High Throughput VLSI Architectures for CRC/BCH Encoders and FFT computations

A THESIS
SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL
OF THE UNIVERSITY OF MINNESOTA
BY

Manohar Ayinala

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
Master Of Science

Keshab K. Parhi

December, 2010

Acknowledgements

I would like to acknowledge the support of my adviser, Prof. Keshab K. Parhi in guiding and providing the necessary direction throughout this work.

I would like to acknowledge Michael Brown for providing the help and contribution to this work. And also all those people who directly and indirectly have helped me in pursuing this work.

The work on FFT architectures is carried out at Leanics Corporation, Minneapolis.

Dedication

I would like to dedicate this work to my family whose support has helped me to achieve my goals and are responsible for what I am today.

Abstract

Linear feedback shift register (LFSR) is an important component of the cyclic redundancy check (CRC) operations and BCH encoders. This thesis presents a mathematical proof of existence of a linear transformation to transform LFSR circuits into equivalent state space formulations. This transformation achieves a full speed-up compared to the serial architecture at the cost of an increase in hardware overhead. This method applies to all irreducible polynomials used in CRC operations and BCH encoders. A new formulation is proposed to modify the LFSR into the form of an IIR filter. We propose a novel high speed parallel LFSR architecture based on parallel Infinite Impulse Response (IIR) filter design, pipelining and retiming algorithms. The advantage of proposed approach over the previous architectures is that it has both feedforward and feedback paths. We further propose to apply combined parallel and pipelining techniques to eliminate the fanout effect in long generator polynomials. The proposed scheme can be applied to any generator polynomial, i.e., any LFSR in general. A comparison between the proposed and previous architectures shows that the proposed parallel architecture achieves the same critical path as that of previous designs with a reduced hardware cost.

Further, this thesis presents a novel approach to develop the pipelined architectures for the fast Fourier transform (FFT). A formal procedure for designing FFT architectures using folding transformation and register minimization techniques is proposed. Novel parallel-pipelined architectures for the computation of fast Fourier transform are derived. The proposed architecture takes advantage of under utilized hardware in the serial architecture to derive L-parallel architectures without increasing the hardware complexity by a factor of L. The operating frequency of the proposed architecture can be decreased which in turn reduces the power consumption. A comparison is drawn between the proposed designs and the previous architectures. The power consumption can be reduced up to 37% and 50% in 2-parallel FFT architectures. The output samples are obtained in a scrambled order in the proposed architectures. Circuits to reorder these scrambled sequences to desired order are presented.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Communication standards continue to be defined that push the bar higher for throughput. For example, 10 Gbps IEEE 802.3ak was standardized in 2003, and recently 100 Gbps IEEE 802.3ba is standardized in 2010. In order to support these high throughput requirements at a reasonable frequency, parallel architectures are required. At the same time, the power consumption and hardware overhead should be kept to a minimum. The research in this thesis is directed towards designing high throughput architectures for two key components of the modern communication standards, CRC/BCH encoders and Fast Fourier Transform (FFT).

Cyclic Redundancy Check (CRC) is widely used in data communications and storage devices as an efficient way to detect transmission errors. Examples of digital communication standards that employ CRC include Asynchronous Transfer Mode (ATM), Ethernet (IEEE 802.3), WiFi (IEEE 802.11) and WiMAX (802.16). The Bose-Chaudhuri-Hochquenghem (BCH) codes are one of the most powerful algebraic codes and are extensively used in modern communication systems. Compared to Reed-Solomon codes, BCH codes can achieve around additional 0.6dB coding gain over the additive white Gaussian noise (AWGN) channel with similar rate and codeword length. Many applications of BCH codes such as long-haul optical communication systems used in International Telecommunication Union-Telecommunication Standardization sector (ITU-T) G.975, magnetic recording systems, solid-state storage devices and digital communications require high throughput as well as large error correcting capability. Hence, BCH codes are of great interest for their efficient and high speed hardware encoding and decoding implementation.

The BCH encoders and CRC operations are conventionally implemented by a linear feedback shift register (LFSR) architecture. While such an architecture is simple and can run at high frequency, it suffers from serial-in and serial-out limitation. In optical communication systems, where throughput over 1 Gbps is usually desired, the clock frequency of such LFSR based encoders cannot keep up with data transmission rate and thus parallel processing must be employed. Doubling the data width, i.e two parallel architecture doesn't double the throughput, the worst case timing path becomes slower. Since the parallel architectures contain feedback loops, pipelining cannot be applied to reduce the critical path. Another issue with the parallel architectures is hardware complexity. A novel parallel CRC architecture based on state space representation is proposed in the literature. The main advantage of this architecture is that the complexity is shifted out of the feedback loop. The full speedup can be achieved by pipelining the feedforward paths. A state space transformation has been proposed to reduce complexity but the existence of such a transformation was not proved and whether such a transformation is unique has been unknown so far. In this thesis, we present a mathematical proof to show that such a transformation exists for all CRC and BCH generator polynomials. We also show that this transformation is non-unique. In fact, we show the existence of infinite such transformations and how these can be derived. We then propose novel schemes based on pipelining, retiming and look ahead computations to reduce the critical path in the parallel architectures based on parallel and pipelined IIR filter design.

The orthogonal frequency division multiplexing (OFDM) technology has become more and more significant in modern communication systems. The key concept of OFDM technique makes use of multi-carrier modulation. For example, WiMAX, DVB-T2, Wireless LAN, ADSL, and VDSL all utilize OFDM technology for baseband modulations. The OFDM systems need FFT and IFFT processors to perform real-time operations for orthogonal multi-carrier modulations and demodulations. Thus, the design of efficient FFT processor is necessarily required. Much research has been carried out in designing parallel architectures. A formal method of developing these architectures has not been proposed until recently which is based on hypercube methodology. Further, most of these architectures are not fully utilized which leads to high hardware complexity. In this research, we present a new approach to designing FFT architectures from the FFT flow graphs. Folding transformation and register minimization techniques are used to derive several known FFT architectures. Novel architectures are developed using the proposed methodology which have not been presented in the literature.

The thesis is organized in the following way.

- Chapter 2 briefly describes how Linear feedback shift register (LFSR) is used for CRC and BCH encoding and the prior designs. The proof for the existence of state space transformation for LFSR is also presented.
- In Chapter 3 proposed LFSR architectures based on IIR filters are presented. The proposed architectures are compared with the previous designs.
- Chapter 4 briefly describes the FFT algorithms and their prior architectures. Further, the proposed approach for designing these architectures is presented.
- Chapter 5 presents the novel parallel FFT architectures derived via folding for radix-2, radix-2^2 and radix-2^3 algorithms. Comparisons with previous architectures and the power consumption analysis is also presented.
- Chapter 6, Conclusions

Chapter 2

CRC/BCH Encoder Architectures

In this chapter, we describe how CRC operations and BCH encoders are implemented using Linear Feedback Shift registers (LFSR). Some of the prior designs including folding and state space based designs have been described. Further, the proof of existence of state space transformation for the irreducible polynomials is presented.

2.1 Linear Feedback Shift Registers (LFSR)

CRC computations and BCH encoders are implemented by using Linear Feedback Shift Registers (LFSR)[1], [2], [3]. A sequential LFSR circuit cannot meet the speed requirement when high speed data transmission is required. Because of this limitation, parallel architectures must be employed in high speed applications such as optical communication systems where throughput of several gigabits/sec is required. LFSRs are also used in conventional Design for Test (DFT) and Built in Self Test (BIST) [4]. LFSRs are used to carry out response compression in BIST, while for the DFT, it is a source of pseudorandom binary test sequences.

A basic LFSR architecture for K^{th} order generating polynomial in $GF(2)$ is shown in Fig. 2.1. K denotes the length of the LFSR, i.e., the number of delay elements and $g_0, g_1, g_2, ..., g_K$ represent the coefficients of the characteristic polynomial. The

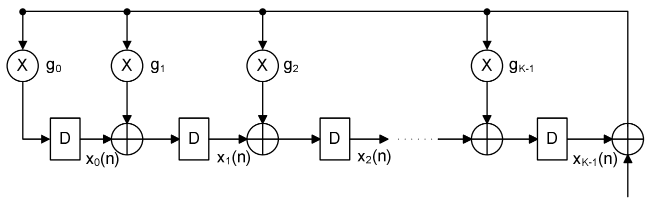


Figure 2.1: Basic LFSR architecture

characteristic polynomial of this LFSR is

$$g(x) = g_0 + g_1x + g_2x^2 + ... + g_Kx^K$$

where $g_0, g_1, g_2, ..., g_K \in GF(2)$. Usually, $g_K = g_0 = 1$. In GF(2), multiplier elements are either open circuits or short circuits i.e., $g_i = 1$ implies that a connection exists. On the other hand $g_i = 0$ implies that no connection exists and the corresponding XOR gate can be replaced by a direct connection from input to output.

Let $u(x)$, for $x = 0, 1, ...N-1$, $u(x) \in GF(2), 0 \leq n \leq N-1$ be input sequence of length N. Both CRC computation and BCH encoding involve the division of the polynomial $u(x)x^K$ by $g(x)$ to obtain the remainder, $Rem(u(x)x^K)_{g(x)}$. During the first N clock cycles, the N-bit message is input to the LFSR with most significant bit (MSB) first. At the same time, the message bits are also sent to the output to form the BCH encoded codeword. After N clock cycles, the feedback is reset to zero and the K registers contain the coefficients of $Rem(u(x)x^K)_{g(x)}$. In BCH encoding, the remaining bits are then shifted out bit by bit to form the remaining systematic codeword bits.

The throughput of the system is limited by the propagation delay around the feedback loop, which consists of two XOR gates. We can increase the throughput by modifying the system to process some number of bits in parallel.

2.2 Prior Designs

In order to meet the increasing demand on processing capabilities, much research has been carried out on parallel architectures of LFSR for CRC and BCH encoders. In [5], first serial to parallel transformation of linear feedback shift register was described and was first applied to CRC computation in [6]. Several other approaches have been recently presented to parallelize LFSR computations [7], [8], [9], [10]. We review some of these approaches below.

2.2.1 Mathematical deduction

In [7] and [8], parallel CRC implementations have been proposed based on mathematical deduction. In these papers, recursive formulations were used to derive parallel CRC architectures. High speed architectures for BCH encoders have been proposed in [8] and [11]. These are based on multiplication and division computations on generator polynomials and can be used for any LFSR of any generator polynomial. They are efficient in terms of speeding up the LFSR but their hardware cost is high. A method to reduce the worst case delay is discussed in [12]. In this paper, CRC is calculated using shorter polynomials that has fewer feedback terms. At the end, a final division is performed on the larger polynomial to get the final result. The implementation is used to process 8 bits in parallel, but large number of parallel bits need larger polynomials which could be problematic to find. The parallel processing leads to a long critical path even though it increases the number of message bits processed. Parallel architecture based on state space representation [13] is described in section III.

2.2.2 Unfolding

Unfolding [14] technique has been used in [8] [9] [10] to implement parallel LFSR architectures. But, when we unfold the LFSR architectures directly, it may lead to a parallel circuit with long critical path. The LFSR architectures can also face fanout issues due to the large number of non-zero coefficients especially in longer generator polynomials. In [8], a new method has been proposed to eliminate the fan-out bottleneck by modifying the generator polynomial. The fanout bottleneck can always be eliminated by retiming. But the unfolded circuit might have the same issue if the number of delays before rightmost XOR gate is less than the unfolding factor J [15].

If the generator polynomial is expressed as

$$g(x) = x^{t_0} + x^{t_1} + ... + x^{t_{m-2}} + 1$$

where $t_0, t_1, ..., t_{m-2}$ are positive integers with $t_0 > t_1 > ... > t_{m-2}$ and m is the total number of nonzero terms of $g(x)$. There are $t_0 - t_1$ consecutive delays at the input of rightmost XOR gate. If a J-unfolded LFSR is desired, $t_0 - t_1 >= J$ should be satisfied so that there will be at least one delay at the input of each of the J copies of the rightmost XOR gate, retiming can be applied to move the delay to the output. An algorithm [8] was proposed to modify the generator polynomial to enable the retiming of the rightmost XOR gate in the J-unfolded architecture. The algorithm finds a polynomial $p(x)$ such that the new polynomial $g(x)p(x)$ will contain the required number of delays at the rightmost XOR gate. The data flow model for the modified LFSR architecture is shown in Fig. 2.2.

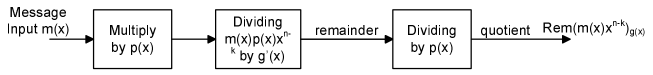


Figure 2.2: Three step implementation of modified BCH encoding

Fig. 2.2 shows a three step implementation. In the first step the message is multiplied by $p(x)$. In the second step, $Rem(m(x)p(x)x^K)_{g'(x)}$ is calculated using a LFSR similar to that in Fig. 2.1. In the third step, $Rem(m(x)p(x)x^K)_{g'(x)}$ needs to be divided by $p(x)$ to get the final result. At this stage, the output is the quotient instead of the remainder from the second stage LFSR.

The fanout problem is now transferred to the third stage. Also, if we directly unfold the LFSR, the iteration bound, i.e., the minimum achievable critical path increases. In addition, the hardware cost also increases due to the additional stages. The later problem is addressed in [9]. Look-ahead pipelining technique is used to reduce the iteration bound in the LFSR, so that the iteration bound of the unfolded architecture is reduced. But the design is applicable only to the generator polynomials that have many zero coefficients between second and the third highest order coefficients. In [10] a two step approach is proposed to reduce the effect of fanout on the third stage which is described below.

- Iteratively multiply $g(x)$ by short length polynomials such as $1 + x^k$ to insert the required number of delay elements in the rightmost feedback loop to eliminate the fanout bottleneck of the LFSR architecture to get $g'(x)$.
- Multiply $g'(x)$ iteratively by $1 + x^k$ to get $g''(x)$, which will lead to the smallest iteration bound for the current $g''(x)$. The iteration exits when the desired iteration

bound or the best iteration bound for certain hardware requirement is reached.

The disadvantage of this method is that the hardware requirement grows considerably as the unfolding factor increases. In general, the complexity of the feedback loop increases in the parallel architecture which leads to an increase in critical path.

In general, parallel architectures for LFSR proposed in the literature lead to an increase in the critical path. The longer the critical path, the speed of operation of the circuit decreases; thus the throughput rate achieved by parallel processing will be reduced. Since these parallel architectures consist of feedback loops, pipelining technique cannot be applied to reduce the critical path. Another issue with the parallel architectures is the hardware cost. A novel parallel CRC architecture based on state space representation is proposed in [13]. The main advantage of this architecture is that the complexity is shifted out of the feedback loop. The full speedup can be achieved by pipelining the feedforward paths. A state space transformation has been proposed in [13] to reduce complexity but the existence of such a transformation was not proved in [13] and whether such a transformation is unique has been unknown so far.

In this thesis, we present a mathematical proof to show that such a transformation exists for all CRC and BCH generator polynomials. We also show that this transformation is non-unique. In fact, we show the existence of infinite such transformations and how these can be derived. We then propose novel schemes based on pipelining, retiming and look ahead computations to reduce the critical path in the parallel architectures based on parallel and pipelined IIR filter design [15]. The proposed IIR filter based parallel architectures have both feedback and feedforward paths, and pipelining can be applied to further reduce the critical path. We show that the proposed architecture can achieve a critical path similar to previous designs with less hardware overhead. Without loss of generality, only binary codes are considered.

2.3 State Space Representation of LFSR

A parallel LFSR architecture based on state space computation has been proposed in [13]. The LFSR shown in Fig. 2.1 can be described by the equation

$$\mathbf{x}(n+1) = \mathbf{A}\mathbf{x}(n) + \mathbf{b}u(n); \quad n >= 0 \tag{2.1}$$

with the initial state $\mathbf{x}(0) = \mathbf{x}_o$. The K-dimensional state vector $\mathbf{x}(n)$ is given by

$$\mathbf{x}(n) = [x_0(n) \quad x_1(n)...x_{K-1}(n)]^T$$

and $\mathbf{A}$ is the $K \times K$ matrix given by

$$\mathbf{A} = \begin{bmatrix} 0 & 0 & 0 & ... & 0 & g_0 \\ 1 & 0 & 0 & ... & 0 & g_1 \\ 0 & 1 & 0 & ... & 0 & g_2 \\ ... & ... & ... & ... & ... & ... \\ 0 & 0 & 0 & ... & 1 & g_{K-1} \end{bmatrix} \tag{2.2}$$

The $K \times 1$ matrix $\mathbf{b}$ is

$$\mathbf{b} = [g_0 \quad g_1...g_{K-1}]^T.$$

The output of the system is the remainder of the polynomial division that it computes, which is the state vector itself. We call the output vector $\mathbf{y}(n)$ and add the output equation $\mathbf{y}(n) = \mathbf{C}\mathbf{x}(n)$ to the state equation in (2.1), with $\mathbf{C}$ equal to the $K \times K$ identity matrix. The coefficients of the generator polynomial $g(x)$ appear in the right-hand column of the matrix $\mathbf{A}$. Note that, this is the *companion matrix* of polynomial $g(x)$ and $g(x)$ is the *characteristic polynomial* of this matrix. The initial state $\mathbf{x}_o$ depends on the specific definition of the CRC for a given application.

The L-parallel system can be derived so that L elements of the input sequence $u(n)$ are processed in parallel. Let the elements of $u(n)$ be grouped together, so that the input to the parallel system is a vector $\mathbf{u}_L(mL)$ where

$$\begin{aligned} \mathbf{u}_L(mL) \quad = \quad & [u(mL+L-1) \quad u(mL+L-2) \quad ... \\ & u(mL+1) \quad u(mL)]; \\ & m = 0, 1, ..., (N/L) - 1 \end{aligned} \tag{2.3}$$

where N is the length of the input sequence. Assume that N is an integral multiple of L. The state space equation can be written as

$$\mathbf{x}(mL+L) = \mathbf{A}^L\mathbf{x}(mL) + \mathbf{B}_L\mathbf{u}_L(mL) \tag{2.4}$$

where the index mL is incremented by L for each block of L input bits.

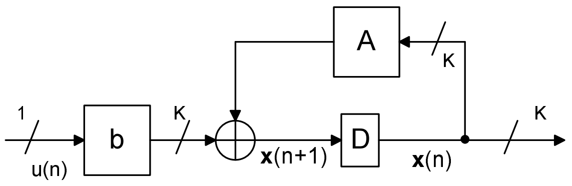


Figure 2.3: Serial LFSR Architecture

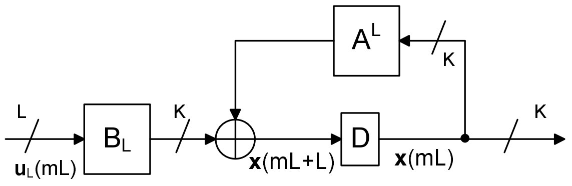


Figure 2.4: M-parallel LFSR Architecture

The matrix $\mathbf{B}_L$ is a $K \times L$ matrix given by

$$\mathbf{B}_L = [\mathbf{b} \quad \mathbf{Ab} \quad \mathbf{A}^2\mathbf{b}...\mathbf{A}^{L-1}\mathbf{b}]$$

The output vector remains the same which is equal to the state vector at $m = N/L$, where N is the message length. Fig. 2.4 shows an L-parallel system which processes L-bits at a time. The issue in this system is that the delay in the feedback loop increases due to the complexity of $\mathbf{A}^L$.

We can compare the throughput of the original system and the modified L-parallel system described by (2.4) using the block diagrams shown in Fig. 2.3 and Fig. 2.4. The throughput of both systems is limited by the delay in feedback loop. The main difference in the two systems is computing the product of $\mathbf{Ax}$ and $\mathbf{A}^L\mathbf{x}$. We can observe that from (2.2), no row of $\mathbf{A}$ has more than two non-zero elements. Thus the time required for the computation of the product $\mathbf{Ax}$ in $GF(2)$ is the delay of a two input XOR gate. Similarly, the time required for the product $\mathbf{A}^L\mathbf{x}$ will depend on the number of non-zero elements in the matrix $\mathbf{A}^L$.

Consider the standard 16-bit CRC with the generator polynomial

$$g(x) = x^{16} + x^{15} + x^{14} + x^2 + x + 1$$

with $L = 16$. The maximum number of non-zero elements in any row of $\mathbf{A}^{16}$ is 5, i.e., the time required to compute the product is the time to compute exclusive or of 5 terms. The detailed analysis for this example is presented in the Appendix. In a similar manner, the maximum number of non-zero entries in $\mathbf{A}^{32}$ for the standard 32-bit CRC is 17. This computation is the bottleneck for achieving required speed-up factor.

We can observe from Fig. 2.4 that a speed-up factor of L is possible if the block $\mathbf{A}^L$ can be replaced by a block whose circuit complexity is no greater than that of block $\mathbf{A}$. This is possible given the restrictions on the generator polynomial $g(x)$ that are always met for the CRC and BCH codes in general. The computation in the feedback loop can be simplified by applying a linear transformation to (2.4). The linear transformation moves the complexity from the feedback loop to the blocks that are outside the loop. These blocks can be pipelined and their complexity will not affect the system throughput.

2.4 State Space Transformation

2.4.1 Transformation

A linear transformation has been proposed [13] to reduce the complexity in the feedback loop. The state space equation of L-parallel system with an explicit output equation is described as

$$\mathbf{x}(mL+L) = \mathbf{A}^L\mathbf{x}(mL) + \mathbf{B}_L\mathbf{u}_L(mL); \mathbf{y}(mL) = \mathbf{C}_L\mathbf{x}(mL) \tag{2.5}$$

where $\mathbf{C}_L = I$, the $K \times K$ identity matrix. The output vector $\mathbf{y}(mL)$ is equal to the state vector which has the remainder at $m = N/L$. Consider the linear transformation of the state vector $\mathbf{x}(mL)$ through a constant non-singular matrix $\mathbf{T}$, i.e.,

$$\mathbf{x}(mL) = \mathbf{T}\mathbf{x}_t(mL)$$

Given $\mathbf{T}$ and its inverse, we can express the state space equation (2.5) in terms of the state vector $\mathbf{x}_t(mL)$, as follows:

$$\mathbf{x}_t(mL+L) = \mathbf{A}_{Lt}\mathbf{x}_t(mL) + \mathbf{B}_{Lt}\mathbf{u}_L(mL); \mathbf{y}(mL) = \mathbf{C}_{Lt}\mathbf{x}_t(mL)$$

where

$$\mathbf{A}_{Lt} = \mathbf{T}^{-1}\mathbf{A}^L\mathbf{T}; \mathbf{B}_{Lt} = \mathbf{T}^{-1}\mathbf{B}_L; \mathbf{C}_{Lt} = \mathbf{T} \tag{2.6}$$

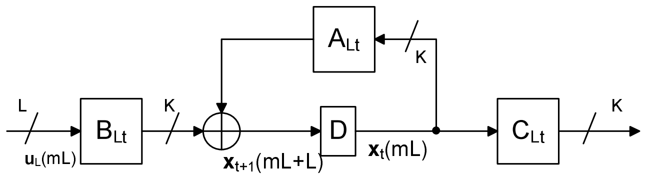


Figure 2.5: Modified LFSR Architecture using state space transformation

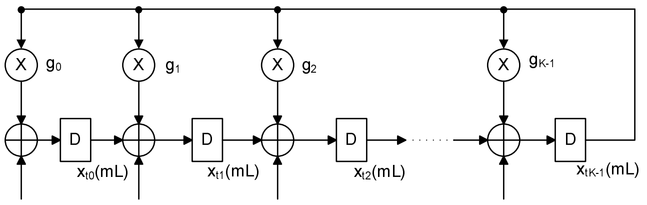


Figure 2.6: Modified feedback loop of Fig. 2.5

and $\mathbf{T}$ is the transformation matrix. The parallel LFSR architecture after the transformation is shown in Fig. 2.5 and the modified feedback loop in Fig. 2.6. We can observe from the figure that if $\mathbf{A}_{Lt}$ is a companion matrix, then the complexity of the feedback loop will be same as that of the original LFSR. If there exists a $\mathbf{T}$ such that $\mathbf{A}_{Lt}$ is a companion matrix, then the complexity in the feedback loop comes down. It is evident that (2.6) represents a similarity transformation and we can state that there exists a $\mathbf{T}$ such that $\mathbf{A}_{Lt}$ is a companion matrix if and only if $\mathbf{A}^L$ is similar to companion matrix. The following theorem proves that $\mathbf{A}^L$ is similar to a companion matrix provided the generator polynomial is irreducible. The latter condition is met for all CRC and BCH codes.

2.4.2 Existence of Transformation matrix

Theorem: Given an irreducible generator polynomial $g(x)$ with its companion matrix A, A^L is similar to a companion matrix.

Proof: A companion matrix is similar to a diagonal matrix if its characteristic polynomial has distinct roots. From the theory of finite fields [17], if f is an irreducible polynomial in $GF(q)$ of degree k, then f has root α in its splitting field $GF(q^k)$. The proof of this statement is presented in the next theorem. Furthermore, all the roots of f are simple and are given by the distinct elements $\alpha, \alpha^q, \alpha^{q^2}, ..., \alpha^{q^{k-1}}$ of $GF(q^k)$.

Since, the generator polynomial is irreducible, $g(x)$ has distinct roots in its splitting field. Therefore, we can diagonalize A as follows:

$$A = V^{-1}DV \tag{2.7}$$

where D is diagonal matrix with distinct roots of $g(x)$ as its elements. Now

$$\begin{aligned} A^L &= (V^{-1}DV)^L \qquad (2.8)\\ &= (V^{-1}DV)(V^{-1}DV)...(V^{-1}DV) \\ &= V^{-1}D^LV \\ \Rightarrow VA^LV^{-1} &= D^L \end{aligned}$$

Let $\lambda_1, \lambda_2, ..., \lambda_k$ be the distinct roots of the polynomial $g(x)$. Then,

$$D^L = \begin{bmatrix} \lambda_1^L & 0 & ... & 0 \\ 0 & \lambda_2^L & ... & 0 \\ ... & ... & ... & ... \\ 0 & 0 & ... & \lambda_k^L \end{bmatrix} \tag{2.9}$$

Since $\lambda_1, \lambda_2, ..., \lambda_k$ are distinct roots in the splitting field $GF(q^k)$, $\lambda_1^L, \lambda_2^L, ..., \lambda_k^L$ will be distinct in the same field. Therefore, there exists a companion matrix C such that

$$\begin{aligned} &C = BD^LB^{-1} \qquad (2.10)\\ or \qquad &D^L = B^{-1}CB \end{aligned}$$

$$\begin{aligned} \therefore \qquad VA^LV^{-1} &= B^{-1}CB \qquad (2.11)\\ (BV)A^L(V^{-1}B^{-1}) &= C \end{aligned}$$

This shows that A^L is similar to a companion matrix.■

The following theorem is presented below from [17].

Theorem: If f is an irreducible polynomial in $GF(q)$ of degree k, then f has a root α in $GF(q^k)$. Furthermore, all the roots of f are simple and are given by the k distinct elements $\alpha, \alpha^q, \alpha^{q^2}, ..., \alpha^{q^{k-1}}$ of $GF(q^k)$.

Proof: Let α be a root of f in the splitting field of f over $GF(q)$. Next we show that if $\beta \in GF(q^k)$ is a root of f, then β^q is also a root of f. Write $f(x) = a_kx^k + ... + a_1x + a_0$

with $a_i \in GF(q)$ for $0 \leq i \leq k$.Then,

$$\begin{aligned} f(\beta^q) &= a_k\beta^{qk} + ... + a_1\beta^q + a_0 \\ &= a_k^q\beta^{qk} + ... + a_1^q\beta^q + a_0^q \\ &= (a_k\beta^{qk} + ... + a_1\beta^q + a_0)^q \\ &= [f(\beta)]^q = 0 \end{aligned} \tag{2.12}$$

Therefore, the elements $\alpha, \alpha^q, \alpha^{q^2}, ..., \alpha^{q^{k-1}}$ are roots of f. It remains to prove that these elements are distinct. Suppose, on the contrary, that $\alpha^{q^i} = \alpha^{q^j}$ for some integers i and j with $0 \leq i \leq j \leq k-1$. By raising this identity to the power q^{k-j}, we get

$$\alpha^{q^{k-j+i}=\alpha^{q^k}} = \alpha.$$

It follows that $f(x)$ divides $x^{q^{k-j+i}} - x$. This is only possible if k divides $k-j+i$. But we have $0 < k-j+i < k$, which contradicts the assumption.■

2.4.3 Construction of matrix T

Given that $\mathbf{A}^L$ is similar to a companion matrix, it remains to construct $\mathbf{A}_{Lt}$ as the appropriate companion matrix and then to find a matrix $\mathbf{T}$ that represents the similarity transformation in (2.6). The matrix $\mathbf{T}$ is chosen such that $\mathbf{A}_{Lt}$ has the same form as $\mathbf{A}$. In general, the matrix $\mathbf{T}$ is not unique.

Theorem: Transformation matrix $\mathbf{T}$ is not unique.

Proof: Using theorem in Section IV(B), a transformation matrix $\mathbf{T}$ exists for a given companion matrix $\mathbf{A}$. Let us assume $\mathbf{T}$ is unique. From (2.6), we have

$$\mathbf{A}_{Lt} = \mathbf{T}^{-1}\mathbf{A}^L\mathbf{T}$$

Let $\mathbf{T}_1 = \mathbf{A}^i\mathbf{T}$, where i is any integer.

$$\begin{aligned} \mathbf{T}_1^{-1}\mathbf{A}^L\mathbf{T}_1 &= \mathbf{T}^{-1}\mathbf{A}^{-i}\mathbf{A}^L\mathbf{A}^i\mathbf{T} \\ &= \mathbf{T}^{-1}\mathbf{A}^L\mathbf{T} = \mathbf{A}_{Lt} \end{aligned}$$

This shows that $\mathbf{T}_1$ is also a transformation matrix which leads to same similarity transformation which by contradiction prove that $\mathbf{T}$ is not unique.■

As a counter example, we obtain two different $\mathbf{T}$ matrices for a given generator polynomial. An example for the standard 16-bit CRC polynomial is presented in the next section.

The approach presented here is based on cyclic vector of the transformation represented by the matrix $\mathbf{A}^L$. Select an arbitrary vector $\mathbf{b}_1$, subject only to the condition that the vectors $\mathbf{A}^{kL}\mathbf{b}_1$, for $k = 0, 1, .., K-1$, are linearly independent. Now, let $\mathbf{T}$ be the matrix whose columns are these vectors, i.e.,

$$\mathbf{T} = [\mathbf{b}_1 \quad \mathbf{A}^L\mathbf{b}_1 \quad \mathbf{A}^{2L}\mathbf{b}_1 ... \mathbf{A}^{(K-1)L}\mathbf{b}_1] \tag{2.13}$$

Since all the columns are linearly independent, $\mathbf{T}$ is non-singular. We can show that matrix $\mathbf{T}$ implements the desired similarity relationship described in (2.6).

$$\mathbf{T}^{-1}\mathbf{A}^L\mathbf{T} = \mathbf{T}^{-1}[\mathbf{A}^L\mathbf{b}_1 \quad \mathbf{A}^{2L}\mathbf{b}_1 ... \mathbf{A}^{KL}\mathbf{b}_1]$$

We can observe that the first $K-1$ columns in the matrix on the right hand side are same as the last $K-1$ columns in the $\mathbf{T}$ matrix. Therefore, $\mathbf{A}_{Lt}$ is similar to a companion matrix since first $K-1$ columns will form the identity matrix and the last column depends on the chosen vector $\mathbf{b}_1$. Then by theorem from page. 230 in [18], we can show that for such a matrix $\mathbf{A}^L$, a cyclic vector exists.

Then the complexity of feedback loop will be identical to that of the original system. The feedback loop after the transformation looks as shown in Fig. 2.6. The complexity has been moved out of the feedback loop. Also the matrix multiplication has to be pipelined to reduce the critical path which leads to an increase in the hardware cost. We propose a new architecture based on parallel IIR filter design to achieve high speed with less hardware cost.

2.5 Example for state space Transformation

Consider the standard 16-bit CRC with the generator polynomial

$$g_{16}(x) = x^{16} + x^{15} + x^{14} + x^2 + x + 1$$

with $L = 16$, i.e., processing 16-bits of input sequence at a time. Using (2.2), we can compute matrices A and A^L are as follows:

$$A = \begin{bmatrix}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1
\end{bmatrix}$$

$$A^L = \begin{bmatrix}
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 \\
0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \\
1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1
\end{bmatrix}$$

We can observe that the maximum number of non-zero elements in any row of $\mathbf{A}^L$ is 5, i.e., the time required to compute the product is the time to compute 5 exclusive or operations. By using the proposed transformation, we can transform the matrix $\mathbf{A}^L$ into a companion matrix $\mathbf{A}_{Lt}$. We can compute $\mathbf{T}$, $\mathbf{T}^{-1}$ and $\mathbf{A}_{Lt}$ from (2.13) and (2.6), which are shown below. In this case we use $\mathbf{b}_1 = [1 \quad 0...0]$.

$$T = \begin{bmatrix}
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 1 \\
0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 1 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 \\
0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\
0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\
0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 0
\end{bmatrix}$$

$$T^{-1} = \begin{bmatrix}
1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 \\
0 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\
0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 \\
0 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\
0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\
0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1
\end{bmatrix}$$

$$A_{Lt} = \begin{bmatrix}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1
\end{bmatrix}$$

We provide another matrix $\mathbf{T}_1 = \mathbf{AT}$ below to show that this transformation is not unique. We get a different $\mathbf{T}$, by just changing the cyclic vector in the $\mathbf{T}$ matrix computation. We can observe that the two $\mathbf{T}$ matrices are different and lead to the same similarity transformation. This shows that multiple solutions exist for transformation matrix **T** .

$$T_1 = \begin{bmatrix}
0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 0 \\
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 1 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 \\
0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 1 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 \\
0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\
0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 \\
0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0
\end{bmatrix}$$

$$T_1^{-1} = \begin{bmatrix}
1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\
1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 \\
0 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\
1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\
0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\
0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\
0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1
\end{bmatrix}$$

It may indeed be noted that any $\mathbf{T}_i = \mathbf{A}^i\mathbf{T}$ is a valid similarity transformation.

Chapter 3

Proposed LFSR Architectures

In this chapter, a new formulation of designing the LFSR architectures based on IIR filter is described. Parallel architectures are developed using pipelining, retiming and look-ahead techniques [21]. Further, combined parallel processing and pipelining techniques are presented to reduce the critical path and eliminate the fanout effect for long BCH codes [22].

3.1 IIR Filter Representation of LFSR

In this section, we propose a new formulation for LFSR architecture. Let's assume that the input to the LFSR is $u(n)$, and the required output, i.e., the remainder, is $y(n)$ as shown in Fig. 3.1. Then the LFSR can be described using the following equations;

$$\begin{aligned} w(n) &= y(n) + u(n) && (3.1) \\ y(n) &= g_{K-1} * w(n-1) + g_{K-2} * w(n-2) + ... \\ &\quad + g_0 * w(n-K) && (3.2) \end{aligned}$$

Substituting (1) into (2), we get

$$\begin{aligned} y(n) &= g_{K-1} * y(n-1) + g_{K-2} * y(n-2) + ... \\ &\quad + g_0 * y(n-K) + f(n) && (3.3) \end{aligned}$$

where,

$$f(n) = g_{K-1} * u(n-1) + g_{K-2} * u(n-2) + ... + g_0 * u(n-K).$$

In the equations above, $'+'$ denotes XOR operation. We can observe that equation resembles an IIR filter with $g_0, g_1, ..., g_{K-1}$ as coefficients.

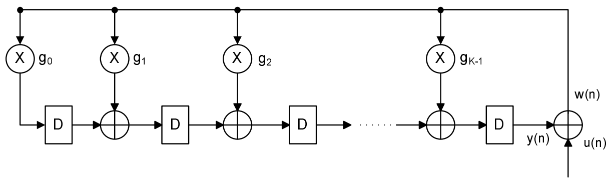


Figure 3.1: General LFSR architecture.

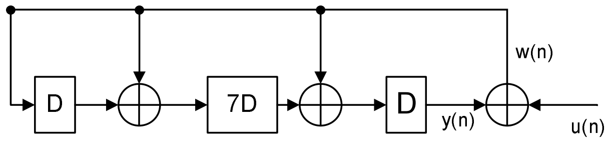


Figure 3.2: LFSR architecture for $g(x) = 1 + x + x^8 + x^9$.

Consider a generator polynomial $g(y) = 1 + x + x^8 + x^9$, the CRC architecture is shown in Fig. 3.2. By using the above formulation,

$$y(n) = y(n-9) + y(n-8) + y(n-1) + f(n) \tag{3.4}$$

where,

$$f(n) = u(n-9) + u(n-8) + u(n-1).$$

The following example illustrates the correctness of the proposed method. The CRC architecture for the above equation is shown in Fig. 3.3. Let the message sequence be 101011010; Table 1 shows the data flow at the marked points of this architecture at different time slots.

In Table 1, we can see that this architecture requires 18 clock cycles to compute the output. Since this is a serial computation, we need to input 0's for the next 9 clock cycles after the message has been processed. In addition, the feedback has to be removed after the message bits are processed for the correct functioning of the circuit. Since this operation is in $GF(2)$, guaranteeing the feedback to be 0 after 9 clock cycles will have the same effect.

Table 3.1: Data flow of Fig. 3.3 when the input message is 101011010

clock	$u(n)$	$f(n)$	$y(n)$
1	1	1	1
2	0	0	1
3	1	1	0
4	0	0	0
5	1	1	1
6	1	1	0
7	0	0	0
8	1	0	0
9	0	1	0
10	0	1	1
11	0	1	0
12	0	1	1
13	0	0	1
14	0	1	0
15	0	1	1
16	0	1	1
17	0	0	0

3.2 Parallel LFSR Architecture based on IIR Filtering

We can derive parallel architectures for IIR filters using look ahead techniques as described in [23]. We use the same look-ahead technique to derive parallel system for a given LFSR. Parallel architecture for a simple LFSR described in the previous section is discussed first.

Consider the design of a 3-parallel architecture for the LFSR in Fig. 3.2. In the parallel system, each delay element is referred to as a block delay where the clock period of the block system is 3 times the sample period. Therefore, instead of (3), the loop update equation should update $y(n)$ using inputs and $y(n-3)$. The loop update process for the 3-parallel system is shown in Fig. 3.4, where $y(3k+3)$, $y(3k+4)$ and $y(3k+5)$ are computed using $y(3k)$, $y(3k+1)$ and $y(3k+2)$. By iterating the recursion or by

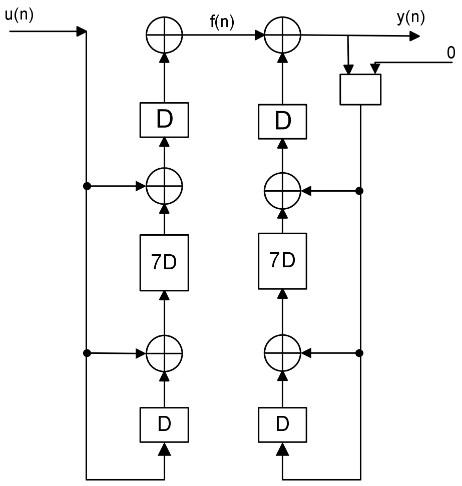


Figure 3.3: LFSR architecture for $G(x) = 1+x+x^8+x^9$ after the proposed formulation.

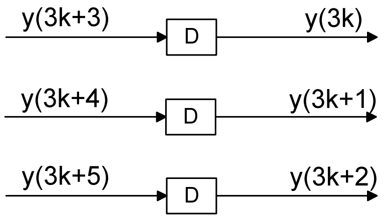


Figure 3.4: Loop update equations for block size $L = 3$.

applying look-ahead technique, we get

$$\begin{aligned} y(n) &= y(n-1) + y(n-8) + y(n-9) + f(n) && (3.5) \\ &= y(n-2) + y(n-8) + y(n-10) + f(n-1) \\ &\quad +f(n) && (3.6) \\ &= y(n-3) + y(n-8) + y(n-11) + f(n-2) \\ &\quad +f(n-1) + f(n). && (3.7) \end{aligned}$$

Substituting $n = 3k+3, 3k+4, 3k+5$ in the above equations, we have the following 3 loop update equations:

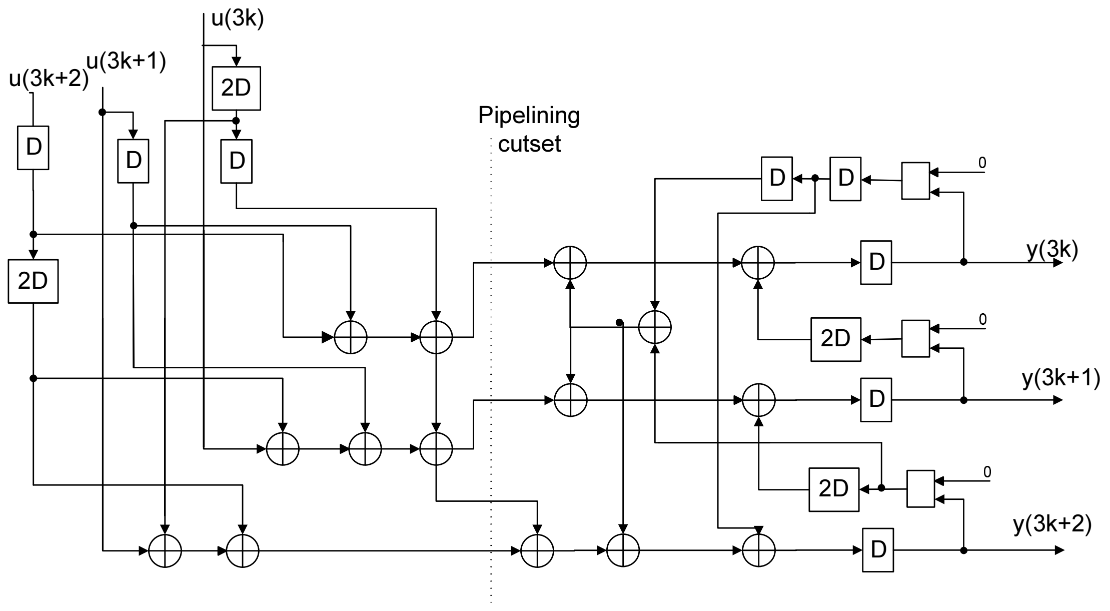


Figure 3.5: Three parallel LFSR architecture after pipelining for Fig. 3.2.

$$\begin{aligned} y(3k+3) &= y(3k+2)+y(3k-5)+y(3k-6) \\ &\quad +f(3k+3) \qquad (3.8) \\ y(3k+4) &= y(3k+2)+y(3k-4)+y(3k-6) \\ &\quad +f(3k+3)+f(3k+4) \qquad (3.9) \\ y(3k+5) &= y(3k+2)+y(3k-3)+y(3k-6) \\ &\quad +f(3k+3)+f(3k+4)+f(3k+5) \qquad (3.10) \end{aligned}$$

where

$$\begin{aligned} f(3k+3) &= u(3k+2)+u(3k-5)+u(3k-6) \\ f(3k+4) &= u(3k+3)+u(3k-4)+u(3k-5) \\ f(3k+5) &= u(3k+4)+u(3k-3)+u(3k-4). \end{aligned}$$

The 3-parallel system is shown in Fig. 3.5. The input message 101011010 in the previous example leads to the data flow as shown in Table 3.2. We can see that 6 clock cycles are needed to encode a 9-bit message. The proposed architecture requires

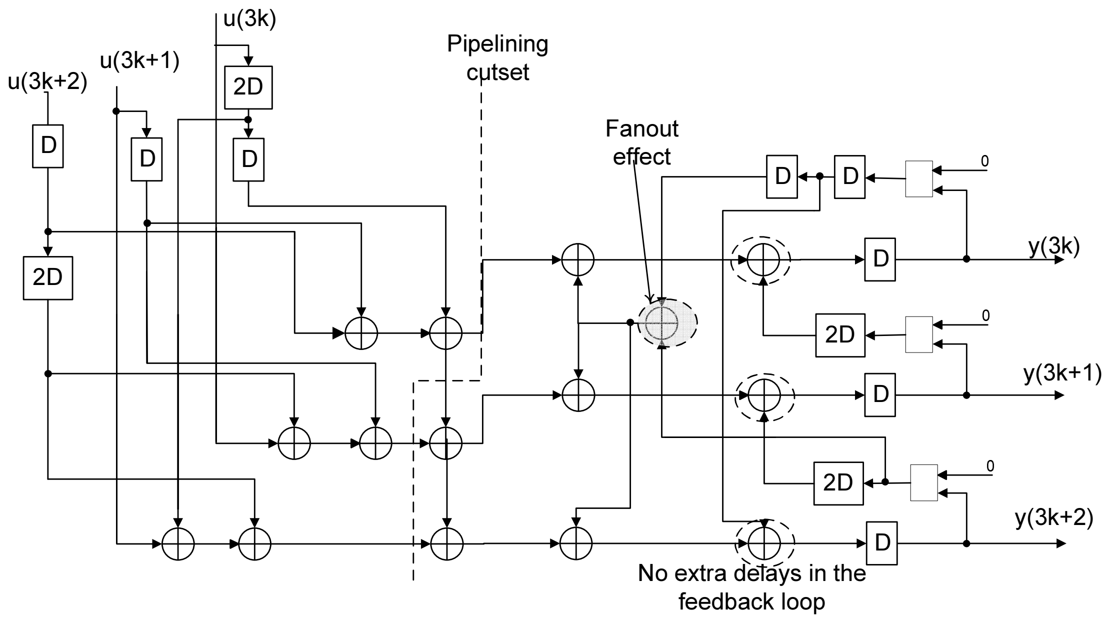


Figure 3.6: Retiming and Pipelining cutsets in three parallel LFSR architecture to reduce C.P to $2T_{xor}$ for Fig. 3.2.

$(N + K)/L$ cycles where N is the length of the message, K is the length of the code and L is the parallellism level. Similar to the serial architecture, the feedback path has to be broken after the message bits have been processed. The critical path (CP) of the design is $6T_{xor}$, where T_{xor} is the XOR gate delay. We can further reduce this delay by pipelining which is explained in the next section.

Table 3.2: Data flow of Fig. when the input message is 101011010

Clock#	$u(3k)$	$u(3k+1)$	$u(3k+2)$	$y(3k)$	$y(3k+1)$	$y(3k+2)$
1	1	0	1	0	1	1
2	0	1	1	0	0	1
3	0	1	0	0	0	0
4	0	0	0	0	1	0
5	0	0	0	1	1	0
6	0	0	0	1	1	0

3.3 Pipelining and retiming for reducing Critical Path

The critical path of the parallel system can be reduced by further pipelining and retiming the architecture. In this section, we show how to pipeline and retime for reducing the critical path to obtain fast parallel architecture by using the example in Fig. 3.2. Pipelining is possible only when there exist feed-forward cutsets [15]. When we unfold an LFSR directly, pipelining is not possible since there are no feed-forward cutsets. In the proposed method, the calculation of $f(n)$ terms is feed-forward. So, we can pipeline at this stage to reduce the critical path.

By designing the 3-parallel system, we obtain the design in Fig. 3.5. It is obvious that critical path of the system is $6T_{xor}$, where T_{xor} is the XOR gate delay. We can reduce it to $3T_{xor}$ by applying pipelining cutset as shown in Fig. 3.5.

Further if the parallelism level (L) is high, the number of $f(n)$ terms to be added will increase leading to a large critical path. We apply tree structure and sub-expression sharing to reduce the critical path while keeping the hardware overhead to a minimum [15]. We observe many common terms in the filter equations if the parallelism level is high. We can reduce the hardware by using substructure sharing [15]. In particular, we used multiple constant multiplication (MCM) [24] method to find the common computations across the filter equations. The algorithm uses an iterative matching process to find the common subexpressions.

Retiming can be applied to further reduce the critical path to $2T_{xor}$. By using the pipelining cutsets as shown in Fig. 3.6 and the retiming around the circled nodes, we can reduce the critical path to $2T_{xor}$. We can observe that there will be some hardware overhead to achieve this critical path. But one path around the bottom circled node has still 3 XOR gates. This is because, one of the feedback paths does not have preexisting delays. This can be solved by using combined parallel and pipelining technique of IIR filter design.

3.3.1 Combined parallel processing and pipelining

We can see from Fig. 3.6 that, when we have additional delays at the feedback loops, we can retime around the commonly shared nodes (as indicated in Fig. 3.6) to remove the large fanout effect. We can further reduce the critical path by combining parallel processing and pipelining concepts for IIR filter design [15]. Instead of using one-step look ahead, we use two-step look ahead computation to generate the filter equations. In this example, we need to compute $y(3k+8), y(3k+7), y(3k+6)$ instead of $y(3k+5), y(3k+4), y(3k+3)$. By doing this, we get two delays in the feedback loop, we can then retime to reduce the critical path and also remove the fanout effect on the shared node.

Now, the loop update equations are

$$\begin{aligned} y(3k+3) &= y(3k+2) + y(3k-2) + y(3k-6) \\ &\quad + f(3k+3) + ... + f(3k+6) \qquad (3.11) \\ y(3k+4) &= y(3k+2) + y(3k-1) + y(3k-6) \\ &\quad + f(3k+3) + ... + f(3k+7) \qquad (3.12) \\ y(3k+5) &= y(3k+2) + y(3k) + y(3k-6) \\ &\quad + f(3k+3) + ... + f(3k+8) \qquad (3.13) \end{aligned}$$

The feedback part of the architecture is shown in Fig. 3.7. We can observe from this

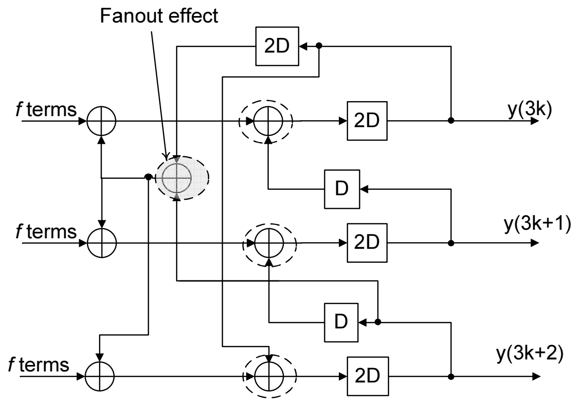


Figure 3.7: Loop update for combined parallel pipelined LFSR

figure that retiming can be applied at all the circled nodes to reduce the critical path in the feedback section. Further, fanout bottleneck can be alleviated using retiming at that particular node. The final architecture after retiming is shown in Fig. 3.8.

The proposed architecture is compared against the prior designs in Chapter 6. A detailed analysis shows that the proposed architecture can reduce the critical path with less hardware complexity compared to previous designs for all the regularly used CRC

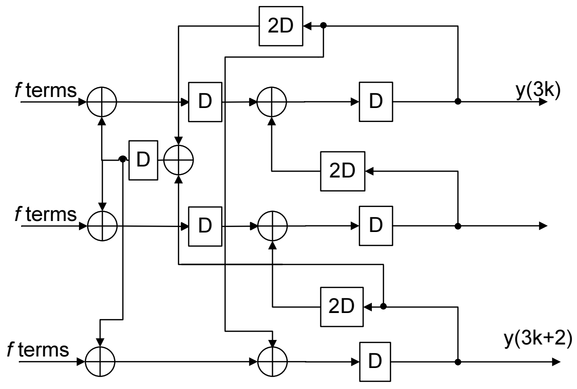


Figure 3.8: Loop update for combined parallel pipelined LFSR

and BCH polynomials. Comparison is made for BCH(8191,7684) code to show that the proposed design works even for long generating polynomials.

3.4 Comparison and Analysis

We limit our discussion to the commonly used generator polynomials for CRC and BCH encoders that are shown in Table 3.3. A comparison between the previous high-speed architectures and the proposed ones is shown in Table 3.4 for different parallelism levels of different generator polynomials. The comparison is made in terms of required number of xor gates (# XOR), required number of delay elements (#D.E.) and critical path (C.P.) of the architectures for different parallelism levels (L).

Table 3.3: Commonly used generator polynomials

CRC-12	$y^{12}+y^{11}+y^3+y^2+y+1$
CRC-16	$y^{16}+y^{15}+y^2+y+1$
SDLC	$y^{16}+y^{12}+y^5+y+1$
CRC-16 REVERSE	$y^{16}+y^{14}+y+1$
SDLC	$y^{16}+y^{11}+y^4+1$
CRC-32	$y^{32}+y^{26}+y^{23}+y^{22}+y^{16}+y^{12}+y^{11}+y^{10}+y^8+y^7+y^5+y^4+y^2+y+1$

The numbers for [13] are calculated after the design is pipelined to have a critical path of $4T_{xor}$. We can further pipeline the architecture to reduce the critical path to

$2T_{xor}$ but it will increase the required number of delay elements. Note that the latency of this design increases by 2. The critical path for design in [10] shown in Table 3.4 is the minimum achievable critical path, i.e., the iteration bound. Tree structure is applied to further reduce the CP to $O(log(CP))$ at the cost of some additional XOR gates.

Table 3.4: Comparison between previous LFSR architectures and the proposed one

Poly (L)	Algorithm	# XOR	# D.E.	C.P.
CRC-12 (12)	[13]	121	35	5
	[10]	276	47	3.15
	Proposed	109	36	5
CRC-16 (16)	[13]	203	48	5
	[10]	400	76	4
	Proposed	113	50	5
SDLC (16)	[13]	223	48	5
	[10]	400	54	5.44
	Proposed	139	50	5
CRC-16 Reverse (16)	[13]	235	46	5
	[10]	592	68	5.97
	Proposed	126	50	5
SDLC Reverse (16)	[13]	688	48	5
	[10]	233	76	7.4
	Proposed	127	50	5
CRC-32 (32)	[13]	1000	96	6
	[10]	6496	344	16.13
	Proposed	794	96	5
BCH (255,223) (32)	[13]	934	96	6
	[10]	4832	276	14
	Proposed	863	96	5

In the table, we can observe that for all the listed commonly used generator polynomials, the proposed design achieves a critical path that is same as the previous designs without increasing the hardware overhead. Furthermore, the number of clock cycles to process N-bits is almost same as the previous designs. Our design can achieve the same throughput rate at the reduced hardware cost.

Comparison of critical path (C.P) and XOR gates of the proposed design with previous parallel long BCH(8191,7684) encoders in [10] and [11] for L-parallel architecture is shown in Table 3.5. From the table, we can observe that the proposed design performs better in terms of hardware efficiency. The proposed design also performs better in terms of critical path for higher levels of parallelism. The values reported in [10] and [11] are iteration bounds i.e., the minimum achievable critical path. Even for the cases $L = 8$ and $L = 16$, we can further pipeline and retime to reduce the critical path as proposed in Section VII.

Table 3.5: Comparison of C.P and XOR gates of the proposed design and previous parallel long BCH(8191,7684) encoders for L-parallel architecture

		$L = 8$	$L = 16$	$L = 24$	$L = 32$
C.P (T_{xor})	Proposed	9	9	9	9
	[10]*	3.5	7.03	10.2	13.63
	[11]*	4.167	7.769	11.111	14.034
XOR gates	Proposed	2012	4069	6125	8229
	[10]	2360	4032	8520	10592
	[11]	2845	5469	9532	12512

*Reported values are minimum achievable C.P

We can see from Fig. 3.8 that critical path in the feedback loop can be reduced. This is possible because of the extra delay in the feedback path. This shows that our proposed approach of combined parallel and pipelining technique can reduce the critical path as a whole. The elimination of fanout effect on the XOR gate (shared subexpression) is another advantage of this approach. This comes at the cost of extra hardware.

Chapter 4

Fast Fourier Transform

In this chapter, a short introduction to fast fourier transform (FFT) and its hardware architectures is provided. Further, this chapter proposes a new approach for the design of FFT hardware architectures. The approach is based on folding and register minimization techniques.

4.1 FFT Algorithms

In this section, radix-2, radix-2^2 and radix-2^3 algorithms are reviewed and comparison between these algorithms is made in terms of multiplications required. The N-point Discrete Fourier Transform (DFT) of a sequence $x[n]$ is defined as

$$X[k] = \sum_{n=0}^{N-1} x[n] W_N^{nk}, \tag{4.1}$$

where $W_N^{nk} = e^{-j(2\pi/N)nk}$.

The FFT includes a collection of algorithms that reduce the number of operations of the DFT. The Cooley-Tukey algorithm [25] is the most used among them. It is based on decomposing the DFT in $n = log_r N$ stages, where r is the radix. This achieves a reduction on the number of operations from order $O(N^2)$ in the DFT to order $O(NlogN)$ in the FFT. The decomposition can be carried out in different ways. The most common ones are the Decimation in Time (DIT) and the Decimation in Frequency (DIF).

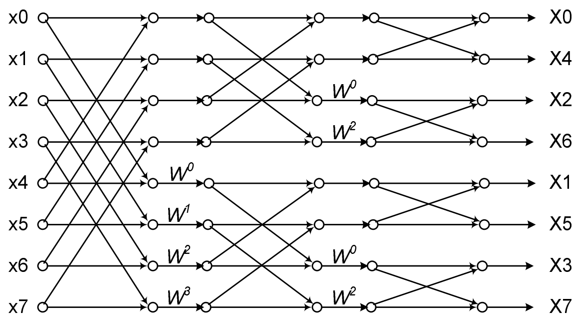


Figure 4.1: Flow graph of a radix-2 8-point DIF FFT.

4.1.1 Radix-2 DIF FFT

For radix-2, the DIF decomposition separates the output sequence $X[k]$ into even and odd samples. The following equations are obtained:

$$\begin{aligned} X[2r] &= \sum_{n=0}^{N/2-1} (x[n] + x[n+N/2])e^{-j\frac{2\pi}{N/2}rn}, r = 0, 1..., N/2-1 \\ X[2r+1] &= \sum_{n=0}^{N/2-1} (x[n] - x[n+N/2])e^{-j2\pi/Nn}e^{-j\frac{2\pi}{N/2}rn}, r = 0, 1..., N/2-1 \quad (4.2) \end{aligned}$$

The N-point DFT is transformed into two $N/2$ point DFTs. Applying the procedure iteratively leads to decomposition into 2-point DFTs. Fig. 4.1 shows the flow graph of 8-point radix-2 DIF FFT.

4.1.2 Radix-2^2 DIF FFT

The radix-2^2 FFT algorithm is proposed in [26]. We can derive the algorithm by using the following new indices,

$$\begin{aligned} n &= <\frac{N}{2}n_1 + \frac{N}{4}n_2 + n_3>_N \\ k &= <k_1 + 2k_2 + 4k_3>_N \end{aligned} \quad (4.3)$$

Substituting (4.3) in (4.1), we get

$$\begin{aligned} &X(k_1 + 2k_2 + 4k_3) \\ &= \sum_{n_3=0}^{\frac{N}{4}-1} \sum_{n_2=0}^{1} \sum_{n_1=0}^{1} x(\frac{N}{2}n_1 + \frac{N}{4}n_2 + n_3)W_N^{nk} \end{aligned} \quad (4.4)$$

By decomposing the twiddle factor, we get

$$W_N^{nk} = (-j)^{n_2(k_1+2k_2)} W_N^{n_3(k_1+2k_2)} W_N^{4n_3k_3} \tag{4.5}$$

Substituting (4.5) in (4.4) and expanding the summation with indices n_1, n_2, and we get,

$$X(k_1 + 2k_2 + 4k_3) = \sum_{n_3=0}^{\frac{N}{4}-1} [H_{\frac{N}{4}}(n_3, k_1, k_2) W_N^{n_3(k_1+2k_2)}] W_{\frac{N}{4}}^{n_3k_3} \tag{4.6}$$

where $H(k_1, k_2, n_3)$ is expressed as,

$$\begin{aligned} H_{\frac{N}{4}}(n_3, k_1, k_2) &= B_{\frac{N}{2}}(n_3, k_1) + (-j)^{(k_1+2k_2)} B_{\frac{N}{2}}(n_3 + \frac{N}{4}, k_1) \\ B_{\frac{N}{2}}(n_3, k_1) &= x(n_3) + (-1)^{k_1} x(n_3 + \frac{N}{2}) \end{aligned} \tag{4.7}$$

Equation (4.7) represents the first two stages of butterflies with only trivial multiplications. After these two stages, full multipliers are required to compute the product of decomposed twiddle factors. The complete radix-2^2 algorithm can be derived by applying (4.6) recursively. Fig. 4.2 shows the flow graph of an $N = 16$ point FFT an decomposed according to decimation in frequency (DIF). The numbers at the inputs and output of the graph represent the index of input and output samples, respectively. The advantage of the algorithm is that it has the same multiplicative complexity as radix-4 algorithms, but still retains the radix-2 butterfly structures. We can observe that, only every other stage of the flow graph has non-trivial multiplications. The $-j$ notion represents the trivial multiplication, which involves only real-imaginary swapping and sign inversion.

4.1.3 Radix-2^3 DIF FFT

The radix-2^3 FFT algorithm is proposed in [27]. Similar to radix-2^2, we can derive the algorithm by using the following new indices,

$$\begin{aligned} n &= < \frac{N}{2}n_1 + \frac{N}{4}n_2 + \frac{N}{8}n_3 + n_4 >_N \\ k &= < k_1 + 2k_2 + 4k_3 + 8k_4 >_N \end{aligned} \tag{4.8}$$

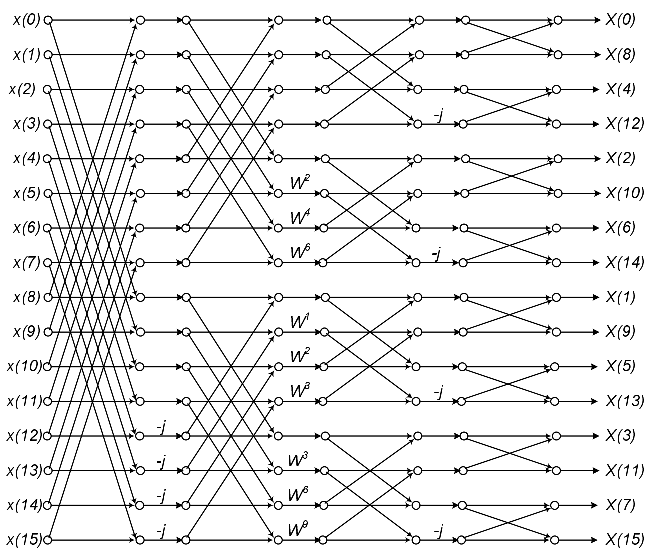


Figure 4.2: Radix-2 Flow graph of a 16-point radix-2^2 DIF FFT.

Substituting (4.8) in (4.1), we get

$$\begin{aligned} & X(k_1 + 2k_2 + 4k_3 + 8k_4) \\ & = \sum_{n_4=0}^{\frac{N}{8}-1} \sum_{n_3=0}^{1} \sum_{n_2=0}^{1} \sum_{n_1=0}^{1} x(\frac{N}{2}n_1 + \frac{N}{4}n_2 + \frac{N}{8}n_3 + n_4) W_N^{nk} \end{aligned} \tag{4.9}$$

The twiddle can be decomposed into the following form

$$\begin{aligned} W_N^{nk} &= (-j)^{n_2(k_1+2k_2)} W_N^{\frac{N}{8}n_3(k_1+2k_2+4k_3)} \\ & .W_N^{n_4(k_1+2k_2+4k_3)} W_N^{8n_4k_4} \end{aligned} \tag{4.10}$$

Substitute (4.10) into (4.9) and expand the summation with regard to index n_1, n_2 and n_3. After simplification we have a set of 8 DFTs of length $N/8$,

$$\begin{aligned} & X(k_1 + 2k_2 + 4k_3 + 8k_4) \\ & = \sum_{n_4=0}^{\frac{N}{8}-1} [T_{\frac{N}{8}}(n_4, k_1, k_2, k_3) W_N^{n_4(k_1+2k_2+4k_3)}] W_{\frac{N}{8}}^{n_4k_4} \end{aligned} \tag{4.11}$$

where a third butterfly structure has the expression of

$$\begin{aligned}&T_{\frac{N}{8}}(n_4, k_1, k_2, k_3)\\&= H_{\frac{N}{4}}(n_4, k_1, k_2) + W_N^{\frac{N}{8}(k_1+2k_2+4k_3)} H_{\frac{N}{4}(n_4+\frac{N}{8}, k_1, k_2)}\end{aligned} \tag{4.12}$$

As in radix-2^2 algorithm, the first two columns of butterflies contain only trivial multiplications. The third butterfly contains a special twiddle factor

$$W_N^{\frac{N}{8}(k_1+2k_2+4k)} = (\frac{1}{\sqrt{2}}(1-j))_1^k(-j)^{k_2+2k_3} \tag{4.13}$$

It can be easily seen that applying this twiddle factor requires only two real multiplications. Full complex multiplications are used to apply the decomposed twiddle factor $W_N^{n_4(k_1+2k_2+4k_3)}$ after the third column. An $N = 64$ example is shown in Fig. 4.3.

4.2 Related Work

Fast Fourier Transform (FFT) is widely used in the field of digital signal processing (DSP) such as filtering, spectral analysis etc., to compute the discrete Fourier transform (DFT). FFT plays a critical role in modern digital communications such as digital video broadcasting and orthogonal frequency division multiplexing (OFDM) systems. Much research has been carried out on designing pipelined architectures for computation of FFT of complex valued signals. Various algorithms have been developed to reduce the computational complexity, of which Cooley-Tukey radix-2 FFT [25] is very popular.

Algorithms including radix-4 [28], split-radix [29], radix-2^2 [26] have been developed based on the basic radix-2 FFT approach. The architectures based on these algorithms are some of the standard FFT architectures in the literature [30]-[35]. Radix-2 Multi-path delay commutator (R2MDC) [30] is one of the most classical approaches for pipelined implementation of radix-2 FFT is shown in Fig. Efficient usage of the storage buffer in R2MDC leads to Radix-2 Single-path delay feedback (R2SDF) architecture with reduced memory [31]. Fig shows a radix-2 feedback pipelined architecture for $N = 16$ points. R4MDC [32] and R4SDF [33], [34] are proposed as radix-4 versions of R2MDC and R4SDF respectively. Radix-4 single-path delay commutator (R4SDC)

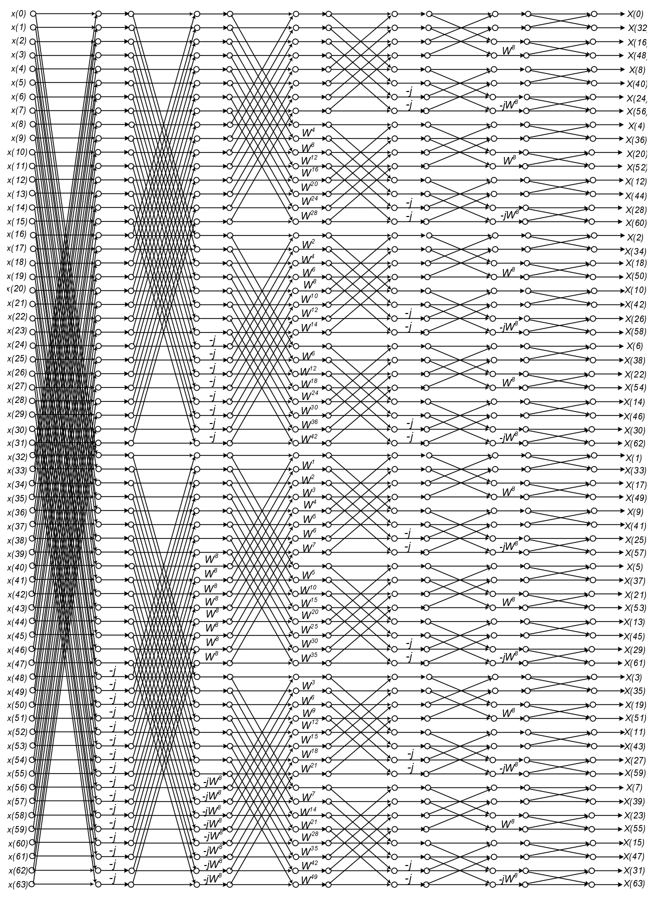


Figure 4.3: Flow graph of a 64-point radix-2^3 DIF Complex FFT

[35] is proposed using a modified radix-4 algorithm to reduce the complexity of R4MDC architecture.

Many parallel architectures for FFT have been proposed in the literature [36]-[39]. A formal method of developing these architectures from the algorithms has not been proposed till now. Further, most of these hardware architectures are not fully utilized which leads to high hardware complexity. In the era of high speed digital communications, high throughput and low power designs are required to meet the speed and power requirements while keeping the hardware overhead to minimum. In this thesis, we present a new approach to design these architectures from the FFT flow graphs. Folding transformation [40] and register minimization techniques [41], [42] are used to derive several known FFT architectures. Novel architectures are developed using the proposed methodology which have not been presented in the literature.

In the folding transformation, all butterflies in the same column can be mapped to one butterfly unit. If the FFT size is N, then this corresponds to a folding factor of $N/2$. This leads to a 2-parallel architecture. In another design, we can choose a folding factor of $N/4$ to design a 4-parallel architectures, where 4 samples are processed in the same clock cycle. Different folding sets lead to a family of FFT architectures. Alternatively, known FFT architectures can also be described by the proposed methodology by selecting the appropriate folding set. Folding sets are designed intuitively to reduce latency and to reduce the number of storage elements. It may be noted that prior FFT architectures were derived in an *adhoc* way, and their derivations were not explained in a systematic way. This is the first attempt to generalize design of FFT architectures for arbitrary level of parallelism in a systematic manner via the folding transformation. In this paper, design of prior architectures is first explained by constructing specific folding sets. Then, several new architectures are derived for various radices, and various levels of parallelism, and for either the decimation-in-time (DIT) or the decimation-in-frequency (DIF) flow graphs. All new architectures achieve full hardware utilization. It may be noted that all prior parallel FFT architectures did not achieve full hardware utilization.

4.3 FFT Architectures via Folding

In this section, we present a method to derive several known FFT architectures in general. The process is described using an 8-point radix-2 DIF FFT as an example. It can be extended to other radices in a similar fashion. Fig. 4.4 shows the flow graph of a radix-2 8-point DIF FFT. The graph is divided into 3 stages and each of them consists of a set of butterflies and multipliers. The twiddle factor in between the stages indicates a multiplication by W_N^k, where W_N denotes the Nth root of unity, with its exponent evaluated modulo N. This algorithm can be represented as a data flow graph (DFG) as shown in Fig. 4.5. The nodes in the DFG represent tasks or computations. In this case, all the nodes represent the butterfly computations of the radix-2 FFT algorithm. In particular, assume nodes A and B have the multiplier operation on the bottom edge of the butterfly.

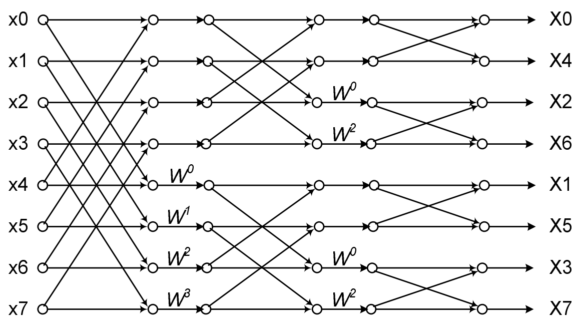


Figure 4.4: Flow graph of a radix-2 8-point DIF FFT.

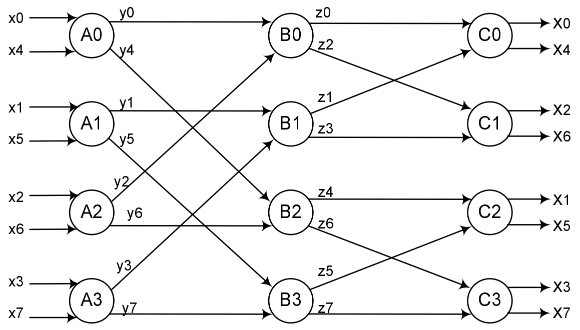


Figure 4.5: Data Flow graph (DFG) of a radix-2 8-point DIF FFT.

The folding transformation is used on the DFG in Fig. 4.5 to derive a pipelined architecture [15]. To transform the DFG, we require a folding set, which is an ordered set of operations executed by the same functional unit. Each folding set contains K entries some of which may be null operations. K is called the folding factor, the number of operations folded into a single function unit. The operation in the j-th position within the folding set (where j goes from 0 to $K-1$) is executed by the functional unit during the time partition j. The term j is the folding order, the time instance to which the node is scheduled to be executed in hardware.

For example, consider the folding set $A = \{\phi, \phi, \phi, \phi, A0, A1, A2, A3\}$ for $K = 8$. The operation $A0$ belongs to the folding set A with the folding order 4. The functional unit A executes the operations $A0, A1, A2, A3$ at the respective time instances and will be idle during the null operations. We use the systematic folding techniques to derive the 8-point FFT architecture. Consider an edge e connecting the nodes U and V with $w(e)$ delays. Let the executions of the l-th iteration of the nodes U and V be scheduled at the time units $Kl + u$ and $Kl + v$, respectively, where u and v are the folding orders of the nodes U and V. The folding equation for the edge e is

$$D_F(U \to V) = Kw(e) - P_U + v - u \tag{4.14}$$

where P_U is the number of pipeline stages in the hardware unit which executes the node U [40].

4.3.1 Feed-forward Architecture

Consider folding of the DFG in Fig. 4.5 with the folding sets

$$\begin{aligned} A &= \{\phi, \phi, \phi, \phi, A0, A1, A2, A3\}, \\ B &= \{B2, B3, \phi, \phi, \phi, \phi, B0, B1\}, \\ C &= \{C1, C2, C3, \phi, \phi, \phi, \phi, C0\}. \end{aligned}$$

Assume that the butterfly operations do not have any pipeline stages, i.e., $P_A = 0$, $P_B = 0$, $P_C = 0$. The folded architecture can be derived by writing the folding equation in (4.14) for all the edges in the DFG. These equations are

$$\begin{aligned} &D_F(A0 \to B0) = 2 && D_F(B0 \to C0) = 1 \\ &D_F(A0 \to B2) = -4 && D_F(B0 \to C1) = -6 \\ &D_F(A1 \to B1) = 2 && D_F(B1 \to C0) = 0 \\ &D_F(A1 \to B1) = -4 && D_F(B1 \to C1) = -7 \\ &D_F(A2 \to B0) = 0 && D_F(B2 \to C2) = 1 \\ &D_F(A2 \to B2) = -6 && D_F(B2 \to C3) = 2 \\ &D_F(A3 \to B1) = 0 && D_F(B3 \to C2) = 0 \\ &D_F(A3 \to B3) = -6 && D_F(B3 \to C3) = 1 \end{aligned} \tag{4.15}$$

For example, $D_F(A0 \to B0) = 2$ means that there is an edge from the butterfly node A to node B in the folded DFG with 2 delays. For the folded system to be realizable, $D_F(U \to V) \geq 0$ must hold for all the edges in the DFG. Retiming and/or pipelining can be used to either satisfy this property or determine that the folding sets are not feasible [15]. We can observe the negative delays on some edges in (4.15). The DFG

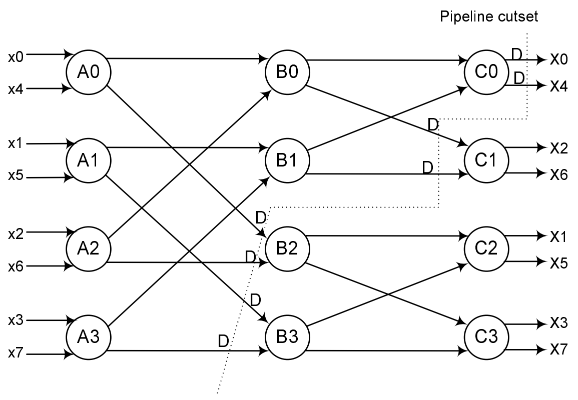


Figure 4.6: Pipelined Data Flow graph (DFG) of a 8-point DIF FFT as a preprocessing step for folding

can be pipelined as shown in Fig. 4.6 to ensure that folded hardware has a non-negative number of delays. The updated folding equations for all the edges are

$$\begin{aligned}
&D_F(A0 \rightarrow B0) = 2 \qquad && D_F(B0 \rightarrow C0) = 1\\
&D_F(A0 \rightarrow B2) = 4 && D_F(B0 \rightarrow C1) = 2\\
&D_F(A1 \rightarrow B1) = 2 && D_F(B1 \rightarrow C0) = 0\\
&D_F(A1 \rightarrow B1) = 4 && D_F(B1 \rightarrow C1) = 1\\
&D_F(A2 \rightarrow B0) = 0 && D_F(B2 \rightarrow C2) = 1\\
&D_F(A2 \rightarrow B2) = 2 && D_F(B2 \rightarrow C3) = 2\\
&D_F(A3 \rightarrow B1) = 0 && D_F(B3 \rightarrow C2) = 0\\
&D_F(A3 \rightarrow B3) = 2 && D_F(B3 \rightarrow C3) = 1
\end{aligned} \tag{4.16}$$

From (4.16), we can observe that 24 registers are required to implement the folded architecture. Lifetime analysis technique [42] is used to design the folded architecture that use the minimum possible registers. For example, in the current 8-point FFT design, consider the variables $y0, y1, ...y7$ i.e., the outputs at the nodes $A0, A1, A2, A3$ respectively. It takes 16 registers to synthesize these edges in the folded architecture. The linear lifetime chart for these variables is shown in Fig. 4.7. From the lifetime chart, it can be seen that the folded architecture requires 4 registers as opposed to 16 registers in a straightforward implementation. The next step is to perform forward-backward

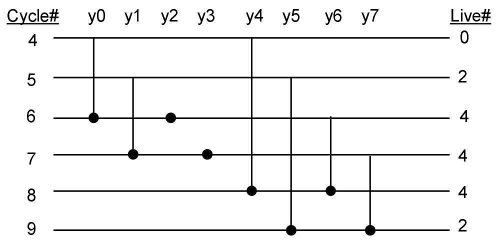
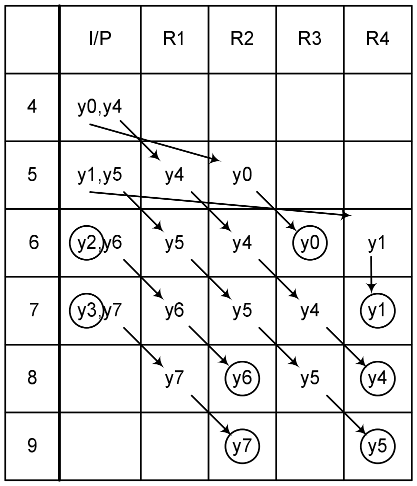


Figure 4.7: Linear lifetime chart for the variables $y0, y1, ...y7$.

register allocation. The allocation table is shown in Fig. 4.8. From the allocation table in Fig. 4.8 and the folding equations in (4.16), the delay circuit in Fig. 4.9 can be synthesized. Fig. 4.10 shows how node A and node B are connected in the folded architecture.



	I/P	R1	R2	R3	R4
4	y0,y4				
5	y1,y5	y4	y0		
6	y2,y6	y5	y4	y0	y1
7	y3,y7	y6	y5	y4	y1
8		y7	y6	y5	y4
9			y7		y5

Figure 4.8: Register allocation table for the data represented in Fig. 4.7.

The control complexity of the derived circuit is high. Four different signals are needed to control the multiplexers. A better register allocation is found such that the number of multiplexers are reduced in the final architecture. The more optimized register allocation for the same lifetime analysis chart is shown in Fig. 4.11. Similarly, we can apply lifetime analysis and register allocation techniques for the variables $x0, ..., x7$ and $z0, ..., z7$, inputs to the DFG and the outputs from nodes $B0, B1, B2, B3$ respectively as shown in Fig. 4.5. From the allocation table in Fig. 4.11 and the folding equations

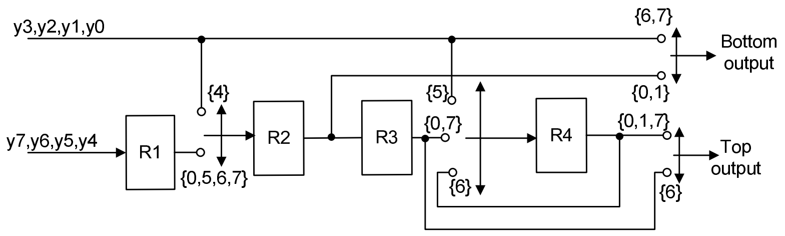


Figure 4.9: Delay circuit for the register allocation table in Fig. 4.8.

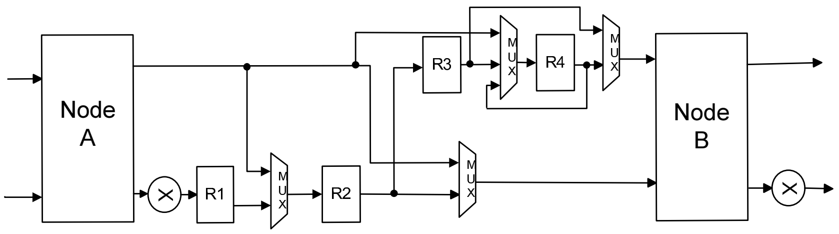


Figure 4.10: Folded circuit between Node A and Node B.

in (4.16), the final architecture in Fig. 4.12 can be synthesized.

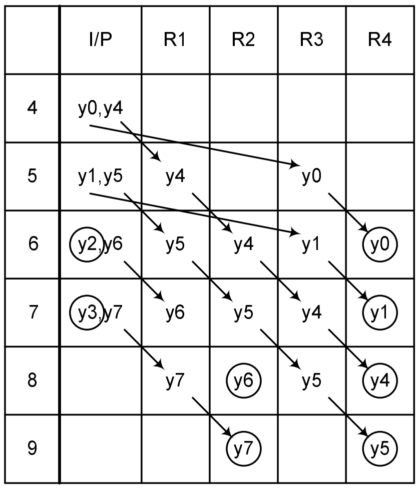
We can observe that the derived architecture is the same as R2MDC architecture [30]. Similarly, the R2SDF architecture can be derived by minimizing the registers on all the variables at once.

4.3.2 Feedback Architecture

We derive the feedback architecture using the same 8-point radix-2 DIT FFT example in Fig. 4.4. Consider the following folding sets

$$A = \{\phi, \phi, \phi, \phi, A0, A1, A2, A3\},$$
$$B = \{\phi, \phi, B2, B3, \phi, \phi, B0, B1\},$$
$$C = \{\phi, C1, \phi, C2, \phi, C3, \phi, C0\}.$$

Assume that the butterfly operations do not have any pipeline stages, i.e., $P_A = 0$, $P_B = 0$, $P_C = 0$. The folded architecture can be derived by writing the folding equation



	I/P	R1	R2	R3	R4
4	y0,y4				
5	y1,y5	y4		y0	
6	y2,y6	y5	y4	y1	y0
7	y3,y7	y6	y5	y4	y1
8		y7	y6	y5	y4
9			y7		y5

Figure 4.11: Register allocation table for the data represented in Fig. 4.7.

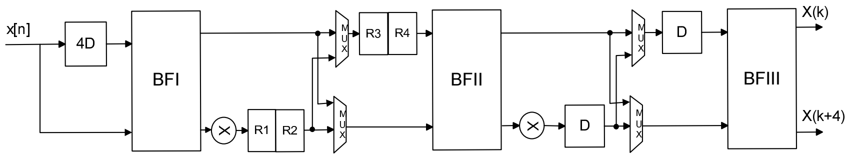


Figure 4.12: Folded architecture for the DFG in Fig. 4.6. This corresponds to the well known radix-2 feed-forward (R2MDC) architecture.

in (4.14) for all the edges in the DFG. The folding equations are

$$\begin{array}{ll}
D_F(A0 \rightarrow B0) = 2 & D_F(B0 \rightarrow C0) = 1 \\
D_F(A0 \rightarrow B2) = -2 & D_F(B0 \rightarrow C1) = -5 \\
D_F(A1 \rightarrow B1) = 2 & D_F(B1 \rightarrow C0) = 0 \\
D_F(A1 \rightarrow B1) = -2 & D_F(B1 \rightarrow C1) = -6 \\
D_F(A2 \rightarrow B0) = 0 & D_F(B2 \rightarrow C2) = 1 \\
D_F(A2 \rightarrow B2) = -4 & D_F(B2 \rightarrow C3) = 3 \\
D_F(A3 \rightarrow B1) = 0 & D_F(B3 \rightarrow C2) = 0 \\
D_F(A3 \rightarrow B3) = -4 & D_F(B3 \rightarrow C3) = 2
\end{array} \tag{4.17}$$

It can be seen from the folding equations in (4.17) that some edges contain negative delays. Retiming is used to make sure that the folded hardware has non-negative number

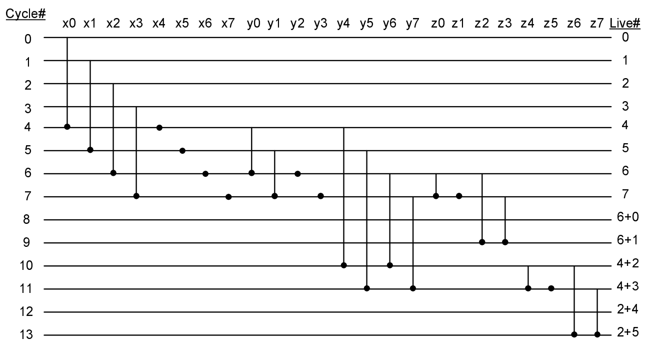


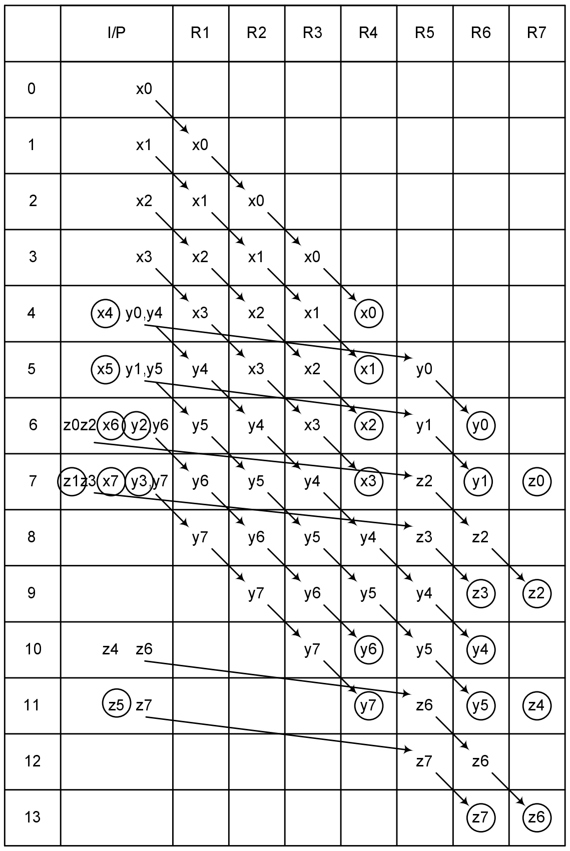
Figure 4.13: Linear lifetime chart for variables for a 8-point FFT architecture.

of delays. The pipelined DFG is the same as the one in the feed-forward example and is shown in Fig. 4.6. The updated folding equations are shown in (4.18).

$$\begin{aligned}
&D_F(A0 \rightarrow B0) = 2 \qquad && D_F(B0 \rightarrow C0) = 1\\
&D_F(A0 \rightarrow B2) = 6 && D_F(B0 \rightarrow C1) = 3\\
&D_F(A1 \rightarrow B1) = 2 && D_F(B1 \rightarrow C0) = 0\\
&D_F(A1 \rightarrow B1) = 6 && D_F(B1 \rightarrow C1) = 2\\
&D_F(A2 \rightarrow B0) = 0 && D_F(B2 \rightarrow C2) = 1\\
&D_F(A2 \rightarrow B2) = 4 && D_F(B2 \rightarrow C3) = 3\\
&D_F(A3 \rightarrow B1) = 0 && D_F(B3 \rightarrow C2) = 0\\
&D_F(A3 \rightarrow B3) = 4 && D_F(B3 \rightarrow C3) = 2
\end{aligned} \tag{4.18}$$

The number of registers required to implement the folding equations in (4.18) is 40. The linear life time chart is shown in Fig. 4.13 and the register allocation table is shown in Fig. 4.14. We can implement the same equations using 7 registers by using these register minimization techniques. The folded architecture in Fig. 4.15 is synthesized using the folding equations in (4.18) and register allocation table.

The hardware utilization is only 50% in the derived architecture. This can also be observed from the folding sets where half of the time null operations are being executed, i.e., hardware is idle. Using this methodology, we present low-power parallel FFT architectures in the next chapter.



	I/P	R1	R2	R3	R4	R5	R6	R7
0	x0							
1	x1	x0						
2	x2	x1	x0					
3	x3	x2	x1	x0				
4	x4 y0,y4	x3	x2	x1	x0			
5	x5 y1,y5	y4	x3	x2	x1	y0		
6	z0z2 x6 y2 y6	y5	y4	x3	x2	y1	y0	
7	z1z3 x7 y3 y7	y6	y5	y4	x3	z2	y1	z0
8		y7	y6	y5	y4	z3	z2	
9			y7	y6	y5	y4	z3	z2
10	z4 z6			y7	y6	y5	y4	
11	z5 z7				y7	z6	y5	z4
12						z7	z6	
13							z7	z6

Figure 4.14: Register allocation table for the data represented in Fig. 4.13.

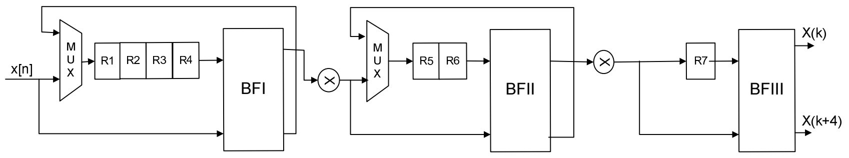


Figure 4.15: Folded architecture for the DFG in Fig. 4.6. This corresponds to the well known radix-2 feedback (R2SDF) architecture.

Chapter 5

Novel Parallel FFT Architectures

This chapter shows the application of the new approach presented in chapter 4 to design novel FFT architectures. A family of high throughput FFT architectures have been obtained by using the new approach based on radix-2 algorithm. They are feed-forward architectures with different number of input samples in parallel. The parallelization can be arbitrarily chosen. Furthermore, architectures based on radix-2^2 and radix-2^3 algorithms have also been developed. The circuits to reorder the output samples are also presented.

5.1 Radix-2 FFT Architectures

In this section, we present parallel architectures for complex valued signals based on radix-2 algorithm. These architectures are derived using the approach presented in the previous section. The same approach can be extended to radix-2^2, radix-2^3 and other radices as well. Due to space constraints, only folding sets are presented for different architectures. The folding equations and register allocation tables can be obtained easily.

5.1.1 2-parallel Radix-2 FFT Architecture

The utilization of hardware components in the feedforward architecture is only 50%. This can also be observed from the folding sets of the DFG where half of the time null operations are being executed. We can derive new architectures by changing the folding

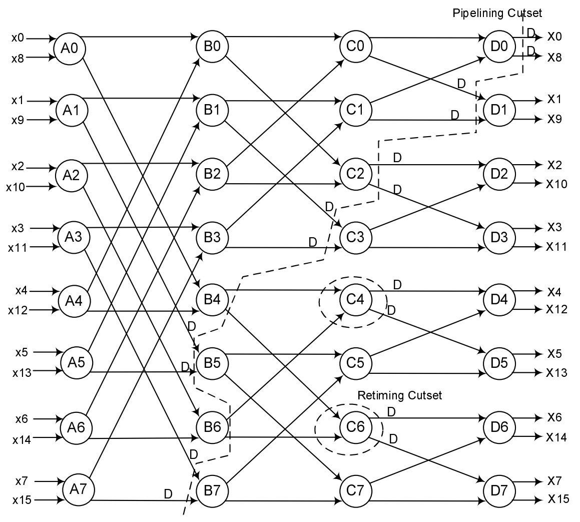


Figure 5.1: Data Flow graph (DFG) of a Radix-2 16-point DIF FFT with retiming for folding.

sets which can lead to efficient architectures in terms of hardware utilization and power consumption. We present here one such example of a 2-parallel architecture which leads to 100% hardware utilization and consumes less power.

Fig. 5.1 shows the DFG of radix-2 DIF FFT for $N = 16$. All the nodes in this figure represent radix-2 butterfly operations. Assume the nodes A, B and C contain the multiplier operation at the bottom output of the butterfly. Consider the folding sets

$$\begin{aligned} A &= \{A0, A2, A4, A6, A1, A3, A5, A7\}, \\ B &= \{B5, B7, B0, B2, B4, B6, B1, B3\}, \\ C &= \{C3, C5, C7, C0, C2, C4, C6, C1\}, \\ D &= \{D2, D4, D6, D1, D3, D5, D7, D0\} \end{aligned} \tag{5.1}$$

We can derive the folded architecture by writing the folding equation in (4.14) for all the edges. Pipelining and retiming are required to get non-negative delays in the folded architecture. The DFG in Fig. 5.1 also shows the retimed delays on some of the edges of the graph. The final folded architecture is shown in Fig. 5.2. The register minimization techniques and forward-backward register allocation are also applied in deriving this architecture as described in Section II. Note the similarity of the datapath

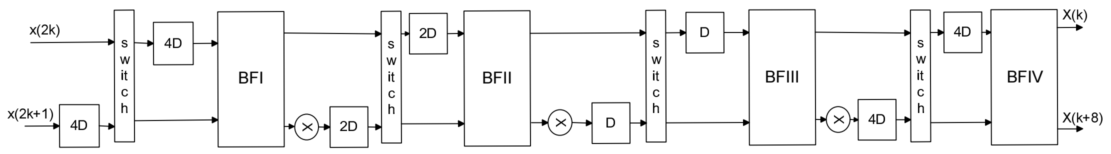


Figure 5.2: Proposed 2-parallel (Architecture 1) for the computation of a radix-2 16-point DIF FFT.

to R2MDC. This architecture processes two input samples at the same time instead of one sample in R2MDC. The implementation uses regular radix-2 butterflies. Due to the spatial regularity of the radix-2 algorithm, the synchronization control of the design is very simple. A log_2N-bit counter serves two purposes: synchronization controller i.e., the control input to the switches, and address counter for twiddle factor selection in each stage.

We can observe that the hardware utilization is 100% in this architecture. In a general case of N-point FFT, with N power of 2, the architecture requires $log_2(N)$ complex butterflies, $log_2(N) - 1$ complex multipliers and $3N/2 - 2$ delay elements or buffers.

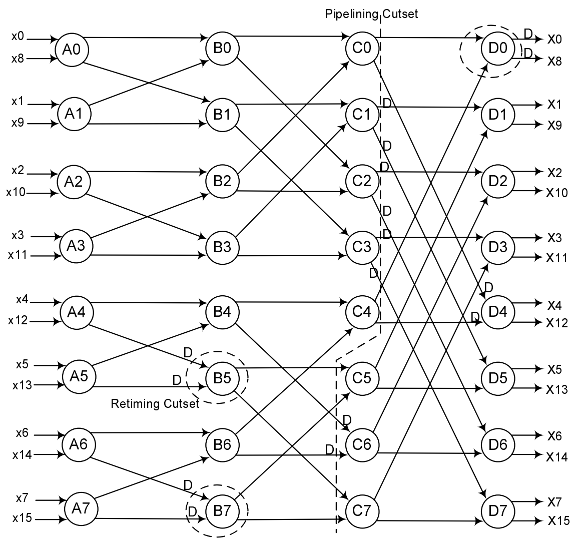


Figure 5.3: Data Flow graph (DFG) of a Radix-2 16-point DIT FFT with retiming for folding.

In a similar manner, we can derive the 2-parallel architecture for Radix-2 DIT FFT

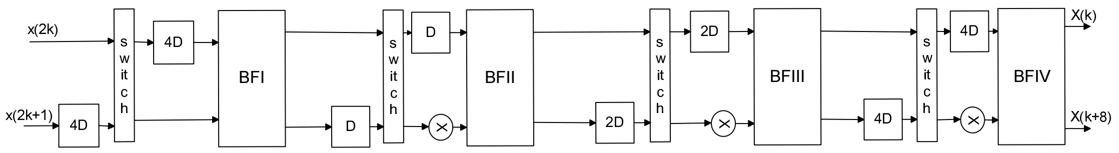


Figure 5.4: Proposed 2-parallel (Architecture 1) for the computation of a radix-2 16-point DIT Complex FFT.

using the following folding sets. Assume that multiplier is at the bottom input of the nodes B, C, D.

$$A = \{A0, A2, A1, A3, A4, A6, A5, A7\},$$
$$B = \{B5, B7, B0, B2, B1, B3, B4, B6\},$$
$$C = \{C6, C5, C7, C0, C2, C1, C3, C4\},$$
$$D = \{D2, D1, D3, D4, D6, D5, D7, D0\}$$

The pipelined/retimed version of the DFG is shown in Fig. 5.3 and the 2-parallel architecture is in Fig. 5.4. The only difference in the two architectures (Fig. 5.2 and Fig. 5.4) is the position of the multiplier in between the butterflies. The rest of the design remains same.

5.1.2 4-parallel Radix-2 FFT Architecture

A 4-parallel architecture can be derived using the following folding sets.

$$A = \{A0, A1, A2, A3\} \qquad A' = \{A'0, A'1, A'2, A'3\}$$
$$B = \{B1, B3, B0, B2\} \qquad B' = \{B'1, B'3, B'0, B'2\}$$
$$C = \{C2, C1, C3, C0\} \qquad C' = \{C'2, C'1, C'3, C'0\}$$
$$D = \{D3, D0, D2, D1\} \qquad D' = \{D'3, D'0, D'2, D'1\}$$

The DFG shown in Fig. 5.5 is retimed to get the non-negative folded delays. The final architecture in Fig. 5.6 can be obtained following the proposed approach. For a N-point FFT, the architecture takes $4(log_4 N - 1)$ complex multipliers and $2N - 4$ delay elements. We can observe that hardware complexity is almost double that of the serial architecture and processes 4-samples in parallel. The power consumption can be reduced by 50% (see Section V) by lowering the operational frequency of the circuit.

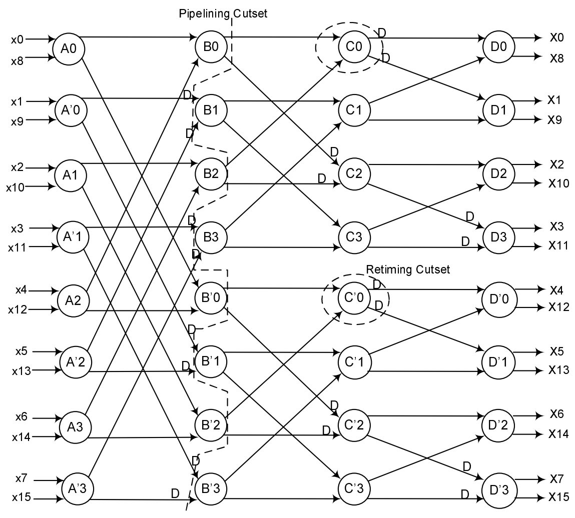


Figure 5.5: Data Flow graph (DFG) of a Radix-2 16-point DIF FFT with retiming for folding for 4-parallel architecture.

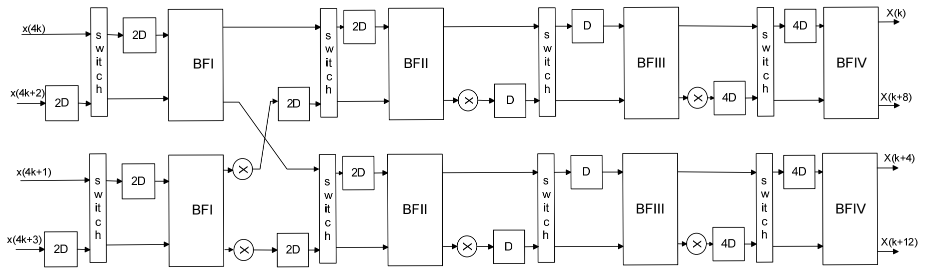


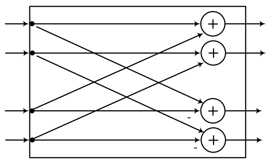
Figure 5.6: Proposed 4-parallel (Architecture 2) for the computation of 16-point radix-2 DIF FFT.

5.2 Radix-2^2 and Radix-2^3 FFT Architectures

The hardware complexity in the parallel architectures can be further reduced by using radix-2^n FFT algorithms. In this section, we consider the cases of radix-2^2 and radix-2^3 to demonstrate how the proposed approach can be used to radix-2^n algorithms. Similarly, we can develop architectures for radix-2^4 and other higher radices using the same approach.

5.2.1 2-parallel radix-2^2 FFT Architecture

The DFG of radix-2^2 DIF FFT for $N = 16$ will be similar to the DFG of radix-2 DIF FFT as shown in Fig. 5.1. All the nodes in this figure represent radix-2 butterfly operations including some special functionality. Nodes A and C represent regular butterfly operations. Nodes B and D are designed to include the $-j$ multiplication factor. Fig. 5.7 shows the butterfly logic needed to implement the radix-2^2 FFT. The factor $-j$ is handled in the second butterfly stage using the logic shown in Fig. 5.7b to switch the real and imaginary parts to the input of the multiplier.


(a) BFI

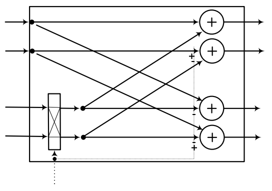

(b) BFII

Figure 5.7: Butterfly structure for the proposed FFT architecture in the complex datapath

Consider the folding sets

$$\begin{aligned}
A &= \{A0, A2, A4, A6, A1, A3, A5, A7\},\\
B &= \{B5, B7, B0, B2, B4, B6, B1, B3\},\\
C &= \{C3, C5, C7, C0, C2, C4, C6, C1\},\\
D &= \{D2, D4, D6, D1, D3, D5, D7, D0\} \qquad (5.2)
\end{aligned}$$

Using the folding sets above, the final architecture shown in Fig. 5.8 is obtained. We can observe that the number of complex multipliers required for radix-2^2 architecture is less compared to radix-2 architecture in Fig. 5.2. In general, for a N-point FFT, radix-2^2 architecture requires $2(log_4 N - 1)$ multipliers.

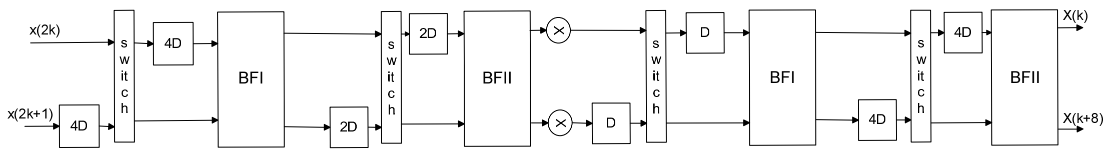


Figure 5.8: Proposed 2-parallel (Architecture 3) for the computation of a radix-2^2 16-point DIF FFT.

Similar to 4-parallel radix-2 architecture, we can derive 4-parallel radix-2^2 architecture using the similar folding sets. The 4-parallel radix-2^2 architecture is shown in Fig. 5.9. In general, for a N-point FFT, 4-parallel radix-2^2 architecture requires $3(log_4N-1)$ complex multipliers compared $4(log_4N-1)$ multipliers in radix-2 architecture. That is, the multiplier complexity is reduced by 25% compared to radix-2 architectures.

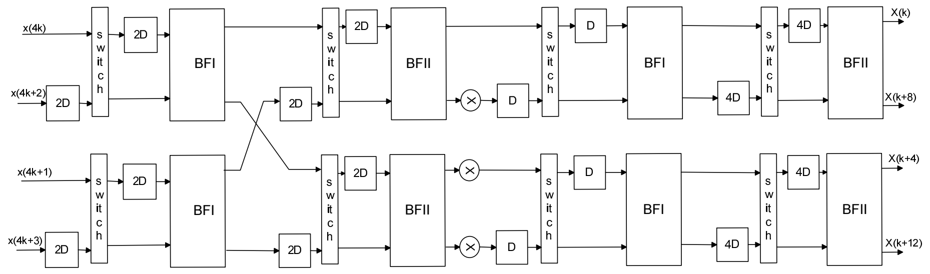


Figure 5.9: Proposed 4-parallel (Architecture 4) for the computation of a radix-2^2 16-point DIF FFT.

5.2.2 2-parallel radix-2^3 FFT Architecture

We consider the example of a 64-point radix-2^3 FFT algorithm [27]. The advantage of radix-2^3 over radix-2 algorithm is its multiplicative complexity reduction. A 2-parallel architecture is derived using folding sets in (5.2). Here the DFG contains 32 nodes instead of 8 in 16-point FFT.

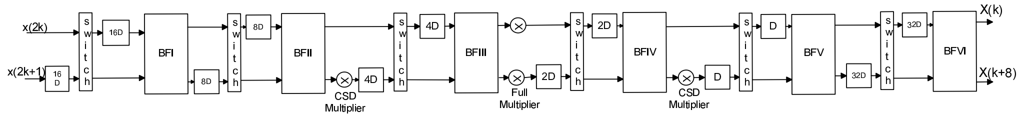


Figure 5.10: Proposed 2-parallel (Architecture 5) for the computation of 64-point radix-2^3 DIF FFT.

The folded architecture is shown in Fig. 5.10. The design contains only two full multipliers and two constant multipliers. The constant multiplier can be implemented using Canonic Signed Digit (CSD) format with much less hardware compared to a full multiplier. For a N-point FFT, where N is a power of 2^3, the proposed architecture takes $2(log_8 N - 1)$ multipliers and $3N/2 - 2$ delays. The multiplier complexity can be halved by computing the two operations using one multiplier. This can be seen in the modified architecture shown in Fig. 5.11. The only disadvantage of this design is that two different clocks are needed. Multiplier has to be run at double the frequency compared to the rest of the design. The architecture requires only $log_8 N - 1$ multipliers.

A 4-parallel radix-2^3 architecture can be derived similar to 4-parallel radix-2 FFT architecture. A large number of architectures can be derived using the approach presented in Section II. Using the folding sets of same pattern, 2-parallel and 4-parallel architectures can be derived for radix-2^2 and radix-2^4 algorithms. We show that changing the folding sets can lead to different parallel architectures. Further DIT and DIF algorithms will lead to similar architectures except the position of the multiplier operation.

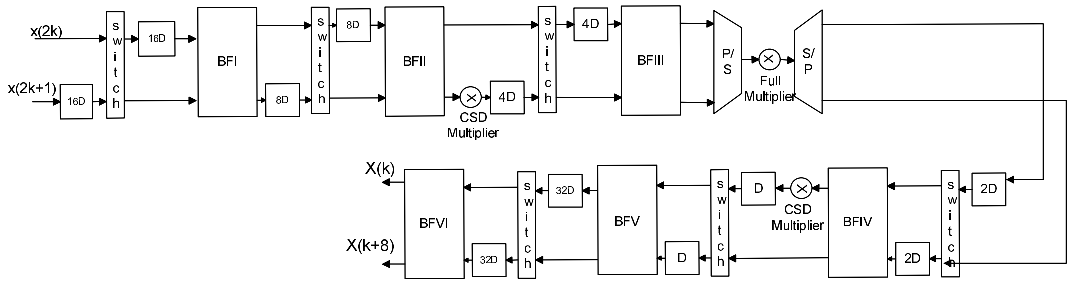
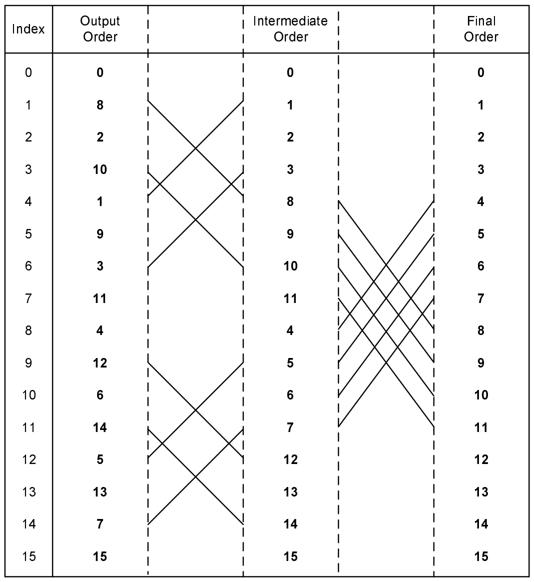


Figure 5.11: Proposed 2-parallel (Architecture 6) for the computation of 64-point radix-2^3 DIF FFT.

5.3 Reordering of the Output Samples

Reordering of the output samples is an inherent problem in FFT computation. The outputs are obtained in the bit-reversal order [30] in the serial architectures. In general the problem is solved using a memory of size N. Samples are stored in the memory in natural order using a counter for the addresses and then they are read in bit-reversal order by reversing the bits of the counter. In embedded DSP systems, special memory addressing schemes are developed to solve this problem. But in case of real-time systems, this will lead to an increase in latency and area.

The order of the output samples in the proposed architectures is not in the bit-reversed order. The output order changes for different architectures because of different folding sets/scheduling schemes. We need a general scheme for reordering these samples. One such approach is presented in this section.



Index	Output Order	Intermediate Order	Final Order
0	0	0	0
1	8	1	1
2	2	2	2
3	10	3	3
4	1	8	4
5	9	9	5
6	3	10	6
7	11	11	7
8	4	4	8
9	12	5	9
10	6	6	10
11	14	7	11
12	5	12	12
13	13	13	13
14	7	14	14
15	15	15	15

Figure 5.12: Solution to the reordering of the output samples for the architecture in Fig. 5.2.

The approach is described using a 16-point radix-2 DIF FFT example and the corresponding architecture is shown in Fig. 5.2. The order of output samples is shown in Fig. 5.12. The first column (index) shows the order of arrival of the output samples. The second column (output order) indicates the indices of the output frequencies. The goal is to obtain the frequencies in the desired order provided the order in the last column. We can observe that it is a type of de-interleaving from the output order and the final order. Given the order of samples, the sorting can be performed in two stages. It can be seen that the first and the second half of the frequencies are interleaved. The intermediate order can be obtained by de-interleaving these samples as shown in the table. Next, the final order can be obtained by changing the order of the samples. It can be generalized for higher number of points, the reordering can be done by shuffling the samples in the respective positions according to the final order required.

A shuffling circuit is required to do the de-interleaving of the output data. Fig. 5.13 shows a general circuit which can shuffle the data separated by R positions. If the multiplexer is set to "1" the output will be in the same order as the input, whereas setting it to "0" the input sample in that position is shuffled with the sample separated by R positions. The circuit can be obtained using lifetime analysis and forward-backward register allocation techniques. There is an inherent latency of R in this circuit.

The life time analysis chart for the 1st stage shuffling in Fig. 5.12 is shown in Fig. 5.14 and the register allocation table is in Fig. 5.15. Similar analysis can be done for the 2nd stage too. Combining the two stages of reordering in Fig. 5.12, the circuit in Fig. 5.16 performs the shuffling of the outputs to obtain them in the natural order. It uses seven complex registers for a 16-point FFT. In general case, a N-point FFT requires a memory of $5N/8 - 3$ complex data to obtain the outputs in natural order.

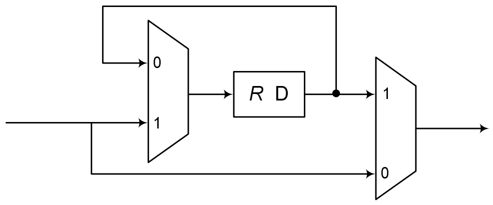


Figure 5.13: Basic circuit for the shuffling the data.

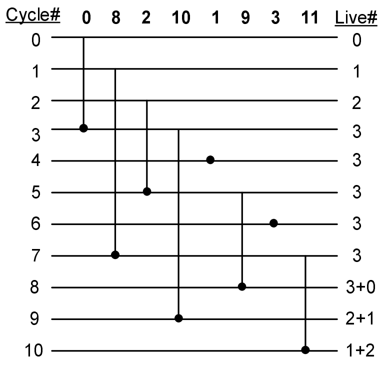
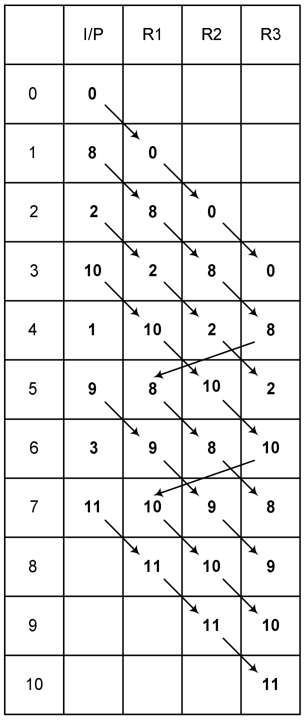


Figure 5.14: Linear lifetime chart for the 1st stage shuffling of the data.



	I/P	R1	R2	R3
0	0			
1	8	0		
2	2	8	0	
3	10	2	8	0
4	1	10	2	8
5	9	8	10	2
6	3	9	8	10
7	11	10	9	8
8		11	10	9
9			11	10
10				11

Figure 5.15: Register allocation table for the 1st stage shuffling of the data.

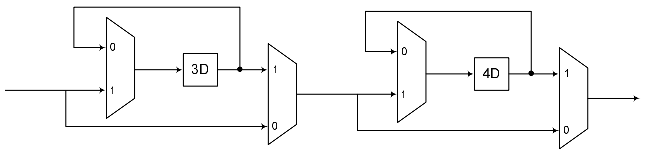


Figure 5.16: Structure for reordering the output data of 16-point DIF FFT.

5.4 Comparison and Analysis

A comparison is made between the previous pipelined architectures and the proposed ones for the case of computing an N-point complex FFT in Table 5.1. The comparison is made in terms of required number of complex multipliers, adders, delay elements and twiddle factors and throughput.

The proposed architectures are all feed-forward which can process 2 samples in parallel, thereby achieving a higher performance than previous designs which are serial in nature. When compared to some previous architectures, the proposed design doubles

Table 5.1: Comparison of pipelined hardware architectures for the computation of N-point FFT

Architecture	# Multipliers	# Adders	# Delays	# Twiddle factors	Throughput
R2MDC	$2(log_4N-1)$	$4log_4N$	$3N/2-2$	N	1
R2SDF	$2(log_4N-1)$	$4log_4N$	$N-1$	N	1
R4SDC	(log_4N-1)	$3log_4N$	$2N-2$	N	1
R2^{2}SDF	(log_4N-1)	$4log_4N$	$N-1$	N	1
R2^{3}SDF*	(log_8N-1)	$4log_4N$	$N-1$	N	1
Proposed Architectures					
Arch 1 (radix-2)	$2(log_4N-1)$	$4log_4N$	$3N/2-2$	N	2
Arch 2 (radix-2)	$4(log_4N-1)$	$8log_4N$	$2N-4$	N	4
Arch 3 (radix-2^2)	$2(log_4N-1)$	$4log_4N$	$3N/2-2$	N	2
Arch 4 (radix-2^2)	$3(log_4N-1)$	$8log_4N$	$2N-4$	N	4
Arch 5 (radix-2^3)*	$2(log_8N-1)$	$4log_4N$	$3N/2-2$	N	2
Arch 6 (radix-2^3)*	log_8N-1	$4log_4N$	$3N/2-2$	N	2

* These architectures need 2 constant multipliers as described in Radix-2^3*algorithm*

the throughput and halves the latency while maintaining the same hardware complexity. The proposed architectures maintain hardware regularity compared to previous designs.

5.4.1 Power Consumption

We compare power consumption of the serial feedback architecture with the proposed parallel feedforward architectures of same radix. The dynamic power consumption of a CMOS circuit can be estimated using the following equation,

$$P_{ser} = C_{ser}V^2 f_{ser}, \tag{5.3}$$

where C_{ser} denotes the total capacitance of the serial circuit, V is the supply voltage and f_{ser} is the clock frequency of the circuit. Let P_{ser} denotes the power consumption of the serial architecture.

In an L-parallel system, to maintain the same sample rate, the clock frequency must be decreased to f_{ser}/L. The power consumption in the L-parallel system can be calculated as

$$P_{par} = C_{par}V^2 \frac{f_{ser}}{L}, \tag{5.4}$$

where C_{par} is the total capacitance of the L-parallel system.

For example, consider the proposed architecture in Fig. 5.2 and R2SDF architecture [31]. The hardware overhead of the proposed architecture is 50% increase in the number of delays. Assume the delays account for half of the circuit complexity in serial architecture. Then $C_{par} = 1.25C_{ser}$ which leads to

$$\begin{aligned} P_{par} &= 1.25C_{ser}V^2\frac{f_{ser}}{2} \\ &= 0.625P_{ser} \end{aligned} \tag{5.5}$$

Therefore, the power consumption in a 2-parallel architecture has been reduced by 37% compared to the serial architecture.

Similarly, for the proposed 4-parallel architecture in Fig. 5.6, the hardware complexity doubles compared to R2SDF architecture. This leads to a 50% reduction in power compared to serial architecture.

Chapter 6

Conclusion and Discussion

This thesis has presented a complete mathematical proof to show that a transformation exists in state space to reduce the complexity of the parallel LFSR feedback loop. This leads to a novel method for high speed parallel implementation of linear feedback shift registers which is based on parallel IIR filter design. Our design can reduce the critical path without increasing the hardware cost at the same time. The design is applicable to any type of LFSR architecture. Further we show that using combined pipelining and parallel processing techniques of IIR filtering, critical path in the feedback part of the design can be reduced. The large fan-out effect problem can also be minimized with some hardware overhead by retiming around those particular nodes.

Further, a novel approach to derive the FFT architectures for a given algorithm. The proposed approach can be used to derive partly parallel architectures of any arbitrary parallelism level. Using this approach parallel architectures have been proposed for the computation of complex FFT based on radix-2^n algorithms. The power consumption can be reduced by 37% and 50% in proposed 2-parallel and 4-parallel architectures respectively. Future work will be directed towards design of FFT architectures for real-valued signals with full hardware utilization.

References

[1] T. V. Ramabadran and S.S. Gaitonde, "A Tutorial on CRC Computations," *IEEE Micro.*, Aug. 1988.

[2] R. E. Blahut, *Theory and Practice of Error Control Codes.* Reading, MA: Addison-Wesley, 1984

[3] W. W. Peterson and D. T. Brown, "Cyclic codes for errot detection", *Proc. IRE,* vol.49, pp. 228-235, Jan.1961

[4] N. Oh, R. Kapur, T. W. Williams, "Fast seed computation for reseeding shift register in test pattern compression," *IEEE ICCAD,* 2002, pp. 76-81.

[5] M. Y. Hsiao and K. Y. Sih, "Serial-to-parallel transformation of linear feedback shift register circuits", *IEEE Trans. Electronic Computers,* vol. EC-13, pp. 738-740, Dec. 1964

[6] A. M. Patel, "A multi-channel CRC register", in *AFIPS Conference Proceedings,* vol. 38, pp. 11-14, Spring 1971

[7] Tong-Bi Pei, Charles Zukowski, "High-Speed Parallel CRC circuits in VLSI", *IEEE Trans. on Communications*, vol. 40, no. 4, April 1992 pp. 653-657.

[8] K. K. Parhi, "Eliminating the fanout bottleneck in parallel long BCH encoders", *IEEE Transactions on Circuits and Systems I, Reg. Papers,* vol. 51, no. 3, pp. 512-516, Mar. 2004.

[9] C. Cheng, K. K. Parhi, "High Speed Parallel CRC Implementation based on Unfolding, Pipelining, Retiming," *IEEE Transaction on Circuits and Systems II, Express Briefs,* vol. 53, no. 10, pp. 1017-1021, Oct. 2006.

[10] C. Cheng, K. K. Parhi, ”High Speed VLSI Architecture for General Linear Feedback Shift Register (LFSR) Structures,” *Proc. of 43rd Asilomar Conf. on Signals, Systems, and Computers,* Nov. 2009, Monterey, CA, pp. 713-717.

[11] X. Zhang and K. K. Parhi, ”High-speed architectures for parallel long BCH encoders,” in *Proc. ACM Great Lakes Symp. VLSI,* Boston, MA, April 2004, pp. 1-6

[12] R. J. Glaise, ”A two-step computation of cyclic redundancy code CRC-32 for ATM networks”, *IBM J. Res. Devel.,* vol. 41, pp. 705-709, Nov. 1997

[13] J. H. Derby, ”High Speed CRC computation using state-space transformation,” in *Proc. Global Telecomm. Conf. 2001, GLOBECOM'01*, vol. 1, pp. 166-170

[14] K. K. Parhi, D. G. Messerschmitt, ”Static-rate optimal schedulinh of iterative dataflow programs via optimum unfolding,” *IEEE Trans. on Computers,* vol. 40, no. 2, pp. 178-195, Feb. 1991.

[15] K. K. Parhi, *VLSI Digital Signal Processing Systems: Design and Implementation.* Hoboken, NJ: Wiley, 1999.

[16] G. Campobello, G. Patane, and M. Russo, ”Parallel CRC Realization,” *IEEE Trans. Computers,* vol. 52, no. 10, pp. 1312 - 1319, Oct 2003

[17] R. Lidl and H. Niederreiter, *Introduction to finite fields and their applications.* Cambridge University Press, 1986.

[18] K. Hoffman and R. Kunze, *Linear Algebra.* Englewood Cliffs, NJ: Prentice Hall, 1971

[19] G. Albertengo and R. Sisto, ”Parallel CRC generation”, *IEEE Micro,* vol. 10, pp. 63-71, Oct. 1990

[20] S. L. Ng and B. Dewar, ”Parallel realization of the ATM cell header CRC”, *Computer Commun.,* vol. 19, pp. 257-263, March 1996

[21] Manohar Ayinala, K. K. Parhi, ”Efficient Parallel VLSI Architecture for Linear Feedback Shift Registers”, *IEEE Workshop on SiPS,* pp. 52-57, Oct. 2010.

[22] Manohar Ayinala, K. K. Parhi, "High Speed Parallel Architectures for Linear Feedback Shift Registers", *IEEE Trans. on Signal Processing,* (under review)

[23] K. K. Parhi, D. G. Messerschmitt, "Pipeline interleaving and parallelism in recursive digital filters -part II: pipelined incremental block filtering," *IEEE Trans. on Acoustics, Speech and Signal Processing,* vol. 37, pp. 1118-1135, July 1989.

[24] M. Potkonjak, M. B. Srivastava, A. P. Chandraksan, "Multiple constant multiplications: efficient and versatile framework and algorithms for exploring common subexpression elimination", *IEEE Trans on Computer-Aided Design for Integrated Circuits and Systems*, 15(2), pp. 151-165, 1996

[25] J. W Cooley and J. Tukey, "An algorithm for machine calculation of complex fourier series," *Math. Comput.,* vol. 19, pp. 297-301, Apr. 1965

[26] S. He and M. Torkelson, "A new approach to pipeline FFT processor," *Proc. of IPPS,* 1996, pp. 766 - 770.

[27] S. He and M. Torkelson, "Design and Implementation of 1024-point pipeline FFT processor," *in Proc. Custom Integr. Circuits Conf.,* SantaClara, CA, May 1998, pp. 131-134.

[28] A. V. Oppenheim, R. W. Schafer, J. R. Buck, *Discrete-Time Singal Processing,* 2nd ed. Englewood Cliffs, NJ: Prentice Hall 1998.

[29] P. Duhamel, "Implementation of split-radix FFT algorithms for complex, real, and real-symmetric data," *IEEE Trans. on Acoust., Speech Signal Process.,* vol. 34, no. 2, pp. 285-295, Apr. 1986

[30] L. R. Rabiner, B. Gold, *Theory and Application of Digital Signal Processing.* Prentice Hall Inc., 1975.

[31] E. H. Wold and A. M. Despain, "Pipeline and parallel-pipeline FFT processors for VLSI implementation," *IEEE Trans. Computers,* C-33(5): 414-426, May 1984.

[32] A. M. Despain, "Fourier transfom using CORDIC iterations," *IEEE Trans. Comput.,* C-233(10): 993-1001, Oct. 1974.

[33] E. E. Swartzlander, W. K. W. Young, S.J. Joseph, "A radix-4 delay commutator for fast Fourier transform processor implementation," *IEEE Journal of Solid-state Cir.,* SC-19(5): 702-709, Oct. 1984.

[34] E. E. Swartzlander, V.K. Jain, H. Hikawa, "A radix-8 wafer scale FFT processor," *Journal. VLSI Signal Process.,* 4(2,3): 165-176, May 1992.

[35] G. Bi, E.V. Jones, "A pipelined FFT processor for word-sequential data," *IEEE Trans. Acoust., Speech, Signal Process.,* 37(12):1982-1985, Dec. 1989.

[36] Y. W. Lin, H. Y. Liu, C. Y. Lee, "A 1-GS/s FFT/IFFT processor for UWB applications," *IEEE Journal of Solid-state Circuits,* vol. 40, no.8 pp. 1726-1735, Aug. 2005.

[37] J. Lee, H. Lee, S. I. Cho, S. S. Choi, "A High-Speed two parallel radix-2^4 FFT/IFFT processor for MB-OFDM UWB systems," *IEICE Trans. on Fundamentals of Electronics, Communications and Computer Sciences,* pp. 1206-1211, April 2008.

[38] J. Palmer, B. Nelson, "A parallel FFT architecture for FPGAs", *Lecture Notes in Computer Science,* vol. 3203, pp. 948-953, 2004.

[39] M. Shin, H. Lee, "A high-speed four parallel radix-2^4 FFT/IFFT processor for UWB applications", *IEEE ISCAS 2008,* pp. 960 - 963, May 2008.

[40] K. K. Parhi, C. Y. Wang, A. P. Brown, "Synthesis of control circuits in folded pipelined DSP architectures," *IEEE Journal Solid State Circuits,* vol. 27, no. 1, pp. 29-43, 1992.

[41] K. K. Parhi, "Systematic synthesis of DSP data format converters using lifetime analysis and forward-backward register allocation," *IEEE Trans. on Circuits and Systems - II,* vol. 39, no. 7, pp. 423-440, July 1992.

[42] K. K. Parhi, "Calculation of minimum number of registers in arbitrary life time chart," *IEEE Trans. on Circuits and Systems - II,* vol. 41, no. 6, pp. 434-436, June 1995.