International Journal of VLSI System Design and Communication Systems

ISSN 2322-0929
Vol.04, Issue.09,
September-2016,
Pages:0781-0788

www.ijvdcs.org

An Efficient Fault Tolerant Parallel Filters Based on Error Correction Codes

B. UMADEVI[1], RAMESH BABU[2]

[1]PG Scholar, Dept of ECE (VLSI), GATES Institute of Technology, Gooty, Andhra Pradesh, India.
[2]Assistant Professor, Dept of ECE, GATES Institute of Technology, Gooty, Andhra Pradesh, India.

Abstract: The complexity of communications and signal processing circuits increases every year. This is made possible by the CMOS technology scaling that enables the integration of more and more transistors on a single device. This increased complexity makes the circuits more vulnerable to errors. At the same time, the scaling means that transistors operate with lower voltages and are more susceptible to errors caused by noise and manufacturing variations. As technology scales, it enables more complex systems that incorporate many filters. In those complex systems, it is common that some of the filters operate in parallel. Soft errors pose a reliability threat to modern electronic circuits. This makes protection against soft errors a requirement for many applications. Communications and signal processing systems are no exceptions to this trend. For some applications, an interesting option is to use algorithmic-based fault tolerance (ABFT) techniques that try to exploit the algorithmic properties to detect and correct errors. Signal processing and communication applications are well suited for ABFT. A general scheme to protect parallel filters is presented. Parallel filters with the same response that process different input signals are considered. The new approach is based on the application of error correction codes (ECCs) using each of the filter outputs as the equivalent of a bit in and ECC codeword. This is a generalization of the scheme presented and enables more efficient implementations when the number of parallel filters is large. The scheme can also be used to provide more powerful protection using advanced ECCs that can correct failures in multiples modules.

Keywords: AES, Effective Implementation, Algorithm, Reversible, Logic, Xilinx, ISE.

I. INTRODUCTION

FIR filters are one of two primary types of digital filters used in digital signal processing (DSP) applications, the other type being IIR. High performance FIR filters have applications in several video processing and digital communications systems. In some applications, the FIR filter circuit must be able to operate at high sample rates, while in other applications, the FIR filter circuit must be a low-power circuit operating at moderate sample rates. The low-power or low-area techniques developed specifically for digital filters can be found in [1, 2, 3, 4, 5, 6, 7]. Traditional FIR filter uses some parallel processing technique to either increase the effective throughput or to reduce the power consumption of the original filter. Parallel processing involves the replication of hardware units. Here the hardware implementation cost is directly proportional to the block size. At the same time if the design area is very limited this technique is not applicable. Therefore, in order to reduce the chip size and to limit the silicon area required to implement the FIR filter it is necessary to realize a new parallel FIR filtering structure that consume less area than traditional parallel FIR filtering. It is common in DSP to say that a filter input and output signals are in time domain. This is because signals are usually created by sampling at regular intervals of time. But this is not the only way sampling can take place. The second most common way of sampling is at equal intervals in space. For example imagine taking simultaneous readings from an array of strain sensors mounted at one centimeter increments along the length of an aircraft wing.

Many other domains are possible; however, time and space are by far the most common. When you see the term time domain in DSP, remember that it may actually refer to samples taken over time, or it may be a general reference to any domain that the samples are taken in. Every linear filter has an impulse response, a step response and a frequency response. Each of these responses contains complete information about the filter, but in a different form. If one of three is specified, the other two are fixed and can be directly calculated. All three of these representations are important, because they describe how the filter will react under different circumstances. The most straightforward way to implement a digital filter is by convolving the input signal with the digital filter's impulse response. All possible linear filters can be made in this manner. When the impulse response is used in this way, filters designers give it a special name: the filter kernel. There is also another way to make digital filters, called recursion. When a filter is implemented by a convolution, each sample in the output is calculated by weighting the samples in the input, and adding then together. Recursive filters are an extension of this, using previously calculated values from the output, besides points from the input. Instead of using a filter kernel, recursive filters are defined by a set of recursion coefficients. For now the important point is that all linear filters have an impulse response, even if you don't use it to implement the filter. To find the impulse response of a recursive filter, simply feed in the impulse and see what comes out.

The impulse responses of recursive filters are composed of sinusoids that exponentially decay in amplitude. In principle, this makes their impulse responses infinitely long. However the amplitude eventually drops below the round off noise of the system, and the remaining samples can be ignored. Because of these characteristics, recursive filters are also called Infinite impulse response or IIR filters. In comparison, filters carried out by convolution are called Finite impulse response or FIR filters.

II. ERROR CORRECTING CODES

A. Introduction

Codes that correct errors are essential to modern civilization and are used in devices from modems to planetary satellites. The theory is mature, difficult, and mathematically oriented, with tens of thousands of scholarly papers and books, but this project will describe only a simple and elegant code, discovered in 1949.

B. Literature survey

A burst of length I is defined as a vector whose nonzero components are confined to /consecutive digit positions, the first and last of which are nonzero. For example, the error vector **e** = (0 0 0 0 1 0 1 1 0 1 0 0 0 0 0) is a burst of length 6. A linear code that is capable of correcting all error bursts of length /or less but not all error bursts of length /+ 1 is called an 1-burst-error-correcting code, or the code is said to have burst-error-correcting capability 1. It is clear that for given code length it and burst-error-correcting capability 1, it is desirable to construct an (a, k) code with as small a redundancy n - k as possible. Next, we establish certain restrictions on n - k for given 1, or restrictions on I for Given 17 - k.

THEOREM 1: A necessary condition for an (a, k) linear code to be able to correct all burst errors of length I or less is that no burst of length 2/or less can be a codeword.

Proof: Suppose that there exists a burst r of length 2/or less as a codeword. This codeword v can be expressed as a vector sum of two bursts a and w of length 1 or less (except the degenerate case, in which v is a burst of length 1). Then, a and w must be in the same coset of a standard array for this code. If one of these two vectors is used as a coset leader (correctable error pattern), the other will be an uncorrectable error burst. As a result, this code will not be able to correct all error bursts of length /or less. Therefore, in order to correct all error bursts of length /or less, no burst of length 2/or less can be a codeword.

THEOREM 2: The number of parity-check digits of an (n, k) linear code that has no burst of length b or less as a codeword is at least b (i.e., - k > b).

Proof: Consider the vectors whose nonzero components are confined to the first b digit positions. There are a total of 2b of them. N_0 two such vectors can be in the same coset of a standard array for this code otherwise, their vector sum, which is a burst of length b or less, would be a codeword. Therefore, these 2/) vectors must be in '21) distinct cosets. There are a total of ',"-k cosets for an. (a, k) code. Thus, a - k must be at least equal to b (i.e., a - k > b). It follows from Theorems 1 and 2 that there must be a restriction on the number of parity-check digits of an 1-burst-error-correcting code.

THEOREM 3: The number of parity-check digits of an l-burst-error-correcting code must be at least 2/; that is,

Proof: For a given n and k, Theorem 3 implies that the burst-error-correcting capability of an (a k) code is at most [(r - k)/2_l that is,- k this is an upper bound on the burst-error-correcting capability of an (n, k) code and is called the Reiger bound [5]. Codes that meet the Reiger bound are said to be optimal. The ratio is used as a measure of the burst-error-correcting efficiency of a code. An optimal code has burst-error-correcting efficiency equal to 1.

It is possible to show that if an (a, k) code is designed to correct all burst errors of length /or less and simultaneously to detect all burst errors of length d > / or less, the number of parity-check digits of the code must be at least I d.An/-burst-error-correcting cyclic code can most easily be decoded by the error-trapping technique presented in Section 5.7, with a slight variation. Suppose that a codeword y(X) from an /-burst-error-correcting (a, k) cyclic code is transmitted. Let r(X) and e(X) be the received and error vectors, respectively. Let be the syndrome of T(X). If the errors in e(X) are confined to the 1 high-order parity-check digit positions, X"-1.-1, …, X"-k-2, X"-k-1, then the /high-order syndrome digits,—..5,,-k-1, - s,,-k-i, match the errors of e(X), and the a - k – 1 low-order syndrome digits, so, St, ° s,,-k--/-1, are zeros. Suppose that the errors in e(X) are not confined to the positions X"-k-i, •—..,, X" k 2, X"-k-1 of r(X) but are confined to /consecutive positions of r(X) (including the end-around case). Then, after a certain number of cyclic shifts of r(X), say i cyclic shifts, the errors will be shifted to the positions X"-k-/…, X"-k-2, X"-k-I. of T(i) (X), the ith shift of T(X). Let s(i) (X) be the syndrome of v(i) (X). Then, the first /high-order digits of sO (X) match the errors at the positions X"-k-1, …, X"-k-2, X"-k-I of z(i) (X), and the rr - k - /low-order digits of s(i) (X) are zeros. Using these facts, we may trap the errors in the syndrome register by cyclic shifting r(X). An error-trapping decoder for an /-burst-correcting cyclic code is shown in Fig.1, where the received vector is shifted into the syndrome register from the left end. The decoding procedure is as follows:

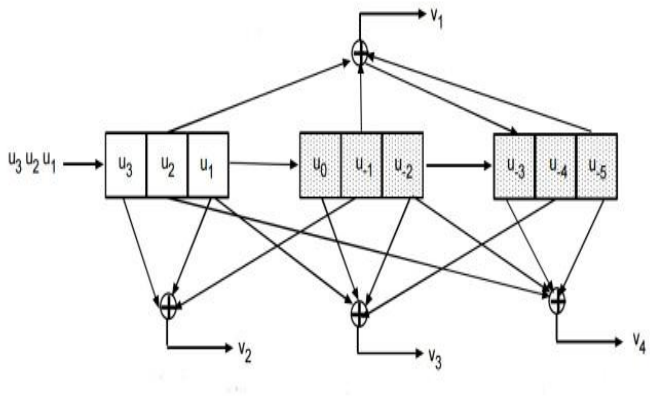


Fig.1. An error-trapping decoder for burst-error-correcting codes.

Step 1: The received vector r(X) is shifted into the syndrome and buffer registers simultaneously. (If we do not want to decode the received parity-check digits, the buffer register needs only k stages.) As soon as r(X) has been shifted into the syndrome register, the syndrome s(X) is formed.

Step 2: The syndrome register starts to shift with gate 2 on. As soon as its - k – 1 leftmost stages contain only zeros, its 1 rightmost stages contain the burst-error pattern. The error correction begins. There are three cases to be considered.

Step 3: If the n - k – 1 leftmost stages of the syndrome register contain all zeros after the ith shift for 0 < i < n - k - I, the errors of the burst e(X) are confined to the parity-check positions of r(X). In this event, the k received information digits in the buffer register are error-free. Gate 4 is then activated, and the k error-free information digits in the buffer are shifted out to the data sink If the a - k -1 leftmost stages of the syndrome register never contain all zeros during the first n - k -1 shifts of the syndrome register, the error burst is not confined to the - k parity-check positions of r(X).

Step 4: If the n - k -1 leftmost stages of the syndrome register contain all zeros after the (7 - k -1+ i)th shift of the syndrome register for 1 < i < 1, the error burst is confined to positions X"', …• Xi-1, of r(X). (This is an end-around burst). In this event, the /- i digits contained in the 1- i rightmost stages of the syndrome register match the errors at the parity-check positions, X°, X I, …, X/-i-1 of r(X), and the i digits contained in the next i stages of the syndrome register match the errors at the positions X"-', …, X"-2, X"-1 of r(X). At this instant, a clock starts to count from (n-k-l+i+1). The syndrome register is then shifted (in step with the clock) with gate 2 turned off. As soon as the clock has counted up to a - k, the i rightmost digits in the syndrome register match the errors at the positions X"-i, …, X"-2, X"-1 of r(X). Gates 3 and 4 are then activated. The received information digits are read out of the buffer register and corrected by the error digits shifted out from Tec syndrome register.

Step 5: If the a - k – 1 leftmost stages of the syndrome register never contain all zeros by the time that the syndrome register has been shifted a k times, the received information highs are read out of the buffer register one at a time with. gate 4 activated. At the same time the syndrome register is shifted with gate 2 activated. As soon as the k - j leftmost stages, of the syndrome register contain all zeros, tire digits in the rightmost stages of the syndrome register match the errors in the net received information digits to come out of the buffer register. Gate 3 is then activated, and the erroneous information digits are corrected by the digits coming out from the syndrome register with gate 2 disabled.

If the n–k–l leftmost stages of the syndrome register never contain all zeros by the time the k information digits have been read out of the buffer, an uncorrectable burst of errors has been detected. With the decoder just described, the decoding process takes Pat clock cycles; the first a clock cycles are required for syndrome computation, and the next a clock cycles are needed for error trapping and error correction. The n clock cycles for syndrome computation are concurrent with the reception of the received vector from the channel; no time delay occurs in this operation. The second a clock cycles for error trapping and correction represent decoding delay. In this decoder the received vector is shifted into the syndrome register from the left end. If the received vector is shifted into the syndrome register from the right end, the decoding operation will be slightly different. This decoder corrects only burst errors of length /or less. The number of these burst-error patterns is n21/2', which for large a, is only a small fraction of 2"2 correctable error patterns (coset leaden). It is possible to modify the decoder is such a way that it corrects all the correctable -burst errors of length a - k or less. That is, besides correcting all the bursts of length/or less, the decoder also corrects those bursts of length /+ 1 to a - It that are used as coset leaders. This modified decoder operates as follows. The entire received rector is lint shifted into the syndrome register. Before performing the error Correction, the syndrome register is cyclically shifted a time (with feedback connections operative). During this cycling the length b of the shortest burst that appears in the h rightmost stages of the syndrome register is recorded by a Counter. This burst is assumed to be the error burst added by the channel. Having completed these pre-correction shifts, the decoder begins its correction process. The syndromes register starts to shift again. As soon as the shortest burst reappears in the b rightmost stages of the syndrome register, the decoder starts to make corrections as described earlier. This decoding is an optimum decoding for burst-error-correcting codes that was proposed by Gallager.

C. Description of the Hamming Code

Richard Hamming found a beautiful binary code that will correct any single error and will detect any double error (two separate errors).The Hamming code has been used for computer RAM, and is a good choice for randomly occurring errors. (If errors come in bursts, there are other good codes.) Unlike most other error-correcting codes, this one is simple to understand. The code uses extra redundant bits to check for errors, and performs the checks with special check equations. A parity check equation of a sequence of bits just adds the bits of the sequence and insists that the sum be even (for even parity) or odd (for odd parity). This section uses even parity. Alternatively, one says that the sum is taken modulo **2** (divide by **2** and take the remainder), or one says that the sum is taken over the integers mod **2**, $\mathbf{Z}_2$. A simple parity check will detect if there has been an error in one bit position, since even parity will change to odd parity. (Any odd number of errors will show up as if there were just 1 error, and any even number of errors will look the same as no error). One has to force even parity by adding an extra parity bit and setting it either to **1** or to **0** to make the overall parity come out even. It is important to realize that the extra parity check bit participates in the check and is itself checked for errors, along with the other bits.

The Hamming code uses parity checks over a portion of the positions in a block. Suppose there are bits in consecutive positions from **1** to **n-1**. The positions whose position number is a power of **2** are used as check bits, whose value must be determined from the data bits. Thus the check bits are in positions **1**, **2**, **4**, **8**, **16**, ..., up to the largest power of **2** that is less than or equal to the largest bit position. The remaining positions are reserved for data bits. Each check bit has a corresponding check equation that covers a portion of all the bits, but always includes the check bit itself. Consider the binary representation of the position numbers: $\mathbf{1 = 1_2}$, $\mathbf{2 = 10_2}$, $\mathbf{3 = 11_2}$, $\mathbf{4 = 100_2}$, $\mathbf{5 = 101_2}$, $\mathbf{6 = 110_2}$, and so forth. If the position number has a **1** as its rightmost bit, then the check equation for check bit **1** covers those positions. If the position number has a **1** as its next-to-rightmost bit, then the check equation for check bit **2** covers those positions. If the position number has a **1** as its third-from-rightmost bit, then the check equation for check bit **4** covers those positions. Continue in this way through all check bits. Table 1 summarizes this pattern.

Table 1. Position of the parity checks for the first 17 positions of the Hamming code (Check bits are in positions 1, 2, 4, 8, and 16, in red italic).

Position	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Bin Rep	1	10	11	100	101	110	111	1000	1001	1010	1011	1100	1101	1110	1111	10000	10001
Check:1	X		X		X		x		X		X		X		X		X
Check:2		X	X			X	x			X	X			X	X		
Check:4				X	X	X	x					X	X	X	X		
Check:8								X	X	X	X	X	X	X	X		
Check:16																X	X

Table 2: The below table assumes one starts with data bits **1101101** (in black below). The check equations above are used to determine values for check bits in positions **1**, **2**, **4**, and **8**, to yield the word **11101010101** below, with check bits in red italic here and below.

Table 2. Implementation of Hamming code for data bits 1101101.

Position	1	2	3	4	5	6	7	8	9	10	11
Binary	1	10	11	100	101	110	111	1000	1001	1010	1011
Word	1	1	1	0	1	0	1	0	1	0	1
Check:1	1		1		1		1		1		1
Check:2		1	1			0	1			0	1
Check:4				0	1	0	1				
Check:8								0	1	0	1

Intuitively, the check equations allow one to ``zero-in" on the position of a single error. For example, suppose a single bit is transmitted in error. If the first check equation fails, then the error must be in an odd position, and otherwise it must be in an even position. In other words, if the first check fails, the position number of the bit in error must have its rightmost bit (in binary) equal to 1; otherwise it is zero. Similarly the second check gives the next-to-rightmost bit of the position in error, and so forth. Table 3: The below table gives the result of a single error in the decimal position **11** (changed from a**1**toa**0**). Three of the four parity checks fail, as shown below. Adding the decimal position number of each failing check gives the position number of the error bit, decimal **11** in this case the below discussion shows how to get single-error correction with the Hamming code. One can also get double-error detection by using a single extra check bit, which is in position **0**. (All other positions are handled as above.) The check equation in this case covers all bits, including the new bit in position **0**.

Table 3. Results of a single error in decimal position 11

Position	1	2	3	4	5	6	7	8	9	10	11	Result of Check
Binary	1	10	11	100	101	110	111	1000	1001	1010	1011	
Word	1	1	1	0	1	0	1	0	1	0	0 (error)	
Check:1	1		1		1		1		1		0	1 fails
Check:2		1	1			0	1			0	0	2 fails
Check:4				0	1	0	1					4 passes
Check:8								0	1	0	0	8 fails

In case of a single error, this new check will fail. If only the new equation fails, but none of the others, then the position in error is the new **0**th check bit, so a single error of this new bit can also be corrected. In case of two errors, the overall check (using position **0**) will pass, but at least one of the other check equations must fail. This is how one detects a double error. In this case there is not enough information present to say anything about the positions of the two bits in error. Three or more errors at the same time can show up as no error, as two errors detected, or as a single error that is ``corrected" with a bogus correction. Notice that the Hamming code without the extra **0**th check bit would correct a double error in some bogus position as if it were a single error. Thus the extra check bit and the double error detection are very important for this code. Notice also that the check bits themselves will also be corrected if one of them is transmitted in error (without any other errors).

III. IMPLEMENTATION OF PROPOSED SYSTEM

A discrete time filter implements the following equation:

$$y[n] = \sum_{l=0}^{\infty} x[n-l]\,.\,h[l] \qquad \mathbf{(1)}$$

where $x[n]$ is the input signal, $y[n]$ is the output, and $h[l]$ is the impulse response of the filter. When the response $h[l]$ is nonzero, only for a finite number of samples, the filter is known as a FIR filter, otherwise the filter is an infinite impulse response (IIR) filter. There are several structures to implement both FIR and IIR filters.

In the following, a set of k parallel filters with the same response and different input signals are considered. These parallel filters are illustrated in Fig. 2. This kind of filter is found in some communication systems that use several channels in parallel. In data acquisition and processing applications is also common to filter several signals with the same response. An interesting property for these parallel filters is that the sum of any combination of the outputs $y_i\,[n]$ can also be obtained by adding the corresponding inputs $x_i\,[n]$ and filtering the resulting signal with the same filter $h[l]$. For example

$$y_1[n]+y_2[n]=\sum_{l=0}^{\infty}(x_1[n-l]+x_2[n-l]).h[l] \qquad \textbf{(2)}$$

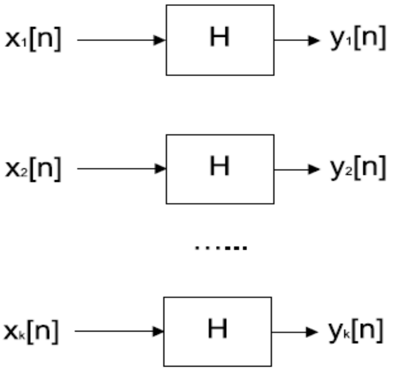


Fig.2. Parallel filters with the same response.

This simple observation will be used in the following to develop the proposed fault tolerant implementation. The new technique is based on the use of the ECCs. A simple ECC takes a block of k bits and produces a block of n bits by adding $n-k$ parity check bits. The parity check bits are XOR combinations of the k data bits. By properly designing those combinations it is possible to detect and correct errors. As an example, let us consider a simple Hamming code with $k = 4$ and $n = 7$. In this case, the three parity check bits $p1$, $p2$, $p3$ are computed as a function of the data bits d_1, d_2, d_3, d_4 as follows:

$$\begin{aligned} p1 &= d_1 \oplus d_2 \oplus d_3 \\ p2 &= d_1 \oplus d_2 \oplus d_4 \\ p3 &= d_1 \oplus d_3 \oplus d_4 \end{aligned} \qquad \textbf{(3)}$$

The data and parity check bits are stored and can be recovered later even if there is an error in one of the bits. This is done by re-computing the parity check bits and comparing the results with the values stored. In the example considered, an error on d_1 will cause errors on the three parity checks; an error on d_2 only in $p1$ and $p2$; an error on d_3 in $p1$ and $p3$; and finally an error on d_4 in $p2$ and $p3$. Therefore, the data bit in error can be located and the error can be corrected. This is commonly formulated in terms of the generating G and parity check H matrixes. For the Hamming code considered in the example, those are

$$G=\begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 & 1 \end{bmatrix} \qquad \textbf{(4)}$$

$$H=\begin{bmatrix} 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 1 \end{bmatrix} \qquad \textbf{(5)}$$

Encoding is done by computing $y = x \bullet G$ and error detection is done by computing $s = y \bullet H^T$, where the operator • is based on module two addition (XOR) and multiplication. Correction is done using the vector s, known as syndrome, to identify the bit in error. The correspondence of values of s to error position is captured in Table 4.

TABLE 4. Error Location in the Hamming Code

$s_1\,s_2\,s_3$	Error Bit Position	Action
0 0 0	No error	None
1 1 1	d_1	correct d_1
1 1 0	d_2	correct d_2
1 0 1	d_3	correct d_3
0 1 1	d_4	correct d_4
1 0 0	p_1	correct p_1
0 1 0	p_2	correct p_2
0 0 1	p_3	correct p_3

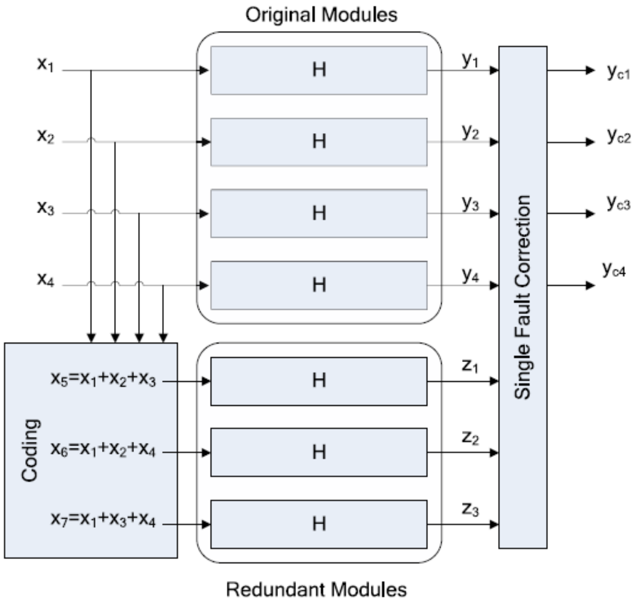


Fig.3. Proposed scheme for four filters and a Hamming code

Once the erroneous bit is identified, it is corrected by simply inverting the bit. This ECC scheme can be applied to the parallel filters considered by defining a set of check filters z_j. For the case of four filters y_1, y_2, y_3, y= and the Hamming code, the check filters would be

$$z_1[n]=\sum_{l=0}^{\infty}(x_1[n-l]+x_2[n-l]+x_3[n-l]).h[l]$$
$$z_2[n]=\sum_{l=0}^{\infty}(x_1[n-l]+x_2[n-l]+x_4[n-l]).h[l]$$
$$z_3[n]=\sum_{l=0}^{\infty}(x_1[n-l]+x_3[n-l]+x_4[n-l]).h[l] \qquad \textbf{(6)}$$

and the checking is done by testing if

$$\begin{aligned} z_1[n] &= y_1[n]+y_2[n]+y_3[n] \\ z_2[n] &= y_1[n]+y_2[n]+y_4[n] \\ z_3[n] &= y_1[n]+y_3[n]+y_4[n] \end{aligned} \qquad \textbf{(7)}$$

For example, an error on filter y_1 will cause errors on the checks of z_1, z_2, and z_3. Similarly, errors on the other filters will cause errors on a different group of z_i. Therefore, as with the traditional ECCs, the error can be located and corrected. The overall scheme is illustrated on Fig. 3. It can be observed that correction is achieved with only three redundant filters. For the filters, correction is achieved by reconstructing the erroneous outputs using the rest of the data and check outputs. For example, when an error on y_1 is detected, it can be corrected by making

$$y_{c1}[n] = z_1[n] - y_2[n] - y_3[n] \tag{8}$$

Similar equations can be used to correct errors on the rest of the data outputs. In our case, we can define the check matrix as

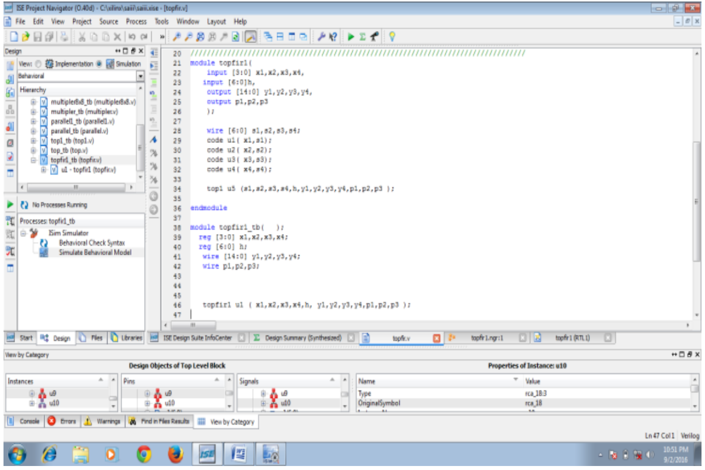
$$H = \begin{bmatrix} 1 & 1 & 1 & 0 & -1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & -1 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & -1 \end{bmatrix} \tag{9}$$

and calculate $s = yH^T$ to detect errors. Then, the vector s is also used to identify the filter in error. In our case, a nonzero value in vector s is equivalent to 1 in the traditional Hamming code. A zero value in the check corresponds to a 0 in the traditional Hamming code.

It is important to note that due to different finite precision effects in the original and check filter implementations, the comparisons in (7) can show small differences. Those differences will depend on the quantization effects in the filter implementations that have been widely studied for different filter structures. The interested reader is referred to for further details. Therefore, a threshold must be used in the comparisons so that values smaller than the threshold are classified as 0. This means that small errors may not be corrected. This will not be an issue in most cases as small errors are acceptable. The detailed study of the effect of these small errors on the signal to noise ratio at the output of the filter is left for future work. The reader can get more details on this type of analysis. With this alternative formulation, it is clear that the scheme can be used for any number of parallel filters and any linear block code can be used. The approach is more attractive when the number of filters k is large. For example, when k = 11, only four redundant filters are needed to provide single error correction. This is the same as for traditional ECCs for which the overhead decreases as the block size increases.

The additional operations required for encoding and decoding are simple additions, subtractions, and comparisons and should have little effect on the overall complexity of the circuit. This is illustrated in which a case study is presented. In the discussion, so far the effect of errors affecting the encoding and decoding logic has not been considered. The encoder and decoder include several additions and subtractions and therefore the possibility of errors affecting them cannot be neglected. Focusing on the encoders, it can be seen that some of the calculations of the z_i share adders. For example, looking at (6), z_1 and z_2 share the term $y_1 + y_2$. Therefore, an error in that adder could affect both z_1 and z_2 causing a mis-correction on y_2. To ensure that single errors in the encoding logic will not affect the data outputs, one option is to avoid logic sharing by computing each of the z_i independently. In that cases, errors will only affect one of the z_i outputs and according to Table I, the data outputs y_j will not be affected. Similarly, by avoiding logic sharing, single errors in the computation of the s vector will only affect one of its bits. The final correction elements such as that in (8) need to be tripled to ensure that they do not propagate errors to the outputs. However, as their complexity is small compared with that of the filters, the impact on the overall circuit cost will be low. This is confirmed by the results presented for a case study.

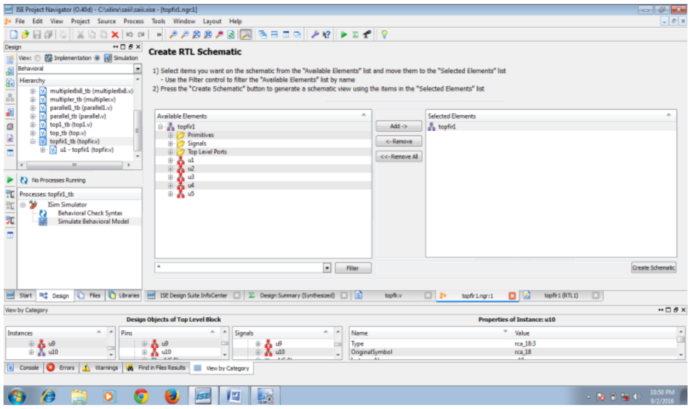
IV, SIMULATION RESULTS



```
///////////////////////////////////////////////////////////////////////////////
module topfir1(
    input [3:0] x1,x2,x3,x4,
   input [6:0]h,
    output [14:0] y1,y2,y3,y4,
    output p1,p2,p3
    );

    wire [6:0] s1,s2,s3,s4;
    code u1( x1,s1);
    code u2( x2,s2);
    code u3( x3,s3);
    code u4( x4,s4);

    top1 u5 (s1,s2,s3,s4,h,y1,y2,y3,y4,p1,p2,p3 );

endmodule

module topfir1_tb(   );
  reg [3:0] x1,x2,x3,x4;
  reg [6:0] h;
   wire [14:0] y1,y2,y3,y4;
   wire p1,p2,p3;



   topfir1 u1 ( x1,x2,x3,x4,h, y1,y2,y3,y4,p1,p2,p3 );
```

File Edit View Project Source Process Tools Window Layout Help
Design Objects of Top Level Block
Properties of Instance: u10
Instances
Pins
Signals
Name
Value
Type
rca_18:3
OriginalSymbol
rca_18
Console
Errors
Warnings
Find in Files Results
View by Category
ISE Design Suite InfoCenter
Design Summary (Synthesized)
topfir.v
topfir1.ngr:1
topfir1 (RTL1)

Fig.4. Program Window.



Create RTL Schematic

1) Select items you want on the schematic from the "Available Elements" list and move them to the "Selected Elements" list
   - Use the Filter control to filter the "Available Elements" list by name
2) Press the "Create Schematic" button to generate a schematic view using the items in the "Selected Elements" list

Fig.5. RTL Schematic generater

In this Fig6 shows, the block diagram of a top level circuit whitch consisting of inputs of parallel filter and outputs of parallel filter. X1,X2,X3,X4 are the input of the parallel filter,h is the impulse response of the parallel filter,Y1,Y2,Y3,Y4 are the output of the parallel filter,P1,P2,P3, are thecheck bits of the output response.

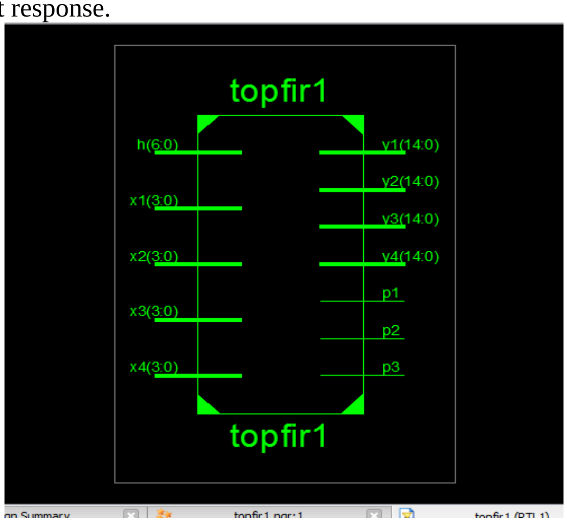


Fig.6. RTL Schematic of Parallel Filter.

In this Fig7 shows, the RTL Schematic of a top module circuit whitch consisting of internal schematic of acircuit.

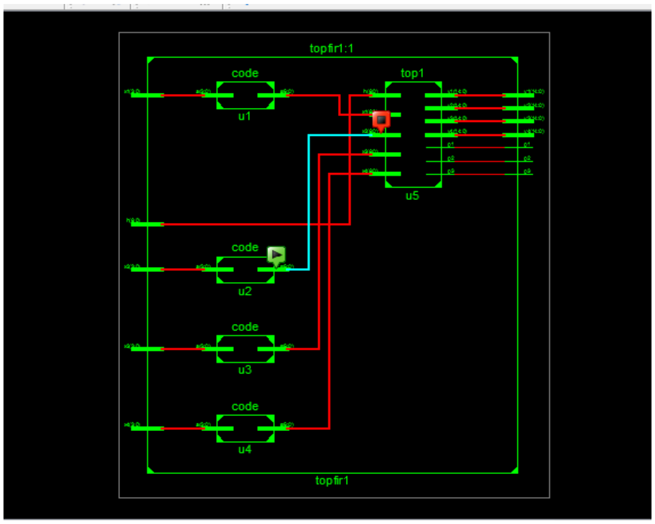


Fig.7. Internal RTL Schematic.

RTL Schematic for Parallel Filtesr trigger:

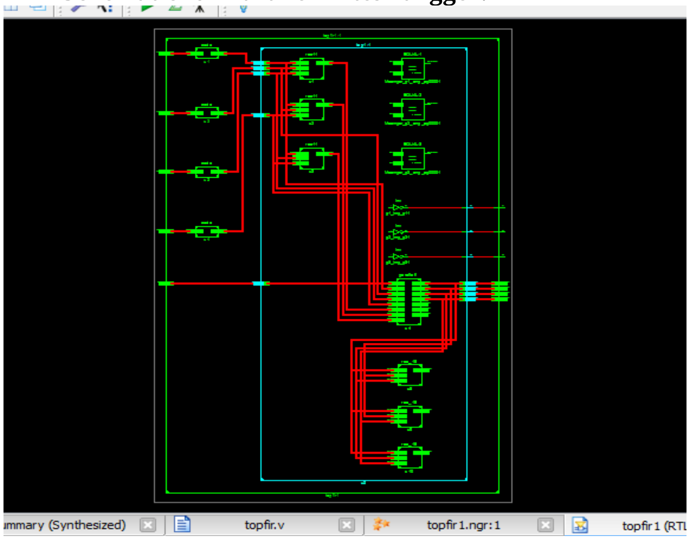


Fig.8. RTL Schematic Trigger

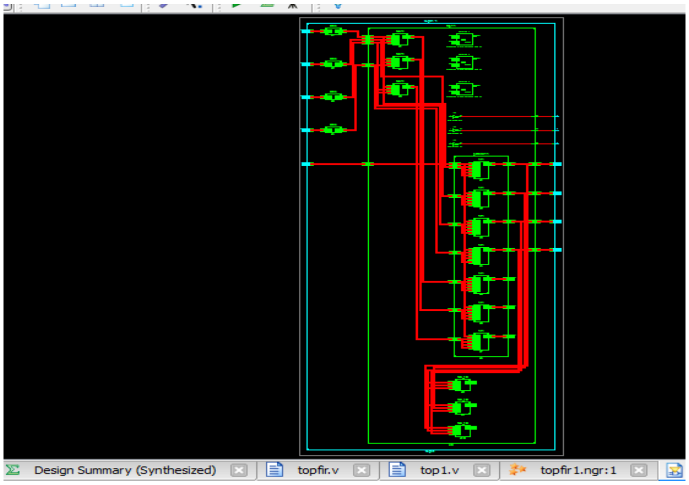


Fig.9. RTL Schematic overview

V. CONCLUSION AND FUTURE SCOPE

A. Conclusion

This brief has presented a new scheme to protect parallel filters that are commonly found in modern signal processing circuits. The approach is based on applying ECCs to the parallel filters outputs to detect and correct errors. The scheme can be used for parallel filters that have the same response and process different input signals. A case study has also been discussed to show the effectiveness of the scheme in terms of error correction and also of circuit overheads. The technique provides larger benefits when the number of parallel filters is large.

B. Future Scope

The proposed scheme can also be applied to the IIR filters. Future work will consider the evaluation of the benefits of the proposed technique for IIR filters. The extension of the scheme to parallel filters that have the same input and different impulse responses is also a topic for future work. The proposed scheme can also be combined with the reduced precision replica approach presented in [3] to reduce the overhead required for protection. This will be of interest when the number of parallel filters is small as the cost of the proposed scheme is larger in that case. Another interesting topic to continue this brief is to explore the use of more powerful multi bit ECCs, such as Bose–Chaudhuri–Hocquenghem codes, to correct errors on multiple filters.

VI. REFERENCES

[1] M. Nicolaidis, “Design for soft error mitigation,” IEEE Trans. Device Mater. Rel., vol. 5, no. 3, pp. 405–418, Sep. 2005.

[2] A. Reddy and P. Banarjee “Algorithm-based fault detection for signal processing applications,” IEEE Trans. Comput., vol. 39, no. 10, pp. 1304–1308, Oct. 1990.

[3] B. Shim and N. Shanbhag, “Energy-efficient soft error-tolerant digital signal processing,” IEEE Trans. Very Large Scale Integr. (VLSI) Syst., vol. 14, no. 4, pp. 336–348, Apr. 2006.

[4] T. Hitana and A. K. Deb, “Bridging concurrent and non-concurrent error detection in FIR filters,” in Proc. Norchip Conf., 2004, pp. 75–78.

[5] Y.-H. Huang, “High-efficiency soft-error-tolerant digital signal processing using fine-grain subword-detection processing,” IEEE Trans. Very Large Scale Integr. (VLSI) Syst., vol. 18, no. 2, pp. 291–304, Feb. 2010.

[6] S. Pontarelli, G. C. Cardarilli, M. Re, and A. Salsano, “Totally fault tolerant RNS based FIR filters,” in Proc. IEEE IOLTS, Jul. 2008, pp. 192–194.

[7] Z. Gao, W. Yang, X. Chen, M. Zhao, and J. Wang, “Fault missing rate analysis of the arithmetic residue codes based fault-tolerant FIR filter design,” in Proc. IEEE IOLTS, Jun. 2012, pp. 130–133.

[8] P. Reviriego, C. J. Bleakley, and J. A. Maestro, “Strutural DMR: A technique for implementation of soft-error-tolerant FIR filters,” IEEE Trans. Circuits Syst., Exp. Briefs, vol. 58, no. 8, pp. 512–516, Aug. 2011.

[9] P. P. Vaidyanathan. Multirate Systems and Filter Banks. Upper Saddle River, NJ, USA: Prentice-Hall, 1993.

[10] A. Sibille, C. Oestges, and A. Zanella, MIMO: From Theory to Implementation. San Francisco, CA, USA: Academic Press, 2010.
[11] P. Reviriego, S. Pontarelli, C. Bleakley, and J. A. Maestro, "Area efficient concurrent error detection and correction for parallel filters," IET Electron. Lett., vol. 48, no. 20, pp. 1258–1260, Sep. 2012.
[12] A. V. Oppenheim and R. W. Schafer, Discrete Time Signal Processing. Upper Saddle River, NJ, USA: Prentice-Hall 1999.
[13] S. Lin and D. J. Costello, Error Control Coding, 2nd ed. Englewood Cliffs, NJ, USA: Prentice-Hall. 2004.
[14] R. W. Hamming, "Error correcting and error detecting codes," Bell Syst. Tech. J., vol. 29, pp. 147–160, Apr. 1950.