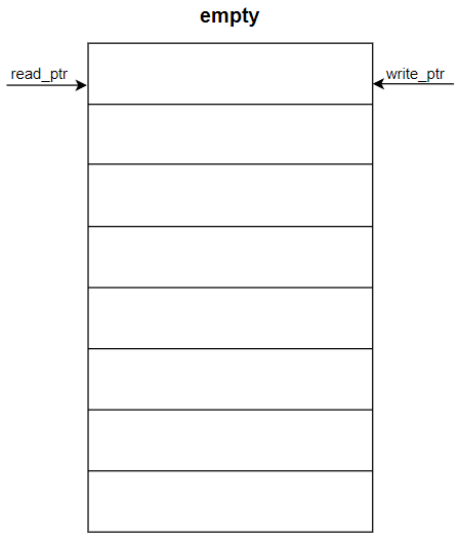
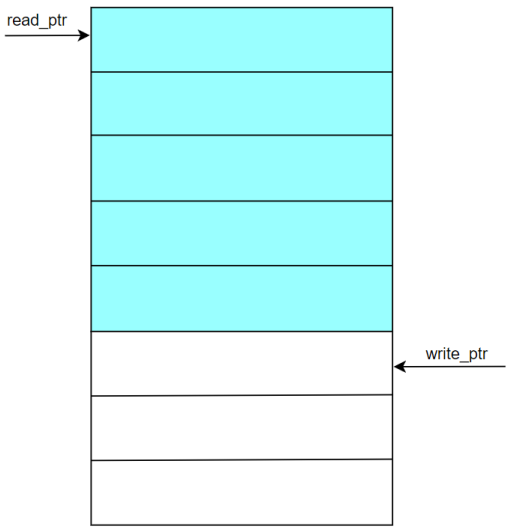
FIFO (First In, First Out)

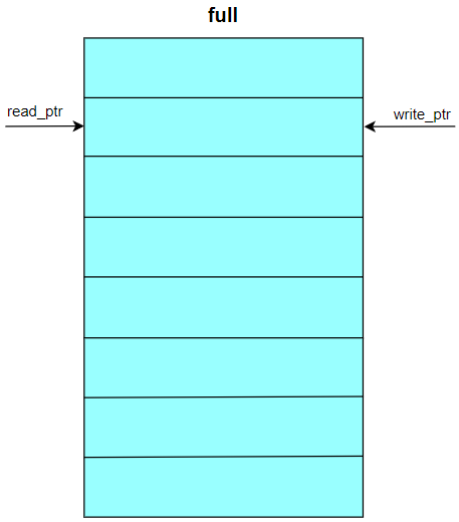
FIFO adından da anlaşılacağı üzere ilk giren ilk çıkar mantığı ile çalışan bir tür veri depolama elemanıdır. Tam olarak veri depolama da diyemeyiz aslında. Biraz daha kuyruğa alma işlemi dersek daha doğru bir tabir kullanmış oluruz. Mantıksal olarak yapılan işlem sadece ilk gelen veri ilk olarak okunur yani çıkmış olur. Bu sıraya alma işleminin belli başlı sebepleri vardır. Bu sebeplerden önce biraz daha FIFO'yu tanımamız gerekirse aşağıdaki görsel biraz daha mantıksal olarak oturmasına sebep olacaktır.



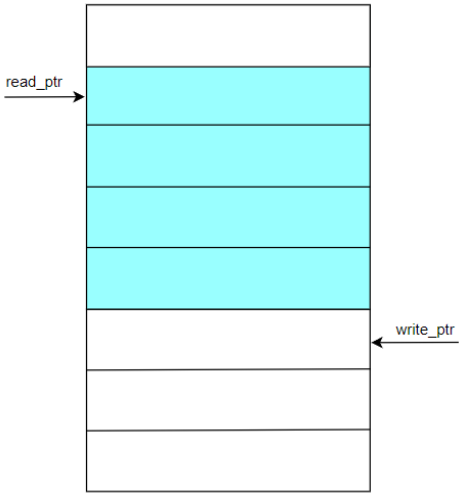
Yukarıdaki basit bir FIFO gösterimidir. Bu FIFO'nun derinliği 8'dir. Kaç bitlik değer yazılacağı da opsiyonel olabilir. Mesela yazılacak olan değer 4 bitlik olsun. Bu FIFO'ya 8 tane 4 bitlik değer sırayla yazılabilir. Şu anda yukarıda gözüken boş bir FIFO görüntüsünü temsil etmektedir. Bunun içerisine veri yazıldıkça içerisi dolmaya başlar. Yine örnek olarak 5 adetlik bir veri yazıldığını düşündüğümüz zaman FIFO görüntüsü şu hali alır:



Henüz okuma yapılmadığı için read_ptr olduğu yerde durmaya devam eder. Okuma yapıldığı zaman da gösterdiği kutucuk boşanır. İlk durum FIFO'nun empty yani boş durumudur. Bu empty condition olarak adlandırılır. Read ve write pointerlar aynı pozisyonda birlikte dururlar. Bir de bu ikisinin aynı pozisyonda olduğu durum full yani tam dolu olduğu durum vardır. Buna da full condition denir.



Eğer 5 adet veri yazılmışsa ve sadece 1 adet veri okunmuş olursa da FIFO görüntüsü aşağıdaki halini alacaktır.



Fiziksel olarak düşündüğümüz zaman ilk yazılan veri ilk okunmuş olacaktır. Böylece first in-first out tanımına uygun bir görsel halini almaktadır.

Yukarıda FIFO derinliğinden ve genişliğinden bahsetmiştik şimdi biraz daha bunu açmak gerekirse; FIFO'nun derinlik ve genişliği, sistemin kullanım amacına ve belirlenen gereksinimlere göre değişiklik gösterebilir. Mühendislikte sıklıkla olduğu gibi burada da

minimum maliyet ve **maksimum verim** ilkesi doğrultusunda bir tasarım yaklaşımı benimsenmelidir.

Bir FIFO tasarımı yapılırken şu prensipler dikkate alınmalıdır:

1. **Gereksinimleri Karşılama:** FIFO'nun derinliği ve genişliği, sistemin veri aktarımı sırasında karşılaşabileceği en kötü durumları ele alabilecek şekilde planlanmalıdır. Gereğinden küçük bir FIFO, veri kaybına veya sistem darboğazlarına neden olabilir.
2. **Optimum Boyutlandırma:** Gereğinden büyük bir FIFO ise işlevsel olarak isteneni karşılasa da özellikle FPGA üzerinde kullanılan **RAM** ve **LUT** kaynaklarını gereksiz yere tüketecektir. FPGA tasarımlarında kaynakların sınırlı olduğu göz önüne alındığında gereksinimlere uygun boyutlarda bir FIFO seçimi kritik öneme sahiptir.

Sonuç olarak, FIFO'nun boyutlandırılması sırasında sistem gereksinimleri detaylı bir şekilde analiz edilmeli ve tasarım, kaynakların verimli kullanımı ile belirlenen gereksinimler arasında bir denge kurularak gerçekleştirilmelidir. Bu yaklaşım, sistem performansını optimize ederken aynı zamanda FPGA kaynaklarının etkin bir şekilde kullanılmasını sağlar.

FIFO içerisine yazılacak verilerin bit büyüklüğü (genişlik(**width**)), sistemde taşınacak veri türüne ve boyutuna göre belirlenir. Bu genişlik, verinin doğru bir şekilde işlenmesi ve aktarılması için önemli bir tasarım parametresidir.

Örneğin:

- UART ile veri aktarımı genellikle 8 bitlik çerçeveler halinde gerçekleştirilir. Bu durumda, FIFO'nun genişliğini **8 bit** olarak tasarlamak en uygun çözüm olacaktır. Bu şekilde, FIFO hem gönderilecek hem de alınacak veriyi doğrudan destekler ve herhangi bir ek dönüştürme işlemi gerektirmez. Fakat yine istenen parametrelere göre farklı da tasarlanabilir, yani illa 8 bit olmak zorunda değildir. Yapılacak projedeki isterlere göre farklılıklar olabilir.

Sonuç olarak, FIFO'nun genişliği, taşınacak verilerin özelliklerine ve sistem gereksinimlerine uygun olarak belirlenmelidir. Bu yaklaşım, tasarımın hem basit hem de kaynak kullanımı açısından verimli olmasını sağlar.

Not: Eğer belirli bir uygulama için FIFO tasarımı yapılıyorsa, veri trafiği analiz edilerek maksimum veri akışı ve bekleme süreleri hesaplanmalı, FIFO derinlik ve genişliği buna göre belirlenmelidir.

Çok basit temelden FIFO derinliğini nasıl hesaplamamız gerektiğinden bahsedeceğim. Daha detaylı halleri internette birçok sitede mevcuttur oralardan bulabilirsiniz.

Formül: Derinlik ≥ (Yazma Hızı - Okuma Hızı) × Gecikme Süresi

- Yazma Hızı: 1 milyon kelime/saniye
- Okuma Hızı: 800 bin kelime/saniye
- Gecikme Süresi: 10 ms
- Derinlik = (1M - 0.8M) × 0.01 = 2000 kelime

Bu örnekte basitçe hesaplama yapılmıştır. FPGA'da çalışılırken frekans hesapları yapılacağı unutulmamalıdır ve hesaplamalar öyle yapılmalıdır. Burası sadece yönerge oluşturmak için yazılmıştır.

Peki bu kadar FIFO diye bahsettik ama ne işe yara bu FIFO, neden verileri sıraya almaya ihtiyaç vardır, neden böyle bir tasarıma ihtiyaç duyulmuştur? Gibi soruların cevabını aramak için biraz daha konuyu irdeleyelim.

Daha önce de bahsettiğimiz gibi FIFO ilk gelen veriyi ilk olarak yazar yani ekler ve daha sonra o yazdığı sıradan okur yani onu da oradan çıkartır. Okuduktan sonra da o FIFO ne kadar okunduysa o kadar boşanmış olur. FIFO'nun temel özelliklerinden şu şekilde bahsedebiliriz:

1. **Ekleme (Write/Enqueue):** Veri kuyruğun sonuna eklenir.
2. **Çıkarma (Read/Dequeue):** Veri kuyruğun başından çıkarılır.
3. **Tam/Boş Durumu:** FIFO, dolu olduğunda daha fazla veri kabul edemez; boş olduğunda çıkarılacak veri kalmaz.
4. **Sabit boyut:** Çoğu durumda, FIFO'lar önceden tanımlanmış bir derinliğe sahiptir.

Kullanım Alanları:

- **Veri tamponları:** Donanım ve yazılım arasında veri transferini senkronize etmek için.
- **İşletim sistemleri:** İşlem planlama ve veri akışında.
- **Ağ iletişimi:** Paketlerin sırasını koruyarak işlenmesi.
- **Dijital elektronik:** FPGA'ler ve ASIC'lerde, modüller arası veri akışını düzenlemek için.

Ayrıca FIFO'nun senkron ve asenkron olmak üzere iki türü vardır. Bu ikisi arasındaki temel fark adlarından da anlaşılacağı üzere senkron veya asenkron çalışmalarıdır.

- **Senkron FIFO:** Okuma ve yazma işlemleri aynı saat işaretiyle çalışır.

- **Asenkron FIFO:** Okuma ve yazma işlemleri farklı saat işaretleriyle çalışır. Saat alanlarını senkronize etmek için ek mantık gereklidir.

Konuyu biraz daha basit teorik olarak anlatmak gerekirse daha doğrusu FPGA'da neden kullanırızı şu şekilde anlatabiliriz:

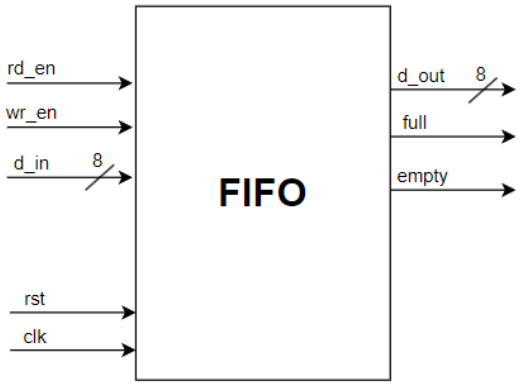
- **Saat Alanlarının Uyumsuzluğu (Clock Domain Crossing - CDC):**
  FPGA içindeki farklı modüller, farklı saat hızlarında çalışabilir. FIFO, bu farklı saat alanları arasında güvenilir bir veri aktarımı sağlamak için kullanılır.
- Örneğin: Bir modül 100 MHz, diğeri 50 MHz'de çalışıyorsa FIFO, bu iki modül arasındaki veri alışverişini tamponlar.
- **Veri Hızlarının Uyumsuzluğu:**
  Veri yazma ve okuma hızları farklı olabilir. FIFO, bu hız farklılıklarını dengelemek için bir tampon görevi görür.
- Örneğin: Yüksek hızlı bir ADC'nin ürettiği veriyi düşük hızlı bir işlemciye aktarmak için.
- **Geçici Veri Depolama:**
  Verinin işlenmek üzere bekletilmesi gerektiğinde FIFO kullanılır. Bu, özellikle veri akışlarının kontrol edilmesinde faydalıdır.
- Örneğin: Video işlem modülleri arasında karelerin geçici olarak tutulması.
- **Pipeline Tasarımları:**
  Paralel veri işleme sırasında, FIFO, veri akışını düzenler ve performansı artırır.

Bu bilgiler ile birlikte FIFO'nun genel olarak tanımını yapıp kullanıldığı yerlerden bahsettik. FIFO türlerinden senkron ve asenkron FIFO tasarımını VHDL ile gerçekleştireceğiz. Öncelikle senkron FIFO ile başlayabiliriz.

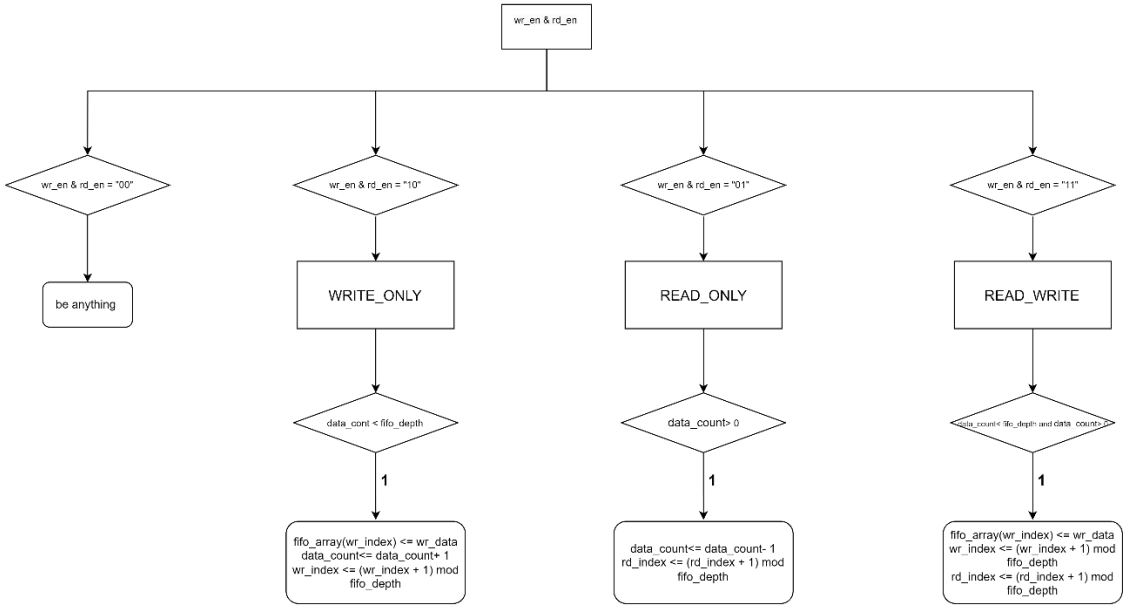
Senkron FIFO

FIFO tanımını daha önce yaptığımız için burada teorik bilgiden daha çok FPGA tarafında tasarımımızı nasıl gerçekleştireceğimizden bahsedeceğim. Bu FIFO adından da anlaşılacağı üzere yazma ve okuma işlemlerini aynı saat kaynağı ile gerçekleştirir. Aynı saat diliminde çalışan modüller arasında veri aktarımı için tercih edilir.

FIFO da aslında bir bellek yapısıdır fakat kullanım alanları diğer bellek yapılarından farklı olabilir. Sonuçta bu da diğer bellek yapıları gibi içerisinde veri depolamaktadır. Öncelikle bizim tasarlayacağımız FIFO'nun temel blok şeması aşağıda görülmektedir. Bu bloğa göre bir tasarım gerçekleştirilecektir. 8 bitlik giriş yapısına sahip bir FIFO'dur.



FIFO hakkında verdiğimiz bilgilerden yola çıkarak bize yol göstermesi ve daha kolay da bir tasarım yapmamız için küçük bir algoritmik durum şeması oluşturabiliriz. Bunu da aşağıda görebilirsiniz.



Tasarımı yaparken biz sayısal tasarımcılar için basit olması açısından bu şekilde algoritmalar oluşturmak işleri her zaman kolaylaştıracaktır. Gözle görülebilen bu şekilde algoritma oluşturmak kodu yazmada kolaylık sağlayacaktır. Burada da 4 adet farklı durum görülmektedir. Bu durumlardan birincisi;

- wr & rd “00” olma durumudur. Bu durumda hiçbir işlem gerçekleşmeyecektir.

-wr & rd “10” olma durumunda sadece yazma işlemi gerçekleşecektir.

-wr & rd “01” olma durumunda sadece okuma işlemleri gerçekleşecektir.

-wr & rd ”11” olma durumunda hem okuma hem yazma işlemleri gerçekleşecektir.

Burada yazan durumlar gerçekleşirken de altında verilen yönergeler doğrultusunda gerçekleşmektedir. Örnek olarak;

-wr & rd “00” olduğunda sadece yazma yapılır, data_count fifo_depth’den büyük ise altındaki bulunan işlemler yapılır ve atamalar gerçekleştirilir.

Bu örnekteki gibi diğer durumlar da teker teker tanımlanmalıdır. Şimdi tasarlanan VHDL koduna geçip neleri ne için yaptığımıza detaylı bir şekilde bakmaya çalışalım. Öncelikle kodun tam halini ardından da detaylı incelemesini aşağıda görebilirsiniz.

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

entity sync_fifo is

    generic (
                fifo_width      :   natural := 8;
                fifo_depth      :   integer := 16
            );

    port (
            d_in_i  : in std_logic_vector ((fifo_width - 1) downto 0);
            wr_en_i : in std_logic;
            rd_en_i : in std_logic;
            clk_i   : in std_logic;
            rst_i   : in std_logic;
            d_out_o : out std_logic_vector ((fifo_width - 1) downto 0);
            full_o  : out std_logic;
            empty_o : out std_logic
        );

end sync_fifo;
```

```
architecture Behavioral of sync_fifo is

type fifo_array_type is array (0 to fifo_depth - 1) of std_logic_vector
(fifo_width - 1 downto 0);
signal s_fifo_array     : fifo_array_type := (others =>(others => '0'));
signal s_wr_index       : integer range 0 to fifo_depth - 1 := 0;
signal s_rd_index       : integer range 0 to fifo_depth - 1 := 0;
signal s_data_count     : integer range -1 to fifo_depth + 1 := 0;
signal s_full           : std_logic;
signal s_empty          : std_logic;

begin

    process (clk_i, rst_i)

    begin
            if rst_i = '1' then
                s_data_count    <= 0;
                s_wr_index  <= 0;
                s_rd_index  <= 0;
            elsif rising_edge (clk_i) then
                case std_logic_vector'(wr_en_i & rd_en_i) is

                    when "00" =>
                        null;

                    --read
                    when "01" =>
                        if s_data_count > 0 then
                            s_data_count <= s_data_count - 1;
                            s_rd_index <= (s_rd_index + 1) mod fifo_depth;
                        end if;

                    --write
                    when "10" =>
                        if s_data_count < fifo_depth then
                                s_fifo_array(s_wr_index) <= d_in_i;
                                s_data_count <= s_data_count + 1;
                                s_wr_index <= (s_wr_index + 1) mod fifo_depth;
                        end if;

                    --read & write
                    when "11" =>
                        if s_data_count < fifo_depth and s_data_count > 0 then
                                s_fifo_array(s_wr_index) <= d_in_i;
                                s_wr_index <= (s_wr_index + 1) mod fifo_depth;
                                s_rd_index <= (s_rd_index + 1) mod fifo_depth;
                        end if;

                    when others =>
                        null;
                end case;
            end if;
    end process;

    d_out_o <= s_fifo_array(s_rd_index) when (s_data_count > 0) else (others =>
'0');
    s_full <= '1' when s_data_count = fifo_depth else '0';
    s_empty <= '1' when s_data_count = 0 else '0';

    full_o <= s_full;
    empty_o <= s_empty;

end Behavioral;
```

Detaylı analiz:

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

entity sync_fifo is

    generic (
                fifo_width      :   natural := 8;
                fifo_depth      :   integer := 16
            );

    port (
            d_in_i  : in std_logic_vector ((fifo_width - 1) downto 0);
            wr_en_i : in std_logic;
            rd_en_i : in std_logic;
            clk_i   : in std_logic;
            rst_i   : in std_logic;
            d_out_o : out std_logic_vector ((fifo_width - 1) downto 0);
               full_o  : out std_logic;
               empty_o : out std_logic
          );

end sync_fifo;
```

Burada önceki sayfalarda blok şemasını oluşturduğumuz FIFO'nun giriş çıkışlarının tanımlamasını yapıyoruz. Ondan önce ise kullanacağımız FIFO yapısını **generic** olarak değerler ile belirtiyoruz. Bunu bu şekilde yapmamızın sebebi daha sonraki seferlerde bu FIFO tasarımının sadece istenen boyutlara göre değerlerini değiştirmek olacaktır. Burada örnek olması için **fifo_width** = 8 varsayılan olarak tanımlanmıştır. Bunun anlamı da FIFO içerisine yazılacak olan verilerin kaç bit olacağıdır. Görsel olarak düşündüğümüzde 8 bit yani 8 birim genişliğinde bir tanımlama yaptık. Yine buradan yola çıkarak **fifo_depth** yani bizim FIFO'muzun derinliği daha anlamlı açıklamak gerekirse bu FIFO içerisine kaç adet 8 bitlik veri yazabilirizin tanımlamasıdır. FIFO derinliğimiz de varsayılan olarak 16 verilmiştir. Yani bizim FIFO'muz 8 genişliğinde ve 16 derinliğinde bir tasarımdır. Burada **natural** ve **integer** farklarına da kısaca değinecek olursak **integer** demek negatif veya pozitif tam sayı değerleri gerekli olduğu zaman bu yazılır daha kapsamlıdır, **natural** ise sadece pozitif ve sıfır değerleri gerekli olduğu zaman kullanılır. İkisi de **natural** olsa daha güvenli olabilir fakat ikisini birlikte kullanarak burada farklı yazma tekniklerini de göstermek istedim. Diğer giriş çıkış portlarının kısaca anlamları ise şu şekildedir;

- **`d_in_i`:** FIFO'ya yazılacak veriler için giriş. Bu verinin genişliği de başta belirlediğimiz generic parametrelerinden olan FIFO genişliği yani **fifo_width** ' e verilen değer ile belirtilmektedir.
- **`wr_en_i`:** Yazma işlemini etkinleştirmek için giriş kontrol sinyali.
- **`rd_en_i`:** Okuma işlemini etkinleştirmek için giriş kontrol sinyali.
- **`clk_i`:** FIFO'yu senkronize etmek için saat sinyali.
- **`rst_i`:** FIFO'yu sıfırlamak için reset sinyali.

- **d_out_o:** FIFO'dan okunan veriler için çıkış.
- **full_o:** FIFO'nun dolu olup olmadığını belirtir.
- **empty_o:** FIFO'nun boş olup olmadığını belirtir.

```
architecture Behavioral of sync_fifo is

type fifo_array_type is array (0 to fifo_depth - 1) of std_logic_vector
(fifo_width - 1 downto 0);
signal s_fifo_array     : fifo_array_type := (others =>(others => '0'));
signal s_wr_index       : integer range 0 to fifo_depth - 1 := 0;
signal s_rd_index       : integer range 0 to fifo_depth - 1 := 0;
signal s_data_count     : integer range -1 to fifo_depth + 1 := 0;
signal s_full           : std_logic;
signal s_empty          : std_logic;
```

Burada mimari yapımızın tasarımına geçmeden önce sinyal tanımlamalarımızı yapıyoruz. Sırayla neyi ne için verdiğimizi açıklamaya çalışırsak şu şekilde açıklayabiliriz;

type fifo_array_type Bu ifade yeni bir veri türü tanımlar. `fifo_array_type`, artık bir isim olarak kullanabileceğiniz özel bir türdür. Türün amacı, FIFO belleği bir dizi olarak temsil etmektir. **is array (0 to fifo_depth - 1)** FIFO belleği bir dizi (array) olarak ifade ediliyor. Dizinin indeks aralığı, `0` ile `(fifo_depth - 1)` arasındadır. `fifo_depth` ise bizim yukarıda tanımladığımız bir generic parametre olduğu için FIFO'nun derleme sırasında belirtilen derinliğe (satırdaki toplam hücre sayısı) göre boyutlandırılmasını sağlar. Diğer sinyallerin açıklaması ise kısaca şöyledir;

- **s_wr_index:** Yazma işlemi için dizinin hangi indeksine veri yazılacağını gösterir.
- **s_rd_index:** Okuma işlemi için dizinin hangi indeksinden veri okunacağını gösterir.
- **s_data_count:** FIFO'daki mevcut veri miktarını tutar.
- **s_full:** FIFO'nun dolu olduğunu belirten sinyal.
- **s_empty:** FIFO'nun boş olduğunu belirten sinyal.

```
begin

    process (clk_i, rst_i)

    begin
            if rst_i = '1' then
                s_data_count    <= 0;
                s_wr_index  <= 0;
                s_rd_index  <= 0;
            elsif rising_edge (clk_i) then
                case std_logic_vector'(wr_en_i & rd_en_i) is

                    when "00" =>
                        null;

                    --read
                    when "01" =>
                              if s_data_count > 0 then
                            s_data_count <= s_data_count - 1;
                                  s_rd_index <= (s_rd_index + 1) mod fifo_depth;
                              end if;
```

Burada artık yapılacak işlemlerimiz teker teker tanımlanır. Asıl işlemler burada yapılır. Hangi durumda hangi davranışı

göstereceğini burada belirleriz. Bunu da yazmak için daha önce küçük bir durum makinesi oluşturmuştuk. Bunu da yukarıda görsel olarak koymuştuk. Oradaki görsel yardımı ile burada bulunan işlemleri yazmak çok daha basit olacaktır. Şimdi yazılanların anlamını kısaca açıklamaya, anlatmaya çalışalım:

Eğer (`rst_i = '1'`) ise, FIFO tamamen temizlenir:

- `s_data_count` sıfırlanır.
- Yazma (`s_wr_index`) ve okuma indeksleri (`s_rd_index`) sıfırlanır.

değil ise ; Her saat sinyalinin yükselen kenarında (`rising_edge(clk_i)`), `wr_en_i` ve `rd_en_i` kontrol sinyallerine göre işlem yapılır.

Durumlar(Case yapısı):

- "00" (Ne okuma ne yazma**):** FIFO'da hiçbir işlem yapılmaz.
- "01" (Okuma**):**
  - FIFO boş değilse (`s_data_count > 0`), veri okunur.
  - Okuma indeksi (`s_rd_index`) bir artırılır.
  - Veri sayacı (`s_data_count`) bir azaltılır.

Son satırdaki **mod** operatörünün biraz açıklamak gerekirse, VHDL'de modülüs (kalan bulma) işlemi yapmak için kullanılır. Bu operatör, bir sayıyı başka bir sayıya böldüğünde kalan kısmı döndürür dersek yanlış olmaz. Buradaki kullanma amacını ise; "FIFO'nun okuma ve yazma indeksleri (yani **s_rd_index** ve **s_wr_index**) **fifo_depth** sınırlarını aşabilir. Bu sınırlar aşıldığında, mod operatörü bu indeksleri 0'dan başlamaya zorlar, böylece indeksler dairesel bir şekilde çalışır." açıklayabiliriz. Bizim buradaki FIFO derinliğimiz 16 olduğundan indiksler 0 ile 15 arasında dönerek çalışır. Yazma indeksi (**s_wr_index**) her yeni veri yazıldığında bir artırılır. Ancak FIFO'nun son hücresine (örneğin, 15) veri yazıldıktan sonra bir sonraki veri ilk hücreye (0) yazılır. Benzer şekilde, okuma indeksi (**s_rd_index**) de FIFO'nun son hücresinden veri okunduktan sonra tekrar başa döner.

- lk yazma işlemi: `s_wr_index = 0` (başlangıç).
- İkinci yazma işlemi: `(0 + 1) mod 4 = 1`.
- Üçüncü yazma işlemi: `(1 + 1) mod 4 = 2`.

.

.

- On altıncı yazma işlemi: `(15 + 1) mod 4 = 0` (dizinin başına döner).

Böylece FIFO sınırını aşmadan tekrardan başa sarmasını sağlamış oluruz.

- **"10" (Yazma):**

```
--write
          when "10" =>
              if s_data_count < fifo_depth then
                      s_fifo_array(s_wr_index) <= d_in_i;
                      s_data_count <= s_data_count + 1;
                      s_wr_index <= (s_wr_index + 1) mod fifo_depth;
              end if;

 --read & write
          when "11" =>
              if s_data_count < fifo_depth and s_data_count > 0 then
                     s_fifo_array(s_wr_index) <= d_in_i;
                     --s_fifo_array(s_rd_index) <= (others => '0');
                     s_wr_index <= (s_wr_index + 1) mod fifo_depth;
                     s_rd_index <= (s_rd_index + 1) mod fifo_depth;
                end if;
            when others =>
                        null;
                end case;
              end if;
    end process;
```

•FIFO dolu değilse (**`s_data_count < fifo_depth`**), giriş verisi FIFO'ya yazılır.
•Yazma indeksi (**`s_wr_index`**) bir artırılır.
•Veri sayacı (**`s_data_count`**) bir artırılır.

- **"11" (Hem okuma hem yazma):**

  - FIFO hem dolu değilse hem de boş değilse (**`0 < s_data_count < fifo_depth`**), aynı anda okuma ve yazma işlemleri yapılır.
  - Yazma ve okuma indeksleri ayrı ayrı artırılır.

Ardından böylece yapılacak işler tanımlanmış olur. If be case blokları da kapatılır. When others durumunun yazılması başka istenmeyen bir durum oluştuğunda onunla işlem yapılmaması istendiğindendir. Biz normalde oluşabilecek 4 durumu da tanımladık fakat daha büyük tasarımlarda tüm durumları kontrol edemeyeceğimizden veya örnek olarak diyelim ki oluşabilecek 16 durumdan sadece 5'inde işlem yapacağız, böyle durumlarda tercih edilir diğer durumlar bir daha teker teker yazılmak zorunda kalınmaz veya hata şansını azaltır. Burada sadece el alışkanlığı olsun, göz aşinalığı olsun diye yapılmıştır.

```
d_out_o <= s_fifo_array(s_rd_index) when (s_data_count > 0) else (others => '0');
    s_full <= '1' when s_data_count = fifo_depth else '0';
    s_empty <= '1' when s_data_count = 0 else '0';

    full_o <= s_full;
    empty_o <= s_empty;

end Behavioral;
```

Burada da çıkış atamaları yapılmıştır. FIFO'nun dolu veya boş durumlarını da yine görmek için sinyalden asıl portlara atama yaparak gerçekleştirilmiştir.

Şimdi bu FIFO'nun testi için basit bir testbench yazıp gerçekleştirelim ve hangi durumlarda nasıl davranışlar sergilediğini gözlemlemeye çalışalım.

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.numeric_std.all;


entity tb_sync_fifo is

end tb_sync_fifo;

architecture Behavioral of tb_sync_fifo is

    constant c_fifo_depth : natural := 16;
    constant c_fifo_width : integer := 8;

    signal  s_d_in      :  std_logic_vector (c_fifo_width-1 downto 0);
    signal  s_wr_en     :  std_logic := '0';
    signal  s_rd_en     :  std_logic := '0';
    signal  s_clk       :  std_logic := '0';
    signal  s_rst       :  std_logic := '0';
    signal  s_d_out     :  std_logic_vector (c_fifo_width-1 downto 0);
    signal  s_full      :  std_logic;
    signal  s_empty     :  std_logic;



    constant clk_i_period : time := 10ns;


    component sync_fifo is

        generic(
                fifo_width : natural := 8;
                fifo_depth : integer := 16
                );

        port (
                d_in_i  : in std_logic_vector ((fifo_width - 1) downto 0);
                wr_en_i : in std_logic;
                rd_en_i : in std_logic;
                clk_i   : in std_logic;
                rst_i   : in std_logic;
                d_out_o : out std_logic_vector ((fifo_width - 1) downto 0);
                full_o  : out std_logic;
                empty_o : out std_logic
            );
    end component;


    begin

    uut: sync_fifo
    generic map(
                    fifo_width => c_fifo_width,
                    fifo_depth => c_fifo_depth
                    )

        port map(
                    d_in_i  => s_d_in,
                    wr_en_i => s_wr_en,
                    rd_en_i => s_rd_en,
                    clk_i   => s_clk,
                    rst_i   => s_rst,
                    d_out_o => s_d_out,
                    full_o  => s_full,
                    empty_o => s_empty
                );

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

```vhdl
        --test

    test_process:   process
                        begin
                        s_rst <= '1';
                        wait for
clk_i_period*2;
                        s_rst <= '0';
                        wait for
clk_i_period*2;

                        s_wr_en <= '1';
                        s_d_in <= X"A3";
                        wait for
clk_i_period*4;
                        s_wr_en <= '0';
                        s_rd_en <= '1';
                        wait for
clk_i_period*3;

                        s_d_in <= X"B7";
                        s_rd_en <= '0';
                        s_wr_en <= '1';
                        wait for
clk_i_period*2;

                        s_d_in <= X"F5";
                        s_rd_en <= '1';
                        wait for
clk_i_period*10;
                        s_wr_en <= '0';
                        s_rd_en <= '0';
                        wait;
                    end process;

    end Behavioral;
```

Detaylı açıklamaya gelecek olursak;

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


entity tb_sync_fifo is
end tb_sync_fifo;

architecture Behavioral of tb_sync_fifo is

    constant c_fifo_depth : natural := 16;
    constant c_fifo_width : integer := 8;

    signal  s_d_in      :  std_logic_vector (c_fifo_width-1
downto 0);
    signal  s_wr_en     :  std_logic := '0';
    signal  s_rd_en     :  std_logic := '0';
    signal  s_clk       :  std_logic := '0';
    signal  s_rst       :  std_logic := '0';
    signal  s_d_out     :  std_logic_vector (c_fifo_width-1
downto 0);
    signal  s_full      :  std_logic;
    signal  s_empty     :  std_logic;
    constant clk_i_period : time := 10ns;
```

En başta kullanılacak kütüphaneler yazılır ardından entity kısmı boş bırakılır. Test bench yazılırken entity yazılmaz. Mimari kısma başlanır ve buradaki **constant** tanımlamaları yapılır.

- **`c_fifo_depth`**: FIFO'nun derinliği (kapasitesi). Bu, FIFO'nun 16 veri öğesi depolayabileceğini belirtir.
- **`c_fifo_width`**: FIFO'nun genişliği (veri genişliği). Bu, FIFO'nun her öğesinin 8 bit olduğunu belirtir.
- **`s_d_in`**: FIFO'ya yazılacak veri (8 bit).
- **`s_wr_en`**: Yazma izni sinyali.
- **`s_rd_en`**: Okuma izni sinyali.
- **`s_clk`**: Saat sinyali.
- **`s_rst`**: Sıfırlama sinyali.
- **`s_d_out`**: FIFO'dan okunacak veri (8 bit).
- **`s_full`**: FIFO'nun dolu olduğunu belirten sinyal.
- **`s_empty`**: FIFO'nun boş olduğunu belirten sinyal.
- **`clk_i_period`**: Saat sinyalinin periyodunu 10ns olarak belirler.

```
    component sync_fifo is

        generic(
                fifo_width : natural := 8;
                fifo_depth : integer := 16
                );

        port (
                d_in_i  : in std_logic_vector ((fifo_width - 1) downto
0);
                wr_en_i : in std_logic;
                rd_en_i : in std_logic;
                clk_i   : in std_logic;
                rst_i   : in std_logic;
                d_out_o : out std_logic_vector ((fifo_width - 1) downto
0);
                full_o  : out std_logic;
                empty_o : out std_logic
            );
    end component;
```

Burada bizim yazdığımız FIFO modülünün bileşenlerini içerir. Component olarak bağlamamız gerekiyor bu yüzden bu tanımlamayı yapmalıyız.

```
    begin

    uut: sync_fifo
    generic map(
                    fifo_width => c_fifo_width,
                    fifo_depth => c_fifo_depth
                    )

        port map(
                    d_in_i  => s_d_in,
                    wr_en_i => s_wr_en,
                    rd_en_i => s_rd_en,
                    clk_i   => s_clk,
                    rst_i   => s_rst,
                    d_out_o => s_d_out,
                    full_o  => s_full,
                    empty_o => s_empty
                );
```

Burada ise test bench modülünün bir üst modül gibi davranmasını sağlamak adına bağlantılarımızı gerçekleştiririz. Yukarıda test bench için tanımlanan sinyaller ile FIFO modülünün giriş çıkışları arsındaki bağlantılar burada gerçekleştirilir.

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

Burada bir saat sinyali üretimi için process oluşturuluyor. Buradaki **`clk_i_period`** olarak belirtilen değere göre saat çalışacaktır.

```
    test_process:   process
                        begin
                        s_rst <= '1';
                        wait for
clk_i_period*2;
                        s_rst <= '0';
                        wait for
clk_i_period*2;

                        s_wr_en <= '1';
                        s_d_in <= X"A3";
                        wait for
clk_i_period*4;
                        s_wr_en <= '0';
                        s_rd_en <= '1';
                        wait for
clk_i_period*3;

                        s_d_in <= X"B7";
                        s_rd_en <= '0';
                        s_wr_en <= '1';
                        wait for
clk_i_period*2;

                        s_d_in <= X"F5";
                        s_rd_en <= '1';
                        wait for
clk_i_period*10;
                        s_rd_en <= '0';

                        s_d_in <= X"C3";

                        wait for
clk_i_period*5;

                        s_wr_en <= '0';
                        wait;
                    end process;
```

Burada bir adet test süreci oluşturulmuştur. FIFO'yu test ettiğimiz senaryo burasıdır. İlk olarak sıfırlama yapılır (`s_rst <= '1'`).

Ardından veriler sırasıyla FIFO'ya yazılır ve okunur. Çıkıştan da okunan değeri görmemiz gerekir.

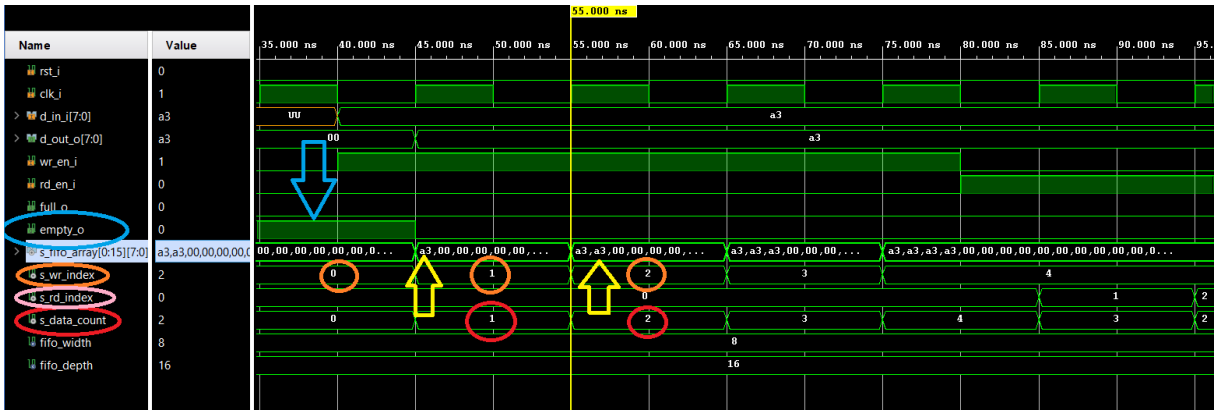
X"A3" verisi FIFO'ya yazılır.

FIFO'dan veri okunur.

X"B7" verisi FIFO'ya yazılır.

X“F5” verisi FIFO'ya dolana kadar yazdırılır.

Bu senaryonun ardından simülasyon ekranından beklediğimiz çıktılar tam olarak istediğimiz gibidir. Bunun değerlerini de aşağıda görebilirsiniz.



Mavi ile belirtilen FIFO'nun boş olduğu durumda 1 olacaktı ve senaryo gerçekleşmiş.

Sarı oklar ile birlikte belirtilen değerler en yukarıdaki d_in portundan gelen değerler ve kırmızı ile belirtilen data_count arttıkça aslında değerlerinde sıra ile yazıldığını ve FIFO'muzun dolmaya başladığını görebiliyoruz.

Turuncu ile belirtilen yerlerde yazma sinyalimizin aktif olduğu anlarda wr_index değerinin artmaya yani yazma işlemini yapmaya devam ettiğinden emin olabiliyoruz.

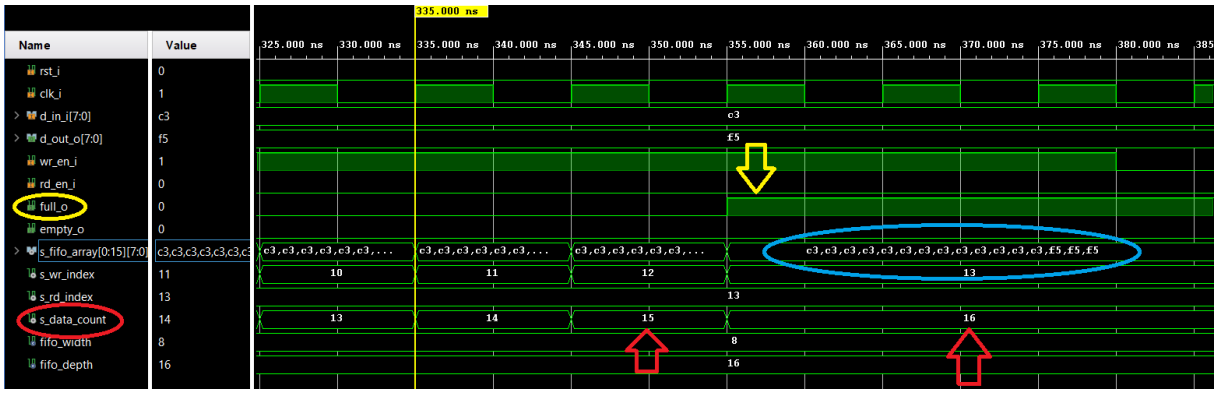
Pembe yuvarlak ise henüz okuma işlemine başlanmadığından rd_index değeri 0 gözüküyor.

Senaryonun sonraki anlarını görmek için biraz daha ilerilere bakalım.

Burada kırmızı ile belirtilen data_count artık okuma yapıldığından azalmaya başlamıştır yani 1 eksilmeye başlamıştır.

Pembe ile belirtilen kısımda okuma işleminin yapıldığını ve rd_index 'in arttığını görebilirsiniz.

Ayrıca data_in değeri değişmiş ve FIFO'ya artık başka değerlerin yazılmaya başlandığını da mavi ile belirtilen kısımda görebilirsiniz.



Burada da yine kırmızı ile belirtine data_count ‘un artık son olarak 16 olduğunu yani fifo_depth ile eşitlendiğini ve sonrasında sarı ile gösterilen full sinyalini aktif edip artık dolu olduğunu söyleyip gelen veriyi mavi ile gösterilen yere yazmayacaktır. Yani artık FIFO doldu ve veri yazmamaktadır.

Simülasyon ekranından kontrol ettiğimizde de bu senaryolar doğru bir şekilde çalışmaktadır.