Asenkron FIFO

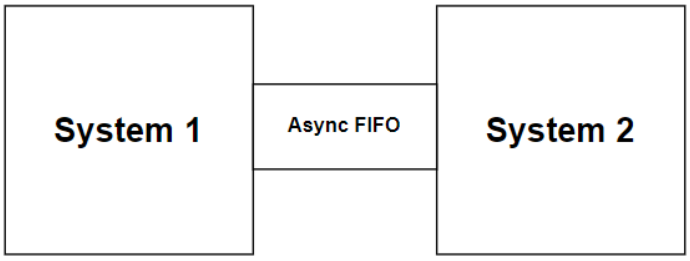
Şimdi FIFO'nun genel türlerinden olan asenkron FIFO hakkında bazı bilgiler göreceksiniz.

NOT: FIFO ne demek daha önce açıklamıştık o yüzden burada sadece asenkron FIFO hakkında bilgiler olacaktır. Eğer FIFO ile ilgili genel bilgileri öğrenmek isterseniz senkron FIFO dosyasında bulunan SYN_FIFO.pdf dosyasından bilgileri öğrenebilirsiniz.

Adından da anlaşılacağı üzere asenkron FIFO, yazma (write) ve okuma (read) işlemlerinin farklı saat domainlerinde gerçekleştirildiği FIFO türüdür. Bu tür FIFO genellikle iki farklı hızda çalışan sistem arasında veri aktarımı için kullanılır. Örneğin bir işlemci ve bir sensör ya da bir gönderici ve alıcı arasında veri alışverişi gerekiyorsa asenkron FIFO ideal bir çözüm sunar. Olayı biraz daha somutlaştırmak gerekirse 100 mHz frekansta çalışan bir genel sistemin içerisinde eğer 50 mHz frekansta çalışması gereken başka bir sistem var ise bu verilerin kayıpsız bir şekilde diğer sisteme aktarılmasını sağlayan bir veri depolama sistemidir. Olayın kafamızda daha net şekilde canlanmasını istersek şu şekilde düşünebiliriz:

Örnek olarak sürekli veri gönderen ve 100 birimlik hız aralıkları ile gönderen bir sistemimiz olsun. Bu sistemin içerisinde 50 birim hızı ile çalışabilme yeteneği olan farklı bir sistem olsun. 100 birimlik hız ile gelen verilerin 50 birim hız ile çalışan sistem tarafından okunması ve anlamlandırabilmesi için önce bu 100 birimlik hızın 50 birime düşürülmesi ve veri kaybının da yaşanmaması için bir sıraya koyulması gerekir. İşte bu yüzden bir asenkron FIFO tasarımı ile bunu gerçekleştirebiliriz. Bunu tam tersi 50'den 100'e olarak da düşünebiliriz.

Kısaca iki farklı saat frekansı ile çalışan sistemler arasında veri alışverişi yapılırken kullanılacak veriyi depolamayı sağlayan ve karşı sisteme aktaran yapının adına **Asenkron FIFO** adı verilir.



Asenkron FIFO'yu daha iyi anlamak için "Clock Domain Crossing(CDC)" konusuna hakim olmak fayda sağlayacaktır. Bu konu çok derin ve detaylıdır. Burada bunu anlatmaktan ziyade asenkron FIFO'yu anlatacağım. Bu yüzden bu konuya internet üzerinden birçok kaynaktan faydalanarak araştırabilir ve öğrenebilirsiniz.

Temel olarak asenkron FIFO'yu tasarlarken nelere dikkat ettiğimizi ve tasarımı nasıl gerçekleştirdiğimizi ele alalım:

1. Bellek Alanı

- FIFO, gelen veriyi depolamak için RAM (Block RAM veya Register) kullanır.
- FIFO'nun genişliği (veri boyutu(width)) ve derinliği (kapasitesi(depth)), sistemin gereksinimlerine göre belirlenir.

2. Yazma İşaretçisi (Write Pointer)

- Yazma işlemi sırasında, yazılacak bellek adresini belirler.
- Yazma domainindeki saat sinyaline bağlı çalışır.

3. Okuma İşaretçisi (Read Pointer)

- Okuma işlemi sırasında, okunacak bellek adresini belirler.
- Okuma domainindeki saat sinyaline bağlı çalışır.

4. Gray Kod Kullanımı

- Yazma ve okuma işaretçileri genellikle **binary** yerine **gray kod** kullanılarak tutulur.
    - Gray kod, bir adımda sadece bir bit değiştiği için senkronizasyon sırasında oluşabilecek belirsizlikleri azaltır.
    - Gray kodlu işaretçiler, karşı saat domainine senkronize edilir ve karşı domainde tekrar binary forma dönüştürülür.

5. Senkronizasyon Devresi

- Farklı saat domainlerindeki işaretçilerin birbirini doğru bir şekilde anlaması için senkronizasyon devreleri kullanılır.
- Yazma işaretçisi okuma domainine, okuma işaretçisi de yazma domainine taşınır.

- Bu işlem genellikle iki veya üç aşamalı flip-flop zincirleri ile yapılır.

6. Full ve Empty Bayrakları

- **Full (Dolu):** FIFO'nun tamamen dolu olduğunu belirtir. Yazma işlemi durur.
- **Empty (Boş):** FIFO'nun tamamen boş olduğunu belirtir. Okuma işlemi durur.
- Full ve Empty durumlarını belirlemek için:
  - **Write Pointer** ve **Read Pointer**'ın değerleri karşılaştırılır.
  - İşaretçiler eşitse FIFO boş, bir adım gerideyse dolu kabul edilir. İkisinin de eşit olup dolu olduğu durum da olabilir. Onun tasarımını da yaparken nasıl olduğunu ileride anlatacağım.

Gray kod nedir ve neden kullanılıra da değinecek olursak kısaca şu şekilde bahsedebiliriz:

Gray kod, sayılar arasında birer birimlik (artış ya da azalış) değişim olduğunda **yalnızca bir bitin değiştiği** özel bir ikili sayı sistemidir. Bu özellik, özellikle dijital sistemlerde ve donanım tasarımlarında, hata oranını azaltmak ve güvenilirliği artırmak için kullanılır. Bizim de özellikle gray kodunu kullanmamızın sebebi metastabilite ihtimalini en aza indirmektir. Gray kod sayesinde 4 bitlik değerde sadece 1 bit değer değişeceği için hata oranını en aza indirmiş oluyoruz.

- Gray kod, bir adımda yalnızca bir bit değiştiği için senkronizasyon sırasında metastabilite (kararsız durum) olasılığını azaltır.
- İşaretçilerin karşı domainde yanlış anlaşılmasını önler.

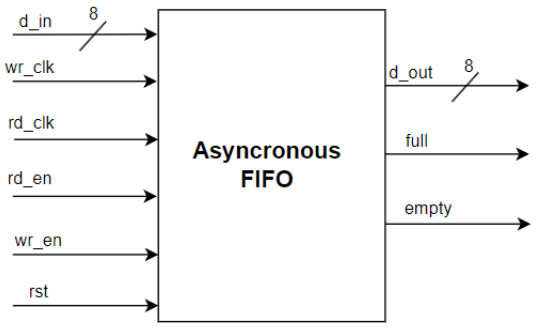
Farklı kodlar da Gray kod yerine kullanılabilir fakat en mantıklı ve hata payı en az olanı Gray kod olduğundan bu tercih edilir.
Temel olarak formülü şu şekildedir:

Binar'den Gray'e => **Gray[i] = Binary[i] XOR Binary[i+1]**

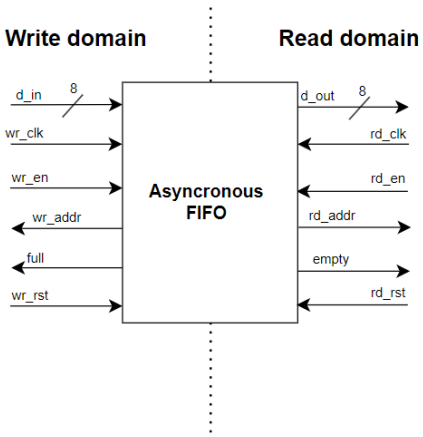
Gray'dan Binary'e => **Binary[i] = Gray[i] XOR Binary[i+1]**

Yine gray kod hakkında daha fazla bilgi edinmek için internette birçok kaynaktan faydalanabilirsiniz.

Tasarımımızda önce temel olarak kafamızda oturması adına kalıp bir blok tasarımını oluşturmakta fayda var. Daha sonrasında bunun daha detayına inip bir RTL şema oluşturacağız.



Tasarımımız, aşağıda verilen blok şemaya göre şekillenmektedir. Bu tasarım, standart bir FIFO'dan yalnızca iki ayrı saat sinyali (clk) barındırması ile ayrılmaktadır. FIFO'muz, farklı saat domainlerinde çalışacak şekilde tasarlandığından, **read** (okuma) ve **write** (yazma) işlemleri birbirinden bağımsız saat domainlerinde gerçekleştirilecektir. Bu yapı, asenkron bir FIFO'nun temel çalışma prensibini yansıtmaktadır. Tasarımı daha iyi anlamak ve çalışma mantığını görselleştirebilmek adına, aşağıdaki blok şema incelenebilir.



Oluşturulan tasarımın ana blok şeması yukarıda gösterildiği gibidir. Sistem şekilde de göründüğü üzere write ve read domaini olmak üzere iki adet domain içermektedir. Bunun anlamı iki farklı saat sinyali üzerinden hareket edilecek demektir.

Şimdi yavaşça daha detaylı şekilde tasarımımıza geçebiliriz. Daha detaylı ve anlaşılabilir olabilmesi açısından modüler bir yaklaşım ile hareket etmeye çalıştım. O yüzden farklı farklı modüller tasarlayıp bunları teker teker açıklamaya çalışacağım.

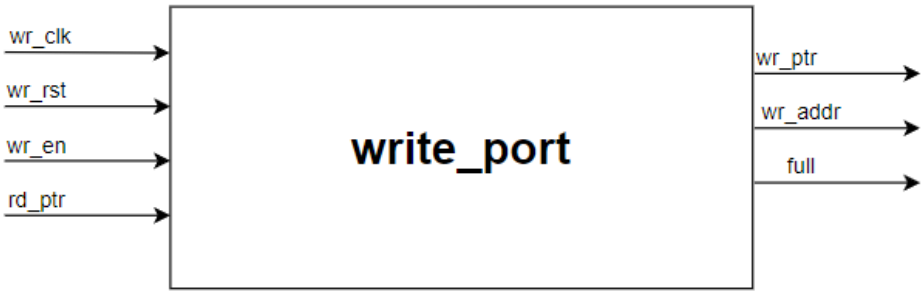
NOT: Daha önce de bahsettiğim üzere CDC(clock domain crossing) konusunu burada anlatmadım. Asenkron fifo tasarımı için öncelikle CDC konusunu internette farklı

kaynaklardan, kitaplarda, videolardan vb. yararlanabilirsiniz. Bu konu hakkında bilgiler için kaynakça bölümüne bakabilirsiniz.

Bu asenkron FIFO tasarımı içerisinde birçok farklı FPGA temelleri bir arada kullanılmıştır. Register, RAM gibi yapıları da görme imkanımız olacaktır.

Write Port Control

İlk olarak yazma domainindeki tasarımdan başlayalım. Yine daha detaylı olması açısından ve ne yaptığımızı daha iyi anlamak için genel hatları ile oluşturulan aşağıdaki blok şemayı inceleyebilirsiniz.



Yukarıdaki şekilde bu bloğun giriş ve çıkışlarını görebilirsiniz. Bu şemaya bakarak bizim oluşturacağımız kodun da nasıl yazıldığını bu şekilde anlayabilirsiniz.

Koda ve tasarıma geçmeden önce temel olarak **write_port** modülünde neler yapılması gerektiğinden bahsedelim. Bu bahsettiklerimden yukarıda da bahsetmiştim fakat bazı şeyleri bazen tekrar tekrar ve daha detaylı şekilde açıklamak öğrenme açısından etkili olabilmektedir. Burada biraz daha detaya gireceğiz.

Write Pointer FIFO'ya verilerin doğru sırayla yazılmasını ve yazma işleminin FIFO'nun kapasitesine uygun şekilde yönetilmesini sağlar. Yazma adresini takip eder ve FIFO doluluğunu kontrol etmek için gerekli bilgiyi sağlar.

Detaylı analize geçmeden önce kodun tamamını aşağıda görebilirsiniz.

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

entity write_port_control is
    generic (DEPTH: natural := 4
            );
    port (
        wr_clk_i    : in std_logic;
        wr_rst_i    : in std_logic;
        wr_en_i     : in std_logic;
        rd_ptr_i    : in std_logic_vector(DEPTH downto 0);
        wr_ptr_o    : out std_logic_vector(DEPTH downto 0);
        wr_addr_o   : out std_logic_vector(DEPTH-1 downto 0);
        full_o      : out std_logic
    );
end write_port_control;

architecture gray_arch of write_port_control is
    signal w_ptr_reg_s    : std_logic_vector(DEPTH downto 0);
    signal w_ptr_next_s   : std_logic_vector(DEPTH downto 0);
    signal gray_s         : std_logic_vector(DEPTH downto 0);
    signal bin_s          : std_logic_vector(DEPTH downto 0);
    signal bin1_s         : std_logic_vector(DEPTH downto 0);
    signal w_addr_msb_s   : std_logic;
    signal r_addr_msb_s   : std_logic;
    signal full_flag_s    : std_logic;

begin
    process (wr_clk_i, wr_rst_i)
    begin
        if (wr_rst_i = '1') then
            w_ptr_reg_s <= (others => '0');
        elsif (rising_edge(wr_clk_i)) then
            w_ptr_reg_s <= w_ptr_next_s;
        end if;
    end process;

    -- binary to gray
    bin_s   <= w_ptr_reg_s xor ('0' & bin_s (DEPTH downto 1));
    bin1_s  <= std_logic_vector(unsigned(bin_s) + 1);
    gray_s <= bin1_s xor ('0' & bin1_s(DEPTH downto 1));


    -- Update write pointer
    w_ptr_next_s <= gray_s when (wr_en_i = '1' and full_flag_s = '0') else
    w_ptr_reg_s;


    --DEPTH (full belirlenmesi icin)
    w_addr_msb_s <= w_ptr_reg_s(DEPTH) xor w_ptr_reg_s(DEPTH-1);


    -- Check for FIFO full_o
    r_addr_msb_s <= rd_ptr_i(DEPTH) xor rd_ptr_i(DEPTH-1);
    full_flag_s  <= '1' when rd_ptr_i(DEPTH) /= w_ptr_reg_s(DEPTH) and
                             rd_ptr_i(DEPTH-2 downto 0) = w_ptr_reg_s (DEPTH-2 downto 0) and
                             r_addr_msb_s = w_addr_msb_s else
                    '0';


    -- Outputs
    wr_addr_o <= bin_s(DEPTH-1 downto 0);
    wr_ptr_o  <= w_ptr_reg_s;
    full_o    <= full_flag_s;

end gray_arch;
```

Detaylı analizini gerçekleştirelim:

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

entity write_port_control is
    generic (DEPTH: natural := 4
            );
    port (
        wr_clk_i    : in std_logic;
        wr_rst_i    : in std_logic;
        wr_en_i     : in std_logic;
        rd_ptr_i    : in std_logic_vector(DEPTH downto 0);
        wr_ptr_o    : out std_logic_vector(DEPTH downto 0);
        wr_addr_o   : out std_logic_vector(DEPTH-1 downto 0);
        full_o      : out std_logic
    );
end write_port_control;
```

Bu kısımda blok şemasını verdiğimiz write_port modülünün giriş ve çıkışlarının tanımlamalarını gerçekleştiriyoruz.

Generic (**DEPTH**): FIFO'nun derinliğini (boyutunu) belirler. Varsayılan değer olarak 4 verilmiştir.

`wr_clk_i` : Yazma işlemi için saat sinyali, Yazma pointer'ı ve diğer ilgili sinyaller bu saat sinyaline göre çalışır.

`wr_rst_i` : Write domainine bağlı resetleme sinyali, aktif olduğunda yazma pointer sıfırlanır.

`wr_en_i` : Yazma etkinleştirme sinyali, aktif yani '1' olduğu zaman yazma işlemi gerçekleşecektir.

`rd_ptr_i` : Okuma pointer'ı, FIFO'nun yazma ve okuma tarafları arasındaki senkronizasyonu sağlar. FIFO doluluk kontrolünde kullanılır

`wr_ptr_o` : Yazma pointer'ı, Yazma pointer'ının gray kodlu mevcut değeri. FIFO'nun yazma tarafındaki mevcut konumunu gösterir.

`wr_addr_o` : FIFO adresi, FIFO belleğinde yazma işlemi için kullanılan adres (binary).

`full_o` : FIFO'nun dolu bayrağı, FIFO'nun dolu olduğunu gösteren sinyal. Eğer 1 ise, FIFO doludur ve yazma işlemi yapılmaz.

```
architecture gray_arch of write_port_control is
    signal w_ptr_reg_s    : std_logic_vector(DEPTH downto 0);
    signal w_ptr_next_s   : std_logic_vector(DEPTH downto 0);
    signal gray_s         : std_logic_vector(DEPTH downto 0);
    signal bin_s          : std_logic_vector(DEPTH downto 0);
    signal bin1_s         : std_logic_vector(DEPTH downto 0);
    signal w_addr_msb_s   : std_logic;
    signal r_addr_msb_s   : std_logic;
    signal full_flag_s    : std_logic;
```

Burada sinyal tanımlamaları yapılmıştır. Bu sinyaller sayesinde bu modül içerisinde yapmak isteyeceğimiz işlemleri gerçekleştirip içerideki atamaları gerçekleştirebiliyoruz.

`w_ptr_reg_s` : Yazma pointer'ı, FIFO yazma pointer'ının mevcut değerini saklar. Bir register oluşturmak için kullanılan sinyaldir.

`w_ptr_next_s` : Sonraki yazma pointer'ı, Bir sonraki Gray kodlu yazma pointer'ını ifade eder. Register kullanımında gereklidir.

`gray_s` : Binary pointer'ın Gray kodundaki karşılığı. FIFO yazma tarafında senkronizasyon için kullanılır.

`bin_s` : Yazma pointer'ının binary formu. Gray koda dönüştürülmeden önceki hali.

`bin1_s` : Binary pointer'ın bir artırılmış hali. Gray kod dönüşümü sırasında kullanılır.

`w_addr_msb_s` : Yazma pointer'ının en yüksek bitinin bir alt bit ile XOR'lanmış hali.

`r_addr_msb_s` : Okuma pointer'ının en yüksek bitinin bir alt bit ile XOR'lanmış hali. FIFO doluluk kontrolü için kullanılır.

`full_flag_s` : FIFO'nun dolu olup olmadığını kontrol eden bayrak.

```
begin
    process (wr_clk_i, wr_rst_i)
    begin
        if (wr_rst_i = '1') then
            w_ptr_reg_s <= (others => '0');
        elsif (rising_edge(wr_clk_i)) then
            w_ptr_reg_s <= w_ptr_next_s;
        end if;
    end process;
```

Burada process'e bağlı bir işlem gerçekleştirilmiştir.

- Reset sinyali aktif (**wr_rst_i** = '1') olduğunda yazma pointer sıfırlanır.
- Saat sinyalinin pozitif kenarında (rising_edge(**wr_clk_i**)), **w_ptr_next_s** değeri **w_ptr_reg_s**'e atanır. Bu şekilde de aslında bir adet register tasarımı yapılmış olunur.

```
    -- binary to gray
    bin_s   <= w_ptr_reg_s xor ('0' & bin_s
(DEPTH downto 1));
    bin1_s  <=
std_logic_vector(unsigned(bin_s) + 1);
    gray_s <= bin1_s xor ('0' & bin1_s(DEPTH
downto 1));
```

Burada binary kodun gray koda dönüşmesi işlemi vardır. Bu işlemi yapmamızın nedenini daha önce açıklamıştık. Bir kez daha hatırlatmakta fayda var. Gray kod, dijital sistemlerde özellikle senkronizasyon ve hata azaltma amacıyla kullanılır. Bizim de burada gelen yazma değerimizin binary değerini gray karşılığına çevirmemiz söz konusudur.

Binary Pointer (**bin_s**): **w_ptr_reg_s**'in mevcut değeri binary olarak hesaplanır.

Binary Pointer +1 (**bin1_s**): Binary pointer bir artırılır. Bu, Gray kod dönüşümünde kullanılır.
Gray Kod (**gray_s**): Binary pointer'ın Gray kodundaki karşılığı hesaplanır.

```
    -- Update write pointer
    w_ptr_next_s <= gray_s when (wr_en_i = '1'
and full_flag_s = '0') else
    w_ptr_reg_s;
```

Burada **w_ptr_next_s** sinyalinin güncellemesi yapılır. **wr_en_i** sinyali 1 değerinde olur ve **full_flag_s** sinyali 0 değerinde olursa **gray_s** değeri **w_ptr_next_s** sinyaline atanır, yoksa **w_ptr_reg_s** sinyali atanır.

```
    --DEPTH (full belirlenmesi icin)
    w_addr_msb_s <= w_ptr_reg_s(DEPTH) xor
w_ptr_reg_s(DEPTH-1);
```

Bu satır, FIFO'nun doluluk durumunu belirlemek için yapılan bir işlemdir.

w_ptr_reg_s(DEPTH): Yazma pointer'ının en yüksek (MSB - Most Significant Bit) biti.

w_ptr_reg_s(DEPTH-1): Yazma pointer'ının bir alt biti.

Sonuç: MSB ile bir alt bit arasındaki XOR işlemi sonucunda **w_addr_msb_s** sinyali elde edilir.

```
    -- Check for FIFO full_o
    r_addr_msb_s <= rd_ptr_i(DEPTH) xor
rd_ptr_i(DEPTH-1);
    full_flag_s  <= '1' when rd_ptr_i(DEPTH) /=
w_ptr_reg_s(DEPTH) and
                          rd_ptr_i(DEPTH-2
downto 0) = w_ptr_reg_s (DEPTH-2 downto 0) and
                          r_addr_msb_s =
w_addr_msb_s else
                  '0';
```

Burada en önemli işlemlerden birisi yapılır. FIFO'nun doluluk durumu kontrol edilir. Buradaki duruma göre FIFO'ya yazma işlemi devam edebilir veya durmak zorundadır.

full_flag_s, yazma ve okuma pointer'larının durumuna göre FIFO'nun dolu olup olmadığını belirler. Burada ilk olarak okuma ve yazma pointerları eşit değilse ve okuma sinyali ile yazma sinyallerinin 3.bitleri eşit ise ve okuma ile yazma adresleri eşit ise **full_flag_s** '1' yani dolu durumdadır. Bu şartlar sağlanmaz ise dolu değil yani **full_flag_s** '0' değeri atanır.

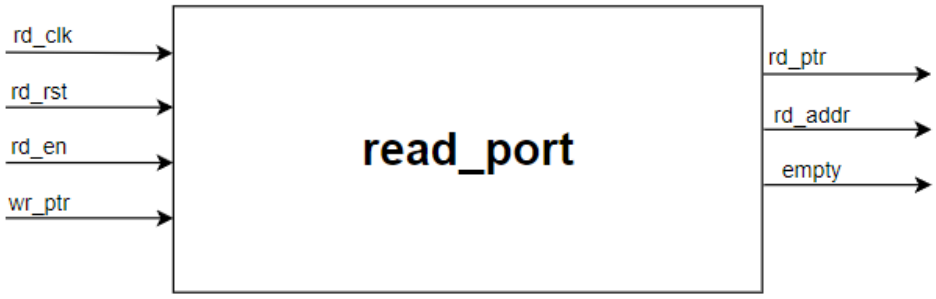
```
    -- Outputs
    wr_addr_o <= bin_s(DEPTH-1 downto 0);
    wr_ptr_o  <= w_ptr_reg_s;
    full_o    <= full_flag_s;

end gray_arch;
```

Burada ise son olarak çıkış değerlerimize atanması gereken sinyallerimiz atanır.

Böylece bu bloğun tasarımını gerçekleştirmiş olduk.

Read Port Control

Burada da önce aşağıdaki bloğumuzu inceleyerek başlayalım.



Bu blok da aslında write bloğuna çok benzer şekildedir. İçerideki yapılan işlemler çoğunlukla aynıdır.

Read Pointer Modülü, FIFO (First In, First Out) yapısının okuma kısmını yöneten ve okuma işlemleri sırasında okuma pointer'ını (read pointer) kontrol eden modüldür. FIFO yapısında okuma pointer'ı, verilerin sırasıyla FIFO'dan alınmasını sağlar. Bu modülün temel amacı, FIFO'dan doğru bir şekilde veri okuma işlemi yapmaktır.

Read pointer modülü, FIFO'dan veri okuma işlemini kontrol eden ve okuma pointer'ını yöneten önemli bir bileşendir. Bu modül:

- FIFO'dan doğru veri okuma işlemi yapar.
- Okuma pointer'ının doğru yönetilmesiyle veri sırasını korur.
- FIFO'nun boş olup olmadığını kontrol eder.
- FIFO'nun okuma işlemi için gereken sinyalleri sağlar.
- Read pointer modülü, FIFO'nun verimli ve doğru çalışması için kritik bir rol oynar ve veri akışının düzenli bir şekilde yönetilmesini sağlar.

Detaylı analize geçmeden önce write port control modülünün tüm kodlarını aşağıda bulabilirsiniz.

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity read_port_control is
    generic (DEPTH : natural := 4
            );
    port (
        rd_clk_i    : in std_logic;
        rd_rst_i    : in std_logic;
        wr_ptr_i    : in std_logic_vector(DEPTH downto 0);
        rd_en_i     : in std_logic;
        rd_ptr_o    : out std_logic_vector(DEPTH downto 0);
        rd_addr_o   : out std_logic_vector(DEPTH-1 downto 0);
        empty_o     : out std_logic
    );
end read_port_control;

architecture gray_arch of read_port_control is
    signal r_ptr_reg_s      : std_logic_vector(DEPTH downto 0);
    signal r_ptr_next_s     : std_logic_vector(DEPTH downto 0);
    signal gray_s           : std_logic_vector(DEPTH downto 0);
    signal bin_s            : std_logic_vector(DEPTH downto 0);
    signal bin1 s           : std logic vector(DEPTH downto 0);
    signal empty_flag_s     : std_logic;
    signal raddr_msb_s      : std_logic;
    signal waddr_msb_s      : std_logic;
begin

    process (rd clk i, rd rst i)
    begin
        if rd_rst_i = '1' then
            r_ptr_reg_s <= (others => '0');
        elsif rising_edge(rd_clk_i) then
            r ptr reg s <= r ptr next s;
        end if;
    end process;

    -- binary to gray
    bin_s   <= r_ptr_reg_s xor ('0' & bin_s(DEPTH downto 1));
    bin1_s  <= std_logic_vector(unsigned(bin_s) + 1);
    gray_s <= bin1_s xor ('0' & bin1_s(DEPTH downto 1));


    -- Update read pointer
    r_ptr_next_s <= gray_s when rd_en_i = '1' and empty_flag_s = '0' else
                    r_ptr_reg_s;


    -- DEPTH-bit Gray counter(empty sinyali icin)
    raddr_msb_s  <= r_ptr_reg_s(DEPTH) xor r_ptr_reg_s(DEPTH-1);
    waddr_msb_s  <= wr_ptr_i(DEPTH) xor wr_ptr_i(DEPTH-1);


    -- Check for FIFO empty_o
    empty_flag_s <= '1' when wr_ptr_i(DEPTH) = r_ptr_reg_s(DEPTH) and
                             wr_ptr_i(DEPTH-2 downto 0) = r_ptr_reg_s(DEPTH-2 downto 0) and
                             raddr_msb_s = waddr_msb_s else
                    '0';


    -- Outputs
    rd_addr_o   <= bin_s(DEPTH-1 downto 0);
    rd_ptr_o    <= r_ptr_reg_s;
    empty_o     <= empty_flag_s;

end gray_arch;
```

Detaylı analizini gerçekleştirelim:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity read_port_control is
    generic (DEPTH : natural := 4
            );
    port (
        rd_clk_i    : in std_logic;
        rd_rst_i    : in std_logic;
        wr_ptr_i    : in std_logic_vector(DEPTH downto 0);
        rd_en_i     : in std_logic;
        rd_ptr_o    : out std_logic_vector(DEPTH downto 0);
        rd_addr_o   : out std_logic_vector(DEPTH-1 downto 0);
        empty_o     : out std_logic
    );
```

Burada kullanılacak kütüphanelerin ve modülün giriş çıkışlarının tanımlamaları yapılır.

`rd_clk_i` : Okuma işlemleri için kullanılan saat sinyali.

`rd_rst_i` : Asenkron reset sinyali. Aktif olduğunda tüm okuma kontrol sinyallerini sıfırlar.

`wr_ptr_i` : Yazma işaretçisinin Gray kodlu adresi. FIFO'nun doluluk durumunu kontrol etmek için kullanılır.

`rd_en_i` : Okuma işleminin aktif hale getirilmesi için kullanılan sinyal.

`rd_ptr_o`: Okuma işaretçisinin Gray kodlu değeri.

`rd_addr_o`: FIFO belleğinde okuma yapılacak adres (binary formatta).

`empty_o`: FIFO'nun boş olduğunu gösteren sinyal.

```
architecture gray_arch of read_port_control is
    signal r_ptr_reg_s      : std_logic_vector(DEPTH downto 0);
    signal r_ptr_next_s     : std_logic_vector(DEPTH downto 0);
    signal gray_s           : std_logic_vector(DEPTH downto 0);
    signal bin_s            : std_logic_vector(DEPTH downto 0);
    signal bin1_s           : std_logic_vector(DEPTH downto 0);
    signal empty_flag_s     : std_logic;
    signal raddr_msb_s      : std_logic;
    signal waddr_msb_s      : std_logic;
```

Burada modül içerisinde kullanacağımız sinyallerin tanımlamalarını gerçekleştirdik.

`r_ptr_reg_s`: Okuma işaretçisinin mevcut Gray kodlu değeri.

`r_ptr_next_s`: Okuma işaretçisinin bir sonraki değeri.

`gray_s`: Gray koduna çevrilen değer.

`bin_s`: Okuma işaretçisinin binary formatındaki değeri.

`bin1_s`: bin_s'nin bir artırılmış hali.

`empty_flag_s`: FIFO'nun boş olup olmadığını kontrol eden sinyal.

`raddr_msb_s` ve `waddr_msb_s`: Okuma ve yazma işaretçilerinin en üst iki biti XOR ile elde edilen değerler. FIFO'nun boş veya dolu durumunu kontrol etmekte kullanılır.

```
    process (rd_clk_i, rd_rst_i)
    begin
        if rd_rst_i = '1' then
            r_ptr_reg_s <= (others => '0');
        elsif rising_edge(rd_clk_i) then
            r_ptr_reg_s <= r_ptr_next_s;
        end if;
    end process;
```

Burada bir register oluşturulur.

Amaç okuma işaretçisinin (Gray kod) değerini güncellemektir.

rd_rst_i aktif olduğunda işaretçi sıfırlanır.

rising_edge(**rd_clk_i**) Saat sinyalinin yükselen kenarında çalışır. İşaretçinin bir sonraki değeri (**r_ptr_next_s**), mevcut işaretçi (**r_ptr_reg_s**) olarak atanır.

```
    -- binary to gray
    bin_s   <= r_ptr_reg_s xor ('0' &
bin s(DEPTH downto 1));
    bin1_s  <=
std_logic_vector(unsigned(bin_s) + 1);
    gray_s <= bin1_s xor ('0' &
bin1_s(DEPTH downto 1));
```

Burada binary kodun gray koda dönüşmesi işlemi vardır.

Binary Pointer (**bin_s**): **w_ptr_reg_s**'in mevcut değeri binary olarak hesaplanır.

Binary Pointer +1 (**bin1_s**): Binary pointer bir artırılır. Bu, Gray kod dönüşümünde kullanılır. Gray Kod (**gray_s**): Binary pointer'ın Gray kodundaki karşılığı hesaplanır.

```
    -- Update read pointer
    r_ptr_next_s <= gray_s when rd_en_i =
'1' and empty_flag_s = '0' else
                    r_ptr_reg_s;
```

rd_en_i = '1': Okuma işlemi etkin.

empty_flag_s = '0': FIFO boş değil.

Şartlar sağlanıyorsa, bir sonraki okuma işaretçisi Gray koduna çevrilen adres (**gray_s**) olur. Aksi halde mevcut işaretçi korunur.

```
    -- DEPTH-bit Gray counter(empty sinyali
icin)
    raddr_msb_s  <= r_ptr_reg_s(DEPTH) xor
r_ptr_reg_s(DEPTH-1);
    waddr_msb_s  <= wr_ptr_i(DEPTH) xor
wr_ptr_i(DEPTH-1);
```

Burada FIFO'nun boşluk durumunu kontrol etmek için adreslerin en üst bitlerini (MSB) kullanarak bir mantıksal karşılaştırma yapmaktır. Gray kod kullanılan işaretçilerde, MSB'nin iki bitlik XOR işlemi ile FIFO'nun boş olup olmadığını tespit etmek daha güvenilir bir yöntemdir.

```
    -- Check for FIFO empty_o
    empty_flag_s <= '1' when
wr_ptr_i(DEPTH) = r_ptr_reg_s(DEPTH) and
                          wr_ptr_i(DEPTH-2
downto 0) = r_ptr_reg_s(DEPTH-2 downto 0)
and
                          raddr_msb_s =
waddr_msb_s else
                  '0';
```

Burada FIFO'nun boş olma durumu gerçekleştirilir. Yazma işaretçisinin ve okuma işaretçisinin en üst bitleri aynıysa (**wr_ptr_i**(DEPTH) = **r_ptr_reg_s**(DEPTH))

Alt bitler eşitse (**wr_ptr_i**(DEPTH-2 downto 0) = **r_ptr_reg_s**(DEPTH-2 downto 0)).

MSB XOR sonuçları eşitse (**raddr_msb_s** = **waddr_msb_s**).

Tüm koşullar sağlanıyorsa FIFO boştur (**empty_flag_s** = '1').

```
    -- Outputs
    rd_addr_o   <= bin_s(DEPTH-1 downto 0);
    rd_ptr_o    <= r_ptr_reg_s;
    empty_o     <= empty_flag_s;

end gray_arch;
```

Burada modül içerisindeki sinyallerin çıkışlara ataması gerçekleştirilir.

Böylece ana işlemlerin yapıldığı yazma ve okuma pointerlarının tasarımını gerçekleştirmiş olduk. Şimdi sırada diğer modüllerin tasarımı var.

Syncronizer

Senkronizerlere ihtiyaç duyulmasının temel nedeni asenkron sinyallerin senkron bir tasarıma güvenli ve kararlı bir şekilde entegre edilmesini sağlamaktır. Bu özellikle farklı saat alanlarıyla çalışan sistemlerde veri iletimi sırasında ortaya çıkan zamanlama belirsizliği (timing uncertainty) ve metastabilite gibi problemleri çözmek için gereklidir.

Asenkron fifo tasarımında en önemli işlem budur diyebiliriz. Senkronizerler sayesinden farklı saat clockları ile çalışmak daha kolay hale gelmektedir. Temel olarak ne için kullandığımızı açıklayacak olursak şu şekilde söyleyebiliriz:

1-)Metastabiliteyi Önlemek:

- Metastabilite, bir flip-flop girişindeki sinyalin saat kenarına yakın bir zamanda değişmesi nedeniyle kararsız bir duruma geçmesi durumudur. Bu durumda flip-flop, kararsız bir süre boyunca karışık bir voltaj seviyesinde kalır ve ardından öngörülemez bir durum üretir.
- Senkronizer, bu tür metastabilite durumlarının diğer devre elemanlarına yayılmasını önler.

2-)Asenkron Sinyallerin Güvenli Alınması:

- Bir sistemde farklı saat alanları (clock domains) varsa, bir saat alanından diğerine veri transfer ederken zamanlama belirsizlikleri oluşabilir.
- Senkronizer, bu belirsizliği gidererek sinyalin güvenli bir şekilde diğer saat alanında kullanılmasını sağlar.

3-)Sistem Güvenilirliğini Artırma:

- Saat alanları arasındaki uyuşmazlıklar veya asenkron bir sinyalin doğrudan senkron bir sistemde kullanılması, hata olasılığını artırır.
- Senkronizer, bu tür hataları azaltarak sistemin kararlılığını artırır.

4-)Saat Alanları Arası Veri Transferini Kolaylaştırma:

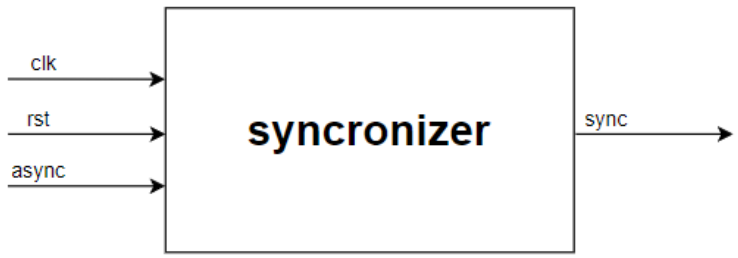
- Farklı saat alanlarına ait sinyaller arasında zamanlama uyuşmazlıklarını çözmek için senkronizerler kullanılır.

Senkronizer devresi genellikle ardışık bağlı iki veya daha fazla flip-flop'tan oluşur:

1. **Birinci Flip-Flop:**
    - Asenkron sinyal bu flip-flop'a girer.
    - Bu flip-flop metastabil olma ihtimali taşır; ancak metastabil durumun devre dışına yayılmasını önlemek için izole edilir.
2. **İkinci Flip-Flop:**
    - Birinci flip-flop'un çıkışını kullanır.
    - İlk flip-flop metastabil durumda olsa bile metastabilitenin bir sonraki flip-flop'a geçme olasılığı çok düşüktür.
    - Böylece ikinci flip-flop güvenilir bir sinyal üretir.

Not: Gerekli görülürse üçüncü bir flip-flop daha eklenerek metastabilite riski daha da azaltılabilir. Üçüncü flip-flop'u eklemek elbette çok daha kesin ve sorunsuz bir tasarım elde edilmesini sağlayacaktır fakat bu sefer de kaynak tüketimi konusunda sıkıntılar yaşanabilir. Bu yüzden tasarım yapılırken bütün parametreler gözden geçirilip ona göre işlem yapılmalıdır.

Biz de asenkron bir FIFO tasarımı gerçekleştirdiğimiz için senkronizere ihtiyaç duyuyoruz. Temelde çok basit tasarımı vardır. Bunun da yine blok tasarımını aşağıda görebilirsiniz.



Asenkron olarak giren bir durumu senkron şekilde dışarıya aktarmaya yarayan araçtır da diyebiliriz kısaca. Şimdi tasarıma başlayıp kodlamasına geçebiliriz.

```vhdl
library ieee;
use ieee.std_logic_1164.all;

entity synchronizer is
     generic (DEPTH : natural := 4
             );
    port (
        clk_i    : in std_logic;
        rst_i    : in std_logic;
        async_i  : in std_logic_vector(DEPTH downto 0);
        sync_o   : out std_logic_vector(DEPTH downto 0)
    );
end synchronizer;

architecture two_ff_arch of synchronizer is
    signal meta_reg_s   : std_logic_vector(DEPTH downto 0);
    signal meta_next_s  : std_logic_vector(DEPTH downto 0);
    signal sync_reg_s   : std_logic_vector(DEPTH downto 0);
    signal sync_next_s  : std_logic_vector(DEPTH downto 0);

begin
    process (clk_i, rst_i)
        begin
            if (rst_i = '1') then
                meta_reg_s <= (others => '0');
                sync_reg_s <= (others => '0');
            elsif rising_edge(clk_i) then
                meta_reg_s <= meta_next_s;
                sync_reg_s <= sync_next_s;
            end if;
    end process;

    -- next-state logic
    meta_next_s <= async_i;
    sync_next_s <= meta_reg_s;


    -- output
    sync_o <= sync_reg_s;



end two_ff_arch;
```

Detaylı analize başlayabiliriz:

```
library ieee;
use ieee.std_logic_1164.all;

entity synchronizer is
     generic (DEPTH : natural := 4
              );
    port (
        clk_i    : in std_logic;
        rst_i    : in std_logic;
        async_i  : in std_logic_vector(DEPTH downto 0);
        sync_o   : out std_logic_vector(DEPTH downto 0)
    );
end synchronizer;
```

Burada blok şemaya uygun şekilde giriş ve çıkış portlarımızın tanımlamasını gerçekleştiriyoruz.

`clk_i`: Sisteme gelecek olan sinyaldir.

`rst_i`: Sistemi resetlemek için bulunan sinyaldir.

`aysnc_i`: Senkron edilmeden önceki hali olan değerdir.

`sync_o`: Senkron hale gelmiş çıkış değeridir.

```
architecture two_ff_arch of synchronizer is
    signal meta_reg_s   : std_logic_vector(DEPTH downto 0);
    signal meta_next_s  : std_logic_vector(DEPTH downto 0);
    signal sync_reg_s   : std_logic_vector(DEPTH downto 0);
    signal sync_next_s  : std_logic_vector(DEPTH downto 0);
```

Burada modül içerisindeki işlemleri gerçekleştirecek sinyallerin tanımlamalarını gerçekleştiriyoruz.

`meta_reg_s` : Birinci flip-flop zincirinin kayıt ettiği geçici sinyaldir. Metastabilite riskini izole eder.

`meta_next_s` : Asenkron giriş sinyali (**async_i**) doğrudan birinci flip-flop'a aktarılır.

`sync_reg_s` : İkinci flip-flop zincirinin kayıt ettiği kararlı sinyaldir. Çıkış sinyali (**sync_o**) bu sinyale bağlıdır.

`sync_next_s` : Birinci flip-flop'un çıkışı ikinci flip-flop'a aktarılır.

```
begin
    process (clk_i, rst_i)
        begin
            if (rst_i = '1') then
                meta_reg_s <= (others => '0');
                sync_reg_s <= (others => '0');
            elsif rising_edge(clk_i) then
                meta_reg_s <= meta_next_s;
                sync_reg_s <= sync_next_s;
            end if;
    end process;
```

Burada process'e göre atamalar gerçekleştirilir.

rst_i = '1' olduğunda, hem birinci (**meta_reg_s**) hem de ikinci (**sync_reg_s**) flip-flop sıfırlanır.

Çıkış sinyalleri sıfırlanmış olur.

Her pozitif saat kenarında;

meta_reg_s: **meta_next_s** değerini alır (yani **async_i** sinyali birinci flip-flop'ta tutulur).

sync_reg_s: **sync_next_s** değerini alır (yani ikinci flip-flop birinci flip-flop'un çıkışını kayıt eder).

```
    -- next-state logic
    meta_next_s <= async_i;
    sync_next_s <= meta_reg_s;


    -- output
    sync_o <= sync_reg_s;


end two_ff_arch;
```

meta_next_s: Asenkron giriş sinyali (**async_i**) birinci flip-flop'a aktarılır.

sync_next_s: Birinci flip-flop'un çıkışı ikinci flip-flop'a aktarılır.

İkinci flip-flop'tan alınan kararlı sinyal çıkış olarak (**sync_o**) atanır.

Kısaca özetleyecek olursak:

1- **Reset Durumu:**

- Sisteme reset uygulandığında (**rst_i** = '1'), tüm flip-floplar sıfırlanır ve çıkışlar (**sync_o**) 0 olur.

2- **Metastabilite İzolasyonu:**

- async_i sinyali birinci flip-flop'a (**meta_reg_s**) aktarılır. Eğer metastabilite oluşursa, bu durum birinci flip-flop ile sınırlandırılır.

3- **Kararlı Çıkış Üretimi:**

- Birinci flip-flop'un çıkışı (**meta_reg_s**), ikinci flip-flop'a (**sync_reg_s**) aktarılır. Bu işlem metastabilite olasılığını büyük ölçüde azaltır.
- İkinci flip-flop'un çıkışı, güvenilir bir şekilde **sync_o** olarak dışarıya verilir.

Dual Port RAM

Şimdi burada da bizim gelen verilerimizi bir yere kayıt etmemiz, yazmamız gerekiyor, Buraya kadar yaptıklarımız hep yazma ve okuma işlemlerini gerçekleştirmek için gereken adımlardı. Şimdi ise gelen veriyi artık bir yere kaydedip sonrasında kaydettiğimiz yerden onu okuma işlemini gerçekleştireceğiz, daha doğrusu çıkışa vereceğiz. Bunu da bir adet dual port RAM ile gerçekleştireceğiz. Öncelikle dual port RAM nedir ve ne işe yarar gibi soruları kısaca cevaplandırdıktan sonra tasarımımıza başlayabiliriz.

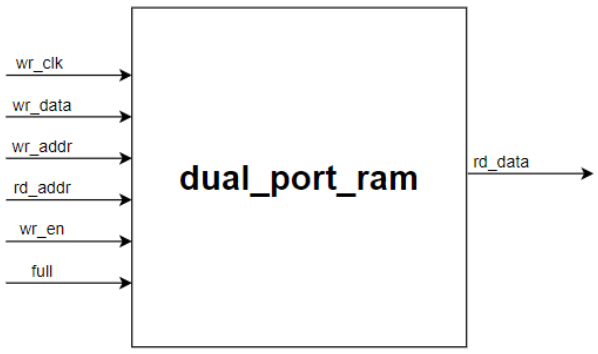
Dual-Port RAM, aynı anda iki farklı port üzerinden erişim sağlayabilen bir tür rastgele erişim belleğidir (**Random Access Memory - RAM**). Temel olarak, **iki ayrı giriş ve çıkış portu** bulunur, böylece iki farklı işlem aynı anda gerçekleşebilir.

Özellikleri

1. **Çift Port:**
    - İki bağımsız **adres**, **veri** ve **kontrol sinyalleri** ile çalışır.
    - Bir port yazma işlemi yaparken diğer port okuma işlemi gerçekleştirebilir.
2. **Eş Zamanlı Erişim:**
    - İki port aynı veya farklı adreslere aynı anda erişebilir.
    - Bu, paralel işlemleri hızlandırır ve performansı artırır.
3. **Çatışma Yönetimi:**
    - Eğer iki port aynı adresi aynı anda yazmaya çalışırsa, bu durumda bir **çakışma (conflict)** oluşabilir. Bu durum genellikle özel bir çözümle yönetilir (örneğin, yazma önceliği).

Dual port RAM'lar aynı anda iki işlem yapılabilir. Farklı işlemciler arasında veri paylaşımı veya bir yazma portunun başka bir okuma portuyla eş zamanlı kullanımını gerçekleştirebilirler ve bizim işimize yarayan kısmı da Farklı portlar farklı saat sinyalleriyle çalışabilir (asenkron dual port RAM). Biz de farklı saat sinyalleri ile çalışacağımız için burada dual port RAM tercihince bulunduk.

Tasarım için önce aşağıdaki temel blok şemayı inceleyebilirsiniz. Tasarımımız buna göre gerçekleştirilecektir. Şemadan sonra kodun tamamını aşağıda görebilirsiniz.




```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity d_p_ram is

    generic (   data_WIDTH  : natural ;
                DEPTH       : natural
            );
     port (
        wr_clk_i    : in std_logic;
        wr_data_i   : in std_logic_vector(data_WIDTH-1 downto 0);
        wr_addr_i   : in std_logic_vector(DEPTH-1 downto 0);
        rd_addr_i   : in std_logic_vector(DEPTH-1 downto 0);
        wr_en_i     : in std_logic;
        rd_data_o   : out std_logic_vector(data_WIDTH-1 downto 0);
        full_i      : in std_logic
    );

end d_p_ram;


architecture str_arch of d_p_ram is

type d_p_ram is array (0 to 2**DEPTH-1) of std_logic_vector(data_WIDTH-1 downto 0);
signal ram: d_p_ram;
signal notfull_s: std_logic;

begin
    process(wr_clk_i)
    begin
        if (wr_clk_i'event and wr_clk_i = '1') then
            if (notfull_s = '1') then
                ram(to_integer(unsigned(wr_addr_i))) <= wr_data_i;
            end if;
        end if;
        rd_data_o <= ram(to_integer(unsigned(rd_addr_i)));
    end process;
    notfull_s <= wr_en_i and not full_i;

end str_arch;
```

Detaylı analize geçelim:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity d_p_ram is

    generic (   data_WIDTH  : natural ;
                DEPTH       : natural
            );
     port (
        wr_clk_i    : in std_logic;
        wr_data_i   : in std_logic_vector(data_WIDTH-1 downto 0);
        wr_addr_i   : in std_logic_vector(DEPTH-1 downto 0);
        rd_addr_i   : in std_logic_vector(DEPTH-1 downto 0);
        wr_en_i     : in std_logic;
        rd_data_o   : out std_logic_vector(data_WIDTH-1 downto 0);
        full_i      : in std_logic
    );

end d_p_ram;
```

Burada öncelikle kullanılacak olan kütüphanelerin tanımı ve ardından da RAM'in giriş ve çıkış portlarının tanımlanması yapılmıştır. Generic olarak RAM'in veri derinliği ve veri genişliği tanımlaması da gerçekleştirilmiştir.

wr_clk_i: Yazma işlemi için kullanılan saat sinyali.

wr_data_i: Yazılacak veri giriş portu.

wr_addr_i: Yazma işlemi için adres giriş portu.

rd_addr_i: Okuma işlemi için adres giriş portu.

wr_en_i: Yazma izni sağlayan kontrol sinyali.

rd_data_o: Okuma işlemi için veri çıkış portu.

full_i: RAM'in dolu olduğunu belirten giriş sinyali.

```
architecture str_arch of d_p_ram is

type d_p_ram is array (0 to 2**DEPTH-1) of
std_logic_vector(data_WIDTH-1 downto 0);
signal ram: d_p_ram;
signal notfull_s: std_logic;
```

Burada davranışsal modelleme gerçekleştirilmiştir. Öncelikle RAM'in tipinin(array) tanımı gerçekleştirilmiştir. RAM'imizin hücre sayısı 2^DEPTH olarak ayarlanmıştır. Yani generic olarak verilen değere göre şekil alacaktır. Ayrıca her bir hücrenin içerisine yazılabilecek veri biti değeri de yine generic olarak tanımlanmıştır ve o da **data_WIDTH** olarak yani 16 bit olarak hücreler içerisinde saklanacaktır.

notfull_s sinyali yazma izni için kullanılacaktır.

d_p_ram: RAM'in bellek alanını temsil eden bir dizi.

```
begin
    process(wr_clk_i)
    begin
        if (wr_clk_i'event and wr_clk_i = '1') then
            if (notfull_s = '1') then
                ram(to_integer(unsigned(wr_addr_i)))
<= wr_data_i;
            end if;
        end if;
        rd_data_o <=
ram(to_integer(unsigned(rd_addr_i)));
    end process;
```

Burada saate bağlı bir process tanımlanmıştır.

İşlem bloğu, sadece yükselen kenarda (rising edge) tetiklenir (**wr_clk_i**'event **and wr_clk_i** = '1').

RAM dolu değilse (**notfull_s** = '1'), yazma işlemi yapılır.

Yazma işlemi için adres **wr_addr_i** ile gösterilir.

Adres, std_logic_vector tipinden integer tipe dönüştürülür ve **wr_data_i** girişindeki veri, belirtilen adrese yazılır.

Okuma işlemi doğrudan RAM dizisinden yapılır.

Okunacak veri adres **rd_addr_i** ile belirlenir.

Adres, integer tipe dönüştürülür ve bu adresin RAM'deki içeriği **rd_data_o** çıkışına atanır.

```
    notfull_s <= wr_en_i and not full_i;

end str_arch;
```

wr_en_i (yazma izni) ve **full_i** (RAM doluluk durumu) sinyallerine dayanarak yazma kontrolü yapılır:

Eğer **wr_en_i** = '1' ve **full_i** = '0' ise **notfull_s** aktif olur ve yazma işlemi gerçekleştirilir.

Kodun çalışma prensibini şu şekilde de açıklayabiliriz:

1. **Yazma Süreci:**

- Eğer **wr_en_i** aktifse ve RAM dolu değilse (**notfull_s = '1'**), yazma işlemi yapılır.
- Yazılacak veri, **wr_data_i** üzerinden alınır ve **wr_addr_i** adresinde saklanır.

2. **Okuma Süreci:**

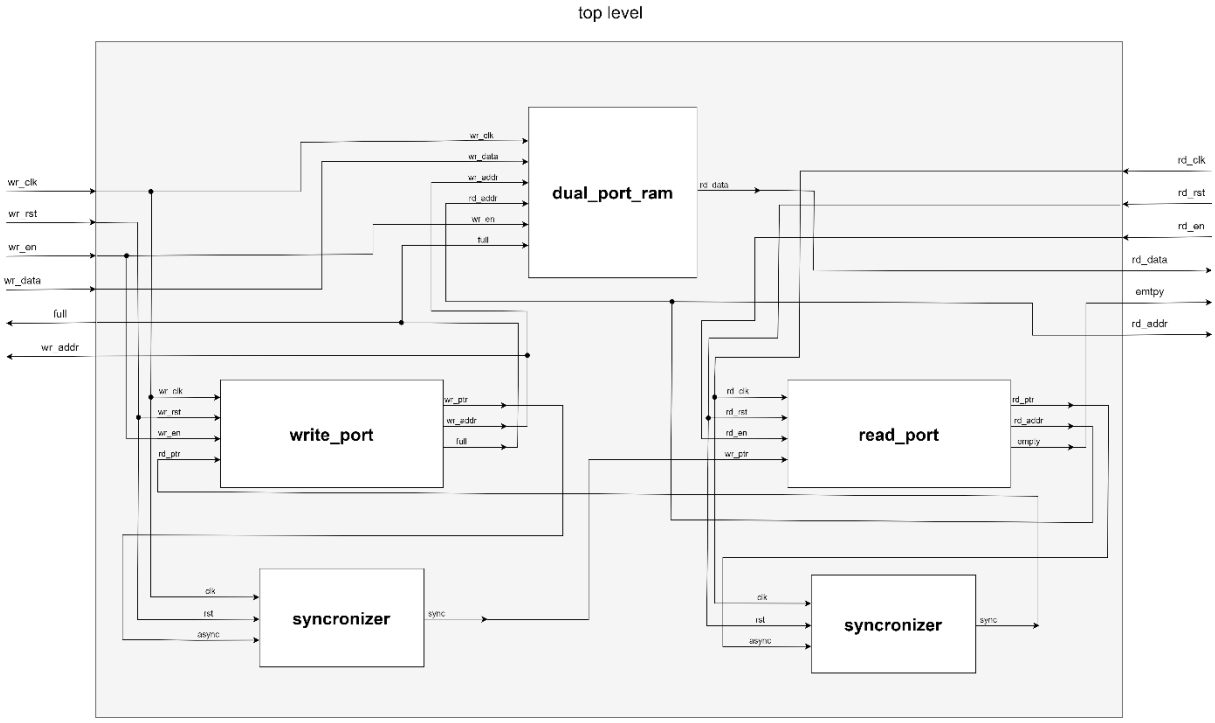
- Belirli bir adresten (**rd_addr_i**) RAM içeriği okunur ve **rd_data_o** çıkışına atanır.
- Okuma işlemi, yazma sürecinden bağımsız çalışır.

3. **Senaryo:**

- Aynı anda bir port veri yazarken, diğer port RAM'in başka bir adresinden veri okuyabilir.

Top Level

Şimdi sırada top level kısmımız var. Buradaki tasarımı da aşağıdaki blok şemadan detaylı olarak görebilirsiniz. Bu kısımda herhangi bir işlem yoktur sadece hangi bloğun nereye bağlandığının detayları bulunmaktadır.




```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity top_async_fifo is
    generic (   data_WIDTH  : natural := 16;
                DEPTH       : natural := 4
            );
    port (
        --write
        wr_clk_i    : in std_logic;
        wr_rst_i    : in std_logic;
        wr_en_i     : in std_logic;
        full_o      : out std_logic;
        wr_data_i   : in std_logic_vector(data_WIDTH-1 downto 0);

        --read
        rd_clk_i    : in std_logic;
        rd_rst_i    : in std_logic;
        rd_en_i     : in std_logic;
        rd_data_o   : out std_logic_vector(data_WIDTH-1 downto 0);
        empty_o     : out std_logic;

        wr_addr_o   : out std_logic_vector(DEPTH-1 downto 0);
        rd_addr_o   : out std_logic_vector(DEPTH-1 downto 0)
    );
end top_async_fifo;

architecture str_arch of top_async_fifo is
    signal r_ptr_in_s    : std_logic_vector(DEPTH downto 0);
    signal r_ptr_out_s   : std_logic_vector(DEPTH downto 0);
    signal w_ptr_in_s    : std_logic_vector(DEPTH downto 0);
    signal w_ptr_out_s   : std_logic_vector(DEPTH downto 0);
    signal r_addr_i_s    : std_logic_vector(DEPTH-1 downto 0);
    signal w_addr_i_s    : std_logic_vector(DEPTH-1 downto 0);
    signal full_o_s      : std_logic;
```

```
    component d_p_ram
        generic (data_WIDTH : natural := 16;
                 DEPTH : natural := 4
                );
        port (
        wr_clk_i    : in std_logic;
        wr_data_i   : in std_logic_vector(data_WIDTH-1 downto 0);
        wr_addr_i   : in std_logic_vector(DEPTH-1 downto 0);
        rd_addr_i   : in std_logic_vector(DEPTH-1 downto 0);
        wr_en_i     : in std_logic;
        rd_data_o   : out std_logic_vector(data_WIDTH-1 downto 0);
        full_i      : in std_logic
            );
    end component;

    component read_port_control
        generic (DEPTH : natural := 4);
        port (
            rd_clk_i    : in std_logic;
            rd_en_i     : in std_logic;
            rd_rst_i    : in std_logic;
            wr_ptr_i    : in std_logic_vector(DEPTH downto 0);
            empty_o     : out std_logic;
            rd_addr_o   : out std_logic_vector(DEPTH-1 downto 0);
            rd_ptr_o    : out std_logic_vector(DEPTH downto 0)
        );
    end component;

    component write_port_control
        generic (DEPTH : natural := 4);
        port (
            wr_clk_i    : in std_logic;
            rd_ptr_i    : in std_logic_vector(DEPTH downto 0);
            wr_rst_i    : in std_logic;
            wr_en_i     : in std_logic;
            full_o      : out std_logic;
            wr_addr_o   : out std_logic_vector(DEPTH-1 downto 0);
            wr_ptr_o    : out std_logic_vector(DEPTH downto 0)
        );
    end component;

    component synchronizer
        generic (DEPTH : natural := 4);
        port (
            clk_i   : in std_logic;
            async_i : in std_logic_vector(DEPTH downto 0);
            rst_i   : in std_logic;
            sync_o  : out std_logic_vector(DEPTH downto 0)
        );
    end component;


begin

    --assignment
    rd_addr_o   <= r_addr_i_s;
    full_o      <= full_o_s;
    wr_addr_o   <= w_addr_i_s;


    fifo_mem: d_p_ram
        generic map (
                    DEPTH => 4,
                    data_WIDTH => 16
                    )

        port map (
                  wr_clk_i   => wr_clk_i,
                  wr_data_i  => wr_data_i,
                  wr_addr_i  => w_addr_i_s,
                  rd_addr_i  => r_addr_i_s,
                  wr_en_i    => wr_en_i,
                  rd_data_o  => rd_data_o,
                  full_i     => full_o_s
                 );

    read_ctrl: read_port_control
        generic map (DEPTH => DEPTH)
        port map (
            rd_clk_i  => rd_clk_i,
            rd_rst_i  => rd_rst_i,
            rd_en_i   => rd_en_i,
            wr_ptr_i  => w_ptr_in_s,
            empty_o   => empty_o,
            rd_ptr_o  => r_ptr_out_s,
            rd_addr_o => r_addr_i_s
        );

    write_ctrl: write_port_control
        generic map (DEPTH => DEPTH)
        port map (
            wr_clk_i  => wr_clk_i,
            wr_rst_i  => wr_rst_i,
            wr_en_i   => wr_en_i,
            rd_ptr_i  => r_ptr_in_s,
            full_o    => full_o_s,
            wr_ptr_o  => w_ptr_out_s,
            wr_addr_o => w_addr_i_s
        );

    sync_w_ptr: synchronizer
        generic map (DEPTH => DEPTH )
        port map (
            clk_i   => wr_clk_i,
            rst_i   => wr_rst_i,
            async_i => w_ptr_out_s,
            sync_o  => w_ptr_in_s
        );

    sync_r_ptr: synchronizer
        generic map (DEPTH => DEPTH )
        port map (
            clk_i   => rd_clk_i,
            rst_i   => rd_rst_i,
            async_i => r_ptr_out_s,
            sync_o  => r_ptr_in_s
        );

end str_arch;
```

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity top_async_fifo is
    generic (   data_WIDTH  : natural := 16;
                DEPTH       : natural := 4
            );
    port (
        --write
        wr_clk_i    : in std_logic;
        wr_rst_i    : in std_logic;
        wr_en_i     : in std_logic;
        full_o      : out std_logic;
        wr_data_i   : in
std_logic_vector(data_WIDTH-1 downto 0);

        --read
        rd_clk_i    : in std_logic;
        rd_rst_i    : in std_logic;
        rd_en_i     : in std_logic;
        rd_data_o   : out
std_logic_vector(data_WIDTH-1 downto 0);
        empty_o     : out std_logic;

        wr_addr_o   : out
std_logic_vector(DEPTH-1 downto 0);
        rd_addr_o   : out
std_logic_vector(DEPTH-1 downto 0)
    );
```

data_WIDTH: FIFO'nun veri genişliğini tanımlar (örneğin, 16 bit).

DEPTH: FIFO'nun derinliğini tanımlar (örneğin, 2^4 = 16 adresli FIFO).

Yazma (Write) Portları:

`wr_clk_i`: Yazma işlemleri için saat sinyali.

`wr_rst_i`: Yazma tarafı sıfırlama sinyali.

`wr_en_i`: Yazma etkinleştirme sinyali.

`full_o`: FIFO'nun dolu olup olmadığını gösterir.

`wr_data_i`: Yazma sırasında FIFO'ya gönderilen veri.

Okuma (Read) Portları:

`rd_clk_i`: Okuma işlemleri için saat sinyali.

`rd_rst_i`: Okuma tarafı sıfırlama sinyali.

`rd_en_i`: Okuma etkinleştirme sinyali.

`rd_data_o`: FIFO'dan okunan veri.

`empty_o`: FIFO'nun boş olup olmadığını gösterir.

Adres Portları:

`wr_addr_o`, `rd_addr_o`: Yazma ve okuma işlemleri için kullanılan adresler.

```
architecture str_arch of top_async_fifo is
    signal r_ptr_in_s    : std_logic_vector(DEPTH
downto 0);
    signal r_ptr_out_s   : std_logic_vector(DEPTH
downto 0);
    signal w_ptr_in_s    : std_logic_vector(DEPTH
downto 0);
    signal w_ptr_out_s   : std_logic_vector(DEPTH
downto 0);
    signal r_addr_i_s    : std_logic_vector(DEPTH-1
downto 0);
    signal w_addr_i_s    : std_logic_vector(DEPTH-1
downto 0);
    signal full_o_s      : std_logic;
```

`r_ptr_in_s`, `r_ptr_out_s`: Okuma pointer'ı (adresleyici) giriş ve çıkış sinyalleri.

`w_ptr_in_s`, `w_ptr_out_s`: Yazma pointer'ı giriş ve çıkış sinyalleri.

`r_addr_i_s, w_addr_i_s`: Okuma ve yazma işlemleri için adres sinyalleri.

`full_o_s`: FIFO'nun dolu olduğunu gösteren dahili sinyal.

```
component d_p_ram
        generic (data_WIDTH : natural := 16;
                 DEPTH : natural := 4
                );
        port (
        wr_clk_i     : in std_logic;
        wr_data_i    : in
std_logic_vector(data_WIDTH-1 downto 0);
        wr_addr_i    : in
std_logic_vector(DEPTH-1 downto 0);
        rd_addr_i    : in
std_logic_vector(DEPTH-1 downto 0);
        wr_en_i      : in std_logic;
        rd_data_o    : out
std_logic_vector(data_WIDTH-1 downto 0);
        full_i       : in std_logic
            );
    end component;
```

a. d_p_ram (FIFO Belleği)

FIFO'nun veri saklama kısmıdır. Çift portlu bir RAM (Dual-Port RAM) olarak kullanılır.

Yazma adresi (**wr_addr_i**) ve yazma verisi (**wr_data_i**).

Okuma adresi (**rd_addr_i**) ve okuma çıktısı (**rd_data_o**).

Yazma etkinleştirme (**wr_en_i**) ve FIFO'nun doluluk durumu (**full_i**).

```
    component read_port_control
        generic (DEPTH : natural := 4);
        port (
            rd_clk_i     : in std_logic;
            rd_en_i      : in std_logic;
            rd_rst_i     : in std_logic;
            wr_ptr_i     : in
std_logic_vector(DEPTH downto 0);
            empty_o      : out std_logic;
            rd_addr_o    : out
std_logic_vector(DEPTH-1 downto 0);
            rd_ptr_o     : out
std_logic_vector(DEPTH downto 0)
        );
    end component;
```

b. read_port_control (Okuma Kontrol)

FIFO'nun okuma işlemlerini kontrol eder.

rd_en_i sinyaline göre okuma pointer'ını (**rd_ptr_o**) artırır.

Okuma pointer'ını adres sinyaline (**rd_addr_o**) dönüştürür.

FIFO'nun boş olup olmadığını (**empty_o**) belirler.

- **Bağlantılar**:

  Okuma saati (**rd_clk_i**), sıfırlama (**rd_rst_i**), okuma pointer'ı (**rd_ptr_o**).

```
    component write_port_control
        generic (DEPTH : natural := 4);
        port (
            wr_clk_i    : in std_logic;
            rd_ptr_i    : in
std_logic_vector(DEPTH downto 0);
            wr_rst_i    : in std_logic;
            wr_en_i     : in std_logic;
            full_o      : out std_logic;
            wr_addr_o   : out
std_logic_vector(DEPTH-1 downto 0);
            wr_ptr_o    : out
std_logic_vector(DEPTH downto 0)
        );
    end component;
```

c. write_port_control (Yazma Kontrol)

FIFO'nun yazma işlemlerini kontrol eder.

wr_en_i sinyaline göre yazma pointer'ını (**wr_ptr_o**) artırır.

Yazma pointer'ını adres sinyaline (**wr_addr_o**) dönüştürür.

FIFO'nun dolu olup olmadığını (**full_o**) belirler.

- **Bağlantılar**:

  Yazma saati (**wr_clk_i**), sıfırlama (**wr_rst_i**), yazma pointer'ı (**wr_ptr_o**).

```
    component synchronizer
        generic (DEPTH : natural := 4);
        port (
            clk_i   : in std_logic;
            async_i : in
std_logic_vector(DEPTH downto 0);
            rst_i   : in std_logic;
            sync_o  : out
std_logic_vector(DEPTH downto 0)
        );
    end component;
```

d. synchronizer (Pointer Senkronizasyonu)

Okuma ve yazma pointer'ları farklı saat bölgelerinde çalıştığı için bu pointer'ları senkronize eder.

- Yazma pointer'ını okuma saat bölgesine, okuma pointer'ını yazma saat bölgesine senkronize eder.

- **Bağlantılar**:

  Yazma pointer'ı (**w_ptr_out_s**) → Okuma saati alanına (**w_ptr_in_s**).

  Okuma pointer'ı (**r_ptr_out_s**) → Yazma saati alanına (**r_ptr_in_s**).

Kodun kalan kısmında sadece bağlantılar gerçekleştirilmiştir. O yüzden geri kalan kısmını bu yukarıdaki şekillerde olduğu gibi detaylı şekilde açıklamalarını yapmıyorum. Neyin nereye bağlanacağını görmek için yukarıdaki blok şemadan da detaylı bir şekilde inceleyerek bağlantıları görerek gerçekleştirebilirsiniz.

Test Bench

Burada yazdığımız kodun doğruluğunu test etmek üzere hayali bir sistem kurup testimizi gerçekleştireceğiz. Analize başlamadan önce kodun tamamını aşağıda görebilirsiniz.

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


entity tb_top_async_fifo is
end tb_top_async_fifo;

architecture Behavioral of tb_top_async_fifo is

    constant data_WIDTH : natural := 16;
    constant DEPTH      : natural := 4;

    --clock periods
    constant w_clk_period : time := 10ns;
    constant r_clk_period : time := 20ns;




    signal wr_clk_i     : std_logic;
    signal wr_rst_i     : std_logic;
    signal wr_en_i      : std_logic;
    signal full_o       : std_logic;
    signal wr_data_i    : std_logic_vector (data_WIDTH-1 downto 0);
    signal rd_clk_i     : std_logic;
    signal rd_rst_i     : std_logic;
    signal rd_en_i      : std_logic;
    signal rd_data_o    : std_logic_vector (data_WIDTH-1 downto 0);
    signal empty_o      : std_logic;
    signal wr_addr_o    : std_logic_vector (DEPTH-1 downto 0);
    signal rd_addr_o    : std_logic_vector (DEPTH-1 downto 0);



    component top_async_fifo
        generic (   data_WIDTH  : natural := 16;
                    DEPTH       : natural := 4
                );
        port (
            --write
            wr_clk_i    : in std_logic;
            wr_rst_i    : in std_logic;
            wr_en_i     : in std_logic;
            full_o      : out std_logic;
            wr_data_i   : in std_logic_vector(data_WIDTH-1 downto 0);

            --read
            rd_clk_i    : in std_logic;
            rd_rst_i    : in std_logic;
            rd_en_i     : in std_logic;
            rd_data_o   : out std_logic_vector(data_WIDTH-1 downto 0);
            empty_o     : out std_logic;

            wr_addr_o   : out std_logic_vector(DEPTH-1 downto 0);
            rd_addr_o   : out std_logic_vector(DEPTH-1 downto 0)


            );

    end component;
```

```
begin

ins_top_async : top_async_fifo
    generic map(
                data_WIDTH  => data_WIDTH,
                DEPTH       => DEPTH
                )
    port map    (

                wr_clk_i    => wr_clk_i,
                wr_rst_i    => wr_rst_i,
                wr_en_i     => wr_en_i,
                full_o      => full_o,
                wr_data_i   => wr_data_i,
                rd_clk_i    => rd_clk_i,
                rd_rst_i    => rd_rst_i,
                rd_en_i     => rd_en_i,
                rd_data_o   => rd_data_o,
                empty_o     => empty_o,
                wr_addr_o   => wr_addr_o,
                rd_addr_o   => rd_addr_o

                );


    --write clock
write_clock_process: process
                        begin
                            wr_clk_i <= '1';
                            wait for w_clk_period/2;
                            wr_clk_i <= '0';
                            wait for w_clk_period/2;
                     end process;

    --read clock
read_clock_process: process
                        begin
                            rd_clk_i <= '1';
                            wait for r_clk_period/2;
                            rd_clk_i <= '0';
                            wait for r_clk_period/2;
                    end process;

--test
uut_process:    process
     begin
wr_rst_i <= '1';
rd_rst_i <= '1';
wr_en_i <= '0';
rd_en_i <= '0';
wait for 20ns;

wr_rst_i <= '0';
rd_rst_i <= '0';
wait for 5ns;

--write
wr_en_i <= '1';
wr_data_i <= x"0A00";
wait for w_clk_period;
wr_data_i <= x"1B25";
wait for w_clk_period;
wr_data_i <= x"3CF2";
wait for w_clk_period;
wr_data_i <= x"076A";
wait for w_clk_period;
wr_data_i <= x"4B01";
wait for w_clk_period;
wr_data_i <= x"2FF3";
wait for w_clk_period;
wr_data_i <= x"0D7A";
wait for w_clk_period;
wr_data_i <= x"7A0c";
wait for w_clk_period;
```

```
wr_data_i <= x"0A10";
wait for w_clk_period;
wr_data_i <= x"1B28";
wait for w_clk_period;
wr data i <= x"2CF2";
wait for w_clk_period;
wr_data_i <= x"076B";
wait for w_clk_period;
wr_data_i <= x"4A01";
wait for w_clk_period;
wr_data_i <= x"2F33";
wait for w_clk_period;
wr_data_i <= x"0D7A";
wait for w_clk_period;
wr_data_i <= x"9A0c";
wait for w_clk_period;
wr_en_i <= '0';
wait for w_clk_period*5;
--read
rd_en_i <= '1';
wait for r_clk_period*20;
wr_en_i <= '1';
rd_en_i <= '1';
wr_data_i <= x"0A00";
wait for w_clk_period;
wr_data_i <= x"1B25";
wait for w_clk_period;
wr_data_i <= x"3CF2";
wait for w_clk_period;
wr_data_i <= x"076A";
wait for w_clk_period;
wr_data_i <= x"4B01";
wait for w_clk_period;
wr_data_i <= x"2FF3";
wait for w_clk_period;
wr_data_i <= x"0D7A";
wait for w_clk_period;
wr_data_i <= x"7A0c";
wait for w_clk_period;
```

```
wr_data_i <= x"0A10";
wait for w_clk_period;
wr_data_i <= x"1B28";
wait for w_clk_period;
wr_data_i <= x"2CF2";
wait for w_clk_period;
wr_data_i <= x"076B";
wait for w_clk_period;
wr_data_i <= x"4A01";
wait for w_clk_period;
wr_data_i <= x"2F33";
wait for w_clk_period;
wr_data_i <= x"2D7A";
wait for w_clk_period;
wr_data_i <= x"9A0c";
wait for w_clk_period;



rd_en_i <= '0';
wr_en_i <= '0';

wait for r_clk_period*10;
wait;
end process;

end Behavioral;
```

Detaylı analize başlayalım:

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


entity tb_top_async_fifo is
end tb_top_async_fifo;

architecture Behavioral of tb_top_async_fifo is

    constant data_WIDTH : natural := 16;
    constant DEPTH      : natural := 4;

    --clock periods
    constant w_clk_period : time := 10ns;
    constant r_clk_period : time := 20ns;
```

Her zaman olduğu gibi öncelikle kütüphaneleri tanımlamamız gerekiyor. Ardından entity kısmını boş bırakıyoruz. Test bench uygulanırken burası boş bırakılır. Sonrasında mimari tasarıma başlıyoruz.

data_WIDTH: FIFO'da her bir veri kelimesinin genişliğini temsil eder. 16-bit veri aktarımı yapılacağını belirtir.

DEPTH: FIFO'nun derinliğini (adres uzunluğunu) ifade eder. 4-bit, FIFO'nun 16 (2^4) derinlikte olduğunu gösterir

Saat frekansları ile göründüğü üzere iki adet farklı saat sinyali tanımlanmıştır. Bunun sebebi asenkron bir yapıda olduğu içindir. Yazma ve okuma periyortları için farklı saat frekansları belirlenmiştir. Burada değerler temsili olarak verilmiştir. Farklı değerler de verilebilir. Burada 10ns 50mHz, 20ns ise 100mHz karşılığına gelmektedir.

```
    signal wr_clk_i     : std_logic;
    signal wr_rst_i     : std_logic;
    signal wr_en_i      : std_logic;
    signal full_o       : std_logic;
    signal wr_data_i    : std_logic_vector
(data_WIDTH-1 downto 0);
    signal rd_clk_i     : std_logic;
    signal rd_rst_i     : std_logic;
    signal rd_en_i      : std_logic;
    signal rd_data_o    : std_logic_vector
(data_WIDTH-1 downto 0);
    signal empty_o      : std_logic;
    signal wr_addr_o    : std_logic_vector (DEPTH-1
downto 0);
    signal rd_addr_o    : std_logic_vector (DEPTH-1
downto 0);
```

Burada sinyal tanımlamaları yapılmıştır. Top levelde bulunuan bütün giriş çıkışları burada sinyal olarak tanımlıyoruz.

`wr_clk_i`: Yazma tarafının saat sinyali.

`wr_rst_i`: Yazma tarafı reset sinyali.

`wr_en_i`: Yazma etkinlik sinyali.

`wr_data_i`: FIFO'ya yazılacak veriyi temsil eder.

`full_o`: FIFO'nun dolu olduğunu gösterir.

`rd_clk_i`: Okuma tarafının saat sinyali.

`rd_rst_i`: Okuma tarafı reset sinyali.

`rd_en_i`: Okuma etkinlik sinyali.

`rd_data_o`: FIFO'dan okunan veriyi temsil eder.

`empty_o`: FIFO'nun boş olduğunu gösterir.

`wr_addr_o` ve `rd_addr_o`: FIFO'nun yazma ve okuma adreslerini temsil eder.

```
    component top_async_fifo
        generic (   data_WIDTH  : natural := 16;
                    DEPTH       : natural := 4
                );
        port (
            --write
            wr_clk_i    : in std_logic;
            wr_rst_i    : in std_logic;
            wr_en_i     : in std_logic;
            full_o      : out std_logic;
            wr_data_i   : in std_logic_vector(data_WIDTH-1
downto 0);

            --read
            rd_clk_i    : in std_logic;
            rd_rst_i    : in std_logic;
            rd_en_i     : in std_logic;
            rd_data_o   : out std_logic_vector(data_WIDTH-1
downto 0);
            empty_o     : out std_logic;

            wr_addr_o   : out std_logic_vector(DEPTH-1
downto 0);
            rd_addr_o   : out std_logic_vector(DEPTH-1
downto 0)


            );

    end component;
```

Bu kısım top levelimizi component olarak tanımlamak için kullanılan bir yöntem. Bu modülü buraya instantiation olarak yapmadan önce bu işlemin gerçekleşmesi gerekiyor. Yani kısaca test benchi de top level üzerindeki bir top level olarak düşünebiliriz. O yüzden burada bu tanımlamayı gerçekleştiriyoruz.

```
begin

ins_top_async : top_async_fifo
    generic map(
                data_WIDTH  => data_WIDTH,
                DEPTH       => DEPTH
                )
    port map    (

                wr_clk_i    => wr_clk_i,
                wr_rst_i    => wr_rst_i,
                wr_en_i     => wr_en_i,
                full_o      => full_o,
                wr_data_i   => wr_data_i,
                rd_clk_i    => rd_clk_i,
                rd_rst_i    => rd_rst_i,
                rd_en_i     => rd_en_i,
                rd_data_o   => rd_data_o,
                empty_o     => empty_o,
                wr_addr_o   => wr_addr_o,
                rd_addr_o   => rd_addr_o

                );
```

Burada da bizim başta tanımladığımız sinyaller ile component olarak tanımladığımız top level bloğunun bağlantılarının gerçekleştirildiği kısım. Sayısal tasarımcılar arasında bu kısma instantiation kısmı denir. Örnekleme işlemi, bağlama işlemi burada gerçekleşiyor.

```
    --write clock
write_clock_process: process
                        begin
                            wr_clk_i <= '1';
                            wait for
w_clk_period/2;
                            wr_clk_i <= '0';
                            wait for
w_clk_period/2;
                    end process;

    --read clock
read_clock_process: process
                        begin
                            rd_clk_i <= '1';
                            wait for
r_clk_period/2;
                            rd_clk_i <= '0';
                            wait for
r_clk_period/2;
                    end process;
```

Burada saat üretme işlemlerini gerçekleştiriyoruz.

Yazma tarafı saat sinyali (10 ns periyot) üretilir.

Okuma tarafı saat sinyali (20 ns periyot) üretilir.

Bu işlemden sonra test senaryomuza başlayabiliriz.

```
--test
uut_process:   process
begin
wr_rst_i <= '1';
rd_rst_i <= '1';
wr_en_i <= '0';
rd_en_i <= '0';
wait for 20ns;

wr_rst_i <= '0';
rd_rst_i <= '0';
wait for 5ns;

--write
wr_en_i <= '1';
wr_data_i <= x"0A00";
wait for w_clk_period;
wr_data_i <= x"1B25";
wait for w_clk_period;
wr_data_i <= x"3CF2";
wait for w_clk_period;
wr_data_i <= x"076A";
wait for w_clk_period;
wr_data_i <= x"4B01";
wait for w_clk_period;
wr_data_i <= x"2FF3";
wait for w_clk_period;
wr_data_i <= x"0D7A";
wait for w_clk_period;
wr_data_i <= x"7A0c";
wait for w_clk_period;
wr_data_i <= x"0A10";

wait for w_clk_period;
wr_data_i <= x"1B28";
wait for w_clk_period;
wr_data_i <= x"2CF2";
wait for w_clk_period;
wr_data_i <= x"076B";
wait for w_clk_period;
wr data i <= x"4A01";
wait for w_clk_period;
wr_data_i <= x"2F33";
wait for w_clk_period;
wr_data_i <= x"2D7A";
wait for w_clk_period;
wr data i <= x"9A0c";
wait for w_clk_period;
```

Öncelikle process içerisinde resetleme işlemleri gerçekleştirilir. Yazma ve okuma tarafı için ayrı ayrı reset sinyali oluşturduğumuz için ikisine de ayrı ayrı resetleme işlemi yaptık.

Önce veri yazma işlemimizi gerçekleştirdik. Örnek olarak bir tanesini söyleyeceğim diğerleri zaten aynı mantık ile hareket ettiği için hepsini açıklamayacağım.

wr_en_i sinyalini öncelikle ‘1’ konumuna getiririz. Bunu getirmemizin sebebi yazma işlemine izin verme yani başlayabilmek içindir. Ardından **wr_data_i** portumuza hekzadesimal olarak **x“0A00”** değerini gönderdik. Burası 16 bittir. Biz daha önce verilerimizin genişliğinin 16 bit olmasını istediğimiz için burada 16 bit değerine karşılık gelen bu rastgele değeri verdik. Ardından yukarıda tanımladığımız yazma saat sinyali periyodu kadar bekliyoruz. Sonrasında ise **wr_data_i** portuna başka bir değer olan **x”1B5”** değerini giriyoruz. Bu şekilde bunu 16 adet olana kadar devam ettiriyorum. 16 adet devam ettirmemin sebebi FIFO derinliğimiz 16’dır. Ben de burada 16 adet veri yazdıktan sonra **full** sinyalinin ‘1’ değerini görmesini ve bunu simülasyon ekranında görmeyi umuyorum.

```
wr_en_i <= '0';
wait for w_clk_period*5;
--read
rd_en_i <= '1';
wait for r_clk_period*20;
wr_en_i <= '1';
rd_en_i <= '1';
wr_data_i <= x"0A00";
wait for w_clk_period;
wr_data_i <= x"1B25";
wait for w_clk_period;
wr_data_i <= x"3CF2";
wait for w_clk_period;
wr_data_i <= x"076A";
wait for w_clk_period;
wr_data_i <= x"4B01";
wait for w_clk_period;
wr_data_i <= x"2FF3";
wait for w_clk_period;
wr_data_i <= x"0D7A";
wait for w_clk_period;
wr_data_i <= x"7A0c";
wait for w_clk_period;


wr_data_i <= x"0A10";
wait for w_clk_period;
wr_data_i <= x"1B28";
wait for w_clk_period;
wr data i <= x"2CF2";
wait for w_clk_period;
wr_data_i <= x"076B";
wait for w_clk_period;
wr_data_i <= x"4A01";
wait for w clk period;
wr_data_i <= x"2F33";
wait for w_clk_period;
wr_data_i <= x"2D7A";
wait for w_clk_period;
wr_data_i <= x"9A0c";
wait for w_clk_period;
rd_en_i <= '0';
wr_en_i <= '0';
wait for r_clk_period*10;
wait;
end process;
end Behavioral;
```

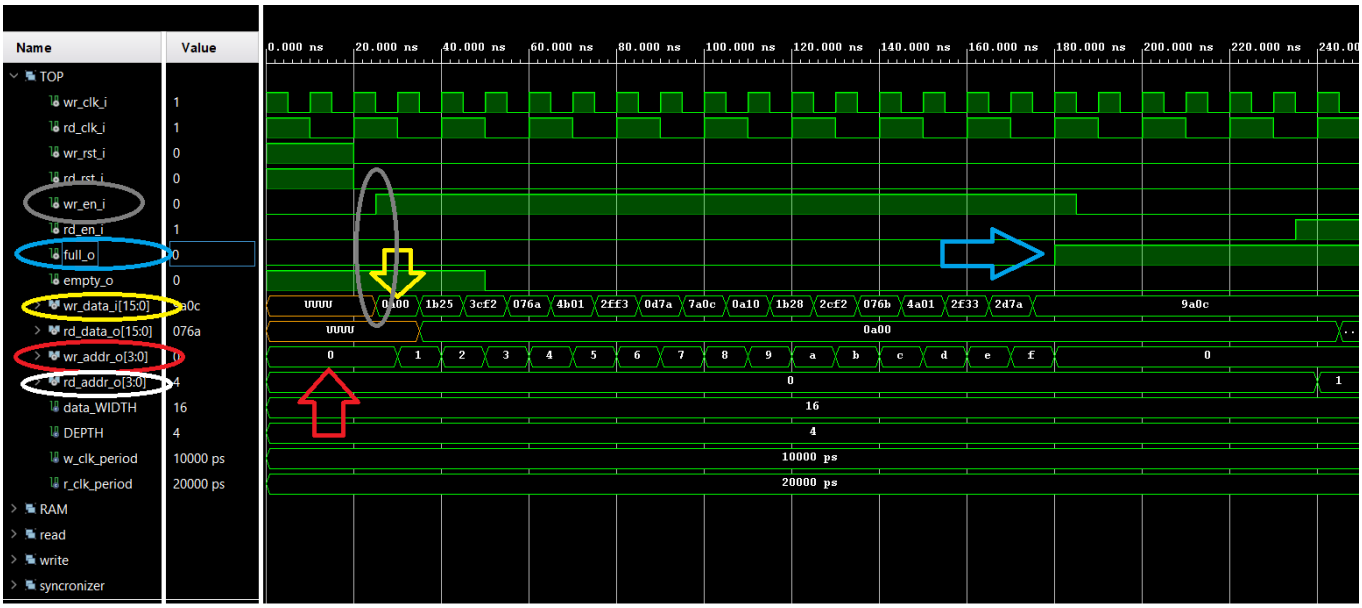
Burada ise önce yazma işlemini durdurmak için **wr_en_i** sinyalini ‘0’ konumuna çekip yazma işlemini durduruyoruz. Sonrasında ise yazma işlemi gerçekleşmemesi için farazi olarak bir süre beklemeye alıyoruz. Ben bunu **w_wlck_period***5 kez dedim. Burada böyle dememin sebebi simülasyon ekranında sonuçları daha rahat görebilmek ve okunabilirliği arttırmak içindir. Farklı süreler de verilebilir. Tam bitirilip ardından başlama işlemine geçilebilir. Bunlar isteğe bağlıdır ve bu bir deneme, örnek bir tasarım olduğu için kafama göre bekleme süresi uyguladım.

Ardından okuma işlemini gerçekleştirmesi için **rd_en_i** sinyalini aktif yapıp ‘1’ konumuna getirdim. Bunu da yine ne kadar süre okumasını istiyorsak o kadar komut verebiliriz. Ben burada yine 20ns süre boyunca okuma yapmasını istedim.

Sonrasında FIFO’nun okuma ve yazma işlemlerini aynı anda yapabilme durumu gerçekleşiyor mu diye bakmak için **wr_en_i** ve **rd_en_i** sinyallerinin ikisini birden aktif ettim. Böylece farklı saat domainlerinde hem okuma hem yazma işlemi gerçekleşebilecektir.

Bu durumu da test ettikten sonra okuma ve yazma aktif sinyallerimi sıfıra çekip simülasyonu bitiriyorum.

Şimdi bu yaptıklarımızın doğruluğunu simülasyon ekranında görelim.

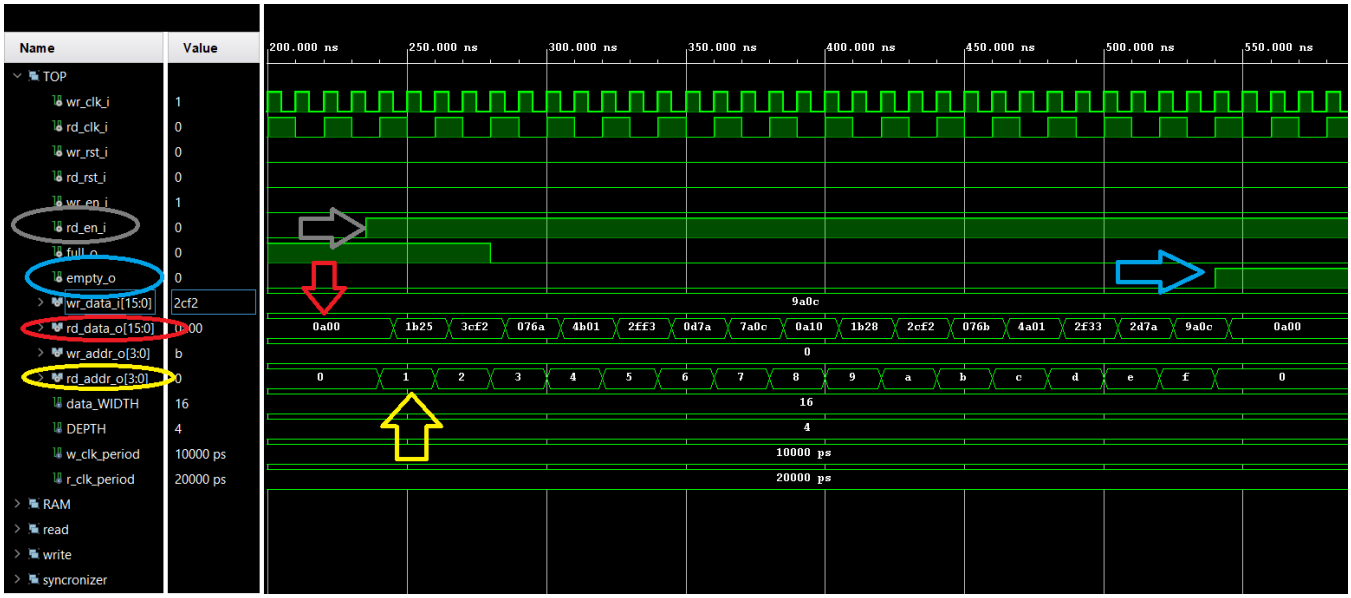


Sarı ile gösterilen işaretlerde **wr_data_i** sinyalinden gelen değerlerin sıra ile yazıldığını görebilirsiniz.

Gri ile gösterilen işaretlerde **wr_en_i** sinyalinin aktif olduktan sonra yazma işlemini yapmaya başladığını görebilirsiniz.

Kırmızı ile gösterilen işaretlerde her yazma işleminden sonra sayma işlemini yaptığını ve sıra ile saydığını görebilirsiniz. Ayrıca bu bizim RAM'e de sıra ile yazdığımızı göstermektedir.

Mavi ile gösterilen işaretlerde **full_o** sinyali RAM dolduktan yani 16 adet veriyi kaydettikten sonra dolu olma durumunu göstermektedir. Bu sinyal aktif olduktan sonra yazma işlemi kesilecektir.

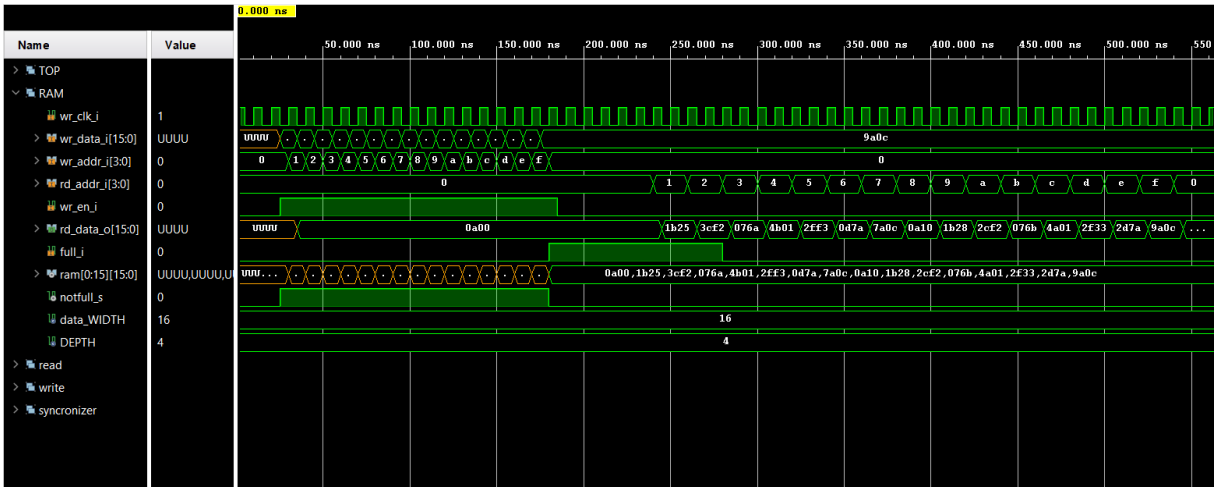


Kırmızı ile gösterilen işaretlerde başta yazdığımız değerleri sırası ile okuma işlemini gerçekleştirdiğini görebilirsiniz.

Sarı ile gösterilen işaretlerde okuma işleminin sıra ile saydığını görebilirsiniz.

Gri ile gösterilen işaretlerde rd_en_i sinyali aktif olduktan sonra okuma işlemine başladığını görebilirsiniz.

Mavi ile gösterilen işaretlerde 16 adet verinin okunduktan sonra FIFO'nun boş olduğunu gösteren empty_o sinyalinin aktif olarak boş olduğunu görebilirsiniz.



Burada RAM içerisine değerlerin kaydedildiğini görebilirsiniz.

Sırayla içerisine kaydedildiğini buradaki **ram** portundan görebilirsiniz.

Yine okuma işleminin de sırayla yapıldığını, full_i sinyalinin FIFO dolduğu zaman '1' olduğunu gözlemleyebilirsiniz.

KAYNAKÇA

https://vlsiverify.com/verilog/verilog-codes/asynchronous-fifo/

https://www.verilogpro.com/asynchronous-fifo-design/

https://www.youtube.com/playlist?list=PL6jcjOP0HjMpLtYgpWJzhe4uPfGimq0Cy

https://www.sunburst-design.com/papers/CummingsSNUG2008Boston_CDC.pdf

https://www.sunburst-design.com/papers/CummingsSNUG2002SJ_FIFO1.pdf