Hafıza elemanlarından olan ve bir tasarım yapılırken sıklıkla tercih edilen **register** ile ilgili açıklamalarda bulunup kısa bir VHDL kod örneği ile **register** tasarımı yapacağız.

Öncelikle basitçe **register**ın tanımını yapacak olursak; **Register**'lar, dijital devrelerde belirli bir veri miktarını saklayan ve gerektiğinde hızlıca erişilebilen hafıza hücreleridir. Genellikle, geçici veri saklama amacıyla kullanılırlar. Örneğin, bir işlemcinin çalışırken ihtiyaç duyduğu veriler ya da geçici sonuçlar **register**'larda tutulur. Her bir **register**, birçok flip-flop'tan (genelde D flip-flop) oluşur ve her bir flip-flop, bir bitlik veri saklar. Böylece, 8 bitlik bir **register** sekiz D flip-flop, 16 bitlik bir **register** ise on altı D flip-flop içerir.

Register'ların dijital tasarımdaki önemli özellikleri olarak şunları söyleyebiliriz:

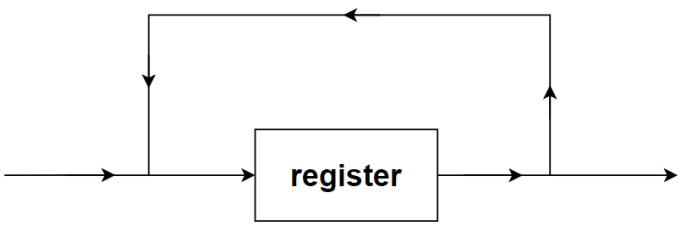
1. **Veri Giriş/Çıkış Yöntemleri**:
    - **Paralel Giriş/Çıkış**: Tüm bitler aynı anda girilir ve aynı anda okunur. Örneğin, 8 bitlik bir **register**'a paralel giriş yapılırsa, tüm 8 bit aynı anda yüklenir.
    - **Seri Giriş/Çıkış**: Veriler tek bir hattan, bir bit bir bit olacak şekilde yüklenir veya çıkarılır. Seri giriş/çıkış, paralel giriş/çıkışa göre daha yavaş olabilir, ancak daha az bağlantı hattı gerektirir.
2. **Kontrol Sinyalleri**:
    - **Clock (Saat Sinyali)**: Saat sinyali, **register**'ların ne zaman veri alacağını veya saklayacağını belirler. Genellikle yükselen kenar (rising edge) veya düşen kenar (falling edge) tetiklemeli olabilir.
    - **Enable (Aktivasyon) Sinyali**: **Register**'a veri yazmak için bu sinyalin aktif olması gerekir. Enable sinyali olmadığında, girişteki veri değişse bile **register** içindeki veri sabit kalır.
    - **Reset/Clear**: **Register**'ın tüm bitlerini sıfırlamak için kullanılan bir sinyaldir. Genellikle başlangıç veya hata durumlarında **register**'ın temizlenmesi gerektiğinde kullanılır.
3. **Saat Kontrollü (Senkron) ve Saat Kontrolsüz (Asenkron) Register'lar**:
    - **Senkron Register**: Saat sinyali ile çalışır ve yalnızca saat sinyali geldiğinde veri alır. Çoğu tasarımda bu tür **register**'lar tercih edilir.
    - **Asenkron Register**: Saat sinyalinden bağımsız olarak çalışabilir. Ancak asenkron **register**'lar, saat sinyali olmadan çalıştıkları için kontrolsüz sinyal değişimlerinden kaynaklanan sorunlara daha yatkındır. Bu nedenle FPGA gibi senkron devrelerde nadiren kullanılırlar.

Register'lar kullanım amacına göre çeşitli türlere ayrılır. İşte en yaygın kullanılan türlerden bazıları:

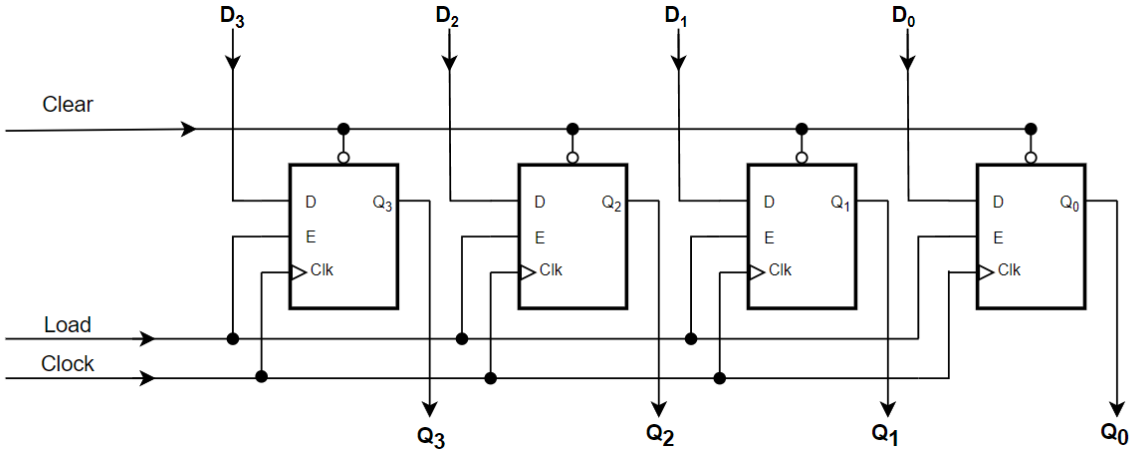
1. **Shift Register (Kaydırmalı Register)**:
    - Bir veri bitini kaydırarak bir sonraki pozisyona taşıyan özel bir **register** türüdür. Örneğin, bir 8 bitlik **shift register**'a her clock darbesinde bir bit veriyi girerken diğer bitler bir pozisyon kaydırılır. **Shift register**'lar veri işleme ve seri-paralel dönüşüm gibi alanlarda sıklıkla kullanılır.
2. **Counter (Sayaç)**:
    - Sayma işlemi yapmak için kullanılan bir **register** türüdür. Counter, clock darbelerine göre artan ya da azalan bir sayısal değer saklar. Çoğu dijital tasarımda zamanlayıcı, frekans bölücü veya olay sayıcı olarak kullanılır.
3. **Accumulator (Biriktirici)**:
    - Toplama ve biriktirme işlemi için kullanılan özel bir **register** türüdür. Genellikle sayısal işlemcilerde veya dijital sinyal işleme devrelerinde bir aritmetik işlem sonrası sonucu saklamak için kullanılır.
4. **Parallel-in, Parallel-out (PIPO)**:
    - Bu tür **register**'larda veriler paralel olarak alınır ve paralel olarak çıkar. Her bit, kendi flip-flop'unda tutulur ve işlem bir clock sinyali ile senkronize edilir.

Peki biz FPGA içerisinde **register**'ları nasıl oluştururuz?

Register, en basit haliyle, bir döngü oluşturarak gelen veriyi girişten alıp çıkışa gönderirken aynı zamanda bu veriyi içerde tutmaya yarayan bir yapı olarak düşünülebilir. Gelen veri, çıkıştan tekrar girişe bağlanarak her saat darbesinde yenilenir ve bu sayede veri sürekli olarak içeride saklanır. Bu işlem, kesintisiz bir şekilde aynı verinin tutulmasını sağlar. Aşağıda, bu mantığı görselleştirmek ve daha iyi anlamak için hayali bir çizim örneği sunulmuştur. Bu çizim, verinin nasıl döndüğünü ve içeride nasıl saklandığını daha net anlamanıza yardımcı olabilir.



Register tanımını VHDL ile yaparken de aslında yukarıdaki görseli kullanırız. Çıkışımızı girişimize bağlarız ve bu şekilde veriyi saklamış oluruz.



Yukarıdaki görselde ise 4 bitlik bir **register**'ın iç devresini görüyorsunuz. Burada 4 adet D Flip-Flop bulunmakta ve her bir flip-flop 1 bit veri sakladığından totalde 4 bitlik bir veriyi depolamış oluyoruz.

Peki **register** bize ne gibi avantaj sağlar diye düşünürsek de şunlardan bahsedebiliriz:

1. **Zamanlama Kontrolü**: Saat sinyaliyle tetiklenmeleri sayesinde senkron bir veri akışı sağlar.
2. **Güvenilir Veri Saklama**: Veriler, sistemin geri kalanından izole bir şekilde saklanır ve clock sinyaline göre güncellenir.
3. **Veri Senkronizasyonu**: Farklı clock domain'lerinde bile veri kaybı olmadan senkronizasyon sağlar.

Şimdi ise basit bir VHDL kodu ile register tanımlaması yapalım. Hatta işin içerisine biraz daha karmaşıklık katıp **register** özelliklerinin şu şekilde olmasını isteyelim:

- 16 bit olsun,
- Veriyi 8 bit 8 bit olarak iki seferde alsın,
- İlk gelen 8 biti registerin ilk 8 bitine,
- İkinci gelen 8 biti registerin ikinci 8 bitine kaydetsin ve çıkışa 16 bit olarak versin.

```vhdl
library IEEE;
use IEEE.std_logic_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity register_16bit is
    Port (
        clk       : in  std_logic;
        rst       : in  std_logic;
        en_i      : in  std_logic;
        d_i       : in  std_logic_vector(7 downto 0);
        q_o       : out std_logic_vector(15 downto 0)
    );
end register_16bit;

architecture Behavioral of register_16bit is
    signal s_reg_data : std_logic_vector(15 downto 0) := (others => '0');
    signal s_state    : std_logic := '0';
begin
    process (clk, rst)
    begin
        if rst = '1' then
            s_reg_data <= (others => '0');
            s_state <= '0';
        elsif rising_edge(clk) then
            if en_i = '1' then
                if s_state = '0' then
                    s_reg_data(7 downto 0) <= d_i;
                    s_state <= '1';
                else
                    s_reg_data(15 downto 8) <= d_i;
                    s_state <= '0';
                end if;
            end if;
        end if;
    end process;
    q_o <= s_reg_data;
end Behavioral;
```

Kodun analizini yapıp detaylı olarak açıklamak gerekirse;

```vhdl
library IEEE;
use IEEE.std_logic_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity register_16bit is
    Port (
        clk       : in  std_logic;
        rst       : in  std_logic;
        en_i      : in  std_logic;
        d_i       : in  std_logic_vector(7 downto 0);
        q_o       : out std_logic_vector(15 downto 0)
    );
end register_16bit;
```

Burada kullanılacak kütüphaneler ve register için giriş çıkış portların tanımı yapıldı.

- **clk:** Saat sinyali, register verilerinin senkron çalışmasını sağlar. Veriler yalnızca saat sinyalinin pozitif kenarında yüklenir.

-rst: Asenkron reset sinyali. Aktif olduğunda ('1'), tüm **register** içeriği sıfırlanır ve işlem başa döner.

- **en:** Enable sinyali, veri yüklenmesine izin verir. '0' olduğunda, **register** hiçbir işlem yapmaz.

-d: 8 bitlik veri girişi. Her bir işlemde bu giriş **register**'e yazılır.

-q: Register'in 16 bitlik çıkışı. Yazma işlemlerinden sonra **register** içeriği buradan okunabilir.

```
architecture Behavioral of register_16bit is
    signal s_reg_data : std_logic_vector(15 downto 0) := (others => '0');
    signal s_state    : std_logic := '0';
```

Burada sinyaller tanımlanır.

- **reg_data: Register**'in iç yapısını temsil eden sinyal. Burada tüm 16 bitlik veri saklanır. İlk başta **others** => **'0'** ifadesiyle sıfırlanır.

- **state:** İşlemi takip eden bir kontrol sinyali:

  - '0': İlk gelen veriyi alt 8 bite yaz.
  - '1': İkinci gelen veriyi üst 8 bite yaz.

```
begin
    process (clk, rst)
    begin
        if rst = '1' then
            s_reg_data <= (others => '0');
            s_state <= '0';
        elsif rising_edge(clk) then
            if en_i = '1' then
                if s_state = '0' then
                    s_reg_data(7 downto 0) <= d_i;
                    s_state <= '1';
                else
                    s_reg_data(15 downto 8) <= d_i;
                    s_state <= '0';
                end if;
            end if;
        end if;
    end process;
```

Burada process bloğu tanımlanır.

Asenkron Reset Kontrolü:

- Eğer **rst** = '1' ise:
- **reg_data** sıfırlanır (**others** => **'0'**).
- **state** sıfırlanır ('0'), yani bir sonraki işlem alt 8 bit ile başlar.

Saat Sinyali ile Çalışma:

- Eğer saat sinyali pozitif kenarda **(rising_edge(clk))** ise ve **en** = '1':
    - **state = '0'**:
        - Giriş verisi **d**, **register**'in alt 8 bitine (reg_data(7 downto 0)) yazılır.
        - **State** '1' yapılır ve bir sonraki işlem üst 8 bit yazma işlemi olur.
    - **state = '1'**:
        - Giriş verisi d**, register**'in üst 8 bitine (reg_data(15 downto 8)) yazılır.
        - **State** '0' yapılır ve bir sonraki işlem alt 8 bit yazma işlemi olur.

```
    q_o <= s_reg_data;
end Behavioral;
```

Son olarak **register** içeriği sürekli olarak çıkış portu **q**'ya atanır.

Böylece 16 bitlik değer ataması olan registerimizi tanımlamış olduk. Şimdi de buna bir adet test bench kodu yazıp testini gerçekleştirelim ve simülasyon ekranından sonuçları okuyalım.

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

entity tb_register_16bit is
end tb_register_16bit;

architecture Behavioral of tb_register_16bit is

    component register_16bit is
        Port (
            clk       : in  STD_LOGIC;
            rst       : in  STD_LOGIC;
            en_i      : in  STD_LOGIC;
            d_i       : in  STD_LOGIC_VECTOR(7 downto 0);
            q_o       : out STD_LOGIC_VECTOR(15 downto 0)
        );
    end component;

    signal s_clk     : STD_LOGIC := '0';
    signal s_rst     : STD_LOGIC := '0';
    signal s_en      : STD_LOGIC := '0';
    signal s_d       : STD_LOGIC_VECTOR(7 downto 0) := (others => '0');
    signal s_q       : STD_LOGIC_VECTOR(15 downto 0);

    constant clk_period : time := 10 ns;
```

```
begin

    uut: register_16bit
        port map (
            clk => s_clk,
            rst => s_rst,
            en_i  => s_en,
            d_i   => s_d,
            q_o   => s_q
        );

    clk_process: process
    begin
        s_clk <= '0';
        wait for clk_period / 2;
        s_clk <= '1';
        wait for clk_period / 2;
    end process;

    stimulus_process: process
    begin
        s_rst <= '1';

        s_en <= '0';
        s_d <= "00000000";
        wait for clk_period * 2;

        s_rst <= '0';
        wait for clk_period ;

        s_en <= '1';
        s_d <= "10101010";
        wait for clk_period ;

        s_en <= '1';
        s_d <= "11001100";
        wait for clk_period;

        s_en <= '0';
        s_d <= "11110000";
        wait for clk_period * 2;

        s_rst <= '1';
        wait for clk_period * 2;

        wait;
    end process;

end Behavioral;
```

Kodun detaylı analizine bakacak olursak;

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

entity tb_register_16bit is
end tb_register_16bit;
```

Burada kütüphane ve entity kısmı tanımlanır. Test bench tasarımının giriş ve çıkış portları olmadığından entity kısmı boş bırakılmıştır. Test benchlerde klasik bir durumdur.

```
architecture Behavioral of tb_register_16bit is

    component register_16bit is
        Port (
            clk       : in  STD_LOGIC;
            rst       : in  STD_LOGIC;
            en_i      : in  STD_LOGIC;
            d_i       : in  STD_LOGIC_VECTOR(7 downto 0);
            q_o       : out STD_LOGIC_VECTOR(15 downto 0)
        );
    end component;
```

Bu bölüm register_16bit adlı register bileşenini tanımlar. Bu bileşen, test edilen registerin ara bağlantılarının tanımlanmasını sağlar. Yani oluşturulan register_16bit adlı bloğu testbench bloğumuz üst modül olacak şekilde buraya bağlarız.

```
    signal s_clk      : STD_LOGIC := '0';
    signal s_rst      : STD_LOGIC := '0';
    signal s_en       : STD_LOGIC := '0';
    signal s_d        : STD_LOGIC_VECTOR(7 downto 0) := (others => '0');
    signal s_q        : STD_LOGIC_VECTOR(15 downto 0);
    constant clk_period : time := 10 ns;
```

Test bench içerisinde kullanılan sinyallerdir. Bu sinyaller, component'in portlarına bağlanır. Ardından bir adet saat sabiti tanımlanır. Saat sinyali, tasarımın çalışmasını senkronize eden ana unsurdur.

- **s_clk**: Saat sinyali (başlangıç değeri '0').
- **s_rst**: Reset sinyali (başlangıçta reset pasif, yani '0').
- **s_en**: Enable sinyali (veri yüklemesini kontrol eder).
- **s_d**: Giriş verisi (8 bitlik veri; başlangıçta sıfır).
- **s_q**: Çıkış verisi (16 bitlik veri; register içeriğini gösterir).

```
begin

    uut: register_16bit
        port map (
            clk => s_clk,
            rst => s_rst,
            en_i  => s_en,
            d_i   => s_d,
            q_o   => s_q
        );
```

- **uut (unit under test):** Test edilecek tasarımın bir örneğidir.
- port map ile test bench'te tanımlanan sinyaller (clk, rst, en, vb.), tasarımın portlarına bağlanır. Böylece, sinyaller üzerinde yapılan değişiklikler doğrudan tasarımı etkiler.

```
clk_process: process
    begin
        s_clk <= '0';
        wait for clk_period / 2;
        s_clk <= '1';
        wait for clk_period / 2;
    end process;
```

Bu process bloğu, saat sinyalini **s_clk** oluşturur. Saat sinyali, **clk_period/2** süresince '0' ve ardından **clk_period/2** süresince '1' olarak atanarak bir kare dalga sinyali üretir. Bu döngü, test süresince devam eder.

```
    stimulus_process: process
    begin

        s_rst <= '1';
        s_en <= '0';
        s_d <= "00000000";
        wait for clk_period * 2;

        s_rst <= '0';
        wait for clk_period;

        s_en <= '1';
        s_d <= "10101010";
        wait for clk_period;

        s_d <= "11001100";
        wait for clk_period;

        s_en <= '0';
        s_d <= "11110000";
        wait for clk_period * 2;

        s_rst <= '1';
        wait for clk_period * 2;

        wait;
    end process;

end Behavioral;
```

Burası artık test senaryosunun döndüğü kısımdır.

- **rst <= '1';**

Reset sinyali aktif hale getirilir ve register sıfırlanır (q = "00000000_00000000").

- **en <= '0';**

Enable sinyali pasif olduğundan veri yükleme yapılmaz.

- **d <= "00000000";**

Veri girişine başlangıçta sıfır atanır.

- **rst <= '0';**

Reset kapatılır. Artık tasarım normal çalışmaya hazırdır.

- **en <= '1';**

Enable sinyali aktif hale getirilir ve veri girişine izin verilir.

- **d <= "10101010";**

Giriş verisi alt 8 bitlik bölüme yazılır. Çıkış şu hale gelir:

q = "00000000_10101010"

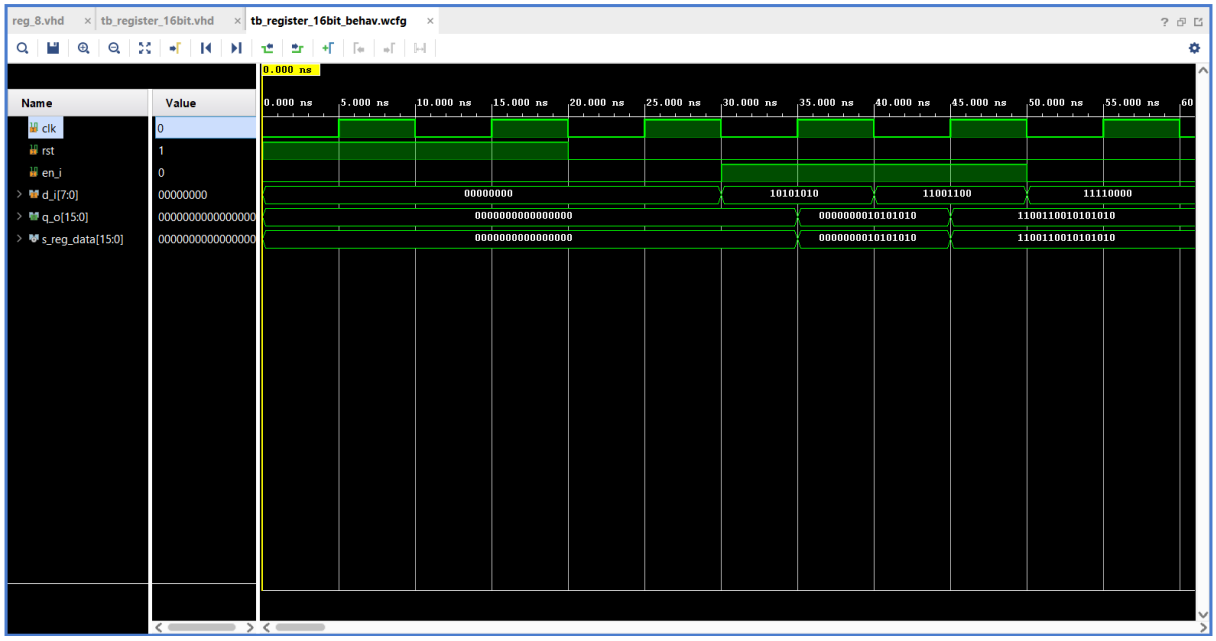
- Giriş verisi üst 8 bitlik bölüme yazılır. Çıkış şu hale gelir:

  q = "11001100_10101010"

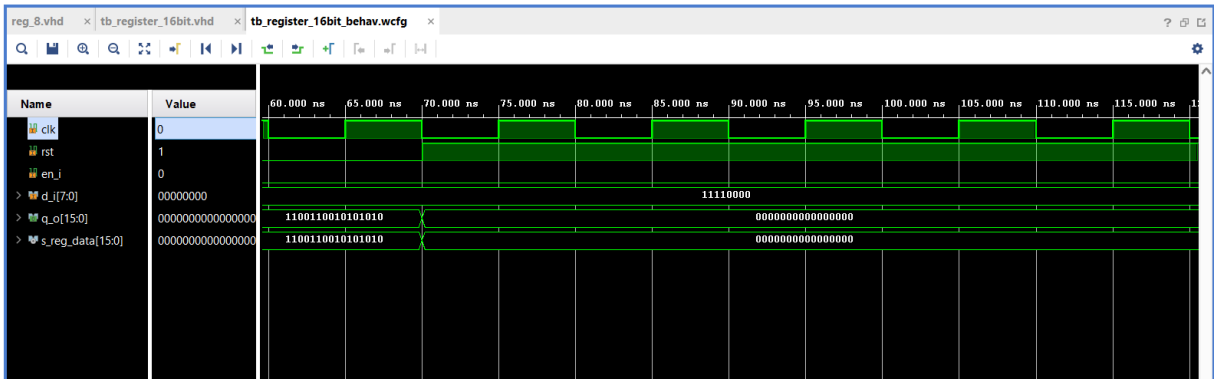
- **en <= '0';**

  Enable pasif olduğundan, giriş verisindeki değişiklikler (d <= "11110000";) çıkışa yansımaz.

- Reset tekrar etkinleştirilir ve çıkış sıfırlanır (**q** = "00000000_00000000").



Simülasyon ekranında da göründüğü gibi öncelikle **state** ‘0’ olduğundan **d_i** portuna gelen veriyi **s_reg_data** sinyalinin ilk 8 bitine yüklüyor, sonrasında **state** sinyalini ‘1’ yapıp bu sefer **d_i** portuna gelen diğer veriyi **s_reg_data** sinyalinin ikinci 8 bitine yüklüyor. Böylece sırası ile “**10101010**” ve “**11001100**” verilerini **q_o** portundan “**1100110010101010**” değerini görebiliyoruz.



Son olarak **rst** sinyalimizi ‘1’ durumuna çekip **register** sıfırlanıyor. Son durumda **d_i** portunda veri olsa bile o veriyi içeriye yazmamasının sebebi **rst** aktif durumda olduğu içindir.