

REED SOLOMON KODÇÖZÜCÜ TASARIMI

REED SOLOMON DECODER DESIGN

LÜGEN ÜNAL

Hacettepe Üniversitesi

Fen Bilimleri Enstitüsü Yönetmeliğinin

Elektrik Elektronik Ana Bilim Dalı İçin Öngördüğü

YÜKSEK LİSANS TEZİ

olarak hazırlanmıştır.

2005

Fen Bilimleri Enstitüsü Müdürlüğü'ne,

Bu çalışma jürimiz tarafından ELEKTRİK ve ELEKTRONİK MÜHENDİSLİĞİ
ANABİLİM DALI 'nda YÜKSEK LİSANS TEZİ olarak kabul edilmiştir.

BAŞKAN

Prof.Dr. Selçuk GEÇİM

ÜYE (DANIŞMAN)

Yrd.Doç.Dr. Ali Ziya ALKAR

ÜYE

Yrd.Doç.Dr. Harun ALTUNER

ÜYE

Dr. Emre AKTAŞ

ÜYE

Dr. Umut SEZEN

ONAY

Bu tez/...../..... tarihinde Enstitü Yönetim Kurulunca belirlenen yukarıdaki jüri
üyeleri tarafından kabul edilmiştir.

...../...../.....

Prof.Dr. Ahmet R. ÖZDURAL

FEN BİLİMLERİ ENSTİTÜSÜ MÜDÜRÜ

REED SOLOMON KODÇÖZÜCÜ TASARIMI

Lügen ÜNAL

ÖZ

Sayısal iletişim sistemlerinde, kanaldaki gürültü ve diğer faktörlerden dolayı oluşan hataları düzeltmek için çeşitli kodlar geliştirilmiştir. Bu kodlardan bir tanesi de yaygın ve çeşitli kullanım alanları olan Reed Solomon kodlardır.

Reed Solomon kodlar üzerine yapılan bir çalışmada, Reed Solomon kodçözücü algoritması olarak kullanılan Berlekamp Massey algoritmasında farklı bir eşitlik kullanılarak uyumsuzluk hesabı yapılabilmiş ve kritik yol gecikmesi önceki algoritmalara göre daha az olan bir yapı oluşturulmuştur. Böylece günümüzün artan hızlı iletişim ihtiyacına cevap verebilmek amaçlanmıştır.

Bu tez çalışmasında Reed Solomon kodlar incelenmiştir. Öncelikle Reed Solomon kodlayıcı, daha sonra BM algoritması, onun bir türevi olan evirmesiz BM algoritması ve evirmesiz BM algoritmasında yapılan değişikliklerle elde edilen yeni algoritma için kodçözücü tasarımları gerçekleştirilmiştir. Sonuçta kodçözücülerin hızları karşılaştırıldığında yeni algoritma için tasarlanan yapının daha hızlı çalıştığı görülmüştür.

Çalışmanın sonuçları değerlendirildiğinde, değişken blok uzunluğundaki kodlar için ve hata düzeltimi yanında bilinmeyen bitlerin tahmininde kullanılan algoritmaların, ayrıca yüksek hız-düşük enerji tüketimi için yapılan algoritma tabanlı çalışmaların yeni algoritmaya uyarlanarak kodçözücü tasarımları gerçekleştirilebileceği ve yapılan tasarımın bu çalışmalara öncülük edeceği düşünülmektedir.

ANAHTAR SÖZCÜKLER: Reed Solomon Kodçözücü, Berlekamp-Massey Algoritması, VHDL tasarımı, FPGA

Danışman : Yard.Doç.Dr. Ali Ziya ALKAR, Hacettepe Üniversitesi, Elektrik ve Elektronik Mühendisliği Bölümü, Elektrik ve Elektronik Mühendisliği Anabilim Dalı

REED SOLOMON DECODER DESIGN

Lügen ÜNAL

ABSTRACT

In digital communication systems, various codes are used to correct errors caused by noise and other factors in the channel. One of them is Reed Solomon codes which have wide and different usage areas.

In a work based on Reed-Solomon codes with the formula change of discrepancy computation in Berlekamp-Massey (BM) algorithm used in the decoding process, a new architecture of the decoder with less critical path delay has been managed. So that, to answer the increasing demand on higher data rate communications.

In this thesis, RS codes have been investigated, firstly an Reed Solomon encoder, and then decoder designs are made based on BM algorithm, one of its derivation inversionless BM algorithm and for the new BM algorithm, In the end, when the speeds of the decoders were compared, it was seen that the decoder based on the new algorithm is the highest one.

When the results of this work were evaluated, algorithms for variable block length codes, for erasures as well as errors and algorithms achieving high speed-low power consumption can be adapted to the new algorithm, and it is thought that this design is going to be inspired by future works.

Keywords : Reed Solomon Decoder, Berlekamp-Massey Algorithm, VHDL design, FPGA

Advisor : Asst.Prof.Dr. Ali Ziya ALKAR, Hacettepe University, Department of Electrical and Electronics Engineering, Electrical and Electronics Engineering Section

TEŞEKKÜR :

Bu çalışmaya başlamamı sağlayan sayın danışmanım Yrd.Doç.Dr. Ali Ziya ALKAR' a ve çalışmam boyunca manevi desteklerini benden esirgemeyen aileme teşekkür ederim.

İÇİNDEKİLER DİZİNİ

	<u>Sayfa</u>
ÖZ.....	i
ABSTRACT.....	ii
TEŞEKKÜR.....	iii
İçindekiler Dizini.....	iv
Simgeler ve Kısaltmalar Dizini.....	vi
Çizelgeler Dizini.....	vii
Şekiller Dizini.....	viii
1. Giriş.....	1
2. Kodlama ve Kodlama Türleri.....	3
2.1. Reed-Solomon Kodlar.....	7
2.1.2. Reed-Solomon Kodların Kullanım Alanları.....	8
2.1.3. Galois Alanlar.....	9
3. Reed Solomon Kodlayıcı.....	11
3.1. Reed-Solomon Kodlayıcı Tasarımı.....	12
4. Reed Solomon Kodçözücü.....	15
4.1. Sendrom Değerlerinin Bulunması.....	17
4.2. Hata Yeri Belirleyici, Hata Değeri Belirleyici Polinom ve Anahtar Eşitlik.....	18
4.2.1. Anahtar Eşitliğin oluşturulması.....	19
4.2.2. Berlekamp-Massey Algoritması.....	21
4.3. Chien Araştırması ve Forney Algoritması.....	25
4.4. Reed-Solomon Kodçözücü Tasarımı.....	28
4.4.1. Berlekamp-Massey Algoritmasına Göre Tasarım.....	29
4.4.2. Evirmesiz Berlekamp-Massey Algoritmasına Göre Tasarım.....	31
4.4.2.1.DC Blok Yapısı.....	32

4.4.2.2 ELU Blok Yapısı.....	34
4.3. Yeni Evirmesiz Berlekamp-Massey Algoritmasına Göre Tasarım.....	35
4.4. Simülasyon Çalışmaları.....	40
6. Sonuç.....	48
KAYNAKLAR.....	50
EK-1 Kodlayıcı ve Kodçözücü İçin Kullanılan Paket VHDL Kodu.....	52
EK-2 Kodlayıcı VHDL Kodu.....	55
EK-3 Sendrom Değerlerini Hesaplayan Blok VHDL Kodu.....	58
EK-4 BM Algoritmasına Göre Anahtar Eşitliği Çözen Blok VHDL Kodu.....	62
EK-5 EBM Algoritmasına Göre Anahtar Eşitliği Çözen Blok VHDL Kodu.....	66
Ek-6 Yeni EBM Algoritmasına Göre Anahtar Eşitliği Çözen Blok VHDL Kodu..	68
EK-7 Chien Araştırma Bloğu ve RS Kodçözücü VHDL Kodu.....	75
ÖZGEÇMİŞ.....	83

SİMGELER VE KISALTMALAR DİZİNİ

BCH	: Bose Chaudhuri Hocquenghem
BM	: Berlekamp Massey
CD	: Compact Disc (Yoğunteker)
CPDP	: Cellular Digital Packet Data (Hücresel Sayısal Paket Veri)
CSEE	: Chien Search Error Evaluation (Chien Araştırması Hata Hesabı)
DC	: Discrepancy Computation (Uyumsuzluk Hesabı)
EBM	: Evirmesiz Berlekamp Massey
ELU	: Error Locator Polynomial Update (Hata Yeri Belirleyici Polinom Güncellemesi)
FPGA	: Field Programmable Gate Array (Alan Programlanabilir Kapı Dizisi)
FT	: Fourier Transform (Fourier Dönüşümü)
GCD	: Greatest Common Divisor (En Büyük Ortak Bölen)
GF	: Galois Field (Galois Alan)
RS	: Reed-Solomon
ITU	: International Telecommunication Union (Uluslar arası Teleiletişim Ortaklığı)
LFSR	: Linear Feedback Shift Register (Geri Beslemeli Kaydıran Yazmaç)
VHDL	: Very High Speed Hardware Description Language (Çok Yüksek Hızlı Donanım Tanımlama Dili)

ŞEKİLLER DİZİNİ

	<u>Sayfa</u>
Şekil 3.1.1. Data Polinomu Üretici Polinoma Bölen Devre.....	12
Şekil 3.1.2. Kodlayıcı Bloğu Simülasyon Sonucu.....	14
Şekil 4.1. RS kodçözümleme teknikleri.....	6
Şekil 4.3.1. Reed Solomon Kodçözücü Yapısı.....	27
Şekil 4.4.1. Alınan Polinomu $x-\alpha^i$ ile bölen devre.....	28
Şekil 4.4.2. Dört Bit Sembolü α ile çarpan devre.....	28
Şekil 4.4.3. Dört Bit Sembolü α^2 ile çarpan devre.....	29
Şekil 4.4.2.1.1. DC Blok Yapısı.....	33
Şekil 4.4.2.1.2. Kontrol Bloğu Yapısı.....	33
Şekil 4.4.2.2.1. ELU Bloklarının Yapısı.....	34
Şekil 4.3.1. RS Kodçözücü FPGA Serimi.....	39
Şekil 4.4.1. Sendrom Bloğu Simülasyon Sonucu.....	42
Şekil 4.4.2. Yeni EBM Algoritması İçin Simülasyon Sonucu.....	42
Şekil 4.4.3. Kodçözücü Simülasyon Sonuçları.....	43

ÇİZELGELER DİZİNİ

	<u>Sayfa</u>
Çizelge 4.4.1.1 BM Algoritması Değişkenleri.....	30
Çizelge 4.4.1.2 BM Algoritması xcv600 Yongası Sentez Sonuçları.....	31
Çizelge 4.4.2.2.1 EBM Algoritması xcv600 Yongası Sentez Sonuçları.....	34
Çizelge 4.3.1 Yeni EBM Algoritması xcv600 Yongası Sentez Sonuçları.....	38
Çizelge 5.1 Tasarım Sonuçları	49

1. GİRİŞ

Gelişen teknolojiyle beraber insanoğlunun hayal gücünün sınırları da genişlemektedir. Bugün artık insanlar, dünyanın en uzak mesafelerine ses, görüntü ve akla gelebilecek her türlü bilgiyi anında aktarmak istemektedirler. Bu da yüksek hızlı iletişime olan gereksinimi ortaya çıkarmaktadır.

Ancak sadece bilginin hızlı aktarımı kaliteli iletişim için yeterli olmamaktadır. İletişimin hızı kadar önemli bir diğer sorun da, hatasız iletişim yapılabilmesidir. Bunun için çeşitli kodlar geliştirilmiştir.

Bu kodlardan bir tanesi de Reed Solomon (RS) kodlardır. RS kodlar doğrusal sistematik blok kodlardır. Kodsözcüklerinin herhangi bir doğrusal birleşiminin yine bir kodsözcüğü olması doğrusal kod özelliğini getirir. İletilecek asıl verinin sonuna kontrol bilgisinin eklenmesi ise sistematik yapıyı oluşturmaktadır. RS kodların belirli sayıda sembollerin yer aldığı kodsözcüklerinden oluşması nedeniyle de bu kodlar, blok kodlardır. İlkel ve ilkel olmayan türleri vardır. İlkel türleri aynı zamanda çevrimsel kodlardır [1]. Çoğuşma hatalarıyla beraber rastgele oluşan hataları da düzeltebildiği için RS kodlar yaygın olarak kullanılan kodlardan bir tanesidir. RS kodların bir diğer özelliği de kodsözcüklerinin Galois alanlara (GF) ait olmasıdır. n blok uzunluğuna sahip ve m -bit sembollerden oluşan bir RS kod $GF(2^m)$ alanına aittir.

RS kodlar oluşturulurken kodlama işleminde iletilecek sembollerin oluşturduğu polinom, üretici polinom adı verilen bir polinoma bölünür, bölme sonucu elde edilen kalan semboller ise kontrol sembolleri olarak iletilecek veriye eklenir ve böylece kodlama işlemi tamamlanmış olur. RS kodlar için kodlama işlemi basit bir işlem olmasına rağmen kodçözme süreci karmaşık ve zordur.

Kodçözme sürecinde, hata yerleri ve hata değerlerine bağlı, doğrusal olmayan eşitlikler elde edilir. Bu eşitlikler arasında anahtar eşitlik denilen bir bağıntı bulunmuştur. Anahtar eşitlik çözülerek hata yerlerine ve hata değerlerine ulaşılmaya çalışılır. Kodçözme işleminin zor ve zaman alıcı kısmı anahtar eşitliğin çözümüdür. Bunun için çeşitli yöntemler geliştirilmiştir. Öncelikle Peterson, matris evirme tekniğine dayanan bir yöntem bulmuş, bu yöntem daha sonra Gorenstein ve Zierler tarafından geliştirilmiştir. Ancak onların yöntemi blok uzunluğu arttıkça

özümü zor ve verimsiz bir yöntem olmaktadır [1]. Anahtar eşitliğı özmek için hızlı ve verimli bir yöntem Berlekamp [5] tarafından geliştirilmiş, daha sonra bu yöntem Massey [8] tarafından da farklı bir şekilde bulunmuştur ve onların yöntemleri Berlekamp Massey (BM) algoritması olarak adlandırılmıştır. Sugiyama ise anahtar eşitliğın Euclid' in en büyük ortak bölen algoritmasıyla (GCD) [6] özölebildiğini göstermiştir.

Kodözömleme algoritması olarak kullanılan en etkili algoritmalar BM ve Euclid algoritmalarıdır. BM algoritması üzerine yapılan bir alıřmada [2], algoritmada gerekleřtirilen bir deęiřiklikle, yeni bir kodözöcü yapısı oluřturulması bařarılmıř ve bu yapının kritik yol gecikmesi önceki BM algoritmalarının ve Euclid algoritmasının kritik yol gecikmeleriyle karřılařtırıldığında oluřturulan yeni yapının kritik yol gecikmesinin daha az olduđu gözlemlenebilmiştir.

Bu tez alıřmasının ikinci kısmında eřitli kodlama türlerinden bahsedilmiş, ardından RS kodlar ve bu kodların kullanım alanlarına deęinilmiş ve RS kod sembollerinin ait olduđu Galois alanlar ve Galois alanlar için geerli olan aritmetik iřlemler üzerine bilgi verilmiştir.

Üüncü kısımda RS kodlayıcı yapısı ve kodlayıcının VHDL (ok Yüksek Hızlı Donanım Tanımlama Dili) [11,16] ile yapılan tasarımı anlatılmıştır. Dördüncü kısımda RS kodözöcü yapısına deęinilmiş, kodözme iřleminde kullanılan BM algoritması, BM algoritmasının bir türevi olan evirmesiz Berlekamp-Massey (EBM) algoritması ve EBM algoritmasında uyumsuzluk hesaplama yönteminde yapılan deęiřikle [2] elde edilen yeni algoritmadan bahsedilmiş ve bu algoritmalar için kodözöcü tasarımları gerekleřtirilerek, Xilinx xcv600 yongası için FPGA (Alan Programlanabilir Kapı Dizini) serimi oluřturulmuştur. Sonuç kısmında üç tasarımın sentez verileri karřılařtırılmıştır.

2. KODLAMA VE KODLAMA T RLER 

İletiřim s recinde, bilginin alıcı ve verici arasında aktarımı ger ekleřirken, iletiim ortamında (kanal) oluřan kayıplar ve g r lt n n etkisiyle, g nderilen veri ile alınan veri arasında hata dediėimiz farklar oluřur. Kanala ya da g r lt ye etki edebilmek g n m z kořullarında olası bir durum olmadıėından, bu hataları  nlemek de m mk n olmamaktadır. Ancak, hataları fark edebilmek ve hatta d zeltebilmek i in  eřitli kodlama y ntemleri geliřtirilmiřtir.

Kodlama t rlerini ikili ve ikili olmayan kodlar olarak iki ana gruba ayırabiliriz. İkili kodlamalarda '1' ve '0' biti olmak  zere, kodlanacak veride iki t r sembol vardır. İkili olmayan kodlarda ise semboller birden fazla bitten oluřur. Ancak t r  ne olursa olsun b t n kodlama t rlerinde yapılan temel iřlem aynıdır. Bu iřlem, g nderilecek veriye (mesaj) alıcı tarafından alındıėında hatasız iletiim yapılıp yapılmadıėının kontrol n  saėlayan semboller eklenmesidir. k-bit mesaj sembolleri alınır, bu sembollere kontrol ama lı kullanılan r-bit eklenerek n-bit ($n=k+r$) kods zc ė  olarak alıcıya g nderilir.

İkili kodların en basitlerinden bir tanesi yinelenen kodlardır. Bu t r kodlamada g nderilecek veri belirli sayıda kendini tekrar eder.  rneėin, mesaj biti='1' ve kontrol bitlerinin sayısı (r) 4 olduėunda kodlanmış veri "11111" olur. Alıcı kendisine gelen verideki '1' ve '0' bit sayılarını sayar, fazla sayıdaki bitin g nderilen mesaj olduėunu kabul eder. Yineleme kodlama y ntemi, hatayı  nlemek i in olduk a etkili bir y ntemdir. Blok uzunluėu arttık a alıcıda yanlış karar verme olasılıėı da azalır, ancak blok uzunluėunun artması bilgi aktarım oranının (k/n) d řmesine neden olur.

Yine ikili kodlama sınıfına ait kod t rlerinden bir tanesi de  ift veya tek-eřlik kontrol (even/odd-parity check control) kodlarıdır. Bu y ntemde mesaj bilgisindeki '1' bitleri sayılır, tek sayıda ise mesaj bitlerine kontrol biti olarak '1' (tek eřlik kontrol  i in '0'), deėilse '0' (tek eřlik kontrol  i in '1') eklenerek kods zc ė  alıcıya iletilir. Aynı řekilde alıcı da aldıėı kods zc ė nde yer alan mesaj bitlerindeki '1' bitlerini tekrar sayar, bu sayıya g re kontrol biti belirler ve vericiden gelen kontrol bitiyle karřılařtırma yapar, aynı ise hata olmadıėını varsayar. Tek/ ift eřlik kontrol ,  ift sayıda ger ekleřen hataları farkedemeyeceėi i in g venilir bir y ntem deėildir, ancak bu t r kodlama da bilgi aktarım oranı ($k/k+1$) olduk a y ksektir.

Diğer bir kod türü de, doğrusal kod sınıfına ait, kodsözcüğündeki tek bir bitin hatalı iletildiği durumlar için kullanılabilen Hamming kodlardır. Hamming kodlamada, kodsözcüğündeki bit sayısı kadar, sıfırdan ve birbirinden farklı ikilik sistemde sayılar belirlenir. Bu sayılar kodsözcüğündeki her bir bitin yerine karşılık gelir ve $(n-k) \times r$ boyutundaki eşlik-kontrol matrisi olarak adlandırılan matrisin sütunlarını oluşturur. Bu matris hata yerinin belirlenmesinde kullanılır.

Örneğin $n=6$ ve $k=3$ olan Hamming kod için eşlik-kontrol matrisi :

$$P = \begin{bmatrix} 110100 \\ 101010 \\ 011001 \end{bmatrix} \text{ olsun.}$$

Kodsözcüğü ise $C = [C_1 C_2 C_3 C_4 C_5 C_6]$ matrisi ile gösterilsin. C_1, C_2, C_3 mesaj bitleri, C_4, C_5, C_6 kontrol bitleridir. Kontrol bitleri, $P \times C^{-1} = 0$ eşitliğini sağlayacak şekilde seçilir ya da eşlik kontrol matrisi bu eşitliğe göre belirlenir. Bu örnekte eşlik kontrol matrisine göre, kontrol bitleri $C_3 = C_1 + C_2$, $C_4 = C_1 + C_3$, $C_5 = C_2 + C_3$ olacak şekilde seçilir. İkili sistemde toplama xor işlemidir. Bu durumda çarpım sonucu oluşan toplamalar ($C_1 + C_2 + C_3$, $C_1 + C_3 + C_4$, $C_2 + C_3 + C_5$) sıfıra eşittir ve böylece $P \times C^{-1} = 0$ olur. Dolayısıyla kanalda hatasız iletim gerçekleştiğinde alınan vektör bir kodsözcüğü olacağından eşlik kontrol matrisi ile tersinin çarpımı sıfırı verecektir. Hatalı iletişim gerçekleştiğinde ise eşlik kontrol matrisi ile alıcıya gelen verinin çarpımı, hata yerlerinin toplamını verecektir. Birden fazla hata olduğunda bu toplamdan tek tek hata yerlerine ulaşmak mümkün olmayıp, ancak tek bit hatalıyken eşlik kontrol matrisi ile çarpım hata yerinin kendisi olacaktır. Bu yüzden Hamming kodlar tek bit hatalarını düzeltebilen kodlardır.

Alıcıya gelen verinin (R) hatalı olduğunu kabul edelim. R' yi elde etmek için mesaj bilgisine hata vektörünü ekleriz. (Bir boyutlu matrisler vektör olarak da adlandırılır.)

$$R = [C_1 \ C_2 \ C_3 \ C_4 \ C_5 \ C_6] + [E_1 \ E_2 \ E_3 \ E_4 \ E_5 \ E_6] \quad (2.1)$$

Eşlik-kontrol matrisi ile R çarpıldığında $P \times C^{-1} = 0$ olduğundan sonuç :

$$P \times R^{-1} = \begin{bmatrix} 110100 \\ 101010 \\ 011001 \end{bmatrix} [E_1 \ E_2 \ E_3 \ E_4 \ E_5 \ E_6] \quad (2.2)$$

$$= E_1 \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix} + E_2 \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix} + E_3 \begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix} + E_4 \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} + E_5 \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} + E_6 \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \quad (2.3)$$

Bu durumda yalnızca bir bit hatalı olduğunda, çarpımın aldığı ve sendrom olarak adlandırılan değer, hatanın yerini gösterir ve yeri bilindiğinden hata düzeltilebilir. Birden fazla hata olduğunda sendrom değeri hata yerlerinin toplamı olacağından yanıltıcı olacaktır ya da hata yerlerine ulaşmak için yetersiz kalacaktır. Hamming kodlamada kodsözcüğünün uzunluğu en fazla $n=2^r-1$ (r-bit grubunun sıfırdan farklı alabileceği değerler) olabilir.

Tek bit hatalarını düzelten Hamming kodlardan yola çıkılarak birden fazla bitte hata olduğu durumlarda kullanılabilen kodlar için çalışmalar yapılmış ve bunların sonucunda ikili Bose Chaudhuri Hocquenghem (BCH) kodlar geliştirilmiştir. Bu kodlar iki-bit hatalarını düzeltebilirler. BCH kodlarda Hamming kodlardan farklı olarak eşlik-kontrol matrisinde fazladan r satır daha vardır. Bu r satırdan hata yerlerinin küplerinin toplamı elde edilir. Hata yerlerinin X_1 , X_2 olduğunu kabul edersek ilk r satırın kodsözcüğü ile çarpımından $X_1 + X_2$ (Birinci sendrom), ikinci r satırın alınan veri vektörü ile çarpımından $X_1^3 + X_2^3$ (Üçüncü sendrom) bulunur. Hata yerlerinin küplerinin alınma nedeni $X_1 + X_2'$ den farklı bir eşitlik elde etmektir. Hata yerlerinin karesini alıp topladığımızda elde edilen sonuç ilk denklemin karesidir. Dolayısıyla farklı bir ikinci denklem elde etmemiş oluruz.

$$X_1 + X_2 = S_1 \quad : \text{Birinci Sendrom} \quad (2.4)$$

$$X_1^2 + X_2^2 = S_2 \quad : \text{İkinci Sendrom} \quad (2.5)$$

$$(X_1 + X_2)^2 = X_1^2 + 2X_1X_2 + X_2^2 = X_1^2 + X_2^2 = S_1^2 = S_2$$

$$X_1^3 + X_2^3 = S_3 \quad : \text{Üçüncü Sendrom} \quad (2.6)$$

$$\begin{aligned} X_1^3 + X_2^3 &= (X_1 + X_2)(X_1^2 - X_1X_2 + X_2^2) \\ &= S_1(X_1X_2 - S_1^2) \Rightarrow \end{aligned} \quad (2.7)$$

$$X_1X_2 = S_1^2 + S_3 / S_1 \quad (2.8)$$

Eğer $S_1 \neq 0$ ise X_1 ve X_2 tarafından $X(S_1 + X) = S_1^2 + S_3 / S_1$ yada $1 + S_1X^{-1} + (S_1^2 + S_3 / S_1)X^{-2}$ eşitliğinin sağlanması gerekir. Bu eşitlik ikinci dereceden bir bilinmeyenli denklemdir ve mod $M(X)$ ' e göre çözümü mümkündür [3].

İkili sayılarla toplama ve çıkarma, sayıların xor işlemidir. Toplama sonucu bit sayısı sabit kalır. Ancak çarpma ve bölme işleminde sonuçta bit sayısının değişmemesi için indirgenemez bir polinoma ($M(X)$ ' e) göre mod alınır.

2.1 Reed Solomon Kodlar

İkili BCH kodlar, 1959 yılında Hocquenghem ve 1960 yılında Hocquenghem' den bağımsız olarak çalışan Bose ve Chaudhuri tarafından bulundu. Bu kodların periyodik yapısı ise Peterson tarafından 1960 yılında ispatlandı. BCH kodların ikili olmayan kodlara genellemesi ise 1961 yılında Gorenstein ve Zierler tarafından gerçekleştirildi. Yine 1960 yılında MIT laboratuvarlarında Hocquenghem, Bose ve Chaudhuri' den bağımsız olarak çalışan Reed ve Solomon tarafından ikili olmayan semboller için bir kodlama yöntemi geliştirildi ve bu yöntemle oluşturulan kodlara yaratıcılarının adları verilerek Reed-Solomon kodlar olarak adlandırıldı [15].

RS kodlar ikili olmayan BCH kodların alt sınıflarından bir tanesidir. Ard arda gelen bit hatalarını ve silinmiş bitleri (erasure: sinyal seviyesine göre '1' ve '0' olduğuna karar verilemeyen bitler) düzeltmek için kullanılırlar. İletimi gerçekleştirilecek mesaj bitlerinin belirli sayıda ve sembol olarak adlandırılan gruplar şeklinde alınıp, bu sembollere kontrol sembollerinin eklenmesiyle oluşturulurlar. Genelde mesaj sembollerinin sayısı k , kodsözcüğündeki sembol sayısı n , her semboldeki bit sayısı m , bu şekilde tanımlanan kodlar da $RS(n,k)$ ile gösterilir. RS kodlama m -bit sembolde ard arda gelen bitlerde oluşan hataların düzeltilmesinde etkili bir yöntem olup, çoğunlukla da bu amaç için kullanılır. Ancak tek bit hatalarını da düzeltilebilirler. Hatalı bir sembolün düzeltilmesiyle o semboldeki bütün hatalı olarak iletilen bitler (en az 1-bit, en çok m -bit) düzeltilmiş olur.

Semboller genel olarak 8-bit gruplardan oluşur. $RS(225,239)$ ITU-T (Uluslararası Teleiletişim Ortaklığı) standartlarından bir tanesidir [17]. $RS(255,223)$ uydu iletişiminde kullanılan bir NASA standardıdır [18].

RS kodlar, mesaj sembollerinin bir kısmı sıfır yapılarak kısaltılabilir. Sıfır olan semboller alıcıya iletilmez, bu durumda alıcı eksik semboller için kendisi sıfır ekler. Örneğin $RS(255,223)$ kodu $RS(200,168)$ koduna kısaltılabilir. Verici 168 sembole 55 sıfır sembolü ekler ve $(225,223)$ kodunu yaratır, ancak sadece 168 mesaj sembolü ve 32 kontrol sembolünü alıcıya iletir.

2.1.2 Reed Solomon kodların kullanım alanları

Sayısal sistemlerde yaygın olarak kullanılan bir kodlama yöntemi olan Reed Solomon kodların başlıca kullanım alanları aşağıdaki gibi sıralanabilir.

- Uydu İletişimi
- Kablosuz İletişim
- Yüksek hızlı modemler (ADSL, XDSL)
- Sayısal Televizyonlar
- Bellekler

RS kodlar hücresel sayısal paket veri (CDPD) hizmetlerinde kullanıcı verilerini korumak için, sayısal abone döngüsü (xDSL) hizmetlerinde ise verileri dürtü (impulse) gürütüden koruma amacıyla kullanılırlar. Ses CD' lerinde (yoğun teker) kullanılan bir çift çapraz serpiştirmeli RS kod ise okuma sürecinde oluşan hataların düzeltiliminde kullanılır [1,13]. Ayrıca RS kodlar kablosuz iletişim sistemlerinde ard arda bağlantılı kodlama sistemlerinin bir parçası olarak kıvrım kodlarla birlikte yaygın olarak kullanılmaktadırlar [7, 10].

2.1.3 Galois (Sonlu) alanlar

Reed Solomon kodsözcüklerindeki semboller, sonlu alanlara ait sembollerdir ve bu sembollerle yapılan işlemler sonlu alanlar için geçerli aritmetik işlemlerdir.

Galois yada sonlu alanlar (GF), sıfırlar ve birlerden oluşmuş elemanlar kümesidir. Çarpma, toplama ve bölme ($*$, $+$, $/$) sonlu alanlarda tanımlı işlemlerdir. Sıfır elemanı dışında bütün elemanların tersi vardır. Sonlu alanlar, çarpma ve toplamaya göre kapalıdır, bir alandaki elemanlarla yapılan işlemlerin yine o alanda yer alan tek bir sonucu vardır.

n elemanlı sonlu alan $GF(n)$ şeklinde gösterilir. Bir sonlu alandaki eleman sayısı n alanın derecesini verir, bir elemanın değerini 1 yapan en küçük kuvveti ise n elemanın derecesidir.

Teorem : Derecesi q olan bir sonlu alanın, derecesi $q-1$ olan ve üsleri n alanın bütün elemanlarını veren bir temel elemanı (α) vardır [3].

p bir asal sayı ve m bir tamsayı iken, $GF(p^m)$ alanını oluşturan, sadeleşemeyen ve m nci dereceden bir en küçük polinom vardır. Her $\alpha \in GF(p^m)$ için $M_\alpha(\alpha)=0$ olan bir $M_\alpha(X)$ polinomu bulunur. En küçük polinomun kökü aynı zamanda temel elemandır ve sonlu alanı oluşturmamızı sağlar. Her elemanın tek bir en küçük polinomu vardır. Aynı en küçük polinoma sahip elemanlar eşlenik olarak adlandırılır.

p asal sayısı için $\{0,1,...,p-1\}$ kümesi bir sonlu alan oluşturur. Bu alandaki toplama ve çarpma işlemleri mod p ' ye göre yapılır. p asal sayı değilse mod p aritmetiği alanlar için geçerliliğini yitirir. Bunu ispatlamak için çelişkiden yararlanılır. Örneğin $p=4$ için $\{0,1,2,3\}$ kümesini ele alalım. $2a \equiv 1 \pmod{4}$ ise, bu koşulu sağlayan a sayısı kümede yer almadığı için a 'nın çarpmaya göre tersi yoktur. Bu da sonlu alan tanımıyla çelişkilidir [4].

p bir asal sayı, m de p 'nin her hangi bir kuvveti olduğunda $GF(p^q)$ kümesindeki elemanlar Reed-Solomon kodlarını oluştururlar.

$GF(p^m)$ alanının temel elemanı α 'nın, 0 'dan m 'ye kadar bütün dereceleri alanın bir elemanına karşılık gelir, aynı zamanda α en küçük polinomun köküdür.

Örnek : $p = 2$, $m=4$ için $GF(2^4)$ 'ü oluşturalım.

$m=4$, minimal polinom = X^4+X+1 , ilkel eleman = α

$$\alpha^4 + \alpha + 1 = 0 \Rightarrow \alpha^4 = \alpha + 1$$

küme elemanları :

$$\alpha^0 = 0001$$

$$\alpha^8 = \alpha^7 \alpha = 0101$$

$$\alpha^1 = 0010$$

$$\alpha^9 = \alpha^8 \alpha = 1010$$

$$\alpha^2 = 0100$$

$$\alpha^{10} = \alpha^9 \alpha = 0111$$

$$\alpha^3 = 1000$$

$$\alpha^{11} = \alpha^{10} \alpha = 1110$$

$$\alpha^4 = \alpha + 1 = 0001$$

$$\alpha^{12} = \alpha^{11} \alpha = 1111$$

$$\alpha^5 = \alpha^4 \alpha = (\alpha + 1) \alpha = 0110$$

$$\alpha^{13} = \alpha^{12} \alpha = 1101$$

$$\alpha^6 = \alpha^5 \alpha = 1100$$

$$\alpha^{14} = \alpha^{13} \alpha = 1001$$

$$\alpha^7 = \alpha^6 \alpha = 1011$$

$$\alpha^{15} = \alpha^0 = 0001$$

Galois alanda iki sayının toplamı ya da farkı sayıların xor işlemidir. Çarpma ise sayıların üslerinin mod 15'e göre toplanması ile bulunur.

$$\text{Örnek : } \alpha^6 \alpha^{12} = \alpha^{(6+12) \bmod 15} = \alpha^3$$

Bu sayede m-bit sembollerle yapılan çarpma, bölme, toplama işlemleri sonucunda yine m-bit tek bir sembol bulunur.

3. REED SOLOMON KODLAYICI

Kodlama işleminin temeli, iletilecek mesaja kontrol sembollerinin eklenmesidir. Kontrol sembollerinin bulunması için bir dizi aritmetik işlem yapılması gerekir. RS kodlarda yapılan bu işlemler sonlu alanlar için geçerli olan işlemlerdir.

Reed Solomon kodlama işleminin matematiksel anlatımında, mesaj ve kontrol sembolleri polinomlarla gösterilir. Her bir sembol m-bit' den oluşmaktadır.

RS(n,k) için :

$$n = 2^m - 1, 2t = n - k,$$

İletilecek veri = $D(X) = d_{k-1}X^{k-1} + d_{k-2}X^{k-2} + \dots + d_1X + d_0$ olarak tanımlanır.

Polinomdaki X ' lerin katsayıları sembol değerlerini, üsleri ise sembol yerlerini belirtir.

Kontrol sembollerinin oluşturulması için de ayrı bir üretici polinom, $G(X)$ kullanılır.

$$G(X) = \prod_{i=0}^{2t-1} (X - \alpha^i) \quad \alpha : GF(n)' \text{ in temel elemanıdır.} \quad (3.1)$$

Kontrol sembolleri $X^{n-k} D(X)$ ' in $G(X)$ ile bölümünden kalan polinomun, $(P(X))$ katsayılarıdır.

$$X^{n-k} D(X) = G(X)Q(X) + P(X) \quad (3.2)$$

$$P(X) = p_{n-k-1}X^{n-k-1} + p_{n-k-2}X^{n-k-2} + \dots + p_1X + p_0 \quad (3.3)$$

Son olarak $P(X)$ ' in $X^{n-k} D(X)$ ' den çıkarılmasıyla kodlama işlemi tamamlanır. Böylece sonuçta elde edilen polinom üretici polinomun bir katı olmuş olur.

$$G(X)Q(X) = X^{n-k} D(X) - P(X) \quad (3.4)$$

$$= d_{k-1}X^{n-1} + d_{k-2}X^{n-2} + \dots + d_1X^{n-k+1} + d_0X^{n-k} + p_{n-k-1}X^{n-k-1} + p_{n-k-2}X^{n-k-2} + \dots + p_1X + p_0$$

$$= c_{n-k}X^{n-k} + c_{n-k-1}X^{n-k-1} + \dots + c_1X + c_0$$

$$= C(X)$$

$C(X)$, mesaj sembollerine kontrol sembollerinin eklenmesiyle oluşan kodlanmış veriyi (kodsözcüğünü) belirten polinomdur.

3.1 Reed Solomon Kodlayıcı Tasarımı

Kodlayıcı için data polinomunun üretici polinoma bölünmesini sağlayan bir donanım hazırlanması gerekir. Polinomların bölme işlemi doğrusal geri beslemeli kaydıran yazmaç (LFSR) ile gerçekleştirilir. Ayrıca, bölme sırasında yapılacak çarpma ve toplama işlemlerini Galois alan aritmetiğine uygun olarak gerçekleştirecek devreler tasarlanması gereklidir.

Bu çalışmada kodlayıcı uygulaması gerçekleştirilirken karmaşıklığı azaltmak ve yapılan hataları daha rahat gözlemleyerek düzeltebilmek için RS kodlarda yaygın olarak kullanılan 8-bit semboller yerine 4-bit semboller için bir tasarım gerçekleştirildi. $m = 4$, $n=2^4-1=15$, $k=9$, $2t=15-9=6$ değerlerine sahip olan RS(15, 9) kodlayıcı oluşturuldu.

RS(15,9) için,

Üretici polinom :

$$G(X) = \prod_{i=0}^{2t-1} (X - \alpha^i) = (X - \alpha^0) (X - \alpha^1) (X - \alpha^2) (X - \alpha^3) (X - \alpha^4) (X - \alpha^5) \quad (3.1.1)$$

$$= X^6 + \alpha^{10}X^5 + \alpha^{14}X^4 + \alpha^4X^3 + \alpha^6X^2 + \alpha^9X + \alpha^6$$

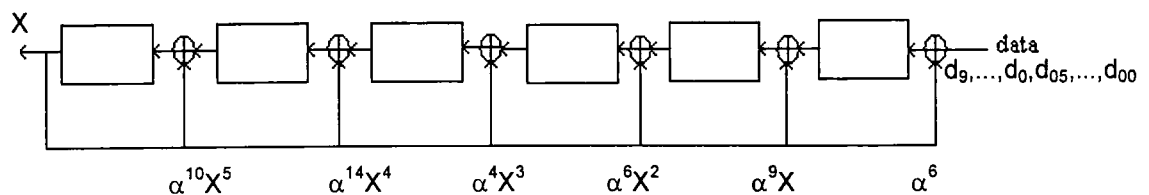
Data polinomu :

$$D(X) = d_8X^8 + d_7X^7 + d_6X^6 + \dots + d_1X + d_0 \quad (3.1.2)$$

$$X^{n-k}D(X) = d_8X^{14} + d_7X^{13} + d_6X^{12} + \dots + d_0X^6 + d_{05}X^5 + \dots + d_{00} \quad (3.1.3)$$

$$d_{05} = d_{04} = d_{03} = d_{02} = d_{01} = d_{00} = "0000"$$

Veriye eklenecek $n-k$ sembolü bulmak için $X^{n-k}D(X)$ üretici polinoma bölünür. Bölme işlemini gerçekleştiren geri beslemeli kaydıran yazmaç şekil 3.1.1 'de gösterilmiştir.



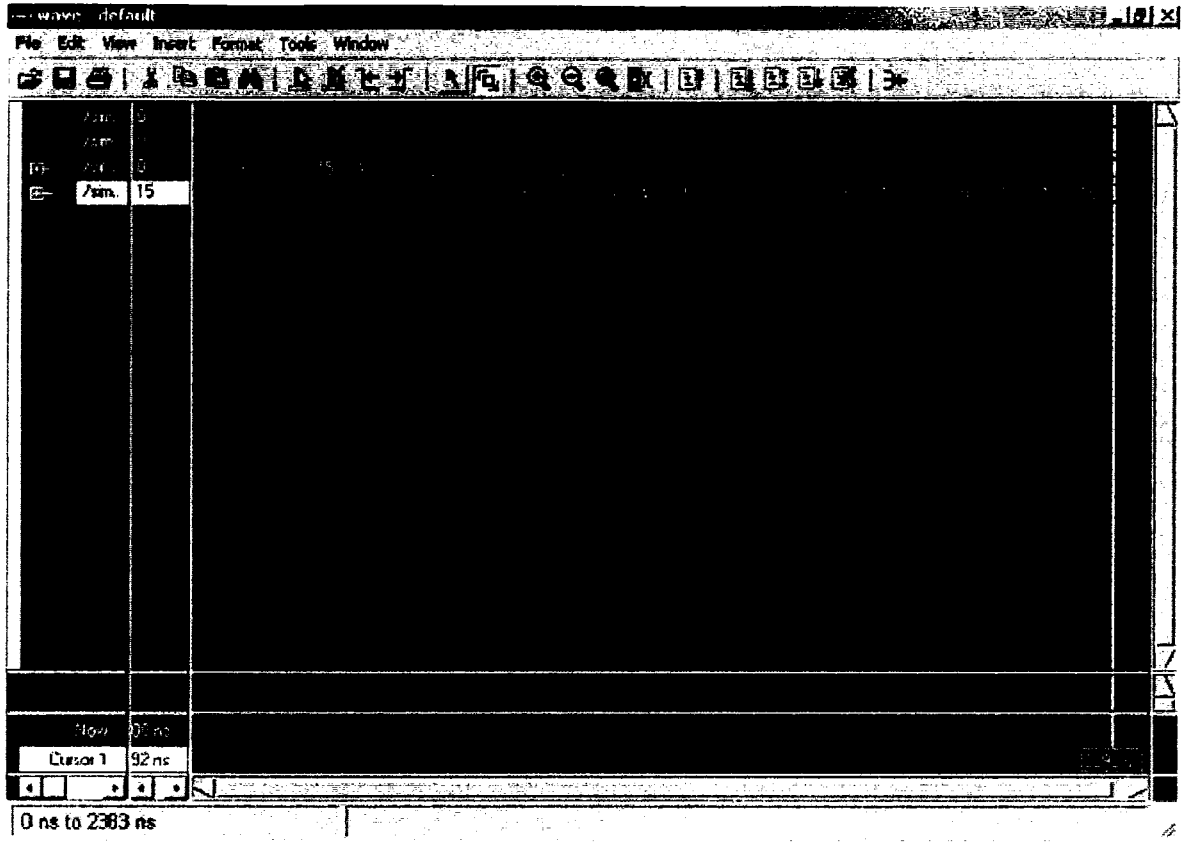
Şekil 3.1.1 Data polinomu üretici polinoma bölen devre

Son data (d_{00}) da alındıktan sonra kaydıran yazmaçtaki değerler kontrol sembollerine eşit olur. Bu sembollerin data polinomuna eklenmesiyle kodsözcüğü polinomu, $C(X)$ oluşur.

$$C(X) = d_9X^{14} + d_8X^{13} + \dots + d_6X^6 + p_5X^5 + p_4X^4 + \dots + p_0$$

Şekil 3.1.1' deki devrenin tasarımı için VHDL kullanıldı. Davranışsal yapıda olan bir VHDL kodu yazıldı. Kodlayıcı için yazılan VHDL kodu Ek-2' de kodlayıcı ve kodçözücü için kullanılan fonksiyon ve veri türlerinin bulunduğu paket Ek-1' de yer almaktadır. 4-bit sembollerin çarpma ve toplama işlemi Galois alan aritmetiğine göre yapıldı. Toplama için xor işlemi kullanıldı. Çarpma işlemi yapılırken önce sembollerin temel elemanın (α ' nın) kaçınıcı kuvveti olduğu bulundu, daha sonra çarpılacak sembollerin üsleri mod 15' e göre toplandı. Ancak sentezlenebilir mod işlemlerinin 2' nin çift kuvvetleri için geçerli olması nedeniyle, mod 15 işlemini yapan bir devre sentezlenemedi. Bu durum mod işlemi yerine, üsler toplamı 15' den büyük ise sayıdan 15 çıkarılmasını sağlayan döngü ve koşul komutları kullanılarak çözüldü. Üslü biçime dönüştürülen semboller toplamak için tekrar 4-bit sayı karşılığı yazılması gerektiği, bunun için -1' den 14' e kadar α ' nın kuvvetlerinin indeks değerleri olduğu, indekse karşılık gelen değerlerin ise sayının 4-bit değeri olduğu bir dizi tanımlandı.

Tasarım tamamlandıktan sonra kodlayıcının çalışması simülasyon yapılarak test edildi. $D=(1001, 1001, 1111, 0100, 0010, 0001, 1000, 0110, 1000)$ vektörü için kodlayıcı simülasyon sonuçları şekil 3.1.2' de yer almaktadır. Simülasyon sonucundan da görülebileceği gibi kodlama sonucu, $p_5=1101$, $p_4=0110$, $p_3=0000$, $p_2=1010$, $p_1=1101$, $p_0=1111$ olacak şekilde kontrol sembollerini elde edilmiştir.



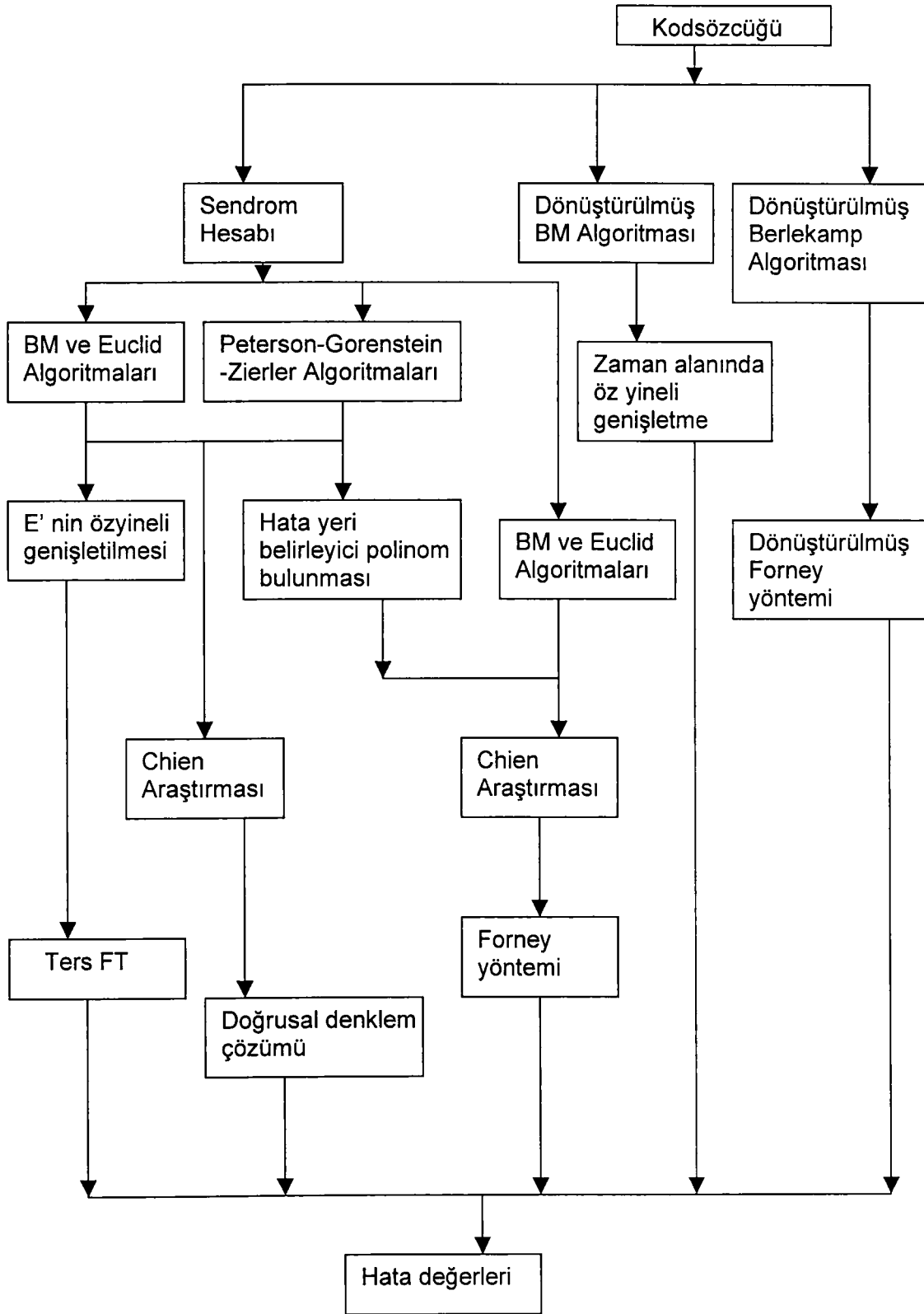
Şekil 3.1.2 Kodlayıcı bloğu simülasyon sonucu

4. REED SOLOMON KODÇÖZÜCÜ

Vericide kodlanan bilgi, kanaldan geçerek alıcıya gelir ve kodçözücüye ulaşır. Kodçözücüye gelen semboller, $R(X)$ polinomunun katsayıları ile gösterilir. $R(X)$, kodsözcüğü polinomu ve kanaldaki gürültüden kaynaklanan hataların oluşturduğu $E(X)$ polinomunun toplamıdır.

$$R(X)=C(X)+E(X)$$

Kodçözücüde yapılan işlem hata polinomunun bulunmasıdır. Hata polinomunu bulabilmek için çeşitli algoritmalar geliştirilmiştir. Bu algoritmalar ve farklı birleşimleri Şekil 4.1' de [1] gösterilmiştir.



Şekil 4.1 RS kodçözümleme teknikleri

4.1 Sendrom Değerlerinin Bulunması

Kodsözcüğü polinomu üretici polinomun bir katıdır. Bu durumda $G(X)$ ' in bütün kökleri aynı zamanda $C(X)$ ' inde kökleridir. Alınan polinom bir kodsözcüğü ise, yani kodsözcüğüne kanalda hata eklenmediyse $G(X)$ ' in kökleri $R(X)$ ' in de kökleri olacaktır; yani X yerine konulduğunda $R(X)=0$ koşulunu sağlaması gerekir.

Üretici polinomun her bir kökü için $R(X)$ ' in aldığı değer sendrom olarak adlandırılır. Bütün sendromlar sıfıra eşit ise $R(X)$ kodlanmış polinomun kendisidir, yani $E(X)=0$ ' dır. Aksi takdirde sendromlardan yararlanarak hata değerlerine ve yerlerine ulaşılır. RS(15.9) kodunda sendromlara ulaşmak için yapılan işlemler aşağıda gösterilmiştir.

$$R(X) = C(X) + E(X)$$

$$E(X) = Y_{v1}X^{v1} + Y_{v2}X^{v2} + \dots + Y_{ve}X^{ve} \quad (4.1.1)$$

e : hata sayısı

Y : hata değeri

X : hata yeri

$$0 \leq v_i \leq 14, 1 \leq i \leq e, i, e \in \mathbb{Z}$$

$$E(X) = \sum_{i=1}^e Y_i X_i^j \quad (4.1.2)$$

$$Y_i = Y_{v_i}$$

$$X_i = \alpha_{v_i}$$

Sendrom Değerleri :

$$S_1 = R(\alpha) = \cancel{C(\alpha)}^0 + E(\alpha) = E(\alpha) \quad (4.1.3)$$

$$S_2 = R(\alpha^2) = \cancel{C(\alpha^2)}^0 + E(\alpha^2) = E(\alpha^2) \quad (4.1.4)$$

$$S_{2^t} = R(\alpha^{2^t}) = \cancel{C(\alpha^{2^t})}^0 + E(\alpha^{2^t}) = E(\alpha^{2^t}) \quad (4.1.5)$$

Eşitliklerden görüldüğü gibi sendromlar, üretici polinomun kökleri için hata polinomunun aldığı değerlerdir.

Sendromların oluřturduđu polinom :

$$S(Z) = \sum_{i=1}^{2t} S_i Z^i \quad (4.1.6)$$

4.1.2 nci eřitliđe gre:

$$\begin{aligned} S_j &= \sum_{i=1}^e Y_i X_i^j \\ S_j &= R(\alpha^j) = C(\alpha^j) + E(\alpha^j) \\ &= 0 + E(\alpha^j) \\ &= Y_{v1} \alpha^{j \cdot v1} + Y_{v2} \alpha^{j \cdot v2} + \dots + Y_{ve} \alpha^{j \cdot ve} \end{aligned} \quad (4.1.7)$$

4.2 Hata Yeri Belirleyici, Hata Değeri Belirleyici Polinom ve Anahtar Eřitlik

Sendrom deđerleri bulunduktan sonra kodzmlleme iřleminde yapılması gereken bu deđerlerden hata yerleri ve deđerlerine ulařmaktır. Bunun iin sendromlar ile hata yeri ve deđerleri arasında bir bađıntının olması gerekir. Bu bađıntıyı oluřturabilmek iin hata yeri belirleyici polinom denilen bir polinom tanımlanmıřtır. Bu polinomun zelliđi kklerinin hata yerlerinin tersi olmasıdır. Hata yeri belirleyici polinom $\sigma(Z)$ ile gsterilir.

Sendrom polinomu ve hata yeri belirleyici polinom arasında yapılan bir dizi matematiksel iřlem sonucunda hata deđerleri belirleyici polinom elde edilir ve bu  polinom arasındaki kodzme iřlemini sađlayacak bađıntı, anahtar eřitlik olarak adlandırılır. Kodzme iřleminin en karmařık ve zaman alıcı sreci de anahtar eřitliđin zlerek hata yeri ve hata deđerleri belirleyici polinomun katsayılarının bulunmasıdır.

4.2.1 Anahtar eşitliğin oluşturulması

Hata yeri belirleyici polinom :

$$\sigma(Z) = \prod_{i=1}^e (1 - X_i Z) = 1 + \sum_{j=1}^e \sigma_j Z^j \quad (4.2.1.1)$$

σ_0 daima 1'e eşittir.

Sendromların oluşturduğu polinom, $S(Z)$:

$$S(Z) = \sum_{j=1}^{\infty} S_j Z^j = \sum_{j=1}^{\infty} \sum_{i=1}^e Y_i X_i^j Z^j = \sum_{i=1}^e Y_i X_i Z / (1 - X_i Z) \quad (4.2.2.2)$$

$S(Z)$ ve $\sigma(Z)$ çarpılır :

$$S(Z) \cdot \sigma(Z) = \sum_{i=1}^e Y_i X_i Z / (1 - X_i Z) \prod_{j=1}^e (1 - X_j Z) = \sum_{i=1}^e Y_i X_i Z \prod_{j \neq i} (1 - X_j Z) \quad (4.2.2.3)$$

Çarpım değerine $\sigma(Z)$ eklenir :

$$S(Z) \cdot \sigma(Z) + \sigma(Z) = [1 + S(Z)] \cdot \sigma(Z) = W(Z) = \sum_{k=0}^e w_k Z^k \quad (4.2.2.4)$$

$W(Z)$ hata değeri belirleyici polinom olarak tanımlanır ve Forney algoritmasında hata değerlerinin bulunması için kullanılır.

Sonuçta elde ettiğimiz $[1 + S(Z)] \sigma(Z) = W(Z)$ eşitliğini çözebilmemiz sadece $(2t)'$ ye kadar olan sendrom değerleri bilindiğinden mümkün değildir. Bulabileceğimiz $\sigma(Z)$ ve $W(Z)$ $[1 + S(Z)] \sigma(Z) \equiv W(Z) \pmod{2^{t+1}}$ eşitliğinde yer alır. Bu eşitlik anahtar eşitlik olarak adlandırılır.

Şekil 4.1' de gösterildiği gibi sendromların hesaplandığı algoritmalarda, ikinci adım anahtar eşitliğin çözülerek hata yeri belirleyici polinomun bulunmasıdır. Bunun için daha önce belirtildiği gibi Peterson, Gorenstein, Zierler algoritmaları ile BM ve Euclid algoritmaları kullanılmaktadır [1].

Hata yeri belirleyici polinom elde edildikten sonra hata değerleri zaman veya frekans alanında bulunabilir. Sendromların bulunmasıyla elde edilen $2t$ eşitlik kodçözücüye gelen bilginin Fourier dönüşümü (FT) olarak göz önüne alınır [14].

Frekans bölgesinde hata dönüşümü, yani hata yerlerinin ve değerlerinin bulunması sendromlar ve hata vektörünün özyineli olarak genişletilip daha sonra ters Fourier dönüşümü alınmasıyla bulunur.

Zaman alanında ise hata yeri belirleyici polinomun kökleri, Chien araştırma yöntemiyle bulunmaktadır. Hata yeri belirleyici polinomun bulunmasının ardından hata değerleri, hata yerleri bulunduktan sonra doğrusal hale gelen eşitliklerin (4.1.3 - 4.1.5) doğrudan çözümü ya da Forney yöntemi [4] ile bulunur. Forney yönteminin kullanılabilmesi için öncelikle hata değeri belirleyici polinomun bulunması gereklidir.

BM ve Euclid algoritmalarında ek bilgiler kullanılarak bu algoritmaların hata yeri belirleyici polinomla beraber hata değeri belirleyici polinomu bulmaları sağlanabilir ya da bu polinom yine BM ve Euclid algoritmaları kullanılarak hata yeri belirleyici polinom bulunduktan sonra anahtar eşitlikten elde edilebilir.

Sendromların hesaplanmadığı algoritmalar da ise bütün değişkenler zaman alanında yer almaktadır. Bu değişkenlerin zaman alanındaki değerlerinin bulunabilmesi için algoritmadaki tüm polinomların ters Fourier dönüşümü alınır [14].

Kodçözümleme işleminde bahsedilen algoritmalarından hangisi kullanılırsa kullanılsın, frekans alanında sendrom değerine karşılık gelen $2t$ eşitliğin veya zaman alanında gerçekleşen çözümleme işleminin belli bir hata sayısına kadar tek bir çözümü vardır. Bu sayı t' ye yani $(n-k)/2'$ ye eşittir. Hata sayısı t değerini geçtiği zaman birden fazla çözüm bulunur [12,14], bu durumda kod çözme işlemi başarısız olur.

Bu tez çalışması aşağıdaki kodçözümleme yöntemlerine dayanmaktadır.

1-Sendromların bulunması.

2-Hata yeri belirleyici ve hata değeri belirleyici polinomların katsayılarının bulunması.

3-Chien araştırma yöntemi ile hata yerlerinin bulunması, Forney algoritması ile hata değerlerinin hesaplanarak hatanın düzeltilmesi.

Bu teknikler, kodçözme işleminde yaygın olarak kullanılan en etkili tekniklerdir. Bu çalışmada anahtar eşitliğin çözümünde BM algoritması üzerine kurulu kodçözümler tasarımı gerçekleştirilmiştir. Öncelikle orjinal BM algoritması için, daha sonra orjinal algoritmanın bir türevi olan evirmesiz BM algoritması için, son olarak da evirmesiz BM algoritmasında yapılan değişiklikle elde edilen yeni algoritma için kodçözümler tasarımı gerçekleştirilmiştir.

4.2.2 Berlekamp-Massey algoritması

1967 yılında, Berlekamp ikili olmayan BCH kodlar ve Reed Solomon kodlar için kodçözme algoritmasını buldu. 1968 yılında da Massey, Berlekamp' in algoritmasının sendrom değerlerini oluşturan en küçük boyutlu doğrusal geri beslemeli kaydıran yazarı bulan algoritmaya eş olduğunu kanıtladı ve bu algoritma Berlekamp Massey algoritması olarak adlandırıldı [14,18].

BM algoritması döngülü bir algoritma olup, anahtar eşitliği 0' dan başlamak üzere $2t$ adımda çözer. Her adımda bir önceki adım için bulunan çözümden yola çıkılarak hata yeri ve değeri belirleyici polinomların yeni değerleri bulunur.

Bu algorithma aşağıda yer alan eşitlik (4.2.2.1) $k=0$ 'dan başlayarak ve $k=2t-1$ oluncaya kadar çözülür, $k=2t-1$ için elde edilen eşitlik anahtar eşitliğin kendisidir.

$$(1+S)\sigma^{(k)} \equiv W^{(k)} \text{ mod } z^{k+1} \quad (4.2.2.1)$$

Başlangıç değerleri, yani $k=0$ için çözüm bellidir. $\sigma_0^{(0)} = 1$ ve $w_0^{(0)} = 1$ 'dir. Bundan sonraki her adımda k için bulunan σ ve W polinomlarının $k+1$ için de eşitliği sağlayıp sağlamadığı kontrol edilir. Bu kontrolü yapabilmek için 4.2.2.2 eşitliğinde $\Delta^{(k)}$ değeri bulunur.

$$(1+S)\sigma^{(k)} \equiv W^{(k)} + \Delta^{(k)} z^{k+1} \text{ mod } z^{k+2} \quad (4.2.2.2)$$

$\Delta^{(k)}$, $(1+S)\sigma^{(k)}$ çarpımında z^{k+1} nci terimin katsayısıdır ve uyumsuzluk olarak adlandırılır. Eğer uyumsuzluk sıfır ise, polinomların yeni değerleri bir öncekine eşittir, yani $\sigma^{(k+1)} = \sigma^{(k)}$ ve $w^{(k+1)} = w^{(k)}$. Ancak uyumsuzluğun sıfır olmadığı durumlarda $\sigma^{(k+1)}$ ve $w^{(k+1)}$ i bulabilmek için yardımcı polinomlar tanımlanması gerekir. Bu yardımcı polinomlar $\tau^{(k)}$ ve $\gamma^{(k)}$ olarak tanımlanmıştır ve aralarındaki ilişki 4.2.2.3' de verilen bağıntı olarak kabul edilir.

$$(1+S) \tau^{(k)} \equiv \gamma^{(k)} + z^k \mod z^{k+1} \quad (4.2.2.3)$$

Hata yeri belirleyici ve hata değeri belirleyici polinomlar bulunurken amaç, mümkün olan en küçük dereceli polinomları bulabilmektir. Çünkü iletişimin gerçekleştiği kanalın iletişime uygun olduğu ve kanalda gürültünün iletişimi imkansız hale getirecek bir etkisi olmadığı kabul edilir. Aynı durum γ ve τ yardımcı polinomları için de geçerlidir. Olabilecek en küçük dereceli yardımcı polinomlar elde edilemeye çalışılır.

σ , W ve τ , γ arasındaki ilişki ise 4.2.2.4 ve 4.2.2.5' de gösterilmiştir.

$$\sigma^{(k+1)} = \sigma^{(k)} - \Delta^{(k)} z \tau^{(k)} \quad (4.2.2.4)$$

$$W^{(k+1)} = W^{(k)} - \Delta^{(k)} z \gamma^{(k)} \quad (4.2.2.5)$$

4.2.2.4 ncü ve 4.2.2.5 nci eşitlikler, aşağıda gösterildiği gibi $(1+S) \sigma^{(k)}$ çarpımından $(1+S) z \Delta^{(k)} \tau^{(k)}$ çıkarıldığında elde edilir.

$$(1+S) \sigma^{(k)} \equiv W^{(k)} + \Delta^{(k)} z^{k+1} \mod z^{k+2}$$

$$(1+S) z \Delta^{(k)} \tau^{(k)} \equiv W^{(k)} + \Delta^{(k)} z \gamma^{(k)} + \Delta^{(k)} z^{k+1} \mod z^{k+2}$$

$$(1+S) \underbrace{(\sigma^{(k)} - \Delta^{(k)} z \tau^{(k)})}_{\sigma^{(k+1)}} \equiv \underbrace{W^{(k)} - \Delta^{(k)} z \gamma^{(k)}}_{W^{(k+1)}} \mod z^{k+2}$$

σ ve W polinomlarının yeni değerlerinin bulunabilmesi için $\tau^{(k+1)}$ ve $\gamma^{(k+1)}$ in bulunması gerekir. $\tau^{(k+1)}$ ve $\gamma^{(k+1)}$ bulunurken alabilecekleri iki farklı değere göre karar verilir. Bu değerler 4.2.2.6 ve 4.2.2.7' de gösterilmiştir.

$$\tau^{(k+1)} = z \tau^{(k)} \quad \text{ve} \quad \gamma^{(k+1)} = z \gamma^{(k)} \quad (4.2.2.6) \text{ yada}$$

$$\tau^{(k+1)} = \sigma^{(k)} / \Delta^{(k)} \quad \text{ve} \quad \gamma^{(k+1)} = W^{(k)} / \Delta^{(k)} \quad (4.2.2.7)$$

Her iki seçeneğinde $(1+S) \tau^{(k+1)} \equiv \gamma^{(k+1)} + z^{k+1} \mod z^{k+2}$ denkleğini sağladığı görülmektedir. Hangi eşitliğin kullanılacağına karar verirken öncelikle $\Delta^{(k)}$ değerinin

sıfır olup olmadığı kontrol edilir. Eğer $\Delta^{(k)} = 0$ ise, 4.2.2.7 nci eşitlik tanımsız olacağından γ ve τ ' nin yeni değerleri 4.2.2.6 ncı eşitlikteki değerlerdir. $\Delta^{(k)}$ ' nin sıfır olmadığı durumlarda en küçük dereceli polinomları veren eşitliğe göre $(k+1)$ nci değerler bulunur.

$\sigma^{(k+1)}$, $W^{(k+1)}$, $\tau^{(k+1)}$, $\gamma^{(k+1)}$ 'nin derecelerinin aldığı değerler 4.2.2.8, 4.2.2.9, 4.2.2.10 ve 4.2.2.11' de verilmiştir.

$$\deg \sigma^{(k+1)} = \begin{cases} \deg \sigma^{(k)} & \text{eğer } \Delta^{(k)} = 0 \text{ veya } \deg \sigma^{(k)} > 1 + \deg \tau^{(k)} \\ 1 + \deg \tau^{(k)} & \text{eğer } \Delta^{(k)} \neq 0 \text{ ve } 1 + \deg \tau^{(k)} > \deg \sigma^{(k)} \end{cases} \quad (4.2.2.8)$$

Dikkat edilecek olursa, 4.2.2.4 ncü eşitliğe göre, $\deg \sigma^{(k)} = 1 + \deg \tau^{(k)}$ ve $\sigma^{(k)}$ ile $\Delta^{(k)} \tau^{(k)}$ ' nin en büyük dereceli terimlerinin katsayıları eşit olduğunda, iki terim birbirini yok edeceğinden $\sigma^{(k+1)}$ ' in derecesi olması gerekenden küçük bir değer alır.

$$\deg \tau^{(k+1)} = \begin{cases} \deg \sigma^{(k)} & \text{eğer 6 ncı eşitlik kullanılırsa} \\ 1 + \deg \tau^{(k)} & \text{eğer 5 nci eşitlik kullanılırsa} \end{cases} \quad (4.2.2.9)$$

$$\deg W^{(k+1)} = \begin{cases} \deg W^{(k)} & \text{eğer } \Delta^{(k)} = 0 \text{ veya } \deg W^{(k)} > 1 + \deg \gamma^{(k)} \\ 1 + \deg \tau^{(k)} & \text{eğer } \Delta^{(k)} \neq 0 \text{ ve } 1 + \deg \gamma^{(k)} > \deg W^{(k)} \end{cases} \quad (4.2.2.10)$$

σ ' nin derecesinde olduğu gibi, W polinomunun da derecesi 4.2.2.5 nci eşitliğe göre $\deg W^{(k)} = 1 + \deg \gamma^{(k)}$ ve $W^{(k)}$ ile $\Delta^{(k)} \gamma^{(k)}$ ' nin en büyük dereceli terimlerinin katsayıları eşit olduğunda iki terim birbirini götüreceğinden $\sigma^{(k+1)}$ ' in derecesi olması gerekenden küçük bir değer alır.

$$\deg \gamma^{(k+1)} = \begin{cases} \deg W^{(k)} & \text{eğer 6 ncı eşitlik kullanılırsa} \\ 1 + \deg \gamma^{(k)} & \text{eğer 5 nci eşitlik kullanılırsa} \end{cases} \quad (4.2.2.11)$$

σ ve W polinomlarının derecelerinin uyumsuzluğun sıfır olmadığı durumlarda gerçek değerlerinden küçük değerler almasını önlemek için σ , W ve yardımcı polinomların dereceleri gerçek değerlerine göre değil, sınır değerlere göre

tanımlanır. Bu sınırları belirlemek için $D(k)$ ve $B(k)$ fonksiyonları tanımlanmıştır [3]. Bu fonksiyonların değerleri 4.2.2.12-16' da gösterilmiştir.

$D(k)$ fonksiyonu :

$$\deg \sigma^{(k)} \leq D(k) \quad (4.2.2.12)$$

$$\deg \tau^{(k)} \leq k-D(k) \quad (4.2.2.13)$$

4.2.2.10, 4.2.2.12 ve 4.2.2.13 ncü eşitliklere göre,

$$D(k+1) = \begin{cases} D(k) & \text{eğer } \Delta^{(k)} = 0 \text{ veya } D(k) \geq (k+1) / 2 \\ k+1-D(k) & \text{eğer } \Delta^{(k)} \neq 0 \text{ ve } D(k) \leq (k+1) / 2 \end{cases} \quad (4.2.2.14)$$

$B(k)$ fonksiyonu :

$$\deg w^k \leq D(k)-B(k) \quad (4.2.2.15)$$

$$\deg \gamma^k \leq k-D(k)-[1-B(k)] \quad (4.2.2.16)$$

$$B(k+1) = \begin{cases} B(k) & \text{5 nci eşitlik kullanılırsa} \\ 1-B(k) & \text{6 ncü eşitlik kullanılırsa} \end{cases}$$

$D(k)$, $B(k)$ ve diğer polinomlar arasındaki ilişki oldukça karmaşık olup, ispatı [2]' de verilmiştir.

Sonuçta elde edilen denklemlerden Berlekamp kodçözme algoritmasına [3] ulaşmıştır. Berlekamp' in algoritmasında $k=0$ ' dan $k=2t-1$ oluncaya kadar aşağıdaki adımlar tekrar edilir.

$k=0$ için başlangıç değerleri : $\sigma^{(k)} = 1$, $\tau^{(0)}=1$, $W^{(0)}=1$, $\gamma^{(0)}=0$, $D(0)=0$, $B(0)=0$

1.adım : Uyumsuzluk değeri hesaplanır.

$(1+S)\sigma^{(k)}$ çarpımında z^{k+1} nci terimin katsayısı $\Delta^{(k)}$ bulunur.

2.adım : Hata yeri belirleyici polinomun katsayıları güncellenir.

$$\sigma^{(k+1)} = \sigma^{(k)} - \Delta^{(k)} z \tau^{(k)} , W^{(k+1)} = W^{(k)} - \Delta^{(k)} z \gamma^{(k)}$$

3.adım : yardımcı polinomlar güncellenir.

Eğer $\Delta^{(k)} = 0$ veya $D(k) > (k+1)/2$ veya $\Delta^{(k)} \neq 0$ ve $D(k) = (k+1)/2$ ve $B(k) = 0$ ise

$$D(k+1) = D(k)$$

$$B(k+1) = B(k)$$

$$\tau^{(k+1)} = z\tau^{(k)}$$

$$\gamma^{(k+1)} = z\gamma^{(k)}$$

Eğer $\Delta^{(k)} \neq 0$ ve $D(k) < (k+1)/2$ veya $D(k) = (k+1)/2$ ve $B(k) = 1$ ise

$$D(k+1) = k+1 - D(k)$$

$$B(k+1) = 1 - B(k)$$

$$\tau^{(k+1)} = \sigma^{(k)} / \Delta^{(k)}$$

$$\gamma^{(k+1)} = W^{(k)} / \Delta^{(k)}$$

$k=2t-1$ olduğunda aradığımız hata yeri belirleyici ve hata değeri belirleyici polinomları elde etmiş oluruz.

Bundan sonraki aşama hata yeri belirleyici polinomun köklerinin yani hata yerlerinin terslerinin ve hata değerlerinin bulunmasıdır. Bu da Chien ve Forney tarafından bulunan yöntemlerle gerçekleştirilir.

4.3 Chien Araştırması ve Forney Algoritması

Hata yeri belirleyici polinomun katsayıları bulunduktan sonra, yapılacak işlem köklerinin bulunmasıdır. Bunun için Chien araştırma [1,19] yöntemi kullanılır.

Reed Solomon kod sembolleri Galois alanlara ait elemanlardı. Aynı şekilde bu sembollerin yerleri de Galois alan elemanları ile gösterilir. Her sembolün yeri Galois alanın temel elemanının bir kuvveti ile eşleştirilir.

Kodçözücüye gelen semboller sıralanırken $n-1$ ' den başlar ardışık olarak azalır ve 0 ' da son bulur. Bu sembollerin yerleri de Galois alanda sırası ile α^{n-1} ' den başlar, α^0 da son bulur.

Hata yerlerini bulmak için semboller kodçözücünden ayrılmadan önce her sembolün yerinin tersi için hata belirleyici polinomun değeri hesaplanır, değeri sıfır ise köklerden biri yani hata yerinin tersi bulunmuş olur. Buda kodçözücüye terk edecek olan sembolün hatalı olduğunu gösterir.

İlk gelen sembolün yeri α^{n-1} ' dir. $\sigma(z)$ polinomunun α^{n-1} e karşılık gelen kökü α^{n-1} ' in tersi yani α' dır. Vericiye gelen ilk sembol için $\sigma(\alpha)$ hesaplanır. İkinci sembolün yeri α^{n-2} , tersi α^2 ' dir. İkinci sembol için $\sigma(\alpha^2)$ hesaplanır. Üçüncü sembolde ise $\sigma(\alpha^3)$ 'ün hesaplanması gerekir. Yani her seferinde bir önceki çarpımda sigma fonksiyonunun terimlerinin aldığı değerler α ile çarpılır. Bu durumda $\sigma(z)$ ' in katsayılarının her saat çevriminde değiştiğini ancak z ' in hep α olarak kaldığını düşünebiliriz.

Kodçözücüde hata yerleri kontrol edilirken aynı zamanda Forney Algoritması [4] kullanılarak hata değerleri de hesaplanır. Bu algoritmaya göre Y_i hata değeri 4.3.1' de verilen eşitlikten bulunur.

$$Y_i = W(Z) / Z \sigma'(Z) \Big|_{Z=X_i^{-1}} \quad (4.3.1)$$

Forney eşitliğine ulaşılırken izlenen adımlar aşağıda gösterilmiştir.

σ' : σ 'nın birinci dereceden türevi

$$W(Z) = \sigma(Z) + S(Z) \cdot \sigma(Z) = \sigma(Z) + \sum_{i=1}^e Y_i X_i Z \prod_{j \neq i}^e (1 - X_j Z)$$

$Z = X_i^{-1}$ ve X_i^{-1} $\sigma(Z)$ 'in bir kökü iken

$$W(X_i^{-1}) = \sigma(X_i^{-1}) + \sum_{i=1}^e Y_i X_i X_i^{-1} \prod_{j \neq i}^e (1 - X_j X_i^{-1}) \quad (4.3.2)$$

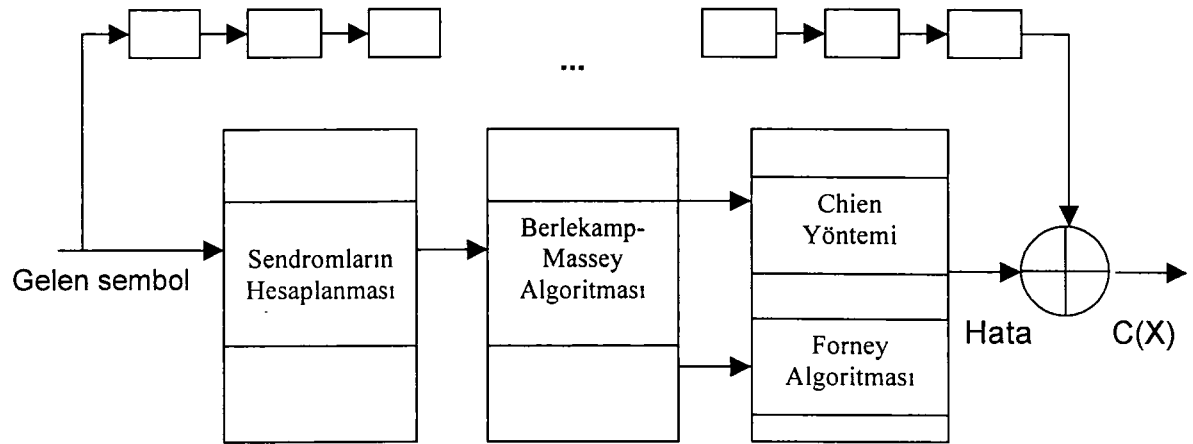
$$W(X_i^{-1}) = Y_i \prod_{j \neq i} (1 - X_j X_i^{-1}) \quad (4.3.3)$$

$$Y_i = W(X_i^{-1}) / \prod_{j \neq i} (1 - X_j X_i^{-1}) \quad (4.3.4)$$

$$= W(X_i^{-1}) / X_i^{-1} \sigma'(X_i^{-1}) \quad (4.3.5)$$

Chien araştırma yöntemi sonucunda $\sigma(z)$ sıfır olduğunda bulunan hata değeri, hatalı sembol kodçözücünden ayrılmadan önce sembole eklenir. Böylece hata yok edilir, kodçözücüü terk eden sembol düzeltilmiş olarak, ilk haline döner.

Kod çözme işlemlerini gerçekleştiren devre şeması şekil-4.3.1' de gösterilmiştir.



Şekil-4.3.1 Reed Solomon kodçözücü yapısı

Kanaldan gelen sembollerin kaydıran yazmaç kullanılarak hata yerleri ve değerleri bulununcaya kadar kodçözücüden ayrılmaması sağlanır. i nci sembol toplama devresine geldiğinde i nci hata değeri de gelmiş olur.

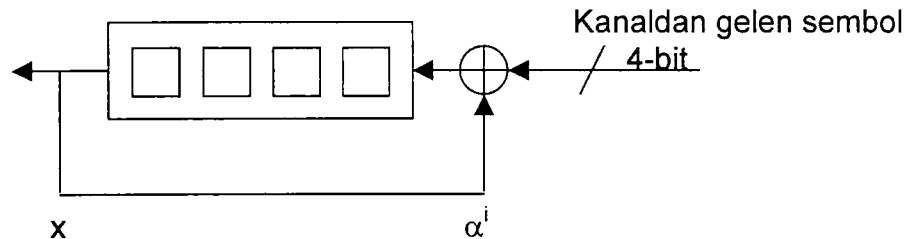
4.4. Reed Solomon Kodözücü Tasarımı

Reed Solomon kodözücü tasarımı için yapılan ilk alışmada [15]' de yer alan Berlekamp Algoritmasının VHDL kodu yazıldı. Kodlayıcıda olduđu gibi kodözücüde de 4-bit sembollü RS kodlar üzerinde alışıldı.

$m=4$, $n=15$ ve $k=9$, $t=(15-9)/2 = 3$ değ erlerine sahip ve sembolleri $GF(2^4)$ alanına ait RS(15,9) kodu için tasarım yapıldı. Tasarım kodözücüyü bloklara ayırarak gerekleştirildi.

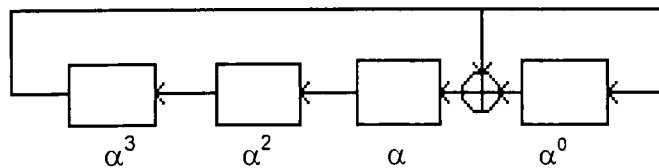
İlk olarak sendrom değ erlerinin hesaplandıđı blok oluřturuldu. Sendromlar, üretici polinom kökleri için alıcıya gelen polinomun aldıđı değ erlerdir. i nci sendrom, $R(\alpha^i)=E(\alpha^i)$, yani alınan polinomun $X-\alpha^i$ ile bölümünden kalan sayıdır. Bundan yola ıkarak üretici polinomun $2t$ kökünün her biri için bölme işlemini gerekleřtirecek VHDL kod yazıldı. Bölme işlemi için, kodlamada olduđu gibi doğrusal geribeslemeli kaydıran yazma esas alındı ve tüm bölme işlemlerinin aynı anda yapılması sağlandı.

$X-\alpha^i$ ile bölme işlemini gerekleřtiren devre řekil 4.4.1' de g sterilmiřtir.

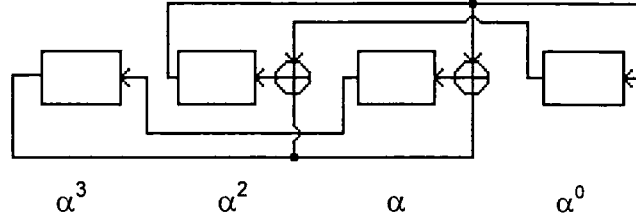


řekil 4.4.1 Alınan polinomu $X-\alpha^i$ ile b len devre

B lme yapılırken α^i nin kanaldan gelen sembollerle arpılması gerekir. Bu arpım Galois alan kurallarına g re yapılır.  rnek olarak herhangi bir sembol  α ve α^2 ile arpan devreler sırasıyla řekil 4.4.2 ve řekil 4.4.3' de g sterilmiřtir.



řekil 4.4.2 D rt bit sembol  α ile arpan devre



Şekil 4.4.3 Dört bit sembolü α^2 ile çarpan devre

Sembollerdeki dört bitin yerlerini sırayla α^0 , α^1 , α^2 ve α^3 e kadar olan sayılar belirtir. Sayının α ile çarpımı derecesinin artarak sola kaymasına neden olur. Derecesi dört ve dörtten büyük sayılar için minimal polinomdan gelen $\alpha^4 = \alpha + 1$ eşitliği kullanılır. Aynı kapana (latch) gelen bitler ise toplanarak çarpım sonucu bulunur.

Bu şekilde bölme ve çarpma işlemleri yapılarak S_0 ' dan S_5 e kadar tüm sendrom değerleri bulunur. Sendromları hesaplayan blok için yazılan VHDL kodu Ek-3' de yer almaktadır.

Sendromlar bulunduktan sonra, bu değerleri kullanarak anahtar eşitliğin çözümünü sağlayan kodçözücü yapısında yer alan ikinci bloğun tasarımı gerçekleştirildi. Bunun için önce [15]' de yer alan Berlekamp-Massey algoritması kullanıldı.

4.4.1 Berlekamp-Massey algoritmasına göre tasarım

Algoritmada izlenen adımlar aşağıda gösterilmiştir.

1. Adım :

$$\text{Uyumsuzluk hesabı : } \Delta^{(k)} = S_{k+1} + \sigma_1^{(k)} S_k + \dots + \sigma_{\text{dgr}}^{(k)} S_{k+1-\text{dgr}}$$

(dgr : $\sigma(Z)$ 'in derecesi)

2. Adım

Eğer $\Delta^{(k)} = 0$ ise

$$\sigma^{(k+1)}(Z) = \sigma^{(k)}(Z)$$

Değilse

k 'dan küçük, uyumsuzluk değeri sıfır olmayan ($\Delta^{(p)} \neq 0$) ve en büyük (p-dgr) değerine sahip p inci adım bulunur. $\sigma(Z)$ güncellenir.

$$\sigma^{(k+1)}(Z) = \sigma^{(k)}(Z) + \Delta^{(k-p)} \Delta^{(p)-1} Z^{k-p} \sigma^{(p)}(Z)$$

Çizelge 4.4.1.1' de algoritmada yer alan değişkenler ve başlangıç değerleri gösterilmiştir.

Çizelge 4.4.1.1 BM algoritması değişkenleri

k	$\sigma^{(k)}(Z)$	$\Lambda^{(k)}$	dgr	k-dgr
-1	1	1	0	-1
0	1	S_0	0	0
1				
2				
.				
.				
.				
2t				

k, 2t-1 olduğunda anahtar eşitlikteki hata yeri belirleyici polinom elde edilmiş olur. Bu algoritmada daha önce anlattığımız algoritmadan farklı olarak hata değeri belirleyici polinomunun katsayısı hata yeri belirleyici polinomun katsayıları ile aynı anda değil, hata yeri belirleyici polinom bulunduktan sonra $(1+S(Z))\sigma(Z)$ çarpımından elde edilir. Bu durumda $W(Z)$ polinomunun katsayılarını bulabilmek için yardımcı polinomlar kullanmasına gerek kalmaz. Böylece daha az bilgi saklandığı için bellek alanından kazanç elde edilir, ancak katsayıların hesaplanması için fazladan saat çevirimi kullanması gerekmektedir.

3. Adım :

Hata değeri belirleyici polinomunun bulunması : $(1+S(Z))\sigma(Z)=W(Z) \bmod Z^{2t+1}$

$$W(Z)=1+(S_1+\sigma_1)Z+(S_2+\sigma_1S_1+\sigma_2)Z^2+\dots+(S_t+\sigma_1S_{t-1}+\sigma_2S_{t-2}+\dots+\sigma_t)$$

Bu algoritma için yazılan VHDL kodunun (Bkz. Ek-4) Xilinx xcv600ebg432 yongası için sentezlenmesinden elde edilen sonuçlar çizelge 4.4.1.2' de yer almaktadır.

Çizelge 4.4.1.2 BM algoritması xcv600ebg432 yongası sentez sonuçları

Kaynak	Kullanılan	Kullanılabilir	Kullanım Oranı
IOs	83	316	%26.27
İşlev Üreteci	771	13824	%5.58
Karmaşık Mantık Blokları	386	6912	%5.58
Tutucular (Dffs)	266	15360	%1.73
Saat Frekansı = 35.9 MHz			

Sentez sonuçları incelendiğinde, FPGA kaynaklarının oldukça küçük bir miktarının kullanıldığı görülmektedir.

Sonraki çalışmada Berlekamp' in algoritmasında hata yeri belirleyici polinomun güncelleme aşamasında yer alan bölme işleminin uzun süren bir sayısal işlem olması ve algoritmanın döngüsel yapısı nedeniyle, yapısında bölme işleminin yer almadığı, orjinal Berlekamp-Massey Algoritmasının bir türevi olan evirmesiz Berlekamp-Massey Algoritması [5] kullanılarak yeni bir blok tasarımı gerçekleştirilip, daha önceki blokla hızları karşılaştırıldı.

4.4.2 Evirmesiz Berlekamp-Massey algoritmasına göre tasarım

Evirmesiz BM Algoritmasının yapısı aşağıda gösterilmiştir.

Başlangıç Değerleri :

$$\sigma_0^{(0)} = b_0^{(0)} = 1, \sigma_i^{(0)} = b_i^{(0)} = 1, i=1,2,...,t \text{ için}$$

$$cs^{(0)} = 0, \gamma^{(0)} = 1$$

k = 0'dan başlayıp 2t-1 oluncaya kadar aşağıdaki işlemler tekrarlanır.

1. Adım :

$$\Delta^{(k)} = S_{k+1} \sigma_0^{(k)} + S_k \sigma_1^{(k)} + \dots + S_{k+1-t} \sigma_t^{(k)}$$

2.Adım :

$$\sigma_i^{(k+1)} = \gamma^{(k)} \sigma_i^{(k)} - \Delta^{(k)} b_{i-1}^{(k)} \quad (i = 0, 1, \dots, t)$$

3. Adım :

eğer $\Delta^{(k)} \neq 0$ ve $cs^{(k)} \geq 0$ ise

$$b_i^{(k+1)} = \sigma_i^{(k)}$$

$$\gamma^{(k+1)} = \Delta^{(k)}$$

$$cs^{(k+1)} = -cs^{(k)} - 1$$

değilse

$$b_i^{(k+1)} = b_{i-1}^{(k)}$$

$$\gamma^{(k+1)} = \gamma^{(k)}$$

$$cs^{(k+1)} = cs^{(k)} + 1$$

$k=2t-1$ ' den sonra hata değeri belirleyici polinomun katsayıları bulunur. Bunun için anahtar eşitlik kullanılır. Forney algoritmasına göre hata değerini hesaplarırken sadece $S(Z)\sigma(Z)$ çarpımını bulmak yeterli olduğundan hata değeri belirleyici polinomun katsayıları bulunurken ayrıca $\sigma(Z)$ ile toplama yapılmasına gerek yoktur. Evirmesiz BM algoritmasında orijinal BM algoritmasından farklı olarak hata yeri belirleyici ve hata değeri belirleyici polinomun sayıl (scalar) bir katı bulunur. Ancak polinomların kökleri değişmeyeceğinden bulunan hata değerleri ve hata yerleri de değişmez.

4. Adım :

$$w_i^{(2t)} = S_{i+1}\sigma_0^{(2t)} + S_i\sigma_1^{(2t)} + \dots + S_1\sigma_i^{(2t)} \quad (i=0,1,\dots,t-1)$$

Evirmesiz BM algoritması tasarımı için iki farklı blok yapısı kullanıldı [2].

1-) Uyumsuzluk hesabının yapıldığı blok (DC)

2-) Hata yeri belirleyici ve diğer yardımcı polinomların güncellendiği blok (ELU)

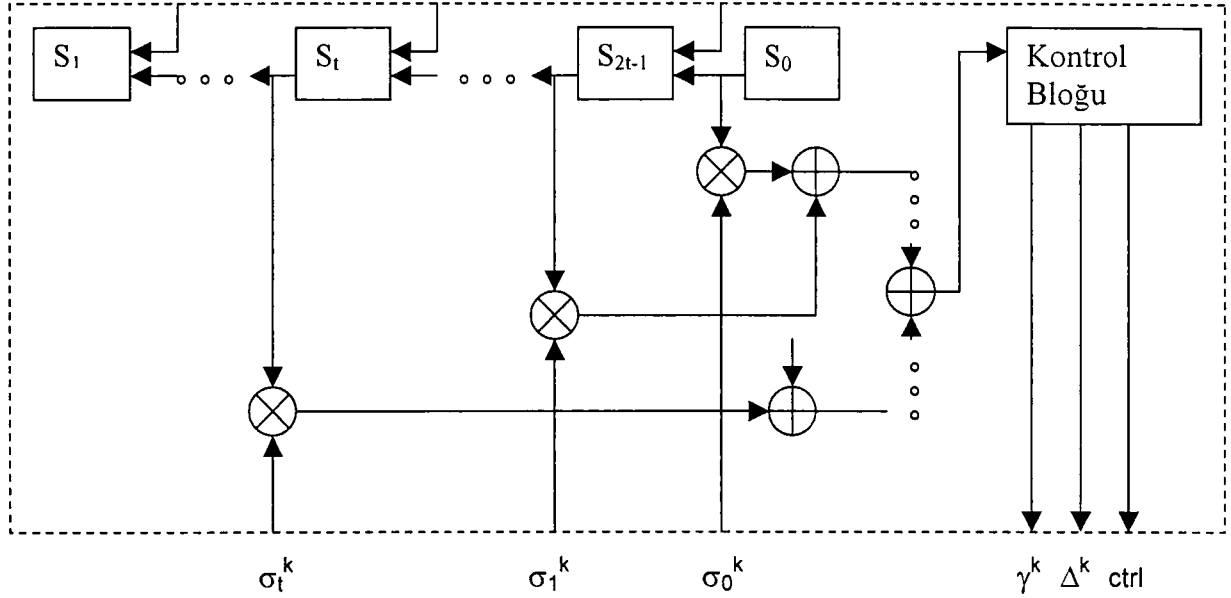
4.4.2.1 DC blok yapısı

DC blok da kendi içinde iki farklı bölümden oluşmaktadır. Bunlardan birinde uyumsuzluk değeri hesaplanırken, diğerinde uyumsuzluk değerine bağlı olarak algoritmada yer alan cs ve γ değeri güncellenir.

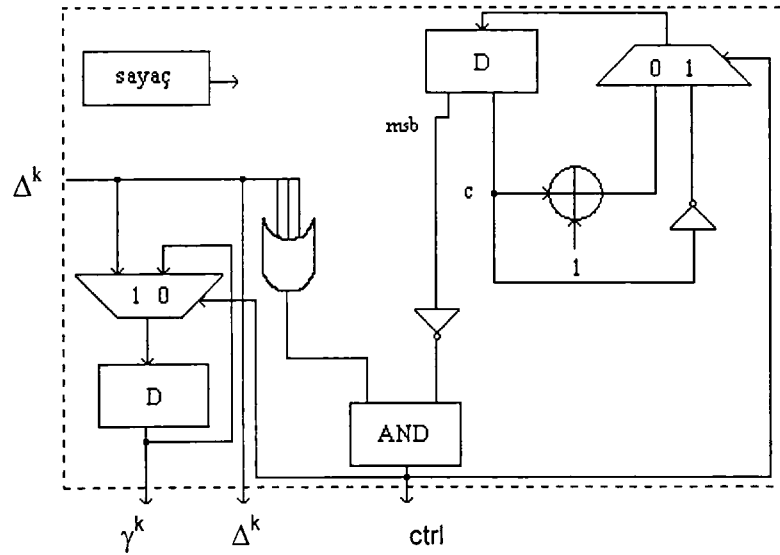
Şekil.4.4.2.1.1' de gösterilen ilk blokta sendrom değerlerini saklayan $2t$ tutucu, toplama ve çarpma işlemi yapan devreler yer alır. Kritik yol gecikmesi, $T_{\Delta} = T_{mult} + \lceil \log_2(t+1) \rceil T_{add}$ dir.

İkinci bloğun yapısı şekil 4.4.2.1.2' de gösterilmiş olup bu blok için kritik yol gecikmesi $T_{ctrl} = \lceil \log_2 m \rceil T_{or} + T_{and} + T_{add}$ 'e eşittir.

DC bloktaki toplam gecikme ise $T_{dc} = T_{mult} + (1 + \lceil \log_2(t+1) \rceil) T_{add} + \lceil \log_2 m \rceil T_{or} + T_{and} + T_{add}$ ' dir.



Şekil 4.4.2.1.1 DC blok yapısı



Şekil 4.4.2.1.2 Kontrol bloğu yapısı

4.3 Yeni Evirmesiz Berlekamp-Massey Algoritmasına Göre Tasarım

Evirmesiz BM algoritmasının ikinci adımında $\sigma^{(k+1)}(Z) = \gamma^{(k)} \sigma^{(k)}(Z) - Z \Delta^{(k)} B^{(k)}(Z)$ eşitliğinden hata yeri belirleyici polinomun yeni değerinin bulunduğunu, üçüncü adımında ise $B^{(k+1)}(Z)$ güncellenirken $\sigma^{(k)}(Z)$ yada $Z \cdot B^{(k)}(Z)$ değerlerinden birini aldığını görüyoruz.

Birinci adımda bulunan uyumsuzluk aslında 4.3.1 eşitliğinden görülebileceği gibi $\sigma^{(k)}(Z) \cdot S(Z)$ çarpımında Z^k teriminin katsayısıdır.

$$\begin{aligned} \sigma^{(k)}(Z) \cdot S(Z) &= \Delta^{(k)}(Z) \\ &= \Delta_0^{(k)} + \Delta_1^{(k)}Z + \Delta_2^{(k)}Z^2 + \dots + \Delta_k^{(k)}Z^k + \dots \end{aligned} \quad (4.3.1)$$

Yeni algoritmaya ulaşmak için $B(Z)$ ve $S(Z)$ çarpımına eşit (4.3.2) bir θ polinomu tanımlanmıştır.

$$\theta^k(Z) = B^k(Z) \cdot S(Z) \quad (4.3.2)$$

Eğer kodçözücü $\Delta^k(Z)$ ve $\theta^k(Z)$ polinomunun tüm katsayılarını bilirse, $\sigma(Z)$ ve $B(Z)$ polinomlarının yeni değerlerini bulmak için sadece $\Delta_k^{(k)}$ değerinin bilinmesi yeterli olmasına rağmen, daha hızlı bir şekilde bu değerleri bulabilir.

İlk saat çevrimi başında $\Delta^k(Z)$ ve $\theta^k(Z)$ polinomlarının ile $\sigma^k(Z)$ ve $B^k(Z)$ ' in tüm katsayılarının bilindiğini varsayalım. Böylece $\Delta_k = \Delta_k^{(k)}$ ilk saat çevriminde elimizde olur ve kodçözücü $\sigma^{k+1}(Z)$ ve $B^{k+1}(Z)$ 'i hesaplayabilir. Aynı zamanda 4.3.1 ve 4.3.2' den uyumsuzluk değerini bulan yeni bir eşitlik (4.3.5) elde edilir.

$$\Delta^{(k+1)}(Z) = \sigma^{(k+1)}(Z) \cdot S(Z) \quad (4.3.3)$$

$$= [\gamma^k \sigma^{(k)}(Z) - Z \Delta_k^{(k)} B^k(Z)] \cdot S(Z) \quad (4.3.4)$$

$$= \gamma^k \Delta^{(k)}(Z) - Z \Delta_k^{(k)} \theta^k(Z) \quad (4.3.5)$$

$\theta^{k+1}(Z)$ de uyumsuzluk ve $\Delta^{(k)}$ değerine bağlı olarak $\Delta^{(k)}(Z) = \sigma^{(k)}(Z) \cdot S(Z)$ yada $Z \theta^k(Z) = Z \cdot B^k(Z) \cdot S(Z)$ değerlerinden birini alır. Yani $\Delta^{k+1}(Z)$, $\theta^{k+1}(Z)$, $\sigma^{k+1}(Z)$, $B^{k+1}(Z)$ aynı saat çevriminde güncellenebilir.

Bu yeni formüle göre oluşan algorithmada başlangıç değerleri $\sigma^0(Z) = B^0(Z) = 1$, $\Delta^0(Z) = \theta^0(Z) = S(Z)$ şeklindedir.

2t döngü sonunda hata yeri belirleyici polinom ve $\sigma^{2t}(Z)$ bulunmuş olur. Hata değeri belirleyici polinom ($W(Z) \equiv [1 + S(Z)] \sigma(Z) \bmod^{2t}$) ise 4.3.1 nci eşitlikten görüldüğü gibi $\Delta^{2t}(Z)$ polinomunun düşük dereceli terimlerinden oluşan polinomun aynısıdır. Bu durumda 2t döngü sonunda hem hata yeri belirleyici polinom hem de hata değeri belirleyici polinom hesaplanmış olur. Böylece önceki algoritmada yapılanın aksine hata değeri belirleyici polinomu hesaplamak için fazladan saat çevrimi kullanılmasına gerek kalmaz.

Aynı şekilde $\Delta^{2t}(Z)$ ' in yüksek dereceli terimlerinden oluşan polinomda hata değerinin hesaplanması için kullanılabilir.

$\Delta^{2t}(Z)$ polinomunu iki polinomun toplamı şeklinde yazalım.

$$\Delta^{2t}(Z) = W^{2t}(Z) + Z^{2t} \cdot W^h(Z) \quad (4.3.6)$$

X_i^{-1} , $\sigma^{2t}(Z)$ 'in bir kökü olsun. 4.3.1 nci eşitliğe göre $\Delta^{2t}(X_i^{-1}) = W^{2t}(X_i^{-1}) + X_i^{-2t} \cdot W^h(X_i^{-1}) = 0$ olur ve Forney algoritması kullanılarak 4.3.7' yi yazabiliriz.

$$Y_i = W^h(X_i^{-1}) / X_i^{-1} \cdot \sigma^{2t}(X_i^{-1}) \quad (4.3.7)$$

Bu algoritmaya göre uyumsuzluk değerini bulan işlemlerle hata yeri belirleyici polinomun güncellenmesi için yapılan işlemlerin aynı olması nedeniyle, uyumsuzluk değeri de EBM algoritması tasarımında yer alan ELU bloğuna benzer bir yapı kullanılarak güncellenebilir. Böylece bu blok için kritik yol gecikmesi önceki yapıya göre azalmış olur. Ancak yeni algoritmada, uyumsuzluk değeri her seferinde farklı bir blokta hesaplandığından uyumsuzluk değerinin kontrol bloğu ve $\sigma^{k+1}(Z)$, $\theta^{k+1}(Z)$ 'in hesaplandığı bloklara iletilmesi için giriş yol seçici kullanılması gerekmektedir. Yeni algoritmada yapılacak bir değişiklikle bu sorun da giderilmiş olur. Bu değişiklik k' dan küçük değerlerin $\Delta_k^{(k)}$ dan sonraki uyumsuzluk değerlerini etkilememesinden yola çıkılarak yapılır. $i < k$ için $\Delta_i^{(k)}$ ve $\theta_i^{(k)}$ değerlerinin saklanmasına gerek yoktur. Yeni algoritmada yer alan tanımlar ve algoritma aşağıda gösterilmiştir.

$$Y \Delta_i^{(k)} = \Delta_{i+k}^{(k)} \quad k=0,1,2,\dots,2t-1$$

$$Y \theta_i^{(k)} = \theta_{i+k}^{(k)}$$

Polinomlar :

$$Y\Delta^k(Z) = \sum_{i=0}^{2t-1} Y\Delta_i^{(k)} Z^i \quad \text{ve} \quad Y\theta^k(Z) = \sum_{i=0}^{2t-1} Y\theta_i^{(k)} Z^i$$

Başlangıç değerleri:

$$Y\Delta^0(Z) = Y\theta^0(Z) = S(Z)$$

Yeni uyumsuzluk değeri :

$$\begin{aligned} Y\Delta_i^{(k+1)} &= \Delta_{i+k+1}^{(k+1)} \\ &= \gamma^k \Delta_{i+k+1}^{(k)} - \Delta_k^{(k)} \theta_{k+i}^{(k)} \\ &= \gamma^{(k)} Y\Delta_{i+1}^{(k)} - Y\Delta_0^{(k)} Y\theta_i^{(k)} \end{aligned}$$

Yeni $\theta(Z)$:

$Y\theta_i^{k+1}$, uyumsuzluk ve k değerine göre $Y\theta_i^k$ ya da $Y\Delta_{i+1}^{(k)}$ değerlerinden birini alır.

$Y\Delta^{2t}(Z) = \Delta_{2t}^{(2t)} + \Delta_{2t+1}^{(2t)} + \dots = W^h(Z)$, böylece $Y\Delta^{2t}(Z)$ hata değerlerini bulmada kullanılabilir.

Buna göre oluşan yeni algoritma :

Başlangıç Değerleri :

$$\sigma_0^0 = b_0^0 = 1, i=1,2, \dots, t \text{ için } \sigma_i^0 = b_i^0 = 0, k^0=0, \gamma^0 = 1$$

$$Y\Delta_i^0 = Y\theta_i^0 = S_i, i=0,1,\dots,2t-1$$

$k=0$ 'dan $2t-1$ 'e kadar devam eden adımlar:

1.Adım :

$$\sigma_i^{k+1} = \gamma^k \sigma_i^k - Y\Delta_0^k b_{i-1}^k \quad i=0,1,\dots,t$$

$$Y\Delta_i^{k+1} = \gamma^k Y\Delta_{i+1}^k - Y\Delta_0^k \theta_i^k \quad i=0,1,\dots,2t-1$$

2.Adım :

Eğer $Y\Delta_0^k \neq 0$ ve $k^k \geq 0$ ise

$$b_i^{k+1} = \sigma_i^k \quad i=0,1,\dots,t$$

$$\theta_i^{k+1} = Y\Delta_{i+1}^k \quad i=0,1,\dots,2t-1$$

$$\gamma^{k+1} = Y\Delta_0^k$$

$$k^{k+1} = -k^k - 1$$

Değilse

$$b_i^{k+1} = b_{i-1}^k \quad i=0,1,\dots,t$$

$$\begin{aligned}\theta_i^{k+1} &= \theta_i^k & i=0,1,\dots,2t-1 \\ \gamma^{k+1} &= \gamma^k \\ k^{k+1} &= k^k + 1\end{aligned}$$

Algoritmada yer alan $b_{-1}^k = Y\Delta_k^{2t}$ bütün k değerleri için sıfıra eşittir.

Yeni algoritmaya göre yazılan VHDL kodunun (Bkz. Ek-6) Xilinx xcv600ebg432 yongası için sentezlenmesi sonucunda Çizelge 4.3.1' de yeralan sonuçlar elde edildi.

Çizelge 4.3.1 Yeni EBM algoritması xcv600ebg432 yongası sentez sonuçları

Kaynak	Kullanılan	Kullanılabilir	Kullanım Oranı
IOs	70	316	%22.15
İşlev Üretici	648	13824	%4.69
Karmaşık Mantık Blokları	324	6912	%4.69
Tutucular	141	15360	%0.92
Saat frekansı = 89.6 MHz			

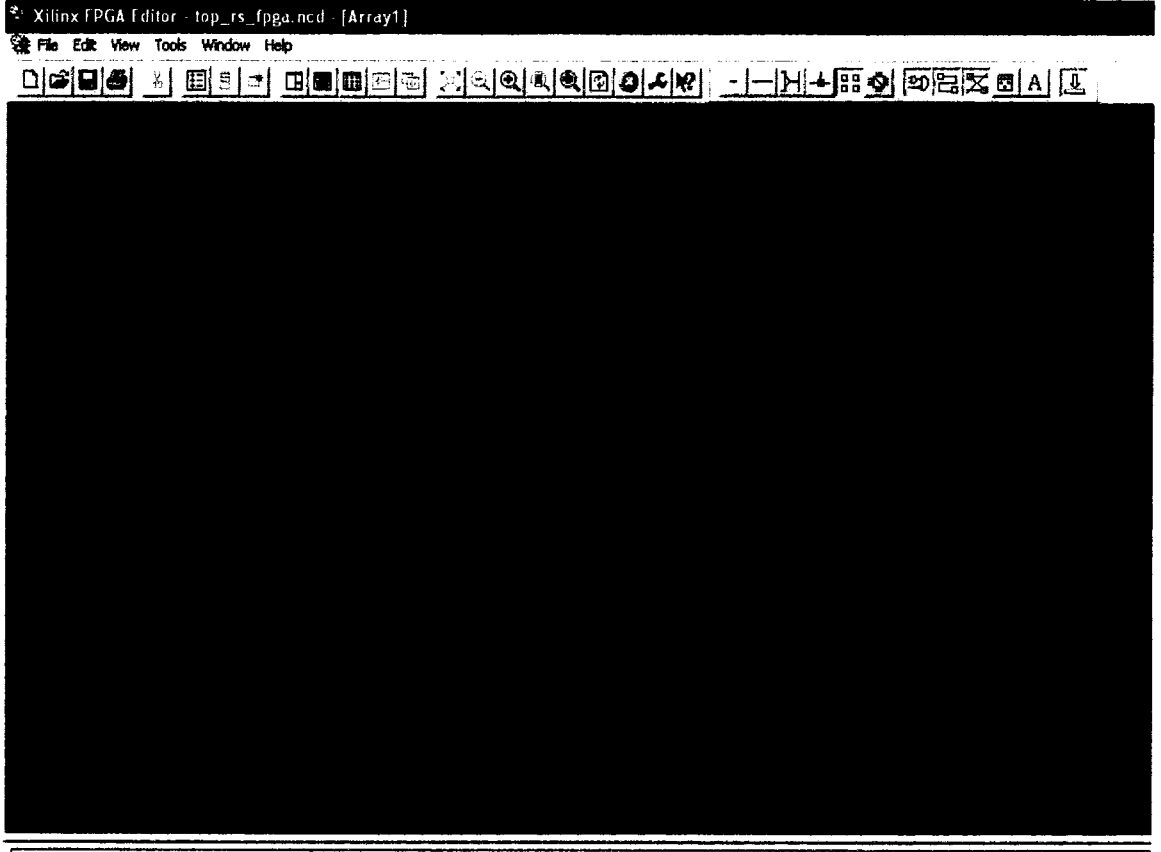
Sonuçlar incelendiğinde, algoritmadaki değişiklik nedeniyle artan polinom sayısına bağlı olarak kaynak kullanımının EBM algoritmasına göre arttığı, ancak daha hızlı bir çalışma frekansı elde edildiği görülmektedir.

Hata yeri belirleyici ve hata değeri belirleyici polinomların katsayıları bulunduğundan sonra Chien araştırması ve hata değerini bularak hatanın düzeltilmesini sağlayan blokların tasarımı (CSEE Blok) gerçekleştirildi.

CSEE blok kendi içinde iki ayrı bloktan meydana gelen bir yapı olarak oluşturuldu. İlk blokta chien araştırması yapılarak kodçözücünden ayrılacak her sembolün yeri için hata yeri belirleyici polinomun değeri ve Forney algoritmasının payda değeri olan $z\sigma'(Z)$ bulunarak ikinci bloğa iletildi, ikinci blokta ise, Forney algoritmasının pay kısmı olan $W(Z)$ değeri sıradaki sembolün yeri için bulundu, hata belirleyici polinomun sıfır olduğu durumlarda hata değeri hesaplandı ve kodçözücünden gönderilecek sembole eklenerek hatanın düzeltilmesi sağlandı.

CSEE blok tasarımı da yapıldıktan sonra, sendrom hesaplayan blok, anahtar eşitliği çözen blok ve CSEE blok arasında kontrolü sağlayacak sinyali üreten bir sayaç bloğu, kanaldan gelen sembollerin hata değeri ve yeri bulunana kadar

gecikmesini saęlayacak yazma tasarımı oluřturuldu. Ayrıca bu bloklarda kullanılan fonksiyon ve veri trlerinin tanımlarının yer aldığı bir paket oluřturuldu. (Bkz. Ek-7) Daha sonra btn bloklar birleřtirilip kodzc tasarımı tamamlanarak sentez iřlemi gerekleřtirildi. Sentez sonucu elde edilen FPGA serimi řekil 4.3.1' de gsterilmiřtir.



řekil 4.3.1 RS Kodzc FPGA serimi

4.4. Simülasyon Çalışmaları

Kodçözücünün işlevsel çalışmasını test etmek için yapılan simülasyon çalışmaları ve sonuçları aşağıda yer almaktadır.

Örnek 4.1

Data sembolleri=(1001, 1001, 1111, 0100, 0010, 0001, 1000, 0110, 1000)

Kodsözcüğü + Hatasözcüğü = Alınansözcük

C ₁₄ : 1001	-	R ₁₄ : 1001
C ₁₃ : 1001	-	R ₁₃ : 1001
C ₁₂ : 1111	-	R ₁₂ : 1111
C ₁₁ : 0100	-	R ₁₁ : 0100
C ₁₀ : 0010	E ₁₀ : 1000	R ₁₀ : 1010
C ₉ : 0001	-	R ₉ : 0001
C ₈ : 1000	E ₈ : 0110	R ₈ : 1110
C ₇ : 0110	-	R ₇ : 0110
C ₆ : 1000	E ₆ : 0001	R ₆ : 1001
C ₅ : 1101	-	R ₅ : 1101
C ₄ : 0110	-	R ₄ : 0110
C ₃ : 0000	-	R ₃ : 0000
C ₂ : 1010	-	R ₂ : 1010
C ₁ : 1101	-	R ₁ : 1101
C ₀ : 1111	-	R ₀ : 1111

Alınan veriye göre bulunan sendrom değerleri :

$$S_0=1100$$

$$S_1=0110$$

$$S_2=1001$$

$$S_3=1100$$

$$S_4=0101$$

$$S_5=0001$$

Sendrom değerlerinden bulunan hata yeri belirleyici polinomun katsayıları :

$$\sigma_0 = 0111, \sigma_1 = 1100, \sigma_2=1001, \sigma_3=0011$$

Anahtar eşitlikten bulunan hata değeri belirleyici polinomun katsayıları :

$$w_7 = 0000, w_8 = 0110, w_9=0011$$

Hata yeri belirleyici polinom :

$$\sigma(Z) = \alpha^{10} + \alpha^6 x Z + \alpha^{14} x Z^2 + \alpha^4 x Z^3$$

Hata bulucu polinom :

$$W^h(Z) = \alpha^5 x Z^8 + \alpha^4 Z^9$$

$$Zx\sigma'(Z) = \alpha^6 x Z + \alpha^4 x Z^3$$

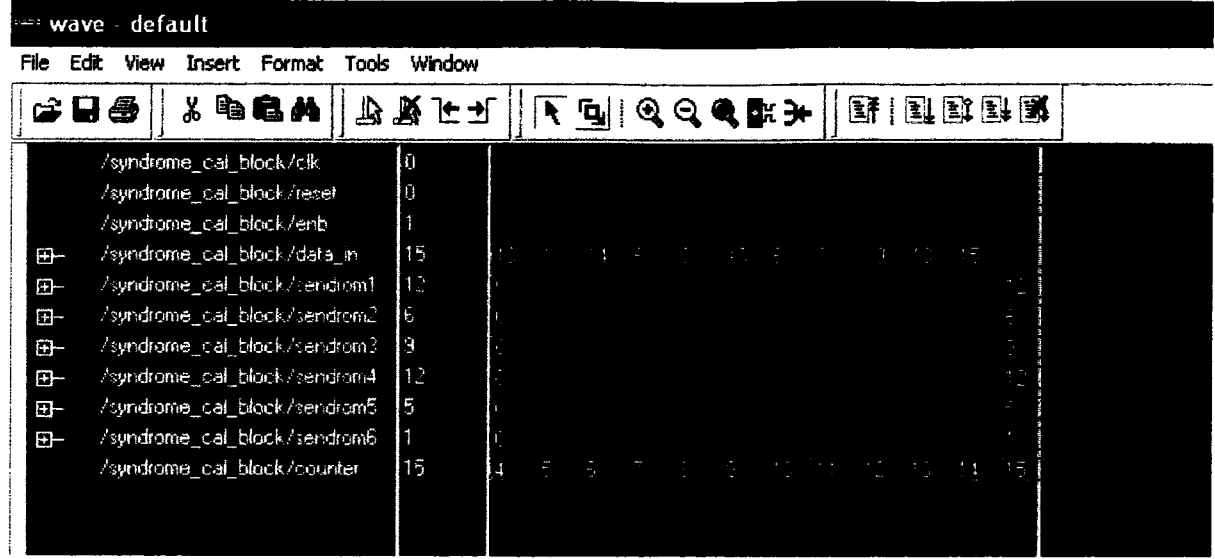
Forney Algoritmasına göre bulunan hata değerleri :

$$\begin{aligned} Y_{10} &= W^h(Z) / Zx\sigma'(Z) \quad Z = \alpha^5 \\ &= (\alpha^5 x \alpha^{40} + \alpha^4 x \alpha^{45}) / (\alpha^6 x \alpha^5 + \alpha^4 x \alpha^{15}) \\ &= (\alpha^0 + \alpha^4) / (\alpha^{11} + \alpha^4) \\ &= \alpha / \alpha^{13} \\ &= \alpha^{-12} \\ &= \alpha^3 \end{aligned}$$

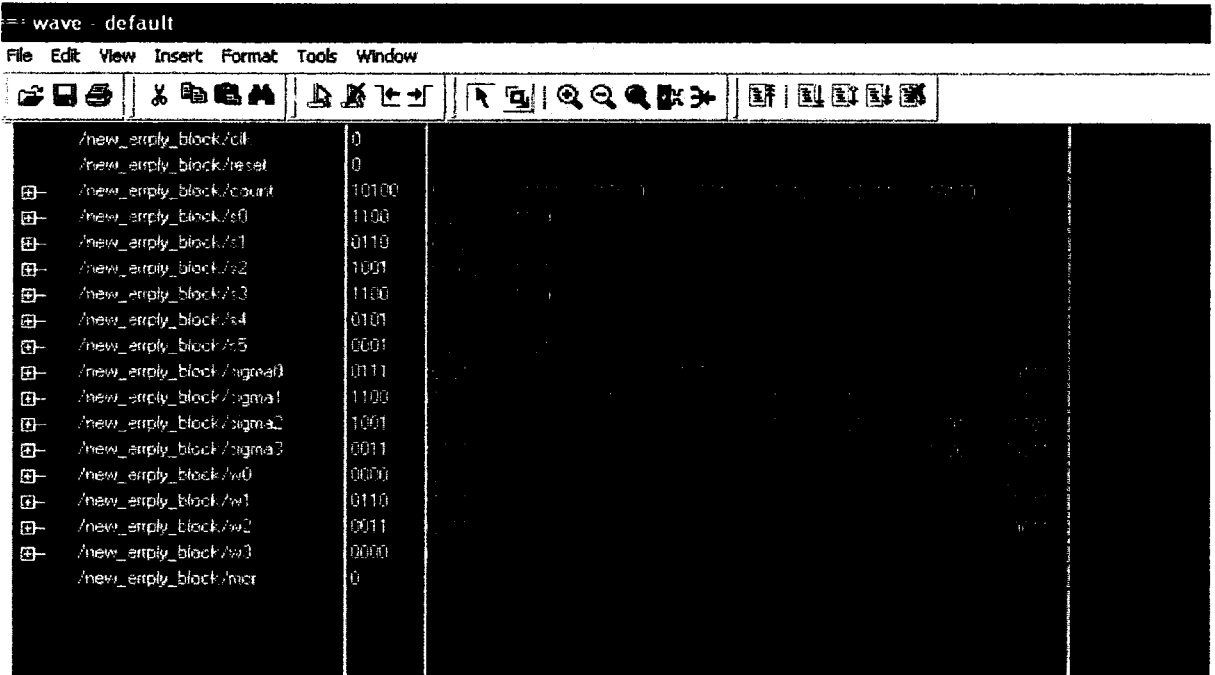
$$\begin{aligned} Y_8 &= W^h(Z) / Zx\sigma'(Z) \quad Z = \alpha^7 \\ &= (\alpha^5 x \alpha^{56} + \alpha^4 x \alpha^{63}) / (\alpha^6 x \alpha^7 + \alpha^4 x \alpha^{21}) \\ &= (\alpha + \alpha^7) / (\alpha^{13} + \alpha^{10}) \\ &= \alpha^{14} / \alpha^9 \\ &= \alpha^5 \end{aligned}$$

$$\begin{aligned} Y_6 &= W^h(Z) / Zx\sigma'(Z) \quad Z = \alpha^9 \\ &= (\alpha^5 x \alpha^{72} + \alpha^4 x \alpha^{81}) / (\alpha^6 x \alpha^9 + \alpha^4 x \alpha^{27}) \\ &= (\alpha^2 + \alpha^{10}) / (\alpha^0 + \alpha) \\ &= \alpha^4 / \alpha^4 \\ &= \alpha^0 \end{aligned}$$

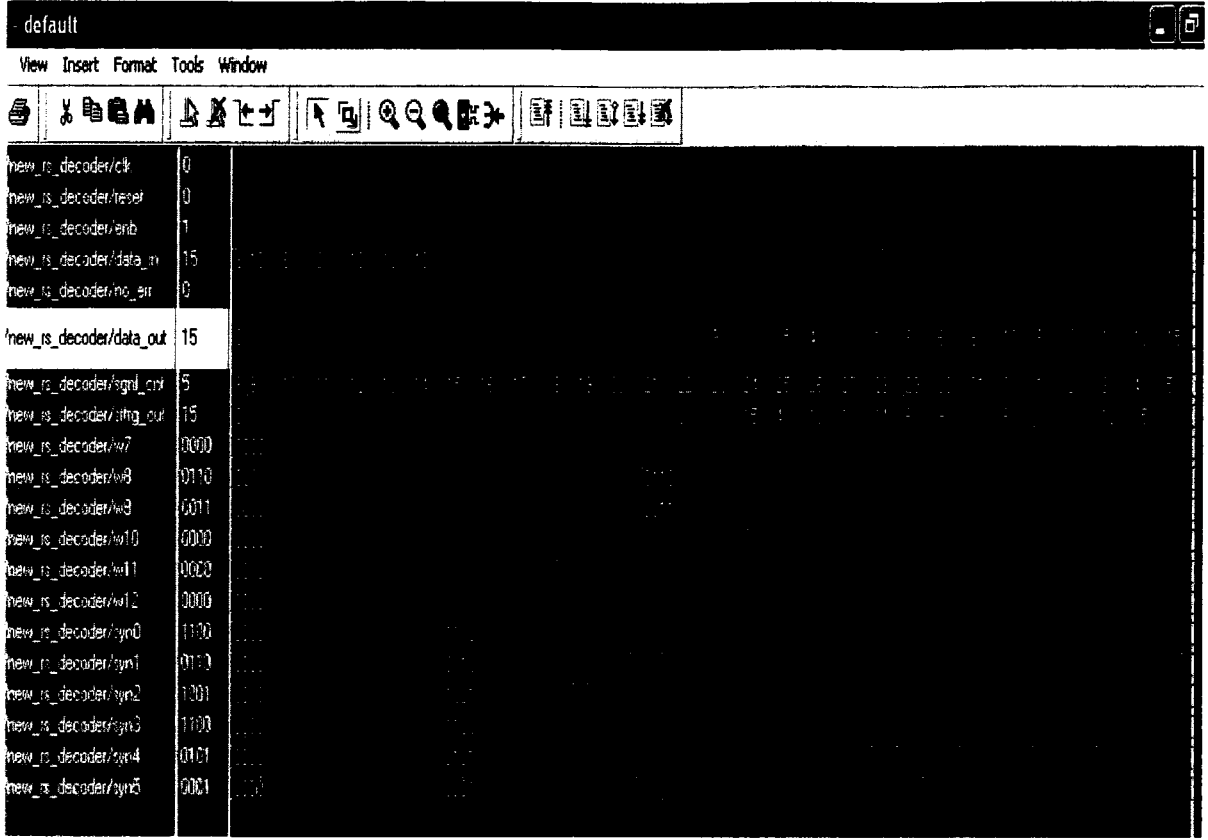
Yukarıda verilen $R(X)$ (alınansözcük) için Modelsim programı ile gerçekleştirilen simülasyon çalışmalarının sonuçları şekil 4.4.1-4.4.3' te gösterilmiştir.



Şekil 4.4.1 Sendrom bloğu simülasyon sonucu



Şekil 4.4.2 Yeni EBM algoritması için simülasyon sonucu



Şekil 4.4.3 Kodçözücü simülasyon sonuçları

Şekil 4.4.3' te yeralan simülasyon sonucuna göre hatalı verinin düzeltilerek kodçözücünden iletildiği görülmektedir.

Örnek 4.2

Data sembolleri=(1101, 0110, 1111, 0000, 1100, 1110, 1010, 0111, 0011)

Kodsözcüğü + Hatasözcüğü = Alınansözcük

C ₁₄ : 1101	-	R ₁₄ : 1101
C ₁₃ : 0110	-	R ₁₃ : 0110
C ₁₂ : 1111	E ₁₂ : 1000	R ₁₂ : 0111
C ₁₁ : 0000	-	R ₁₁ : 0000
C ₁₀ : 1100	E ₁₀ : 1001	R ₁₀ : 0101
C ₉ : 1110	-	R ₉ : 1110
C ₈ : 1010	E ₈ : 0001	R ₈ : 1011
C ₇ : 0111	-	R ₇ : 0111
C ₆ : 0011	-	R ₆ : 0011
C ₅ : 1101	-	R ₅ : 1000
C ₄ : 1000	-	R ₄ : 1000
C ₃ : 1001	-	R ₃ : 1001
C ₂ : 1001	-	R ₂ : 1001
C ₁ : 0001	-	R ₁ : 0001
C ₀ : 1001	-	R ₀ : 1001

Alınan veriye göre bulunan sendrom değerleri :

$$S_0=1110$$

$$S_1=1110$$

$$S_2=1001$$

$$S_3=0010$$

$$S_4=1100$$

$$S_5=0000$$

Sendrom değerlerinden bulunan hata yeri belirleyici polinomun katsayıları :

$$\sigma_0 = 1010, \sigma_1 = 1011, \sigma_2=0100, \sigma_3=1010$$

Anahtar eşitlikten bulunan hata değeri belirleyici polinomun katsayıları :

$$W_7 =0010, w_8 =0001, w_9 =0000$$

Hata yeri belirleyici polinom :

$$\sigma(Z) = \alpha^9 + \alpha^7 x Z + \alpha^2 x Z^2 + \alpha^9 x Z^3$$

Hata değeri belirleyici polinom :

$$W^h(Z) = \alpha^7 x Z^7 + Z^8$$

$$Zx\sigma'(Z) = \alpha^7 x Z + \alpha^9 x Z^3$$

Forney Algoritmasına göre bulunan hata değerleri :

$$\begin{aligned} Y_{12} &= W^h(Z) / Zx\sigma'(Z) \quad Z = \alpha^3 \\ &= (\alpha x \alpha^{21} + \alpha^{24}) / (\alpha^7 x \alpha^3 + \alpha^9 x \alpha^9) \\ &= (\alpha^7 + \alpha^9) / (\alpha^{10} + \alpha^3) \\ &= \alpha^0 / \alpha^{12} \\ &= \alpha^{-12} \\ &= \alpha^3 \end{aligned}$$

$$\begin{aligned} Y_{10} &= W^h(Z) / Zx\sigma'(Z) \quad Z = \alpha^5 \\ &= (\alpha x \alpha^{35} + \alpha^{40}) / (\alpha^7 x \alpha^5 + \alpha^9 x \alpha^{15}) \\ &= (\alpha^6 + \alpha^{10}) / (\alpha^{12} + \alpha^9) \\ &= \alpha^7 / \alpha^8 \\ &= \alpha^{-1} \\ &= \alpha^{14} \end{aligned}$$

$$\begin{aligned} Y_8 &= W^h(Z) / Zx\sigma'(Z) \quad Z = \alpha^7 \\ &= (\alpha x \alpha^{49} + \alpha^{56}) / (\alpha^7 x \alpha^7 + \alpha^9 x \alpha^{21}) \\ &= (\alpha^5 + \alpha^{11}) / (\alpha^{14} + \alpha^0) \\ &= \alpha^3 / \alpha^3 \\ &= \alpha^0 \end{aligned}$$

Örnek 4.3

Dataword=(1100, 0101, 0011, 1001, 1101, 1011, 1000, 0001, 1111)

Kodsözcüğü + Hatasözcüğü = Alınansözcük

C ₁₄ : 1100	-	R ₁₄ : 1100
C ₁₃ : 0101	E ₁₃ : 1111	R ₁₃ : 1010
C ₁₂ : 0011	-	R ₁₂ : 0011
C ₁₁ : 1001	-	R ₁₁ : 1001
C ₁₀ : 1101	E ₁₀ : 1000	R ₁₀ : 0101
C ₉ : 1011	-	R ₉ : 1011
C ₈ : 1000	E ₈ : 0011	R ₈ : 1011
C ₇ : 0001	-	R ₇ : 0001
C ₆ : 1111	-	R ₆ : 1111
C ₅ : 0011	-	R ₅ : 0011
C ₄ : 1111	-	R ₄ : 1111
C ₃ : 0001	-	R ₃ : 0001
C ₂ : 1110	-	R ₂ : 1110
C ₁ : 1111	-	R ₁ : 1111
C ₀ : 0101	-	R ₀ : 0101

Alınan veriye göre bulunan sendrom değerleri :

$$S_0=0101$$

$$S_1=0110$$

$$S_2=1001$$

$$S_3=0010$$

$$S_4=1000$$

$$S_5=0010$$

Sendrom değerlerinden bulunan hata yeri belirleyici polinomun katsayıları :

$$\sigma_0 = 0001, \sigma_1 = 1111, \sigma_2=0001, \sigma_3=0010$$

Anahtar eşitlikten bulunan error evaluator polinomun katsayıları :

$$w_7 = 0001, w_8 = 0001, w_9=0100$$

Hata yeri belirleyici polinom :

$$\sigma(Z) = \alpha^0 + \alpha^{12} x Z + \alpha^0 x Z^2 + \alpha x Z^3$$

Hata değeri belirleyici polinom :

$$W^h(Z) = \alpha^0 x Z^7 + \alpha^0 x Z^8 + \alpha^2 x Z^9$$

$$Zx\sigma'(Z) = \alpha^{12} x Z + \alpha x Z^3$$

Forney Algoritmasına göre bulunan hata değerleri :

$$Y_{13} = W^h(Z) / Zx\sigma'(Z) \quad Z = \alpha^2$$

$$= (\alpha^0 x \alpha^{14} + \alpha^0 x \alpha^{16} + \alpha^2 x \alpha^{18}) / (\alpha^{12} x \alpha^2 + \alpha x \alpha^6)$$

$$= (\alpha^{14} + \alpha + \alpha^5) / (\alpha^{14} + \alpha^7)$$

$$= \alpha^{13} / \alpha$$

$$= \alpha^{12}$$

$$Y_{10} = W^h(Z) / Zx\sigma'(Z) \quad Z = \alpha^5$$

$$= (\alpha^0 x \alpha^{35} + \alpha^0 x \alpha^{40} + \alpha^2 x \alpha^{45}) / (\alpha^{12} x \alpha^5 + \alpha x \alpha^{15})$$

$$= (\alpha^5 + \alpha^{10} + \alpha^2) / (\alpha^2 + \alpha)$$

$$= \alpha^8 / \alpha^5$$

$$= \alpha^3$$

$$Y_8 = W^h(Z) / Zx\sigma'(Z) \quad Z = \alpha^7$$

$$= (\alpha^0 x \alpha^{49} + \alpha^0 x \alpha^{56} + \alpha^2 x \alpha^{63}) / (\alpha^{12} x \alpha^7 + \alpha x \alpha^{21})$$

$$= (\alpha^4 + \alpha^{11} + \alpha^5) / (\alpha^4 + \alpha^7)$$

$$= \alpha^7 / \alpha^3$$

$$= \alpha^4$$

5. SONUÇ

Bu çalışmada Reed Solomon kodlar üzerine araştırma yapılmış, ilk aşama olarak RS(15,9) kodlar için kodlayıcı tasarımı VHDL ile gerçekleştirilerek, FPGA serimi oluşturulmuştur.

Sonraki aşamada kodçözme algoritmaları hakkında bilgi verilmiş, kodçözme sürecinin bir parçası olan anahtar eşitliğin çözümü için yaygın olarak kullanılan Berlekamp-Massey algoritması ve değişik türevleri incelenmiş, ve bu algoritmalara dayanan kodçözücü tasarımları üzerinde çalışılmıştır.

Reed Solomon kodlar üzerine gerçekleştirilmiş algoritma tabanlı bir çalışmada [2], BM algoritmasının bölme işlemi içermeyen bir türü olan evirmesiz BM algoritması değiştirilerek kritik yol gecikmesi önceki algoritmalara göre daha düşük bir yapı oluşturmayı sağlayan yeni bir algoritma elde edilmiştir.

RS kodçözümleme işleminde anahtar eşitliğin çözümünü sağlayan blok tasarımı için Berlekamp Massey algoritması, BM algoritmasının bir türevi olan evirmesiz BM algoritması ve evirmesiz BM algoritmasında yapılan değişikliklerle elde edilen yeni algoritmaya göre VHDL kod yazılarak sentezlenmiştir. Çizelge 5.1' de sentez sonuçları karşılaştırıldığında yeni evirmesiz BM algoritmasının daha hızlı çözüm sağladığı ve daha yüksek işlem hacmine sahip olduğu görülmüştür. Ayrıca yeni EBM algoritması için elde edilen işlem hacminin daha önceki uygulamalarda elde edilen 144Mb/s ve 1.28 Gb/s arasında değişen oranlarla da [2] kıyaslanabilir olduğu gözlemlenmiştir.

Çizelge 5.1 Tasarım sonuçları

	Kaynaklar					
	IOs	İşlev Üreteçleri	CLB Kesitleri	Tutucular	Saat Frekansı	İşlem Hacmi
BM Algoritması	%26.27	%5.58	%5.58	%1.73	%35.9 MHz	143 Mb/s
EBM Algoritması	%15.82	%1.59	%1.59	%0.52	57.1 MHz	228 Mb/s
Yeni EBM Algoritması	%22.15	%4.69	%4.69	%0.92	89.6 MHz	358 Mbp/s

BM algoritması için yapılan tasarımdan sonra, kodçözücü yapısında yer alan diğer blokların da tasarımları VHDL ile yapılarak kodçözücü tamamlanmış, FPGA serimi oluşturularak, kodçözücünün işlevsel çalışması simülasyon programı kullanılarak test edilmiştir.

Bu çalışmada yapılan tasarım geliştirilerek, hata düzeltimi yanında, değeri bilinmeyen verilerin tahmini işlemlerini yerine getirebilen, ayrıca değişken blok uzunluğundaki RS kodlar için çalışan kodçözücü tasarımları gerçekleştirilmesi mümkündür.

Tasarımda kullanılan algoritmanın, RS kodçözücüler üzerine yapılmış yüksek hız ve düşük enerji tüketimi sağlayan algoritma tabanlı diğer çalışmalara uyarlanarak, daha hızlı ve verimli yapılar oluşturulabileceği ve gerçekleştirilen tasarımın bu çalışmalara zemin hazırlayacağı düşünülmektedir.

KAYNAKLAR

1. Arun Raghupathy and K.J.R. Liu. Algorithm-Based Low-Power/High-Speed Reed-Solomon Decoder Design, IEEE Transactions on Circuits and Systems, Vol.47, No.11 November 2000, pp.1254-1270.
2. Dilip V. Sarwate ve Naresh R. Shanbhag. High-Speed Architectures for Reed-Solomon Decoders, IEEE Transactions Very Large Scale Integration Systems (VLSI), Vol.9, No.5 October 2001, pp.641-655.
3. Elwyn R. Berlekamp, Algebraic Coding Theory, McGraw-Hill Book Company-1968.
4. G.D.Forney On Decoding BCH Codes IEEE Transactions on Information Theory Vol.11, No.4 October 1965 pp.549-557.
5. Herbert O. Burton, Inversionless Decoding of Binary BCH Codes, IEEE Transactions on Information Theory, Vol.17, No.4, July, 1971, pp.464-466
6. H.Shao and I.S.Reed. On The VLSI Design of a Pipeline Reed Solomon Decoder Using Systolic Arrays. IEEE Transactions on Computers Vol.37 No.10 October 1988. pp.1273-1280.
7. J.B.Cain and D.N.McGregor. A Recommended Error Control Architecture for ATM Networks with Wireless Links. IEEE Journal on Selected Areas in Communications. Vol.15, No.11 January 1997 pp.16-28.
8. J.L.Massey, Shift Register Synthesis and BCH Coding. IEEE Transactions on Information Theory Vol.15 January 1969 pp.122-127.
9. James S. Plank, A Tutorial on Reed-Solomon Coding for Fault-Tolerance in RAID-like Systems, <http://www.cs.utk.edu/~plank/plank/papers/CS-96-332.html> adresinden elde edilebilir.
10. R.D.Cideciyan and E.Eleftheriou, Concatenated Reed-Solomon/Convolutional Coding For Data Transmission in CDMA-Based Cellular Systems. IEEE Transactions on Communications Vol.45 No.10 October 1997, pp.1291-1303.
11. R.D.M.Hunter (Portsmouth Üniversitesi), T.T.Jonson (Summit Design Inc.) Introduction to VHDL, Chapman & Hall, 1996.
12. R.E. Blahut, Theory and Practice of Error Control Codes Addison Wesley, 1984.
13. S.B.Wicker Error Control Systems for Digital Communication And Storage Prentice-Hall 1995.
14. Stephen B.Wicker Vijay K. Bhargava, Reed-Solomon Codes and Their Applications, IEEE press, 1994.

15. Shu Lin / Daniel J. Costello Error Control Coding Fundamentals and Applications, 1983.
16. Zoran Salcic (Auckland Üniversitesi), Vhdl and Fplds In Digital Systems Design, Prototyping and Customization, Kluwer Academic Publishers.
17. Hanhoo Lee and Asad Azam. A low-Complexity, High Speed Reed Solomon Decoder [http:// www. ee.uconn.edu/~lee/mypapers/SOC03pdf](http://www.ee.uconn.edu/~lee/mypapers/SOC03pdf) adresinden elde edilebilir.
18. Reed/Chen Error Coding for Data Network [http:// www. ee.nctu.tw /course /channel coding](http://www.ee.nctu.tw/course/channel_coding) adresinden elde edilebilir.
19. R.T.Chien Cyclic Decoding Procedure for the Bose-Chaudhuri-Hocquenghem Codes, IEEE Transaction on Information Theory, vol.IT-10,pp 357-363.

EK-1 Kodlayıcı ve Kodçözücü için kullanılan paket VHDL kodu

--Bu paket RS(15,9) kodlayıcı ve kodçözücü bloklarında ortak olarak
--kullanılan fonksiyon ve veri türlerini tanımlamak için oluşturulmuştur.

```
library ieee;
use ieee.std_logic_1164.all;
package mypack is

function std_logic2int (value : std_logic_vector(3 downto 0)) return integer;
function mlt_of_vector(vector1, vector2 : std_logic_vector(3 downto 0))
return std_logic_vector;
function inv_of_vector(vector : std_logic_vector(3 downto 0))
return std_logic_vector;
subtype int_1to14 is integer range -1 to 14;
type typalpha_to is array(-1 to 14) of std_logic_vector(3 downto 0);
type typindex_of is array (0 to 15) of int_1to14;

--index_of 0' dan 15'e kadar indeks değerleri için tanımlı 16 boyutlu bir dizidir.
--indekslere karşılık gelen değer, o indeksi veren temel elemanın ( $\alpha$ ) kuvvetidir.
--Ondalık değeri bilinen GF elemanının  $\alpha$ ' nın kaçınıcı kuvveti olduğunu bulur.

constant index_of : typindex_of :=(-1,0,1,4,2,8,5,10,3,14,9,7,6,13,11,12);

--alpha_to,  $\alpha$ ' nın -1' den 14' e kadar kuvvetlerinin 4-bit karşılığını bulan dizidir.

constant alpha_to : typalpha_to :=("0000","0001","0010","0100", "1000","0011",
"0110", "1100","1011","0101", "1010","0111","1110","1111","1101","1001");
end mypack;

package body mypack is

-- 4-bit sembollerin tamsayı değerini bulan fonksiyon

function std_logic2int (value : std_logic_vector(3 downto 0)) return integer is
variable result : integer:=0;
begin
for i in 0 to 3 loop
if value(i)='1' then
result:=result+2**i;
else
result:=result;
end if;
end loop;
return result;
end std_logic2int;
```

--4-bit iki sembolün GF aritmetiğine göre çarpımını gerçekleştiren fonksiyon

```
function mlt_of_vector(vector1, vector2 : std_logic_vector(3 downto 0))return
std_logic_vector is
variable result : std_logic_vector(3 downto 0):="0000";
begin
    if vector1(0)='1' then
        result(0):=vector2(0);
        result(1):=vector2(1);
        result(2):=vector2(2);
        result(3):=vector2(3);
    end if;
    if vector1(1)='1' then
        result(0):=result(0) xor vector2(3);
        result(1):=result(1) xor vector2(0) xor vector2(3);
        result(2):=result(2) xor vector2(1);
        result(3):=result(3) xor vector2(2);
    end if;
    if vector1(2)='1' then
        result(0):=result(0) xor vector2(2);
        result(1):=result(1) xor vector2(2) xor vector2(3);
        result(2):=result(2) xor vector2(0) xor vector2(3);
        result(3):=result(3) xor vector2(1);
    end if;
    if vector1(3)='1' then
        result(0):=result(0) xor vector2(1);
        result(1):=result(1) xor vector2(1) xor vector2(2);
        result(2):=result(2) xor vector2(2) xor vector2(3);
        result(3):=result(3) xor vector2(0) xor vector2(3);
    end if;
return result;
end mlt_of_vector;
```

--4-bit sembolün GF aritmetiğine göre tersini bulan fonksiyon

```
function inv_of_vector(vector : std_logic_vector(3 downto 0))return
std_logic_vector is
variable result : std_logic_vector(3 downto 0);
begin
result:="0001";
    if vector="0010" then
        result := "1001";
    elsif vector="0100" then
        result:="1101";
    elsif vector="1000" then
        result:="1111";
    elsif vector="0011" then
        result:="1110";
    end if;
```

```
elseif vector="0110" then
    result:="0111";
elseif vector="1100" then
    result:="1010";
elseif vector="1011" then
    result:="0101";
elseif vector="0101" then
    result:="1011";
elseif vector="1010" then
    result:="1100";
elseif vector="0111" then
    result:="0110";
elseif vector="1110" then
    result:="0011";
elseif vector="1111" then
    result:="1000";
elseif vector="1101" then
    result:="0100";
elseif vector="1001" then
    result:="0010";
end if;
return result;
end inv_of_vector;
end mypack;
```

EK-2 Kodlayıcı VHDL Kodu

--RS(15,9) Kodlayıcı Bloğu

```
library ieee;
use ieee.std_logic_1164.all;
use work.mypack.all;
```

```
entity simple_rs_encoder is
  port( clk      : in std_logic;
        reset    : in std_logic;
        data_in   : in std_logic_vector (3 downto 0);
        data_out  : out std_logic_vector ( 3 downto 0)
  );
end simple_rs_encoder;
```

```
architecture first of simple_rs_encoder is
```

```
type genpoly is array (0 to 6) of integer;
```

```
-- genpoly üretici polinomun katsayılarını tutan dizidir.
```

```
constant gen_poly_index : genpoly:=(6,9,6,4,14,10,0);
```

```
--parity_symbol kontrol sembollerini tutan dizi olarak tanımlanmıştır.
```

```
type parity_symbol is array (0 to 5) of std_logic_vector(3 downto 0);
```

```
begin
process(clk,reset,data_in)
variable feedback : integer;
variable indice   : integer;
variable indice2   : integer;
variable parity,data : parity_symbol;
variable indice0   : integer;
variable counter   : integer;
```

```
begin
```

```
--reset '1' olduğunda başlangıç değerleri atanır. Kontrol sembollerine başlangıç
--değeri olarak sıfır atanır.
```

```
if reset='1' then
  data_out <="----";
  counter :=0;
  for i in 0 to 5 loop
    parity(i):="0000";
  end loop;
elsif clk'event and clk='1' then
```

--counter 15 değerini alana kadar LFSR' ye gelen semboller alınır (15 sembol)
--counter 15 olduğunda bölme işlemi tamamlanır. Feedback LFSR' nin çıktısıdır.

if counter/=15 then
 indice :=std_logic2int(parity(5));
 feedback:=index_of(indice);

--feedback sıfıra eşit değilken yapılan işlemler :

if feedback/=-1 then
 for i in 5 downto 1 loop
 if gen_poly_index(i)/= -1 then

--feedback ve üretici polinomun katsayıları çarpılırken önce üslerinin toplamı
--alınır ve toplam indice2 değişkeninde saklanır.

 indice2:=gen_poly_index(i)+feedback;

--Üssün mod 15 değeri bulunur.

 if indice2>=15 then
 indice2:=indice2-15;
 end if;

--Üretici polinomun i nci katsayısı sıfır değilse, feedback ile çarpımı ve kaydırma
--işlemi sonucu gelen i-1 nci sembol toplanıp i nci kontrol sembolü bulunur. Sıfır
--ise i nci kontrol sembolü i-1 nci' ye eşittir.

 parity(i):=parity(i-1) xor alpha_to(indice2);
 else
 parity(i):=parity(i-1);
 end if;
end loop;

--Son kontrol sembolü bulunur.

 indice0:=gen_poly_index(0)+feedback;
 if indice0>=15 then
 indice0:=indice0-15;
 end if;
 parity(0):=alpha_to(indice0)xor data_in;

--feedback sıfır olduğunda sadece kaydırma işlemi yapılarak kontrol sembolleri
--yeni değerlerini alır.

else
 for j in 5 downto 1 loop
 parity(j):=parity(j-1);
 end loop;

```

        parity(0):=data_in;
    end if;
-----
--Counter 6 değerini aldıktan sonra kodlayıcıdan mesaj sembolleri sırayla
--gönderilir.
-----
    if counter<6 then
        data(counter):=data_in;
        counter:=counter+1;
    elsif counter>=6 and counter<=14 then
        data_out<=data(0);
        for j in 0 to 4 loop
            data(j):=data(j+1);
        end loop;
        counter:=counter+1;
        data(5):=data_in;
    end if;
-----
--counter 15 olduğunda bölme işlemi tamamlanır, kontrol sembolleri mesaj
--sembollerinin ardından sırayla kaydırılarak gönderilir.
-----
    elsif counter=15 then
        data_out<=parity(5);
        for l in 5 downto 1 loop
            parity(l):=parity(l-1);
        end loop;
    end if;
end if;
end process;
end first;

```

EK-3 Sendrom Değerlerini Hesaplayan Blok VHDL kodu

--RS(15,9) kodlar için kodçözümleme işleminde sendrom değerlerini bulan blok.
--Altı process' ten oluşur. Her bir process' te kanaldan alınan polinom, üretici
--polinomun bir çarpanı ile bölünür. Örneğin s1 process' i için $(x-\alpha)$ ile s2 için $(x-\alpha^2)$
--ile bölme yapılır ve sırayla $(x-\alpha^6)$ ' e kadar devam eder. Counter 14 olduğunda
--bölme işlemi tamamlanır, sendrom değerleri bulunmuş olur ve bu değerler çıktı
--olarak gönderilir. Counter değerinin her saat çevriminde artması için ayrı bir
--process tanımlanmıştır.

```
library ieee;
use ieee.std_logic_1164.all;
use work.mypack.all;

entity syndrome_cal_block is
    port(clk      : in std_logic;
         reset    : in std_logic;
         data_in  : in std_logic_vector(3 downto 0);
         data_out1 : out std_logic_vector(3 downto 0);
         data_out2 : out std_logic_vector(3 downto 0);
         data_out3 : out std_logic_vector(3 downto 0);
         data_out4 : out std_logic_vector(3 downto 0);
         data_out5 : out std_logic_vector(3 downto 0);
         data_out6 : out std_logic_vector(3 downto 0));
end syndrome_cal_block;

architecture first of syndrome_cal_block is
    subtype int_0to40 is integer range 0 to 40;
    signal counter : int_0to40;

    begin
        count: process(clk,reset)

        begin
            if reset='1' then
                counter<=0;
            else
                if clk'event and clk='1' then
                    counter<=counter+1;
                end if;
            end if;
        end process;

        s1: process(data_in,clk,reset)
            variable syndrome1, hold: std_logic_vector(3 downto 0);

            begin
                if reset='1' then
                    syndrome1:="0000";
                    data_out1<="0000";
```

```

else
  if clk'event and clk='1' then
    hold(0):=syndrome1(3) xor data_in(0);
    hold(1):=syndrome1(0) xor syndrome1(3) xor data_in(1);
    hold(2):=syndrome1(1) xor data_in(2);
    hold(3):=syndrome1(2) xor data_in(3);
    syndrome1:=hold;
    if counter=14 then
      data_out1<=syndrome1;
    end if;
  end if;
end if;
end process;

```

```

s2: process(data_in,clk,reset)
  variable syndrome2,hold: std_logic_vector( 3 downto 0);

```

```

begin
  if reset='1' then
    syndrome2:="0000";
    data_out2<="0000";
  else
    if clk'event and clk='1' then
      hold(0):=syndrome2(2) xor data_in(0);
      hold(1):=syndrome2(2) xor syndrome2(3) xor data_in(1);
      hold(2):=syndrome2(0) xor syndrome2(3) xor data_in(2);
      hold(3):=syndrome2(1) xor data_in(3);
      syndrome2:=hold;
      if counter=14 then
        data_out2<=syndrome2;
      end if;
    end if;
  end if;
end process;

```

```

s3: process(data_in,clk,reset)
  variable syndrome3,hold: std_logic_vector( 3 downto 0);

```

```

begin
  if reset='1' then
    syndrome3:="0000";
    data_out3<="0000";
  else
    if clk'event and clk='1' then
      hold(0):=syndrome3(1) xor data_in(0);
      hold(1):=syndrome3(1) xor syndrome3(2) xor data_in(1);
      hold(2):=syndrome3(2) xor syndrome3(3) xor data_in(2);
      hold(3):=syndrome3(0) xor syndrome3(3) xor data_in(3);
      syndrome3:=hold;
      if counter=14 then
        data_out3<=syndrome3;
      end if;
    end if;
  end if;
end process;

```

```

        end if;
    end if;
end if;
end process;

s4: process(data_in,clk,reset)
variable syndrome4,hold: std_logic_vector( 3 downto 0);

begin
if reset='1' then
    syndrome4:="0000";
    data_out4<="0000";
else
    if clk'event and clk='1' then
        hold(0):=syndrome4(0) xor syndrome4(3) xor data_in(0);
        hold(1):=syndrome4(0) xor syndrome4(1) xor syndrome4(3) xor data_in(1);
        hold(2):=syndrome4(1) xor syndrome4(2) xor data_in(2);
        hold(3):=syndrome4(2) xor syndrome4(3) xor data_in(3);
        syndrome4:=hold;
        if counter=14 then
            data_out4<=syndrome4;
        end if;
    end if;
end if;
end process;

s5: process(data_in,clk,reset)
variable syndrome5,hold: std_logic_vector( 3 downto 0);

begin
if reset='1' then
    syndrome5:="0000";
    data_out5<="0000";
else
    if clk'event and clk='1' then
        hold(0):=syndrome5(2) xor syndrome5(3) xor data_in(0);
        hold(1):=syndrome5(0) xor syndrome5(2) xor data_in(1);
        hold(2):=syndrome5(0) xor syndrome5(1) xor syndrome5(3) xor data_in(2);
        hold(3):=syndrome5(1) xor syndrome5(2) xor data_in(3);
        syndrome5:=hold;
        if counter=14 then
            data_out5<=syndrome5;
        end if;
    end if;
end if;
end process;

```

```

s6: process(data_in,clk,reset)
variable syndrome6,hold: std_logic_vector( 3 downto 0);

begin
if reset='1' then
    syndrome6:="0000";
    data_out6<="0000";
else
    if clk'event and clk='1' then
        hold(0):=syndrome6(1) xor syndrome6(2) xor data_in(0);
        hold(1):=syndrome6(1) xor syndrome6(3) xor data_in(1);
        hold(2):=syndrome6(0) xor syndrome6(2) xor data_in(2);
        hold(3):=syndrome6(0) xor syndrome6(1) xor syndrome6(3) xor data_in(3);
        syndrome6:= hold;
        if counter=14 then
            data_out6<=syndrome6;
        end if;
    end if;
end if;
end process;
end first;

```

EK-4 BM Algoritmasına Göre Anahtar Eşitliği Çözen Blok VHDL Kodu

```
library ieee;
use ieee.std_logic_1164.all;
use work.mypack.all;
```

--Anahtar eşitliğinin BM Algoritmasına göre çözüldüğü blok.

```
entity err_plyblock2 is
port ( clk                : in std_logic;
      reset               : in std_logic;
      syndrome1,syndrome2,syndrome3 : in std_logic_vector(3 downto 0);
      syndrome4,syndrome5,syndrome6 : in std_logic_vector(3 downto 0);
      syn1,syn2,syn3      : out std_logic_vector(3 downto 0);
      imp_corr            : out bit;
      no_err              : out bit;
      sigma1,sigma2,sigma3 : out std_logic_vector(3 downto 0);
      deg                 : out integer);
end err_plyblock2;
```

```
architecture first of err_plyblock2 is
type typsigma is array ( 0 to 3) of std_logic_vector(3 downto 0);
type typsyndrome is array (-1 to 8) of std_logic_vector(3 downto 0);
subtype int_1to7 is integer range -1 to 7;
```

```
begin
```

--process a, hata kontrolünün yapıldığı kısımdır. Sendromların sıfıra eşit olup
--olmadığı kontrol edilir. Tüm sendromlar sıfır ise no_err, '1' değerini alır.

```
a: process(clk,reset,syndrome1,syndrome2,syndrome3,syndrome4,
syndrome5,syndrome6)
```

```
begin
  if reset='1' then
    no_err<='1';
    syn1<="0000"; syn2<="0000"; syn3<="0000";
  else
    if clk'event and clk='1' then
      syn1<=syndrome1;
      syn2<=syndrome2;
      syn3<=syndrome3;
      if syndrome1/= "0000" then
        no_err<='0';
      elsif syndrome2/= "0000" then
        no_err<='0';
      elsif syndrome3/= "0000" then
        no_err<='0';
      elsif syndrome4/= "0000" then
```

```

        no_err<='0';
    elsif syndrome5/="0000" then
        no_err<='0';
    elsif syndrome6/="0000" then
        no_err<='0';
    end if;
end if;
end if;
end process a;

```

--process b, BM algoritmasının uyulandığı kısımdır. sigma_p, BM algoritmasının gerçekleşen adımları içinde (p nci adım) uyumsuzluk değeri sıfır olmayan ve p-sigma_p' nin derecesinin maksimum olduğu adımda bulunan hata yeri belirleyici polinomu tutar. d_p, p nci adımdaki uyumsuzluk değerini; d, son olarak gerçekleşen adımda bulunan uyumsuzluk değerini; degree_p, p nci adımda bulunan hata yeri belirleyici polinomun derecesini; degree, son gerçekleşen adımdaki hata yeri belirleyici polinomun derecesini; u_degree_p, en son gerçekleşen adım sayısı ile degree_p arasındaki farkı tutar.

```

b: process( clk,reset,syndrome1,syndrome2,syndrome3,syndrome4,
            syndrome5,syndrome6)
    variable syndrome           : tpsyndrome;
    variable sigma, sigma_p,sigma_hold : tpsigma;
    variable d, d_hold          : std_logic_vector(3 downto 0);
    variable hold1, hold2, hold3 : std_logic_vector(3 downto 0);
    variable u,p                : int_1to7;
    variable d_p                : std_logic_vector (3 downto 0);
    variable degree, degree_p ,degree_hold,u_degree_p : integer;

begin
    if reset='1' then
        sigma1<="0000"; sigma2<="0000"; sigma3<="0000";
        imp_corr<='0';
        u:=0;
        syndrome(-1):="0000"; syndrome(0):="0000";
        degree:=0;
        degree_p:=0; p:=-1; d_p:="0001";
        sigma(0):="0001"; sigma_p(0):="0001";
        sigma(1):="0000"; sigma_p(1):="0000";
        sigma(2):="0000"; sigma_p(2):="0000";
        sigma(3):="0000"; sigma_p(3):="0000";
        u_degree_p:=-1;
    else
        if clk'event and clk='1' then
            if u=0 then
                syndrome(-1):="0000"; syndrome(0):="0000";
                syndrome(1):=syndrome1;
                syndrome(2):=syndrome2;
                syndrome(3):=syndrome3;
                syndrome(4):=syndrome4;
            end if;
        end if;
    end if;
end process b;

```

```
syndrome(5):=syndrome5;  
syndrome(6):=syndrome6;
```

--İlk adımdaki uyumsuzluk değeri birinci sendroma eşittir.

```
d:=syndrome1;  
end if;  
if d/="0000" then
```

-- $\sigma^{(k+1)}(Z) = \sigma^{(k)}(Z) + \Delta^{(k-p)} \Delta^{(p)} Z^{k-p} \sigma^{(p)}(Z)$ formülünden yeni hata yeri belirleyici
--polinomun önce derecesi sonra katsayıları bulunur.

```
degree_hold:=degree;  
if degree_p+u-p>degree then  
    degree:=degree_p+u-p;  
end if;  
if degree>3 then  
    imp_corr<='1';  
end if;
```

-- σ' nın katsayıları değişmeden önce, p nci adım belirlendikten sonra sigma_p
--değişkenine aktarılmak üzere saklanır.

```
sigma_hold(1):=sigma(1);  
sigma_hold(2):=sigma(2);  
sigma_hold(3):=sigma(3);  
hold2:=inv_of_vector(d_p);  
for i in 0 to 3 loop  
    if (u+i-p)<4 then  
        hold1:=mlt_of_vector(d,sigma_p(i));  
        hold3:=mlt_of_vector(hold1,hold2);  
        sigma(i+u-p):=hold3 xor sigma(i+u-p);  
    end if;  
end loop;  
end if;
```

--Yeni uyumsuzluk değeri hesaplanır.

```
d_hold:=d;  
if u<5 then  
    d:=syndrome(u+2);  
    for j in 1 to 3 loop  
        hold1:=mlt_of_vector(syndrome(u+2-j),sigma(j));  
        d:=d xor hold1;  
    end loop;  
end if;
```

--Yeni p değeri bulunur.

```

if u-degree_hold>=u_degree_p then
  p:=u; d_p:=d_hold;
  degree_p:=degree_hold;
  u_degree_p:=u-degree_hold;
  sigma_p(1):=sigma_hold(1);
  sigma_p(2):=sigma_hold(2);
  sigma_p(3):=sigma_hold(3);
end if;
u:=u+1;

```

--u, 6 olduğunda anahtar eşitlik çözülüp hata yeri belirleyici polinom bulunmuş
 --olur ve sonraki blokta kullanılmak üzere katsayıları çıktı olarak gönderilir.

```

if u=6 then
  deg<=degree;
  sigma1<=sigma(1);
  sigma2<=sigma(2);
  sigma3<=sigma(3);
end if;
end if;
end process b;
end first;

```

EK-5 Evirmesiz BM Algoritmasına Göre Anahtar Eşitliği Çözen Blok VHDL Kodu

--Anahtar eşitliğin evirmesiz BM algoritmasına göre çözüldüğü blok.

```
library ieee;
use ieee.std_logic_1164.all;
use work.mypack.all;

entity evirmesiz_BM is
    port (clk,reset          : in std_logic;
          count              : in std_logic_vector(4 downto 0);
          s0,s1,s2,s3,s4,s5  : in std_logic_vector(3 downto 0);
          sigma0, sigma1, sigma 2, sigma 3 : inout std_logic_vector(3 downto 0);
          gamma              : inout std_logic_vector(3 downto 0));
end evirmesiz_BM;

architecture first of evirmesiz_BM is

    type typhold is array (0 to 5) of std_logic_vector(3 downto 0);
    signal b0,b1,b2 : std_logic_vector(3 downto 0);

    begin
        process(clk,reset,s0,s1,s2,s3,s4,s5)
            variable hold : typhold;
            variable var_delta, hold_5 : std_logic_vector(3 downto 0);
            variable k : integer;

            begin
                if reset='1' then
                    gamma<="0001"; k:=0; var_delta:="0000";
                    sigma 0<="0001"; sigma 1<="0000"; sigma 2<="0000"; sigma 3<="0000";
                    b0<="0001"; b1<="0000"; b2<="0000";
                else
                    if clk'event and clk='1' then
                        if count="01111" then
                            hold(0):=s0;
                            hold(1):=s5;
                            hold(2):=s4;
                            hold(3):=s3;
                            hold(4):=s2;
                            hold(5):=s1;
                            var_delta:=mlt_of_vector(hold(0), sigma 0) xor
                                mlt_of_vector(hold(1), sigma 1) xor mlt_of_vector(hold(2), sigma2)
                                xor mlt_of_vector(hold(3), sigma 3);
                        else
                            var_delta:=mlt_of_vector(hold(0), sigma0) xor
                                mlt_of_vector(hold(1), sigma1) xor mlt_of_vector(hold(2), sigma2)
                                xor mlt_of_vector(hold(3), sigma3);
                        end if
                    end if
                end if
            end process
        end
```

```

end if;
sigma0<=mlt_of_vector(gamma, sigma0);
sigma1<=mlt_of_vector(gamma, sigma1) xor mlt_of_vector(var_delta,b0);
sigma2<=mlt_of_vector(gamma, sigma2) xor mlt_of_vector(var_delta,b1);
sigma3<=mlt_of_vector(gamma, sigma3) xor mlt_of_vector(var_delta,b2);
if var_delta/="0000" and k>=0 then
    b0<= sigma0;
    b1<= sigma1;
    b2<= sigma2;
    gamma<=var_delta;
    k:=-k-1;
else
    b0<="0000";
    b1<=b0;
    b2<=b1;
    gamma<=gamma;
    k:=k+1;
end if;
hold_5:=hold(5);
for i in 5 downto 1 loop
    hold(i):=hold(i-1);
end loop;
hold(0):=hold_5;
end if;
end if;
end process;
end first;

```

Ek-6 Yeni Evirmesiz BM Algoritmasına Göre Anahtar Eşitliği Çözen Blok VHDL Kodu

--Anahtar eşitliğin yeni evirmesiz BM algoritmasına göre çözüldüğü blok. 6 ayrı
--bileşenden oluşmuştur. mycounter, diğer bileşenlerle new_errply_block
--arasındaki zamanlamayı ayarlar. pew ve pew0, hata değeri belirleyici polinomun
--katsayılarının bulunduğu bileşenlerdir. pes0, pes ve pes3 hata yeri belirleyici
--polinomun katsayılarının bulunduğu bileşenlerdir.

```
library ieee;
use ieee.std_logic_1164.all;
use work.mypack.all;

entity new_errply_block is
    port ( clk,reset          : in std_logic;
          count              : in std_logic_vector(4 downto 0);
          s0,s1,s2,s3,s4,s5  : in std_logic_vector(3 downto 0);
          sigma0,sigma1,sigma2,sigma3 : inout std_logic_vector(3 downto 0);
          w0,w1,w2,w3,w4,w5  : out std_logic_vector(3 downto 0));
end new_errply_block;

architecture first of new_errply_block is
    type typz1 is array (1 to 7) of std_logic_vector(3 downto 0);

    component pew
        port(clk,reset : in std_logic; count : in std_logic_vector(4 downto 0);
             s0,s_teta,s_delta,delta_in,gamma,delta0: in std_logic_vector(3 downto 0);
             mc : in bit; delta_out : out std_logic_vector(3 downto 0);
             w_out : out std_logic_vector(3 downto 0));
    end component;

    component pew0
        port(clk,reset : in std_logic; count : in std_logic_vector(4 downto 0);
             s0,s1,delta_in : in std_logic_vector(3 downto 0); mc : inout bit;
             delta0_out,gamma_out, w_out : out std_logic_vector(3 downto 0));
    end component;

    component pes
        port(clk,reset : in std_logic; count : in std_logic_vector(4 downto 0);lambda :
             inout std_logic_vector(3 downto 0); s0,gamma, delta0,b_in : in
             std_logic_vector(3 downto 0); mc : in bit;
             b_out : out std_logic_vector(3 downto 0));
    end component;

    component pes0
        port(clk,reset : in std_logic; count : in std_logic_vector(4 downto 0);lambda :
             inout std_logic_vector(3 downto 0); s0,b_in,gamma,delta0:in
```

```

        std_logic_vector(3 downto 0); mc : in bit; b_out : out std_logic_vector(3
        downto 0));
end component;

component pes3
    port(clk,reset : in std_logic; lambda : inout std_logic_vector(3 downto 0);
        b_in,gamma_in,delta0_in,s0 : in std_logic_vector(3 downto 0);
        count : in std_logic_vector(4 downto 0));
end component;

signal z1,z2: typz1;
signal mcr : bit;

begin
    z2(4)<="0000";
    u5: pew    port map(clk,reset,count,s0,s5,z2(4),z2(4),z1(2),z1(1),mcr,z1(3),w5);
    u4: pew    port map(clk,reset,count,s0,s4,s5,z1(3),z1(2),z1(1),mcr,z1(4),w4);
    u3: pew    port map(clk,reset,count,s0,s3,s4,z1(4),z1(2),z1(1),mcr,z1(5),w3);
    u2: pew    port map(clk,reset,count,s0,s2,s3,z1(5),z1(2),z1(1),mcr,z1(6),w2);
    u1: pew    port map(clk,reset,count,s0,s1,s2,z1(6),z1(2),z1(1),mcr,z1(7),w1);
    u0: pew0    port map(clk,reset,count,s0,s1,z1(7),mcr,z1(1),z1(2),w0);
    p0: pes0    port map(clk,reset,count,sigma0,s0,z2(4),z1(2),z1(1),mcr,z2(1));
    p1: pes     port map(clk,reset,count,sigma1,s0,z1(2),z1(1),z2(1),mcr,z2(2));
    p2: pes     port map(clk,reset,count,sigma2,s0,z1(2),z1(1),z2(2),mcr,z2(3));
    p3: pes3    port map(clk,reset,sigma3,z2(3),z1(2),z1(1),s0,count);
end first;

```

--hata değeri belirleyici polinomun katsayıları (W_i^{k+1} $i \neq 0$) hesaplandığı bloktur.

```

library ieee;
use ieee.std_logic_1164.all;
use work.mypack.all;

entity pew is
    port ( clk,reset                : in std_logic;
          count                    : in std_logic_vector(4 downto 0);
          s0,s_teta,s_delta,delta_in,gamma,delta : in std_logic_vector(3 downto 0);
          mc                       : in bit;
          delta_out                 : out std_logic_vector(3downto 0);
          w_out                    : out std_logic_vector(3 downto 0)
        );
end pew;

architecture first of pew is
    signal teta : std_logic_vector(3 downto 0);

    begin
        process(clk)
        begin
            if reset ='1' then

```

```

w_out<="0000"; delta_out<="0000"; teta<="0000";
else
  if clk'event and clk='1' then

```

```

--count 15 olduğunda sendromlar gelir ve formülde başlangıç değerleri kullanılır.

```

```

  if count="01111" then
    delta_out<=mlt_of_vector(s_delta,"0001") xor mlt_of_vector(s0,s_teta);
    if s0/"0000" then
      teta<=s_delta;
    else
      teta<=s_teta;
    end if;
  else
    delta_out<=mlt_of_vector(delta_in,gamma) xor mlt_of_vector(delta0,teta);
    if count="10100" then w_out<=mlt_of_vector(delta_in,gamma) xor
      mlt_of_vector(delta0,teta);
    end if;
    if mc='0' then
      teta<=teta;
    else
      teta<=delta_in;
    end if;
  end if;
end if;
end process;
end first;

```

```

--Hata değeri belirleyici polinomun ilk katsayısının ( $Y\Delta_0^{(k+1)}$  'in) yani uyumsuzluk
--değerinin ayrıca  $cs^{k+1}$  'in hesaplandığı bloktur. Uyumsuzluk değeri hata değeri
--belirleyici polinomun diğer katsayılarının hesaplandığı bloklara iletilir.

```

```

library ieee;
use ieee.std_logic_1164.all;
use work.mypack.all;

```

```

entity pew0 is
  port ( clk,reset          : in std_logic;
         count              : in std_logic_vector(4 downto 0);
         s0,s1,delta_in     : in std_logic_vector(3 downto 0);
         mc                 : inout bit;
         delta0_out,gamma_out,w_out : out std_logic_vector(3 downto 0));
end pew0;

```

```

architecture first of pew0 is
  signal teta : std_logic_vector(3 downto 0);

```

```

begin

```

```

process(clk)
variable k : integer; variable hold,delta0,gamma: std_logic_vector(3 downto 0);
variable mcr: bit;
begin
if reset ='1' then
    gamma:="0001"; k:=0; hold:="0000"; delta0:="0000";
    w_out<="0000"; gamma_out<="0001"; teta<="0000"; delta0_out<="0000";
    mc<='0';
else
    if clk'event and clk='1' then
        if count="01111" then
            hold:=milt_of_vector(s1,gamma) xor milt_of_vector(s0,s0);
            delta0_out<=hold; delta0:=hold;
            if s0/="0000" then
                teta<=s1; gamma_out<=s0; gamma:=s0; k:=-1;
            else
                teta<=s0; gamma_out<="0001"; gamma:="0001"; k:=1;
            end if;
            if hold/="0000" and k>=0 then
                mc<='1';
            else
                mc<='0';
            end if;
        elsif count>"01111" then
            hold:=milt_of_vector(delta_in,gamma) xor milt_of_vector(delta0,teta);
            if mc='1' then
                gamma_out<=delta0; gamma:=delta0;
                teta<=delta_in; k:=-k-1;
            else
                teta<=teta;
                gamma_out<=gamma; k:=k+1;
            end if;
        end if;
        if hold/="0000" and k>=0 then
            mc<='1';
        else
            mc<='0';
        end if;
        delta0_out<=hold; delta0:=hold;
        if count="10100" then w_out<=hold; end if;
    end if;
end if;
end process;
end first;

```

--Hata yeri belirleyici polinomda σ_0 katsayısının hesaplandığı bloktur. b_0^0 , $B(z)$ 'in
--diğer katsayılarından farklı olarak 0' a değil 1' eşit olduğundan ayrı bir blok olarak
--tanımlanmıştır.

```

library ieee;
use ieee.std_logic_1164.all;
use work.mypack.all;

entity pes0 is
    port ( clk,reset                : in std_logic;
           count                    : in std_logic_vector(4 downto 0);
           lambda                   : inout std_logic_vector(3 downto 0);
           s0,b_in,gamma,delta0    : in std_logic_vector(3 downto 0);
           mc                       : in bit;
           b_out                    : out std_logic_vector(3 downto 0) );
end pes0;

architecture first of pes0 is

begin
    process(clk)
    begin
        if reset='1' then
            b_out<="0001"; lambda<="0001";
        else
            if clk'event and clk='1' then
                if count="01111" then
                    lambda<=mlt_of_vector("0001","0001") xor mlt_of_vector(s0,"0000");
                    if s0/"0000" then
                        b_out<="0001";
                    else
                        b_out<=b_in;
                    end if;
                elsif count>"01111" then
                    lambda<=mlt_of_vector(gamma,lambda) xor mlt_of_vector(delta0,b_in);
                    if mc='0' then
                        b_out<=b_in;
                    else
                        b_out<=lambda;
                    end if;
                end if;
            end if;
        end if;
    end process;
end first;

```

--pes hata yeri belirleyici polinomun katsayılarını bulur. $\sigma_i^{k+1} = \gamma^k \sigma_i^k - Y\Delta_0^k b_{i-1}^k$
--formülüne göre yeni katsayılar bulunur, mc' nin yani cs^k 'nin değerine göre b_i^{k+1}
--değerleir belirlenir.

```
library ieee;
use ieee.std_logic_1164.all;
use work.mypack.all;
```

```
entity pes is
  port ( clk,reset          : in std_logic;
         count              : in std_logic_vector(4 downto 0);
         lambda             : inout std_logic_vector(3 downto 0);
         s0,gamma,delta0,b_in : in std_logic_vector(3 downto 0);
         mc                 : in bit;
         b_out              : out std_logic_vector(3 downto 0));
end pes;
```

architecture first of pes is

```
begin
process(clk)
```

```
begin
if reset='1' then
  b_out<="0000"; lambda<="0000";
else
  if clk'event and clk='1' then
    if count="01111" then
      lambda<=mlt_of_vector("0001",lambda) xor mlt_of_vector(s0,b_in);
      if s0/="0000" then
        b_out<="0000";
      else
        b_out<=b_in;
      end if;
    elsif count>"01111" then
      lambda<=mlt_of_vector(gamma,lambda) xor mlt_of_vector(delta0,b_in);
      if mc='0' then
        b_out<=b_in;
      else
        b_out<=lambda;
      end if;
    end if;
  end if;
end if;
end process;
end first;
```

-- σ_3 ' ün hesaplandığı bloktur. Çıktı olarak b katsayısı iletmesi gerekmediğinden
--farklı bir blok olarak tanımlanmıştır.

```
library ieee;
use ieee.std_logic_1164.all;
use work.mypack.all;

entity pes3 is
  port ( clk,reset                : in std_logic;
         lambda                   : inout std_logic_vector(3 downto 0);
         b_in, gamma_in, delta0_in,s0 : in std_logic_vector(3 downto 0);
         count                    : in std_logic_vector(4 downto 0));
end pes3;

architecture first of pes3 is

begin
process(clk)
begin
if reset='1' then
  lambda<="0000";
else
  if clk'event and clk='1' then
    if count="01111" then
      lambda<=mlt_of_vector("0001", lambda) xor mlt_of_vector(s0, b_in);
    else
      lambda<=mlt_of_vector(gamma_in,lambda)xormlt_of_vector(delta0_in,b_in);
    end if;
  end if;
end if;
end process;
end first;
```

Ek-7 Chien Araştırma Bloğu ve RS kodçözücü blok VHDL kodu

--Chien araştırması ve forney algoritması kullanılarak hata yeri ve değerinin
--hesaplanıp hatanın düzeltildiği bloktur. İki bileşenden oluşur.

```
library ieee;  
use ieee.std_logic_1164.all;  
use work.mypack.all;
```

```
entity csee_block is  
port( clk                      : in std_logic;  
      reset                    : in std_logic;  
      data_in                  : in std_logic_vector(3 downto 0);  
      sigma0,sigma1,sigma2, sigma3 : in std_logic_vector(3 downto 0);  
      w7,w8,w9,w10,w11,w12      : in std_logic_vector(3 downto 0);  
      data_out                  : out std_logic_vector(3 downto 0));  
end csee_block;
```

architecture first of csee_block is

```
component csee_1  
  port(clk,reset : in std_logic; sigma0, sigma1, sigma2, sigma3 : std_logic_vector(3  
    downto 0); control, zXprime_sigma : out std_logic_vector(3 downto 0));  
end component;
```

```
component csee_2  
  port(clk,reset : in std_logic; data_in, zXprime_sigma, control, w7, w8, w9 : in  
    std_logic_vector(3 downto 0); data_out : out std_logic_vector(3 downto 0));  
end component;
```

```
signal ctrl_wire, zXprm_wire : std_logic_vector(3 downto 0);  
begin  
u1:csee_1 port map(clk,reset,sigma0,sigma1,sigma2,sigma3,ctrl_wire,zXprm_wire);  
u2: csee_2 port map(clk,reset,data_in,zXprm_wire,ctrl_wire,w7,w8,w9,data_out);  
end first;
```

--csee_1 blokta sıradaki sembolün yerinin tersi için hata yeri belirleyici polinomun
--değeri ve Forney formülündeki payda değeri hesaplanır.

```
library ieee;  
use ieee.std_logic_1164.all;  
use work.mypack.all;
```

```
entity csee_1 is  
port( clk, reset              : in std_logic;  
      sigma0,sigma1,sigma2,sigma3 : in std_logic_vector(3 downto 0);  
      control                    : out std_logic_vector(3 downto 0);  
      zXprime_sigma              : out std_logic_vector(3 downto 0));  
end csee_1;
```

architecture first of csee_1 is

```
begin
process(clk,sigma0,sigma1,sigma2,sigma3)
variable a1,a2,a3,sgm0_hld : std_logic_vector(3 downto 0);
variable hold1,hold2,hold3 : std_logic_vector(3 downto 0);
variable counter           : integer;
begin
```

```
  if reset='1' then
    counter:=0; a1:="0000"; a2:="0000";a3:="0000";
    control<="0000";
    zXprime_sigma<="0000";
  else
    if clk'event and clk='1' then
```

--counter 21 olduğunda, hata yeri belirleyici polinomun katsayıları bloğa gelir. $\sigma(\alpha)$
--hesaplanır.

```
    if counter=21 then
      a1(0):=sigma1(3);
      a1(1):=sigma1(0) xor sigma1(3);
      a1(2):=sigma1(1);
      a1(3):=sigma1(2);

      a2(0):=sigma2(2);
      a2(1):=sigma2(2) xor sigma2(3);
      a2(2):=sigma2(0) xor sigma2(3);
      a2(3):=sigma2(1);

      a3(0):=sigma3(1);
      a3(1):=sigma3(1) xor sigma3(2);
      a3(2):=sigma3(2) xor sigma3(3);
      a3(3):=sigma3(0) xor sigma3(3);

      sgm0_hld:=sigma0;
    else
      hold1(0):=a1(3);
      hold1(1):=a1(0) xor a1(3);
      hold1(2):=a1(1);
      hold1(3):=a1(2);

      hold2(0):=a2(2);
      hold2(1):=a2(2) xor a2(3);
      hold2(2):=a2(0) xor a2(3);
      hold2(3):=a2(1);
      hold3(0):=a3(1);

      hold3(1):=a3(1) xor a3(2);
      hold3(2):=a3(2) xor a3(3);
      hold3(3):=a3(0) xor a3(3);
```

```

        a1:=hold1; a2:=hold2; a3:=hold3;
    end if;
    counter:=counter+1;
    control<=sgm0_hld xor a1 xor a2 xor a3;
    zXprime_sigma<=a1 xor a3;
end if;
end if;
end process;
end first;

```

```

--csee_2 blokta, forney formülünün pay değeri hesaplanır. csee_1 bloktan gelen
--hata yeri belirleyici polinomun değeri sıfırsa hata değeri hesaplanır, hatalı
--sembole eklenerek hata düzeltilir.

```

```

library ieee;
use ieee.std_logic_1164.all;
use work.mypack.all;

entity csee_2 is
port( clk                : in std_logic;
      reset              : in std_logic;
      data_in, zXprime_sigma : in std_logic_vector(3 downto 0);
      control            : in std_logic_vector(3 downto 0);
      w7,w8,w9           : in std_logic_vector(3 downto 0);
      data_out           : out std_logic_vector(3 downto 0));
end csee_2;

architecture first of csee_2 is

    signal a7,a8,a9      : std_logic_vector(3 downto 0);

begin
    process(clk)
        variable count    : integer;
    begin
        if reset='1' then
            count:=0;
        else
            if clk'event and clk='1' then
                if count=21 then
                    a7(0)<=w7(0) xor w7(1) xor w7(3);
                    a7(1)<=w7(0) xor w7(2) xor w7(3);
                    a7(2)<=w7(1) xor w7(3);
                    a7(3)<=w7(0) xor w7(2);

                    a8(0)<=w8(0) xor w8(2);
                    a8(1)<=w8(1) xor w8(2) xor w8(3);
                    a8(2)<=w8(0) xor w8(2) xor w8(3);
                    a8(3)<=w8(1) xor w8(3);

```

```

a9(0)<=w9(1) xor w9(3);
a9(1)<=w9(0) xor w9(1) xor w9(2) xor w9(3);
a9(2)<=w9(1) xor w9(2) xor w9(3);
a9(3)<=w9(0) xor w9(2) xor w9(3);
else
a7(0)<=a7(0) xor a7(1) xor a7(3);
a7(1)<=a7(0) xor a7(2) xor a7(3);
a7(2)<=a7(1) xor a7(3);
a7(3)<=a7(0) xor a7(2);

a8(0)<=a8(0) xor a8(2);
a8(1)<=a8(1) xor a8(2) xor a8(3);
a8(2)<=a8(0) xor a8(2) xor a8(3);
a8(3)<=a8(1) xor a8(3);

a9(0)<=a9(1) xor a9(3);
a9(1)<=a9(0) xor a9(1) xor a9(2) xor a9(3);
a9(2)<=a9(1) xor a9(2) xor a9(3);
a9(3)<=a9(0) xor a9(2) xor a9(3);
end if;
count:=count+1;
if control="0000" then
data_out<=mlt_of_vector((a7 xor a8 xor a9), inv_of_vector(zXprime_sigma))
xor data_in;
else
data_out<=data_in;
end if;
end if;
end if;
end process;
end first;

```

--RS kodçözücü bileşenleri arasındaki zamanlamayı sağlayan sinyali üretmek için
--kullanılır.

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity mycounter is
port(clk, reset : in std_logic;
count : out std_logic_vector(4 downto 0));
end mycounter;

architecture first of mycounter is

begin
process(clk,reset)
variable hold : unsigned(4 downto 0);

```

```

begin
  if reset='1' then
    hold:="00000"; count<="00000";
  else
    if clk'event and clk='1' then
      hold:=hold + 1;
      count<=std_logic_vector(hold);
    end if;
  end if;
end process;
end first;

```

--Hata kontrolünün yapıldığı bloktur. Sendrom değerlerinin hepsi sıfırsa no_err '1'
 --değerini alır.

```

library ieee;
use ieee.std_logic_1164.all;
use work.mypack.all;

entity err_control is
  port( clk,reset      : in std_logic;
        s1,s2,s3,s4,s5,s6 : in std_logic_vector(3 downto 0);
        no_err          : out bit);
end err_control;

architecture first of err_control is
  type typhold is array (1 to 6) of std_logic_vector(3 downto 0);
begin

  process(clk)
    variable hold : typhold;

  begin
    if reset='1' then
      no_err<='0';
    else
      if clk'event and clk='1' then
        hold(1):=s1;
        hold(2):=s2;
        hold(3):=s3;
        hold(4):=s4;
        hold(5):=s5;
        hold(6):=s6;
        for i in 1 to 6 loop
          if hold(i)/="0000" then
            no_err<='0';
          else
            no_err<='1';
          end if;
        end loop;
      end if;
    end if;
  end process;
end first;

```

```

        end loop;
    end if;
end if;
end process;
end first;

```

--shift_reg, kanaldan gelen sembollerin, hata değeri ve hata yerleri
--hesaplanıncaya kadar geciktirilmesini sağlayan bloktur.

```

library ieee;
use ieee.std_logic_1164.all;
use work.mypack.all;

entity shift_reg is
    port(clk,reset : in std_logic;
          data_in   : in std_logic_vector(3 downto 0);
          data_out  : out std_logic_vector(3 downto 0));
end shift_reg;

architecture first of shift_reg is
    type typ_shftrg is array (0 to 20) of std_logic_vector(3 downto 0);

    begin
        process(clk,reset,data_in)
            variable shftrg : typ_shftrg;

            begin
                if reset='1' then
                    for i in 0 to 20 loop
                        shftrg(i):="0000";
                    end loop;
                else
                    if clk'event and clk='1' then
                        data_out<=shftrg(20);
                        for j in 20 downto 1 loop
                            shftrg(j):=shftrg(j-1);
                        end loop;
                        shftrg(0):=data_in;
                    end if;
                end if;
            end process;
        end first;
    end first;

```

--RS(15,9) Kodçözücü bloğu

```
library ieee;
use ieee.std_logic_1164.all;
use work.mypack.all;
```

```
entity New_RS_Decoder is
  port(clk,reset : in std_logic;
        data_in : in std_logic_vector(3 downto 0);
        no_err : out bit;
        data_out : out std_logic_vector(3 downto 0));
end New_RS_Decoder;
```

architecture first of New_RS_Decoder is

```
component mycounter
  port(clk,reset : in std_logic; count : out std_logic_vector(4 downto 0));
end component;
```

```
component err_control
  port(clk,reset : in std_logic; s1, s2, s3, s4,s5,s6 : in std_logic_vector(3 downto 0);
        no_err : out bit);
end component;
```

```
component shift_reg
  port(clk,reset : in std_logic; data_in : in std_logic_vector(3 downto 0); data_out :
        out std_logic_vector(3 downto 0));
end component;
```

```
component syndrome_cal_block
  port(clk, reset : in std_logic; data_in : in std_logic_vector(3 downto 0);
        data_out1, data_out2, data_out3, data_out4, data_out5, data_out6 : out
        std_logic_vector(3 downto 0));
end component;
```

```
component new_errply_block
  port(clk,reset : in std_logic; count : in std_logic_vector(4 downto 0);
        s0,s1,s2,s3,s4,s5 : in std_logic_vector(3 downto 0); sigma0, sigma1, sigma2,
        sigma3 : inout std_logic_vector(3 downto 0); w0, w1, w2, w3, w4, w5 : out
        std_logic_vector(3 downto 0));
end component;
```

```
component csee_block
  port(clk,reset : in std_logic; data_in, sigma0, sigma1, sigma2, sigma3 : in
        std_logic_vector(3 downto 0); w7, w8, w9, w10, w11, w12 : in
        std_logic_vector(3 downto 0); data_out : out std_logic_vector(3 downto 0));
end component;
```

```
signal sgnl_cnt : std_logic_vector(4 downto 0);
signal sftgr_out : std_logic_vector(3 downto 0);
```

```

signal w7,w8,w9,w10,w11,w12      : std_logic_vector(3 downto 0);
signal syn0,syn1,syn2,syn3,syn4,syn5 : std_logic_vector(3 downto 0);
signal sgm0,sgm1,sgm2,sgm3       : std_logic_vector(3 downto 0);

begin

u0:mycounter port map(clk,reset,sgnl_cnt);
u1:err_control port map(clk,reset,syn0,syn1,syn2,syn3,syn4,syn5,no_err);
u2:shift_reg port map(clk,reset,data_in,sftrg_out);
u3:syndrome_cal_block port map(clk, reset, data_in, syn0, syn1, syn2, syn3, syn4,
    syn5);
u4:new_errply_block port map(clk,reset,sgnl_cnt,syn0,syn1,syn2,syn3,syn4, syn5,
    sgm0,sgm1,sgm2,sgm3,w7,w8,w9,w10,w11,w12);
u5:csee_block port map(clk, reset,sftrg_out, sgm0, sgm1, sgm2, sgm3, w7, w8,
    w9, w10,w11,w12,data_out);
end first;

```

ÖZGEÇMİŞ

Adı Soyadı : Lügen ÜNAL

Doğum Yeri : Ankara

Doğum Yılı : 1980

Medeni Hali : Bekar

Eğitim ve Akademik Durum

Lise : Özel Yüce Fen Lisesi- Ankara /1994-1997

Lisans : Başkent Üniversitesi / 1997-2002

Yabancı Dil : İngilizce

İş Tecrübesi

Milli Savunma Bakanlığı / 2003-

