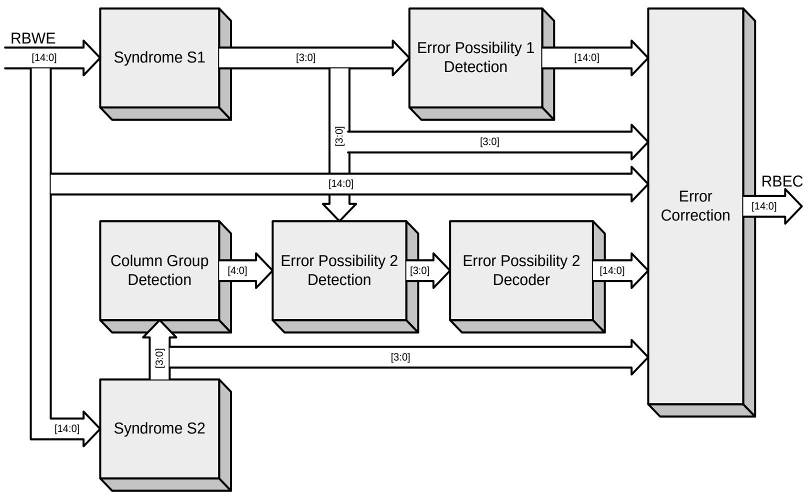


Figure 3 Block diagram of the BCH (15,7) decoder.

2.2.1 Syndrome Calculation

In this process, syndrome blocks *S1* and *S2* generate the syndrome bits of received bits with error (*RBWE*). The generation polynomial *G*(*x*) for error-checking is given by,

$$G_1(x) = 1 + x + x^4 \tag{5}$$

$$G_2(x) = 1 + x + x^2 + x^3 + x^4 \tag{6}$$

$G_1(x)$ and $G_2(x)$ are related to syndrome *S1* and syndrome *S2* respectively. If there is no error in the received code, both the remainder polynomial of $G_1(x)$ and that of $G_2(x)$ remain zero. Suppose the received bits with error (*RBWE*) are expressed as,

$$\begin{aligned} RBWE(x) = {} & \hat{u}_6 \cdot x^{14} + \hat{u}_5 \cdot x^{13} + \hat{u}_4 \cdot x^{12} + \hat{u}_3 \cdot x^{11} + \hat{u}_2 \cdot x^{10} \\ & + \hat{u}_1 \cdot x^9 + \hat{u}_0 \cdot x^8 + \hat{r}_7 \cdot x^7 + \hat{r}_6 \cdot x^6 + \hat{r}_5 \cdot x^5 \\ & + \hat{r}_4 \cdot x^4 + \hat{r}_3 \cdot x^3 + \hat{r}_2 \cdot x^2 + \hat{r}_1 \cdot x^1 + \hat{r}_0 \end{aligned} \tag{7}$$

Based on Eq. (5), we can derive the remainder polynomial for x^5, x^6,…,x^{14} and then substitute it in Eq.(7), hence syndrome *S1* is:

$$\begin{aligned} S1(x) = {} & \hat{u}_6 \cdot (1+x^3) + \hat{u}_5 \cdot (1+x^2+x^3) + \hat{u}_4 \cdot (1+x+x^2+x^3) \\ & + \hat{u}_3 \cdot (x+x^2+x^3) + \hat{u}_2 \cdot (1+x+x^2) + \hat{u}_1 \cdot (x+x^3) \\ & + \hat{u}_0 \cdot (1+x^2) + \hat{r}_7 \cdot (1+x+x^3) + \hat{r}_6 \cdot (x^2+x^3) + \hat{r}_5 \cdot (x+x^2 \\ & + \hat{r}_4 \cdot (1+x) + \hat{r}_3 \cdot x^3 + \hat{r}_2 \cdot x^2 + \hat{r}_1 \cdot x^1 + \hat{r}_0 \end{aligned} \tag{8}$$

Thus,

$$\begin{aligned} & S1(0) = \hat{u}_6 + \hat{u}_5 + \hat{u}_4 + \hat{u}_2 + \hat{u}_0 + \hat{r}_7 + \hat{r}_4 + \hat{r}_0 \ ; \\ & S1(1) = \hat{u}_4 + \hat{u}_3 + \hat{u}_2 + \hat{u}_1 + \hat{r}_7 + \hat{r}_5 + \hat{r}_4 + \hat{r}_1 \ ; \\ & S1(2) = \hat{u}_5 + \hat{u}_4 + \hat{u}_3 + \hat{u}_2 + \hat{u}_0 + \hat{r}_6 + \hat{r}_5 + \hat{r}_2 \ ; \\ & S1(3) = \hat{u}_6 + \hat{u}_5 + \hat{u}_4 + \hat{u}_3 + \hat{u}_1 + \hat{r}_7 + \hat{r}_6 + \hat{r}_3 \end{aligned} \tag{9}$$

In the same way, syndrome *S2* can be expressed as,

$$\begin{aligned} & S2(0) = \hat{u}_6 + \hat{u}_2 + \hat{u}_1 + \hat{r}_5 + \hat{r}_4 + \hat{r}_0 \ ; \ S2(1) = \hat{u}_6 + \hat{u}_3 + \hat{u}_1 + \hat{r}_6 + \hat{r}_4 + \hat{r}_1 ; \\ & S2(2) = \hat{u}_6 + \hat{u}_4 + \hat{u}_1 + \hat{r}_7 + \hat{r}_4 + \hat{r}_2 ; \ S2(3) = \hat{u}_6 + \hat{u}_5 + \hat{u}_1 + \hat{u}_0 + \hat{r}_4 + \hat{r}_3 \end{aligned} \tag{10}$$

By replacing each (+) sign with an *XOR* gate, in total 48 *XOR* gates are required for the syndrome calculation block. However, this can be reduced by sharing the same logic, such as $\hat{u}_6$ *XOR* $\hat{u}_5$, used in *S1*(3) as well as in *S1*(0). This will reduce the number of *XOR* gates from 48 to 37.

2.2.2 Error Position Detection

The next process is error position detection based on the values of syndrome *S1* and *S2*. Eq. (10) will be re-applied and rearranged, becoming:

$$\begin{aligned}
S2(0) &= (\hat{u}_2 + \hat{r}_5 + \hat{r}_0) + (\hat{u}_6 + \hat{u}_1 + \hat{r}_4) \\
S2(1) &= (\hat{u}_3 + \hat{r}_6 + \hat{r}_1) + (\hat{u}_6 + \hat{u}_1 + \hat{r}_4) \\
S2(2) &= (\hat{u}_4 + \hat{r}_7 + \hat{r}_2) + (\hat{u}_6 + \hat{u}_1 + \hat{r}_4) \\
S2(3) &= (\hat{u}_5 + \hat{u}_0 + \hat{r}_3) + (\hat{u}_6 + \hat{u}_1 + \hat{r}_4) \qquad (11)
\end{aligned}$$

It is clear that an error occurs in $\hat{u}_2$, $\hat{r}_5$ or $\hat{r}_0$ and has only resulted on $S2(0)$. In the same way, an error in $\hat{u}_4$, $\hat{r}_7$ or $\hat{r}_2$ only has an effect on $S2(2)$. In Table 1, from the table relationship between the syndrome bits and error bits position, it can be seen that they occupy the same column. Thus, there are 5 column groups (CG), i.e.

CG0, consisting of$\hat{u}_2$, $\hat{r}_5$ and $\hat{r}_0$, corresponds to *S2* = “1000”;
CG1, consisting of $\hat{u}_3$, $\hat{r}_6$ and $\hat{r}_1$, corresponds to *S2* = ”0100”;
CG2, consisting of $\hat{u}_4$, $\hat{r}_7$ and $\hat{r}_2$, corresponds to *S2* = “0010”;
CG3, consisting of $\hat{u}_5$,$\hat{u}_0$ and $\hat{r}_3$, corresponds to *S2* = “0001”;
CG4, consisting of $\hat{u}_6$,$\hat{u}_1$ and $\hat{r}_4$, corresponds to *S2* = “1111”;

Table 1 Relation between syndrome bits and error bits position.

Error Position		***S2* (S_{20}, S_{21}, S_{22}, S_{23})**															
		0000	**0001**	**0010**	**0011**	**0100**	**0101**	**0110**	**0111**	**1000**	**1001**	**1010**	**1011**	**1100**	**1101**	**1110**	**1111**
S1 (S_{10}, S_{11}, S_{12}, S_{13})	0000	None															
	0001	8,13	3					2,6	0,14			10,12	1,9	5,11	4,7		
	0010	7,12		2			3,6		4,10		0,8		9,11	1,5		13,14	
	0011	1,11			2,3	6			5,9		0,13	7,10			4,12	8,14	
	0100	6,11			12,13	1			0,4		8,10	2,5			7,14	3,9	
	0101	4,14			8,12		1,3	2,11			10,13	0,7		5,6			9
	0110	0,10			7,13		3,11	1,2		5			6,9		12,14	4,8	
	0111	1,6			7,8	11			10,14		3,5	0,12			2,9	4,13	
	1000	5,10			2,8		6,13	11,12		0			1,4		7,9	3,14	
	1001	4,9			2,13		6,8	1,7			0,3	5,12		10,11			14
	1010	3,13	8					7,11	4,5			0,2	6,14	1,10	9,12		
	1011	3,8	13					1,12	9,10			5,7	4,11	0,6	2,14		
	1100	9,14			3,7		11,13	6,12			5,8	2,10		0,1			4
	1101	2,12		7			8,11		0,9		5,13		1,14	6,10		3,4	
	1110	0,5			3,12		1,8	6,7		10			11,14,		2,4	9,13	
	1111	2,7		12			1,13		5,14		3,10		4,6	0,11		8,9	

Therefore, the recognition of a CG can be based on the position of bit ‘1’ in S2. For example, S2 = “0101” means there is an error in CG1 and an error in CG3.

However, if an error in CG4is introduced from another CG, the recognition scheme becomes different. For example, an error occurs in CG4 as well as an error in CG2, where S2 = “1101” cannot be recognized from the bit ‘1’ position. In this case, an inverter is required before recognition of the bit ‘1’ position takes place. However, the inverter only works if the number of bit ‘1’ is more than two. Therefore, the code group detection system consists of the number of bit ‘1’ calculation, selectors, and bit ‘1’ position recognition.

The first component in the code group detection system is the number of bit ‘1’ calculation. A 4-bit adder can be used to implement it. However, it may need big resources since an adder consist of *XOR* and *AND* gates in four bits. Since our target does not actually count the number of bit ‘1’ but only recognizes that the number of bit ‘1’ is more than two, we propose a combinational circuit that only uses four *AND* gates and three *OR* gates. This is expressed in term of syndrome S2 as SEL.

SEL is ‘1’ when the number of bit ‘1’ within S2 bits is more than two. This equation also expresses CG4 detection, since CG4 always exists if the number of bit ‘1’ is more than two. The next component is the selector. Its function isto select between S2 and (NOTS2). The *XOR* gate has been chosen as the selector. The inputs are S2 and SEL, and the output belongs to the code group. Therefore, bit ‘1’ position recognition is no longer required. Finally, the code group detection circuit shown in Figure 4consists of 4*AND* gates, 3 *OR* gates, and 4 *XOR* gates.

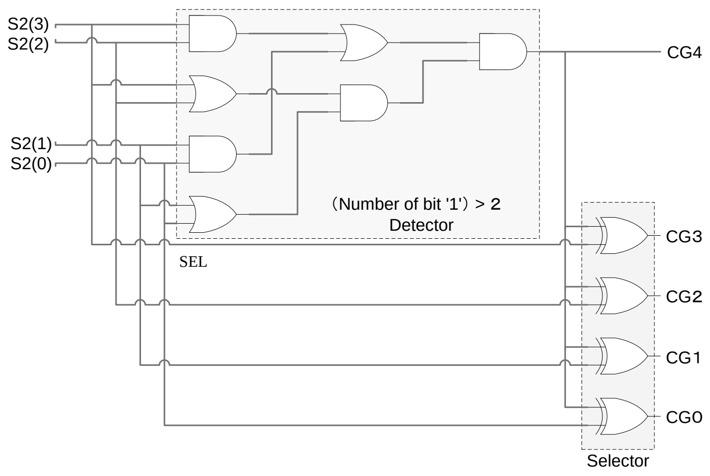


Figure 4 Code group detection circuit.

Furthermore, to simplify the error detection process, we divide the error possibilities into two groups: error possibility 1 (EP1) and error possibility 2 (EP2). EP1 consists of two errors occurring at the same time and in the same CG. Notice Eq. (11), when two errors in the same CG occur at the same time, it makesS2 = “0000”. In the next discussion, all error possibilities in theS2 = “0000” column of Table 1 are categorized as EP1.

The next error group is error possibility 2 (EP2). This group includes all errors in every column of Table 1 except column S2 = 0000. Note that "all errors" means, all errors that can be recognized and recovered by this error correction algorithm.

2.2.2.1 Error Possibility 1 (EP1)

EP1 occurs when two errors come from the same CG. The property of this case is syndrome *S2* = "0000" AND S1≠ "0000". The only way to detect the errors is direct mapping between *S1* and the error bit position within 15 received bits. Based on Table 1, column *S2* = "0000", the combinational circuit for *ERROR1*(14:0) can be expressed as:

ERROR1(14) = S1_12 OR S1_5 *ERROR1(13) = S1_1 OR S1_10*
ERROR1(12) = S1_2 OR S1_13 *ERROR1(11) = S1_4 OR S1_3*
ERROR1(10) = S1_6 OR S1_8 *ERROR1(9) = S1_9 OR S1_12*
ERROR1(8) = S1_1 OR S1_11 *ERROR1(7) = S1_2 OR S1_15*
ERROR1(6) = S1_4 OR S1_7 *ERROR1(5) = S1_8 OR S1_14*
ERROR1(4) = S1_5 OR S1_9 *ERROR1(3) = S1_10 OR S1_11*
ERROR1(2) = S1_13 OR S1_15 *ERROR1(1) = S1_3 OR S1_7*
ERROR1(0) = S1_6 OR S1_14 (12)

where,

S1_1 = (NOT S1(0)) AND (NOT S1(1)) AND (NOT S1(2)) AND S1(3)
S1_2 = (NOT S1(0)) AND (NOT S1(1)) AND S1(2) AND (NOT S1(3))
S1_3 = (NOT S1(0)) AND (NOT S1(1)) AND S1(2) AND S1(3)
S1_4 = (NOT S1(0)) AND S1(1) AND (NOT S1(2)) AND (NOT S1(3))
S1_5 = (NOT S1(0)) AND S1(1) AND (NOT S1(2)) AND S1(3)
S1_6 = (NOT S1(0)) AND S1(1) AND S1(2) AND (NOT S1(3))
S1_7 = (NOT S1(0)) AND S1(1) AND S1(2) AND S1(3)
S1_8 = S1(0) AND (NOT S1(1)) AND (NOT S1(2)) AND (NOT S1(3))
S1_9 = S1(0) AND (NOT S1(1)) AND (NOT S1(2)) AND S1(3)
S1_10 = S1(0) AND (NOT S1(1)) AND S1(2) AND (NOT S1(3))
S1_11 = S1(0) AND (NOT S1(1)) AND S1(2) AND S1(3)
S1_12 = S1(0) AND S1(1) AND (NOT S1(2)) AND (NOT S1(3))
S1_13 = S1(0) AND S1(1) AND (NOT S1(2)) AND S1(3)
S1_14 = S1(0) AND S1(1) AND S1(2) AND (NOT S1(3))
S1_15 = S1(0) AND S1(1) AND S1(2) AND S1(3)

This requires 45 *AND* gates, 15 *OR* gates and 28 *NOT* gates. However, sharing computation is introduced in *S1_1* to *S1_15*, so that:

S1_1 = C1_00 AND C0_01	*S1_2 = C1_00 AND C0_10*
S1_3 = C1_00 AND C0_11	*S1_4 = C1_01 AND C0_00*
S1_5 = C1_01 AND C0_01	*S1_6 = C1_01 AND C0_10*
S1_7 = C1_01 AND C0_11	*S1_8 = C1_10 AND C0_00*
S1_9 = C1_10 AND C0_01	*S1_10 = C1_10 AND C0_10*
S1_11 = C1_10 AND C0_11	*S1_12 = C1_11 AND C0_00*
S1_13 = C1_11 AND C0_01	*S1_14 = C1_11 AND C0_10*
S1_15 = C1_11 AND C0_11	(13)

where,

C1_00 = (NOT S1(0)) AND (NOT S1(1))
C1_01 = (NOT S1(0)) AND S1(1)
C1_10 = S1(0) AND (NOT S1(1))
C1_11 = S1(0) AND S1(1)
C0_00 = (NOT S1(2)) AND (NOT S1(3))
C0_01 = (NOT S1(2)) AND S1(3)
C0_10 = S1(2) AND (NOT S1(3))
C0_11 = S1(2) AND S1(3)

This scheme only needs 23 *AND* gates (a half less than before) and 8 *NOT* gates (reduced to 28%).

EP1 consists of three parts, as shown in Figure 5. The first part computes *C0_00*, *C0_01*, *C0_10*, *C0_11*, *C1_00*, *C1_01*, *C1_10* and *C1_11* simultaneously. The second part computes *S1_1* up to *S1_14* as expressed in Eq. (13). The last part computes *ERROR1*(14:0) based on Eq. (12). All computations are done without buffer and latency. Finally, the EP1 block requires 23 *AND* gates, 15 *OR* gates and 8 *NOT* gates.

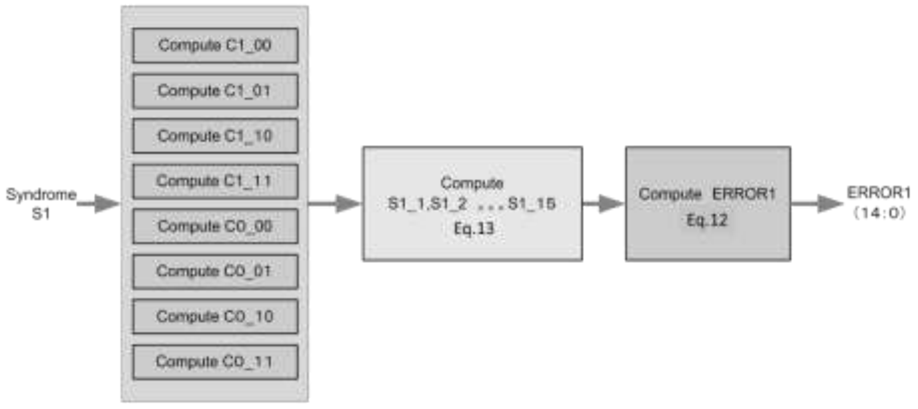


Figure 5 Block diagram of EP1.

2.2.2.2 Error Possibility 2 (EP2)

Error detection in this group is performed based on the code group and syndrome *S1*. There are 105 possible error positions in this group. The detection concept consists of three steps. First, the code group is used to generate a maximum of nine candidates in term of syndrome $\hat{S}1$. Next, all possible combinations of the syndrome $\hat{S}1$candidates are prepared and compared with the actual syndrome *S1*. As a result, a syndrome $\hat{S}1$ candidate that has the same pattern as the actual syndrome *S1* is recognized. Finally, this result is converted to the error position, within 0 to 15. The general architecture of EP2 detection is shown in Figure 6. Each step is explained in detail below.

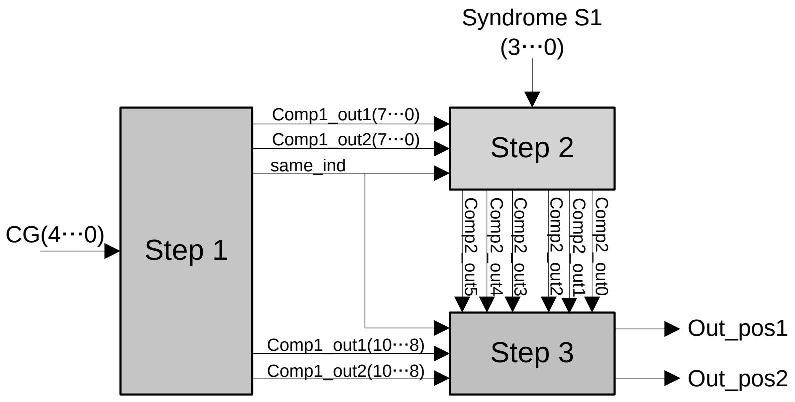


Figure 6 General architecture of EP2.

The main process of Step 1is to generate all $\hat{S}1$ candidates based on the received code group (CG). A maximum of two groups can be detected at the same time, where each group belongs to three $\hat{S}1$ candidates; a maximum of nine $\hat{S}1$ candidate combinations are produced in the Step 1 block. The$\hat{S}1$ candidates are based on Table 1. They are:

$CG0 \rightarrow \hat{S}1$ = “1000”, “0110”, “1110”
$CG1 \rightarrow \hat{S}1$ = “0100”, “0011”, “0111”
$CG2 \rightarrow \hat{S}1$ = “0010”, “1101”, “1111”
$CG3 \rightarrow \hat{S}1$ = “0001”, “1010”, “1011”
$CG4 \rightarrow \hat{S}1$ = “1100”, “0101”, “1001”

Note that *S1*, *S2* and $\hat{S}1$ have the same configuration, where the most left is the least significant bit (LSB –e.g. *S1*(0)) and the most right is the most significant bit (MSB –e.g. *S1*(3)).

Notice that one *Ŝ1*occurring in a CG is equal to the *XOR* of two other *Ŝ1s*. Therefore, two *Ŝ1s* must be mentioned in the process of Step 1. The details of the architecture of Step 1 are shown in Figure 7; it consists of ten selectors and a comparator to recognize a single error, since a single error will give the same value in both outputs. The ten bits on each selector represent two *Ŝ1s* (each 4 bits) and a representative error position (3 bits).

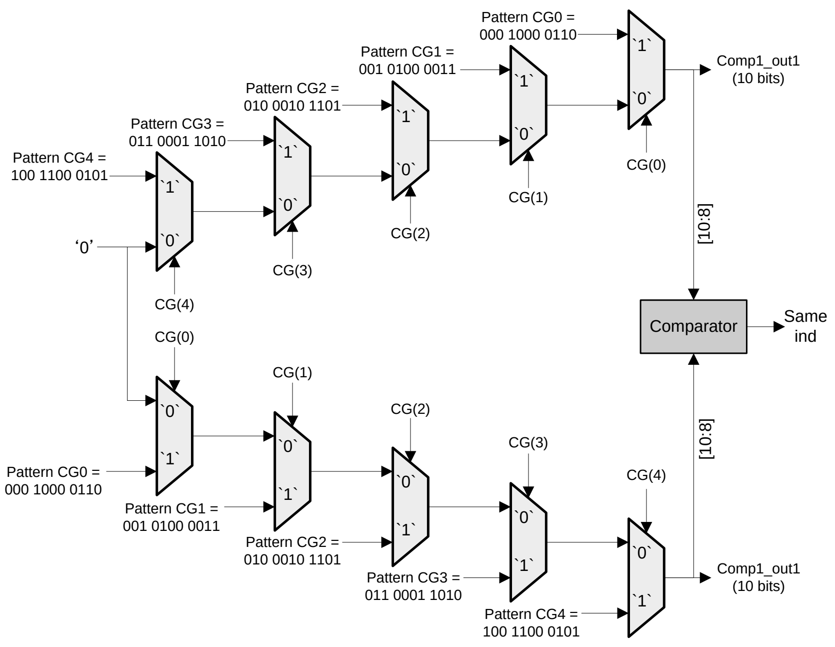

Pattern CG0 = 000 1000 0110
Pattern CG1 = 001 0100 0011
Pattern CG2 = 010 0010 1101
Pattern CG3 = 011 0001 1010
Pattern CG4 = 100 1100 0101
'0'
`1`
`0`
CG(4)
CG(3)
CG(2)
CG(1)
CG(0)
Comp1_out1 (10 bits)
[10:8]
Comparator
Same ind

Figure 7 Detailed architecture of Step 1.

The main process of Step 2 consists of *Ŝ1* combinations generation and a comparison of *Ŝ1* combinations with the actual syndrome *S1*. Since each CG contributes three *Ŝ1s* and there is a maximum of two errors with a different CG, the maximum number of combinations is nine. For example, the first CG gives*Ŝ1* = *A1*, *B1*, and *C1*, and the other CG gives *Ŝ1* = *A2, B2,* and *C2*. Therefore, the combinations of *Ŝ1* are (*A1 XOR A2*), (*A1 XOR B2*), (*A1 XOR C2*), (*B1 XOR A2*), (*B1 XOR B2*), (*B1 XOR C2*), (*C1 XOR A2*), (*C1 XOR B2*) and (*C1 XOR C2*). One of them should be the same as the actual *S1*. Figure 8 shows the details of the architecture of Step 2.

In Step 3, error positions are recognized based on comparing the results of Step 2 with a representative error from Step 1. A representative error is the smallest error position in each CG, for example, the representative error in *CG2*is "010". We can recognize two other errors because they have a special pattern, i.e.

interval five. Thus, when the representative error is “010”, the other errors are “0111” and “1100”. However, to recognize the actual error, the output of Step 2 has to be considered.

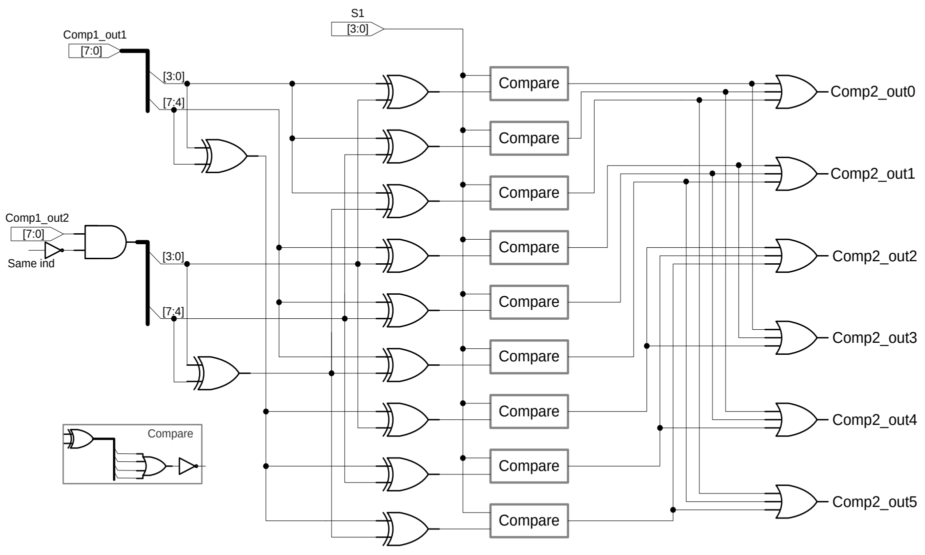


Figure 8 Detailed architecture of Step 2.

Figure 9 shows the architecture of the process of Step 3, which consists of *OR* gates, adders and a selector. The Step 3 output is served in 4-bit format. However, the bit correction pattern is in 15-bit format. Therefore an EP2 decoder is required.

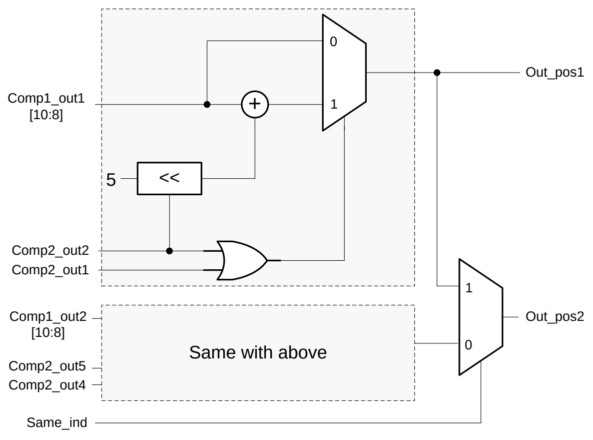


Figure 9 Detailed architecture of Step 3.

2.2.2.3 EP2 Decoder

The function in this section convertsEP2 output from 4-bit format to the error position within 15 bits. These 15-bit patterns are also called *ERROR2*. The relationship between the two port inputs (*out_pos1* and *out_pos2*) and the 15-bits *ERROR2* is expressed as:

ERROR2(0) = out_pos1_0000 OR out_pos2_0000
ERROR2(1) = out_pos1_0001 OR out_pos2_0001
ERROR2(2) = out_pos1_0010 OR out_pos2_0010
ERROR2(3) = out_pos1_0011 OR out_pos2_0011
ERROR2(4) = out_pos1_0100 OR out_pos2_0100
ERROR2(5) = out_pos1_0101 OR out_pos2_0101
ERROR2(6) = out_pos1_0110 OR out_pos2_0110
ERROR2(7) = out_pos1_0111 OR out_pos2_0111
ERROR2(8) = out_pos1_1000 OR out_pos2_1000
ERROR2(9) = out_pos1_1001 OR out_pos2_1001
ERROR2(10)= out_pos1_1010 OR out_pos2_1010
ERROR2(11)= out_pos1_1011 OR out_pos2_1011
ERROR2(12)= out_pos1_1100 OR out_pos2_1100
ERROR2(13)= out_pos1_1101 OR out_pos2_1101
ERROR2(14)= out_pos1_1110 OR out_pos2_1110 (14)

where,

out_pos1_0000 = (NOT out_pos1(0)) AND (NOT out_pos1(1)) AND
(NOT out_pos1(2)) AND (NOT out_pos1(3))
…
out_pos1_1110 = (NOT out_pos1(0)) AND out_pos1(1) AND out_pos1(2) AND out_pos1(3)

out_pos2_0000 = (NOT out_pos2(0)) AND (NOT out_pos2(1)) AND
(NOT out_pos2(2)) AND (NOT out_pos2(3))
…
out_pos2_1110 = (NOT out_pos2(0)) AND out_pos2(1) AND out_pos2(2) AND out_pos2(3)

Therefore, it can be implemented using a combinational circuit consisting of *AND* and *OR* gates. Some parts of this circuit are shown in Figure 10.

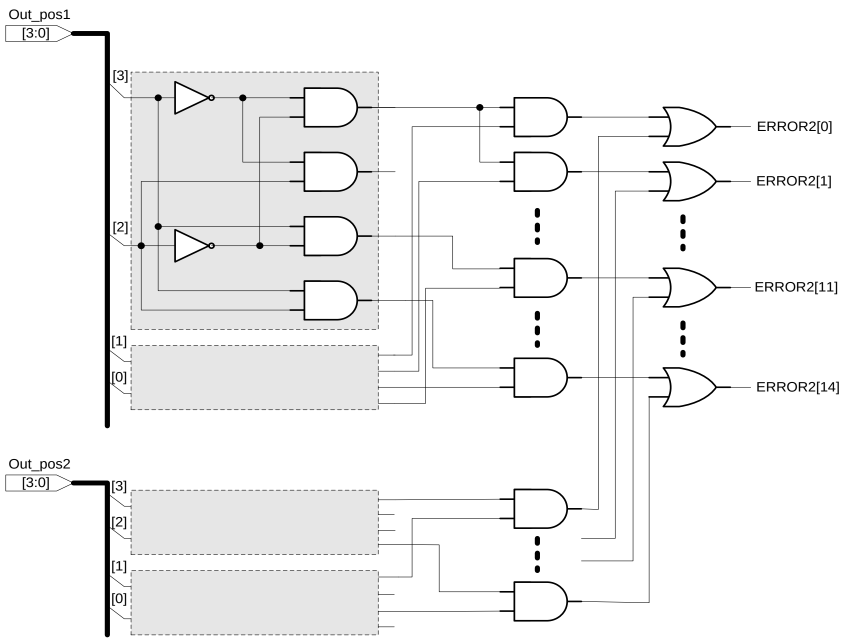


Figure 10 Architecture of the EP2 decoder.

Finally, the total complexity of EP2 and its decoder is given in Table 2, consisting of two multiplexers, a comparator, 20 *XOR* gates, 32 *OR* gates, 49 *AND* gates, 18 *NOT* gates, and two adders.

Table 2 Complexity of EP2 and EP2 decoder.

Block	Component	Number of gate	Total number of gate
EP2	Step 1	*MUX* = 10	*MUX* = 13
		Comparator = 1	Comparator = 1
	Step 2	*XOR* = 20	*XOR* = 20
		OR = 15	*OR* = 32
		AND = 1	*AND* = 49
		NOT = 10	*NOT* = 18
	Step 3	*OR* = 2	Adder = 2
		MUX = 3	
		Adder = 2	
EP2 decoder	EP2 decoder	*AND* = 48	
		NOT = 8	
		OR = 15	

2.2.3 Error Correction

The last step is error correction. The main concept of the error correction system is *XOR*-ing the received information with the pattern correction built fromEP1 and EP2. However, we must also consider syndrome *S1* and *S2* for selecting correction patterns, which can be expressed as,

$$RBEC = \begin{cases} 0, & \text{when } S1 = 0 \text{ and } S2 = 0 \\ RBWE \otimes ERROR1, & \text{when } S1 \neq 0 \text{ and } S2 = 0 \\ RBWE \otimes ERROR2, & \text{when others} \end{cases} \tag{15}$$

where,

RBEC	= received bit error correction, [14:0]
RBWE	= received bit with error, [14:0]
S1, S2	= syndrome *S1, S2* [3:0]
ERROR1	= error possibility 1, [14:0]
ERROR2	= error possibility 2, [14:0]
$\otimes$	= *XOR* operation

Therefore, the implementation uses six *OR* gates and one *XOR* gate, as well as a multiplexer, as shown in Figure11.

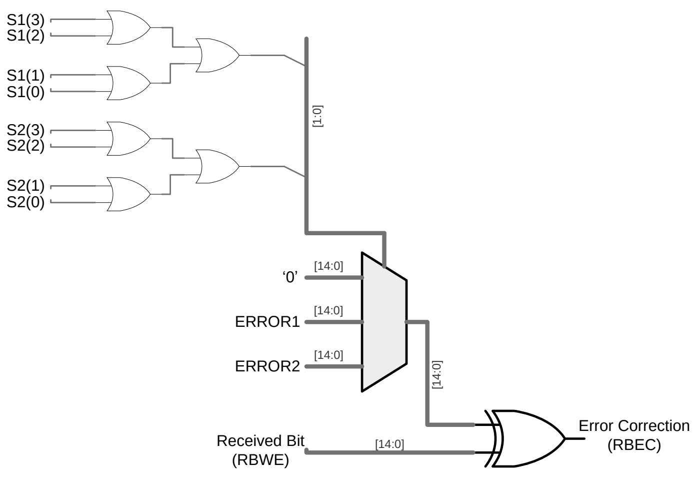


Figure 11 Error correction architecture.

3 Proposed Design Complexity

In section 2, the proposed RTL design was presented along with the complexity. Furthermore, the total complexity of each block is re-typed and shown in table 3. Note that a multiplexer is equal to 3 logic gates, and a comparator is equal to an adder and a logic gate.

Table 3 Total complexity of the proposed design.

No.	Block	Complexity	
		Logic gate	**Adder**
1	Syndrome calculation	37	-
2	Column group detection	11	-
3	EP1 detection	46	-
4	EP2 detection	88	3
5	EP2 decoder	71	-
6	Error correction	10	-
	Total	**263**	**3**

Based on the calculation shown in Table 3, the proposed BCH (15,7) decoder needs 263 logic gates and 3 adders. We now consider a simple algorithm proposed by Hong [9] as a comparer. Hong's algorithm for 2-bit errors gives a result of in total 110 multipliers, excluding the other components such as adders. The 110 multipliers are distributed such that 56 multipliers are used for syndrome evaluation, 6 multipliers for the error locator polynomial, 44 multipliers for root finding, and 4 multipliers for error evaluation.

Considering a 2-bit multiplier, its complexity is equal to 4 logic gates and an adder [10]. Thus, Hong's algorithm for 2-bit errors is equal to 440 logic gates and 110 adders. Therefore, the proposed system has a lower complexity than Hong's algorithm.

4 Simulation, Compilation and Synthesis Results

In order to ensure that the developed system has been worked out properly, we did a verification based on the block diagram in Figure 12. All parts were implemented in Very High Hardware Description Language (VHDL) and simulated using ModelSim 6.3. A snapshot of the functional simulation is shown in Figure 13. It is clear that all errors can be recovered by the decoder.

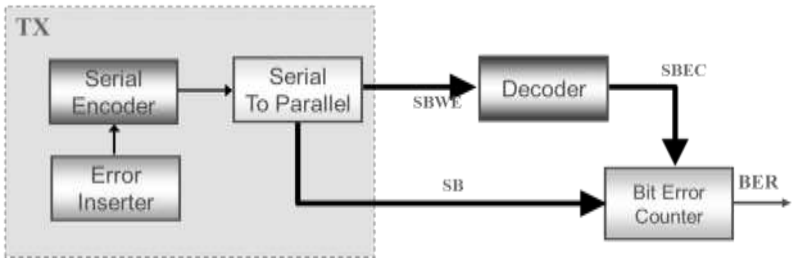


Figure 12 Block diagram for verification.

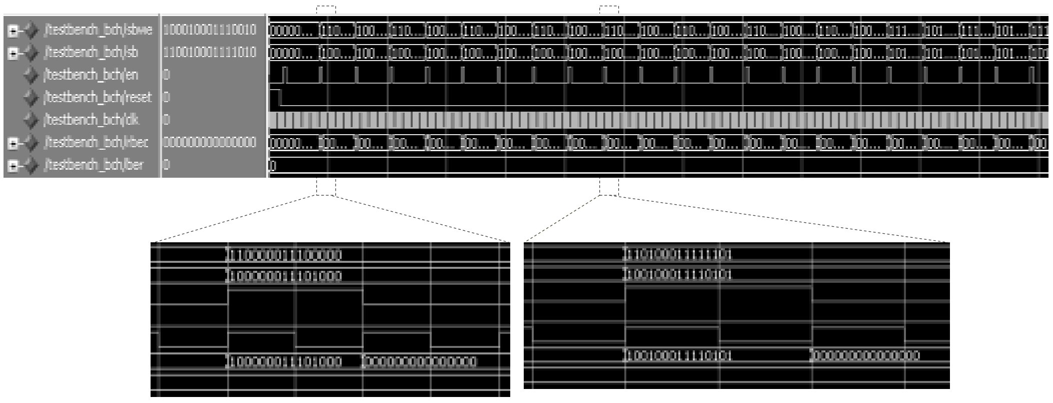


Figure 13 Snapshot of simulation result.

Furthermore, using a clock period of 200 ns, a long simulation was performed in 3 seconds, or 15 x 10^6 clock cycles. Within this period, the BCH decoder received approximately 1,000,000 data. The result was that there were no errors, as shown in Figure14, which means the bit error rate was zero, or all received bits were corrected perfectly.

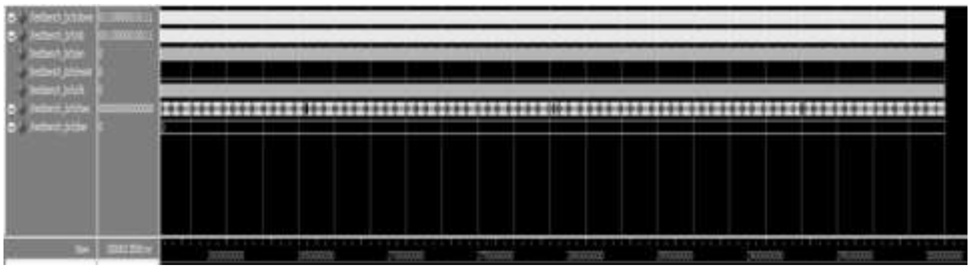
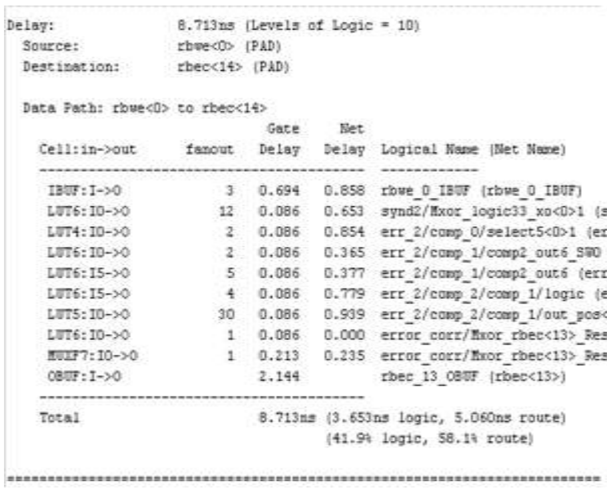


Figure 14 Simulation snapshot of one million data.

The compilation was processed using design tool ISE 11.2. The result shows that the critical path appears from register input *RBWE*(0) to register output *RBEC*(14), as shown in the snapshot of the compilation result in Figure 15. This path is through syndrome *S2*, column group detection, error possibility 2, and the error correction block. The critical path delay is 8.713 ns for Virtex 5 FX70TFF1136 implementation. The critical path can be reduced by pipelining. Without pipelining the proposed design has a maximum clock frequency of 114.771 MHz. Since the computation process is done in 15-bit parallel processing, the maximum throughput that can be achieved is 1.7 Gbps.



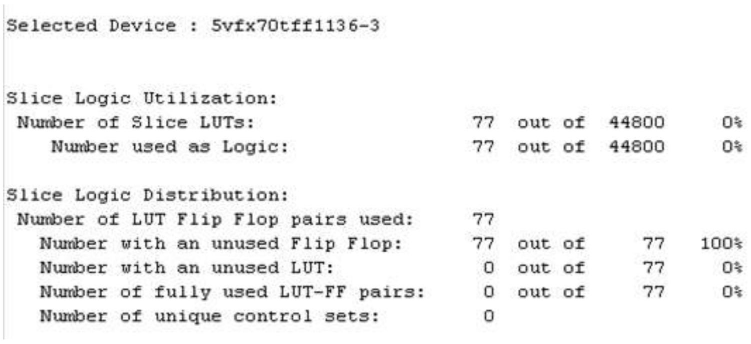
```
Delay:                8.713ns (Levels of Logic = 10)
  Source:             rbwe<0> (PAD)
  Destination:        rbec<14> (PAD)

  Data Path: rbwe<0> to rbec<14>
                                Gate     Net
    Cell:in->out      fanout   Delay   Delay  Logical Name (Net Name)
    ----------------------------------------  ------------
     IBUF:I->O             3   0.694   0.858  rbwe_0_IBUF (rbwe_0_IBUF)
     LUT6:I0->O           12   0.086   0.653  synd2/Mxor_logic33_xo<0>1 (syndrome_s2<3>)
     LUT4:I0->O            2   0.086   0.854  err_2/comp_0/select5<0>1 (err_2/comp1_out1<0>)
     LUT6:I0->O            2   0.086   0.365  err_2/comp_1/comp2_out6_SW0 (N32)
     LUT6:I5->O            5   0.086   0.377  err_2/comp_1/comp2_out6 (err_2/comp2_out2)
     LUT6:I5->O            4   0.086   0.779  err_2/comp_2/comp_1/logic (err_2/comp_2/comp_1/logic)
     LUT5:I0->O           30   0.086   0.939  err_2/comp_2/comp_1/out_pos<3>1 (out_pos1<3>)
     LUT6:I0->O            1   0.086   0.000  error_corr/Mxor_rbec<13>_Result12 (error_corr/Mxor_rbec<13>_Result11)
     MUXF7:I0->O           1   0.213   0.235  error_corr/Mxor_rbec<13>_Result1_f7 (rbec_13_OBUF)
     OBUF:I->O                 2.144          rbec_13_OBUF (rbec<13>)
    ----------------------------------------
    Total                      8.713ns (3.653ns logic, 5.060ns route)
                                       (41.9% logic, 58.1% route)

=========================================================================
```

Figure 15 Snapshot of the timing report.

From a circuit area point of view, the proposed architecture of the BCH (15,7) decoder requires 77 slice LUTs without flip-flop, as shown in Figure 16. Since all components are made from a combinational circuit, there are no sequential components such as a register or a memory. Thus, clock latency is zero.



```
Selected Device : 5vfx70tff1136-3


Slice Logic Utilization:
 Number of Slice LUTs:                   77  out of  44800     0%
    Number used as Logic:                77  out of  44800     0%

Slice Logic Distribution:
 Number of LUT Flip Flop pairs used:     77
   Number with an unused Flip Flop:      77  out of     77   100%
   Number with an unused LUT:             0  out of     77     0%
   Number of fully used LUT-FF pairs:     0  out of     77     0%
   Number of unique control sets:         0
```

Figure 16 Snapshot of the resource summary.

The 1.7 Gbps throughput is higher than the decoder architecture proposed by A. Kumar, *et.al.* [11], which can reach a data rate of up to 1.6 Gbps with a maximum clock of 200 MHz in an application-specific integrated circuit (ASIC) implementation. In addition, the proposed system has no latency since no sequential circuit is included. The decoder proposed by A. Kumar, *et.al.* [11] has a clock latency of 284. Thus, the proposed system has a lower latency than Kumar's decoder.

5 Conclusions

We have designed a BCH (15,7) hardware implementation ina combinational circuit instead of a sequential circuit to avoid high computation requirements and iteration processes. The simulation results using ModelSim 6.3 show that the developed circuit has correct functional processes. Furthermore, based on the compilation and synthesis results, the BCH decoder occupies 77 LUTs out of the 44800 LUTs on the target device Virtex 5 FX70TFF1136.The critical path delay is 8.713 ns in15-bit parallel processing. Thus the maximum throughput can reach 1.7 Gbps. Since sequential circuits are no longer involved, there is no process latency and the output can be executed in one clock cycle.

References

[1] Costello Jr., D.J., Hangenauer, J., Imai, H. & Wicker, S.B., *Applications of Error Control Coding*, IEEE Transactions on Information Theory, **44**(6), pp. 2531-2560,Oct.1998.
[2] ETSI, *Digital Video Broadcasting (DVB); Frame Structure Channel Coding and Modulation for A Second Generation Digital Terrestrial Television Broadcasting System (DVB-T2)*, European Std. ETSI EN 302 755, V1.1.1, Sept.2009.
[3] IEEE, *Part 16: Air Interface for Fixed and Mobile Broadband Wireless Access Systems*, IEEE Std. 802.16e-2005, Feb. 2006.
[4] Lin, S. & Costello Jr., D.J., *Error Control Coding*, Prentice Hall, 2004.
[5] Massey, J., *Step By Step Decoding of The Bose Chaudhuri Hocquenghem Codes*, IEEE Transactions on Information Theory, **11**(4), pp. 580-585, Oct. 1965.
[6] Szwaja, Z., *On Step By Step Decoding of The BCH Binary Codes*, IEEE Transactions on Information Theory, **13**(2), pp.350-351, Apr. 1967
[7] Wei, S.W. & Wei, C.H., *A High Speed Real Time Binary BCH Decoder*, IEEE Transactions on Circuits and Systems for Video Technology, **3**(2), pp.138-147,April 1993.
[8] Massey, J.L., Shift-Register Synthesis and BCH Decoding, *IEEE Transactions on Information Theory,* **15**(1), pp. 122-127, Jan.1969.
[9] Hong, J. & Vetterli, M., *Simple Algorithms for BCH Decoding*, IEEE Transactions on Communications, **43**(8), pp. 2324-2333, Aug. 1995.
[10] Carter, N., *Schaum's Outlines of Theory and Problems of Computer Architecture*, Indian Special Edition, McGraw Hill, 2002.
[11] Kumar, A. & Sawitzki, S., *High-Throughput and Low-Power Architectures for Reed Solomon Decoder*, Proc. of IEEE 39th Asilomar Conference on Signal, Systems, and Computers, pp.990-994, Nov. 2005.