RISC-V TABANLI İŞLEMCİ 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 başlangıç tasarım verilen bir RISC-V işlemcisinin ALU ve instruction decoder blokları temel SystemVerilog dili özellikleri kullanılarak tasarım ve doğrulama çalışmaları yapılacaktır.

Anahtar Kelimeler — RISC-V, SystemVerilog, CPU

Abstract— In this project, basic SystemVerilog language reviews and test tests will be carried out on the ALU and instruction decoder blocks of a RISC-V processor given the initial design.

Keywords — RISC-V, SystemVerilog, CPU.

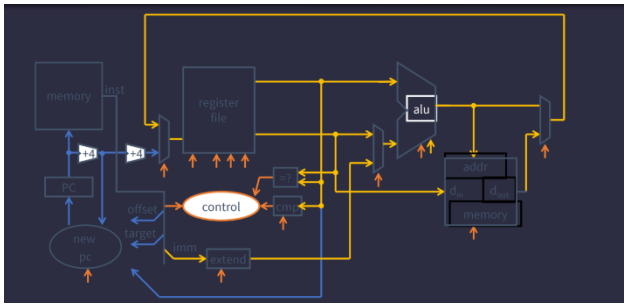
I. Giriş

Tasarlanılan işlemci birçok bölümden oluşmaktadır. Bunun nedeni; tasarım büyüdükçe tasarım kolaylığı/ okunabilirliği/ yönetilebilirliği zorlaşacağı için bölümlere bölerek bu zorluğun önüne geçilir. Test kodunda daha önceden hazırlanmış olan bir CPU test uygulamasının makine diline döndürülmüş halini, işlemciye besleyerek sonucunu kontrol eden bir uygulama bulunmaktadır. Amacımız; ALU ve instruction decoder bloklarının tasarımını SystemVerilog dili ile kodlamak ve doğrulama çalışmalarını yapmak.

II. Sistem Mimarisi

RISC-V tabanlı işlemci tasarlanırken kullanılan 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.

RISC-V, RISC prensiplerini kullanan açık kaynak bir Komut Seti Mimarisidir(ISA). University of California Berkeley'in oluşturduğu ve herkese açık (kişisel, ticari, akademik…) bir ISA olan RISC-V, herhangi bir lisans parası ödemeden herkesin ortak kabul ettiği bir mimaride işlemci üretebilmeyi sağlar.



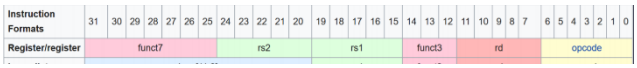
Risc-v, Harvard mimarisi ile geliştirildi. Bu mimarinin avantajı; iki adet bellek bulunmasıdır. Bir tanesinde işlemci hangi komutları destekliyorsa o komutları (and, xor, or…) tutar; bir diğerinde ise dataları (sayıları vb) tutar. Bu sayede aynı anda iki farklı belleğe erişebilinir. Paralel çalışması, daha hızlı çalışmasını sağlar. PC mekanizması, bellekteki hangi adresteki komutu çalıştırdığını ifade eder. PC o anki değerine göre bellekten instruction çekip, işlemler yapıp geriye bir şeyler yazarız. Register File mekanizmasında, işlemcinin içerisindeki çeşitli saklayıcılar bulundurulur. Alu'da, temel aritmetik işlemler(toplama, çıkarma…) yapılır ve sonrasında belleğe yazar. Kontrol mekanizması, karşılaştırma yapar. Genel olarak Risc-v işlemcisi böyle çalışır.

Risc-v belleklerinde 32 bitlik data in girişi ve 32 bitlik data out çıkışları vardır ve bu bellekler 32 bit adres genişliğine sahiplerdir. Memory kontrol ve enable isimli 2 girişi daha var. Çalışma mekanizması; belleğe memory kontrol portundan verilen girişe göre bellek ya bir şeyler yazar ya da bir şeyler okur. Örneğin memory kontrolden okuma komutu geldiyse gelen adresteki sayı, çıkıştan gönderilir ve okuma işlemi gerçekleşmiş olur.

Bir komut/ instruction bu sırayla çalışır:

1. Fetch: Instruction'ı bellekten yakalar.
2. Decode: Instruction'nın çözülmesi/ işlemin ne olduğuna karar verir.
3. Execute: İstenilen görevin yapılması.
4. Memory Acces: Belleğe erişim yapılır. Bilgi yazılacaksa yazılır.
5. Write Back: Saklayıcıların değeri güncellenecekse onlar ve PC değeri güncellenir.

Bütün Risc-v komutları 32 bit uzunluğundadır.



- Opcode: Aritmetik mi, bellek işlemi mi, kontrol işlemi mi… Hangi işlem olduğu bilgisi tutulur. Örneğin; aritmetik işlemler için R-type (2 tane kaynak saklayıcı ve 1 tane geri yazdığımız header saklayıcı var.), I-type(1 tane kaynak saklayıcı) ve U-type(kaynak saklayıcı bulunmaz, direkt yazılır.) typeları vardır. 7 bittir.

- Func7+func3: Toplam 10 bittir. Bu 10 bitlik sayının içinde işlemin toplama mı, çıkartma mı vb olduğu bilgisi bulunur.
- Rs1,2: İlk ve ikinci operand kayıtlarıdır. 5 bittirler.
- Rd: Hesaplama sonucu, hedef kayıdı gösterir.

Projede decoder modülünde 32 bitlik girişi parçalayıp; Alu modülünde de istenen aritmetik işlemler yapılacaktır. Genel olarak işlemciye göz atacak olursak;

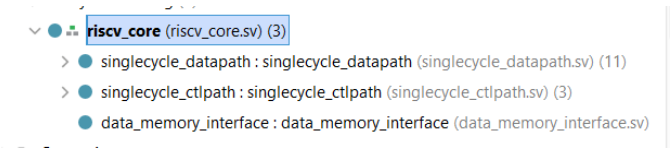
```
module riscv_core (
    input  clock,
    input  reset,

    output [31:0] bus_address,
    input  [31:0] bus_read_data,
    output [31:0] bus_write_data,
    output [3:0]  bus_byte_enable,
    output        bus_read_enable,
    output        bus_write_enable,

    input  [31:0] inst,
    output [31:0] pc
);
```

Tepe modülümüz "riscv_core"da, clk ve reset girişleri var. Tasarımını yaptığımız bir işlemci olduğu için, işlemci komutlarını RAM üzerinden alıp RAM'e geri yazıyor. Bu işlemi yapabilmesi için; RAM'in adres girişini okuyan port, RAM'e bir şey yazmak istediğimiz zaman 32 bitlik çıkış çıkıyor, "bus_byte_enable" ile 32 bitlik sayının hangi bytelarının yazılacağını, hangilerinin yazılmayacağına karar veriyoruz. "bus_byte_enable" ile adresi vermiştik; o adresten bir şey okumak istediğimiz zaman "bus_read_enable" aktif ediyoruz. Yazmak istediğimiz zaman da "bus_write_enable" aktif ediyoruz. "bus_write_enable" aktif ederken "bus_byte_enable"ın ilgili hangi byte'lara yazmak istiyorsak onu aktif ediyoruz yani içine bir şeyler yazıyoruz. "bus_write_data" o esnada hangi içerik yazılacaksa RAM'e onları verdiğimiz bir port. Yani işlemci temelde RAM'e bir şeyler yazıp bir şeyler okuyan bir mekanizma.

Testbench için kullanılan, "inst" girişi ile şu an işlemcinin hangi komutu işlediği; "pc" çıkışı ile program counterın'ın hangi numaralı komutu işlediği gösterilir.



Tepe modülünün 3 tane alt modülü var.

1. data_memory_interface: Bellek işlemlerini yapar. RAM'in, RAM ile olan ilişkisini kuran bir modül. Çeşitli operasyonlar yapar.

2. singlecycle_ctlpath: Alu'nun ve işlemcinin o anda ne yapacağını karar veren ünitedir. Başka bir deyişle kontrol ünitesidir. dataPath'in ne zaman, nerede çalıştırılacağından bu ünite sorumludur.

3. singlecycle_datapath: İçerisinde işlemlerin gerçekten yapıldığı, işlem kabiliyetine sahip olan modülleri barındıran ünite. Örneğin; aritmetik işlemler burada gerçekleşir.

"singlecycle_datapath"te bir çok alt modül bulunmakta. Örneğin bunlardan bir tanesi;

"adder_pc_plus_4" modülü. Bu modül normal bir toplayıcı devresi. operand_a ve operand_b sayıları geliyor, toplanıyor ve dışarı çıkartılıyor.

```
`include "config.sv"
`include "constants.sv"

module adder #(
    parameter  WIDTH = 32
) (
    input  [WIDTH-1:0] operand_a,
    input  [WIDTH-1:0] operand_b,
    output [WIDTH-1:0] result
);

    assign result = operand_a + operand_b;

endmodule
```

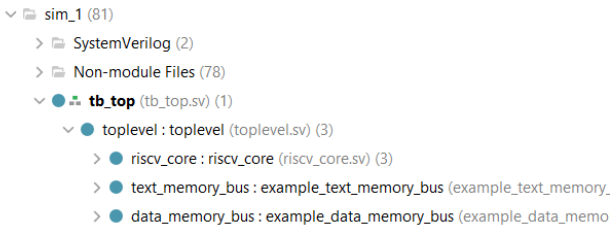
Başka bir örnek daha vermek gerekirse;

```
`include "config.sv"
`include "constants.sv"

module regfile (
    input  clock,
    input  write_enable,
    input  [4:0] rd_address,
    input  [4:0] rs1_address,
    input  [4:0] rs2_address,
    input  [31:0] rd_data,
    output [31:0] rs1_data,
    output [31:0] rs2_data
);
```

"regfile" modülü. Bizim ürettiğimiz sonuçlar bu modülde flip-floplara(saklayıcılara) atanıyor. Bu modülde 32 bitlik 32 tane register grubu var. write_enable'ı registerların içine bir şeyler yazmak istediğimizde kullanıyoruz. Hangi registerlara yazmak istediğimizi bu 2'lü portlardan(datasını) belirtiyoruz. Registerlardan okumak istenildiği zaman da rs1 ve rs2'lerden okunuyor.

Simülasyon dosyalarındaki sim_1'in altında "tb_top" isimli bir modül bulunmakta.



Bu modül; RISC-V çekirdeğini kapsayan "toplevel" isimli bir modülü içermekte. Aslında biz riscv_core'un tasarımını yapıyoruz ama onu da kapsayan toplevel isminde bir modül var. "toplevel" içerisinde; "data_memory_bus" ve "text_memory_bus" isminde iki tane memory var. İkisi de büyük memorylerdir. "tb_top"tan daha önceden hazırlanmış olan bir programı cycle cycle bu modüle besliyoruz. Daha önceden hazırlanmış olan program şöyle çekiliyor; başlangıç olarak text_memory_bus içinde text_hex ve data_memory_bus içinde data_hex isminde iki tane dosya var.

```
`ifdef DATA_HEX
    initial $readmemh(`DATA_HEX, mem);
`endif


`ifdef TEXT_HEX
    initial $readmemh(`TEXT_HEX, mem);
`endif
```

Simülasyon başladığı zaman zaten bellek, işlemcinin okuduğu bellekte program yüklü olarak başlıyor ve cycle cycle program counter değiştikçe farklı adreslerden farklı içerikler okuyarak gidiyor.

```
repeat (100000) begin
    @(posedge clk);

    $display("PC: %h, Inst: %h, Addr: %h, Rd-Dt: %h, Rd-En %d, Wr-Dt: %h, WrE

    if (bus_write_enable && bus_address == 32'hfffffff0) begin
        if (bus_write_data !== 0) begin
            $display("Pass");
            $finish;
        end else begin
            $display("Fail");
            $finish;
        end
    end
end

$display("Timeout - Fail");
$finish;
```

Risc-v instruction setinde yazılmış bir program bittiği zaman en sonunda bus adresin buraya yani "tb_top"a çekilmesi gerekiyor. Burada bus_write_enable geldiyse ve bus_adress 32'hfffffff0 ise programın doğru bitip bitmediğini bir koşul ile kontrol ederiz. Koşul doğru olduğu zaman "Pass" çıktısı verir. Doğru değil ise "Fail" çıktısı verir. Eğer 10.000 cycle'dan fazla şekilde işlemci işini bitiremediyse bu seferde "Timeout-Fail" çıktısı vererek işini/ çalışmayı bitirir.

III. KULLANILAN YAZILIM

Riscv_core modülünün alt modülü olan singlecycle_datapath'in alt modülü ve tamamlamamız istenen modüller "alu" ve "instruction_decoder" idi.

Decoder RAM'den çekilen instructionı parçalayıp ne iş yapılacağını modülün geri kalanlarına söyler. Bu bilgi olmadan Alu'ya bir bilgi transfer edilmeyeceği için Alu'ya hiçbir zaman anlamlı bir şey gitmeyecektir. Bu yüzden önce instruction_decoder tasarımına başladık.

```
4  module instruction_decoder(
5      input [31:0] inst,
6      output [6:0] inst_opcode,
7      output [2:0] inst_funct3,
8      output [6:0] inst_funct7,
9      output [4:0] inst_rd,
10     output [4:0] inst_rs1,
11     output [4:0] inst_rs2
12 );
```

32 bitlik instruction'ın girişinin, instruction'un parse edilmiş hali, yani decode edilmiş hali çıkış olarak verilmektedir. Buna göre instruction sinyalini parçalayarak ilgili sinyallerin üzerlerine atama yapılmıştır. 6 farklı çıkış bulunmaktadır. İlk 7 bit; operasyon kodudur ve ne işlem yapılacağı bilgisini taşır. 3 bitlik function3 ve 7 bitlik function7 aritmetik işlemin ne olduğu bilgisini taşır. 5'er bitlik Rs1 ve Rs2 saklayıcılarında gelen inputlarla işlemler yapılmak ve sonuç 5 bitlik Rd'nin içine yazılmak üzere bölümlere ayrılır.

```
13     logic [6:0] op;
14     logic [2:0] f3;
15     logic [6:0] f7;
16     logic[4:0] RD;
17     logic [4:0] RS1;
18     logic [4:0] RS2;
```

Outputları direkt olarak always_comb bloğunda kullanamadığımız için aynı bit genişliğine sahip geçici değer tutan logicler tanımladık.

Tasarımın bu modülünde clk almadığı için kombinasyonel devre şeklinde tasarlanmıştır. SystemVerilog'ta kombinasyonel devreyi ifade etmek için "always_comb" bloğu kullanılır. Bu bloğun içinde 32 bitlik komutu logiclere dağıttık.

```
20     always_comb begin
21         op = inst[6:0];
22         RD= inst[11:7];
23         f3 = inst[14:12];
24         RS1 = inst[19:15];
25         RS2 = inst[24:20];
26         f7= inst[31:25];
27     end
```

Bu bloğun dışında da üzerine geçici atama yaptığımız logicleri çıkış sinyallerimize yeniden atamasını gerçekleştirdik.

```
28     assign inst_opcode = op;
29     assign inst_rd = RD;
30     assign inst_funct3 = f3;
31     assign inst_rs1 = RS1;
32     assign inst_rs2 = RS2;
33     assign inst_funct7 = f7;
```

32 bitlik komut parçalandıktan sonra hangi işlem yapılacağı, hangi sayılarla işlem yapılacağı gibi bilgilere Alu'ya geldi.

```
4  module alu (
5      input        [4:0]  alu_function,
6      input signed [31:0] operand_a,
7      input signed [31:0] operand_b,
8      output logic [31:0] result,
9      output              result_equal_zero
10 );
```

"alu_function" alu'nun o esnada ne yapacağını söylediği girdi gelir yani hangi işi yapacağımızı buradan öğreniriz. 2^5 yani 32 farklı operasyona kadar işlem destekleyebilen bir işlemcidir. Biz proje kapsamında 11 tane operasyon kullandık yani bazı girişler kullanılmadık. 32 bitlik 2 tane sayı operand_a ve operand_b ile gelir. Bu 2 sayı ile işlem yapılacak. "result" çıkışta o operasyonun gereksinimine göre hangi işlemi yaptıysak sonuç kısmında 32 bitlik olarak o sayıyı veriyoruz. "result_equal_zero"da ise eğer result 0 ise bu çıkıştan 1 çıkartıyoruz. Ama eğer result 0'ın haricinde başka bir şeyse 0 çıktısı veriliyor.

```
11     logic REZ;
```

"result_equal_zero" logic olarak tanımlanmadığı için always_comb bloğunda kullanmıyoruz. Bu yüzden always_comb bloğu bittikten sonra "result_equal_zero" üzerine assign edebilmek ve bu bloğun içerinde kullanabilmek

adına REZ isimli bir logic tanımladık. Bu şekilde bloğun içerisinde değer güncellenebilecek.

```
always_comb begin
   if (alu_function==5'b00001) begin //ADD
       result = operand_a + operand_b;
       if (result==0) begin
           REZ=1;
       end else begin
           REZ=0;
       end
   end else if(alu_function==5'b00010)begin //SUB
       result = operand_a - operand_b;
       if (result==0) begin
           REZ=1;
       end else begin
           REZ=0;
       end
   end else if(alu_function==5'b00011)begin //SLL
       result = operand_a << operand_b;
       if (result==0) begin
           REZ=1;
       end else begin
           REZ=0;
       end
   end else if(alu_function==5'b00100)begin //SRL
       result = operand_a >> operand_b;
       if (result==0) begin
           REZ=1;
       end else begin
           REZ=0;
       end

   end else if(alu_function==5'b00101)begin //SRA
       result = operand_a >>> operand_b;
       if (result==0) begin
           REZ=1;
       end else begin
           REZ=0;
       end

   end else if(alu_function==5'b00110)begin //SEQ
       result = (operand_a == operand_b);
       if (result==0) begin
           REZ=1;
       end else begin
           REZ=0;
       end

   end else if(alu_function==5'b00111)begin //SLT
       result = operand_a < operand_b;
       if (result==0) begin
           REZ=1;
       end else begin
           REZ=0;
       end

   end else if(alu_function==5'b01000)begin //SLTU
       result =  $unsigned(operand_a) < $unsigned(operand_b) ;
       if (result==0) begin
           REZ=1;
       end else begin
           REZ=0;
           end

     end else if(alu_function==5'b01001)begin //XOR
     result = operand_a ^ operand_b;
         if (result==0) begin
             REZ=1;
         end else begin
             REZ=0;
         end

     end else if(alu_function==5'b01010)begin //OR
     result = operand_a | operand_b;
         if (result==0) begin
             REZ=1;
         end else begin
             REZ=0;
         end

     end else if(alu_function==5'b01011)begin //AND
     result = operand_a & operand_b;
         if (result==0) begin
             REZ=1;
         end else begin
             REZ=0;
         end

     end

end
```

Tasarımın bu modülünde clk almadığı için kombinasyonel devre şeklinde tasarlanmıştır. Bloğun içerisinde gelen alu_function değerine göre hangi aritmetik işlemin yapılacağı seçiliyor. Örneğin 5'b00001 sayısı geldiyse bu toplama işlemini ifade ettiği için operand a ve b toplanır ve result değerine atanır. Sonrasında result değerinin 0 olup olmamasına göre REZ'e 0 veya 1 değeri atanır.

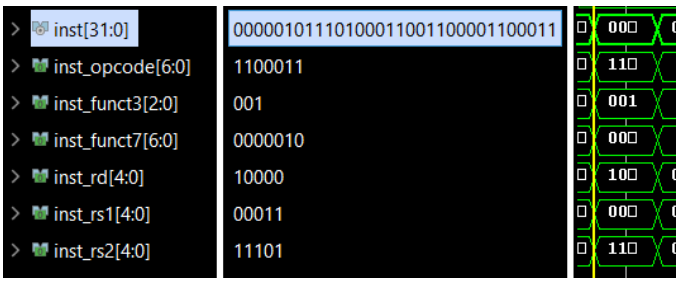
```
assign result_equal_zero = REZ;
```

Gerekli işlem ve kontroller yapıldıktan ve always_comb bloğunun dışına çıkıldıktan sonra güncellenen REZ değeri çıkış yapabilmesi için "esult_equal_zero"ya assign edilir.

IV. SONUÇLAR

Yazdığımız kodları Vivado'nun simülasyon programıyla çalıştırdığımızda waveform üzerinden giriş ve çıkışların değerlerini gözlemleyebiliyoruz.

Önce alu'daki aritmetik işlemlerin yapılabilmesi için instruction_decoder'da komutların parçalanması gerekiyordu. Waveformda bunu gözlemlediğimizde gelen 32 bitlik instruction bilgisinin doğru bir şekilde parçalara ayrıldığını gördük. Her clk'un posedge'inde (sağ taraftaki sarı çizgi clk'un posedge olduğu kısmı göstermektedir.) yeni instruction bilgisi decoder'a gelir ve parçalara ayrılır/ decode edilir.



Proje kapsamında, Alu'nun desteklediği 11 tane operasyon vardır. Bu operasyonlar;

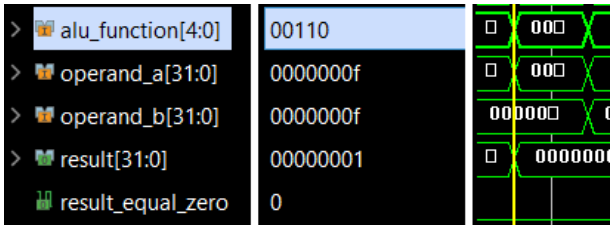
ALU_ADD	5'b00001
ALU_SUB	5'b00010
ALU_SLL	5'b00011
ALU_SRL	5'b00100
ALU_SRA	5'b00101
ALU_SEQ	5'b00110
ALU_SLT	5'b00111
ALU_SLTU	5'b01000
ALU_XOR	5'b01001
ALU_OR	5'b01010
ALU_AND	5'b01011

Alu'nun waveformlarına baktığımızda modül kominasyonel olarak tasarlandığı için sonuç direkt olarak o cycle'da dışarı verilir.

Alu_functiona gelen 5'b00001 biti, toplama işlemini ifade etmektedir. Add; Aritmetik toplama işlemi yapar(a ve b operatörleri toplanır). Operand_a ve operand_b'ye gelen 32 bitlik değerler toplandığında bulunan sonuç result'a yazılır.

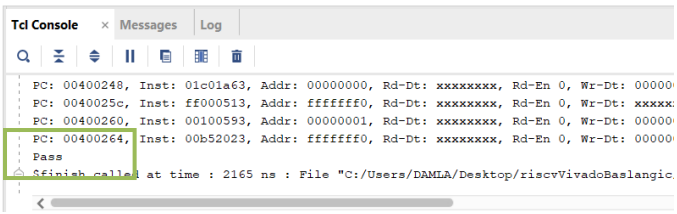
Sonuç 0'dan farklı olduğu için result_equal_zero 0 çıktısı vermiştir. Ama toplamı işlemindeki a ve b sayıları değiştiğinde result 0 olunca result_equal_zero 1 çıktısı verdiğini gözlemledik.

Sub; Aritmetik çıkarma işlemi yapar(a operatöri, b operatörlerinden çıkartılır.). Sll; Sayıyı sola kaydırır (a << b ifadesi a sayısını b bit sola kaydırır). Srl; Sayıyı sağa kaydırır (a >> b ifadesi a sayısını b bit sağa kaydırır). Sra; (aritmetik sağa kaydırma) belirtilen sayıda biti sağa kaydırır, ifade işaretliyse işaret bitinin değeriyle doldurur, aksi takdirde sıfırla doldurur.



Alu_functiona gelen 5'b00110 biti eşit mi kontrolünü ifade etmektedir yani Seq; a ve b operatörleri eşit mi diye kontrol eder. A ve B operatörleri birbirlerine eşit olduğu için result 1'dir. Result 0'dan farklı olduğu için result_equal_zero 0 çıktısını verir.

Slt; b sayısı a sayısından büyük mü diye kontrol eder. Sltu; gelen sayılar two's complement'dir. Bu işlem ile sayların işareti atılıp(işaretsiz hale getirilip) öyle karşılaştırılır. Xor; Bit - bit XOR işlemini uygular. Or; Bit - bit VEYA işlemini uygular. And; Bit - bit VE işlemini uygular.



Tasarım simülasyonu başlatılıp play tuşuna basıtığımızda, sonuç hesaplandığı için şekilde görülebilen "Pass" çıktısını vermektedir.

Geliştirdiğimiz ve kodunu tamamlandığımız işlemcinin sonucunda Riscv'nin operasyon kodlarının çalışma mekanizmasını anlamış olduk. 32 bitlik instruction'ın girişinin decode edilmiş hali instruction_decoder'da yapılmakta sonrasında da Alu'da, işlemlere göre a ve b sayıları, result isminde sonuç çıkışı ve sonuç eğer sıfır ise, ayrı bir çıkış olarak sonucun sıfır olması durumunda 1 olan bir çıktı verdiğini simülasyon üzerinde cycle cycle gözlemledik. Algoritmik düşünce yeteneğimiz gelişmiş ve system verilog diline olan hakimiyetimiz artmıştır.

Proje Ekibi

Damla Su KARADOĞAN, 11.02.2001 yılında doğdu. 2019 yılında Özel Envar Anadolu Lisesinden mezun oldu. Şu anda Fenerbahçe Üniversitesinde Endüstri 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

https://www.youtube.com/watch?v=UYmHU6rr_Ic

Kaynaklar

(2021, 05 1). HTTPS://COMPAS.CS.STONYBROOK.EDU/~NHONARMAND/COURSES/SP15/CSE502/SLIDES/03-SYSTEMVERILOG.PDF ADRESINDEN ALINDI

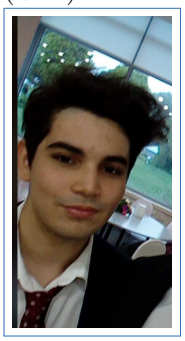
(2021, 05 1). http://www.cs.columbia.edu/~simha/teaching/4340_fa14/lectures/Unit2.pdf adresinden alındı

(2021, 05 1). https://www.cl.cam.ac.uk/teaching/1112/ECAD+Arch/files/SystemVerilogCheatSheet.pdf adresinden alındı

(2021, 05 1). https://lab.deepcontrol.net/risc-v-nedir/#:~:text=RISC%2DV(risk%2Dfayf,firmalara%20%C3%B6rnek%20ise%20ARM%20verilebilir. adresinden alındı

levent, e. (2021, 05 1). levent.tc. levent.tc: http://www.levent.tc/files/courses/computer_architecture/lectures/lec9/BLM202_hafta9_RISCV_design.pdf adresinden alındı

(+90) 554 644 86 11



Alp Eren Gurle

Student

Education

2019–NOW **Computer Engineering**, *Fenerbahce University*, İstanbul-Turkey.

2014–2018 **High School**, *Ugur Anadolu Lises*, Denizli-Turkey.

Projects

Verilog

2020 **FB-CPU RTL Design**.

C++

2020 **Telephone Registration System**.

Python

2020 **Population Management System**.

Computer skills

Languages

Basic HTML

Intermediate PYTHON

Advanced C/C++

Languages

Turkish **Mothertongue**

English **Intermediate** *Conversationally fluent*

Interests

- Listening music
- Video Games
- Running
- Chess
- Basketball
- Swimming

Atasehir,İstanbul – Atasehir,İstanbul • ✉ *alpgurle1@gmail.com*

Damla Su KARADOĞAN

CV

EĞİTİM

2019–Günümüz **Endüstri Mühendisliği Lisans Bölümü**, *Fenerbahçe Üniversitesi*, İstanbul/Türkiye.

2020–Günümüz **Bilgisayar Mühendisliği Lisans Bölümü**, *Fenerbahçe Üniversitesi*, İstanbul/Türkiye.

2015–2019 **Özel Envar Anadolu Lisesi**, Antalya/Türkiye.

PROJELER

Python

2021 **Nüfus Yönetim Sistemi**.
Veri tabanı üzerinde kişi kimlik bilgilerini ekleme/silme/güncelleme/arama

Verilog

2021 **FB-CPU RTL Tasarım**.
FB işlemcisinde verilog dili ile aritmatik işlemler

PROGRAMLAMA DİLLERİ

Orta Düzey Python,C, AutoCAD,Office Programları

Temel Verilog, JAVA

DİL

Türkçe **Anadil**

İngilizce **Orta Düzey**

İLGİ ALANI

- Programlama
- Tiyatro/Sinema
- Sanat
- Voleybol

0537 214 26 62 • damla.karadogan@stu.fbu.edu.tr

(+90) 536 819 52 50

TAHA YASIN OZTURK

Student

Education

2019–NOW **Computer Engineering**, *Fenerbahce University*, İstanbul-Turkey.

2014–2018 **High School**, *Erbakir Fen Lisesi*, Denizli-Turkey.

Projects

Verilog

2020 **FB-CPU RTL Design**.

C++

2020 **Telephone Registration System**.

Python

2020 **Population Management System**.

Computer skills

Languages

Basic HTML

Intermediate PYTHON

Advanced C/C++

Languages

Turkish **Mothertongue**

English **Intermediate** *Conversationally fluent*

Interests

- Listening music
- Video Games
- Running
- Design
- Basketball
- Swimming

Atasehir,İstanbul – Atasehir,İstanbul • ✉ *ozturktahayasin@gmail.com*