SIRALI MANTIK (SEQUENTIAL LOGIC)

1. GİRİŞ

Bugüne kadar Verilog dilini kapı seviyesinde ve davranışsal modelleme yaparak sadece tümleşik (combinational) devreleri gerçekleştirmek için kullandık. Tümleşik devrelerde çıkışlar sadece o anda verilen girişlere bağlı olarak değişir. Bundan sonraki haftalarda tasarımlarımızda sıralı mantık da kullanacağız. Tümleşik devrelerden farklı olarak, sıralı mantık (sequential logic) devrelerinde çıkışlar sadece o anki girişlere bağlı olarak değil, giriş sinyallerinin geçmiş değerlerine bağlı olarak da hesaplanır. Sıralı mantık içeren devreler her saat vuruşunda işlem yapar ve bir saat vuruşundan bir sonraki saat vuruşuna kadar bilgi saklayan hafıza elemanları içerirler.

Dökümanın devamında, Verilog ile Davranışsal Modelleme kullanarak **sıralı mantık** devrelerinin nasıl gerçekleştirildiği anlatılacaktır.

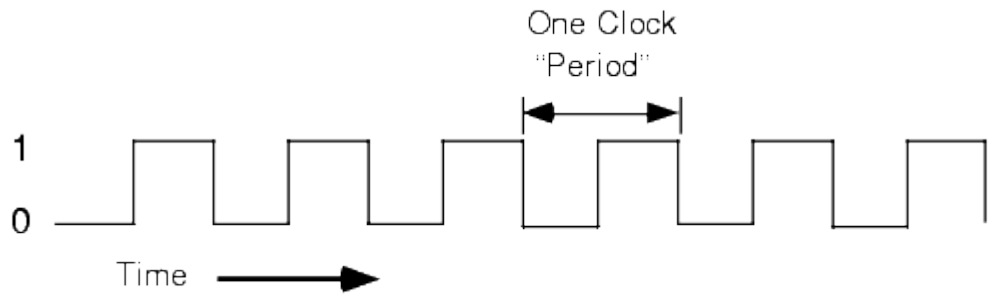
2. SIRALI MANTIK DEVRELERİ

Sıralı Mantık Devreleri, çıkışları yalnızca o anki girişlere değil, girişlerin geçmiş değerlerine de bağlıdır. Girişlerin geçmiş durumları ile bilgiler sıralı mantık devrelerindeki hafıza elemanlarında saklanır. Sıralı mantıkta işlemler bir **saat** sinyaline bağlı olarak yapılır. Flip-flop gibi hafıza birimleri ardışık iki saat vuruşu arasında bilgi tutulmasını sağlar.

Sıralı mantığa örnek asansörlerde kullanılan denetim birimlerinden verilebilir. 10 katlı bir binada, bulunmakta olduğunuz 5. kattan asasörü çağırma düğmesine bastığınızı düşünün. Asansörü kontrol eden donanım, asansörün yukarı mı çıkacağı, aşağı mı ineceği ya da yalnızca kapıları mı açacağı kararını sadece sizin 5. kattan gönderdiğiniz sinyale göre vermez. Bu sinyalin yanında, asansörün hangi katta bulunduğu bilgisi de kullanılır. Örneğin; düğmeye bastığınız anda eğer asansör 5'ten yukarıda bir katta bekliyorsa, asansör aşağıya inmeye başlayacaktır. Asansörün hangi katta bulunduğu bilgisi, asansörün denetim biriminde bulunan sıralı mantık ile gerçekleştirilmiş hafıza birimlerinde tutulur.

Verilog ile D Flip-Flop Gerçekleştirilmesi

Aşağıdaki kodda D Flip-Flop modülünün Verilog ile nasıl gerçekleştirilebileceği gösterilmiştir. D Flip-Flop birimi, kendisine gelen tek bitlik veri girişini her saat vuruşunda bir sonraki saat vuruşu için kaydeder. Koddaki "D" girişi sözü edilen tek bitlik veri girişidir. "clk" girişi ise periyodik olarak sürekli mantık-0 veya mantık-1 değerlerini alan saat (clock) sinyalidir. Herhangi bir saat sinyalinin karedalga görünümü aşağıda gösterildiği gibidir. Periyot, ardışık iki yükselen veya ardışık iki alçalan kenar arasındaki zaman farkı olarak tanımlanır.



```
module DFlipFlop (
        input D,
        input clk,
        input rst,
        input en,
        output reg Q);

        always@(posedge clk) begin
                if(rst) begin
                        Q <= 1'b0;
                end
                else begin
                        if(en)
                                Q <= D;
                end
        end
endmodule
```

Kodda, always bloğunun Sensitivity List'i "posedge clk" (Positive Edge) olarak tanımlanmış. Bu, always bloğunun saatin ("clk" sinyalinin) her yükselen kenarında çalıştırılacağı anlamına gelmektedir. Always'in alçalan kenarda çalışmasını sağlamak için Sensitivity List'i "negedge clk" (Negative Edge) olarak belirtmemiz gerekir.

Always bloğu "clk" sinyalinin her yükselen kenarında "rst" ve "en" giriş sinyallerinin durumuna göre farklı iş yapar. "rst" girişinin mantık-1 olması durumunda "Q" çıkışına mantık-0 değeri atanır. Always bloğu bir daha ancak bir sonraki yükselen kenarda çalıştırılacağı için, bir periyotluk süre boyunca bu modülün "Q" çıkışı hep mantık-0 kalır. Saatin yükselen kenarında "rst" girişinin mantık-0 olması durumunda ise, eğer "en" sinyali mantık-1 ise "D" girişinden gelen veri "Q" çıkışına yönlendirilir. Yine benzer şekilde, bir periyotluk süre boyunca (always bloğu saatin bir sonraki yükselen kenarında tekrar çalıştırılıncaya kadar) "Q" çıkışı hep saatin o yükselen kenarındaki "D" girişine eşit olur. Eğer "en" girişi mantık-0 ise, "Q" çıkışı bir önceki durumunu korumaya devam eder (Q'ya herhangi bir atama yapılmadığı için).

"Q" çıkışına atama yapmak için yukarıdaki kodda kullanılan "<=" operatörüne *Non-Blocking Assignment Operator* denir. Non-blocking atama yapıldığında, atama yapılan reg'in değeri always bloğunun sonunda değişir. Yani, "Q" atama yapıldıktan sonra okunsaydi, atama yapılmadan önceki değeri kullanılmış olacaktı. "=" operatörü ile gösterilen *Blocking Assignment* kullanıldığında ise atama yapıldıktan sonraki satırlarda, atama yapılan reg'in yeni değeri (atama sonrası) kullanılır. Blocking ve Non-blocking atama operatörleri ile ilgili ayrıntılı açıklamalar 3. Bölümde verilmiştir.

3. BLOCKİNG VE NON-BLOCKİNG ASSİGNMENT

Verilog'da always blokları içinde reg türündeki sinyallere iki şekilde atama yapmak mümkündür. "=" operatörü ile Blocking Assignment (atama), "<=" operatörü ile ise Non-Blocking Assignment yapılır.

Blocking atamada operatörün sağındaki işlemlerin yapılıp, atama yapılacak değerin hesaplanması ve o değerin operatörün sağındaki reg'e atanması işlemleri beraber o an yapılır. Non-blocking atamada ise, operatörün sağ tarafı hemen hesaplanır fakat değer operatörün solundaki reg'e hemen yazılmaz. Reg'e yeni değer ancak always bloğunun sonunda, diğer tüm işlemler yapıldıktan sonra yazılır. Yani, eğer reg'in değeri non-blocking atamadan sonra okunmak istenirse, henüz o reg'e atama yapılmamış olacağından, reg'in eski değeri okunmuş olur.

"=" ve "<=" operatörlerinin farkını daha iyi anlamak için aşağıdaki örnek kod parçalarını inceleyelim.

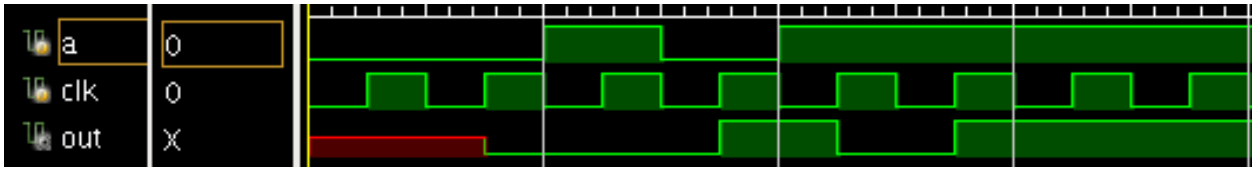
```
module mBlocking (
        input a,
        input clk,
        output reg out);

        reg b;
        always@(posedge clk) begin
                b = a;
                out = b;
        end
endmodule
```

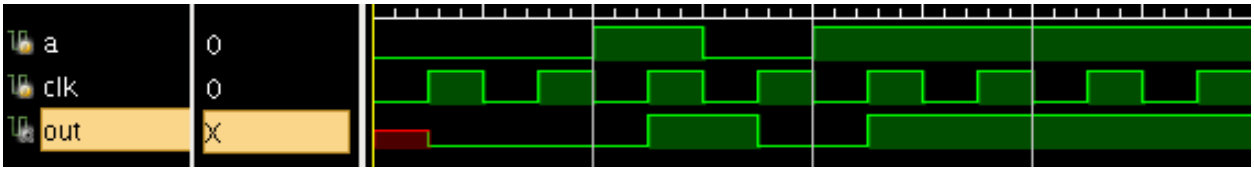
```
module mNonBlocking (
        input a,
        input clk,
        output reg out);

        reg b;
        always@(posedge clk) begin
                b <= a;
                out <= b;
        end
endmodule
```

Gördüğünüz gibi iki kod parçası arasındaki tek fark always bloğu içinde kullanılan atama operatörüdür. Sol tarafta bulunan blocking atama kullanılan kod parçasında "a" girişinin değeri ne ise, "clk" sinyalinin yükselen kenarından sonra "out" çıkışından o değer okunur. Sağ taraftaki non-blocking atama yapılan durumda ise "clk" sinyalinin yükselen kenarından sonra "out" çıkışından "b" reg'inin eski değeri (önceki çevrimdeki "a" girişi) okunur.



Şekil 1 - Blocking Assignment Waveform Görüntüsü



Şekil 2 - Non-Blocking Assignment Waveform Görüntüsü

Yukarıda yer alan iki waveform görünümünde blocking ve non-blocking atama yapılan durumlarda "out" çıkışının nasıl değiştiği gösterilmiştir. Şekil 1'de, "a" girişinin mantık-1 olmasından sonra "clk" sinyalinin ilk yükselen kenarıyla "out" çıkışı da mantık-1 değerini almıştır.

Şekil 2'deki non-blocking atama kullanılması durumunda ise, "out" çıkışı ikinci yükselen kenarda mantık-1 değerini almıştır. Bunun sebebi, non-blocking atama kullanılan durumda, ilk yükselen kenarda "b" reg'inin eski değerinin (yani "a" girişinin bir önceki çevrimdeki değeri olan mantık-0) "out" çıkışına atama yaparken kullanılmasıdır.

Güzel ve anlaşılır Verilog kodu yazmak için, konu ile ilgili deneyimi olan donanım geliştiricileri, tümleşik devre tasarımları (always@* şeklindeki bloklar) için blocking atama ("="), sıralı mantık (always@(posedge clk) şeklinde bloklar) için ise non-blocking atama ("<=") kullanılmasını önermektedirler. Siz de uygulamalarınızda bu önerilere uygun olarak kod yazarsanız beklenmedik ve nedeninin tespit edilmesi zor problemlerle karşılaşma olasılığınızı düşürebilirseniz. Fakat atama operatörlerini ne yaptığınızın farkında olarak karışık olarak da kullanmayı tercih edebilirsiniz.

4. SAYAÇ ÖRNEĞİ

Bu bölümde, sıralı mantık içeren basit bir sayaç gerçekleştireceğiz. Bu sayaç 3 bitlik olup, devreye güç verilmesiyle birlikte 0'dan 7'ye kadar saatin her yükselen kenarında bir artarak sayacaktır. Sayaç, 7 değerine ulaştıktan sonra ise benzer şekilde geriye doğru 0'a kadar azalacaktır. Bu şekilde sürekli 0 ile 7 değerleri arasında gidip gelecektir. Ayrıca devre, sayacı duraklatma özelliğine de sahip olacaktır.

Yukarıdaki paragrafta anlatılan özellikleri sağlayan sayaç modülü Verilog ile aşağıdaki gibi gerçekleştirilebilir.

```
module Sayac (
    input duraklat,
    input clk,
    output[2:0] out_sayac
  );

  localparam YUKARI_SAY = 1'b0;
  localparam ASAGI_SAY = 1'b1;

  reg[2:0] sayi = 3'd0, sayi_sonraki;
  reg durum = YUKARI_SAY; // yukari veya asagi yonde sayma
          // varsayilan olarak baslangicta yukari dogru sayiyoruz
  reg durum_sonraki;


  always@* begin

    sayi_sonraki = sayi;
    durum_sonraki = durum;

    if(!duraklat) begin
      case(durum)
        YUKARI_SAY: begin
          sayi_sonraki = sayi + 3'd1;

          if(sayi_sonraki == 3'd7)
            durum_sonraki = ASAGI_SAY;
        end //YUKARI_SAY

        ASAGI_SAY: begin
          sayi_sonraki = sayi - 3'd1;

          if(sayi_sonraki == 3'd0)
            durum_sonraki = YUKARI_SAY;
        end //ASAGI_SAY
      endcase
    end
  end

  always@(posedge clk) begin
    sayi <= sayi_sonraki;
    durum <= durum_sonraki;
  end

  assign out_sayac = sayi;

endmodule
```

Sayaç modülünde, "sayi" ve "durum" reg sinyalleri ile bilgiyi bir sonraki çevrim için saklayan hafıza elemanları gerçekleştirilmiştir. "sayi" ile sayacın o çevrimde gösterdiği ve modülün "out_sayac" çıkışına verilen sayısal değer tutulmaktadır. "durum" ise sayacın yukarı veya aşağı yönde sayması gerektiğini gösterir.

Yukarıdaki kod, tümleşik mantık ile sıralı mantığı birbirinden tamamen ayrıştıracak şekilde yazılmıştır. *always@(posedge clk)* şeklinde olan blokta sadece sıralı mantık, *always@** şeklinde olan blokta ise sadece tümleşik mantık yer almaktadır. Bu yöntem, hata yapma ihtimalini düşürdüğünden giriş seviyesindeki donanım geliştiricileri için önerilmektedir. Ayrıca, bu yöntemi kullanarak sıralı mantık için her zaman "<=" atama operatörü, tümleşik mantık için ise her zaman "=" atama operatörü kullanıldığından bu operatörlerin yanlış kullanımından doğabilecek hataların da önüne geçilebilmektedir.

Yukarıdaki kodda kullanılan kodlama yönteminde, always@(posedge clk) bloğunda atama yapılarak hafıza oluşturacak her bir reg sinyalinin bir de tümleşik mantık için kullanılacak bir kopyası oluşturulmaktadır. Bu kopya, hafıza elemanının bir sonraki çevrimde alacağı değeri hesapladığından, kodda bu sinyaller "_sonraki" şeklinde bir uzantıyla gösterilmiştir (Örneğin; "sayi_sonraki" ve "durum_sonraki") . Tümleşik mantığı oluşturan always@* bloğunun en başında "_sonraki" şeklinde reg'lere, hafızayı oluşturan diğer reg'ler atanır. Böylece, eğer o çevrimde bazı koşullar sağlanmaz (Örneğin; "duraklat" girişi mantık-1 ise) ve hafızadaki değer değiştirilmek istenmezse, hafızadaki bilgiler olduğu gibi kalmış olur. "_sonraki" şeklindeki sinyallerin tümleşik devre bloğunda yeni değerleri hesaplanarak, saatin yükselen kenarında sıralı mantık bloğunda hafıza birimlerine ("sayi" ve "durum") atanır.

Sayaç modülü için testbench dosyaysı aşağıdaki gibi yazılır. Sadece tümleşik mantık içeren modüller için daha önce yazdığımız testbench'ten tek farkı "clk" saat girişi için periyodik bir sinyal oluşturmamız gerekmesidir. Aşağıda verilen testbench'te periyodu 10 nanosaniye olan bir saat sinyali oluşturulmuştur. Bunun için sensitivity list'i olmayan always bloğu kullanılmıştır. Bu şekildeki always blokları sonsuz döngü halinde çalışır. "clk" sinyalinin her 5 nanosaniyede (#5) bir tersi alınarak periyodu 10 nanosaniye olan bir saat sinyali üretilmiştir.

Dikkat: *Sensitivity list'i olmayan (yani sonsuz döngü oluşturan) always blokları içerisinde mutlaka belli bir süre bekleme olmalıdır (Örneğin; #10;). Eğer bu bekleme olmazsa always bloğu hiç duraksamadan çalışacağından bilgisayarınız sürekli o always bloğunu çalıştırmaya çalışacağından başka işlemler için yanıt vermeyebilir.*

```
module tb_sayac();

  reg clk = 0;
  reg duraklat;
  wire[2:0] sayac;

  Sayac uut(
    .duraklat(duraklat),
    .clk(clk),
    .out_sayac(sayac)
  );

  // periyodu 10ns olan saat sinyali uretimi
  always begin
    #5;
    clk = ~clk;
  end

  initial begin
    duraklat = 1'b0;
    #100;
    duraklat = 1'b1;
    #20;
    duraklat = 1'b0;
  end
endmodule
```