BİLGİSAYAR MİMARİSİ VE ORGANİZASYONU

PERFORMANS GÖREVİ-3 PROJE RAPORU

Ders Adı: Bilgisayar Mimarisi ve Organizasyonu

Öğretim Görevlisi: Serkan Dereli

Ödevi Yapanlar : Fatma Yaşar (23010903055)

Sümeyye Gül (23010903049)

Proje Amacı: Bu projede, YILDIZ isimli özel 16-bit işlemcinin donanım seviyesinde Verilog HDL (Hardware Description Language) kullanılarak gerçekleştirilmesi amaçlanmaktadır. Önceki performans görevlerinde tasarımı yapılan CPU'nun veriyolu, kontrol birimi, ALU ve bellek yapısı gibi temel bileşenleri donanımsal olarak modellenmiş, ardından bu modüller bir araya getirilerek bütünsel bir CPU yapısı oluşturulmuştur.

1. Genel Yapı

Bu projede geliştirilen 16-bit CPU mimarisi, Verilog HDL ile modüler yapıda tasarlanmış olup aşağıdaki ana bileşenlerden oluşmaktadır:

Modül Adı	Açıklama
`alu.v`	Aritmetik ve mantıksal işlemleri gerçekleştirir.
`ram_16bit.v`	16-bit genişliğinde, harici belleği temsil eder.
`control_unit.v`	Komutlara göre kontrol sinyallerini üreten FSM içerir.
`data_path.v`	Tüm registerlar, ALU, MUX ve veri yollarını barındırır.
`register_bank.v`	Genel amaçlı register dosyasını içerir (R0–R7) ve SP, ISR.
`yildiz_cpu_16bit.v`	CPU bileşenlerini entegre eden üst seviye birim.
`YildizCPU16.v`	Ana CPU test veya entegrasyon modülüdür.
`YildizCPU16_tb.v`	CPU'nun çalışma doğruluğunu gözlemlemek için kullanılan testbench modülüdür.

2. Veri Yolu ve Kontrol Birimi Etkileşimi

CPU'nun çalışma mantığı, kontrol birimi tarafından yönetilen bir **finite state machine (FSM)** sistemi ile düzenlenmiştir. Her komutun kendi durumları (FSM aşamaları) vardır. Aşağıda control_unit kodumuzda kullandığımız kendi komutlarımızdan örnek bazı komutlar seçilip içinden FSM akışları verilmiştir:

2.1.Komutların Mikroişlem Aşamaları (FSM)

Her komut yürütülmeden önce **Fetch** ve **Decode** evrelerinden geçer. Bu aşamalar tüm komutlar için ortaktır.

FETCH ve DECODE Aşamaları

Komutun bellekten alınması ve çözülmesi:

FETCH Aşamaları:

- `S_FETCH_0:`
  `Bus ← PC, Bus_Sel = 11`
  `AR ← Bus, AR_Load = 1`
  `Address ← AR`
- `S_FETCH_1:`
  `PC ← PC + 1, PC_Inc = 1`
- `S_FETCH_2:`
  `Bus ← from_memory`
  `IR ← Bus, IR_Load = 1`

DECODE Aşaması:

- `S_DECODE_3:`
  `IR_Value ← IR` (Kontrol birimi komutu çözümler)

2.2. Seçtiğimiz Komutlara Ait FSM Aşamaları

Kullanılacak Komutlar

Zinde CPU'da aşağıdaki komutlar örnek olarak kullanılacaktır:

1. S → SUBM ADR A
2. S → ADDM ADR A
3. R → SUB REG A
4. I → LDA IMM A
5. I → INC A

NOT: Bu aşamada gösterilen komutlar için verilen adımlar **EXECUTE** aşamasını göstermektedir.

1. Komut: S → SUBM ADR A

`SUBM $20`

Açıklama: `AC ← AC - M[20]`
Örnek Kod: `00 10 010011 010100`

FSM Adımları: (SUBM Bellek):

- `S_SUBM_ADR_4`: `AR ← IR[9:0]`, `AR_Load ← 1`
- `S_SUBM_ADR_5`: `Address ← AR`, `Mem_Read ← 1`
- `S_SUBM_ADR_6`: (Boş – veri gelmesi beklenir)
- `S_SUBM_ADR_7`: `Bus ← from_memory`, `DR_Load ← 1`
- `S_SUBM_ADR_8`: `AC ← AC - DR`, `AC_Load ← 1`
- `S_SUBM_ADR_9`: `Flags ← Güncellenir`

2. Komut: S → ADDM ADR A

`ADDM $20`

Açıklama: `AC ← AC + M[20]`
Örnek Kod: `00 10 010010 010100`

FSM Adımları: (ADDM Bellek)

- S_ADDM_ADR_4: AR ← IR[9:0], AR_Load ← 1
- S_ADDM_ADR_5: Address ← AR, Mem_Read ← 1
- S_ADDM_ADR_6: (Boş adım)
- S_ADDM_ADR_7: DR ← from_memory, DR_Load ← 1
- S_ADDM_ADR_8: AC ← AC + DR, AC_Load ← 1
- S_ADDM_ADR_9: Flags ← Güncellenir

3. Komut: R → SUB REG A

SUB R2

Açıklama: AC ← AC - R2
Örnek Kod: 000001 000 010 0000

FSM Adımları:

- S_SUB_REG_4: DR ← R[IR[3:0]], DR_Load ← 1
- S_SUB_REG_5: AC ← AC - DR, AC_Load ← 1
- S_SUB_REG_6: Flags ← Güncellenir

4. Komut: I → LDA_IMM A

LDA #05H

Açıklama: AC ← 05H
Örnek Kod: 00 01 001010 000101

FSM Adımları: (LDA Immediate):

- S_LDA_IMM_4:
  Bus ← PC, Bus_Sel = 11
  AR ← Bus, AR_Load = 1
  Address ← AR
- S_LDA_IMM_5: PC ← PC + 1, PC_Inc = 1
- S_LDA_IMM_6: Bus ← from_memory, DR_Load = 1
- S_LDA_IMM_7: AC ← DR, AC_Load = 1

5. Komut: I → INC A

INC

Açıklama: AC ← AC + 1
Örnek Kod: `00 01 001110 000000`

FSM Adımları:

- `S_INC_A_4`: AC ← AC + 1, AC_Inc ← 1
- `S_INC_A_5`: Flags ← Güncellenir

3.Test Senaryosu

YILDIZ CPU da şu komutlar örnek olarak kullanılacaktır:

Bellek Durumu Ön Bilgi:

- M[50H] = 07H → Bellekte 50 adresinde 7 var
- M[60H] = 02H → Bellekte 60 adresinde 2 var

İşlem Sırası (Program Akışı)

1. LDA #05H = AC ← 05
2. ADDM $50H =AC ← 05 + 07 = 0C
3. SUBM $60H = AC ← 0C - 02 = 0A
4. INC = AC ← 0A + 1 = 0B
5. STA $70H = M[70H] ← 0B

Adres	Veri	Açıklama
10H	0A	LDA #05H opcode
11H	05	Immediate değer 05
12H	12	ADDM $50H opcode
13H	50	Bellek adresi 50H
14H	13	SUBM $60H opcode
15H	60	Bellek adresi 60H
16H	0E	INC opcode
17H	11	STA $70H opcode
18H	70	Bellek adresi 70H
...	...	...
50H	07	Veri: M[50H] = 07
...	...	...
60H	02	Veri: M[60H] = 02
...	...	...
70H	??	Sonuç (program sonunda 0B olacak)

4.1.Modül Açıklaması: `YildizCPU16`

Bu Verilog modülü, 16 bit genişliğinde bir CPU'nun top-level (üst düzey) entegre yapısını temsil etmektedir. Modül, CPU çekirdeği (yildiz_cpu_16bit) ile 4K x 16-bit genişliğinde bir RAM modülünü (ram_16bit) birbirine bağlamaktadır.

```
module YildizCPU16 (
    input clkn, rstn,                           // Saat (clock) ve reset sinyalleri
    output [15:0] data_out, data_in,            // Bellek veri çıkışı ve giriş veri yolu
    output [15:0] tbDR, tbAC, tbIR, tbBus,      // Test çıkışları: Data Register, Accumulator, Instruction Register, Bus
    output [11:0] tbAR, tbPC,                    // Test çıkışları: Address Register ve Program Counter (12-bit)
    output [127:0] tbRegs,                       // Test çıkışı: Düzleştirilmiş (flat) genel amaçlı kayıt bankası R0-R7
    output [11:0] tbSP, tbISR,                   // Test çıkışları: Stack Pointer ve Interrupt Service Register
    output [3:0]  tbFLAGS,                       // Test çıkışı: Durum bayrakları (Flags)
    output [7:0]  tbINPR, tbOUTPR                // Test çıkışları: Giriş ve çıkış portları (Input & Output Port)
);

    wire we;                                     // Bellek yazma sinyali (write enable)
    wire [15:0] from_mem, to_mem;                // Bellekten veri çıkışı ve belleğe veri girişi
    wire [11:0] adr;                             // Bellek adres hattı

    // Test sinyalleri için ara kablolar (wires)
    wire [15:0] testDR, testAC, testIR, testBus; // Dahili CPU kayıtları ve veri yolu
    wire [11:0] testAR, testPC;                  // Dahili adres kayıtları
    wire [3:0]  testFLAGS;                        // Durum bayrakları
    wire [7:0]  testINPR, testOUTPR;             // Giriş/Çıkış portları
    wire [127:0] regbank_flat;                    // Düzleştirilmiş genel amaçlı kayıt bankası (R0-R7)
    wire [11:0] sp_wire, isr_wire;                // Stack Pointer ve Interrupt Service Register sinyalleri

    // Bellek veri bağlantıları: RAM'den CPU'ya ve CPU'dan RAM'e veri yolu atanması
    assign data_out = from_mem;                   // RAM'den CPU'ya gelen veri
    assign data_in  = to_mem;                     // CPU'dan RAM'e gönderilen veri

    // Test çıkışlarının modül çıkışlarına atanması
    assign tbDR     = testDR;
    assign tbAC     = testAC;
    assign tbIR     = testIR;
    assign tbBus    = testBus;
    assign tbAR     = testAR;
    assign tbPC     = testPC;
    assign tbFLAGS  = testFLAGS;
    assign tbINPR   = testINPR;
    assign tbOUTPR  = testOUTPR;

    assign tbRegs   = regbank_flat;              // Genel amaçlı registerların toplu çıkışı
    assign tbSP     = sp_wire;                    // Stack Pointer çıkışı
    assign tbISR    = isr_wire;                   // Interrupt Service Register çıkışı

    // CPU çekirdeği instantiation (yerleştirme)
    yildiz_cpu_16bit cpu (
        .clk(clkn),                               // Saat sinyali
        .rst(rstn),                               // Reset sinyali
        .from_memory(from_mem),                   // Bellekten gelen veri
        .to_memory(to_mem),                       // Belleğe yazılan veri
        .address(adr),                            // Bellek adresi
        .write(we),                               // Yazma enable sinyali

        // Dahili CPU sinyalleri test çıkışlarına bağlanıyor
        .testDR(testDR),
        .testAC(testAC),
        .testIR(testIR),
        .testAR(testAR),
        .testPC(testPC),
        .testBus(testBus),
        .testFLAGS(testFLAGS),
        .testINPR(testINPR),
        .testOUTPR(testOUTPR),
        .registers_out_flat(regbank_flat),        // R0-R7 kayıt bankası çıkışı
        .sp_out(sp_wire),                          // SP çıkışı
        .isr_out(isr_wire)                         // ISR çıkışı
    );

    // RAM modülü instantiation
    ram_16bit #(
        .DATA_WIDTH(16),                          // Veri genişliği 16 bit
        .ADDR_WIDTH(12)                           // Adres genişliği 12 bit (4096 adres)
    ) ram4096byte (
        .clk(clkn),                               // RAM saat sinyali
        .we(we),                                  // Yazma enable sinyali
        .addr(adr),                               // RAM adresi
        .din(to_mem),                             // RAM'e yazılacak veri
        .dout(from_mem)                           // RAM'den okunan veri
    );

endmodule
```

4.2. Modül Açıklaması: yildiz_cpu_16bit

Bu Verilog modülü, 16 bit genişliğinde bir CPU çekirdeğini tanımlar. İçinde kontrol birimi, veri yolu, ALU ve register bankası bulunur. Komutlar bellekten alınır, decode edilir ve işlem birimlerinde gerçekleştirilir. Modül, işlemci içindeki çeşitli ara sinyalleri ve register değerlerini test ve gözlem amaçlı dışa verir. Bellek adresleme ve yazma sinyalleri CPU'nun çalışmasını sağlar. Register bankası 8 genel amaçlı register, stack pointer ve interrupt servis registerı içerir.

```
module yildiz_cpu_16bit (
    input clk, rst,
    input [15:0] from_memory,
    output [15:0] to_memory,
    output [11:0] address,
    output write,

    // Test çıkışları
    output [15:0] testDR, testAC, testIR, testBus,
    output [11:0] testAR, testPC,
    output [3:0]  testFLAGS,
    output [7:0]  testINPR, testOUTPR,
    output [127:0] registers_out_flat, // R0-R7: 8×16 = 128 bit düzleştirilmiş çıktı
    output [11:0] sp_out, isr_out
);

    // Ara sinyaller
    wire IR_Load, DR_Load, PC_Load, AR_Load, AC_Load, FLAGS_Load;
    wire DR_Inc, AC_Inc, PC_Inc;
    wire [3:0] alu_sel;
    wire [2:0] bus_sel;

    // Register bankası yazma kontrolü
    wire [3:0]  rb_write_sel;
    wire reg_we;
    wire [15:0] rb_data1, rb_data2;

    // Bellek arabirim sinyalleri
    wire [15:0] to_mem;
    wire [11:0] adr;
    wire write_en;

    // Test amaçlı iç sinyaller
    wire [15:0] tDR, tAC, tIR, tBus;
    wire [11:0] tAR, tPC;
    wire [3:0] FLAGS_Value;
    wire [15:0] IR_Value;

    // Bellek arabirimi çıkışları
    assign to_memory = to_mem;
    assign address   = adr;
    assign write     = write_en;

    // Test gözlemleme sinyalleri
    assign testDR    = tDR;
    assign testAC    = tAC;
    assign testIR    = tIR;
    assign testBus   = tBus;
    assign testAR    = tAR;
    assign testPC    = tPC;
    assign testFLAGS = FLAGS_Value;
    assign testINPR  = 8'b0;
    assign testOUTPR = 8'b0;
```

```
    // Kontrol Ünitesi instantiation (komutları çözümleyip sinyalleri üretir)
    control_unit control (
        .clk(clk),
        .rst(rst),
        .IR_Value(tIR),            // Komut registerından komut değeri
        .FLAGS_Value(FLAGS_Value), // Durum bayrakları
        .IR_Load(IR_Load),         // Komut register yükleme
        .DR_Load(DR_Load),         // Data register yükleme
        .PC_Load(PC_Load),         // Program sayacı yükleme
        .AR_Load(AR_Load),         // Adres register yükleme
        .AC_Load(AC_Load),         // Akümülatör yükleme
        .FLAGS_Load(FLAGS_Load),   // Bayraklar yükleme
        .DR_Inc(DR_Inc),           // Data register arttırma
        .AC_Inc(AC_Inc),           // Akümülatör arttırma
        .PC_Inc(PC_Inc),           // Program sayacı arttırma
        .alu_sel(alu_sel),         // ALU seçim sinyali
        .bus_sel(bus_sel),         // Veri yolu seçim sinyali
        .reg_we(reg_we),           // Register bankası yazma etkinleştirme
        .reg_write_sel(rb_write_sel), // Yazılacak register seçimi
        .write_en(write_en)        // Bellek yazma etkinleştirme
    );

    // Register Bankası instantiation
    register_bank regbank_inst (
        .clk(clk),
        .rst(rst),
        .write_sel(rb_write_sel),   // Yazılacak register seçimi
        .write_en(reg_we),          // Yazma etkin sinyali
        .write_data(tBus),          // Yazılacak veri (veri yolu çıkışı)
        .read_data1(rb_data1),      // Okunan veri 1
        .read_data2(rb_data2),      // Okunan veri 2
        .regs_out_flat(regs_out_flat_internal)  // Düzleştirilmiş register çıkışı
    );

    // Veri yolu ve hesaplama birimi instantiation (Data Path)
    data_path datapath (
        .clk(clk),
        .rst(rst),
        .IR_Load(IR_Load),
        .DR_Load(DR_Load),
        .PC_Load(PC_Load),
        .AR_Load(AR_Load),
        .AC_Load(AC_Load),
        .FLAGS_Load(FLAGS_Load),
        .DR_Inc(DR_Inc),
        .AC_Inc(AC_Inc),
        .PC_Inc(PC_Inc),
        .alu_sel(alu_sel),
        .bus_sel(bus_sel),
        .from_memory(from_memory),
        .to_memory(to_mem),
        .address(adr),
        .IR_Value(IR_Value),
        .FLAGS_Value(FLAGS_Value),
        .tDR(tDR),
        .tAC(tAC),
        .tIR(tIR),
        .tAR(tAR),
        .tPC(tPC),
        .tBus(tBus),
        .rb_data1(rb_data1),
        .rb_data2(rb_data2),
        .bus(tBus)
    );

endmodule
```

4.3. Modül Açıklaması: data_path

Data_path modülü, CPU'nun veri yolunu ve temel registerlarını içerir. Komut (IR), veri (DR), akümülatör (AC), adres (AR) ve program sayacı (PC) gibi registerların güncellenmesini sağlar. ALU'dan gelen işlemleri yürütür, durum bayraklarını (FLAGS) günceller ve veri yolunda hangi verinin aktarılacağını kontrol eder. Ayrıca, bellek arayüzü için gerekli adres ve veri sinyallerini üretir. CPU içi veri akışının merkezi olarak çalışır.

```
module data_path (
    input clk, rst,                             // Saat ve reset sinyalleri

    input IR_Load, DR_Load, PC_Load, AR_Load, AC_Load, FLAGS_Load,  // Register yükleme kontrol sinyalleri
    input DR_Inc, AC_Inc, PC_Inc,              // Register artırma kontrol sinyalleri
    input [3:0] alu_sel,                        // ALU işlem seçici
    input [2:0] bus_sel,                        // Veri yolu seçim sinyali

    input [15:0] from_memory,                   // Bellekten gelen veri

    output [15:0] to_memory,                    // Belleğe yazılacak veri
    output [11:0] address,                      // Bellek adresi

    output [15:0] IR_Value,                     // Komut register içeriği (test çıkışı)
    output [3:0] FLAGS_Value,                   // Durum bayrakları (test çıkışı)

    output [15:0] tDR, tAC, tIR, tBus,         // Ara register ve bus test çıkışları
    output [11:0] tAR, tPC,

    input [15:0] rb_data1,                      // Register bankası 1. okuma verisi
    input [15:0] rb_data2,                      // Register bankası 2. okuma verisi
    output [15:0] bus                           // CPU iç veri yolu
);

    // Dahili CPU registerları
    reg [15:0] IR, DR, AC;                      // Komut, veri ve akümülatör registerları
    reg [11:0] AR, PC;                          // Adres ve program sayacı
    reg [3:0] FLAGS;                            // Durum bayrakları (Zero, Negatif, Taşma, vs.)

    wire [15:0] alu_out;                        // ALU işlem sonucu

    // Veri yolu seçimi: bus_sel sinyaline göre hangi verinin bus'a bağlanacağı belirlenir
 assign bus = (bus_sel == 3'b000) ? rb_data1 :
              (bus_sel == 3'b001) ? rb_data2 :
              (bus_sel == 3'b010) ? from_memory :
              (bus_sel == 3'b011) ? {4'd0, PC} :     // PC 12-bit'ten 16-bit'e genişletiliyor
              (bus_sel == 3'b100) ? DR :
              (bus_sel == 3'b101) ? AC :
              (bus_sel == 3'b110) ? 16'd0 :
              16'd0;  // Default değer

    // Bellek arayüzü çıkışları
    assign to_memory = bus;                     // Belleğe yazılacak veri bus üzerinden gönderilir
    assign address = AR;                        // Bellek adresi AR registerından alınır

    // Test çıkışları: register ve bus değerleri gözlem için dışa verilir
    assign tDR = DR;
    assign tAC = AC;
    assign tIR = IR;
    assign tAR = AR;
    assign tPC = PC;
    assign tBus = bus;
    assign IR_Value = IR;
    assign FLAGS_Value = FLAGS;
```

```
    // IR (Instruction Register) güncelleme
    always @(posedge clk or posedge rst) begin
        if (rst)
            IR <= 16'd0;                        // Reset durumunda sıfırla
        else if (IR_Load)
            IR <= bus;                          // Bus üzerindeki veriyi yükle
    end

    // DR (Data Register) güncelleme ve artırma işlemleri
    always @(posedge clk or posedge rst) begin
        if (rst)
            DR <= 16'd0;
        else if (DR_Load)
            DR <= bus;
        else if (DR_Inc)
            DR <= DR + 1;
    end

    // AC (Accumulator) güncelleme ve artırma işlemleri
    always @(posedge clk or posedge rst) begin
        if (rst)
            AC <= 16'd0;
        else if (AC_Load)
            AC <= alu_out;                      // ALU sonucu yüklenir
        else if (AC_Inc)
            AC <= AC + 1;
    end

    // AR (Address Register) güncelleme
    always @(posedge clk or posedge rst) begin
        if (rst)
            AR <= 12'd0;
        else if (AR_Load)
            AR <= bus[11:0];                    // Bus'tan 12-bit adres al
    end

    // PC (Program Counter) güncelleme ve artırma
    always @(posedge clk or posedge rst) begin
        if (rst)
            PC <= 12'd0;
        else if (PC_Load)
            PC <= bus[11:0];
        else if (PC_Inc)
            PC <= PC + 1;
    end

    // FLAGS registerı (Z, N, V, C bayrakları) güncelleme
    always @(posedge clk or posedge rst) begin
        if (rst)
            FLAGS <= 4'b0000;
        else if (FLAGS_Load) begin
            FLAGS[0] <= (alu_out == 16'd0);             // Zero flag (sonuç sıfır mı?)
            FLAGS[1] <= alu_out[15];                    // Negative flag (işaret biti)
            FLAGS[2] <= ^{AC[15], DR[15], alu_out[15]}; // Overflow flag (taşma durumu)
            FLAGS[3] <= alu_out[15];                    // Carry flag (taşma/işaret benzeri)
        end
    end

    // ALU modülü: AC ve DR registerlarından gelen verilerle işlem yapar
    alu alu_uut (
        .s1_in(AC),
        .s2_in(DR),
        .islem_in(alu_sel),
        .s_out(alu_out)
    );

endmodule
```

4.4. Modül Açıklaması: control_unit

`control_unit` modülü, verilen 16 bitlik komutları (IR_Value) çözerek CPU içindeki kontrol sinyallerini üretir. Desteklenen komutlara göre (SUBM, ADDM, SUB, LDA immediate, INC) uygun işlem adımlarını FSM ile yönetir ve ALU, registerlar ile veri yolu için kontrol sinyallerini sağlar. Böylece CPU'nun doğru sırayla ve doğru işlemle çalışmasını kontrol eder.

```
//16 bitlik CPU veriyollu
// Sadece SUBM, ADDM, SUB (register), LDA #imm, INC komutlarını destekleyen bir control_unit

module control_unit(
    input clk,
    input rst,
    input [15:0] IR_Value,
    input [3:0] FLAGS_Value,
    output reg IR_Load, DR_Load, PC_Load, AR_Load, AC_Load,FLAGS_Load,
    output reg AC_Inc, PC_Inc, write_en, DR_Inc,
    output reg [3:0] alu_sel, // 000=ADD, 001=SUB, ...
    output reg [2:0] bus_sel,
     output reg [3:0] reg_sel,          // ALU'ya veri verecek kaynak register (src)
    output reg  reg_write_sel,   // Yazma hedefi (dest)
    output reg reg_we           // Register'a yazma enable
);

    reg [7:0] current_state, next_state;

    // Genel FSM Durumları
    localparam [7:0]
        S_FETCH_0 = 8'd0,
        S_FETCH_1 = 8'd1,
        S_FETCH_2 = 8'd2,
        S_DECODE_3 = 8'd3,

        // SUBM $addr
        S_SUBM_4 = 8'd10,
        S_SUBM_5 = 8'd11,
        S_SUBM_6 = 8'd12,
        S_SUBM_7 = 8'd13,
        S_SUBM_8 = 8'd14,
        S_SUBM_9 = 8'd15,

        // ADDM $addr
        S_ADDM_4 = 8'd20,
        S_ADDM_5 = 8'd21,
        S_ADDM_6 = 8'd22,
        S_ADDM_7 = 8'd23,
        S_ADDM_8 = 8'd24,
        S_ADDM_9 = 8'd25,

        // SUB Rx
        S_SUBR_4 = 8'd30,
        S_SUBR_5 = 8'd31,
        S_SUBR_6 = 8'd32,

        // LDA #imm
        S_LDAI_4 = 8'd40,
        S_LDAI_5 = 8'd41,
        S_LDAI_6 = 8'd42,
        S_LDAI_7 = 8'd43,

        // INC
        S_INC_4 = 8'd50,
        S_INC_5 = 8'd51;

    // Komutlar ------ IR_Value değeri ile gelecek ----------

 localparam [5:0]
    OPCODE_SUBM = 6'b100011,
    OPCODE_ADDM = 6'b100010,
    OPCODE_SUBR = 6'b000001,
    OPCODE_LDAI = 6'b010010,
    OPCODE_INC  = 6'b010111;


  // ALU işlemleri
    localparam [3:0]
        ALU_ADD = 4'b0000,
        ALU_SUB = 4'b0001;


    // Geçiş: state logic
    always @(posedge clk) begin
        if (rst==1'b0) begin
           current_state <= S_FETCH_0;
           end
        else begin current_state <= next_state;
    end
    end

     // next state logic, sonraki durum güncellemesi
    always @(*) begin
        case (current_state)
            S_FETCH_0: next_state = S_FETCH_1;
            S_FETCH_1: next_state = S_FETCH_2;
            S_FETCH_2: next_state = S_DECODE_3;

            S_DECODE_3:
                case (IR_Value[15:10]) // İlk 6 bit opcode
                    OPCODE_SUBM: next_state = S_SUBM_4;
                    OPCODE_ADDM: next_state = S_ADDM_4;
                    OPCODE_SUBR: next_state = S_SUBR_4;
                    OPCODE_LDAI: next_state = S_LDAI_4;
                    OPCODE_INC:  next_state = S_INC_4;
                    default:     next_state = S_FETCH_0;
                endcase
```

```
                endcase

                // SUBM FSM
                S_SUBM_4: next_state = S_SUBM_5;
                S_SUBM_5: next_state = S_SUBM_6;
                S_SUBM_6: next_state = S_SUBM_7;
                S_SUBM_7: next_state = S_SUBM_8;
                S_SUBM_8: next_state = S_SUBM_9;
                S_SUBM_9: next_state = S_FETCH_0;

                // ADDM FSM
                S_ADDM_4: next_state = S_ADDM_5;
                S_ADDM_5: next_state = S_ADDM_6;
                S_ADDM_6: next_state = S_ADDM_7;
                S_ADDM_7: next_state = S_ADDM_8;
                S_ADDM_8: next_state = S_ADDM_9;
                S_ADDM_9: next_state = S_FETCH_0;

                // SUBR FSM
                S_SUBR_4: next_state = S_SUBR_5;
                S_SUBR_5: next_state = S_SUBR_6;
                S_SUBR_6: next_state = S_FETCH_0;

                // LDAI FSM
                S_LDAI_4: next_state = S_LDAI_5;
                S_LDAI_5: next_state = S_LDAI_6;
                S_LDAI_6: next_state = S_LDAI_7;
                S_LDAI_7: next_state = S_FETCH_0;

                // INC FSM
                S_INC_4: next_state = S_INC_5;
                S_INC_5: next_state = S_FETCH_0;

                default: next_state = S_FETCH_0;
            endcase
        end

        // Output logic
        always @(*) begin
            // Varsayılan: her şey sıfır
            IR_Load = 0; DR_Load = 0; PC_Load = 0;
            AR_Load = 0; AC_Load = 0; AC_Inc = 0;
            PC_Inc = 0; write_en = 0;
            alu_sel = 3'b000; bus_sel = 3'b000;
            reg_sel = 4'b0000;
            reg_write_sel = 4'b0000;
            reg_we = 0;

            case (current_state)
                S_FETCH_0: begin bus_sel = 3'b011; AR_Load = 1; end
                S_FETCH_1: begin PC_Inc = 1; end
                S_FETCH_2: begin IR_Load = 1; end
                S_DECODE_3: begin end

                // SUBM $addr
                S_SUBM_4: begin AR_Load = 1; end // AR ← IR[9:0]
                S_SUBM_5: begin end // Bellekten veri bekleniyor
                S_SUBM_6: begin end
                S_SUBM_7: begin DR_Load = 1; end
                S_SUBM_8: begin AC_Load = 1; alu_sel = 3'b001; end // SUB

                // ADDM $addr
                S_ADDM_4: begin AR_Load = 1; end
                S_ADDM_5: begin end
                S_ADDM_6: begin end
                S_ADDM_7: begin DR_Load = 1; end
                S_ADDM_8: begin AC_Load = 1; alu_sel = 3'b000; end // ADD

                // SUB Rn
                S_SUBR_4: begin DR_Load = 1; // DR ← R[IR[3:0]]

                reg_sel = IR_Value[3:0];
                    end
                S_SUBR_5: begin AC_Load = 1; alu_sel = 3'b001; end // AC ← AC -DR

                // LDA #imm
                S_LDAI_4: begin AR_Load = 1; bus_sel = 3'b011; end
                S_LDAI_5: begin PC_Inc = 1; end
                S_LDAI_6: begin DR_Load = 1; end
                S_LDAI_7: begin AC_Load = 1; end

                // INC
                S_INC_4: begin AC_Inc = 1; end

                //  ▼ Bu eklenen kısım: tüm sinyalleri sıfırlayan güvenli default
                default: begin
                    IR_Load = 1'b0;
                    DR_Load = 1'b0;
                    PC_Load = 1'b0;
                    AR_Load = 1'b0;
                    AC_Load = 1'b0;
                    AC_Inc = 1'b0;
                    PC_Inc = 1'b0;
                    write_en = 1'b0;
                    DR_Inc = 1'b0;
                    alu_sel = 4'b0000;
                    bus_sel = 2'b00;
                    reg_sel = 4'b0000;
                    reg_write_sel = 4'b0000;
                    reg_we = 1'b0;
                end
            endcase
        end

endmodule
```

4.5. Modül Açıklaması: alu

`alu` modülü, 4-bit işlem koduna (islem_in) göre iki 16-bit operand üzerinde aritmetik ve mantıksal işlemler yapar. Sonucu `s_out` çıkışına verir ve negatif (N), sıfır (Z), taşma (V) ve taşıma (C) bayraklarını `flags` ile bildirir. Toplama, çıkarma, artırma, azaltma, mantıksal AND, OR, XOR, NOT ve karşılaştırma işlemlerini destekler.

```
module alu(
    input [3:0] islem_in,        // 4-bit işlem kodu (OpCode)
    input [15:0] s1_in, s2_in,   // Giriş operandları
    output [15:0] s_out,         // ALU sonucu
    output [3:0] flags           // [N Z V C] bayrakları
);

    reg [15:0] sx;               // ALU çıkışı (işlem sonucu)
    reg [16:0] sy;               // Carry/taşma kontrolü için geniş sonuç

    assign s_out = sx;

    // FLAGS [3:0] = [N Z V C]
    assign flags[3] = sx[15]; // N: Negatif (sonucun işaret biti)
    assign flags[2] = (sx == 16'b0); // Z: Zero
    assign flags[1] = ((s1_in[15] == s2_in[15]) && (sx[15] != s1_in[15]) &&
                       (islem_in == 4'b0000 || islem_in == 4'b0001)) ? 1'b1 : 1'b0; // V: Overflow (sadece ADD & SUB)
    assign flags[0] = sy[16]; // C: Carry

    always @(*) begin
        case (islem_in)
            4'b0000: begin // ADD
                sx = s1_in + s2_in;
                sy = {1'b0, s1_in} + {1'b0, s2_in};
            end
            4'b0001: begin // SUB
                sx = s1_in - s2_in;
                sy = {1'b0, s1_in} - {1'b0, s2_in};
            end
            4'b0010: begin // INC
                sx = s1_in + 1;
                sy = {1'b0, s1_in} + 17'd1;
            end
            4'b0011: begin // DEC
                sx = s1_in - 1;
                sy = {1'b0, s1_in} - 17'd1;
            end
            4'b0100: begin // AND
                sx = s1_in & s2_in;
                sy = 0;
            end
            4'b0101: begin // OR
                sx = s1_in | s2_in;
                sy = 0;
            end
            4'b0110: begin // XOR
                sx = s1_in ^ s2_in;
                sy = 0;
            end
            4'b0111: begin // NOT
                sx = ~s1_in;
                sy = 0;
            end
            4'b1000: begin // CMP
                sx = s1_in - s2_in;
                sy = {1'b0, s1_in} - {1'b0, s2_in};
            end
            default: begin
                sx = 16'b0;
                sy = 0;
            end
        endcase
    end
endmodule
```

4.6. Modül Açıklaması: ram_16bit

`ram_16bit` modülü, 16-bit genişliğinde veri saklayan ve 12-bit adres genişliği ile 4096 kelimeye erişim sağlayan senkron bir bellektir. `clk` saat sinyaliyle çalışır, `we` sinyali ile yazma (1) veya okuma (0) işlemi seçilir. Adres (`addr`) ve yazılacak veri (`din`) girişlerinden alınan bilgiler belleğe yazılır veya bellekteki veri `dout` çıkışından okunur.

```
module ram_16bit #(
    parameter DATA_WIDTH = 16,       // 16-bit veri genişliği
    parameter ADDR_WIDTH = 12        // 12-bit adres hattı: 2^12 = 4096 adres
)(
    input clk,                       // Saat sinyali
    input we,                        // Write Enable (1: yazma, 0: okuma)
    input [ADDR_WIDTH-1:0] addr,     // Bellek adresi
    input [DATA_WIDTH-1:0] din,      // Yazılacak veri
    output [DATA_WIDTH-1:0] dout     // Okunan veri
);

    // 4096 x 16-bit bellek dizisi
    reg [DATA_WIDTH-1:0] bellek [0:(1<<ADDR_WIDTH)-1];
    reg [DATA_WIDTH-1:0] data;

    always @(posedge clk) begin
        if (we) begin
            bellek[addr] <= din;         // Yazma işlemi
        end else begin
            data <= bellek[addr];        // Okuma işlemi
        end
    end

    assign dout = data;

endmodule
```

4.7. Modül Açıklaması: register_bank

`register_bank` modülü, 8 adet genel amaçlı 16-bit register (R0-R7) ile birlikte iki özel amaçlı register olan Stack Pointer (SP) ve Interrupt Service Register (ISR)'ı barındırır. Bu yapı, işlemcinin hem genel veri işleme ihtiyaçlarını hem de kesme (interrupt) yönetimini destekleyecek şekilde tasarlanmıştır.

Modül, aynı anda iki farklı registerdan bağımsız olarak eşzamanlı okuma ve bir adet registera yazma işlemi gerçekleştirebilir. Yazma işlemi, `write_en` sinyali aktifken yapılır ve hedef register `write_sel` girişine göre seçilir. `write_sel` değeri:

- 0–7 arasında ise genel amaçlı registerlardan biri seçilir,
- 8 olduğunda Stack Pointer (SP),
- 9 olduğunda Interrupt Service Register (ISR) hedef alınır.

SP register'ı, kesmeler veya alt program çağrıları sırasında yığıt (stack) kontrolü için kullanılırken; ISR register'ı, hangi kesmeye hizmet verildiğini göstermek için kullanılır. Bu özel registerlar sayesinde sistem, kesme anında işlemci durumunu güvenli şekilde yedekleyip geri yükleyebilir.

Modül ayrıca, tüm genel amaçlı registerların (R0–R7) içeriğini tek bir 128-bit çıkış olarak (düzleştirilmiş biçimde) sunar. Bu, harici modüller için hızlı durum gözlemi sağlar. Sistem sıfırlandığında (`rst` aktif olduğunda), tüm registerlar temizlenir ve başlangıç değerine döner.

```
module register_bank (
    input clk, rst,
    input [3:0] read_sel1, read_sel2, write_sel,
    input write_en,
    input [15:0] write_data,
    output [15:0] read_data1, read_data2,
    output [127:0] regs_out_flat, // R0-R7 düzleştirilmiş
    output [11:0] sp_out, isr_out // SP & ISR ayrı çıkışlar
);
    reg [15:0] registers [0:7]; // R0-R7
    reg [11:0] SP;              // Stack Pointer
    reg [11:0] ISR;             // Interrupt Service Register

    integer i;
    always @(posedge clk or posedge rst) begin
        if (rst) begin
            for (i = 0; i < 8; i = i + 1)
                registers[i] <= 16'b0;
            SP <= 12'd0;
            ISR <= 12'd0;
        end else if (write_en) begin
            if (write_sel < 8)
                registers[write_sel] <= write_data;
            else if (write_sel == 8)
                SP <= write_data[11:0];
            else if (write_sel == 9)
                ISR <= write_data[11:0];
        end
    end

    assign read_data1 = (read_sel1 < 8) ? registers[read_sel1] :
                        (read_sel1 == 8) ? {4'd0, SP} :
                        (read_sel1 == 9) ? {4'd0, ISR} :
                        16'd0;

    assign read_data2 = (read_sel2 < 8) ? registers[read_sel2] :
                        (read_sel2 == 8) ? {4'd0, SP} :
                        (read_sel2 == 9) ? {4'd0, ISR} :
                        16'd0;

    // R0-R7 düzleştirme
    genvar j;
    generate
        for (j = 0; j < 8; j = j + 1) begin : flatten
            assign regs_out_flat[j*16 +: 16] = registers[j];
        end
    endgenerate

    // SP & ISR çıkışı
    assign sp_out = SP;
    assign isr_out = ISR;

endmodule
```

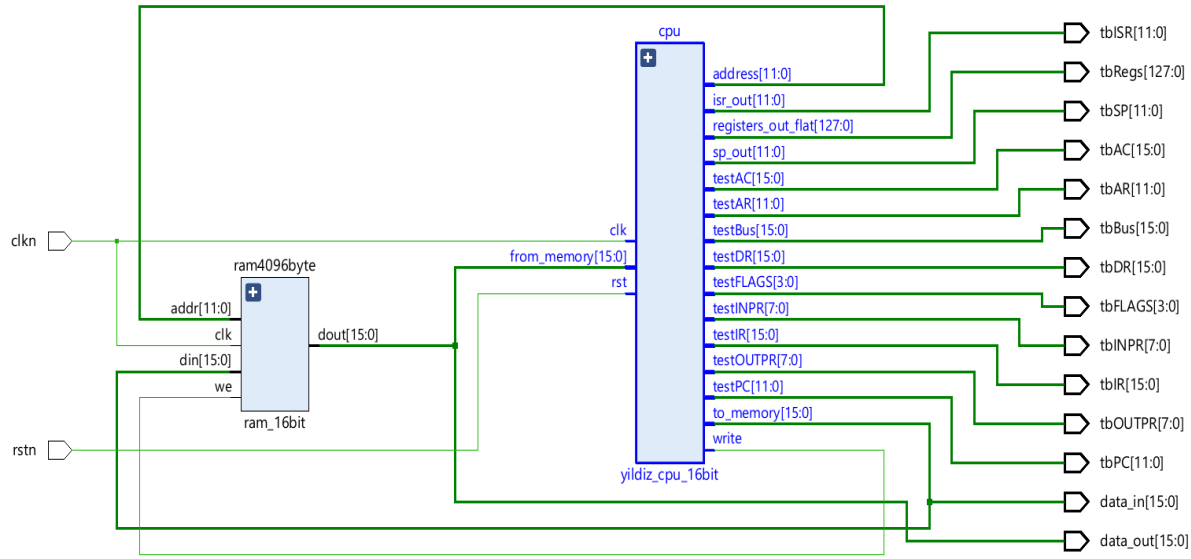
4.8. Modül Açıklaması: YildizCPU16_tb.v

Bu testbench modülü, `YildizCPU16` tasarımının doğruluğunu simülasyon ortamında test etmek için saat ve reset sinyalleri üretir; temel çalışma süreci gözlemlenir.

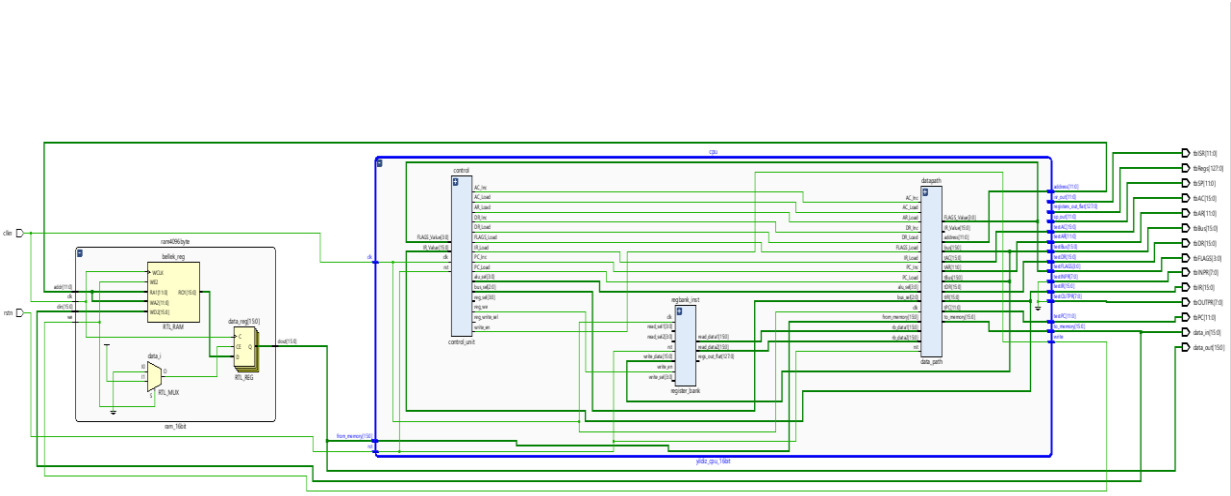
```
`timescale 1ns / 1ps

module YildizCPU16_tb;

    reg clk;
    reg rst;

    wire [15:0] data_out, data_in;
    wire [15:0] tbDR, tbAC, tbIR, tbBus;
    wire [11:0] tbAR, tbPC;
    wire [127:0] tbRegs;
    wire [11:0] tbSP, tbISR;
    wire [3:0] tbFLAGS;
    wire [7:0] tbINPR, tbOUTPR;

    // Instantiate the CPU top module
    YildizCPU16 uut (
        .clkn(clk),
        .rstn(rst),
        .data_out(data_out),
        .data_in(data_in),
        .tbDR(tbDR),
        .tbAC(tbAC),
        .tbIR(tbIR),
        .tbBus(tbBus),
        .tbAR(tbAR),
        .tbPC(tbPC),
        .tbRegs(tbRegs),
        .tbSP(tbSP),
        .tbISR(tbISR),
        .tbFLAGS(tbFLAGS),
        .tbINPR(tbINPR),
        .tbOUTPR(tbOUTPR)
    );

    // Clock generation
    always #5 clk = ~clk;  // 10ns period (100MHz)

    initial begin
        $display("----- YildizCPU16 Testbench Başlatılıyor -----");

        // Başlangıç değerleri
        clk = 0;
        rst = 0;

        #10;
        rst = 1;  // Reset aktif
        #20;
        rst = 0;  // Reset bırak
        $display("Reset bırakıldı");

        // Simülasyonu bir süre çalıştır
        #500;

        // Gözlemleme örneği
        $display("AC: %h", tbAC);
        $display("DR: %h", tbDR);
        $display("IR: %h", tbIR);
        $display("PC: %h", tbPC);
        $display("FLAGS: %b", tbFLAGS);
        $display("Registers [R0-R7]:");
        $display("R0: %h", tbRegs[15:0]);
        $display("R1: %h", tbRegs[31:16]);
        $display("R2: %h", tbRegs[47:32]);
        $display("R3: %h", tbRegs[63:48]);
        $display("R4: %h", tbRegs[79:64]);
        $display("R5: %h", tbRegs[95:80]);
        $display("R6: %h", tbRegs[111:96]);
        $display("R7: %h", tbRegs[127:112]);

        // Simülasyon sonu
        $finish;
    end

endmodule
```

5.YıldızCPU16 CPU, (temel mimari)

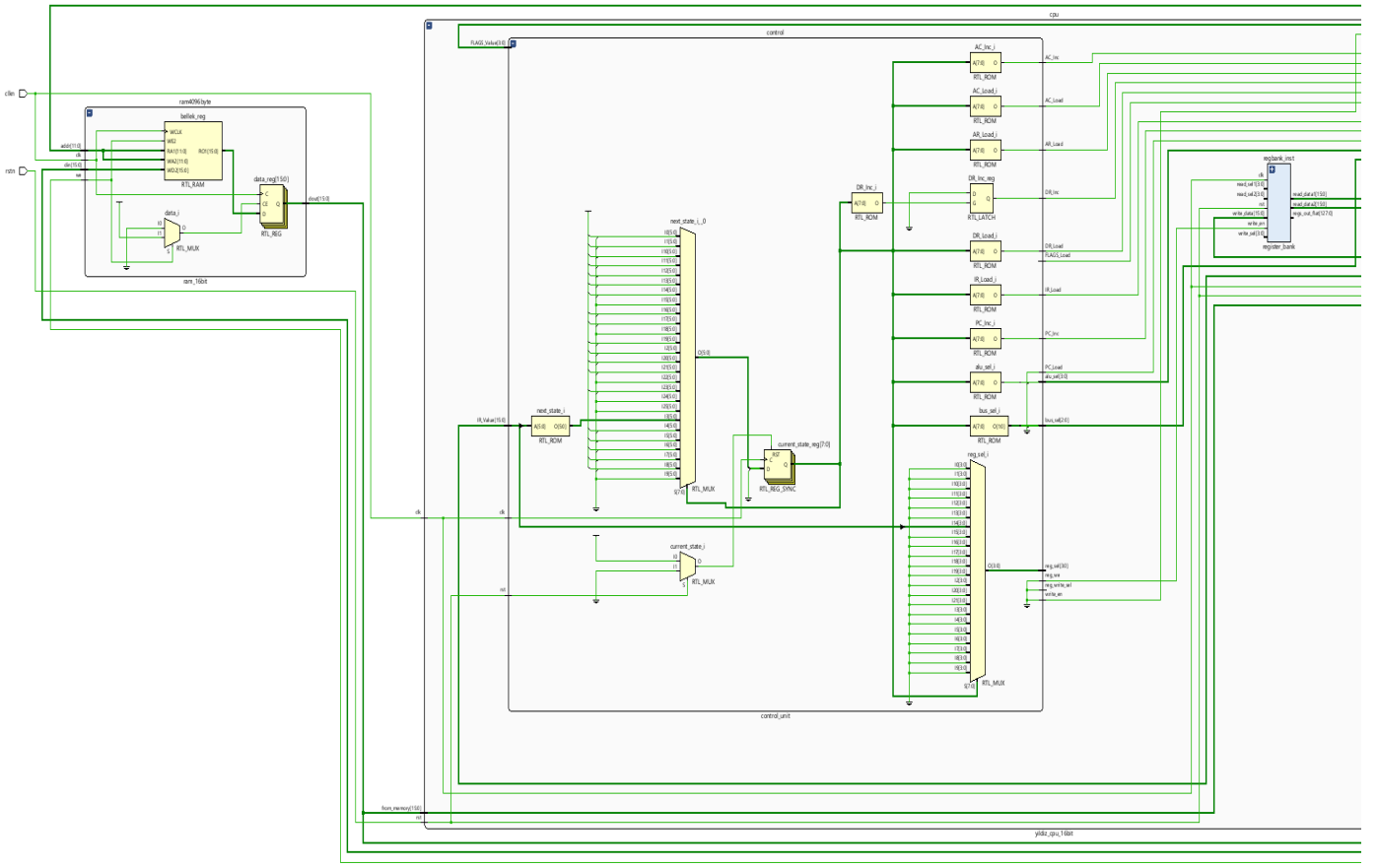
Bu proje kapsamında geliştirilen 16-bitlik CPU mimarisi aşşağıda görülmektedir.

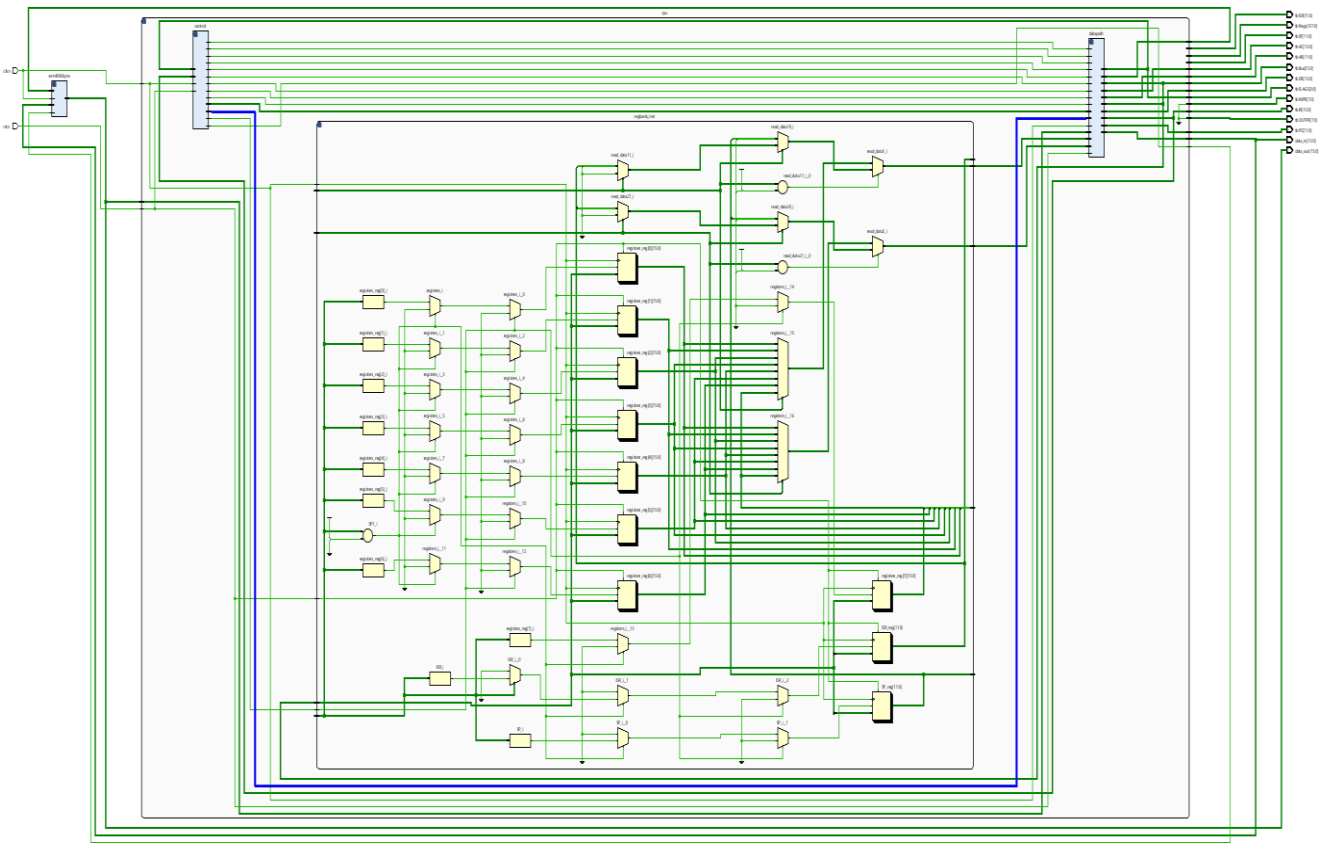


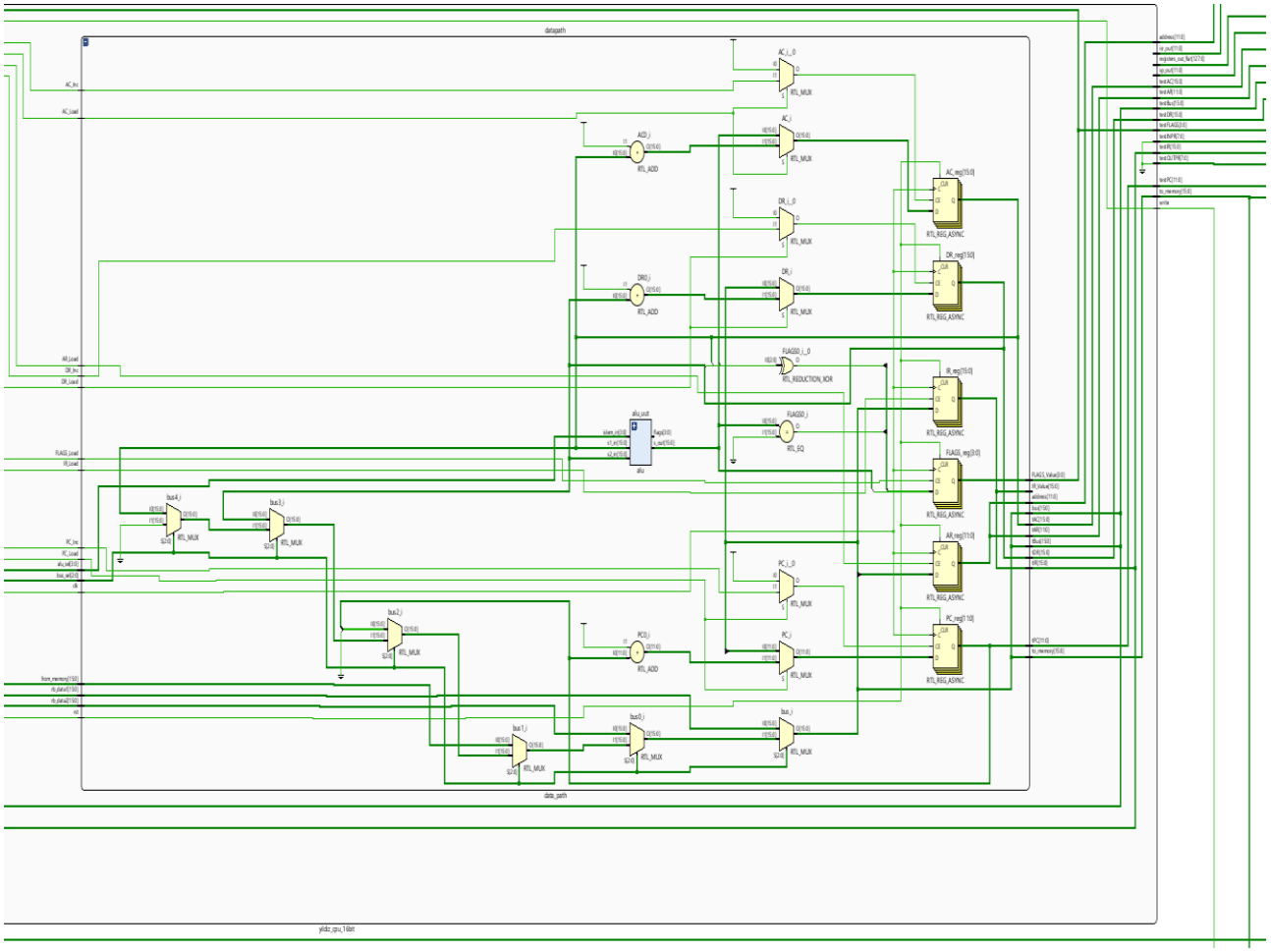
Aşşağıda CPU iç yapısı görülmektedir. Bu yapıda Veriyolu, kontrol birimi ve reg bank açıkça görülmektedir.



Bu blok şemada tüm alt modüllerin açıkça görülmektedir.

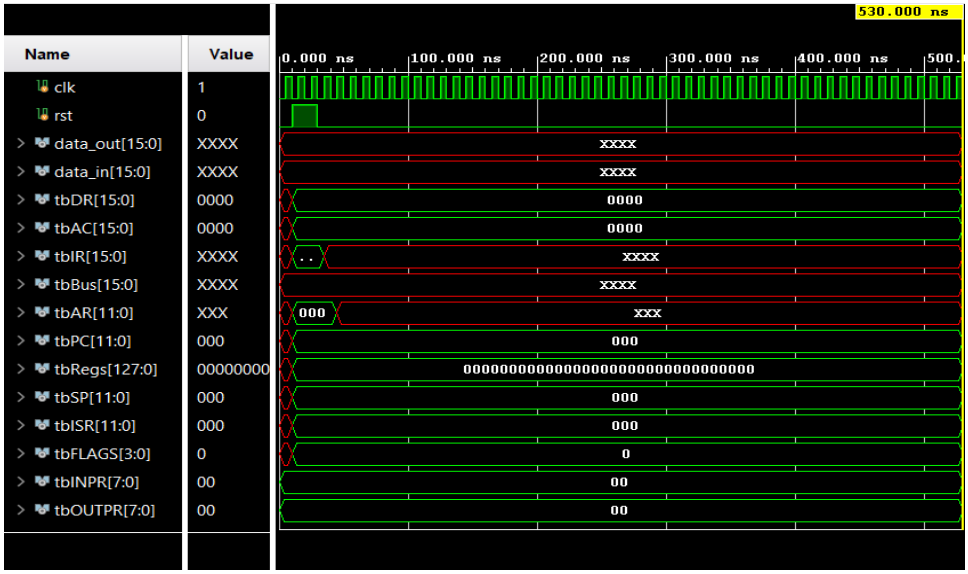

cpu
control
ram4096byte
bellek_reg
RTL_RAM
data_reg[15:0]
RTL_REG
data_i
RTL_MUX
ram_16bit
AC_inc_i
AC_Load_i
AR_Load_i
DR_inc_i
DR_inc_reg
RTL_LATCH
DR_Load_i
IR_Load_i
PC_inc_i
alu_sel_i
bus_sel_i
RTL_ROM
next_state_i_0
current_state_reg[7:0]
RTL_REG_SYNC
current_state_i
reg_sel_i
control_unit
regbank_inst
register_bank
yildiz_cpu_16bit



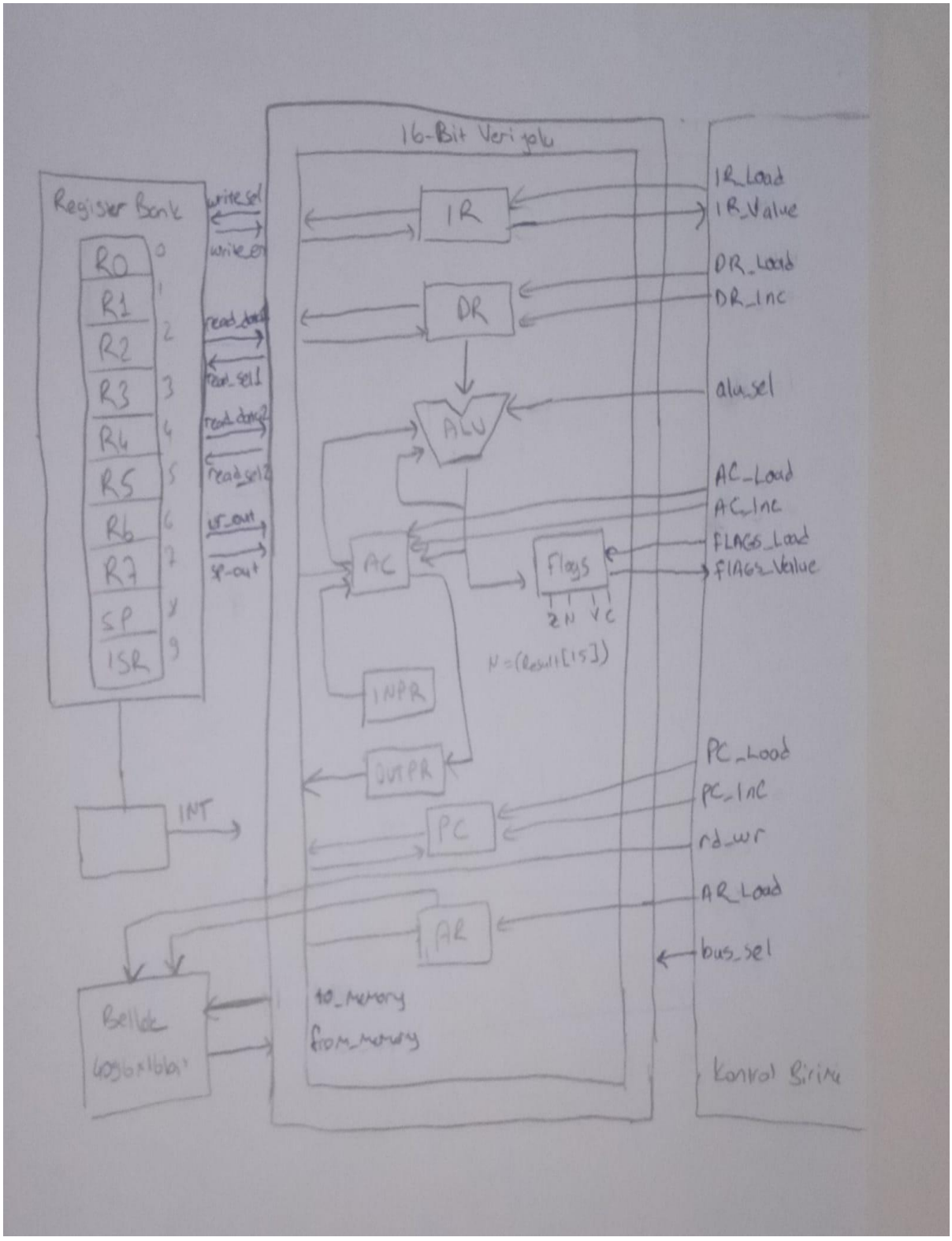

datapath
AC_inc
AC_load
AR_load
DR_inc
DR_load
FLAGS_load
IR_load
PC_inc
PC_load
clk
from_memory[15:0]
rst
AC_reg[15:0]
DR_reg[15:0]
IR_reg[15:0]
FLAGS_reg[3:0]
AR_reg[11:0]
PC_reg[11:0]
RTL_REG_ASYNC
RTL_MUX
RTL_ADD
RTL_REDUCTION_NOR
RTL_EQ
alu
data_path
ydidc_cpu_16bit

6. Testbench Simülasyon Çıktısı

Aşağıdaki şekilde `YildizCPU16_tb.v` testbench'i çalıştırıldığında oluşan simülasyon dalgaformları görülmektedir. Simülasyon, saat (`clk`) ve reset (`rst`) sinyallerine tepki olarak CPU'nun temel bileşenlerinin (örneğin PC, IR, AC, DR, FLAGS) nasıl değiştiğini göstermektedir. Bu dalgaformları, sistemin senkron ve doğru çalıştığını doğrulamak için kullanılmıştır.



Blok şemamız


16-Bit Veriyolu
Register Bank
R0
R1
R2
R3
R4
R5
R6
R7
SP
ISR
write_sel
write_en
read_data
read_sel
read_data2
read_sel2
sp_out
sp_out
IR
DR
ALU
AC
Flags
Z N V C
N = (Result[15])
INPR
OUTPR
PC
AR
INT
Bellek
to_Memory
from_Memory
IR_Load
IR_Value
DR_Load
DR_Inc
alu_sel
AC_Load
AC_Inc
FLAGS_Load
FLAGS_Value
PC_Load
PC_Inc
rd_wr
AR_Load
bus_sel
Kontrol Sinyalleri