VERİLOG İLE DAVRANIŞSAL MODELLEME

1. GİRİŞ

Şu ana kadar Verilog dilini, donanım birimlerini yalnızca kapı seviyesinde tanımlamak üzere kullandık. Bu yöntem, sayısal devrelerin çok detaylı olarak tasarlanabilmesine imkan sağlamasına rağmen, büyük devreler için karmaşıklık çok arttığından bu devreleri kapı seviyesinde tasarlamak oldukça güçtür.

Karmaşık devrelerin daha kolay gerçekleştirilebilmesi için **Davranışsal Modelleme** kullanılır. Verilog ile Davranışsal Modelleme yaparak devrenin sağlaması gereken işlevsellik daha soyut bir biçimde ifade edilir. Bu soyut tanımlamanın çalışan bir devre haline dönüştürülmesi donanım sentezleme yazılımları (Örneğin; Vivado) tarafından yapılır.

Bu dökümanın devamında, Verilog ile Davranışsal Modellemenin nasıl yapıldığı anlatılacaktır.

2. DAVRANIŞSAL MODELLEME

Davranışsal Modellemede devrenin sağlaması gereken işlevsellik yordamsal (procedural) olarak tanımlanır. Verilog'da "*initial*" ve "*always*" keyword'leri yordam tanımlamak için kullanılır. Bu yordamlara *initial* veya *always* **bloğu** denir. *initial* ve *always* blokları birçok statement'tan oluşur (Statement'lar ";" ile ayrılan kod parçalarıdır.).

initial blokları: Bu blokların içinde tanımlanan işlemler, devre çalışmaya başlar başlamaz yalnızca bir kere yapılır. initial blokları, kodunuz ilk defa çalıştığında bazı değerlere atama yapmak için kullanılır. Örneğin bir saatin açıldığında saat 02.13'ten başlamasını, initial bloğunun içine yazacağınız kod ile tanımlayabilirsiniz.

always blokları: Bu bloklar, verilen *Sensitivity List*'e göre sürekli çalıştırılır. Tasarım dosyalarında devrenin çalışma mantığı bu kısımda tanımlanır.

Önceki haftalarda kapı seviyesinde gerçekleştirdiğimiz 3 bitlik toplayıcı devresi, Davranışsal Modelleme kullanılarak aşağıdaki gibi kodlanabilir.

```
module Adder3Bit_behavioral (
        input[2:0] num1,
        input[2:0] num2,
        output reg[3:0] sum);

        always@(num1 or num2)
begin
                sum = num1 + num2;
        end
endmodule
```

Davranışsal modellemede, giriş/çıkış sinyalleri kapı seviyesinde olduğu gibi modül isminden hemen sonra parantez içinde belirtilir. Daha önce tasarladığınız modüllerden farklı olarak, yukarıdaki örnekte çıkış sinyalinin "output **reg**" olarak tanımlanmış olduğuna dikkat edin. Bunun sebebi, "sum" çıkışına bir "always" bloğu içinde atam yapılmasıdır.

always bloğunda "@(num1 or num2)" şeklinde yazılan kısım, bu bloğun çalıştırılma koşulunu belirtir. "@" simgesinden sonra parantez içinde verilen koşullara **Sensitivity List** (hassasiyet listesi) denir. Verilen sensitivity list'teki değerlerde herhangi bir değişiklik olduğunda *always* bloğu tekrar çalıştırılır. Yukarıdaki örnekte, sensitivity list'te toplamı hesaplanacak iki giriş sinyali vardır. Bu giriş sinyallerinin değerleri (yani toplanacak sayı değerleri) değiştiğinde "sum" çıkışı için yeni bir değer hesaplanır ve toplam bu çıkıştan dışarı verilir.

always bloğunda sensitivity list "@*" olarak verildiğinde, bu sensitivity list'te *always* bloğu içinde okunan tüm sinyallerin (eşittirlerin sağındaki sinyallerin) yer alacağı anlamına gelmektedir. Yukarıdaki örnekte "@(num1,num2)" yerine "@*" yazdığımızda aynı devre elde edilir, çünkü *always* bloğu içinde "num1" ve "num2" sinyalleri okunarak, bunların toplamı "sum" sinyaline yazılmaktadır.

3. WİRE VE REG

(Bu bölüm https://inst.eecs.berkeley.edu/~cs150/Documents/Nets.pdf dökümanından yararlanılarak yazılmıştır.)

Verilogda "**wire**" olarak tanımlanan değişkenleri, elektronik devrelerde kullanılan kablolar gibi düşünebilirsiniz. Bu tip değişkenlerin özellikleri aşağıda verilmiştir:

- "wire" sinyalleri bir modül örneğinin (instance) girişine başka bir modül örneğinin çıkışını bağlamak için kullanılır.
- "wire" sinyalleri mutlaka bir devre tarafından sürülmelidir. Bu sinyaller kendi başlarına değer saklayamazlar.
- "wire" sinyalleri tasarlanan modülün giriş/çıkışlarını tanımlarken kullanılır.
- "wire" sinyallerine always blokları içinde "=" veya "<=" operatörleri ile atama yapılamaz.
- "wire" sinyallerine yalnızca "assign" kullanılarak atama yapılabilir veya bu sinyaller doğrudan modül örneklerinin giriş/çıkışlarına bağlanabilir.
- "wire" ile yalnızca tümleşik (combinational) devre tasarımı yapılabilir.

Aşağıda "wire" sinyalleri için kullanım örnekleri verilmiştir.

```
wire A , B , C , D , E ; // tek bitlik wire sinyalleri
wire [8:0] Wide ; // 9 bitlik wire sinyali
reg I ;

assign A = B & C ; // assign ile wire'a atama yapmak

 always @ ( B or C ) begin
        I = B | C ; // always blogunda atama operatorunun saginda (okunan deger olarak)
                    // wire kullanimi
end

mymodule mymodule_instance (. In ( D ) ,
                            . Out ( E ) ) ; // wire'larin modul orneklerinin
                                            // giris/cikislarina baglanmasi
```

"reg" sinyalleri "wire" sinyallerine benzemekle beraber, bunlardan farklı olarak durum (state) bilgisi tutarak hafıza birimleri (yazmaç (register) gibi) oluşturulmasını da sağlar. Bu sinyallerin kullanımı aşağıdaki gibidir:

- "reg" sinyalleri modül örneklerinin girişlerine bağlanabilir.
- "reg" sinyalleri modül örneklerinin çıkışlarına bağlanamaz.
- Tasarlanan modülün çıkışları "reg" olarak tanımlanabilir.
- Tasarlanan modülün girişleri "reg" olarak tanımlanamaz.
- always bloğu içinde sadece "reg" türü sinyallere "=" ve "<=" operatörleri ile atama yapılabilir.
- Genellikle testbench dosyalarında kullanılan initial bloklarında yalnızca "reg" türü sinyallere atama yapılabilir.
- "reg" sinyallerine "assign" ile atama yapılamaz.
- always@(posedge clock) ile "reg" sinyalleri hafıza tutan birimlere dönüştürülebilir.
- "reg" sinyalleri hem tümleşik (combinational) hem de sıralı (sequential) devreler oluşturmak için kullanılabilir

Aşağıda "reg" sinyalleri için kullanım örnekleri verilmiştir.

```
wire A , B ;
reg I, J, K ; //1 bitlik reg sinyalleri
reg[8:0] Wide; //9 bitlik reg sinyali

 always @ ( A or B ) begin
        I = A | B ; // always blogunda atama operatorunun solunda (yazılan deger olarak)
                    // reg kullanimi
end

initial begin // initial blogu icinde reg kullanimi
        J = 1'b1;
        #1;
        J = 1'b0;
end

always @(posedge clock) begin
        K <= I; // reg kullanarak saatin yukselen kenarı ile calisan yazmac (register)
                // olusturmak
end
```

"wire" ve "reg" sinyalleri bazı durumlarda birbirleri yerine kullanılabilir:

- Her ikisi de atama işlemlerinde (hem "assign" hem "always" içinde) eşitliğin sağında yer alabilir.
- Her ikisi de modül örneklerinin girişlerine bağlanabilir

Tablo 1: wire ve reg değişkenlerinin kullanımı.

		wire	reg
Tasarlanan Modül	giriş	✓	✗
	çıkış	✓	✓
Eklenen Modül	giriş	✓	✓
	çıkış	✓	✗
Kapı Seviyesinde Modelleme	wire atamaları	assign	assign
	reg atamaları	✗	✗
Davranışsal Modelleme	wire atamaları	✗	✗
	reg atamaları	=	=

4. DAVRANIŞSAL MODELLEME İLE ADDERLEDS MODÜLÜ

Önceki haftalarda yarım ve tam toplayıcılar kullanarak ve kapı seviyesinde tasarım yaparak iki tane iki bitlik sayının toplamı kadar LED ışık yanmasını sağlayan bir devre gerçekleştirmiştir. Bu devre; davranışsal modelleme kullanarak, devreyi kapı seviyesinde oluşturmadan, daha hızlı bir şekilde gerçekleştirilebilir.

```
module AdderLeds_behavioral (
        input[1:0] val1,
        input[1:0] val2,
        output reg[5:0] leds);

        wire[3:0] sum;
        Adder3bit adder(
                .num1({1'b0, val1}),
                .num2({1'b0, val2}),
                .sum(sum)
         );

        //1. Yontem (if-else)
        always@* begin
                leds = 6'b000_000;
                if(sum == 4'd0) begin
                        leds = 6'b000_000;
                end
                else if(sum == 4'd1)
                        leds = 6'b000_001;
                else if(sum == 4'd2)
                        leds = 6'b000_011;
                else if(sum == 4'd3)
                        leds = 6'b000_111;
                else if(sum == 4'd4)
                        leds = 6'b001_111;
                else if(sum == 4'd5)
                        leds = 6'b011_111;
                else if(sum == 4'd6)
                        leds = 6'b111_111;
                // adder modulune bagladigimiz girislerin en anlamli (most-significant)
                // bitleri 0 oldugundan, sum en fazla 6 olabilir.
                // Diger degerleri if ile kontrol etmemiz gerekmiyor.
        end
endmodule
```

Girişlerin toplamını, daha önce yapmış olduğunuz "Adder3bit" modülünü kullanarak hesaplayabilirsiniz. Geriye sadece bu toplam kadar LED ışığı yakmak kalıyor. "leds" çıkışını "reg" olarak tanımladıktan sonra, bu çıkışa *always* bloğu içinde atama yapabilirsiniz. *always* bloklarındaki işlemler sıralı olarak yapıldığından bu bloğun içindeki ilk satırda "leds" çıkışına 6 bit mantık-0 göndererek tüm ledlerin sönmesini sağlayın. Daha sonra, belli koşullara göre bu çıkışın bazı bitlerini mantık-1 yapmanız gerekiyor.

Always blokları içinde *if-else* keywordlerini kullanarak koşula bağlı olarak atama işlemleri gerçekleştirebilirsiniz. Verilen kodda ilk *if* koşulu, toplamın sıfır olup olmadığını kontrol etmektedir. Eğer bu koşul doğruysa LEDler kapalı durumda kalmaya devam eder. *if* koşulundan sonra "begin" ve "end" kullanarak bu koşula bağlı birden fazla işlem yapılabilir. Eğer sadece bir işlem yapılacaksa (tek statement), "begin" ve "end" yazmaya gerek yoktur (kodda yer alan diğer koşullardaki gibi).

```
module AdderLeds_behavioral2 (
        input[1:0] val1,
        input[1:0] val2,
        output reg[5:0] leds);

        wire[3:0] sum;
        Adder3bit adder(
                .num1({1'b0, val1}),
                .num2({1'b0, val2}),
                .sum(sum)
         );

        //2. Yontem (case)
        always@* begin
                leds = 6'b000_000;
                case (sum)
                        4'd0: begin
                                leds = 6'b000_000;
                        end
                        4'd1: leds = 6'b000_001;
                        4'd2: leds = 6'b000_011;
                        4'd3: leds = 6'b000_111;
                        4'd4: leds = 6'b001_111;
                        4'd5: leds = 6'b011_111;
                        4'd6: leds = 6'b111_111;
                endcase
        end
endmodule
```

"sum" sıfır olmadığında diğer koşulların kontrol edilmesini, kodun devamında *else if* kullanarak sağlayacağız. "**else if (sum == 4'd1)**" koşulu, toplamın 1 olup olmadığına bakmakta. Toplamın bir olması durumunda, en sağdaki LED açık duruma getirilmektedir. always bloğu içinde önceki satırlarda atama yapılan bir reg sinyaline tekrar atama yapıldığında, ilk atama geçersiz (hiç yapılmamış gibi) sayılmaktadır. Verilen koşula benzer şekilde, toplamın alabileceği diğer değerler de kontrol edilerek, LEDleri uygun bir şekilde yakan devre tamamlanır.

if-else kullanmak dışında, aynı işlevselliği sağlayan devre, *case* keywordu kullanılarak da gerçekleştirilebilir. AdderLeds modülünün *case* kullanılarak gerçekleştirilmiş hali aşağıdaki kod parçasında verilmiştir.

case keywordünden sonra parantez içinde kontrol edilecek sinyal yazılır. Daha sonra bu sinyalin muhtemel değerlerine göre yapılacak işlemler "<DEĞER>: **begin** <Statements> **end**"

şeklinde bir syntax ile tanımlanır. Eğer tek bir statement varsa, "begin" ve "end" kullanımı isteğe bağlıdır.

<u>Not:</u> Her iki örnekte de always bloğunun sensitivity listinin "*" olarak belirlenmiş olduğuna dikkat edin. Always bloğu içinde değeri okunan sinyal yalnızca "sum" olduğundan (karşılaştırma yaparken), bu örnekler için "always@*" kod parçası "always@(sum)" kod parçası ile aynı anlama gelmektedir.