LUT (Look-Up Table), FPGA içerisindeki temel yapı taşlarından biridir. Temel amacı, dijital mantık devrelerini esnek bir şekilde uygulayabilmektir. FPGA'lerde milyonlarca **LUT** bulunduğundan, karmaşık dijital tasarımlar bu **LUT**'ların yapılandırılmasıyla gerçekleştirilebilir.

Bir **LUT**, aslında basit bir bellek yapısıdır. Bu yapı, bir doğruluk tablosu gibi çalışır; belirli giriş kombinasyonlarına karşılık gelen çıkış değerlerini içerir. FPGA, **LUT**'ları kullanarak çeşitli mantık fonksiyonlarını uygulayabilir.

Örneğin, dört girişli bir LUT düşünelim:

- Bu **LUT**, her bir girişin 0 veya 1 değerini alabileceği 16 farklı giriş kombinasyonuna sahiptir (2^4 = 16).
- Her giriş kombinasyonu için bir çıkış değeri tanımlanır (örneğin 0 veya 1).
- Bu doğruluk tablosu, **LUT**'un hafızasında saklanır.
- Bu sayede, belirli bir giriş kombinasyonu verildiğinde, **LUT** doğrudan buna karşılık gelen çıkış değerini verir.

Bu mekanizma sayesinde, bir **LUT** ile basit **AND**, **OR** veya **XOR** gibi fonksiyonlardan çok daha karmaşık mantık fonksiyonları bile oluşturulabilir.

LUT'un çalışma prensibini daha iyi anlamak için, bir örnek üzerinden inceleyelim. Dört girişli bir LUT ele alalım:

1. **Girişler**: **LUT**'un dört adet girişi (A, B, C, D) vardır. Bu girişlerin her biri 0 veya 1 olabilir.
2. **Çıkışlar**: Her olası giriş kombinasyonuna göre bir çıkış değeri atanır. Örneğin, **LUT** şu mantık fonksiyonunu hesaplayabilir: Y = (A **AND** B) **OR** (C **AND** D).

Bu örnekte:

- Giriş kombinasyonu A = 1, B = 1, C = 0, D = 1 ise çıkış Y = 1 olur.
- Diğer tüm giriş kombinasyonları için farklı çıkış değerleri olabilir, bu değerler **LUT** içinde saklanır.

FPGA içinde tasarım yaparken, çoğu dijital devreyi **LUT**'lar ile oluştururuz. Bunun nedenlerine şunları söyleyebiliriz:

Esneklik: Bir **LUT**, farklı doğruluk tabloları yüklenerek çeşitli mantık fonksiyonlarına uyarlanabilir.

Yüksek Performans: LUT, kombinasyonel mantığı hızlı bir şekilde gerçekleştirir, bu da FPGA'nin performansını artırır.

Kaydedicilerle Entegre Çalışma: LUT'lar genellikle flip-flop'larla birlikte kullanılır. Flip-flop'lar, LUT'ların çıkışlarını kaydederek saat sinyaliyle senkronize çalışmayı sağlar.

Örneğin, FPGA'de bir **MUX** (multiplexer) yapmak istiyorsak, ilgili giriş kombinasyonlarına karşılık gelen çıkış değerlerini **LUT**'a yazarız ve böylece **MUX** işlevi yerine getirilmiş olur.

LUT, işte bu saklanan değerlere bakarak belirli bir giriş kombinasyonu geldiğinde ilgili çıkışı verir. Bu yapı sayesinde, farklı mantık fonksiyonları aynı **LUT** üzerinde, sadece doğruluk tablosunu değiştirerek gerçekleştirilebilir.

Peki daha teknikselden ziyade daha bilgi bazlı konuşacak olursak "LUT'ları bizler tasarlayabilir miyiz?" sorusunun cevabını kolayca verebiliriz.

FPGA içerisinde **LUT**'ları kendimiz tasarlamayız. FPGA üreticileri (örneğin Xilinx, Intel vs.), FPGA'lerin içinde hazır olarak **LUT**'ları tasarlar ve bunları donanımın temel bileşenleri olarak kullanıma sunar. Ancak biz tasarımcılar, **LUT**'ların nasıl kullanılacağına dair fonksiyonları belirleriz. Tasarladığımız devre sonrasında yazılan kodlarımızdaki atanan değerlere göre LUT'lar kendiliğinden oluşurlar. Kaç adet LUT kullanıldığını da yine Vivado gibi simülasyon uygulamalarında implementation sonrasında çıkan verilerde görebiliriz. Aynı işi yapan bir yazılım düşünelim.

Diyelim ki bir donanım tasarlansın ve iki farklı mühendis bunları özgün şekilde yazılıma aktarsın. Sonuç olarak ikisi de aynı işlevleri yapsa bile kullanılan **LUT** sayısında farklılık olabilir. Bunun sebebi de iki farklı mühendisin farklı şekilde yazdıkları ve atamalar yaptıkları değerler olabilir. Bu yüzden de implementation sonrasında farklı sayıda **LUT** kullanılmış olabilir. Farklı sayıdaki LUT sayılarının farklı sebepleri de olabilir. Bunlardan da kısaca şu şekilde bahsedebiliriz:

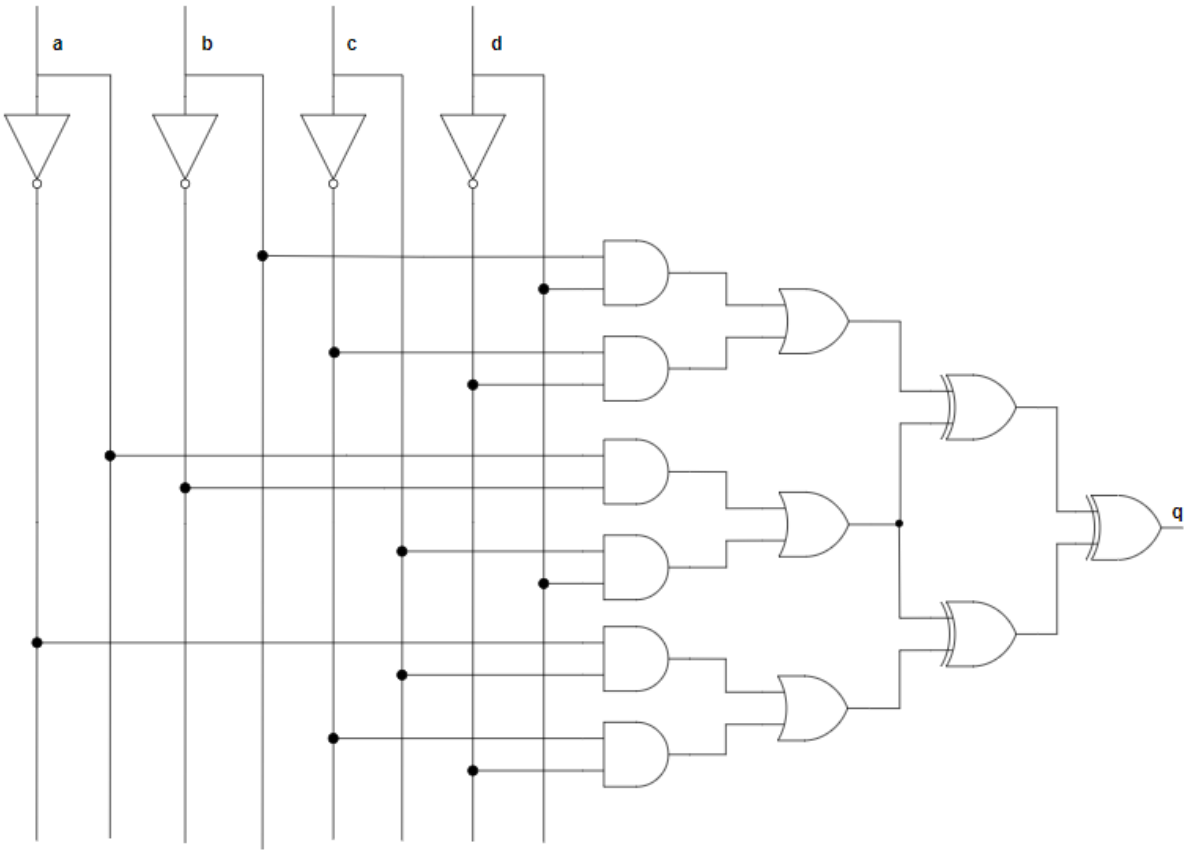
1. **Kodlama Tarzı**: Kodlama tarzı, kullanılan dil yapıları ve optimize etme şekli, sentez aracının tasarımı nasıl analiz edip optimize edeceğini etkiler. Örneğin, bir kişi işlem blokları (process) ve koşullu ifadelerle daha optimize edilmiş bir kod yazarken, diğeri daha karmaşık veya verimsiz bir mantık oluşturabilir. Bu da daha fazla LUT kullanımına neden olabilir.
2. **Optimize Edilebilirlik**: Sentez araçları, bazı durumlarda kullanılan mantık yapılarından bazılarını daha verimli şekilde yeniden düzenleyebilir veya sadeleştirebilir. Kodlama sırasında kullanılan farklı yaklaşımlar (örneğin, koşullu ifadeler veya case ifadeleri) sentez aracının devreyi nasıl yorumladığını ve optimize ettiğini etkileyebilir.
3. **Yüksek Seviyeli Yapılar**: Kodda kullanılan sayısal veya aritmetik işlemler, sentez sırasında bazıları doğrudan LUT'lara daha kolay map edilen yapılara dönüştürülebilirken, bazıları karmaşık yapılara dönüştürülebilir. Aynı işlevi gerçekleştiren iki farklı koddan biri daha basit bir mantık gerektirebilir ve daha az LUT kullanabilir.
4. **Optimize Edilmiş Yapılar ve Kapasite**: Bazı FPGA'larda belirli sayıda LUT ve optimize edilmiş yapılar (RAM, DSP blokları vb.) sınırlıdır. Sentez aracı, kullanılabilir kaynaklara bağlı olarak tasarımı farklı şekilde yerleştirip bağlantılandırabilir, bu da LUT kullanımında fark oluşturabilir.

Özetle, iki farklı kod aynı işlevi gerçekleştirse bile, tasarımcıların kodlama şekli, sentez aracının optimizasyon yöntemleri ve kullanılan FPGA'nın özelliklerine bağlı olarak farklı sayıda LUT kullanabilir.

Şimdi ise bir tane 4 girişli LUT tasarımının örneğini ve VHDL kodunu sizlere aşağıda vereceğim:

Öncelikle oluşturulacak devrenin en alt kapı seviyesinde tasarımı lazım. Bu tasarım önce gerçekleştirilir ve ardından doğruluk tablosu oluşturulur.

Lojik diyagram



Burada gözüken lojik devreye göre bir doğruluk tablosu oluşturmak istersek aşağıdaki gibi olur:

Doğruluk tablosu

A	B	C	D	Q
0	0	0	0	0
0	0	0	1	0
0	0	1	0	1
0	0	1	1	1
0	1	0	0	0
0	1	0	1	1
0	1	1	0	1
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1

Lojik devre ve doğruluk tablosuna göre Q çıkış değerlerimizi orada verilen girişler sonrası atama yaparsak bir LUT elde etmiş oluruz. Bu, el ile tasarlanan bir LUT fakat tekrar hatırlatmak

gerekirse bu tasarımı yapmaya ihtiyacımız yok çünkü sentez araçları (Vivado vs.) bunları kendisi otomatikmen oluşturuyor.

Yukarıda verilen lojik devreye göre bizim matematiksel olarak ifademiz şu hali almaktadır:

Q = {[(B.D) + (C'.D')] **xor** [(A.B') + (C.D)]} **XOR** {[(A.B') + (C.D)] **xor** [(A'.C) + (C'.D')]}

Böylece çıkış için devrenin genel matematiksel ifadesini de yazmış olduk.

Bunların da sadece '1' durumlarını yazacak olursak;

Q = (A'.B'.C.D') + (A'.B'.C.D) + (A'.B.C'.D) + (A'.B.C.D') + (A.B.C.D)

Şeklinde formülize edebiliriz. Bu aynı zamanda en basit hale indirgenmiş halidir. Bunu VHDL kodunu yazarken kullanabiliriz. Şimdi ise bunun VHDL kodunu yazıp nasıl yazmamız gerektiği hakkında fikrimizi oluşturalım:

Bu kodu iki farklı şekilde yazabiliriz:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity lut4 is
    Port (
        A : in STD_LOGIC;
        B : in STD_LOGIC;
        C : in STD_LOGIC;
        D : in STD_LOGIC;
        Q : out STD_LOGIC
    );
end lut4;

architecture Behavioral of lut4 is
begin
    process (A, B, C, D)
    begin
        case (A & B & C & D) is
            when "0000" => Q <= '0';
            when "0001" => Q <= '0';
            when "0010" => Q <= '1';
            when "0011" => Q <= '1';
            when "0100" => Q <= '0';
            when "0101" => Q <= '1';
            when "0110" => Q <= '1';
            when "0111" => Q <= '0';
            when "1000" => Q <= '0';
            when "1001" => Q <= '0';
            when "1010" => Q <= '0';
            when "1011" => Q <= '0';
            when "1100" => Q <= '0';
            when "1101" => Q <= '0';
            when "1110" => Q <= '0';
            when "1111" => Q <= '1';
            when others => Q <= null;
        end case;
    end process;
end Behavioral;
```

İlk olarak bu şekilde tüm değerleri atama yaparak işimizi de daha garantiye alarak, hata şansını en aza indirecek şekilde kodlamamızı bu şekilde yapabiliriz. Buradaki atamalar doğruluk tablosunda hangi değerin hangi sonuca geldiğine bakılarak teker teker atanması sonucunda yazılmıştır.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity lut_4_2 is
    Port (
        A : in STD_LOGIC;
        B : in STD_LOGIC;
        C : in STD_LOGIC;
        D : in STD_LOGIC;
        Q : out STD_LOGIC
    );
end lut_4_2;

architecture Behavioral of lut_4_2 is
begin
    process(A, B, C, D)
    begin
        Q <= ((not A) and (not B) and (C) and (not D)) or
             ((not A) and (not B) and (C) and (D)) or
             ((not A) and (B) and (not C) and (D)) or
             ((not A) and (B) and (C) and (not D)) or
             ((A) and (B) and (C) and (D));
    end process;
end Behavioral;
```

Bu da yine ilk yazılan kod ile aynı işlevi yapan bir diğer kod. Bu şekilde de yazarak doğruluk tablosunun değerlerini atamış oluruz. Burada biraz daha lojik kapıları işin içerisine sokarak işlem yapmış oluruz ama amaç ikisinde de aynıdır.

Yazılan bu iki kod aynı **LUT** için bir örnektir. İkisi de aynı kombinasyonlar sonucunda atanmış ve istenildiği zaman kullanılmak üzere bir **LUT**'a dönüştürülmüş devredir.