FB-CPU RTL TASARIMI

Damla Su KARADOĞAN, Alp Eren GÜRLE, Taha Yasin ÖZTÜRK
Fenerbahçe Üniversitesi
Bilgisayar Mühendisliği
İstanbul, Türkiye
E-mail: { damla.karadogan, alp.gurle, taha.ozturk }@fbu.edu.tr

Özetçe— Bu proje kapsamında FB-CPU isminde bir işlemcinin Verilog dili ile RTL tasarımı ve tasarlanan işlemci üzerinde makine dili ile yazılan çeşitli kod parçacıkları yazılacaktır. Proje sonunda basit bir işlemcideki RAM, Kontrol Ünitesi ve Saklayıcıların bir arada çalışıp, makine dilindeki kod parçacıklarını nasıl yürütebildiği gözlemlenecektir. Kullanılacak Basys3 FPGA geliştirme kartı üzerinde FBCPU demo'su yapılacaktır.

Anahtar Kelimeler — FPGA, CPU

Abstract— In this project, RTL design of a processor named FB-CPU with Verilog language and various code snippets written on the computer with machine language will be written. At the end of the project, it will be observed how RAM, Control Unit and Registers in a simple processor can work together and execute code snippets in the machine language. FBCPU demo will be made on Basys3 FPGA development board to be used.

Keywords — FPGA, CPU.

I. GİRİŞ

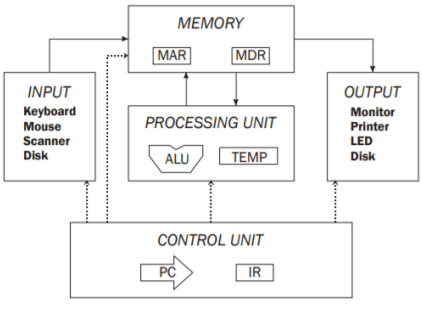
FB-CPU isimli işlemcinin, tasarımda Von Neumann mimarisi baz alınacaktır ve durum makinesi kullanılacaktır. RAM, saklayıcılar, kontrol üniteleri ve ALU birlikte kullanılarak yazılan kod çalıştırılacak ve assembly dilindeki test kodlarıyla test edilecektir.

II. SİSTEM MİMARİSİ

FB-CPU tasarlanırken Xilinx Vivado Design Suite; Verilog, VHDL vb… donanım tasarım dillerini alarak, FPGA'e konfigüre edilebilecek (Xilinx firması FPGA'leri için .bit uzantılı dosyalar) tasarım dosyasını oluşturur.

FB-CPU'nun mimarisini görselleştiren, veri akışının gözlemlenebildiği "FB-CPU Simülatörü" test yazılımlarının nasıl çalıştığını görmemize yardımcı olmuştur.

FB-CPU RTL tasarımı, Von Neumann mimarisindedir.



Şekil 2. Von Neumann Mimarisi

Temel olarak 4 elemanı vardır; saklayıcılar (Şekil 2'de Processing Unit'in altındaki Temp değişkeni), bellek(RAM), İşlem Ünitesi (ALU), Kontrol Ünitesi.

Von Neumann mimarisinde kullanılan ünitelerin görevleri: Bellek, operasyon komutlarını ve değişkenleri tutmaktadır. İşlemci Ünitesi, aritmetik ve mantık işlemlerini yapmaktadır. Kontrol Ünitesi, komutların çözülmesi için gereklidir. Saklayıcılar, çeşitli görevlerde kullanılan saklama alanları.

FB-CPU'nun tasarımda 4 adet saklayıcı bulunmaktadır. Durum; durum makinasında, hangi durumda olduğunu bilgisi tutulur. PC; RAM'deki hangi adresteki komutun çalıştığı bilgisi tutulur. IR; o anda çalışan komutun kendisi tutulur. ACC; geçici saklama alanı.

```
always@(posedge clk) begin
    durum       <= #1 durumNext;
    PC          <= #1 PCNext;
    IR          <= #1 IRNext;
    ACC         <= #1 ACCNext;
end
```

FB-CPU durum makinaları yöntemi ile gerçeklenecektir. Yani bu işlemci durum ismindeki saklayıcının değerine göre $2^3 = 8$ farklı durumda çalışan bir tasarımı olacaktır (İşlemcinin desteklemesi istenen işlemlerin tamamı 8 farklı durumda yapılabilmektedir). Diğer tüm saklayıcılar, durum saklayıcısının değişimine göre çalışacaktır. Yani durumun değerine göre tüm saklayıcıların giriş sinyalleri değişmektedir. Tasarımda giriş çıkış portlarına bağlı olan bellek sinyalleri aşağıda verilmektedir.

- MAR (6 Bit): Memory Address Register isminde bir saklayıcıdır. Bu saklayıcı RAM'in adres girişine bağlanmıştır. RAM'in 2^6 lokasyonu olduğu için MAR 6 bitliktir. Saklayıcı RAM'in içerisindedir.

- MDRIn (10 Bit): Memory Data Register In, RAM'e bir veri yazılacağı zaman kullanılan saklayıcıdır. RAM'in bir lokasyonu 10 bitlik olmasından ötürü, saklayıcı 10 bittir. Saklayıcı RAM'in içerisindedir.

- RAMWr (1 Bit): RAM'e veri yazılacağı durumlarda aktif edilmektedir. 1 olmadığı durumlarda RAM'e veri yazılmaz. Saklayıcı RAM'in içerisindedir.

- MDROut (10 Bit): Memory Data Register, RAM'den veri okunacağı zaman kullanılan saklayıcıdır. RAM'in bir lokasyonu 10 bit olmasından dolayı, saklayıcı 10 bittir. Saklayıcı RAM'in içerisindedir.

Bellek (RAM, Random Access Memory); FB-CPU'nun komutları okuyup, hesaplanan değerleri geri yazacağı bir Block RAM mekanizması bulunmaktadır. Test kodunun instantiate ettiği bellek memory.v dosyasında bulunmaktadır. RAM'e bağlı 4 saklayıcı, clock ve reset sinyali bulunmaktadır. RAM'e bağlı saklayıcıların görevleri saklayıcılar bölümünde açıklanmıştır.

```
input clk;
input rst;
input i_we;
input [SIZE-1:0] i_addr;
input [9:0] i_ram_data_in;
output reg [9:0] o_ram_data_out;
```

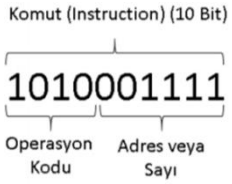
Aynı zamanda memory.v içerisinde bulunan include testCase'ler, tb_fbcpu.v dosyasında parametre ile çağırdığımız test kodunun içeriğini alıp if bloğunun altına getiriyor. Bu şekilde yapmamızın amacı daha sade ve anlaşılır olmasıdır yoksa dosyanın içeriğini direkt kopyalasak da olurdu.

```
initial begin
    if(TEST_CASE == 1) begin
       `include "testCase1.v"
    end else if(TEST_CASE == 2) begin
       `include "testCase2.v"
    end else if(TEST_CASE == 3) begin
       `include "testCase3.v"
    end
end
```

İşlem Ünitesi (ALU, Arithmetic Logic Unit); aritmetik işlemlerin gerçekleştirildiği bölümdür. FB-CPU'da 3 adet aritmetik işlem vardır. Bunlar toplama, çıkartma ve çarpma, gelen operasyon koduna göre işlemleri gerçekleştirip ACC saklayıcısına yazmaktadır.

Kontrol Ünitesi; Saklayıcılar, Aritmetik İşlem Ünitesi ve RAM'e verilerin birbirleri arasında transferinden sorumludur. İşlemci içi veri akışını yönetir.

Şekil 3'te FB-CPU'nun 10 bitlik komutunun, operasyon ve adres için bitlerinin ayrılması gösterilmiştir.



Şekil 3. FB-CPU Örnek Komut Binary Gösterimi

İşlemcimizin tepe modülü olan, tasarımı barındıran fbcpu_core.v modülü vardır. İşlemcinin kendisini tanımlar. memory.v isimli bir başka dosyada da RAM'in kendisini tanımlanır. Bununla birlikte bu iki dosyayı aynı anda test edebilmek ve birbirine bağlamak için tb_fbcpu.v isimli test bench kodu vardır. Bu test bench kodunda fbcpu_core ile block RAM'i bulunduran memory.v instance edilmiştir. Burada ikisinin sinyalleri birbirine bağlanmaktadır.

```
FBCPU #(
    ADDRESS_WIDTH,
    DATA_WIDTH
) FBCPU_Inst(
    .clk(clk),
    .rst(rst),
    .MDRIn(data_toRAM),
    .RAMWr(wrEn),
    .MAR(addr_toRAM),
    .MDROut(data_fromRAM),
    .PC(pCounter)
);

blram #(ADDRESS_WIDTH, 64, TEST_CASE) blram(
  .clk(clk),
  .rst(rst),
  .i_we(wrEn),
  .i_addr(addr_toRAM),
  .i_ram_data_in(data_toRAM),
  .o_ram_data_out(data_fromRAM)
);
```

Aynı zamanda test bench kodunun içinde TEST_CASE isminde bir paramatremiz var. Bu test case hocanın bize sunmuş olduğu 3 adet kodu ifade ediyor. İşlemcinin okuyabilmesi için belleklere yazılmış olan programı test etmemizi sağlayan test dosyalarıdır.

```
parameter TEST_CASE = 1;
```

```
memory[0] = 10'b0000_110010; // LOD 50, (ACC = *50), Hex = 32
memory[1] = 10'b0010_110011; // ADD 51, ACC = ACC + (*51), Hex = B3
memory[2] = 10'b0001_110100; // STO 52, (*52) = ACC, Hex = 74
memory[3] = 10'b1001_000000; // Halt, Hex = 240
memory[50] = 10'b0000_000101; // Hex = 5
memory[51] = 10'b0000_001010; // Hex = A
```

Örneğin, test bench kodundaki parametre ile TEST_CASE 1'i çağırırsak kodu çalıştırdığımızda assembly yani makine diliyle yazılmış bu dosya çağırılmış olur. Bu dosyalardan hangisinin o esnada test edileceğini parametreye test casenin değerini vererek söylüyoruz.

Test bench'imizin içinde aynı zamanda parametreden girdiğimiz değere göre gerekli değerleri atayan bir if'li seçim mekanizması bulunmaktadır. Örneğin test case 1'i denemek istersek işlemci çalışana kadar 10.000 cycle beklememiz gerekmektedir.

```
initial begin
  rst = 1;
  repeat (10) @(posedge clk);
  rst <= #1 0;
  repeat (10000) @(posedge clk);

  if(TEST_CASE == 1)
    memCheck(52,15);

  else if(TEST_CASE == 2)
    memCheck(52,50);

  else if(TEST_CASE == 3)
    memCheck(52,50);

  $finish;
end
```

10.000 cycle'ın sonunda 52. Adreste 15 olması gerekir. A ile 5'i toplanır. Diğer caselerde de farklı işlemlere aynı yaklaşım uygulanır.

```
task memCheck;
    input [31:0] memLocation, expectedValue;
    begin
      if(blram.memory[memLocation] !== expectedValue) begin
            $display("Test Hatali Tamamlandi!");
      end else begin
            $display("Test Basarili Tamamlandi!");
      end
    end
endtask
```

Bunların doğru olup olmadığını kontrol eden otomatik bir yapı var. Simülasyonu başlattığımız zaman ekrana, testin hatalı tamamlanıp tamamlanmadığını gösteren bir çıktı veriyor.

Fbucpu_core.v'de önce saklayıcıları tanımladık sonra da always@(posedge clk) bloğunun içerisinde saklayıcılarımızın atamalarını gerçekleştirdik. #1 diyerek de bir cycle sonra atamanın gerçekleşmesini sağladık. Yani yükselen kenar gelir gelmez değil de gelmesinden bir nano saniye sonra çıktı vermeye başlayacak şekilde tasarlandı.

```
reg [DATA_WIDTH - 1:0] IR, IRNext;
reg [5:0] PCNext;
reg [9:0] ACC, ACCNext;
reg [2:0] durum, durumNext;

always@(posedge clk) begin
    durum       <= #1 durumNext;
    PC          <= #1 PCNext;
    IR          <= #1 IRNext;
    ACC         <= #1 ACCNext;
end
```

Tasarımdaki bir diğer önemli nokta ise RAM'e giden sinyalleri ayrıca bir saklayıcıdan geçirmiyoruz. RAM'e giderken zaten içinde saklayıcı var. 2 defa saklayıcıdan geçirmiş gibi oluyoruz bu yüzden de geri veri gelirken 2 cycle beklemek zorunda kalıyoruz. Bu yüzden alt ifadelerde bunların nextli olan ifadeleri değil direkt kendileri kullanılıyor. Yani bunlar kombinasyonel devre olarak dışarı çıkıyorlar.

```
output reg [DATA_WIDTH-1:0] MDRIn,
output reg RAMWr,
output reg [ADDRESS_WIDTH-1:0] MAR,
```

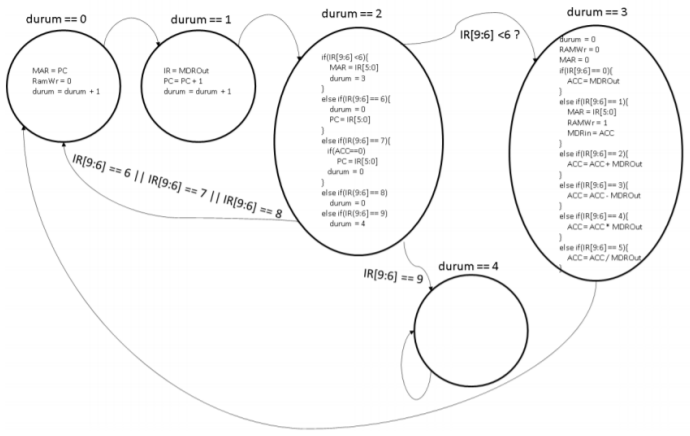
İlk reset anında bu durumlarla beraber beklemede. Rst biter bitmez case'de durum saklayıcısının değerine bakacak şekilde aktifleşir.

```
if(rst) begin
    durumNext   = 0;
    PCNext      = 0;
    MAR         = 0;
    RAMWr       = 0;
    IRNext      = 0;
    ACCNext     = 0;
    MDRIn       = 0;
end else begin
```

III. KULLANILAN YAZILIM

Tasarım, bir durum makinası şeklinde kodlandı. Case duruma bakıyor ve durumun içerisindeki yapılara göre sürekli ilerliyor. FB-CPU'nun durum diyagramı olarak ifade edilmiş hali Şekil 5'te verilmektedir. İşlemcinin adım adım yapması gereken işler bir arada gösterilmektedir.

Durum saklayıcısı kullanmamızın sebebi; sistemde çalışacak mekanizmanın ne zaman çalışacağını belirten saklayıcı bu adımlara göre çalışır.



Şekil 5. FB-CPU Durum Makinası Gösterimi

- Durum 0'da, Program Counter (PC) isimli saklayıcımız, bellekteki hangi komutu çalıştıracaksak onu tuttuğumuz saklayıcının değerini yani 0'ı, MAR'ın üstüne attı. Bir yazma işlemi yapmadığımız için yazma sinyalimizi RamWr'ı 0'a çektik ve durumu 1 arttırarak 1 cycle sonra durum 1'e atladık. Yani burada PC'nin değerini MAR'a verdik böylelikle RAM'den 0. adresin içeriğini istemiş olduk durumu değiştirdiğimiz zaman RAM bize kendi data outputundan (MDROut'tan) 0. adresin içeriğini döndürecek.

```
0: begin
    MAR = PC;
    RAMWr = 0;
    durumNext = durum + 1;
end
```

- Durum 1'e geçtiğimizde Memory Data Out'tan, gelen instruction bilgisi unutulmaması için Instruction Register (IR)'a yazılır. İşlemler sonrası tekrar durum 0'a döneceğimiz için durum 0'da iken daha önce okuduğumuz 0. adres değil de 1. adresi okumamız lazım. Bunun için PC'yi 1 arttırdık. Bu işlemi yaptıktan sonra yeni işlemi yapmak için durumumuzu da 1 arttırıyoruz ve durum 2'ye geçiyoruz.

```
1: begin
    IRNext = MDROut;
    PCNext = PC + 1;
    durumNext = durum + 1;
end
```

- Durum 2'de bir önceki adımda IR'ye atadığımız saklayıcının 9'dan 6. biti yani soldan ilk 4 biti, 6'dan küçük mü diye bakıyoruz. Bakma sebebimiz 6'dan küçük olan işlemlerinin hepsinin RAM'den bir adres okuma ihtiyacı(SUB, ADD vs.) olmasıdır. Bir tek STO'da içerik bilmeye ihtiyacımız yoktur. İlk 6 operasyonda içeriği okuma ihtiyacımız olduğu için hepsinin içeriğini okuma ve sonrasında durum 3'e götürme ve kalan işleri durum 3'de bitirme mantığıyla işlemci tasarlanmıştır. 9'dan 6'ya olan bitler 6'dan küçükse, adrese ve adresin içeriğine ihtiyacımız olacağı için adres IR'nin ilk 6 bitinde tutuluyor. Bunu RAM'in MAR'ına yazdık. Yani RAM'den tekrar

bir adres istiyoruz ama bu sefer istediğimiz adres, oradaki komutun yanında yazılan adres. Bu işlemlerden sonra durumu 3'e götürdük. Yani ilk 6 işlemden(operasyondan) birisiyse, adres içeriğini tekrar okumamız gerektiği için RAM'den o bilgiyi istedik. Yani durum değişimimiz ilk 6 bite bağlı. Makine kodumuzun içindeki soldan ilk 4 biti 6'dan küçük geldiyse durumu 3'e atlattık.

```
2: begin
    if(IR[9:6] < 6) begin
        MAR = IR[5:0];
        durumNext = 3;
    end else if(IR[9:6] ==  6) begin
        PCNext = IR[5:0];
        durumNext = 0;
    end else if(IR[9:6] == 7) begin
        if(ACC == 0) begin
            PCNext = IR[5:0];
        end
        durumNext = 0;
    end else if(IR[9:6] == 8) begin
        durumNext = 0;
    end else if(IR[9:6] == 9) begin
        durumNext = 4;
    end
end
```

Eğer gelen değer 6 ise, PC değerini IR'nin 5'ten 0'a olan değerine karşılık gelen sayıya eşitliyoruz. Durumu da tekrar 0'a döndürüyoruz. Yani aslında JMP komutu geldi. PC'nin yeni değerini yazdık, durumu tekrar 0'a götürdüğümüz zaman artık PC değeri güncellenmiş olacağı için bizim RAM'den okuyacağımız işlem artık RAM'den çekeceğimiz kodun yeri, güncellenmiş yerden olacak ve biz JMP işlemi yapmış olacağız. 7. komutun (operasyonun) özelliği koşullu JMP işlemidir. ACC değeri 0 ise PC değerini, IR'nin ilk 6 bitinin değerini atıyoruz ve durum 0'a gidiyoruz; değilse de PC değerini değiştirmiyoruz, hiç bir işlem yapmadan durumu 0'a gönderiyoruz. (ACC'ın 0 ya da 1 olmasından bağımsız olarak durum 0'a gidiyor) Eğer komut 8 geldiyse NOP komutu yani "no operation". Hiçbir işlem yapılmaz, durumu sadece direkt 0'a döndürüyoruz. Operasyon kodu 9 gelirse HLT yapılır. Yani kod durur. Durması için durum 4'e atlatıyoruz.

- Durum 3'ü, ilk 6 operasyonu RAM'den okuduktan sonra işlem yapmak için kullanıyoruz. Durum 3'e geçtiğimizde doğrudan gelecek durumumuzu 0'a götürüyoruz yani yaptığımız işlemlerden bir cycle sonra direkt durum 0'a gidiyoruz. Yukarıda RAMWr ve MAR'ı 0'a eşitledik ama alt taraftaki işlemlerde değerlerini güncelleyebiliriz. IR 9'dan 6'ya, 0'a eşitse LOD işlemi yaptığımız anlamına gelir. LOD, o adresin içeriğini alıp ACC içine atıyor. Biz aslında o durumun içeriğini durum 2'de MAR'a vererek istedik ve bu bize bir cycle sonra MDROut'tan geliyor. Bunu ACC'a atıyoruz. Operasyon kodu 1 olduğunda STO komutu gelmiştir. Bu komutla ACC'nin içerisindeki değeri, belleğin içindeki değere atamamız lazım. Onu yapmak için de hangi adrese değeri yazacaksak, adres içeriğini söylediğimiz yer MAR'dır. MAR'a, IR 5'ten 0'ına söylediğimiz yerin adresini verdik. RAMWr'ı 1 yaptık. Aynı zamanda da RAM'in data içeriğini yüklüyoruz. MDRIn'ine de ACC değerini verdik. Burada ACC içeriğini Memory'nin Data portundan verdik WriteIn'i 1 yaptık aynı anda adresi de STO işleminden gelen adresi vermiş olduk. STO'nun tamamı 10 bitlik bir sayı, onu da IR saklayıcısında tutuyorduk. IR saklayıcısının 5'den 0'ını verdiğimiz zaman 6 bitlik o adresi MAR'a vermiş olduk.

```
3: begin
    durumNext = 0;
    RAMWr = 0;
    MAR = 0;
    if (IR[9:6]==0)begin
        ACCNext=MDROut;
    end else if(IR[9:6]==1) begin
        MAR=IR[5:0];
        RAMWr=1;
        MDRIn=ACC;
    end else if (IR[9:6]==2) begin
        ACCNext=ACC+MDROut;
    end else if (IR[9:6]==3) begin
        ACCNext=ACC-MDROut;
    end else if (IR[9:6]==4) begin
        ACCNext=ACC*MDROut;
    end else if (IR[9:6]==5) begin
    end
end
```

Operasyon kodu 2 olduğunda ADD işlemi yapılır. Bir cycle önce adres içeriğini okuma işlemini yapmıştık. Okuduğumuz adres, Memory data read register outtan geldi, ACC ile toplayıp tekrar ACC yazdık. Operasyon kodu 3 olduğunda SUB yani çıkartma işemi geldiğinde ADD işlemindeki adımları çıkartma işlemine uyarlamış olduk. Yani Memory data read register outtan gelen değeri ACC ile çıkartıp tekrar ACC'a yazdık. Operasyon kodu 4 olduğunda MUL komutuyla çarpma işlemi gerçekleştirilir. MDROuttan gelen değer ACC ile çarpılıp tekrar ACC'ın üstüne yazılır. Operasyon kodu 5 olduğunda DIV komutuyla bölme işlemi yapılması istenir ama bizim düzeyimizin üstünde bir işlem karmaşıklığı içerdiği için biz operasyona kodu 5 geldiğinde hiçbir işlem yaptırmadan durumu 0'a göndereceğiz.

- Durum 4'te hiçbir şey kontrol edilmiyor, hiçbir şey yapılmaz sadece üzerinde döner. Bir çıkış komutu da yoktur. HLT işlemi geldiğinde işlemcinin gücü kesilip yeniden verilmediği takdirde buradan yani durum 4'ten çıkması mümkün olmayacaktır.

```
4: begin
end
```

Test Yazılımı 1:

```
memory[0] = 10'b0000_110010; // LOD 50, (ACC = *50), Hex = 32
memory[1] = 10'b0010_110011; // ADD 51, ACC = ACC + (*51), Hex = B3
memory[2] = 10'b0001_110100; // STO 52, (*52) = ACC, Hex = 74
memory[3] = 10'b1001_000000; // Halt, Hex = 240
memory[50] = 10'b0000_000101; // Hex = 5
memory[51] = 10'b0000_001010; // Hex = A
```

FB-CPU için bellekte 50 adrese hexadecimal karşılığı 5 olan sayıyı ACC saklayıcısının içine LOD komutuyla beraber yüklenir. ADD komutuyla ACC saklayıcımızın içinde bulunan 50. adresin değeri, 51 adresteki hexadecimal karşılığı A olan ifade toplanır. STO komutuyla ACC saklayıcısının değer (50 ve 51. adreslerin toplamı F'dir) 52. adrese kayıt edilmiş olur. HLT komutuyla da test durdurulmuş olur.

memory[0:63]	032,0b3,074,
[0][9:0]	032
[1][9:0]	0b3
[2][9:0]	074
[3][9:0]	240

[50][9:0]	005
[51][9:0]	00a
[52][9:0]	00f
[53][9:0]	XXX

Test Yazılımı 2:

```
memory[0] = 10'b0000_110010; // LOD 50, (ACC = *50), Hex = 32
memory[1] = 10'b0100_110011; // MUL 51, ACC = ACC * (*51), Hex = 133
memory[2] = 10'b0001_110100; // STO 52, (*52) = ACC, Hex = 74
memory[3] = 10'b1001_000000; // Halt, Hex = 240
memory[50] = 10'b0000_000101; // Hex = 5
memory[51] = 10'b0000_001010; // Hex = A
```

FB-CPU için bellekte 50 adrese hexadecimal karşılığı 5 olan sayıyı ACC saklayıcısının içine LOD komutuyla beraber yüklenir. MUL komutuyla ACC saklayıcımızın içinde bulunan 50. adresin değeri, 51 adresteki hexadecimal karşılığı A olan ifade çarpılır. STO komutuyla ACC saklayıcısının değer (50 ve 51. adreslerin çarpımı 32'dir) 52. adrese kayıt edilmiş olur. HLT komutuyla da test durdurulmuştur.

memory[0:6	032,133,074,240,
[0][9:0]	032
[1][9:0]	133
[2][9:0]	074
[3][9:0]	240

[50][9:0]	005
[51][9:0]	00a
[52][9:0]	032

Test Yazılımı 3:

```
memory[0]= 10'b0000_110011; // LOD 51, ACC = *51, Hex = 33
memory[1]= 10'b0011_110001; // SUB 49, ACC = ACC - *49, Hex = F1
memory[2]= 10'b0111_001010; // JMZ 10, döngü bittiyse, döngüden çıkartacaktır (ACC-49 == 0), 10. Satır, Hex = 1CA
memory[3]= 10'b0000_110000; // LOD 48, temp değerini yükle, başlangıçta 0, Hex = 30
memory[4]= 10'b0010_110010; // ADD 50, ikinci sayıyı ACC'nin üstüne ekle, Hex = B2
memory[5]= 10'b0001_110000; // STO 48, ACC'nin değerini temp'e ata, Hex = 70
memory[6]= 10'b0000_110001; // LOD 49, ACC = i, Hex = 31
memory[7]= 10'b0010_101110; // ADD 46, ACC = i + 1, Hex = AE
memory[8]= 10'b0001_110001; // STO 49, i = i + 1, Hex = 71
memory[9]= 10'b0110_000000; // JMP 0, döngünün başına dön 0. satır, Hex = 180
memory[10]= 10'b0000_110000; // LOD 48, ACC = temp, Hex = 30
memory[11]= 10'b0001_110100; // STO 52, *52 = ACC, Hex = 74
memory[12]= 10'b1001_000000;// HLT, bitirme, Hex = 240

memory[46]= 10'b1; // 1 sayısı
memory[48]= 10'b0; // Hex = 0, temp
memory[49]= 10'b0; // Hex = 0, i index'i için
memory[50]= 10'b0000000101; // Hex = 5
memory[51]= 10'b0000001010; // Hex = A
```

FB-CPU için bizden yapılması istenen işlem çarpma işlemidir ama bunu MUL komutuyla değil döngülerle yapmamız gerekmektedir. Bellekte 51. adresteki değer LOD komutuyla ACC'a kayıt edilir. SUB komutuyla ACC içinde olan değerden 49 çıkartılır ve tekrar ACC saklayıcısının içine kaydedilir. JMZ komutu ile bir önceki basamakta yaptığımız işlemin (ACC-49) sonucu 0'a eşit mi değil mi diye kontrol edilir. 0'a eşit ise döngüden çıkar ve 10. adresteki işlemi gerçekleştirmeye gider. Yani 48. adresteki temp'in değerini ACC saklayıcısın içine yükler, STO ile ACC içindeki değeri 52. adrese kayıt eder ve sonra da HLT ile kodu bitirir. Bu şekilde döngülerle, MUL komutu kullanmadan sonuç belleğe kayıt edilir. Ama JMZ 0'a eşit değilse ACC-49, 0 oluncaya kadar dönmeye devam eder. Kontrolden sonra 48. adrese temp değerini LOD ile yükler. 50. adresteki değeri ACC'ın üstüne ADD ile ekler. 48. adrese gider ve ACC'nin değerini STO temp'e kayıt eder. 49. adresteki değeri ACC'nin içine atar ve 46. adresteki değer ile toplar ve 49. adrese tekrar kayıt eder. JMP komutuyla tekrar 0. Satıra yani döngünün başına döner.

[48][9:0]	032
[49][9:0]	00a
[50][9:0]	005
[51][9:0]	00a
[52][9:0]	032

memory[0:63][9:0]	033,0f1,1ca,030,0b
[0][9:0]	033
[1][9:0]	0f1
[2][9:0]	1ca
[3][9:0]	030
[4][9:0]	0b2
[5][9:0]	070
[6][9:0]	031
[7][9:0]	0ae
[8][9:0]	071
[9][9:0]	180
[10][9:0]	030
[11][9:0]	074
[12][9:0]	240

IV. SONUÇLAR

Geliştirilen işlemcinin desteklediği işlemler, FB-CPU ISA (Instruction Set Architecture) tablosunda verilmiştir. FB işlemcisini verilog dilinde ifade ederken ALU yani aritmetik işlem ünitesi aritmetik işlemlerin gerçekleştirildiği bölümdür. FB-CPU'da 3 adet aritmetik işlem vardır. Bunlar toplama, çıkarma ve çarpmadır. Gelen operasyon koduna göre işlemler yapılıp ACC saklayıcısına yazılır.

Test yazılımı(test_case)1, test yazılımı 2 ve test yazılımı 3 tek tek denenerek, Vivado üzerindeki simülasyon üzerinden incelenmiştir. Algoritmik düşünce yeteneğimiz gelişmiş ve verilog diline olan hakimiyetimiz artmıştır.

Tablo 1. FB-CPU ISA (Instruction Set Architecture)

Komut Adı	Görevi	Operasyon Kodu
LOD ADDR	Yükleme (Load), Bellekteki verilen adresin içerisinden değeri alıp, ACC saklayıcısına yerleştirir. ACC = *(ADDR)	0000
STO ADDR	Kaydetme (Store), ACC'nin içerisindeki değeri alıp, bellekte verilen adrese yazar. *(ADDR) = ACC	0001
ADD ADDR	Bellekteki verilen adresteki değeri alır, ACC ile toplayıp, ACC'nin üzerine yazar. ACC = ACC +*(ADDR)	0010
SUB ADDR	Bellekteki verilen adresteki değeri alır, ACC ile çıkartıp, ACC'nin üzerine yazar. ACC = ACC - *(ADDR)	0011
MUL ADDR	Bellekteki verilen adresteki değeri alır, ACC ile çarpıp, ACC'nin üzerine yazar. ACC = ACC * (*(ADDR))	0100
JMP SAYI	PC = Sayı olur.	0110
JMZ SAYI	ACC'ın değeri 0 ise, verilen sayı değerini PC'e atar, değilse işlem yapmaz.	0111
NOP	No Operation, hiçbir işlem yapılmaz.	1000
HLT	Uygulama durur	1001

PROJE EKİBİ

Damla Su KARADOĞAN, 11.02.2001 yılında doğdu. 2019 yılında Özel Envar Anadolu Lisesinden mezun oldu. Şu anda Fenerbahçe Üniversitesinde Endüstir Mühendisliği bölümünde lisans eğitimini almakta ve Bilgisayar Mühendisliğinde ÇAP eğitimi alıyor.Öğrenci numarası, 190302016.

Alp Eren Gürle, 13.10.2000 yılında doğdu. 2018 yılında Denizli Uğur Anadolu Lisesinden mezun oldu. Şu anda Fenerbahçe Üniversitesinde Bilgisayar Mühendisliği bölümünde lisans eğitimi almakta. Öğrenci numarası, 190301028.

Taha Yasin Öztürk, 20.11.2000 yılında doğdu. 2019 yılında Erbakır Fen Lisesinden mezun oldu. Şu anda Fenerbahçe Üniversitesinde Bilgisayar Mühendisliği bölümünde lisans eğitimi almakta. Öğrenci numarası, 190301027.

REFERANS DOSYALAR

Sunu: FB_CPU_RTL_TASARIMI.pptx

CV: cvmst.pdf

Github: https://github.com/damlasu/FBCPU

Youtube: https://www.youtube.com/watch?v=_GoEawF9GOA

KAYNAKLAR

Kaynakça

[1]*Levent, V. E. (2021, Ocak 7). Mantıksal Sistem Tasarımı. levent.tc: http://www.levent.tc/files/courses/digital_design/project/BLM201_proje_spesifikasyonlari.pdf adresinden alındı*