EK A

VHDL DONANIM TANIMLAMA DİLİ

Elektronik sistemlerin karmaşıklığının artması tasarım yöntemlerinin de gelişmesini gerektirmiştir. Bu sebeple, geleneksel "kağıt ve kalem kullanarak tasarımı yap" ve "devreyi kurarak denemeleri yap" yöntemlerinin yerini "tanımla ve sentezle" yöntemleri almıştır. Donanım Tanımlama Dilleri'nin (Hardware Description Languages : HDLs) "tanımla ve sentezle" yönteminde önemli bir rolü vardır. HDLs bir elektronik sistemin tanımlanmasında, test edilmesinde ve sentezlenmesinde kullanılırlar. Pek çok donanım tanımlama dillerinin arasında VHDL (**V**HSIC (**V**ery **H**igh **S**peed **I**ntegrated **C**ircuit) **H**ardware **D**escription **L**anguage) en yaygın kullanılanıdır.

VHDL, IEEE (IEEE standard 1076) ve Birleşik Devletler Silahlı Kuvvetleri (United States Military Department / MIL-STD-454L) tarafından standart bir tanımlama dili olarak kabul edilmiştir. 1980 lerin ortasında U.S savunma bölümü ve IEEE yüksek kapasiteli bir donanım tanımlama dili gerçekleştirmenin sponsorluğunu yaptılar.

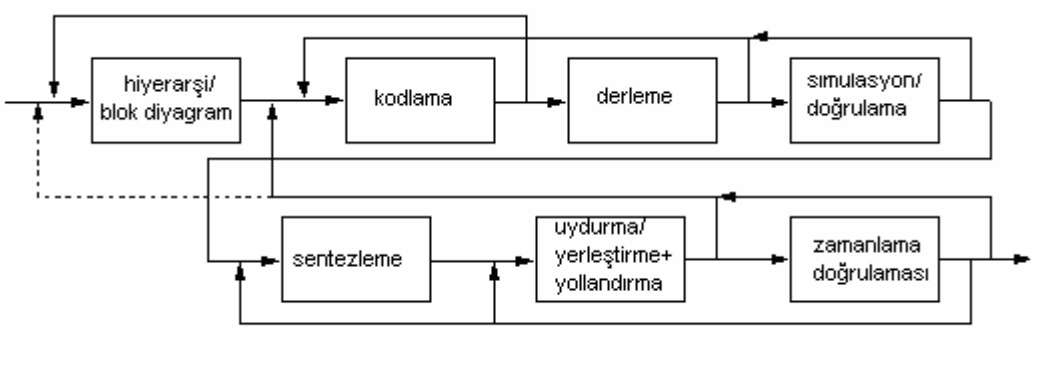
VHDL in özellikleri aşağıdaki gibidir:

- Tasarımlar hiyerarşili şekilde bileşenlerine ayrılabilir.
- Her bir tasarım elemanı iyi tanımlı bir arayüze (diğer elemanlarla bağlantı için) ve hatasız davranışsal tanımlamaya sahip olmalıdır.
- Davranışsal tanımlama yapılırken algoritma veya gerçek donanım yapıları kullanılarak elemanların işlemi belirtilir.
- Uyumluluk, zamanlama ve saatle denetim modellenebilir. VHDL senkron ve asenkron ardışıl devre yapılarını gerçekleyebilir.
- Bir tasarımın lojik işlemleri ve zaman davranışının simulasyonu yapılabilir.

VHDL, önce sadece dökümantasyon ve modelleme dili olarak ele alındı. Sayısal bir sistem davranışını tanımlamak ve simülasyonunu yapmak için geliştirildi. Zamanla yaygınlaşmaya başlayınca VHDL sentezleme araçları geliştirildi. Bu programlar VHDL davranışsal tanımından yola çıkarak lojik devre yapısını oluşturmaktadır. Bir yonga üzerinde, basit bir kombinezonsal devreden karmaşık bir mikroişlemciye kadar herhangi bir sayısal tasarım VHDL kullanılarak tasarımı, simülasyonu ve sentezlemesi yapılabilir.

A.1. TASARIM AKIŞI

VHDL tabanlı tasarım işlemi birkaç adımda yapılabilir. Bu adımlar HDL tabanlı bütün tasarımlara uygulanabilir. Şekil 1'de tasarım adımlarının taslağı görülmektedir.



Şekil A.1. VHDL Tasarım Adımları Blok Şeması

Birinci adımda yani , yapılacak tasarımın temel yaklaşımları ve blok diyagram seviyesinde tasarımı yapılır. Büyük sayısal tasarımlar genel olarak hiyerarşik bir yapıya sahiptir. VHDL , modülleri ve bunların ara yüzlerini tanımlama, daha sonra detaylarına girme konusunda iyi bir çalışma çerçevesi vermektedir.

Bir sonraki adım, modüllerin VHDL kodlarının yazılmasıdır. Bu kodlama işleminde, modüllerin ara yüzleri ve yapıları detaylı bir şekilde yazılır. VHDL, metin temelli dil olduğundan dolayı, bu işlemi yapmak için metin editörü kullanılabilir.

Kod yazıldıktan hemen sonra yapılması gereken, kodun doğruluğunu kontrol etmek için derleme işleminin yapılmasıdır. VHDL derleyicisi , yazılan kodların söz dizim kurallarına uygunluğunu ve diğer modüllerle uyumluluğunu analiz eder. Derleyici aynı zamanda simülasyon işleminde kullanılacak dosyaları oluşturur.

Derleme işlemi hatasız yapıldıktan sonra bir sonraki adım olan simülasyona geçilir. VHDL simülatörü, tasarımdaki girişleri uygulamayı ve çıkışları gözlemlemeyi herhangi bir fiziksel devreye sahip olmadan gerçekleştirir. Küçük projelerde girişleri üretmek ve çıkışları gözlemlemek elle yapılabilir. Fakat, büyük projeler için, kolaylık açışından "test bench" oluşturulup, otomatik olarak girişlerin uygulanması ve çıkışların gözlemlenmesi yapılabilir.

Simülasyon adımına aynı zamanda doğrulama da diyebiliriz. Simüle edilen devrenin istenen çıkışı verdiğini görmemiz gerekir. Burada fonksiyonel ve zamanlama olma üzere iki farklı doğrulama var. Fonksiyonel simülasyonda, zamanlamadan bağımsız olarak devrenin sayısal işlemi üzerinde durulur. Kapı gecikmeleri ve diğer zamanlama parametreleri sıfır olarak düşünülür. Zamanlama doğrulamasında , devrenin tahmini yaklaşık gecikmesini ve diğer zamanlama gereksinimleri incelenir. Burada elde edilen zamanlama sonuçlar tahmini değerlerdir. Gerçek sonuçlar sentezleme ve yerleştirme sonuçlarına bağlıdır.

Simülasyon adımında doğrulama işlemi yapıldıktan sonra sentezleme adımına geçilir. Bu adımda VHDL tanımları, donanımsal olarak hedef teknolojiye uygulanır. Sentezleme işlemi kullanılan araçlara ve hedef teknolojiye göre farkı şekillerde yapılır. Devre gerçekleştirilirken çalışılacak üretici (Xilinx, Altera Atel, vs) seçilir. Lojik sentezlemenin

gerçekleştirilmesi için kullanılan teknolojiyi destekleyen bir sentezleyiciye ihtiyaç vardır. Sentezleme araçları, hedef teknolojiye göre, kullanılacak kapıların listesini ve bunlar arasındaki ara bağlantıları belirten bir bağlantı listesi (netlist) üretir. Sentezleme sırasında yapılacak optimizasyonun alan veya hıza göre yapılabilmesi kullanıcı tarafından sağlanabilmektedir.. Ayrıca, devrenin çalışma frekansı, bazı bağlantıların gecikme bilgileri, saat ofseti gibi kısıtlamalar sentezleme sırasında verilebilir ve bu kısıtlamalara göre en uygun devrenin sentezlenmesi sağlanabilir.

Sentezleme sırasında elde edilen bağlantı listesi yerleştirme adımında yerleştirme araçları tarafından kullanılır. Bu araçlar sentezlenmiş bileşenleri hedef devreye yerleştirir. Bu adımda tasarımcı, Modüllerin yerleşimi, dış giriş ve çıkış pinlerinin atanması gibi ek kısıtlamalar tanımlayabilir.

En son aşama da, yerleştirilen devrenin zamanlama doğrulamasının yapılmasıdır. Sadece bu adımda, kablo uzunluğu, elektriksel yükleme ve diğer faktörlerden kaynaklanan gerçek devre gecikmesi hesaplanabilir. Eğer hesaplanan değerler isteği karşılıyorsa tasarım tamamlanır.

Tasarım yapılırken, şekil de görüldüğü gibi, adımlar arasında ileri ve geri gitme olabilir. Örneğin, kodlama sırasında herhangi bir problemle karşılaşılırsa, geriye dönülüp iyerarşi tekrar incelenir. Simülasyon sonucu istenildiği gibi değilse kodlama işlemi gözden geçirilir.

VHDL kullanımını en büyük avantajları şu şekilde sıralanabilir:

- Tasarım tanımlamaların bağımsız olması
- Bir çok üreticinin kullanılabilmesi
- Teknolojinin gerisinde kalan elemanların kullanımının engellenmesi
- Tasarımın güncellenmesi
- Belgelemenin standart bir şekilde hazırlanması
- Tasarım süresini kısaltır
- Ürünün pazara hızlı çıkmasını sağlar.
- Tasarım maliyetini azaltır
- Tasarım kalitesi artar.
- Fonksiyonların yüksek seviyede kontrolü yapılabilir.
- Tasarım elemanlarının tekrar kullanımı sağlanır.

A.2. VHDL TEMEL YAPILARI

Sayısal bir sistemin VHDL tanımlaması yapılırken dört temel yapı kullanılır. Bu yapılar entity bölümü, mimari bölümü, konfigürasyon ve paketlerdir.

Sayısal bir sistem, modüllerin bir hiyerarşi ile birleştirilmesiyle oluşturulur. Her modül VHDL de bir entity ile ifade edilir. Her entity ile giriş çıkışları ve fonksiyonu tamamen belirlenmiş olan bir donanım yapısı gösterilir. Her tanımlamanın entity bildirimi ve mimari bölümleri olmak üzere iki bölümü vardır. Entity bildirimi tasarımın dış bağlantılarının, mimari bölümü ise içeride yapılacak işlemlerin gösterilmesinde kullanılır. Pakette pek çok tanımlamada ortak olarak kullanılacak genel bilgiler verilir. Konfigürasyon ile yapısal tanımlamada kullanılacak alt sistemlerin giriş çıkış bilgileri tanımlanır.

A.2.1. ENTITY BİLDİRİMİ

Entity bildiriminde bir modülün bağlantıları ve bağlantı yollarının türü tanımlanır. Fakat bunların arasındaki ilişki konusunda bilgi verilmez. Yazılış kuralı şu şekildedir:

```
entity modülün ismi is
  [jenerik_bildirimi]
  [giriş/çıkış tanımları]
  {entity_bildirim_elemanları}
[begin
     devrenin çalışma şartlarının kontrol edildiği bölüm]
end [modülün ismi];
```

Şekil A.2: Entity bildiriminin genel yapısı

Generic_bildirimi modülün yapısını veya davranışını kontrol etmek üzere kullanılan sabitlerin tanımlandığı alandır. Yazılış şekli aşağıda verilmiştir.

```
generic (
  sabit_ismi : tipi [:= ilk değeri]
  {;sabit_ismi : tipi [:= ilk değeri]} );
```

Şekil A.3. Generic bildiriminin genel yapısı

Giriş/çıkış tanımları modülün bağlantı yollarının tanımlandığı alandır ve yazılış şekli aşağıda verilmiştir.

```
port (
  port_ismi : [mod] tipi [:= ilk değeri]
  {;port_ismi : tipi [:= ilk değeri]}  );
```

Şekil A.4. Generic bildiriminin genel yapısı

Giriş/çıkışlar modülün çevre modüllerle haberleşmesini sağlayan yollardır. Her giriş/çıkış bir mod (in, out, inout, buffer) ve bir veri tipi ile gösterilir. Giriş/çıkışların dört modu şu şekildedir:

- **in:** sadece okunabilir. Sadece giriş için kullanılır.
- **out:** sadece bir değer verilebilir, okunamaz. Sadece çıkış olarak kullanılır.
- **buffer:** okunabilir ve değer yazılabilir. Devrenin içinden sadece bir süreni olabilir.
- **inout:** okunabilir ve değer yazılabilir. Devrenin içinden birden fazla süreni olabilir.

Aşağıda bir bitlik bir tam toplayıcının bağlantıları görülmektedir. Modülün ismi TAM_TOPLAYICI'dır. Modülün, isimleri A, B ve EG olan üç adet girişi ve isimleri TOPLAM ve EÇ olan iki adet çıkışı vardır. Bütün giriş/çıkışlar STD_LOGIC tipindedirler. Bu özelliklere sahip birimin VHDL tanımlaması şu şekildedir:

```
entity TAM_TOPLAYICI is
  port (A, B, EG : in STD_LOGIC;
   TOPLAM, EC : out STD_LOGIC);
end TAM_TOPLAYICI;
```

Şekil A.5. Tam Toplayıcının entity bildirimi

Modülün yapısı ve zamanlaması jenerik sabitler kullanılarak kontrol edilebilir. Örneğin, aşağıda verilen VHDL tanımlamasında jenerik sabit N toplanacak sayıların basamak sayısını, M ise entity'nin zamanlama davranışını göstermek için kullanılmıştır. Bu tanımlamaya karşı düşen dört bitlik toplayıcının bağlantıları aşağıda gösterilmiştir. Sentezleme ve simülasyon sırasında jenerik sabitlerin değeri ihtiyaca göre değiştirilebilir.

```
entity TOPLAYICI is
  generic (N : INTEGER :=4;
        M : TIME := 10 ns);
  port (A, B : in STD_LOGIC_VECTOR(N-1 downto 0);
       T : out STD_LOGIC_VECTOR(N-1 downto 0);
       EG : in STD_LOGIC;
       EÇ : out STD_LOGIC);
end TOPLAYICI;
```

Şekil A.6. Tam Toplayıcının sabit tanımlamalı entity bildirimi

A.2.2.MİMARİ BİLDİRİMLERİ

Mimaride modülün girişleri ile çıkışları arasındaki ilişkiler tanımlanır. Tanımlama davranışsal, veri akışı veya yapısal olmak üzere üç farklı şekilde yapılabilir.

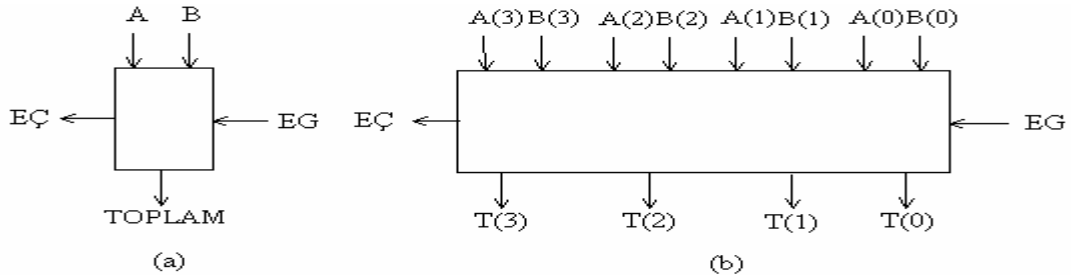
Bir mimari aşağıdaki yazım şekli kullanılarak tanımlanır:

```
architecture mimari_ismi of modülün_ismi is
  {mimari_bildirim_bölümü}
begin
  {eşzamanlı_satırlar}
end [mimari_ismi];
```

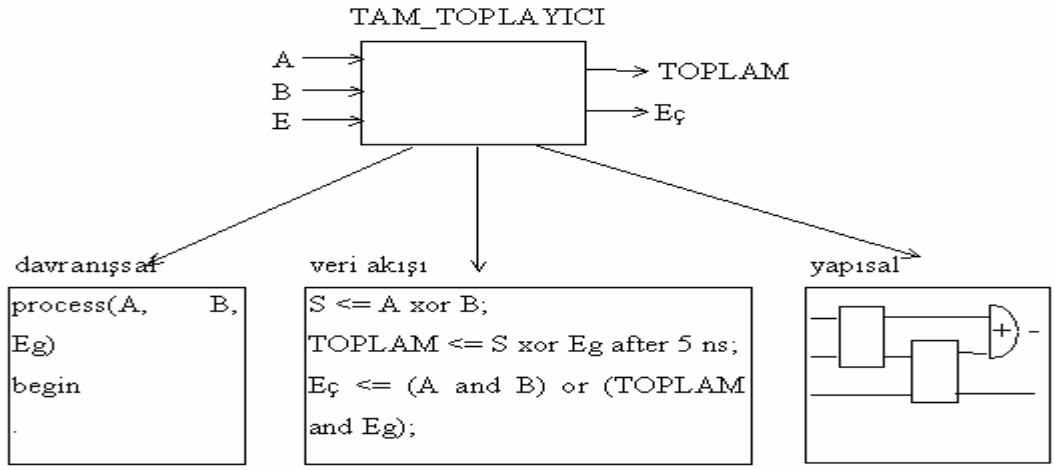
Şekil A.6. Mimari bildiriminin genel yapısı

Mimari bildirim bölümü ile mimari tanımlaması sırasında gereken ön tanımlar, eşzamanlı satırlar ile modülün çalışma şekli verilir. Mimari bildirim bölümünde mimarinin alt parçalarını birbirine bağlanmasında kullanılan işaretlerin tanımlaması yapılır. Her işaret taşıdığı verinin tipini belirten bir veri tipi ile tanımlanır. Her modül farklı mimarilerle

tanımlanabilir. Şekil A.7'de bir bitlik toplayıcıdan elde edilen dört bitlik toplayıcının blok diyagramı gösterilmiştir. Bu mimariler, farklı şekilde gerçekleme ya da aynı şekilde gerçeklemenin farklı şekilde yazılması olabilir. Örneğin, bir toplayıcı elde öngörülü (carry look ahead adder: CLA) veya hareket eden eldeli (ripple carry adder: RCA) şekilde gerçeklenebilir. Aynı şekilde gerçekleme de farklı mimari türleri kullanılarak tanımlanabilir. Şekil A.8'de TAM_TOPLAYICI'ya ait farklı mimari yapıları gösterilmiştir.



Şekil A.7. Entity Bildirimi (a) Bir bitlik toplayıcı (b) Dört bitlik toplayıcı.



Şekil A.8. Tam toplacıya ilişkin farklı şekildeki mimari yapılar.

Davranışsal Mimari

Davranışsal mimaride sistemin yaptığı işlem process'ler kullanılarak program benzeri bir şekilde yazılır, fakat tasarımın nasıl gerçekleneceği detaylandırılmaz.
Aşağıdaki örnekte TAM_TOPLAYICI'ya ait davranışsal mimari görülmektedir. Mimaride, duyarlılık listesinde A, B ve Eg olan bir process vardır. Process'in duyarlılık listesindeki işaretlerde herhangi bir değişiklik olmadıysa, process çalıştırılmaz. Ne zaman duyarlılık listesindeki işaretlerden biri değer değiştirirse, o zaman process aktif hale gelir ve içindeki satırlar ardışıl olarak işletilir.

```
architecture BEHAVIOUR of TAM_TOPLAYICI is
begin
  process (A, B, EG)
  begin
    if (A='0' and B='0' and EG='0') then
      TOPLAM <= '0';
      EÇ <= '0';
    elsif   (A='0' and B='0' and EG='1') or
        (A='0' and B='1' and EG='0') or
        (A='1' and B='0' and EG='0') then
      TOPLAM <= '1';
      EÇ <= '0';
    elsif   (A='0' and B='1' and EG='1') or
        (A='1' and B='0' and EG='1') or
        (A='1' and B='1' and EG='0') then
      TOPLAM <= '0';
      EÇ <= '1';
    else
      TOPLAM <= '1';
      EÇ <= '1';
    endif;
  end process;
end BEHAVIOUR;
```

Şekil A.9. Tam toplayıcının davranışsal mimari yapısı

Bir mimaride birden fazla process varsa hepsi eş zamanlı çalıştırılır, fakat process'ler içindeki satırlar yukarıdan aşağıya doğru sırayla çalıştırılır.

Veri Akışı Mimari

Veri akışı mimaride, kontrol işaretleri ve verilerin, toplayıcı, karşılaştırıcı, kod çözücü ve basit lojik kapılar gibi kombinezonsal devre elemanları üzerindeki eş zamanlı hareketleri tanımlanır. Aşağıda TAM_TOPLAYICI için yazılmış veri akışı şeklindeki mimari gösterilmiştir. Bu mimaride üç adet eş zamanlı işlenen işaret ataması vardır. Her satır, duyarlılık listesi sağ taraftaki işaretler olan bir process olarak düşünülebilir. İlk satırda herhangi bir gecikme süresi verilmediğinden, işlem simülatörde bir delta süresi kadar gecikmeli olarak yapılacaktır. İkinci satırdaki tanımlaya göre, TOPLAM işareti, S ve EG'nin değerlerinde değişiklik varsa, bu değişiklikten 10 ns sonra S xor EG değerini alacaktır. Aynı şekilde, A, B, S veya EG'nin değerinde bir değişiklik varsa bu değişiklikten 5 ns sonra EÇ işareti, (A and B) or (S and EG) değerini alacaktır.

```
architecture DATAFLOW of TAM_TOPLAYICI is
  signal S : STD_LOGIC;
begin
  S <= A xor B;
  TOPLAM <= S xor EG after 10 ns;
  EÇ <= (A and B) or (S and EG) after 5 ns;
end DATAFLOW;
```

Şekil A.10. Tam toplayıcının veri akışı mimari yapısı

Gecikme parametreleri olarak jenerik sabitler kullanılabilir. Aşağıdaki örnek ile bir gecikme parametresinin tanımı ve mimari de kullanımını gösterilmiştir.

```
entity TAM_TOPLAYICI is
  generic (N : TIME := 5 ns);
  port (A, B, EG : in STD_LOGIC;
       TOPLAM, EÇ : out STD_LOGIC);
end TAM_TOPLAYICI;

architecture DATAFLOW of TAM_TOPLAYICI is
  signal S : STD_LOGIC;

begin
  S <= A xor B;
  TOPLAM <= S xor EG after 2*N;
  EÇ <= (A and B) or (S and EG) after N;
 end DATAFLOW;
```

Şekil A.11. Tam toplayıcının sabit kullanılan veri akışı mimari yapısı

Yapısal Mimari

Yapısal mimaride eş zamanlı çalışan alt modüllerin listesi ve birbirlerine bağlantıları tanımlanır. Aşağıda bir bitlik bir tam toplayıcının yapısal mimarisi verilmiştir. Yapı isimleri YARI_TOPLAYICI ve VEYA_KAPISI olan iki alt modüle ayrılmıştır. Tam toplayıcının tasarımında iki yarı toplayıcı ve bir VEYA kapısı kullanılmıştır.

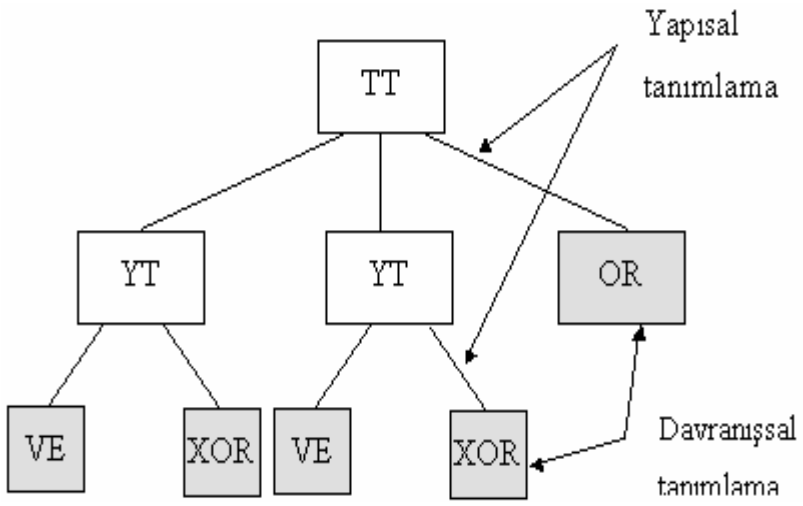
```
architecture STRUCTURE of TAM_TOPLAYICI is
  component YARI_TOPLAYICI
    port (G1, G2 : in STD_LOGIC;
     ELDE, TOPLAM : out STD_LOGIC);
  end component;
  component VEYA_KAPISI
    port (G1, G2 : in STD_LOGIC;
     ÇIKIŞ : out STD_LOGIC);
  end component;
  signal N1, N2, N3 : STD_LOGIC;

begin
  YT1 : YARI_TOPLAYICI port map (A, B, N1, N2);
  YT2 : YARI_TOPLAYICI port map (N2, EG, N3, TOPLAM);
  VEYA1 : VEYA_KAPISI port map (N1, N3, EÇ);
end STRUCTURE;
```

Şekil A.11. Tam toplayıcının yapısal mimari tasarımı

Tasarımın karmaşıklığı arttıkça bütün tasarımı bir defada tanımlamak oldukça zorlaşır. Bu sebeple, sistem daha az karmaşık alt sistemlere ayrılır. Her alt sistem tek başına gerçekleştirilebilen veya alt sistemlere ayrılabilen birimlerdir.
VHDL'de yapısal mimari kullanılan yüksek seviyedeki tasarım, sadece daha önce tanımlanmış alt sistemlerin listesi ve onların birbiriyle olan bağlantılarından oluşur.

Şekil A.4'de tam toplayıcının alt sistemlere ayrılması gösterilmiştir. Yarı toplayıcılar da alt sistemlere ayrılarak VE ve EXOR kapılarının uygun şeklide bağlanmasıyla elde edilmiştir.



Şekil A.12. Tam toplayıcının alt sistemlere ayrılışı.

```
entity YARI_TOPLAYICI is
     port (G1, G2 : in STD_LOGIC;
    ELDE, TOPLAM : out STD_LOGIC);
end YARI_TOPLAYICI;

architecture STRUCTURE of YARI_TOPLAYICI is
   component EXOR_KAPISI
     port (G1, G2 : in STD_LOGIC;
      ÇIKIŞ : out STD_LOGIC);
   end component;
   component VE_KAPISI
     port (G1, G2 : in STD_LOGIC;
      ÇIKIŞ : out STD_LOGIC);
   end component;

begin
   B1 : EXOR_KAPISI port map (A, B, TOPLAM);
   B2 : VE_KAPISI port map (A, B, ELDE);
end STRUCTURE;
```

Şekil A.11. Yarı toplayıcının yapısal mimari yapısı kullanılarak VHDL tasarımı

Yukarıda kullanılan her birimin daha önce tanımlanması ve çalışılan kütüphanenin içinde saklanması gerekir. Örneğin, EXOR kapısı aşağıdaki şekilde tanımlanmalıdır.

```
entity EXOR_KAPISI is
   port (G1, G2 : in STD_LOGIC;
    ÇIKIŞ : out STD_LOGIC);
end EXOR_KAPISI;

architecture BEHAVIOUR of EXOR_KAPISI is
begin
   process(G1, G2)
   begin
     if (G1='0' and G2='0') or
```

```
        (G1='1' and G2='1') then
        ÇIKIŞ <= '0';
      else
        ÇIKIŞ <= '1';
      endif;
    end process;
  end BEHAVIOUR;
```

Şekil A.12. EXOR lojik kapısının VHDL tasarımı

A.2.3. PAKETLER

Paketlerin kullanılmasındaki amaç, iki veya daha fazla birim tarafından ortak olarak kullanılan elemanları birarada toplamaktır. Bir paket iki bölümden oluşur: Paket bildirimi ve paket tanımı. Paket bildiriminde bazı veri tipleri, sabitler ve alt program isimleri tanımlanır.Paket tanımı, paket bildiriminde tanımlanan alt programların çalışmasının yazıldığı bölümdür. Eğer paket bildiriminde alt program yoksa paket tanımı yazılmaz.

Aşağıdaki örnekte bir paket bildirimi tanımlanmıştır. Paket ismi ÖR_PK dir. Bu paket bildiriminde 0 ile 255 aralığında INT8 isimli bir veri tipi, değerleri 0 ve 100, isimleri de SIFIR ve MAKSIMUM olan iki sabit ve BİR_ARTIRICI isimli bir fonksiyon tanımlanmıştır.

```
package ÖR_PK is
  subtype INT8 is INTEGER range 0 to 255;
  constant SIFIR : INT8 :=0;
  constant MAKSIMUM : INT8 :=100;
  procedure BIR_ARTIRICI (variable sayı : inout INT8);
end ÖR_PK;
```

Şekil A.13. Paket bildirimi

Paket bildirimi içinde alt program tanımlandığından paket tanımı yazılmalıdır.

```
package body ÖR_PK is
  procedure BIR_ARTIRICI (variable veri : inout INT8) is
  begin
    if (sayı >= MAKSIMUM) then
      sayı := SIFIR;
    else
      sayı := sayı+1;
    endif;
  end BIR_ARTIRICI;
end ÖR_PK;
```

Şekil A.13. Paket bildirimi içinde alt program tanımlama

A.3. VERİ NESNELERİ(DATA OBJECTS)

VHDL kodunda bildi veri nesnesi olarak sunulur.Üç farklı veri nesnesi vardır.

- Sinyaller(Signals)
- Sabitler (Constants)
- Değişkenler(Variables)

Lojik devreleri tanımlamak için en önemli veri nesnesi sinyallerdir.Sinyaller devredeki lojik sinyalleri tanımlar.Sabitler ve değişkenler ise seyrek olarak kullanılmaktadır.

Veri nesnesi isimleri

Veri nesnesi isimlerini tanımlamak için alfanumerik karakterler ve '_' karakteri kullanılabilir.

Kullanılan isim VHDL'in anahtar kelimeleri olmamalıdır.Harf ile başlamalı ve sonu '_' ile bitmemelidir.Kullanılabilecek örnek isimler x,x1,x_y,Byte olabilir.1x,_y,x__y,entity kullanılamaz.VHDL büyük veya küçük harf ayrımı yapmaz.Fakat okunması kolay olması için Anahtar kelimeler büyük harfle yazılır.

Veri nesne değerleri

Sinyal veri nesnesi bir devrede bit düzeyinde veya tamsayı olarak tanımlanabilir.Bir bitlik tanımlamada tek tırnak içine yazılır.'1' ,'0' gibi.Çok bitli tanımlamalar ise çift tırnak içerisinde ifade edilir.Dört bitlik bir sinyal "1001" şeklinde tanımlanır."1001" dört bitlik sinyal '1','0','0','1' şeklinde de tanımlanabilir.Aynı zamanda $(1001)_2=(9)_{10}$ şeklinde tamsayı olarak ta tanımlanabilir.Sabit ve Değişken veri nesneleri de aynı şekilde tanımlanır.

Sinyaller (signals)

Sinyal veri nesneleri aşağıdaki gibi tanımlanır.

```
SIGNAL sinyal_ismi :tip_ismi[:=değer] ;
```

Örnek:

```
SIGNAL Beep:BIT:='0'
SIGNALl Temp:STD_LOGIC_VECTOR(8 DOWNTO  0);
```

Sabitler(constants)

Sabitler,başlangıçta değer atanır ve sonradan değiştirilemezler.Genel tanımı

```
CONSTANT sabit_ismi:tip_ismi[:=değer] ;
```

Örnek:

```
CONSTANT Yes :BOOLEAN:=TRUE
CONSTANT Char7 :BIT_VEKTOR(4 DOWNTO O):="00111";
```

Değişkenler(variables)

Değişkenler geçici bilgi saklar.Process ve Subprogram 'da ifade edilir.

```
VARIABLE değişken_ismi:tip_ismi[:=değer] ;
```

Örnek:

```
VARIABLE X,Y:BIT;
```

VHDL de bütün veri nesneleri bir veri tipi ile tanımlanır.10 farklı veri tipi vardır.BIT,BIT_VEKTOR,STD_LOGIC,STD_LOGIC_VECTOR,SIGNED,UNSIGNED ,INTEGER,ENUMERATION,BOOLEAN

Bit ve bit_vector

'0','1'	BIT
"0101010"	BIT_VECTOR

Örnek:

```
SIGNAL      x1      :BIT;
SIGNAL      C       :BIT_VECTOR(1 TO 4);
SIGNAL      Byte    :BIT_VECTOR(7 DOWNTO 0);
```

C<=”1010”; ise C(1)=’1’,C(2)=’0’,C(3)=’1’,C(4)=’0’ olur.
Byte<=”10011000”; ise Byte(7)=’1’,Byte(6)=’0’.....Byte(0)=’0’

Std_logic ve std_logic_vector

STD_LOGIC tipi BIT tipine göre daha esnek bir yapıya sahiptir.Bu tipi kullanmak için programın başında USE ieee.std_logic_1164.all; ifadesi yazılmalıdır.

STD_LOGIC veri tipleri:0,1,Z,-,L,U,X ve W değerlerini alır.
Z:yüksek empedans, - :önemsiz,L:zayıf 0,H:zayıf 1,U:başlangıç değeri olmayan,X:bilinmeyen

```
SIGNAL X1,X2        :STD_LOGIC;
SIGNAL C            :STD_LOGIC_VECTOR(1 TO 4);
SIGNAL X,Y          :STD_LOGIC_VECTOR(3 DOWNOT 0);
```

Boolean

İki değeri vardır.TRUE ve FALSE.TRUE=1,FALSE=0

Enumeration

Bütün değerleri kullanıcı tarafından yazılabilir.

```
TYPE color IS (red.orange.yellow);
SIGNAL y :color;
```

A.4. OPERATÖRLER

VHDL de kullanılan operatörler sınıflara ayrılarak Tablo A.1 de verilmiştir.

Operatör Sınıfı	Operatör
Miscallaneus	**,ABS,NOT
Multiplying	*,/,MOD
Sign	+,-
Adding	+,-,&
Relational	=,/=,<,<=,>,>=
Logical	AND,OR,NAND,NOR,XOR,XNOR

ABS:Mutlak değer
&:Ampersand

Tablo A.1. Operatorlar

A.5 ARDIŞIL ATAMA DEYİMLERİ

Bir mimari içindeki aynı anda oluşan assignment (atama) deyimlerin sırası kodun anlamını değiştirmez. Mantık devrelerinin birçoğu bu deyimlerin kullanılmasıyla tanımlanabiliyor. Ancak kodda deyimlerin sırası kodun anlamını etkileyebileceği olasılığı

olduğu için VHDL ardışıl atama ifadeleri(*sequential assignment statements*,) diye adlandırılan başka deyim çeşitleri sunmaktadır. İF Deyimi , CASE Deyimi ve LOOP Deyimi olmak üzere 3 farklı ardışıl atama ifadesi vardır.

A.5.1 PROCESS DEYİMİ

VHDL dilinde ardışıl(sequential) deyimlerin sırası önemlidir fakat aynı anda oluşmuş (concurrent) deyimlerin sırası önemli değildir bu yüzden ardışıl deyimler aynı anda oluşan deyimlerden ayrılmalıdır. Bu bir PROCESS deyimi kullanımıdır. PROCESS deyimi bir mimari içinde bulunur ve kendi içinde diğer deyimleri çevirir. IF ,CASE ve LOOP deyimleri sadece bir deyimin içinde gözükebilir.

```
[process_label:]
PROCESS [( signal name {, signal name} )]
[VARIABLE declarations]
BEGIN
      [WAIT statement]
      [Simple Signal Assignment Statements]
      [Variable Assignment Statements]
      [IF Statements]
      [CASE Statements]
      [LOOP Statements]
END PROCESS [process_label] ;
```

Şekil A.14. PROCESS deyiminin genel formu

PROCESS deyiminin genel yapısı bir dereceye kadar mimariye benzer. Değişken veri nesneleri sadece PROCESS içinde bildirilir. Bildirilen herhangi bir değişken PROCESS içinde sadece kod ile kullanılabilir. Değişkenin faaliyet alanı sadece kod içinde sınırlıdır. PROCESS dışındaki bir değişkenin değerini kullanmak için bir signal atanabilir.

IF , CASE, ve LOOP deyimleri kombinezonsal yada ardışıl devrelerinden herhangi birini tanımlamak için kullanılır. Anlaşılması daha kolay olması için kombinezonsal devrelerle ilgili verilen örneklerle bu deyimleri tanıtacağız. Ardışıl devreler A.10 da anlatılmıştır.

A.5.2 IF DEYİMİ

IF deyiminin genel formu Şekil A.14 de verilmiştir.

```
IF expression THEN
      statement ;
      {statement ;}
ELSIF expression THEN
      statement ;
      {statement ;}
ELSE
      statement ;
      {statement ;}
END IF ;
```

Şekil A.15. IF deyiminin genel formu

Kombinezonsal logic devrelerde IF deyiminin kullanımıyla ilgili bir örneği Şekil A.16 da verilmiştir.

```
IF Sel = ’0’ THEN
      f <= x1 ;
ELSE
      f <= x2 ;
END IF ;
```
Şekil A.16. IF deyiminin bir örneği

Bu kod 2-to-1 multiplexer göstermektedir. Ardışıl lojik devrede IF deyiminin kullanımıyla ilgili örnek A.17 de verilmiştir. Şekil A.17 deki IF deyimiyle tanımlanan bir multiplexer, x1 veya x2 her iki giriş çıkış olan f’e atanmaktadır

```
PROCESS ( Sel, x1, x2 )
BEGIN
      IF Sel = ’0’ THEN
      f <= x1
      ELSE
      f <= x2 ;
      END IF ;
END PROCESS ;
```
Şekil A.17. IF deyiminin bir diğer örneği

IF deyimi kullanarak multiplexer tanımlamanın diğer bir yolu Şekil A.18 de verilmiştir.

```
PROCESS ( Sel, x1, x2 )
BEGIN
      f <= x1 ;
      IF Sel = 1 THEN
      f <= x2 ;
      END IF ;
END PROCESS ;
```
Şekil A.18. IF deyimi kullanılarak multiplexer tasarımının bir diğer örneği

İlk olarak ‘ f< = x1 ’ deyimini değerlendirelim. X1 değeri f ’ e değerine atanamayabilir. Çünkü PROCESS içinde kodda f ‘e sonradan gelen bir atama(assignment) olabilir. PROCESS içinde bu noktada eğer f’e hiçbir atama (assignment) olmazsa x1, f için varsayılan değer olarak kabul edilir. Eğer Se1 = 1 kabul edersek daha sonra ‘ f <= x2 ’ deyimini değerlendirilecek. Bu ikinci atama ilk atamadan daha önceliklidir. Bundan dolayı Se1 = 1 olduğu zaman PROCESS sonucu yani f’in değeri x2 dir. Eğer Se1 = 0 kabul edersek IF satırı geçersiz sayılır ve f’e varsayılan x1 değeri atanır.

Bu örnekte deyimlerin sırasının önemini örneklerle açıkladık. Eğer iki deyim yer değiştirseydi ve IF deyimi ilk yazılıp ‘ f <= x1 ;’ deyimi en sona yazılsaydı bu durumda f daima x1 değerini alırdı.

A.5.3 CASE DEYİMİ

CASE deyiminin genel formu:

```
CASE expression IS
      WHEN constant_value =>
```

```
        statement;
        {statement ;}
        WHEN constant_value =>
        statement;
        {statement ;}
        WHEN OTHERS =>
        statement;
        {statement ;}
    END CASE;
```
Şekil A.19. CASE deyiminin genel formu

Kombinezonsal lojik devrelerde CASE deyiminin kullanımıyla ilgili bir örneği Şekil A.20 de verilmiştir.

```
    CASE Sel IS
        WHEN '0' =>
        f <= x1;
        WHEN OTHERS =>
        f <= x2;
    END CASE;
```
Şekil A.20. CASE deyiminin genel formu

Bu kod kısım A.5.2 de kullanılan IF deyiminde tanımlanan 2x1 multiplexer ile aynı şeyi ifade ediyor. WHEN cümleleri için kullanılan olası bütün biçilmiş değerler listelenmelidir. Bu yüzden OTHERS anahtar sözcüğüne ihtiyaç duyulur. Aynı zamanda CASE deyimi içindeki WHEN cümleleri ortak özellikte olmalıdır. Ardışıl lojik devrede CASE deyiminin kullanımıyla ilgili örnek A.10 da verilmiştir

A.5.4 LOOP DEYİMİ

VHDL 2 çeşit LOOP deyimi sunmaktadır. Bunlar FOR_LOOP ve WHILE_LOOP. FOR_LOOP ve WHILE_LOOP deyiminin genel formu sırasıyla Şekil A.18 de verilmiştir.

```
    [loop_label:]                          [loop_label:]
    FOR variable_name IN range             WHILE boolean_expression
  LOOP                                     LOOP
    statement ;                            statement ;
    {statement ;}                          {statement ;}
    END LOOP [loop_label] ;                END LOOP [loop_label] ;
```
Şekil A.21. FOR_LOOP ve WHILE_LOOP deyiminin genel formu

Bu ifadeler bir FOR GENERATE ifadenin, aynı anda oluşan atama (concurrent assignment) deyimlerini tekrarlamak için kullanıldığı gibi bir veya daha çok sıralı atama (sequential assignment) ifadelerini tekrarlamak içinde kullanılır. A.9.7 de FOR_LOOP için örnek verilmiştir.

A.5.5 PROCESS İÇİNDE DEĞİŞKEN (VARİABLE) KULLANIMI

Bir signal den farklı olarak değişken data nesneleri, devrede bir tel parçasını belirtmez. Böylece bir signal kullanımının mümkün olmadığı durumlarda bir değişken lojik bir devrenin fonksiyonelliğini tanımlayabilirdi. Bu durumun örneği:

```
LIBRARY ieee;
USE ieee. std_logic_1164.all;

      ENTITY numbits IS
      PORT ( X: IN STD_LOGIC_VECTOR(1 TO 3) ;
      Count: BUFFER INTEGER RANGE 0 TO 3 ) ;
      END numbits;

ARCHITECTURE Behavior OF numbits IS
BEGIN
      PROCESS ( X )
      BEGIN
      Count <= 0 ;
      FOR i IN 1 TO 3 LOOP
      IF X(i) = '1' THEN
      Count <= Count + 1;
      END IF;
      END LOOP;
      END PROCESS;
END Behavior;
```

Şekil A.22. Buffer kullanarak bitleri sayan devre tasarımı

Bu kodun amacı 1' e eşit olan 3 bit X signalinde bitleri sayan logic devre tanımlamaktadır.2 bit unsigned integer olan sayma, Count olarak çağırılan bir çıkıştır. Dikkat edilecek olunursa Count Buffer mod ile tanımlanmıştır. Çünkü mimari içinde atama (assignment) operatörün sağ ve sol olmak üzere her iki kısmında da kullanılır. PROCESS içinde ilk olarak Count 0 ' a eşitlenir. Bu durumda hiçbir alıntı 0 sayısı için kullanılmaz çünkü VHDL 'de ondalık sayılar kabul edilir. Ondalık sayılar bir tam sayı signaline atayabilmek için alıntılar olmadan ifade edilir. Bu kodda i değişken indeksiyle FOR_LOOP kullanılmıştır. i değişkeninin 1 den 3 e kadar ki değeri için IF deyimi içinde FOR_LOOP x(i) nin her bir bit değerini kontrol eder; eğer x(i) 1 e eşitse Count değerini 1 artırır. Şekil A.22 de verilen kod, yasal VHDL kodudur ve herhangi bir hatayı oluşturmadan derlenebilir.

Şekil A.22 deki kodun amaçsız çalışmasının 2 nedeni vardır. İlki, PROCESS içinde signal Count için çeşitli atama(assignment) deyimleri vardır. Bir önceki örnekte açıkladığımız gibi etki sadece atama(assignment) ifadelerinin sonunda olacak. Bu yüzden eğer x 'in herhangi bir biti 1 ise ilk deyim olan 0 ı Count a atama deyiminin (Count <= 0) beklenen etkisi olamayacak. Çünkü FOR_LOOP içindeki atama(assignment) deyimiyle count değeri değişecek. Aynı zaman FOR_LOOP istenildiği gibi çalışmaz çünkü x(i) nin 1 olduğu her bir iteration bir önceki iterasyonun etkini göstermesini engeller. İkinci neden ise bu kodun " Count <= Count+'1' "; deyimi mantıklı değil çünkü geri beslemeli bir devre tanımlanmıştır. Devrenin kombinezonsal olduğu için, böyle geri besleme, salınımlarla sonuçlanacak ve devre, kararlı olmayacak.

Şekil A.22 deki VHDL kodunda signal yerine değişken kullanılarak kod istenilen hale getirilebilir.

```
LIBRARY ieee;
USE ieee. std_logic_1164.all;

ENTITY Numbits IS
      PORT ( X: IN STD_LOGIC_VECTOR(1 TO 3) ;
      Count: OUT INTEGER RANGE 0 TO 3 ) ;
END Numbits;

ARCHITECTURE Behavior OF Numbits IS
BEGIN
      PROCESS ( X )
      VARIABLE TMP: INTEGER;
      BEGIN
      Tmp := 0;
      FOR i IN 1 TO 3 LOOP
      IF X(i) = '1' THEN
      Tmp := Tmp + 1;
      END IF;
      END LOOP;
      Count <= Tmp;
      END PROCESS;
END Behavior;
```

Şekil A.23. Değişken kullanarak bitleri sayan devre tasarımı

Şekil A.23 de PROCESS içindeki Count signali yerine Tmp değişkeni kullanılmıştır. Tmp' nin değeri PROCESS in sonundaki Count 'a atanmıştır. <= operatörün aksine := operatörüyle gösterilen Tmp atama (assignment) deyimine dikkat edelim. := operatörüne variable assignment operator denir. <= operatöründen farklı olarak PROCESS 'in sonuna kadar programlanmıştır, atamayla sonuçlanmıyor. Doğrudan doğruya değişken atama yer alır. Bu atama Şekil A.22 deki koddaki iki problemin ilkini çözer. İkinci problem signal yerine değişken kullanarak çözüldü. Çünkü bir değişken, devrede bir teli temsil edemez, geri beslemeli bir devre gibi FOR_LOOP kelimesi kelimesine yorumlayıcı olmaya ihtiyacı yoktur. Değişken kullanarak FOR_LOOP sadece istenilen bir davranışı veya pratikliği temsil eder. Kod çevrildiği zaman, VHDL derleyicisi bir kombinezonsal devre meydana getirir bu FOR_LOOP içinde fonksiyonel araçları ifade eder.

A.6. SIRALI DEVRELER

Kombinezonsal devreler eşzamanlı ve sıralı tanımlanabilirler.Sıralı devreleri tanımlamak için sıralı tayin ifadelerini kullanırız.Sıralı devirleri temsil eden bazı örnekler verelim.

A.6.1. D TUTUCU (LATCH)

Şekil A.24'ta D Tutucusunun kodu verilmiştir.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY latch IS
  PORT (      D, clk      : IN  STD_LOGIC ;
              Q           : OUT       STD_LOGIC ) ;
END latch ;

ARCHITECTURE Behavior OF latch IS
BEGIN
  PROCESS ( D, clk )
  BEGIN
        IF clk = '1' THEN
              Q <= D ;
        END IF ;
  END PROCESS ;
END Behavior ;
```

Şekil A.24 D Tutucusunun VHDL kodu.

Süreç hassasiyet listesi mandalın veri girişi(D) ve saati kapsar(clk).Bu yüzden D veya clk in değerinde değişiklik meydana gelirse süreç aktif olur. Saat 1 ise IF yapısı içinde Q'nun alacagı değer D'ye eşit olur. Bizim, daha önce açıkladığımız gibi, ELSE yapısı yoksa ve IF bloğuna girilmiyorsa Q kendi değerini tutmaya devam eder.

A.6.2. D FLİP-FLOPU

Şekil A.25, Şekil A.24 den biraz farklıdır.Hassasiyet listesi saatin değerini içerir yani süreç saat sinyaline bağlı olarak aktif hale geçer.

VHDL'de EVENT bir nitelik çağırır. EVENT i Clock gibi bir sinyal ismiyle birleştirmek mantıklıdır.Birinci durumda Clock'EVENT and Clock = '1' dir ve Q değeri D nin alacağı değere bağlıdır.Clock tekrar '1' olduğunda meydana gelen değişiklere bağlı olarak flip flop tekrar tetiklenir. Bu durum saat sinyalinin alçak-yüksek bir geçişini tanımlar.Bu yüzden kod, pozitif kenarda başlatılan bir D flip flopunu tanımlar. std_logic_1164 paketi rising_edge ve falling_edge isimli iki fonksiyon içerir. Bunlar saati kontrol edecek daha kısa notasyonlar olarak kullanılabilirler. Şekil A.25'de "IF Clock'EVENT AND Clock = '1' THEN" satırıyla "IF rising_edge(Clock) THEN" ifadesini yer değiştirebiliriz.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY flipflop IS
  PORT (      D, Clock    : IN  STD_LOGIC ;
              Q                : OUT       STD_LOGIC ) ;
END flipflop ;

ARCHITECTURE Behavior OF flipflop IS
BEGIN
  PROCESS ( Clock )
  BEGIN
```

```
        IF Clock'EVENT AND Clock = '1' THEN
              Q <= D ;
        END IF ;
  END PROCESS ;
END Behavior ;
```

Şekil A.25. D flip-flop.

A.6.3. WAIT UNTIL İFADESİNİN KULLANIMI

Şekil A.25'de D flip-flopu için farklı bir sözdizimi kullanılmıştır. Saat darbesiyle senkronizasyon sağlanması "WAIT UNTIL Clock = '1' ;". İfadesinin kullanımıyla sağlanabilir.Bu ifade "saatin sonraki pozitif kenar tetiklenme durumuna kadar bekle" anlamını taşır.WAIT UNTIL yapısı özel bir durumdadır. Çünkü hassasiyet listesi ihmal edilebilir. Hassasiyet listesinde sadece Clock yer alır.Bu ifadeden önce başka ifade kullanılmamalıdır.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY flipflop IS
      PORT (      D, Clock    : IN  STD_LOGIC ;
                  Q                : OUT       STD_LOGIC ) ;
END flipflop ;

ARCHITECTURE Behavior OF flipflop IS
BEGIN
      PROCESS
      BEGIN
            WAIT UNTIL Clock = '1' ;
            Q <= D ;
      END PROCESS ;
END Behavior ;
```

Şekil A.26 WAIT UNTIL ifadesi kullanılarak D flip-flop tasarımı

A.6.4. ASENKRON RESETLİ FLİP-FLOP

Şekil A.27, şekil A.25 ile benzer olan bir süreci verir. D flip-flopu için asenkron reset veya sil girişi tanımlanmıştır.Reset sinyalinin ismi Resetn'dir. Resetn=0 durumunda Q çıkışlarının hepsi 0' lanır.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY flipflop IS
      PORT (      D, Resetn, Clock  : IN  STD_LOGIC ;
                  Q                 : OUT       STD_LOGIC ) ;
END flipflop ;

ARCHITECTURE Behavior OF flipflop IS
BEGIN
      PROCESS ( Resetn, Clock )
```

```
        BEGIN
              IF Resetn = '0' THEN
                    Q <= '0' ;
              ELSIF Clock'EVENT AND Clock = '1' THEN
                    Q <= D ;
              END IF ;
        END PROCESS ;
  END Behavior ;
```

Şekil A.27. Asenkron reset li D flip-flop

A.6.5. SENKRON RESET

Şekil A.28'de flip-flopun senkron reset girişleriyle çalışması görülür.

```
  LIBRARY ieee ;
  USE ieee.std_logic_1164.all ;

  ENTITY flipflop IS
        PORT (      D, Resetn, Clock  : IN  STD_LOGIC ;
                    Q                 : OUT       STD_LOGIC ) ;
  END flipflop ;

  ARCHITECTURE Behavior OF flipflop IS
  BEGIN
        PROCESS
        BEGIN
              WAIT UNTIL Clock = '1' ;
              IF Resetn = '0' THEN
                    Q <= '0' ;
              ELSE
                    Q <= D ;
              END IF ;
        END PROCESS ;
  END Behavior ;
```

Şekil A.28. Senkron reset li D flip-flop

A.6.7. SAKLAYICILAR (REGISTERS)

Yüksek bitli saklayıcılar için yapılması gereken şey entity bildirimi içinde flip flopların çoklu kullanımıdır. Şekil A.29 da dört bitlik STD_LOGIC_VECTOR girişleri D ve çıkışları Q kullanılır. Bu kod asenkron sil girişleriyle dört bitlik saklayıcıı tanımlar.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY reg4 IS
      PORT (      D     : IN  STD_LOGIC_VECTOR(3 DOWNTO 0) ;
                  Resetn, Clock     : IN  STD_LOGIC ;
                  Q     : OUT       STD_LOGIC_VECTOR(3 DOWNTO 0)
```

```
);
END reg4 ;

ARCHITECTURE Behavior OF reg4 IS
BEGIN
      PROCESS ( Resetn, Clock )
      BEGIN
            IF Resetn = '0' THEN
                  Q <= "0000" ;
            ELSIF Clock'EVENT AND Clock = '1' THEN
                  Q <= D ;
            END IF ;
      END PROCESS ;
END Behavior ;
```

Şekil A.29 Asenkron resetli 4 bit saklayıcı

Şekil A.30 ise entity regn olarak isimlendirilmiştir.Bu kodda saklayıcı n bite çıkarılmıştır.Flip flop sayısı degişkendir ve n'e baglıdır .

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY regn IS
      GENERIC ( n : INTEGER := 4 ) ;
      PORT (      D           : IN  STD_LOGIC_VECTOR(n-1 DOWNTO 0) ;
                  Resetn, Clock     : IN  STD_LOGIC ;
                  Q           : OUT       STD_LOGIC_VECTOR(n-1 DOWNTO 0)
) ;
END regn ;

ARCHITECTURE Behavior OF regn IS
BEGIN
      PROCESS ( Resetn, Clock )
      BEGIN
            IF Resetn = '0' THEN
                  Q <= (OTHERS => '0') ;
            ELSIF Clock'EVENT AND Clock = '1' THEN
                  Q <= D ;
            END IF ;
      END PROCESS ;
END Behavior ;
```

Şekil A.30. Asenkron Resetli n-bit saklayıcı

Şekil A.31 de ise enable girişi kullanılarak n bit saklayıcı tanımlanmıştır..Aktif kenar tetiklendiğinde saklayıcıların içindeki flip floplarda, enable değeri E=0 ise , durum korunur.E=1 olduğunda ise saklayıcı aktif saat darbesine bağlı yüklenir.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY regne IS
```

```
      GENERIC ( n : INTEGER := 4 ) ;
      PORT (      D           : IN  STD_LOGIC_VECTOR(n-1 DOWNTO 0) ;
                  Resetn      : IN  STD_LOGIC ;
                  E, Clock    : IN  STD_LOGIC ;
                  Q           : OUT       STD_LOGIC_VECTOR(n-1 DOWNTO 0)
) ;
END regne ;

ARCHITECTURE Behavior OF regne IS
BEGIN
      PROCESS ( Resetn, Clock )
      BEGIN
            IF Resetn = '0' THEN
                  Q <= (OTHERS => '0') ;
            ELSIF Clock'EVENT AND Clock = '1' THEN
                  IF E = '1' THEN
                        Q <= D ;
                  END IF ;
            END IF ;
      END PROCESS ;
END Behavior ;
```

Şekil A.31. Enable girişli n-bit saklayıcı

A.6.8 ÖTELEMELİ SAKLAYICILAR (SHIFT REGISTERS)

4 bit ötelemeli saklayıcının örneği şekil A.32 de gösterilmiştir. Koddaki satırlar referansın kolay olması için numaralandırılmıştır. Ötelemeli saklayıcının bir *w* seri girişi ve *Q* paralel çıkışları vardır. En sağdaki bit *Q(4)*,en soldaki bit *Q(1)* dir. Öteleme sağdan sola doğru yapılır. Saklayıcıın tapısında öteleme işlemini tanımlamak için kullanılan *Sreg* sinyali bulunur. *Sreg* e tüm atamalar IF şartı tarafından clock edgelerle eş zamanda yapılır. Böylece *Sreg* flip flopların çıkışlarını gösterir. 13. satırdaki durum *Sreg(4)* ün *w* değerine atandığını gösterir. Daha önce açıkladığımız gibi,bu atama hemen etkili olmaz; işlemin sonunda meydana gelecek şekilde programa koyulmuştur. 14. satırda *Sreg(4)* ün 13. satırın sonucuna ötelenmeden önceki geçerli değeri *Sreg(3)*e atanır.15. ve 16. satırlar öteleme işlemini tamamlarlar. *Sreg(3)* ve *Sreg(2)* nin 14. ve 15. satırların sonucuyla değiştirilmeden önceki değerlerini *Sreg(2)* ve *Sreg(3)* e ayrı ayrı atarlar. Sonuçta *Sreg*, *Q* çıkışlarına atanmış olur.

Şekil A.32 deki dikkat çekici nokta,13–16. satırlardaki atama ifadelerinin, işlem sonlanana kadar etkili olmamasıdır. Bundan dolayı, tüm flip floplar, ötelemeli saklayıcıda da istenildiği gibi, değerlerini aynı anda değiştirirler. 13–16.satırlardaki ifadeleri, hangi sırayla yazarsak yazalım, kod değişmez.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY shift4 IS
      PORT (      w, Clock    : IN  STD_LOGIC ;
                  Q           : OUT       STD_LOGIC_VECTOR(1 TO 4) ) ;
END shift4 ;
```

```
ARCHITECTURE Behavior OF shift4 IS
      SIGNAL Sreg : STD_LOGIC_VECTOR(1 TO 4) ;
BEGIN
      PROCESS ( Clock )
      BEGIN
            IF Clock'EVENT AND Clock = '1' THEN
                  Sreg(4) <= w ;
                  Sreg(3) <= Sreg(4) ;
                  Sreg(2) <= Sreg(3) ;
                  Sreg(1) <= Sreg(2) ;
            END IF ;
      END PROCESS ;
      Q <= Sreg ;
END Behavior ;
```

Şekil A.32. 4 bit ötelemeli saklayıcı

Bölüm A.5.5 de, değişkenleri tanıttık ve sinyallerden farkını gösterdik. Değişkenlerin kullanımını da içeren bir başka örneği olarak, Şekil A.32 deki kodu, Şekil A.33 de *Sreg* sinyal yerine değişken olarak tanımlanıp verilmiştir. 13. satırdaki ifade *w* nun değerini *Sreg(4)* e atar. *Sreg* değişken olduğundan dolayı ,atamalar hemen etkili olur. 14. satırda, değeri artık *w* olan *Sreg(4), Sreg(3)* e atanır. Böylece 14.satırda *Sreg(3) = w* olur. Benzer olarak,15 ve 16. satırlarda *Sreg(2)* ve *Sreg(1)* in değeri de *w* olur. Kod isten en ötelemeli saklayıcıyı tanımlamaz, tersine tüm flip floplara *w* girişinin değerini yükler.

Şekil A.33 daki kodun, bir ötelemeli saklayıcıyı doğru olarak tanımlayabilmesi için 13.-16. satırların sırası ters çevrilmelidir. Sonra ilk olarak *Sreg(1), Sreg(2)* ye atanır; sonra *Sreg(2), Sreg(3)* e atanır; vs. Her ardışık atama kendinden öncekinden etkilenmez, bundan dolayı değişken kullanımının semantiği bir problem yaratmaz. Bölüm A.5.5 de söylediğimiz gibi, sinyalleri ve değişkenleri aynı anda kullanmak kafa karıştırıcı olabilir.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY shift4 IS
      PORT (      w, Clock    : IN  STD_LOGIC ;
                  Q           : OUT       STD_LOGIC_VECTOR(1 TO 4) ) ;
END shift4 ;

ARCHITECTURE Behavior OF shift4 IS
BEGIN
      PROCESS ( Clock )
            VARIABLE Sreg : STD_LOGIC_VECTOR(1 TO 4) ;
      BEGIN
            IF Clock'EVENT AND Clock = '1' THEN
                  Sreg(4) := w ;
                  Sreg(3) := Sreg(4) ;
                  Sreg(2) := Sreg(3) ;
                  Sreg(1) := Sreg(2) ;
            END IF ;
            Q <= Sreg ;
```

```
      END PROCESS ;
END Behavior ;
```

Şekil A.33. Değişken kullanarak 4 bit ötelemeli saklayıcı

A.6.9. SAYAÇLAR (COUNTERS)

Şekil A.34 de asenkron sil girişli bir 4-bit sayacın kodu gösterilmiştir. Sayacın ayrıca bir enable girişi vardır. Pozitif clock edgede, eğer E enable girişi 1 ise sayaç arttırılır. Eğer E=0 ise, sayaç geçerli değerini korur.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE ieee.std_logic_unsigned.all ;

ENTITY count4 IS
       PORT ( Resetn          : IN  STD_LOGIC ;
                  E, Clock    : IN  STD_LOGIC ;
                  Q           : OUT       STD_LOGIC_VECTOR (3 DOWNTO 0)
) ;
END count4 ;

ARCHITECTURE Behavior OF count4 IS
      SIGNAL Count : STD_LOGIC_VECTOR (3 DOWNTO 0) ;
BEGIN
      PROCESS ( Clock, Resetn )
      BEGIN
            IF Resetn = '0' THEN
                  Count <= "0000" ;
            ELSIF (Clock'EVENT AND Clock = '1') THEN
                  IF E = '1' THEN
                        Count <= Count + 1 ;
                  END IF ;
            END IF ;
      END PROCESS ;
      Q <= Count ;
END Behavior ;
```

Şekil A.33. 4 bitlik sayıcı tasarımı