FPGA ile Gerçeklenen Devrelerde Hataya Dayanıklılık

Tolga Ayav[1], *Kadir Atilla Toker*[2]

[1]Bilgisayar Mühendisliği Bölümü
İzmir Yüksek Teknoloji Enstitüsü, Urla - İzmir
tolgaayav@iyte.edu.tr

[2]Kablosuz Haberleşme Ağları ve Çoklu Ortam Araştırma Merkezi
İzmir Yüksek Teknoloji Enstitüsü, Urla - İzmir
atillatoker@iyte.edu.tr

Özetçe

Hataya dayanıklılık (Fault-Tolerance) özellikle güvenliğin önemli olduğu kritik uygulamalarda ve kontrol sistemlerinde büyük önem taşımaktadır. Bu çalışmada FPGA ile gerçeklenen devrelerin otomatik olarak aksaklığa dayanıklı hale dönüştürülmesi için geliştirilen bir teknik sunulmaktadır. VHDL dilinde yazılmış olan devre kodu, göreceli kısa bir dönüşüm betiği ile otomatik olarak hataya dayanıklı hale getirilmektedir. Önerilen metot, FPGA üzerinde gerçeklenmiş bir sonsuz darbe cevaplı (IIR) süzgeç ile sınanarak benzetim sonuçları verilmekte, kullanılan tekniğin basitliği ve buna karşın önemi ve etkinliği sunulmaktadır.

1. Giriş

Hataya dayanıklılık olası donanım ve yazılım hatalarına karşı sistemin bağışıklı olmasıdır ve kritik sistemlerde vazgeçilmez bir özelliktir [1]. Cihazların, güç kaynaklarının ve entegrelerin küçültülmesi ve çalışma hızının artırılması devrelerin gürültü marjinlerinin azalmasına yol açmakta, bu da cihaz içerisinden kaynaklanan elektriksel gürültülerin en az dışarıdan gelenler kadar önemli olmasına yol açmaktadır. Bu sebeple hataya dayanıklılık konusu özellikle radyasyonun çok etkili olduğu uzay uygulamalarının yanısıra gelecek nesil tüm yeryüzü cihazlarında da önemli bir yer oluşturacaktır.

Yongalarda çalışma zamanında oluşabilecek hatalar, günümüz sayısal sistemlerinin vazgeçilmezlerinden biri olan FPGA'leri (Field Programmable Gate Array), özellikle de uygulama geliştirme ve kullanımdaki esnekliğinden dolayı tercih edilen statik rasgele erişimli bellek tabanlı FPGA'leri (SRAM-based FPGA) olumsuz etkilemektedir.

Bu çalışmada önerilen kod dönüştürme tekniğiyle VHDL dilinde yazılmış olan FPGA kodlarının otomatik olarak hataya dayanıklı hale getirilmesi hedeflenmiştir. Otomatik kod dönüşümü sayesinde kod geliştirici işlevsel devre koduna konsantre olarak devresini tasarlayabilecek, daha sonra kendisi veya başkası tarafından yazılan kısa dönüşüm betikleri kullanılarak kod kolaylıkla hataya dayanıklı hale dönüştürülebilecektir. Hataya dayanıklılığın yedeklilik prensibine dayanması nedeniyle, hataya dayanıklı kod genellikle işlevsel koddan bir kaç kat daha büyük olur. Bu sebeple otomatik kod dönüşümü modüler kod geliştirmenin yanısıra, kodlamanın daha hızlı olmasını da sağlar. Kod dönüşüm betikleriyle tanımlanan hataya dayanıklılık kısmının işlevsel koddan ayrı olması daha sonra bunlarda yapılacak değişikliklerin birbirlerini etkilememesini sağlar.

Çalışmanın organizasyonu şu şekildedir. Bölüm 2'de hataya dayanıklılık konusunda genel bir bilgi verilmektedir. Bir sonraki bölümde ise önerilen metodu üzerinde gösterdiğimiz bir örnek uygulama olan IIR süzgeç sunulmaktadır. Otomatik kod dönüştürmeyle ilgili bilgiler ve örnek dönüştürme betikleri bölüm 4'de verilmektedir. Son olarak bölüm 5 ile çalışma sonlandırılmaktadır.

2. Hataya Dayanıklılık

Hataya dayanıklılık, bir sistemin olası donanım ve yazılım hatalarında bile kendisinden beklenen performans ve sonuçları verebilmesi olarak tanımlanabilir. Hataya dayanıklı sistemler [2], [4], [7] ve [9]'da detaylı olarak anlatılmaktadır. Bu kısımda hataya dayanıklılıkla ilgili yapılan çalışmaya dayanak oluşturacak bazı temel bilgiler verilecektir.

Hataya dayanıklılıkta aksaklık çıkış noktasını teşkil eder ve sıklıkla üç farklı terimle ifade edilir: *bozukluk (fault)*, *hata (error)* ve *arıza (failure)*. *Bozukluk* donanım veya yazılımda elektriksel gürültü, radyasyon veya programcı hatasından kaynaklanan istenmeyen bir fiziksel değişimi ifade etmektedir. Örneğin mikroişlemcinin kayıtçılarından birinde oluşan bir bit değişikliği bozukluk olarak adlandırılır. Bu bit değişikliği herhangi bir etki yaratmayabileceği gibi kendini bir *hata* olarak da gösterebilir. Bir kayıtçının değişen değerinden ötürü bir uçağın yüksekliğinin hatalı ayarlanması buna örnek olarak gösterilebilir. Hatalı ayarlanan yükseklik bir sorun çıkarmayabileceği gibi uçağın bir engele çarpmasıyla sonuçlanması gibi geri dönülmesi imkansız bir duruma neden olduğunda bu *arıza* olarak adlandırılır. Herhangi bir komponentin arızası bağlı olduğu sistem açısından bir bozukluk olacağından yine benzer sırayla bu sistemin hatasına ardında da arızasına sebep olur. Bu sebeple sözkonusu kavramlar sıklıkla

aşağıda gösterilen hataya dayanıklılık zinciri ile ifade edilir.

$$\cdots \rightarrow \text{Bozukluk} \rightarrow \text{Hata} \rightarrow \text{Arıza} \rightarrow \text{Bozukluk} \rightarrow \cdots$$

Bozukluk, süresi, yapısı ve kaynağı bakımından farklı sınıflandırılır. Süresi bakımından geçici, aralıklı ve kalıcı bozukluk olarak üç farklı biçimde olabilir [7] [9]. Geçici bozukluk genellikle gürültü, radyasyon gibi çevresel etkilerden kaynaklanır, verilerin bozulmasına yol açar, etkisi kısa sürede ortadan kalkar ve tekrar etmeyen bir yapıya sahiptir. Bu tür bozukluklar literatürde SEU (Single Event Upset) adıyla anılmaktadır ve bozukluğun tekrarlanma oranı SER (Soft Error Rate) olarak adlandırılmaktadır. Geçici bozukluklar günümüz teknolojisinde üretilen cihazlarda çok önemli bir yere sahiptir. Aralıklı bozukluk, cihazın kısa aralıklarla arızalanıp düzelmesine yol açan bir bozukluk olup, genelde kararsız çalışan bir bileşenden kaynaklanır. Kalıcı bozukluk ise bir bileşenin arızalanması, fiziksel bir hasar veya tasarım hatalarından kaynaklanan, zaman içerisinde düzelmeyen bozukluklardır. Geçici ve aralıklı bozukluklar kalıcı bozukluklardan daha sık oluşurlar, bir hata oluşturup ortadan kalktıkları için de tespit edilmesi daha zordur. Pratikte bozuklukların %80'inden fazlasının geçici ve aralıklı bozukluklar olduğu saptanmıştır. Bozuklukların yapısı ve kaynaklarına göre yapılan diğer sınıflandırmaları için [7]'ye başvurulabilir.

Hataya dayanıklı sistem, yedeklilik (redundancy) sayesinde gerçekleştirilir. Yedeklilik temelde aşağıdaki gibi sınıflandırılabilir [4]:

1. Fiziksel/mekansal yedeklilik
2. Zamansal yedeklilik

Bunlardan ilki olan fiziksel yedeklilik, donanıma veya yazılıma yedeklilik kazandırılması prensibine dayanır. Donanımda fiziksel yedeklilik üç kısımda incelenebilir:

Pasif yedekleme NMR (N-Modular Redundancy) olarak adlandırılan teknik bu çalışmada da kullanılmıştır. 2 veya daha fazla modül aynı anda çalıştırılarak sonuçları karşılaştırılır ve genellikle çoğunluğa dayalı oylama ile sonuç belirlenir.

Aktif yedekleme Yedekte bekleyen modülün, arızanın tespit edilmesiyle birlikte arızalanan modülün yerini alması esasına dayanmaktadır [11].

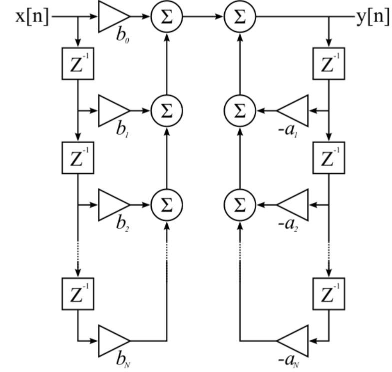
Karma yedekleme Yedekte bekleyen modüllerle birlikte NMR'nin uygulanması esasına dayanır. Örneğin 5MR sadece iki arızalı modülü tolere edebilirken, karma olan 3MR+2 yedek aynı anda 3 arızayı tolere edebilir.

Yazılımdaki fiziksel yedeklilik çalışmanın konusuna uzak olması nedeniyle burada incelenmeyecek olup, denetim noktaları oluşturma ve geriye dönüş (checkpointing and rollback), toparlama blokları (recovery blocks), N-versiyon programlama (N-version programming) gibi teknikleri içermektedir [11].

Zamansal yedeklilik ise herhangi bir görevin tekrarlanması için yedek zaman ayrılması esasına dayanır. Aynı programın veya bir parçasının üst üste birden fazla kez çalıştırılıp sonuçlarının oylanması, ağ üzerinden gönderilen verinin bir kaç kez gönderilmesi bu yedekliliğe örnek verilebilir.

3. Örnek Uygulama: Sonsuz Darbe Cevaplı Süzgeç

Bu kısımda sayısal sinyal işlemede sıklıkla kullanılan sonsuz darbe cevaplı (Infinite Impulse Response - IIR) süzgeç örnek olarak ele alınacaktır. IIR süzgeç FPGA üzerinde gerçeklenerek, çalışma zamanında FPGA üzerinde oluşabilecek bozuklukların sisteme ne şekilde etkidiği incelenecektir. Bir sonraki kısımda da, önerdiğimiz hataya dayanıklılık metodunun uygulanması ve bunun sonuçları incelenecektir. IIR süzgecin genel yapısı ve sayısal sistemlerdeki uygulaması şekil 1 ve formül 1 ile verilmiştir.



Şekil 1: Sonsuz darbe cevaplı (Direct Form I) sayısal süzgecin blok diyagramı.

$$\begin{aligned} y(n) &= b_0x(n) + b_1x(n-1) + \ldots + b_Nx(n-N) \quad (1) \\ & \quad a_1y(n-1) + a_2y(n-2) + \ldots + a_Ny(n-N) \end{aligned}$$

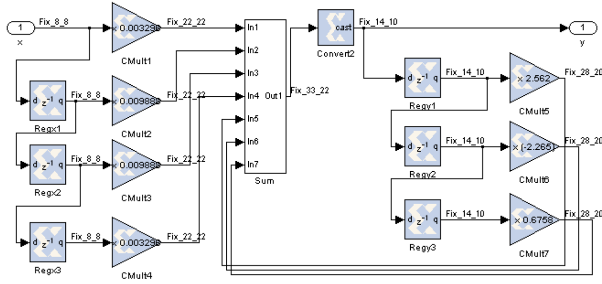
Bu çalışmada örnek olarak seçtiğimiz 3. derece alçak geçiren bir IIR süzgecin formülü 4'de verilmektedir. Süzgecin parametrelerinin hesabıyla ilgili burada bilgi verilmeyecek olup daha ayrıntılı bilgi için okuyucu [13]'e başvurabilir.

$$\begin{aligned} y(n) &= 0.0033x(n) + 0.0099x(n-1) + 0.0099x(n-2) \\ & \quad +0.0033x(n-3) + 2.5618y(n-1) \\ & \quad +2.2643y(n-2) + 0.6762y(n-3) \end{aligned} \quad (2)$$

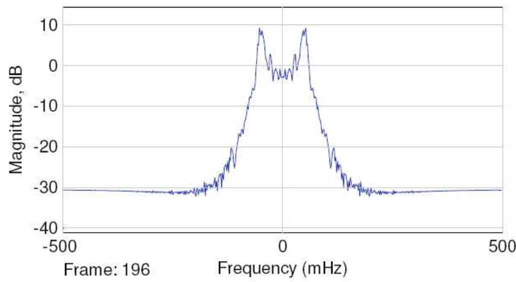
Formülden de görüldüğü gibi, süzgecin çıkışı hem girişin hem de çıkışın önceki değerlerine bağlıdır. Süzgeç gerçeklenirken önceki değerleri saklamak için kayıtçılar kullanılır. Bu filtrenin Matlab® ve Simulink® ile gerçeklenmiş diyagramı Şekil 2'de görülmektedir. Şekildeki 6 adet $[d\ z^{-1}\ q]$ bloğu kayıtçıları belirtmektedir.

Süzgecin karakteristiği (frekans cevabı) ise Şekil 3'de verilmektedir.

FPGA (Field Programmable Gate Array), PLD (Programmable Logic Device) ailesinin gelişmiş bir versiyonudur ve günümüzde sayısal devre tasarımında çok yaygın olarak kullanılmaktadır. FPGA'lar karmaşık programlanabilir mantık devreleridir. Örneğin Altera'nın FLEX10K250 yongası 250,000 mantık kapısı, 40,960 bit RAM içermektedir.



Şekil 2: 3. Derece IIR süzgecin blok diyagramı



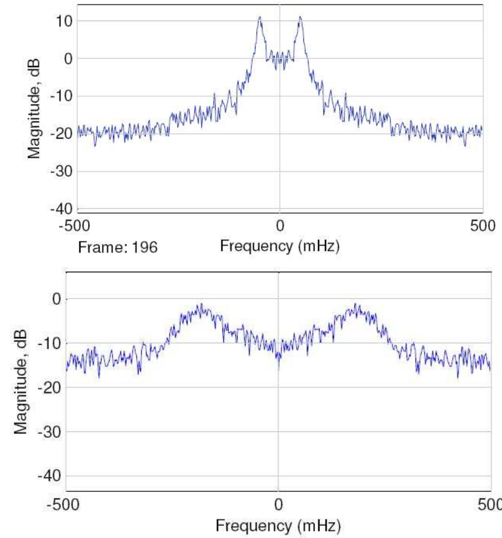
Şekil 3: Örnek IIR süzgecin karakteristiği

FPGA yongaları üzerinde sayaçlar, ALU ve sonlu durum makinalarının yanısıra bellek fonksiyonları, mikroişlemci ve sayısal sinyal işleme gibi karmaşık mantıksal fonksiyonların uygulaması mümkündür [3].

FPGA ile tasarımda artık günümüzde standart haline gelmiş olan ve yaygın olarak kullanılan metot, gerçeklenecek donanımı VHDL (**V**ery high speed integrated circuit **H**ardware **D**escription **L**anguage) ile kodlamaktır. Kod, bir derleyici ve sentezleyici yardımıyla "netlist" adı verilen bir forma dönüştürüldükten sonra yongaya yüklenir. VHDL ile devre sentezleme konusunda detaylı bilgi için [10]'a başvurulabilir. Şekil 5'de sözkonusu süzgecin VHDL kodu yer darlığı sebebiyle kısmi olarak verilmektedir.

IIR süzgeçler, dahili bir geri beslemeye sahip olması sebebiyle FIR (Finite Impulse Response) süzgeçlerden farklı olarak kararlılık problemiyle karşı karşıyadır. Bu nedenle kayıtçılarda meydana gelebilecek bozulmalar süzgecin kararsız çalışmasına neden olabilir. Şekil 4 süzgecin kayıtçılarında oluşan geçici bozukluklar nedeniyle karakteristiklerinin ne şekilde değişebildiğini göstermektedir. Bozukluklar sadece Regy1 kayıtçısında (Bknz. şekil 2) oluşturulmuş, kayıtçının herhangi bir bitinde iki farklı sıklıkta hata oluşturularak süzgecin karakteristikleri çıkartılmıştır. Şekilden de görüldüğü gibi, tek bir kayıtçıdaki bir bitlik hata bile süzgecin yapısını bozmaya yetmektedir.

Takip eden kısımlarda IIR ve benzeri sayısal devreleri hataya dayanıklı hale getirmek için yaygın olarak kullanılan 3-modüler yedekleme tekniği ve bu çalışmada önerdiğimiz otomatik kod dönüştürme tekniği sunulacaktır.



Şekil 4: Kayıtçılarda hata oluştuğu durumdaki süzgeç karakteristikleri. Üstteki şekil sadece Regy1 kayıtçısında 100 örnekleme periyodunda ortalama 1 hata oluştuğu durumu, alttaki şekilse 20 hata oluştuğu durumu göstermektedir.)

4. Otomatik Kod Dönüşümü

4.1. Kod Dönüşümünün Temel Prensipleri

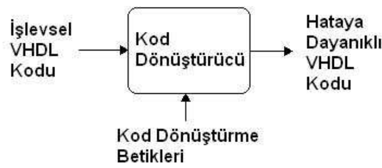
Otomatik kod dönüşümü, işlevsel kodun programa işlevsel olmayan özellikleri kazandırmak amacıyla dönüştürülmesi işlemidir. Bu dönüşüm, programcının belirlediği parametreleri veya yazdığı betikleri giriş olarak kabul eden otomatik dönüşüm araçları ile yapılır. Kod dönüşümü tasarım sırasında kaynak veya hedef kod üzerinde yapılabileceği gibi (statik kod dönüşümü), çalışma zamanında bir izleyicinin yardımıyla dinamik olarak da gerçekleştirilebilir [8][5][6]. Örneğin, IIR devresini FPGA üzerinde gerçeklemek için yazılan VHDL kodu işlevseldir. Bu devreyi hataya dayanıklı hale getirmek için işlevsel kodun üzerinde bazı değişiklikler yapmak gereklidir. Sözkonusu değişiklikler kodun işlevselliğinden, diğer bir deyişle fonksiyonundan bağımsız olup belirli bir sistematiğe bağlıdır ve kod üzerinde bir çok yerde tekrar edebilir. Dolayısıyla bu işlem otomatize edilebilir ve bir dönüşüm programı yardımıyla gerçekleştirilebilir. Otomatik kod dönüşümü işlevsel olan ve olmayan kodun birbirinden ayrılmasına olanak sağladığı gibi, kod geliştirmeyi de hızlandırır.

Bu çalışmada tanımlanacak olan kod dönüştürücü kullanıcının yazacağı kısa bir dönüştürme betiğini kullanarak VHDL ile geliştirilmiş olan devre kodunu otomatik olarak hataya dayanıklı hale getirmektedir. Görüldüğü gibi, sözkonusu kod dönüştürücü kaynak kod üzerinde işlem yapmakta olup, bir çeşit ön derleyici gibi düşünülebilir. Bu işlemden sonra geliştirmeci normal olarak bir VHDL sentezleyici kullanarak hedef devreyi sentezleyebilecektir. Kod dönüştürücünün işlevi şekil 6 ile gösterilmiştir.

Kod dönüştürücünün programlama dili aşağıdaki sözdizimine sahiptir:

```
entity iir_struc is
    port ( x:  in signed(15 downto 0);
    clk:  in std_logic;
    y:  out signed(15 downto 0));
end iir_struc;

architecture Behavioral of iir_struc is
   coefficients is array (0 to 3) of signed(15 downto 0);
   signal regx1, regx2, regx3:  signed(15 downto 0);
   signal regy1, regy2, regy3:  signed(15 downto 0);
   signal y_in:  signed(31 downto 0);
   constant a, b:  coefficients;

   component REG is port( input:  in signed(15 downto 0);
      clk:  in std_logic;
      output:  out signed(15 downto 0));
   end component;

begin
   lregx1:  REG port map(x,clk,regx1);
   lregx2:  REG port map(regx1,clk,regx2);
   lregx3:  REG port map(regx2,clk,regx3);

   process (clk)
   variable acc:  signed(31 downto 0) := (others= >'0');
   begin
     if (clk'event and clk= '1') then
       acc := b(0)*x + b(1)*regx0 + b(2)*regx1+
         b(3)*regx2 + a(1)*regy0 + a(2)*regy1 + a(3)*regy2;
     end if;
     y_in <=  acc;
   end process;

   y<= y_in(15 downto 8);
   lregy1:  REG port map(y_in(15 downto 8),clk,regy1);
   lregy2:  REG port map(regy1,clk,regy2);
   lregy3:  REG port map(regy2,clk,regy3);
end Behavioral;
```

Şekil 5: IIR süzgecin VHDL kodu



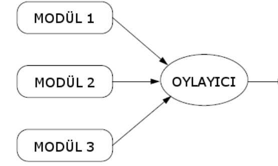
Şekil 6: Kod dönüştürücünün işlevi

$$
\begin{aligned}
\texttt{match}\{\mathtt{S_1}\ \texttt{in}\ \mathtt{S_1'}\} \quad \Rightarrow \quad & \texttt{define}\{\mathtt{S_2}\} \\
& \texttt{replace}\{\mathtt{S_3}\} \\
& \texttt{before}\{\mathtt{S_4}\} \\
& \texttt{after}\{\mathtt{S_5}\}
\end{aligned}
$$

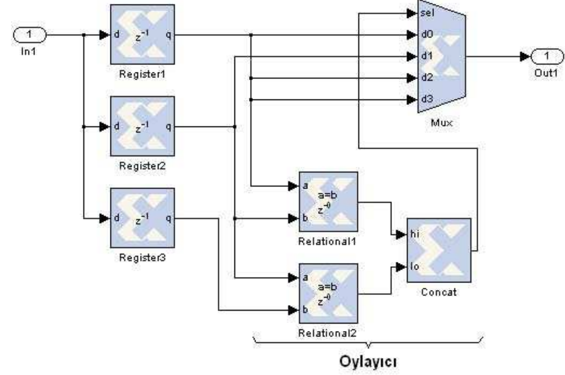
Bu sözdizimine göre, dönüştürücü VHDL kodu üzerinde S_1' içerisinde yer alan S_1 örüntüsünü her yakaladığında, değişken tanımlamalarını içeren S_2 ifadesini değişken tanımlamalarının olduğu kısma yerleştirir ("begin" komutundan hemen önce), S_1 ifadesini S_3 ile yer değiştirir. S_4'ü S_1'in hemen öncesine, S_5'i ise hemen sonrasına yerleştirir. Dönüştürücü, örüntü eşleştirme işlemini kodu başından sonuna doğru bir defa tarayarak tamamlar.

4.2. Kod Dönüştürmenin IIR Üzerinde Uygulanması

Örnek olarak verdiğimiz IIR süzgeci hataya dayanıklı hale getirmek için içerisinde yer alan 6 adet kayıtçı 3-modüler yedekleme (Triple Modular Redundancy - TMR) tekniği ile çoğullanabilir. 3-modüler yedekleme, hataya dayanıklı hale getirilecek modülün Şekil 7'de görüldüğü gibi çoğullanması ve bu modüllerin sonucunun bir oylayıcı yardımıyla karşılaştırılarak olası bir hatanın ortadan



Şekil 7: 3-Modüler Yedekleme Tekniği



Şekil 8: 3-modüler yedekleme tekniğiyle hataya dayanıklı hale getirilmiş bir kayıtçının blok diyagramı

kaldırılmasına dayanmaktadır [12]. Oylayıcıda ortalama hesaplama, çoğunluğu bulma gibi teknikler uygulanabilir. Bu çalışmada çoğunluk hesaplama kullanılmıştır. Çoğunluk hesaplayıcı temelde formül 3 ile hesaplandığı gibi daha farklı şekillerde de bulunabilir.

$$out := (in1\ \texttt{and}\ in2)\ \texttt{or}\ (in1\ \texttt{and}\ in3)\ \texttt{or}\ (in2\ \texttt{and}\ in3) \tag{3}$$

Çoğunluk hesaplayıcı kayıtçılardan birinde bir hata olması durumunda diğer ikisini kullanarak doğru sonucu üretir. İki kayıtçıda bozukluk olması durumunda ise hata kurtarılamaz. Karşılaştırma kayıtçıların tümü üzerinde olabileceği gibi bit bazında da yapılabilir. Kayıtçının ve oylayıcının VHDL kodları yer azlığı sebebiyle bu çalışmada verilmeyecek olup ilgili kodlar [10]'da bulunabilir. Bununla birlikte kayıtçı ve oylayıcı hakkında fikir vermek için şekil 8 verilmiştir. Bu şekilde 3-modüler yedekleme tekniği ile hataya dayanıklı hale getirilen bir kayıtçının Simulink® blok diyagramı görülmektedir.

4.3. Örnek Kod Dönüşüm Betikleri

IIR süzgecin VHDL kodunda yer alan kayıtçılara 3-modüler yedekleme tekniğini uygulamak için şekil 9'da verilen dönüştürücü programını yazabiliriz. Bu programa göre dönüştürücü, kod içerisinde bulduğu her `REG` öğesi için ϑ sinyalini tanımlar, `REG` öğesini ise `replace{...}` kısmında yer alan kod ile değiştirir. Örüntüde yer alan β, α_1, α_2 ve α_3 parametreleri de `replace` bölümüne aktarılır. Burada λ_1 ve λ_2 ise dönüştürücü tarafından üretilen etiketlerdir. Voter, önceden tanımlanmış olan, oylayıcı fonksiyonunu gerçekleştiren modüldür. Bu betiğe göre, kod çevirici şekil 5'de görülen IIR süzgeç kodundaki tüm kayıtçıları şekil 12'de

```
match { β:  *REG port map(α1,α2,α3); } ⇒
define {
  signal ϑ is array(2 downto 0) of type(α3);
}
replace {
   β:  for k in 2 downto 0 generate
          λ1:  REG port map(α1,α2,ϑ(k));
       end generate β;
   λ2:  voter port map(ϑ(0),ϑ(1),ϑ(2),α3);
};
```

Şekil 9: Kod dönüşüm betiği 1

görüldüğü gibi değiştirir. Burada dönüştürülmüş kodun yer darlığı sebebiyle sadece bir kısmı verilmiştir.

VHDL'de temelde iki türlü programlama vardır: 1) Yapısal (Structural) 2) Davranışsal (Behavioral). Süzgeçte yaptığımız kayıtçı tanımlamaları yapısal programlama örneğidir. Görüldüğü gibi altı adet kayıtçı altı satırda ayrı bileşenler olarak belirtilmiş, giriş ve çıkışları önceden tanımlanmış olan sinyaller kullanılarak birbirlerine bağlanmıştır. VHDL programlamada `architecture ... begin` **S** `end` yapısındaki `S` cümlesinde yer alan tüm satırlar FPGA'da paralel çalışacak şekilde gerçeklenir. Diğer bir deyişle `S` cümlesini `S=` $s_1; s_2; s_3; \cdots ; s_n;$ şeklinde düşünürsek, s_1, s_2, ..., ve s_n paralel olarak çalıştırılırlar. 1 numaralı kod dönüşüm programına göre yedeklenen kayıtçıların paralel olarak çalıştığına, dolayısıyla kayıtçılar için kullanılan kaynağın yaklaşık üç katına çıktığına ancak bunun devrenin çalışma hızında veya saat periyodunda bir değişiklik yaratmadığına dikkat edilmelidir. VHDL'de davranışsal programlama ise $s_i =$ `process ... begin` **S'** `end` bloğu kullanılarak yapılır. Bu programlama metodunda, bileşenleri tanımlayıp bunları sinyallerle bağlamak yerine, algoritma yüksek seviyeli bir programlama dilinde olduğu gibi `process` bloklarının içerisinde yazılır. Bu bloklarda ise, `S'=` $s'_1; s'_2; s'_3; \cdots ; s'_n;$ cümlesinde yer alan s'_1, s'_2, ..., ve s'_n satırları ardışıl olarak çalıştırılırlar. Örneğin Şekil 5'da verilen IIR süzgecin VHDL kodunda, kayıtçı tanımlamaları ve `process` bloğu paralel olarak çalışmakta, ancak `process` bloğunda yer alan komutlar kendi içinde ardışıl olarak çalışmaktadır. Diğer bir deyişle, bu bloktaki `y_in <= acc` cümlesi `if`...`end if` cümlesinden sonra çalıştırılacaktır. Sentezlenmek istenen devre VHDL'de tamamıyla yapısal olarak kodlanabileceği gibi, tamamıyla davranışsal veya bizim bu çalışmada yaptığımız gibi karma bir şekilde kodlanabilir.

Şekil 10'da verilen kod dönüşüm betiği ise `process` bloğu içerisinde kullanılan değişkenlere 3-modüler yedekleme tekniğini uygulamaya yöneliktir. Bu dönüşüm programı, kod içerisinde yer alan tüm `process` bloklarını ve içlerinde yer alan değişken ataması yapılan satırları yakalar ve buralardaki atama yapılan değişkenleri yedekler. Betikteki `majority`, formül 3'e göre yazılmış olan bir fonksiyondur. Kısaca açıklamak gerekirse, örüntü eşleştirici `process` bloklarının içerisinde yakaladığı $\alpha := \beta$ şeklindeki her atama için öncelikle α değişkeninin tipine sahip iki adet yeni değişken tanımlar (ϑ_1 ve ϑ_2). $\alpha := \beta$ komutundan hemen sonra bu iki yeni değişkene β'nın atandığı satırları, daha sonra da α, ϑ_1 ve ϑ_2 değişkenleri arasında çoğunluk oylamasının yapıldığı ve sonucun yine α'ya atandığı α:= majority(α,ϑ_1,ϑ_2) satırını ekler. Bu dönüşümün, `process` bloğunun içerisine kod eklendiğinden, çalışma süresini artırdığına dikkat edilmelidir. Bu betiğin sonucu olarak IIR süzgecin kodunda meydana gelen değişiklik şekil 13'de görülmektedir.

```
match { α := β; in process * (*) * ; }⇒
define {
  variable ϑ1, ϑ2:  type(α);
}
after {
   ϑ1  := β;
   ϑ2  := β;
   α:= majority(α,ϑ1,ϑ2);
};
```

Şekil 10: Kod dönüşüm betiği 2

Şekil 11'de verilen dönüşüm betiği ise bir öncekine benzer şekilde, parametrelerinden biri `clk` olan tüm process'leri ve bunların içlerinde yer alan ve sinyal ataması yapılan tüm satırları yakalar ve önce bu sinyalleri yedekler, daha sonra da `process` bloğunun hemen arkasına bu üç sinyalin karşılaştırıldığı oylayıcıyı yerleştirir.

```
match { α <= β; in process * (*,clk,*) * ; }⇒
define {
  signal ϑ1, ϑ2, ϑ2:  type(α);
}
replace {
   ϑ1 <= β;
   ϑ2 <= β;
   ϑ3 <= β;
};
match { process * (*,clk,*) * ; }⇒
after {
   λ1:  voter port map(ϑ1, ϑ2, ϑ2, α);
};
```

Şekil 11: Kod dönüşüm betiği 3

Şekil 12'de verilen kod parçası sadece bir numaralı kod dönüşüm betiğinin süzgecin VHDL kodu üzerinde yaptığı değişikliğin bir kısmını, şekil 13'de verilen kod parçası ise iki numaralı betiğin yarattığı değişikliği göstermektedir. Üç numaralı dönüşüm betiğinin işlevi ise yine bir öncekine çok benzerdir. Bu dönüşüme ait VHDL kodu şekil 14'de görülebilir.

5. Sonuç

Bu çalışmada VHDL dili için bir otomatik kod dönüştürücü tanımlanmıştır. Kod dönüştürme işlemi programa sonradan eklenecek ve işlevsel koddan bağımsız olan kısımları daha etkin, hızlı bir şekilde gerçekleştirmeyi sağlar ve programın modülerliğini de artırır. Sözkonusu kod dönüşümü VHDL ile sentezlenen devrelere hataya dayanıklılık özelliğini

```
  .
  .
  .
signal fregx1 is array(2 downto 0) of signed (15 downto 0);
signal fregx2 is array(2 downto 0) of signed (15 downto 0);
signal fregx3 is array(2 downto 0) of signed (15 downto 0);

  .
  .
  .
begin
  lregx1:  for k in 2 downto 0 generate
        lregx1a:  REG port map(x,clk,fregx1(k));
      end generate lregx1;
  voter1:  voter port map(fregx1(0),fregx1(1),fregx1(2),regx1);

  lregx2:  for k in 2 downto 0 generate
        lregx2a:  REG port map(regx1,clk,fregx2(k));
      end generate lregx2;
  voter2:  voter port map(fregx2(0),fregx2(1),fregx2(2),regx2);

  lregx3:  for k in 2 downto 0 generate
        lregx3a:  REG port map(regx2,clk,fregx3(k));
      end generate lregx3;
  voter3:  voter port map(fregx3(0),fregx3(1),fregx3(2),regx3);


  .
  .
  .
```

Şekil 12: Hataya dayanıklı IIR süzgecin VHDL kodundan bir kısım (1 numaralı betiğin sonucu olarak).

```
  .
  .
  .
variable acc_1, acc_2 :  signed (31 downto 0);
begin
    if (clk'event and clk= '1') then
      acc := b(0)*x + b(1)*regx0 + b(2)*regx1+
        b(3)*regx2 + a(1)*regy0 + a(2)*regy1 + a(3)*regy2;
      acc_1 := b(0)*x + b(1)*regx0 + b(2)*regx1+
        b(3)*regx2 + a(1)*regy0 + a(2)*regy1 + a(3)*regy2;
      acc_2 := b(0)*x + b(1)*regx0 + b(2)*regx1+
        b(3)*regx2 + a(1)*regy0 + a(2)*regy1 + a(3)*regy2;
      acc := majority(acc,acc_1,acc_2);
    end if;


  .
  .
  .
```

Şekil 13: Hataya dayanıklı IIR süzgecin VHDL kodundan bir kısım (2 numaralı betiğin sonucu olarak).

kazandırmak için kullanılmıştır. Öncelikle hataya dayanıklılık konusu ve önemi bir IIR süzgeç örneği üzerinde belirtilmiş, daha sonra IIR süzgecin VHDL kodu üzerinde yapılan bazı kod dönüştürme işlemleriyle sentezlenen devrenin ne şekilde hataya dayanıklı hale getirildiği gösterilmiştir. Kod dönüştürme için yazılması gereken göreceli kısa betikler ile büyük programlar kolaylıkla hataya dayanıklı hale getirilebilmektedir. Kod dönüştürme işlemi devreye hataya dayanıklılık özelliği eklemede kullanılabileceği gibi başka özellikler kazandırmada da kullanılabilir. Bu çalışma, VHDL için genel amaçlı bir "aspect-oriented" programlama dili geliştirilmesinin önemi ve gerekliliğini de bir anlamda ortaya koymaktadır.

6. Teşekkür

Bu çalışmada örnek uygulama olarak kullandığımız IIR süzgecin tasarımı ve geçicici bozuklukların süzgeç karakteristiğine etkileri konusunda değerli fikirleriyle bize yardımcı olan Dr. Serdar Özen'e teşekkür ederiz.

```
  .
  .
  .
signal y_in_1, y_in_2, y_in_3 :  signed (31 downto 0);

  .
  .
  .
    y_in_1 <=  acc;
    y_in_2 <=  acc;
    y_in_3 <=  acc;
  end process;
voter_y:  voter port map(y_in_1, y_in_2, y_in_3, y_in);

  .
  .
  .
```

Şekil 14: Hataya dayanıklı IIR süzgecin VHDL kodundan bir kısım (3 numaralı betiğin sonucu olarak).

7. Kaynakça

[1] M. Baleani, A. Ferrari, L. Mangeruca, M. Peri, S. Pezzini, and A. Sangiovanni-Vincentelli. Fault-tolerant platforms for automotive safety-critical applications. In *International Conference on Compilers, Architectures and Synthesis for Embedded Systems, CASES'03*, San Jose, USA, November 2003. ACM.

[2] F. Cristian. Understanding fault-tolerant distributed systems. *Communication of the ACM*, 34(2):56–78, February 1991.

[3] Enoch O. Hwang. *Digital Logic and Microprocessor Design with VHDL*. Brooks / Cole, 2005.

[4] P. Jalote. *Fault-Tolerance in Distributed Systems*. Prentice Hall, 1994.

[5] S. S. Kulkarni. *Component-based design of fault-tolerance*. PhD thesis, Ohio State University, 1999.

[6] S. S. Kulkarni and A. Arora. Automating the addition of fault-tolerance. *Formal Techniques in Real-Time and Fault-Tolerant Systems*, 2000.

[7] J. C. Laprie. *Dependability—Basic Concepts and Terminology*, volume 5 of *Dependable Computing and Fault-tolerant Systems*. Springer-Verlag, 1992. IFIP WG 10.4.

[8] Zhiming Liu. *Fault-Tolerant Programming by Transformations*. PhD thesis, University of Warwick, 1991.

[9] Victor P. Nelson. Fault-tolerant computing: Fundamental concepts. *Computer*, 23(7):19–25, 1990.

[10] Volnei A. Pedroni. *Circuit Design with VHDL*. MIT Press, 2004.

[11] C. Tanzer, S. Poledna, E. Dilger, and T. Führer. A fault-tolerance layer for distributed fault-tolerant hard real-time systems. in www.tttech.com/technology/docs/history/TTTech_1998-Fault-Tolerance_Layer.pdf, 1998.

[12] Torres Wilfredo. Software fault tolerance: A tutorial. Technical report, NASA Langley Technical Report Server, 2000.

[13] Steve Winder. *Analog and Digital Filter Design*. Newness, Elsevier Science, 2002.