Pipeline ve Hazard Unit Yapisina Sahip 32-bit RISC-V İşlemci Tasarimi

Fatih Sarıduman

December 20, 2025

PART I: RISC-V İŞLEMCI TASARIMI I

PART I: RISC-V İŞLEMCI TASARIMI I

Part II: Instruction Bazli Veri Yolu Analizi(Data Flow / Datapath Analysis) I

PART III: RISC-V BASE INSTRUCTION SET ENTEGRASYON TESTLERI I I

PART III: RISC-V BASE INSTRUCTION SET ENTEGRASYON TESTLERI II I

Part I

RISC-V İŞLEMCI TASARIMI

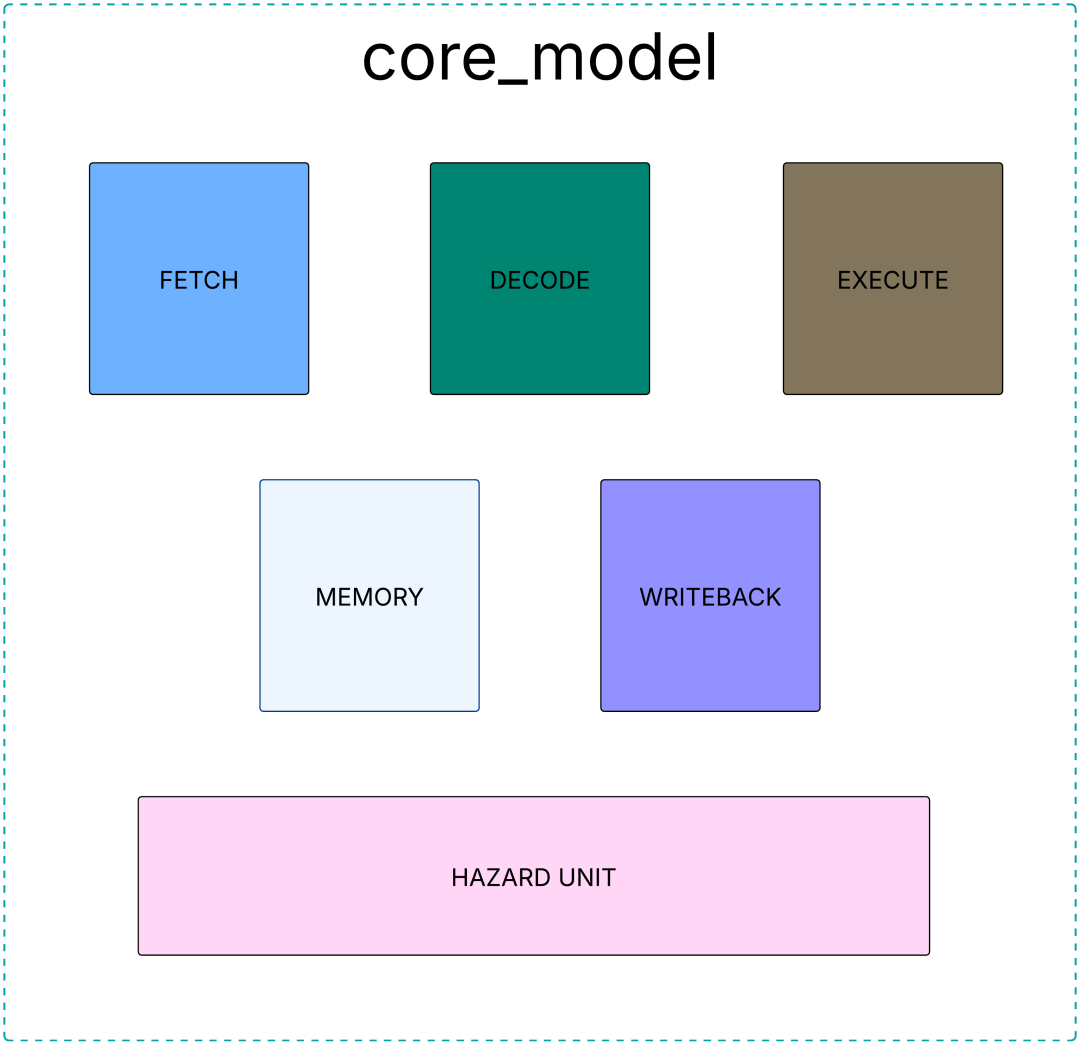

Figure. RISC-V Temel Modül Yapısı

Modüler mimari yapı sayesinde tasarım her aşama ayrı bir modül olacak şekilde tasarlanmıştır. SystemVerilog kodu tek bir **core_modul** içerisinde yazılmıştır, fakat bu modül içerisinde her modül **mantıksal olarak birbirinden ayrılmıştır** . Her aşamanın **input, output ve internal değişkenleri net bir şekilde belirtilmiştir** bu sayede kablolama karmaşıklığı minimalize edilmiştir. Temel mimari yapı **Figure. RISC-V Temel Modül Yapısı**'nda verilmiştir. Ayrıca kod içinde ayrımın nasıl yapıldığının ön gösterimi ise **Listing. Modül Ayrımı Ön Gösterimi**'nde gösterilmiştir.

Listing. Modül Ayrımı Ön Gösterimi

```
1     ////////////////////////////////FETCH AŞAMASI//////////////////////////
2     // INPUTLAR
3     // INTERNAL DEĞİŞKENLER
4     // OUTPUTLAR
5     // FETCH AŞAMASI KODLARI
6     ////////////////////////////////DECODE AŞAMASI//////////////////////////
7     ////////////////////////////////EXECUTE AŞAMASI/////////////////////////
8     ////////////////////////////////MEMORY AŞAMASI//////////////////////////
9     ////////////////////////////////WRITE BACK AŞAMASI/////////////////////
10    ////////////////////////////////HAZARD UNIT////////////////////////////
11
```

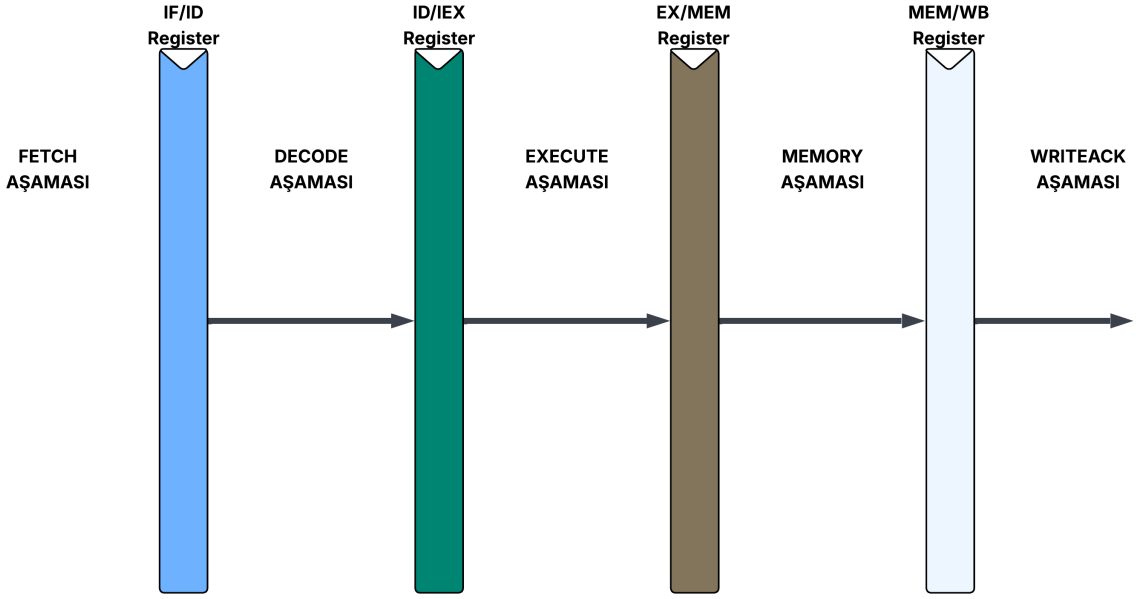

Figure. Pipeline Yapısı

- **Pipeline** yapısı sayesinde işlemci aynı anda birden fazla komut çalıştırma yetisine sahip olur. Bu sayede işlemcinin **verimi** artar ve performansı yükselir.
- **Figure. Pipeline Yapısı**'nda pipeline yapısının genel gösterimi verilmiştir. Bu yapıyı incelediğimizde **FETCH, DECODE, EXECUTE, MEMORY, WRITEBACK** şeklinde 5 aşama görüyoruz. Bu aşamalar arasında 4 adet **IF/ID, ID/EX, EX/MEM, MEM/WB** olmak üzere **pipeline registerlar** bulunmaktadır.
- Bu registerlar sayesinde her aşama kendi ürettiği değerleri güvenli bir şekilde bir sonraki aşamaya **her clock döngüsünde** aktarır. Ayrıca bu registerlar sayesinde senkronizasyon sağlanmış olur.
- Bir talimatın(instruction) pipeline üzerinde nasıl ilerlediği **Figure. Pipeline Talimat(Instruction) Yürütme Örneği**'nde gösterilmiştir. İncelediğimizde herbir clock döngüsünde yeni bir talimatın işlemciye geldiği ve bir önceki talimatın ise pipeline hattı üzerinde bir sonraki aşamaya ilerlediğini görüyoruz. Bu durumda 5. clock döngüsünden sonra her clock döngüsünde bir talimat tamamlanıyor ve aynı anda 5 adet talimat işleniyor bu da pipeline yapısının paralelliğini net bir şekilde gösteriyor.

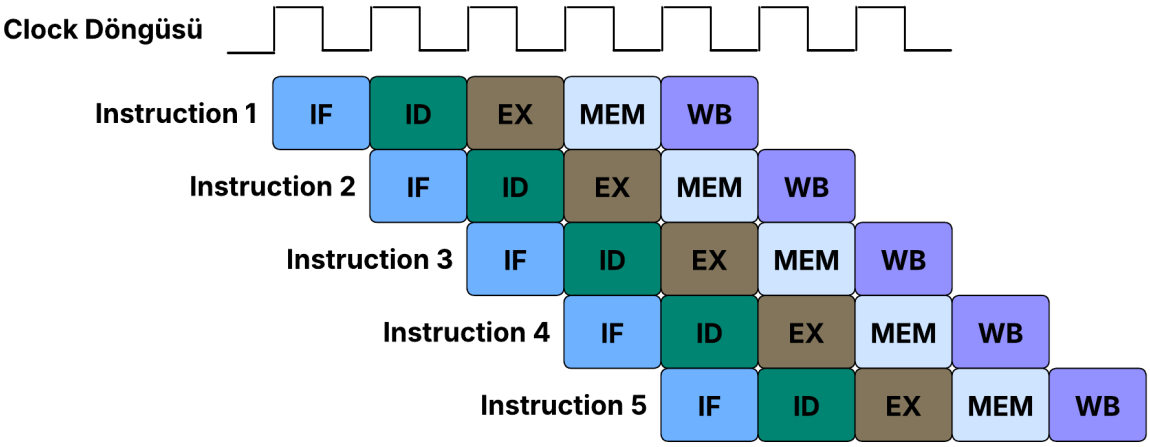

Figure. Pipeline Talimat(Instruction) Yürütme Örneği

Table. RISC-V RV32I Temel Komut Formatları

RV32I Instruction Formats						
31-25	**24-20**	**19-15**	**14-12**	**11-7**	**6-0**	**Type**
funct7	rs2	rs1	funct3	rd	opcode	R-TYPE
imm[11:0]		rs1	funct3	rd	opcode	I-TYPE
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode	S-TYPE
imm[12|10:5]	rs2	rs1	funct3	imm[4:1|11]	opcode	B-TYPE
imm[31:12]				rd	opcode	U-TYPE
imm[20|10:1|11|19:12]				rd	opcode	J-TYPE

Table. RV32I Temel Komut Kümesi

RV32I Base Instruction Set						
31-25	24-20	19-15	Field 3 14-12	11-7	6-0	Instruction
imm[31:12]				rd	0110111	LUI
imm[31:12]				rd	0010111	AUIPC
imm[20|10:1|11|19:12]				rd	1101111	JAL
imm[11:0]		rs1	000	rd	1100111	JALR
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	BGEU
imm[11:0]		rs1	000	rd	0000011	LB
imm[11:0]		rs1	001	rd	0000011	LH
imm[11:0]		rs1	010	rd	0000011	LW
imm[11:0]		rs1	100	rd	0000011	LBU
imm[11:0]		rs1	101	rd	0000011	LHU
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW
imm[11:0]		rs1	000	rd	0010011	ADDI
imm[11:0]		rs1	010	rd	0010011	SLTI
imm[11:0]		rs1	011	rd	0010011	SLTIU
imm[11:0]		rs1	100	rd	0010011	XORI
imm[11:0]		rs1	110	rd	0010011	ORI
imm[11:0]		rs1	111	rd	0010011	ANDI
0000000	shamt	rs1	001	rd	0010011	SLLI
0000000	shamt	rs1	101	rd	0010011	SRLI
0100000	shamt	rs1	101	rd	0010011	SRAI
0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU
0000000	rs2	rs1	100	rd	0110011	XOR
0000000	rs2	rs1	101	rd	0110011	SRL

31-25	24-20	19-15	14-12	11-7	6-0	Instruction
0100000	rs2	rs1	101	rd	0110011	SRA
0000000	rs2	rs1	110	rd	0110011	OR
0000000	rs2	rs1	111	rd	0110011	AND

FETCH AŞAMASI

FETCH AŞAMASI BLOK DIYAGRAMI

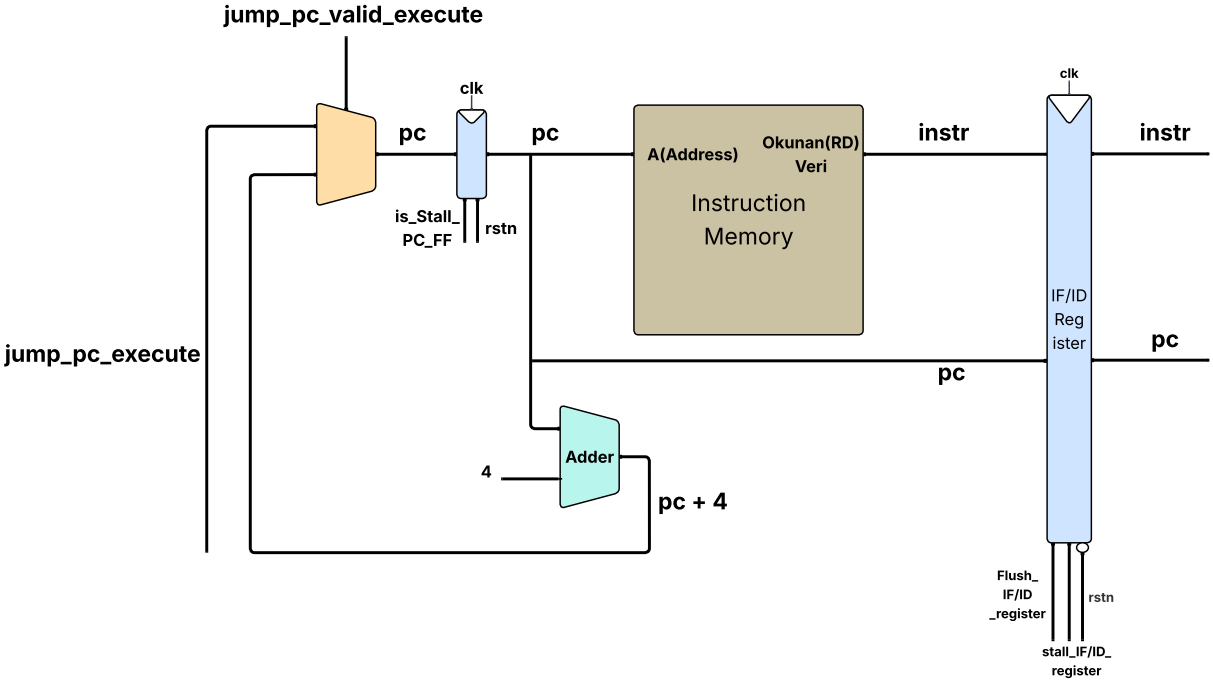

Figure. FETCH Aşaması

FETCH AŞAMASI

PROGRAM COUNTER CHANGE COMB

Listing. Program Counter Change Kod Bloğu

```
//Procram_Counter_Change_Comb
logic [XLEN-1:0] pc_d_fetch;

always_comb begin : program_counter_change_comb
if(jump_pc_valid_d_execute)
    pc_d_fetch = jump_pc_d_execute;
else
    pc_d_fetch = pc_q_fetch + 4;
end

```

FETCH AŞAMASI

PROGRAM COUNTER CHANGE COMB

- **Listing. FETCH Aşaması Kod Bloğu**
- **Program Counter Chang Comb** bloğu program counter değerinin değişmesini sağlayan, **kombinasyonel** bir bloktur. Burada iki dururum kontrol edilir, bunlar **junp_pc_valid_d_execute** sinyalinin durumlarıdır.
- Eğer bu sinyal **1** ise, bir atlama(jump/branch) durumu söz konusudur, bu durumda ilgili mux **jump_pc_d_execute** sinyalini seçer.
- Eğer sinyal **0** ise atlama(jump/branch) söz konusu değildir bu durumda ilgili mux **pc_q_fetch + 4** sinyalini seçer ve program doğal akışında devam eder.

FETCH AŞAMASI

PROGRAM COUNTER CHANGE FF

Listing. Program Counter Change FF Kod Bloğu

```
//Program_Counter_Change_FF
logic [XLEN-1:0] pc_q_fetch;
logic            update_q_fetch;

always_ff @(posedge clk or negedge rstn) begin : program_counter_change_ff
    if(!rstn) begin
        pc_q_fetch <= 'h8000_0000;
        update_q_fetch <= 0;
    end
    else if(is_Stall_PC_FF) begin
        pc_q_fetch <= pc_q_fetch;
        update_q_fetch <= 0;
    end
    else begin
        pc_q_fetch <= pc_d_fetch;
        update_q_fetch <= 1;
    end
end

```

- **Listing. Program Counter Change FF Kod Bloğu**
- Mux çıkışındaki **pc_d_fetch** sinyali **program_counter_change_ff** register'ına gider ve bu register'a giren sinyallere göre **rstn, is_Stall_PC_FF** register'ın çıkış değeri olan **pc_q_fetch** belirlenir.
- **rstn** sinyali **0** olduğunda register ne olursa olsun sıfırlanır.
- Eğer rstn sinyali 1 ve **is_Stall_PC_FF** sinyali de 1 ise program_counter_change_ff bir cycle boyunca stall edilir yani değeri değişmez.

FETCH AŞAMASI

INSTRUCTION READ COMB

Listing. Instruction Read Comb Kod Bloğu

```
1           //Instruction_Read_Comb
2           logic [31:0] instruction_memory [MEM_SIZE-1:0]; // Intruction memory tanımı
3           // Test dosyasını memory'e yüklüyoruz.
4           initial $readmemh("./test/test.hex", instruction_memory, 0, MEM_SIZE);
5           //initial $readmemh("./test/instruction3.hex", instruction_memory, 0, MEM_SIZE);
6
7           logic [XLEN-1:0] instr_d_fetch;
8
9           always_comb begin : instruction_read_comb
10              instr_d_fetch = instruction_memory[pc_q_fetch[$clog2(MEM_SIZE*4) - 1 : 2]];
11          end
12
```

FETCH AŞAMASI

INSTRUCTION READ COMB

- **Listing. Instruction Read Comb Kod Bloğu**
- Bu blokta **Instruction Memory** tanımlaması yapılır. 32 bitlik MEM_SIZE adet komut saklayabilecek bir instruction memory tanımlanmış. MEM_SIZE 1024 olduğunu düşünürsek bu durumda 321024 = 32768 bit = 4096 byte yani 4 KB'lık bir bellek tanımlanmış olur.
- **initial** bloğu sayesinde test komutlarına sahip hex dosyası simülasyon ortamında belleğe yüklenir.
- pc_q_fetch değerine göre Instruction Memory içerisinden ilgili instruction okunur ve **instr_d_fetch** 'e aktarılır.
- **pc_q_fetch[$clog2(MEM_SIZE * 4) - 1 : 2]** ifadesini inceleyecek olursak;
- **(MEM_SIZE * 4)** ifadesi ile öncelikle toplam instruction sayısı ile instruction boyutu(4 byte) çarpılıyor ve adresleme için gerekli olan toplam byte adresi bulunuyor.
- **$clog2** (MEM_SIZE * 4) - 1 clog2 ifadesi ile bu bulunan toplam byte değerinin 2 tabanında logaritması alınıp sonrasında bu değerden 1 çıkarılıyor ve adresleme için gerekli olan bit sayısı hesaplanıyor.
- $clog2(MEM_SIZE * 4) - 1 **: 2** ifadesi ise program counter'ın **alt iki bitini** atmamızı sağlıyor. Çünkü her instruction 4 byte olduğu için program counter'da 4'ün katları olarak ilerler, son iki biti atıp 0 yaptığımız zaman bunu sağlamış oluruz. Eğer instruction'un **belirli bir byte'ına** erişmek istersek o zaman son 2 biti kullanmalıyız.

FETCH AŞAMASI

IF/ID REGISTER

Listing. IF/ID Register

```
// IF/ID Register
logic [XLEN-1:0] instr_q_fetch;
logic [XLEN-1:0] pc_q_fetch_to_decode;
logic            update_q_fetch_to_decode;

always_ff @(posedge clk or negedge rstn) begin : IF_ID_REGISTER
    if(!rstn || is_Flush_IF_ID_Register) begin
        instr_q_fetch <= 0;
        pc_q_fetch_to_decode <= 0;
        update_q_fetch_to_decode <= 0;
    end
    else if(is_Stall_IF_ID_Register) begin
        instr_q_fetch <= instr_q_fetch;
        pc_q_fetch_to_decode <= pc_q_fetch_to_decode;
        update_q_fetch_to_decode <= update_q_fetch_to_decode;
    end
    else begin
        instr_q_fetch <= instr_d_fetch;
        pc_q_fetch_to_decode <= pc_q_fetch;
        update_q_fetch_to_decode <= update_q_fetch;
    end
end

```

FETCH AŞAMASI

IF/ID REGISTER

- **Listing. IF/ID Register Kod Bloğu**
- **IF/ID Register** bloğu fetch aşamasında üretilen sinyallerin, clock döngüsü ile bir sonraki aşamaya(decode) aktarıldığı bloktur.
- Eğer **rstn** sinyali 0 ise **veya** **is_Flush_IF_ID_Register** sinyali 1 ise register içeriği sıfırlanır/temizlenir. Bu aslında pipeline'da yanlış bir instruction'un ilerlemesini engeller.
- Eğer **rstn** sinyali 1 ise **ve** **is_Flush_IF_ID_Register** sinyali 0 ise **ve** **is_Stall_IF_ID_Register** sinyali 1 ise register stall edilir yani içeriği korunur. Bu sayede bir sonraki aşamaya aynı değerler tekrar gönderilmiş olur.
- Eğer yukarıdaki iki durum da sağlanmıyorsa o halde fetch aşamasında üretilen sinyal değerleri register'a yüklenir ve bir sonraki clock döngüsünde diğer aşamaya aktarılır.

DECODE AŞAMASI

DECODE AŞAMASI BLOK DIYAGRAMI

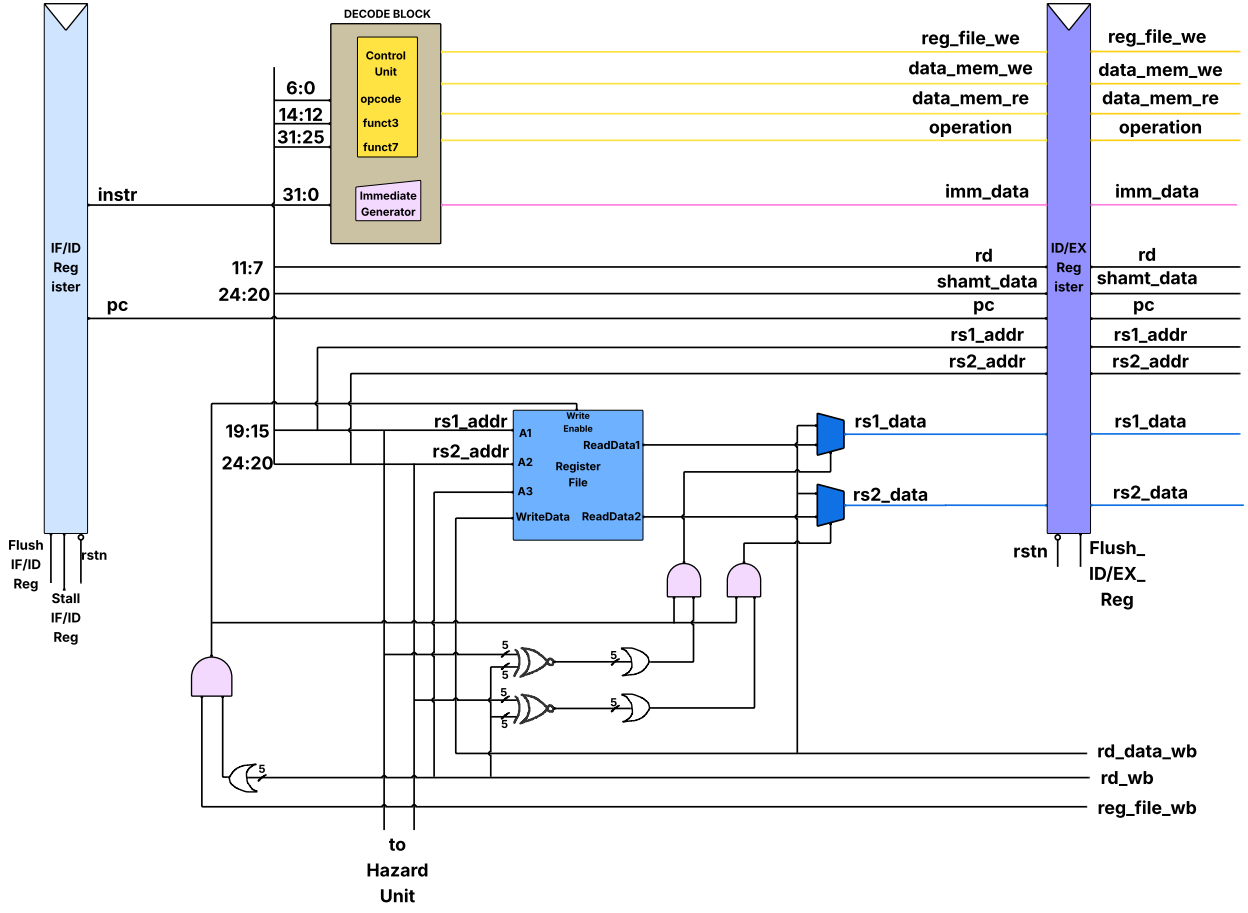

Figure. DECODE Aşaması

DECODE AŞAMASI

DECODE BLOCK

Listing. Decode Block Kod Bloğu (Opcode)

```
always_comb begin : DECODE_BLOCK
    //Başlangıç değer atamaları yapılır
    case(instr_d_decode[6:0])
        //....
        //....
        OpodeJal: begin
            imm_data_d_decode = get_j_type_imm(instr_d_decode);
            operation_d_decode = JAL;
            register_file_write_enable_d_decode = 1;
        end
        //....
        //....
        OpcodeBranch: begin
            imm_data_d_decode = get_b_type_imm(instr_d_decode);
            case(instr_d_decode[14:12])
                F3_BEQ:  operation_d_decode = BEQ;
                //....
                //....
            endcase
        end
        //....

```

DECODE AŞAMASI

DECODE BLOCK

Listing. Decode Block Kod Bloğu (Funct3)

```
                //....
                OpcodeLoad: begin
                    imm_data_d_decode = get_i_type_imm(instr_d_decode);
                    register_file_write_enable_d_decode = 1;
                    data_memory_read_enable_d_decode = 1;
                    case(instr_d_decode[14:12])
                        F3_LB:  operation_d_decode = LB;
                        //....
                            //....
                    endcase
                end
                OpcodeStore: begin
                    imm_data_d_decode = get_s_type_imm(instr_d_decode);
                    data_memory_write_enable_d_decode = 1;
                    case(instr_d_decode[14:12])
                        F3_SB: operation_d_decode = SB;
                        //....
                        //....
                    endcase
                end
                //....

```

DECODE AŞAMASI

DECODE BLOCK

Listing. Decode Block Kod Bloğu (Funct7)

```
//....
OpodeOp: begin
    register_file_write_enable_d_decode = 1;
    case(instr_d_decode[14:12])
        F3_ADD_SUB: begin
            case(instr_d_decode[31:25])
                F7_ADD: operation_d_decode = ADD;
                F7_SUB: operation_d_decode = SUB;
                default: ;
            endcase
        end
        //....
        //....
    endcase
end

```

DECODE AŞAMASI

DECODE BLOCK

- **Decode Block** içerisindeki **Control Unit bloğu** , fetch aşamasından gelen instruction'un çözümlenerek gerekli kontrol sinyallerini üretir, aynı zamanda içerisindeki **Immediate Generator** bloğu farklı türdeki immediate değerlerini(I,J,B,S,U) instruction'dan çıkarır.
- SystemVerilog kodu içerisinde Control Unit bloğu ile Immediate Generator bloğu ayrı ayrı yazılmayıp koda gömülü olarak eklenmiştir, blok diyagramında daha anlaşılabilir olması için ayrı ayrı gösterilmiştir.
- Instruction'un **opcode(6:0)** alanına göre instruction'un hangi operasyon koduna sahip olduğu çözümlenir. Bunun için bir örnek **Listing. Decode Block Kod Bloğu (Opcode)**'da verilmiştir.
- Yalnızca opcode'un yeterli olmadığı durumlarda bu sefer instruction'un **funct3(14:12)** alanına bakılır ve gelen instruction bu şekilde ayrıştırılır. Bunun için bir örnek **Listing. Decode Block Kod Bloğu (Funct3)**'da verilmiştir. OpcodeLoad üzerinden örnek vericek olursak, load işleminin **LB, LH, LW, LBU, LHU** işlemlerinden hangisi olduğunu anlamak için funct3 alanına bakılır.
- Hem opcode hem de funct3 alanlarının birlikte yeterli olmadığı durumlarda ise instruction'un **funct7(31:25)** alanına bakılır ve instruction bu şekilde ayrıştırılır. Bunun için bir örnek **Listing. Decode Block Kod Bloğu (Funct7)**'da verilmiştir. OpodeOp üzerinden örnek vericek olursak, hangi Op işlemi olduğunu anlamak için önce funct3 alanına bakılmış ve F3_ADD_SUB işlemine karar verilmiş, sonrasında nihai işlem **ADD** işlemi mi yoksa **SUB** işlemimi olduğunu anlamak için funct7 alanına bakılmıştır.

DECODE AŞAMASI

REGISTER FILE

Listing. Register File Kod Bloğu

```
// REGISTER FILE
logic [XLEN-1:0] register_file [31:0];

assign rs1_data_d_decode =(register_file_write_enable_d_writeback &&
(rd_d_writeback != 0) && (rd_d_writeback == rs1_addr_d_decode)) ? rd_data_d_writeback : register_file[rs1_addr_d_decode];

assign rs2_data_d_decode =(register_file_write_enable_d_writeback &&
(rd_d_writeback != 0) && (rd_d_writeback == rs2_addr_d_decode)) ? rd_data_d_writeback : register_file[rs2_addr_d_decode];

always_ff @(posedge clk or negedge rstn) begin : REGISTER_FILE_WRITEBACK
    if(!rstn) begin
        for(int i = 0; i <32; i++) begin
            register_file[i] <= 0;
        end
    end
    else if(register_file_write_enable_d_writeback && (rd_d_writeback != 0))
        register_file[rd_d_writeback] <= rd_data_d_writeback;
end

```

- **Listing. Register File Kod Bloğu**
- **Register File** bloğunda register file tanımlaması, register file'dan veri okuma ve register file'a veri yazma işlemi yapılıyor.
- logic [XLEN-1:0] register_file [31:0]; satırı ile 32 adet XLEN(32) bitlik register tanımlanıyor.
- **assign rs1_data_d_decode = register_file[rs1_addr_d_decode];** satırı ile **rs1_addr(5 bit)** değerine karşılık gelen register file içeriği **rs1_data(32 bit)** sinyaline atanıp çıkışa aktarılıyor. rs2 için de aynısı geçerli.
- Register File'da **öncelik sırası okuma işlemine verilir** . Yani eğer aynı clock döngüsü içerisinde hem register file'a yazma işlemi hem de register file'dan okuma işlemi yapılıyorsa öncelik her zaman okuma işleminindir. Bu da şu şekilde yapılır, eğer **writeback aşamasından gelen rd(register destination) değeri 0 değilse, aynı zamanda rd değeri rs1 ya da rs2 adres değerleri ile aynıysa ve son olarak register file write enable sinyali aktif ise** bu durumda okunacak veri yine aynı cycle içerisinde register'a yazılacak verinin ta kendisidir. Yazma işlemi zaman alacağı için aynı clock döngüsünde doğru şekilde okuma yapılamaz, bu nedenle eğer bu şartlar sağlanıyorsa rs1_data veya rs2_data değerlerine register'dan okunan(yanlış değer) değil, **writeback aşamasından gelen ve henüz register'a yazılamamış** veri atanır. Bu şekilde aslında forwarding yapılarak okuma işlemine öncelik verilmiş olunur.
- always_ff bloğu ile register file'a her clock döngüsünde veri yazma işlemi yapılıyor.
- Eğer **rstn** sinyali 0 ise tüm register'lar sıfırlanıyor.
- Eğer rstn sinyali 0 değilse, bu durumda yazma işleminin yapılabilmesi için **register_file_write_enable_d_writeback** sinyalinin 1 olması gerekiyor ve yazılacak register adresi olan **rd_d_writeback** sinyalinin 0 olmaması gerekiyor. Çünkü RISC-V mimarisinde x0 register'ı her zaman 0 değerine sahiptir ve değiştirilemez. Eğer bu koşullar sağlanırsa register file'ın ilgili adresine ilgili veri yazılıyor.

DECODE AŞAMASI

ID/IEX REGISTER

Listing. ID/IEX Register

```
// ID/IEX Register
if(!rstn || is_Flush_ID_IEX_Register) begin
    // rstn sinyali 0 ise veya is_Flush_ID_IEX_Register sinyali 1 ise register temizlenir
end
else begin
    // Diğer durumlarda decode aşamasından gelen sinyaller register'a yüklenip execute aşamasına aktarılır.
end

```

EXECUTE AŞAMASI

EXECUTE AŞAMASI BLOK DIYAGRAMI

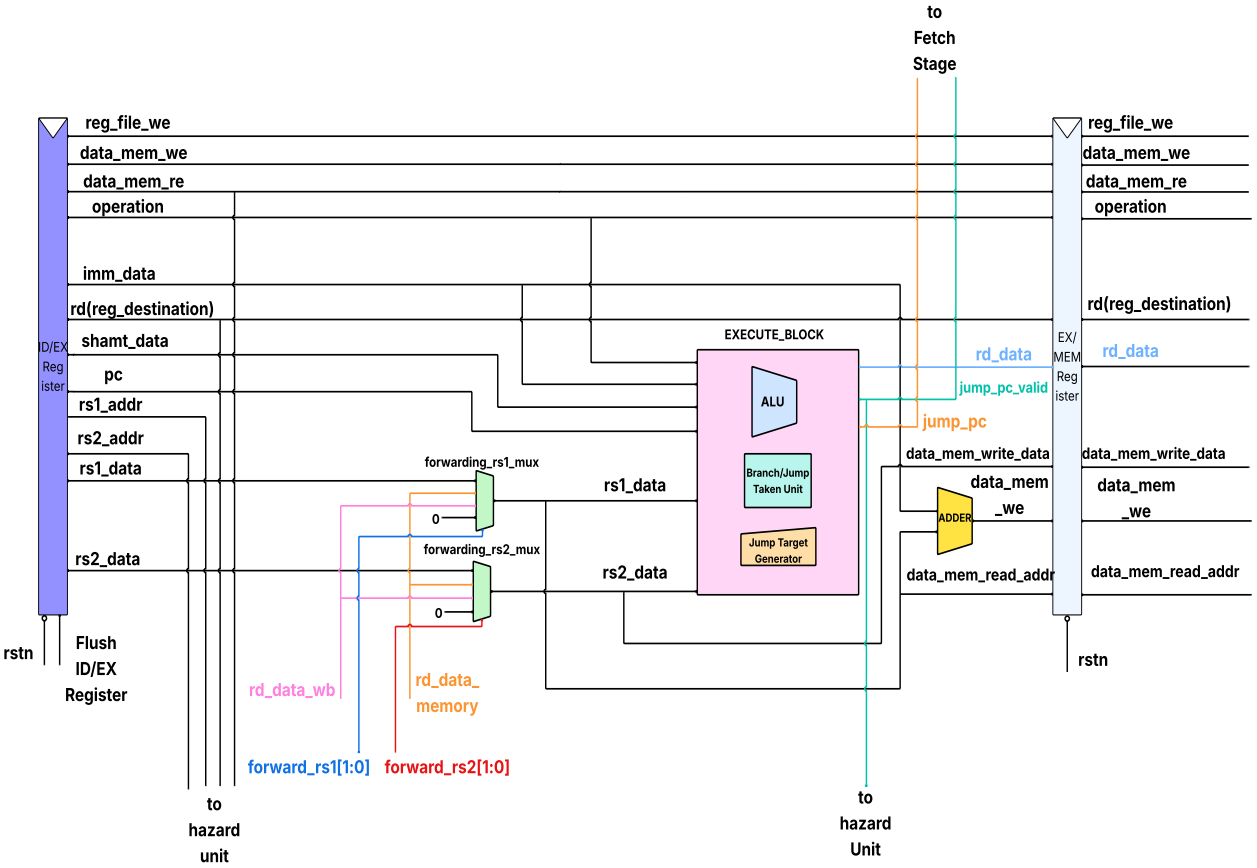

Figure. EXECUTE Aşaması

EXECUTE AŞAMASI

EXECUTE BLOCK

Listing. Execute Block(Branch/Jump/Auipc/Lui)

```
// EXECUTE BLOCK

always_comb begin : EXECUTE_BLOCK
jump_pc_valid_d_execute = 0;
jump_pc_d_execute = 0;
rd_data_d_execute = 0;

case(operation_d_execute)
    LUI:   rd_data_d_execute = imm_data_d_execute;
    AUIPC: rd_data_d_execute = pc_d_execute + imm_data_d_execute;
    JAL: begin
        rd_data_d_execute = pc_d_execute + 4;
        jump_pc_valid_d_execute = 1;
        jump_pc_d_execute = pc_d_execute + imm_data_d_execute;
    end
    JALR: begin
        rd_data_d_execute = pc_d_execute + 4;
        jump_pc_valid_d_execute = 1;
        jump_pc_d_execute = (rs1_data_d_execute + imm_data_d_execute) & ~1;
    end
    BEQ: begin
        if(rs1_data_d_execute == rs2_data_d_execute) begin
            jump_pc_valid_d_execute = 1;
            jump_pc_d_execute = pc_d_execute + imm_data_d_execute;
        end
    end
    BNE: begin
        if(rs1_data_d_execute != rs2_data_d_execute) begin
            jump_pc_valid_d_execute = 1;
            jump_pc_d_execute = pc_d_execute + imm_data_d_execute;
        end
    end

```

EXECUTE AŞAMASI

EXECUTE BLOCK

Listing. Execute Aşaması(Branch/Jump/Auipc/Lui) Devam

```
                BLT: begin
                    if($signed(rs1_data_d_execute) < $signed(rs2_data_d_execute)) begin
                        jump_pc_valid_d_execute = 1;
                        jump_pc_d_execute = pc_d_execute + imm_data_d_execute;
                            end
                end
                BGE: begin
                    if($signed(rs1_data_d_execute) >= $signed(rs2_data_d_execute)) begin
                        jump_pc_valid_d_execute = 1;
                        jump_pc_d_execute = pc_d_execute + imm_data_d_execute;
                    end
                end
                BLTU: begin
                    if(rs1_data_d_execute < rs2_data_d_execute) begin
                        jump_pc_valid_d_execute = 1;
                        jump_pc_d_execute = pc_d_execute + imm_data_d_execute;
                    end
                end
                BGEU: begin
                    if(rs1_data_d_execute >= rs2_data_d_execute) begin
                        jump_pc_valid_d_execute = 1;
                        jump_pc_d_execute = pc_d_execute + imm_data_d_execute;
                    end
                end


```

- **Listing. Execute Block(Branch/Jump/Auipc/Lui)**
- Bu blokta Exeucte Block içindeki **Branch/Jump/AUIPC/LUI** işlemleri gösterilmiştir.
- Branch komutları(BEQ, BNE, BLT, BGE, BLTU, BGEU) incelendiğinde, gerekli karşılaştırma yapılır ve eğer bu karşılaştırma sonucu gerekli koşulu sağlıyorsa **jump_pc_valid_d_execute** sinyali 1 yapılır ve atlanacak adres olan **jump_pc_d_execute** hesaplaması yapılır.
- jump_pc_valid_d_execute ve jump_pc_d_execute sinyaleri **kombinasyonel** sinyallerdir ve fetch aşamasına ex/mem register'ına aktarılmadan direkt olarak iletilirler, bu sinyaller sayesinde fetch aşamasında bir sonraki program counter değeri pc + 4 olarak değil, hesaplanan atlama adresi olarak seçilir.
- Jump komutları(JAL, JALR) incelendiğinde ise bu komutlar branch komutları ile çok benzer çalışır fakat bu komutlarda branch komutları gibi bir **koşul ifadesi** yoktur ve ayrıca ekstra olarak bir sonraki program counter değeri bir register'a kaydedilir.
- JALR komutundaki özel duruma dikkat çekmek gerekir, bu komutta atlanacak adres hesaplanırken **& 1** işlemi ile en son bit 0 yapılarak program counter 2 byte hizalanmış olur. Bu işlemci 32 bitlik komutlardan oluştuğu için aslında bu durum elzem değildir çünkü PC her zaman 4'ün katı olur lakin eğer **compress komutlar** kullanıyor olsaydık o zaman bu durum elzem hale gelecekti.
- AUIPC ve LUI komutları ise herhangi bir atlama işlemi yapmaz, bu komutlarda immediate değerinin üst 20 biti hedef register'a kaydedilir veya mevcut program counter ile toplanıp hedef register'a kaydedilir.
- AUIPC komutu, **PC-Relative Addressing** yapabilmeyi sağlar, bu sayede kod taşınabilir olur yani program hafızada başka bir adres bloğuna taşınsa dahi doğru çalışır. Ayrıca bu komut sayesinde çok büyük adreslere erişmek mümkün hale gelir.
- LUI komutu ise genel olarak büyük sabit değerleri register'lara yazmak için kullanılır. RISC-V komutları 32 bit olduğu için tek bir komut ile 32 bitlik bir sabiti register'a yazamayız. Bu nedenle LUI komutu ile önce 32 bitlik sabitin üst 20 bitini register'a yükler ve alt 12 biti sıfırlamış oluruz, ardından ADDI komutu ile elimizdeki 32 bitlik verinin alt 12 bitini ilgili register'a yükleriz. Bu sayede 32 bitlik bir sabiti iki komut ile register'a yükleyebilmiş oluruz.

EXECUTE AŞAMASI

EXECUTE BLOCK

Listing. Execute Aşaması(ALU)

```
            ADDI:  rd_data_d_execute = $signed(rs1_data_d_execute) + $signed(imm_data_d_execute);
            SLTI:  if($signed(rs1_data_d_execute) < $signed(imm_data_d_execute)) rd_data_d_execute = 1;
            SLTIU: if(rs1_data_d_execute < imm_data_d_execute) rd_data_d_execute = 1;
            XORI:  rd_data_d_execute = rs1_data_d_execute ^ imm_data_d_execute;
            ORI:   rd_data_d_execute = rs1_data_d_execute | imm_data_d_execute;
            ANDI:  rd_data_d_execute = rs1_data_d_execute & imm_data_d_execute;
            SLLI:  rd_data_d_execute = rs1_data_d_execute << shamt_data_d_execute;
            SRLI:  rd_data_d_execute = rs1_data_d_execute >> shamt_data_d_execute;
            SRAI:  rd_data_d_execute = $signed(rs1_data_d_execute) >>> shamt_data_d_execute;
            ADD:   rd_data_d_execute = $signed(rs1_data_d_execute) + $signed(rs2_data_d_execute);
            SUB:   rd_data_d_execute = $signed(rs1_data_d_execute) - $signed(rs2_data_d_execute);
            SLL:   rd_data_d_execute = rs1_data_d_execute << rs2_data_d_execute[4:0];
            SLT:   if($signed(rs1_data_d_execute) < $signed(rs2_data_d_execute)) rd_data_d_execute = 1;
            SLTU:  if(rs1_data_d_execute < rs2_data_d_execute) rd_data_d_execute = 1;
            XOR:   rd_data_d_execute = rs1_data_d_execute ^ rs2_data_d_execute;
            SRL:   rd_data_d_execute = rs1_data_d_execute >> rs2_data_d_execute[4:0];
            SRA:   rd_data_d_execute = $signed(rs1_data_d_execute) >>> rs2_data_d_execute[4:0];
            OR:    rd_data_d_execute = rs1_data_d_execute | rs2_data_d_execute;
            AND:   rd_data_d_execute = rs1_data_d_execute & rs2_data_d_execute;
            default: ;
        endcase
    end

```

EXECUTE AŞAMASI

EXECUTE BLOCK

- **Listing. Execute Aşaması(ALU)**
- Bu kod bloğunda Execute Block içindeki **ALU** işlemleri gösterilmiştir.
- ALU işlemleri aritmetik işlemler ve mantıksal işlemler olarak tanımlanır.

EXECUTE AŞAMASI

RS1/RS2 FORWARDING MUX

Listing. rs1/rs2 Forwarding Mux Kod Bloğu

```
logic [XLEN-1:0] rs1_data_d_execute;
always_comb begin : forwarding_rs1

    if(is_forward_rs1 == NO_FORWARD)
        rs1_data_d_execute = rs1_data_q_decode;
    else if(is_forward_rs1 == FORWARD_MEMORY)
        rs1_data_d_execute = rd_data_d_memory;
    else if(is_forward_rs1 == FORWARD_WRITEBACK)
        rs1_data_d_execute = rd_data_d_writeback;
    else
        rs1_data_d_execute = 0; // beklenmedik durum
end

logic [XLEN-1:0] rs2_data_d_execute;
always_comb begin : forwarding_rs2
    if(is_forward_rs2 == NO_FORWARD)
        rs2_data_d_execute = rs2_data_q_decode;
    else if(is_forward_rs2 == FORWARD_MEMORY)
        rs2_data_d_execute = rd_data_d_memory;
    else if(is_forward_rs2 == FORWARD_WRITEBACK)
        rs2_data_d_execute = rd_data_d_writeback;
    else
        rs2_data_d_execute = 0; // beklenmedik durum
end

```

EXECUTE AŞAMASI

RS1/RS2 FORWARDING MUX

- **Listing. rs1/rs2 Forwarding Mux Kod Bloğu**
- Forwarding mux'ları sayesinde veri bağımlılığı durumlarında gerekli veriler bir önceki aşamalardan alınabilir, bu sayede hazard oluşumu engellenmiş olur.
- Örnek vermek gerekirse, **ADD x1, x2, x3** komutu gelsin ve hemen ardından **SUB x4, x1, x5** komutu gelmiş olsun bu durumda x1 register'ına yazılmacak veri SUB komutunda kullanılacaktır fakat x1 register'ına daha veri yazılmadan SUB komutunda x1 register'ı çağırılmıştır ama x1 register'ında hala eski veri bulunmaktadır. Bu durumda veriyi forward ederek SUB komutunun doğru veriyi kullanmasını sağlamış oluruz.
- Eğer **is_forward_rs1** sinyali **NO_FORWARD** ise rs1 verisi decode aşamasından alınır.
- Eğer is_forward_rs1 sinyali **FORWARD_MEMORY** ise rs1 verisi memory aşamasından alınır.
- Eğer is_forward_rs1 sinyali **FORWARD_WRITEBACK** ise rs1 verisi writeback aşamasından alınır.
- rs2 için de aynı durum geçerlidir.

EXECUTE AŞAMASI

EX/MEM REGISTER

Listing. EX/MEM Register Kod Bloğu

```
// EX/MEM Register
if(!rstn) begin
    // rstn sinyali 0 ise register temizlenir
end
else begin
    // rstn sinyali 1 ise execute aşamasından gelen sinyaller register'a yüklenip memory aşamasına aktarılır.
end

```

MEMORY Aşaması

MEMORY Aşaması Blok Diyagramı

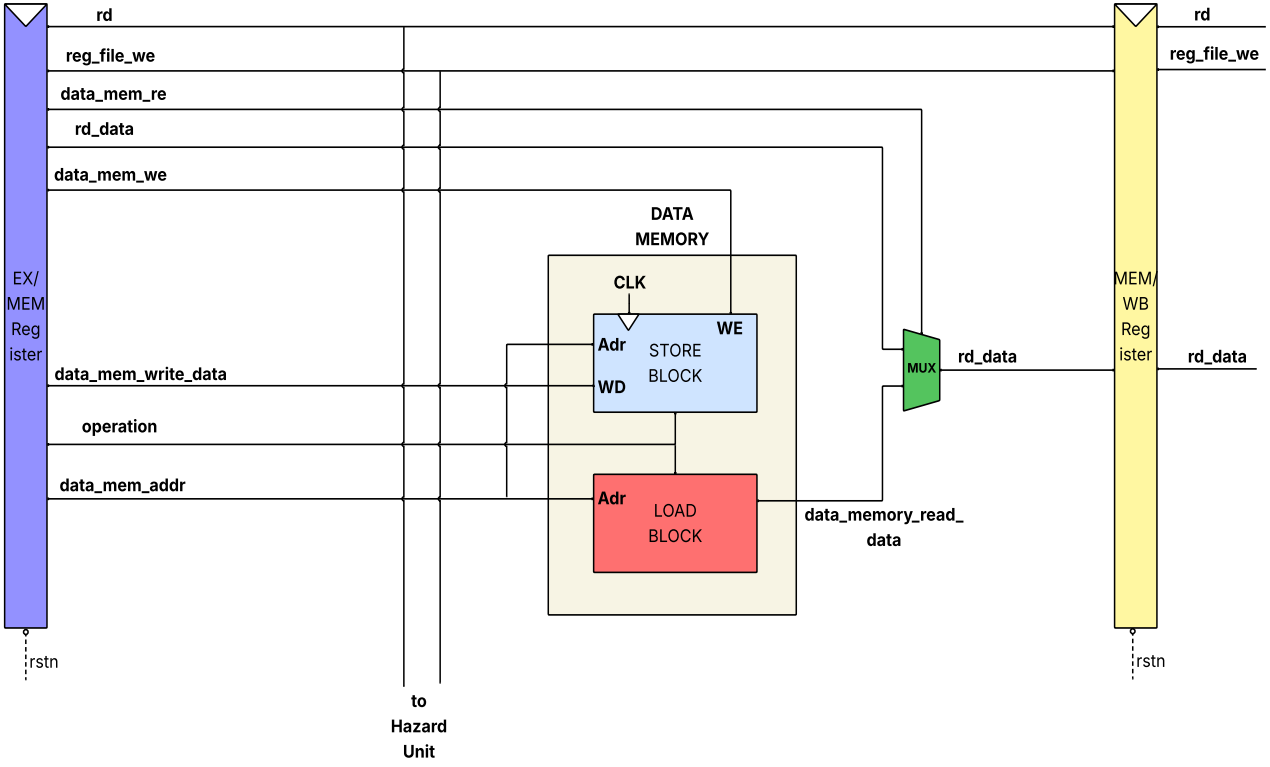

Figure. MEMORY Aşaması

MEMORY AŞAMASI

DATA MEMORY

Listing. Data Memory(Store Block) Kod Bloğu

```
always_ff @(posedge clk or negedge rstn) begin : STORE_BLOCK
    if(!rstn)
            ;
    else if(data_memory_write_enable_d_memory) begin
        case(operation_d_memory)
            SB:
                case(data_memory_write_address_d_memory[1:0])
                    2'b00: data_memory[data_memory_write_address_d_memory[$clog2(MEM_SIZE)+1:2]][7:0]   <= data_memory_write_data_d_memory[7:0];
                    2'b01: data_memory[data_memory_write_address_d_memory[$clog2(MEM_SIZE)+1:2]][15:8]  <= data_memory_write_data_d_memory[7:0];
                    2'b10: data_memory[data_memory_write_address_d_memory >> 2][23:16] <= data_memory_write_data_d_memory[7:0];
                    2'b11: data_memory[data_memory_write_address_d_memory >> 2][31:24] <= data_memory_write_data_d_memory[7:0];
                endcase
            SH:
                case(data_memory_write_address_d_memory[1])
                    1'b0: data_memory[data_memory_write_address_d_memory >> 2][15:0] <= data_memory_write_data_d_memory[15:0];
                    1'b1: data_memory[data_memory_write_address_d_memory[$clog2(MEM_SIZE)+1:2]][31:16] <= data_memory_write_data_d_memory[15:0];
                endcase
            SW: data_memory[data_memory_write_address_d_memory[$clog2(MEM_SIZE)+1:2]] <= data_memory_write_data_d_memory;
            //  data_memory[data_memory_write_address_d_memory >> 2] <= data_memory_write_data_d_memory; şeklinde de olur.
            default: ;
        endcase
    end
end

```

- **Listing. Data Memory(Store Block) Kod Bloğu**
- Store Block içerisinde data memory'e veri yazılması gerçekleştirilmektedir. **SB(Store Byte)** , **SH(Store Halfword)** ve **SW(Store Word)** olmak üzere üç farklı store işlemi vardır. Register içindeki 32 bitlik verinin 31:2 bitleri ile hedef satırı ifade eder, 1:0 bitleri ise hedef satırın hangi byte'ına yazılacağına karar verir.
- **SB** işleminde, yazılacak verinin yalnızca **alt 8 biti(0. byte)** kullanılır. Bu 8 bit'in yazılacağı satır gelen adresin **[31:2]** bitleri ile belirlenirken, adresin **[1:0]** bitleri ile yazılacak verinin hangi byte'a yazılacağı seçilir.
  - **00** $\rightarrow$ Byte 0 seçilir. **01** $\rightarrow$ Byte 1 seçilir.
  - **10** $\rightarrow$ Byte 2 seçilir. **11** $\rightarrow$ Byte 3 seçilir.
- **SH** işleminde, yazılacak verinin yalnızca **alt 16 biti(0. ve 1. byte)** kullanılır. Bu 16 bit'in yazılacağı satır gelen adresin **[31:2]** bitleri ile belirlenirken, adresin **[1]** biti ile yazılacak verinin hangi halfword'a yazılacağı seçilir.
  - **0** $\rightarrow$ Halfword 0 (Byte 0 ve Byte 1) seçilir.
  - **1** $\rightarrow$ Halfword 1 (Byte 2 ve Byte 3) seçilir.
- **SW** işleminde data memory içerisinde işaret edilen adresin tamamına yazılacak verinin **tüm 32 biti(4 byte)** , yazılır. Hangi adres satırına yazılacağı ise yine adresin **[31:2]** bitleri ile belirlenir.

MEMORY AŞAMASI

DATA MEMORY

Listing. Data Memory(Load Block) Kod Bloğu

```
always_comb begin : LOAD_BLOCK
        data_memory_read_data_d_memory = 0;

        case(operation_d_memory)
            LB:
                case(data_memory_read_address_d_memory[1:0])
                    2'b00: data_memory_read_data_d_memory = {{24{data_memory[addr_index][7]}}, data_memory[addr_index][7:0]};
                    2'b01: data_memory_read_data_d_memory = {{24{data_memory[addr_index][15]}}, data_memory[addr_index][15:8]};
                    2'b10: data_memory_read_data_d_memory = {{24{data_memory[addr_index][23]}}, data_memory[addr_index][23:16]};
                    2'b11: data_memory_read_data_d_memory = {{24{data_memory[addr_index][31]}}, data_memory[addr_index][31:24]};
                endcase
            LH:
                case(data_memory_read_address_d_memory[1])
                    1'b0: data_memory_read_data_d_memory = {{16{data_memory[addr_index][15]}}, data_memory[addr_index][15:0]};
                    1'b1: data_memory_read_data_d_memory = {{16{data_memory[addr_index][31]}}, data_memory[addr_index][31:16]};
                endcase
            LW:  data_memory_read_data_d_memory = data_memory[addr_index];
            LBU:
                case(data_memory_read_address_d_memory[1:0])
                    2'b00: data_memory_read_data_d_memory = {24'b0, data_memory[addr_index][7:0]};
                    2'b01: data_memory_read_data_d_memory = {24'b0, data_memory[addr_index][15:8]};
                    2'b10: data_memory_read_data_d_memory = {24'b0, data_memory[addr_index][23:16]};
                    2'b11: data_memory_read_data_d_memory = {24'b0, data_memory[addr_index][31:24]};
                endcase
            LHU:
                case(data_memory_read_address_d_memory[1])
                    1'b0: data_memory_read_data_d_memory = {16'b0, data_memory[addr_index][15:0]};
                    1'b1: data_memory_read_data_d_memory = {16'b0, data_memory[addr_index][31:16]};
                endcase
            default: ;
        endcase
    end

```

- **Listing. Data Memory(Load Block) Kod Bloğu**
- Load Block içerisinde data memory'den veri okunması gerçekleştirilmektedir. **LB(Load Byte)** , **LH(Load Halfword)** , **LW(Load Word)** , **LBU(Load Byte Unsigned)** ve **LHU(Load Halfword Unsigned)** olmak üzere beş farklı load işlemi vardır.
- **LB** işleminde gelen adres değerinin **[31:2]** bitleri data memory'nin ilgili satırı işaret ederken, **[**1:0] bitleri ise o satırın hangi byte'ının okunması gerektiğini gösteren offset değerleridir. Bu offset değerine göre ilgili satırın hangi byte'ının okunacağına kadar verilir. Okunan 8 bit'in MSB biti ile işaretli genişletme yapılır.
  - **00** → Byte 0 okunur. **01** → Byte 1 okunur.
  - **10** → Byte 2 okunur. **11** → Byte 3 okunur.
- **LH** işleminde gelen adres değerinin **[31:2]** bitleri data memory'nin ilgili satırı işaret ederken, **[1]** biti ise o satırın hangi halfword'unun okunması gerektiğini gösteren offset değeridir. Bu offset değerine göre ilgili satırın hangi halfword'unun okunacağına kadar verilir. Okunan 16 bit'in MSB biti ile işaretli genişletme yapılır.
  - **0** → Halfword 0 (Byte 0 ve Byte 1) okunur.
  - **1** → Halfword 1 (Byte 2 ve Byte 3) okunur.
- **LW** işleminde gelen adres değerinin **[31:2]** bitleri data memory'nin ilgili satırını işaret eder.LW işleminde ilgili satırın **tüm 32 biti(4 byte)** okunur.
- **LBU** işleminde gelen adres değerinin **[31:2]** bitleri data memory'nin ilgili satırı işaret ederken, **[**1:0] bitleri ise o satırın hangi byte'ının okunması gerektiğini gösteren offset değerleridir. Bu offset değerine göre ilgili satırın hangi byte'ının okunacağına kadar verilir.LB işleminden farkı ise okunan 8 bit'in MSB biti yerine **0(zero)** ile işaretsiz genişletme yapılır.
  - **00** → Byte 0 okunur. **01** → Byte 1 okunur.
  - **10** → Byte 2 okunur. **11** → Byte 3 okunur.
- **LHU** işleminde gelen adres değerinin **[31:2]** bitleri data memory'nin ilgili satırı işaret ederken, **[1]** biti ise o satırın hangi halfword'unun okunması gerektiğini gösteren offset değeridir. Bu offset değerine göre ilgili satırın hangi halfword'unun okunacağına kadar verilir.LH işleminden farkı ise okunan 16 bit'in MSB biti yerine **0(zero)** ile işaretsiz genişletme yapılır.
  - **0** → Halfword 0 (Byte 0 ve Byte 1) okunur.
  - **1** → Halfword 1 (Byte 2 ve Byte 3) okunur.

MEMORY AŞAMASI

RD_DATA SEÇIM MUX'U

Listing. rd data Seçim Mux'u Kod Bloğu

```
1          assign rd_data_d_memory = data_memory_read_enable_d_memory ? data_memory_read_data_d_memory : rd_data_q_execute;
2
```

- Bu yapıda eğer **data_memory_read_enable** sinyali **1** ise bu durumda gelen instruction load instruction'udur ve mux'un çıkışına data memory'den okunan veri atanır, aksi durumda execut aşamasından gelen rd data değeri çıkışa atanır.

MEMORY AŞAMASI

MEM/WB REGISTER

Listing. MEM/WB Register Kod Bloğu

```
// MEM/WB Register
if(!rstn) begin
    // rstn sinyali 0 ise register temizlenir
end
else begin
    // rstn sinyali 1 ise memory aşamasından gelen sinyaller register'a yüklenip writeback aşamasına aktarılır.
end

```

WRITEBACK Aşamasi

WRITEBACK Aşamasi Blok Diyagrami

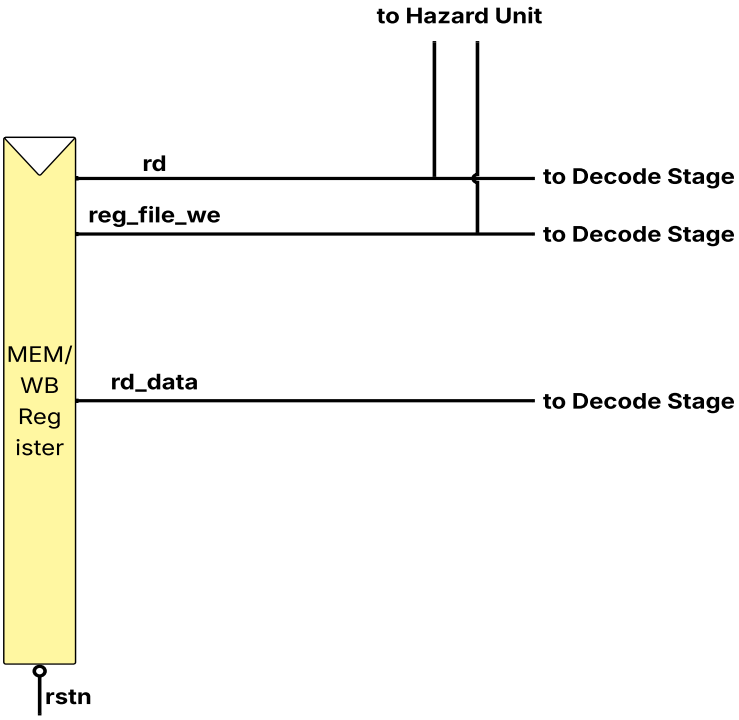

Figure. WRITEBACK Aşaması

WRITEBACK AŞAMASI

WRITEBACK AŞAMASI BLOK DIYAGRAMI

- Writeback aşamasında herhangi bir operasyon bulunmamaktadır, lakin bu aşamada memory aşamasından gelen veriler register file içerisine yazılmak için hazırlanır.
- Writeback aşamasında herhangi bir operasyon yapılmıyor olması onu önemsiz yapmaz, aksine pipeline'ın doğru işleyebilmesi için writeback aşaması olmazsa olmazdır.
- Writeback aşaması olmazsa eğer veri memory aşamasından geliyorsa, veri okunur okunmaz yazılmak zorundadır bu da mümkün olan bir durum değildir çünkü okuma işlemi zaman alan bir işlemdir.
- Writeback aşaması ayrıca hazard yönetimini de kolaylaştırır. Writeback aşaması olmazsa çok daha karmaşık bir forwarding yapısı gerekir ve ayrıca load-use hazard'ları yönetmek çok daha zorlaşır.
- Writeback aşaması olmazsa bazı komutların sonucu direkt execute aşamasında register file'a hemen yazılmak istenir bu da memory aşamasından da register file'a yazılacak veriler gelebileceği için register file'ın yazma portunun sıkışmasına sebep olur hatta çakışmalar olacağı için büyük problemler ortaya çıkar.
- Writeback aşaması aynı zamanda pipeline'ın düzenli olmasını sağlar ve her aşamanın yalnızca kendi işini yapmasını sağlar. Ayrıca clock frekansını artırarak performansı artırır.

Hazard Unit

Hazard Unit Blok Diyagrami

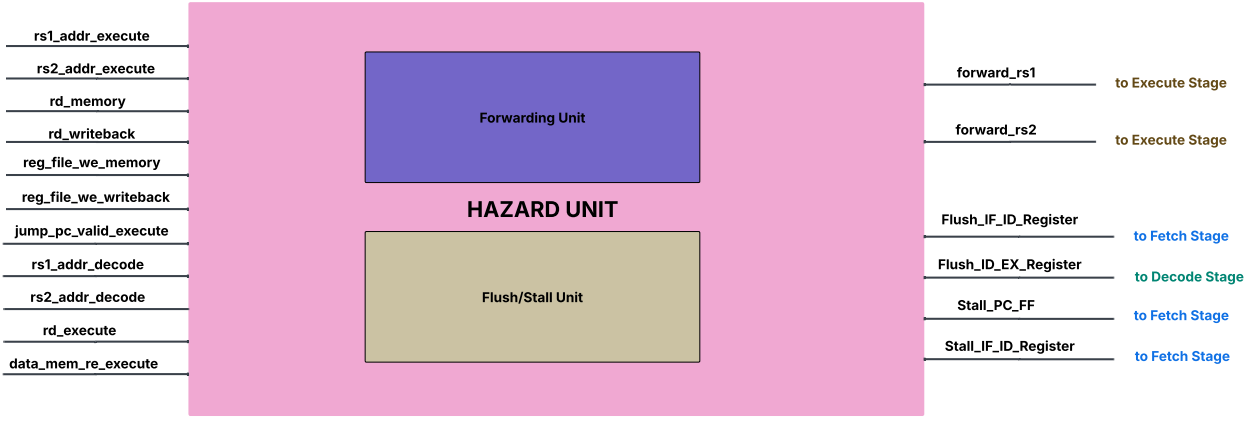

Figure. Hazard Unit

Hazard Unit

Forwarding Unit

Listing. Forwarding Unit Kod Bloğu

```
        // FORWARDING
        Forward_Type_enum is_forward_rs1;
        Forward_Type_enum is_forward_rs2;

        always_comb begin : FORWARDING_RS1
            is_forward_rs1 = NO_FORWARD;

            if((rs1_addr_d_execute == rd_d_memory) && (rd_d_memory != 0) && register_file_write_enable_d_memory)
                is_forward_rs1 = FORWARD_MEMORY;
            else if((rs1_addr_d_execute == rd_d_writeback) && (rd_d_writeback != 0) && register_file_write_enable_d_writeback)
                is_forward_rs1 = FORWARD_WRITEBACK;
            else ;

        end

        always_comb begin : FORWARDING_RS2
            is_forward_rs2 = NO_FORWARD;

            if((rs2_addr_d_execute == rd_d_memory) && (rd_d_memory != 0) && register_file_write_enable_d_memory)
                is_forward_rs2 = FORWARD_MEMORY;
            else if((rs2_addr_d_execute == rd_d_writeback) && (rd_d_writeback != 0) && register_file_write_enable_d_writeback)
                is_forward_rs2 = FORWARD_WRITEBACK;
            else ;
        end

```

- **Listing. Forwarding Unit Kod Bloğu**
- Forwarding Unit, **Data Hazard** 'ların çözümünü sağlar.
- Forwarding unit, bir instruction'ın execute aşamasındayken kullanacağı verilerin kendisinden önce gelen insturction'ların sonuçlarına bağımlı olduğu fakat o instruction'ların sonuçlarının henüz register file'a yazılmadığı durumlarda devreye girer ve gerekli verileri register file'a yazılmasını beklemeksizin bir execute aşamasına aktararak veri bağımlılığını çözer.
- Eğer execute aşamasında kullanılan rs(register source) adresi, memory aşamasındaki rd(register destination) ile aynıysa, rd(register destination) **sıfır değilse** ve register file write enable sinyali aktif ise **memory aşamasından** forwarding yapılır.
- Eğer execute aşamasında kullanılan rs(register source) adresi, writeback aşamasındaki rd(register destination) ile aynıysa, rd(register destination) **sıfır değilse** ve register file write enable sinyali aktif ise **writeback aşamasından** forwarding yapılır.

HAZARD UNIT

FLUSH/STALL UNIT

Listing. Flush/Stall Unit Kod Bloğu

```
        // FLUSH/STALL
        logic is_Flush_IF_ID_Register;
        logic is_Flush_ID_IEX_Register;

        logic is_Stall_PC_FF;
        logic is_Stall_IF_ID_Register;

        always_comb begin : FLUSH_STALL_BLOCK
            is_Flush_IF_ID_Register = 0;
            is_Flush_ID_IEX_Register = 0;

            is_Stall_PC_FF = 0;
            is_Stall_IF_ID_Register = 0;

            if(jump_pc_valid_d_execute) begin
                is_Flush_IF_ID_Register = 1;
                is_Flush_ID_IEX_Register = 1;

                is_Stall_PC_FF = 0;
                is_Stall_IF_ID_Register = 0;
            end
            else if(((rs1_addr_d_decode == rd_d_execute) || (rs2_addr_d_decode == rd_d_execute)) && (rd_d_execute != 0) &&
    data_memory_read_enable_d_execute) begin
                is_Flush_IF_ID_Register = 0;
                is_Flush_ID_IEX_Register = 1;

                is_Stall_PC_FF = 1;
                is_Stall_IF_ID_Register = 1;
            end
            else ;
        end

```

- **Listing. Flush/Stall Unit Kod Bloğu**
- Bazı hazard'lar(**Control Hazards ve Load-Use Hazards gibi**) forwarding yapılarak çözülemez bu durumda pipeline'ı **flush(temizleme)** veya **stall(dondurma)** yapılarak hazard'lar çözülür.
- Control hazard'ı detaylandırmak gerekirse, eğer execute aşamasında bir talimatın **jump veya branch** olduğu tespit edilirse, bu durumda fetch ve decode aşamasındaki talimatlar geçersiz talimatlar olabilir çünkü program counter olması gerekenden farklı bir adrese atlatılmıştır bu nedenle fetch ve decode aşaması içerisindeki talimatlar flush edilir yani temizlenir.
- Load-use hazard'ı detaylandırmak gerekirse, eğer decode aşamasındaki bir talimatın kaynak register'larından herhangi biri(rs1, rs2), execute aşamasındaki talimatın hedef register'ı(rd) ile aynıysa ve execute aşamasındaki talimat bir **load** talimatı ise bu durumda veri forward edilemez çünkü decode aşamasındaki talimat execute aşamasına geçtiğinde execute aşamasındaki load talimatı memory aşamasına geçer fakat **data memory içerisindeki fiziksel gecikmelerden dolayı** henüz data memory'den veriyi okuyamaz bu nedenle forwarding yapılsa bile yanlış veri gönderilir. Bu durum, decode aşamasındaki talimatın **1 döngü boyunca beketilmesi(stall)** ile çözülür, bu durumda execute aşamasındaki talimat veriyi okur ve memory aşamasına geçer ve decode aşamasındaki veri ise hala decode aşamasındadır, bir sonraki döngüde ise memory aşamasındaki data memory'den okunan veri writeback aşamasına geçer decode aşamasında bekleyen talimat ise execute aşamasına geçer, bu durumdan sonra artık load-use hazard çözülmüştür ve data hazard oluşur, data hazard ise writeback aşamasından execute aşamasına forwarding yapılarak çözülür.

RISC-V Pipelined İşlemci Blok Diyagramı(Tam Hali)

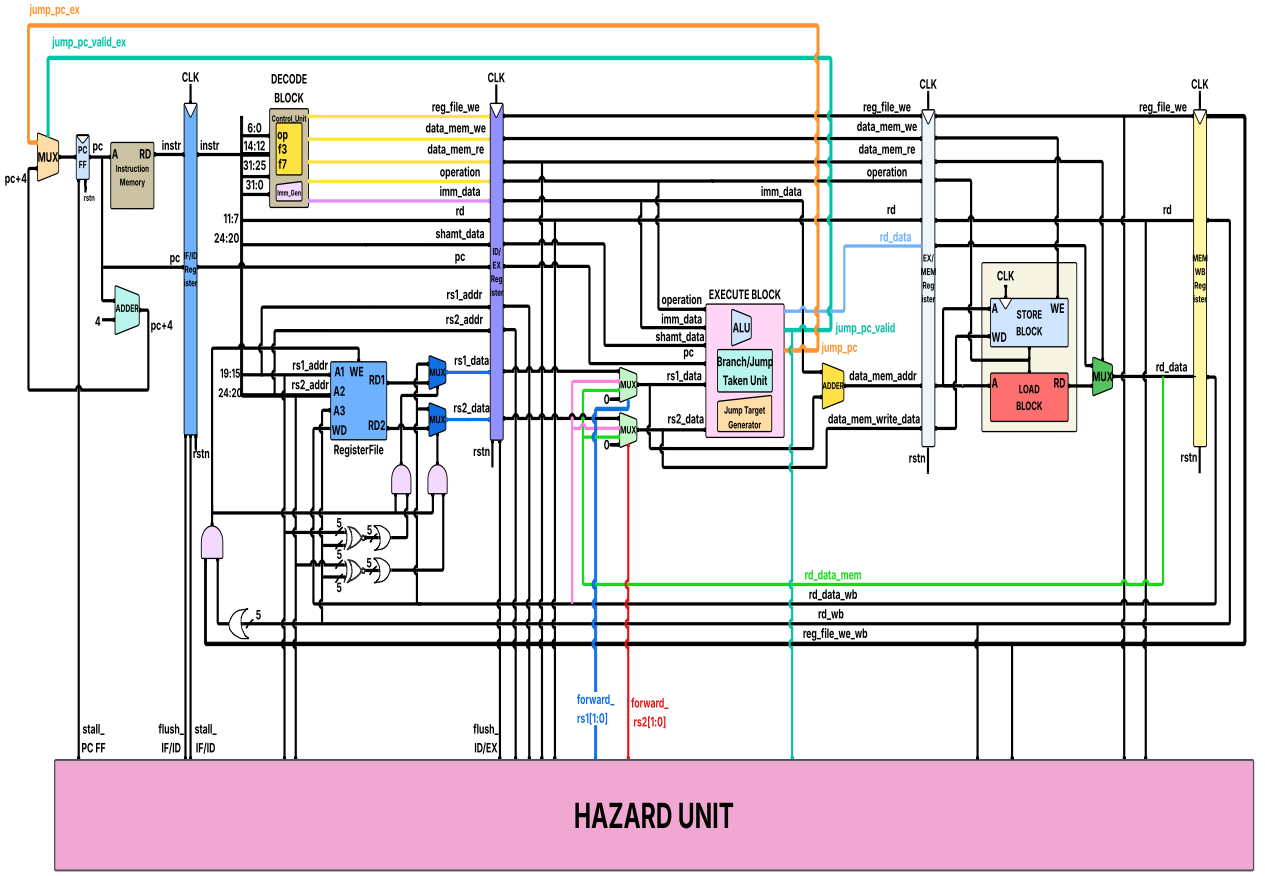

Figure. RISC-V Pipelined İşlemci Blok Diyagramı(Tam Hali)

Part II

Instruction Bazli Veri Yolu Analizi(Data Flow / Datapath Analysis)

R-TYPE INSTRUCTIONS

R-TYPE INSTRUCTION NO FORWARD

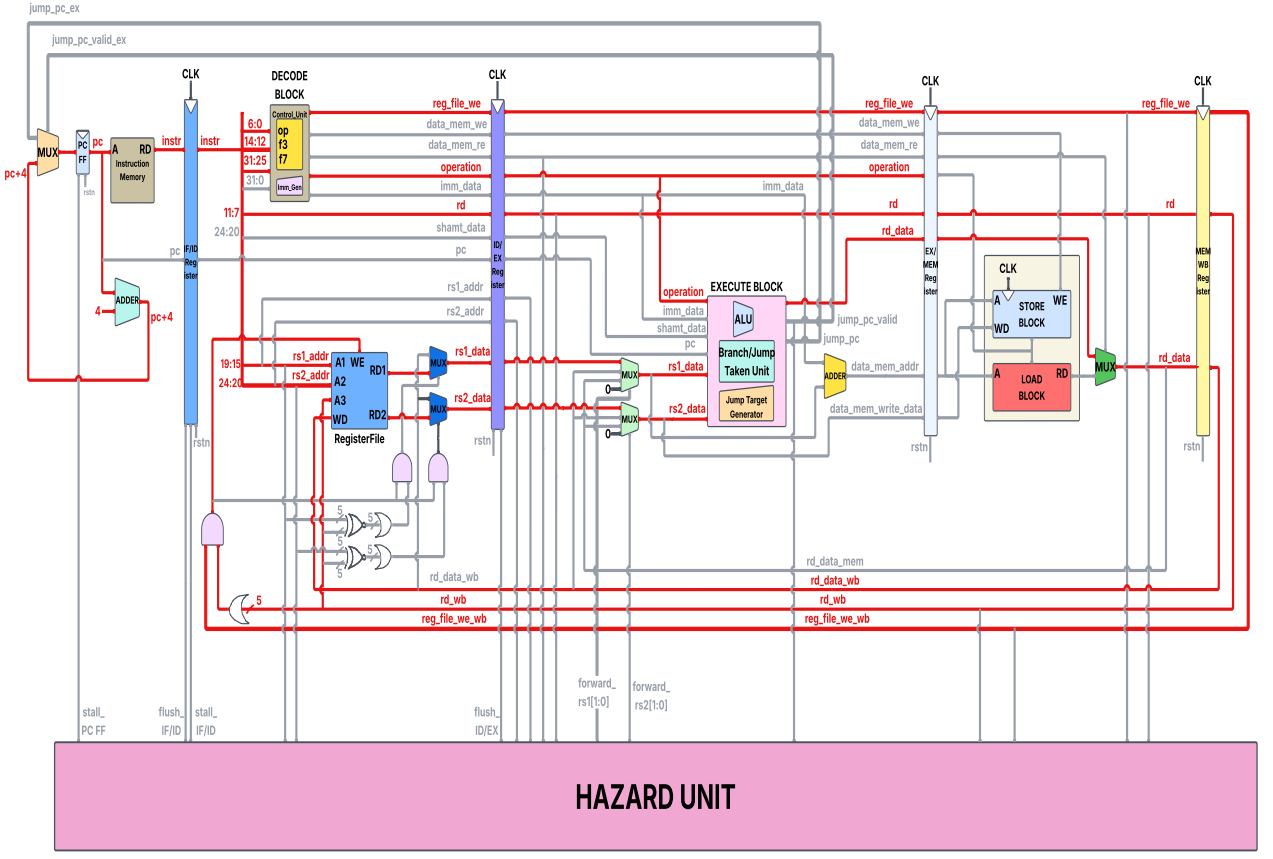

Figure. R-TYPE No Forward Instruction Data Flow

R-TYPE INSTRUCTIONS

R-TYPE INSTRUCTION FORWARD MEMORY

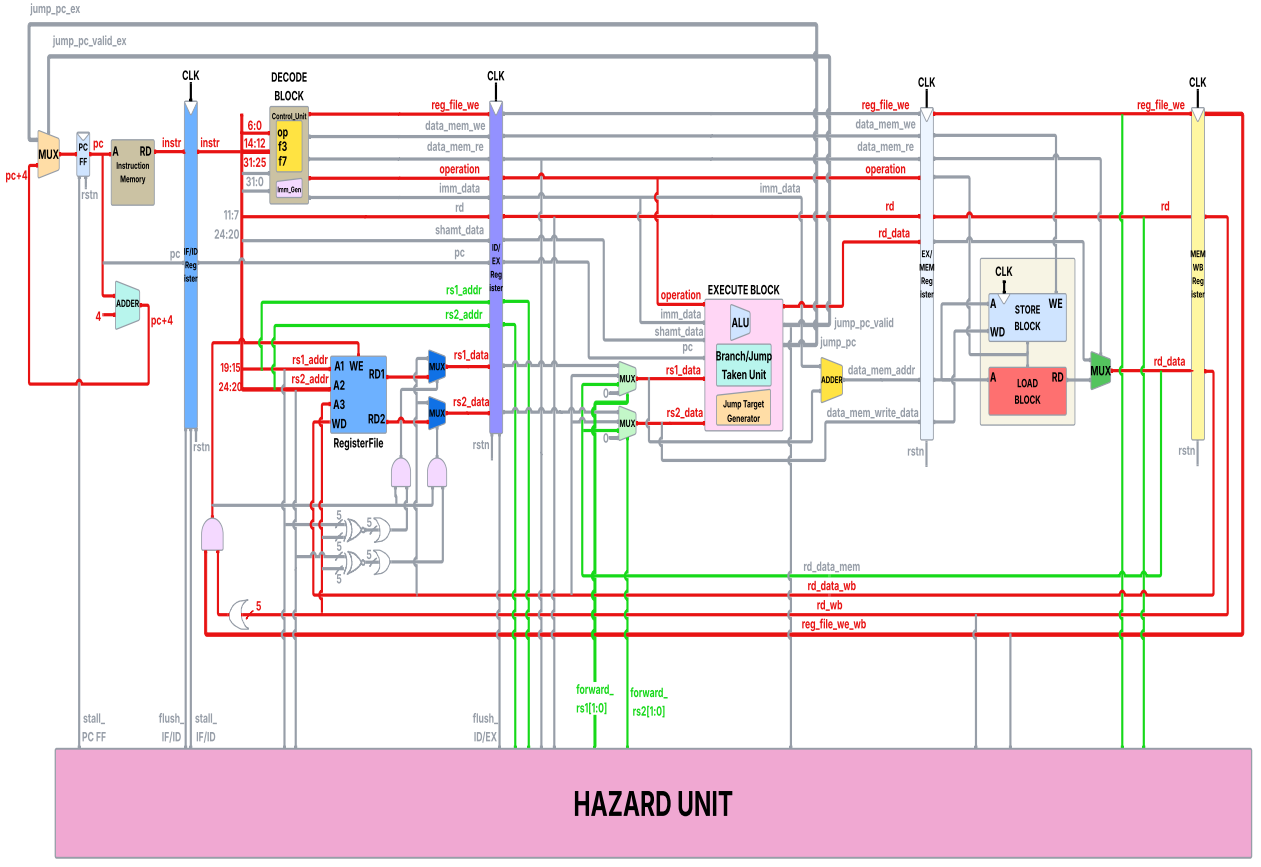

Figure. R-TYPE Forward Memory Instruction Data Flow

R-TYPE INSTRUCTIONS

R-TYPE INSTRUCTION FORWARD WRITEBACK

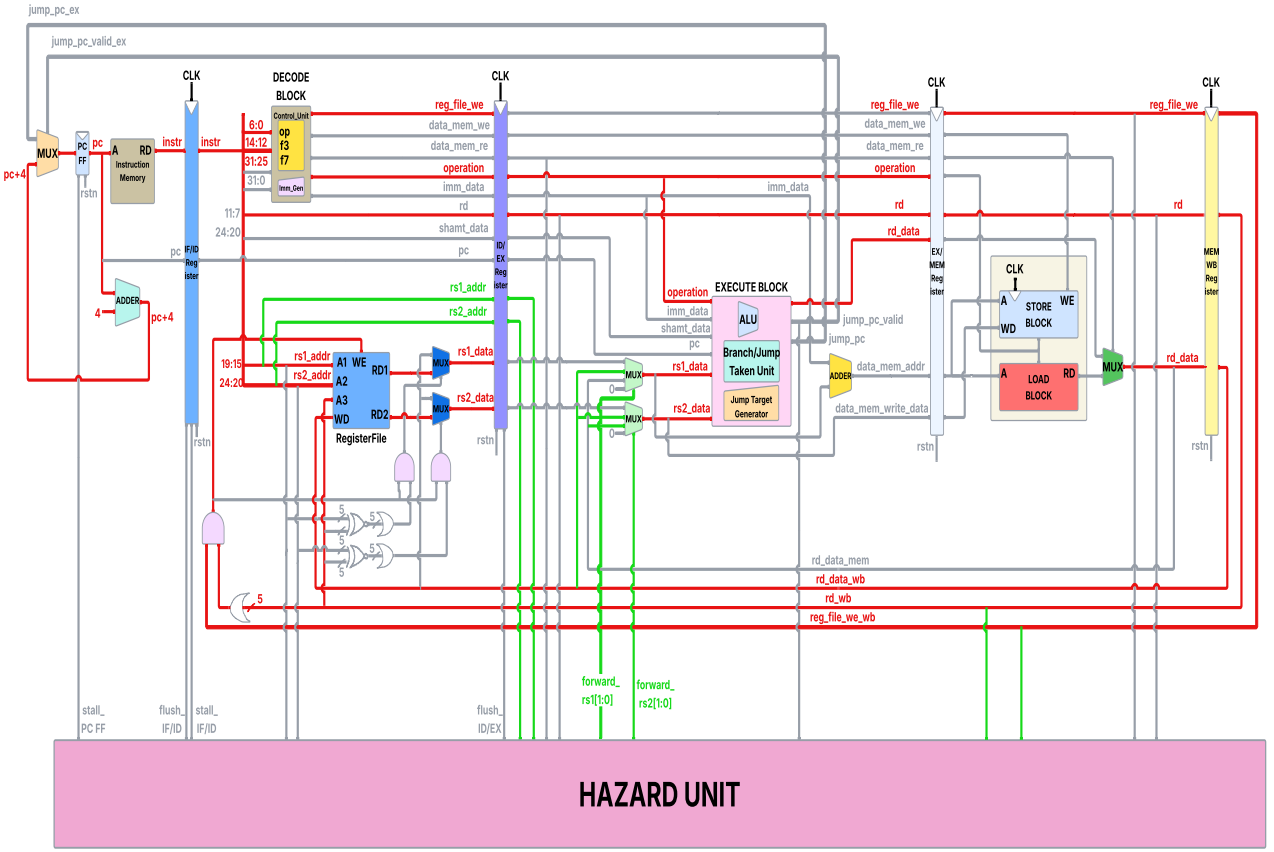

Figure. R-TYPE Forward Writeback Instruction Data Flow

I-TYPE

Load Instruction

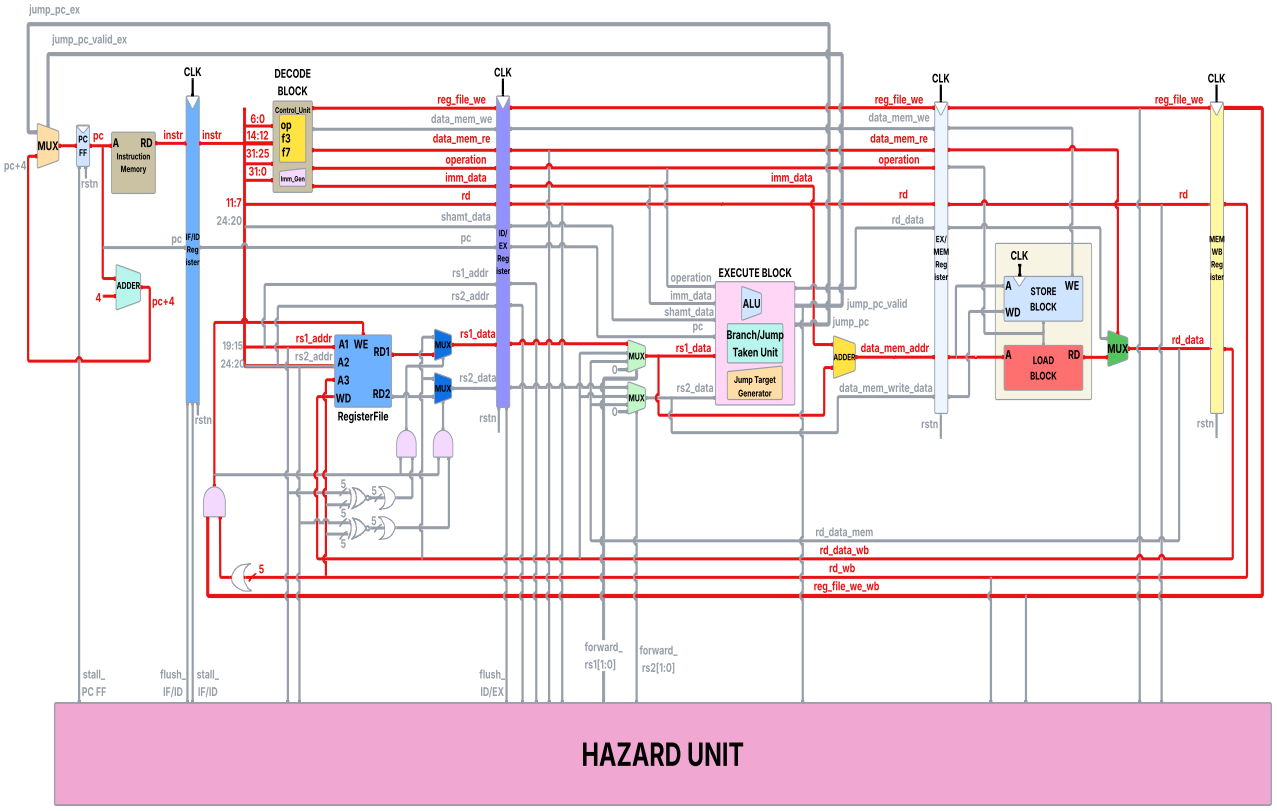

Figure. I-TYPE Load Instruction Data Flow

I-TYPE

Load-Use Hazard

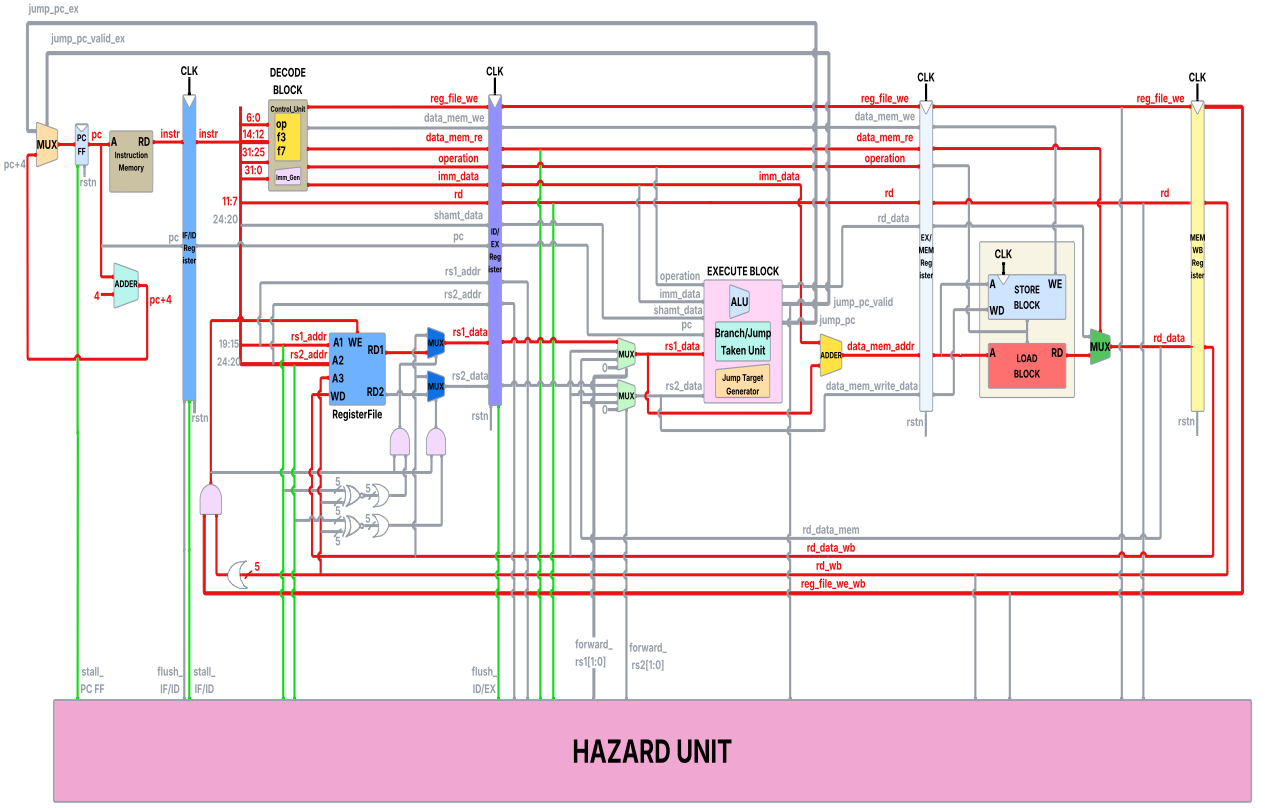

Figure. Load-Use Hazard Instruction Data Flow

I-TYPE

IMMEDIATE OPERATION INSTRUCTION

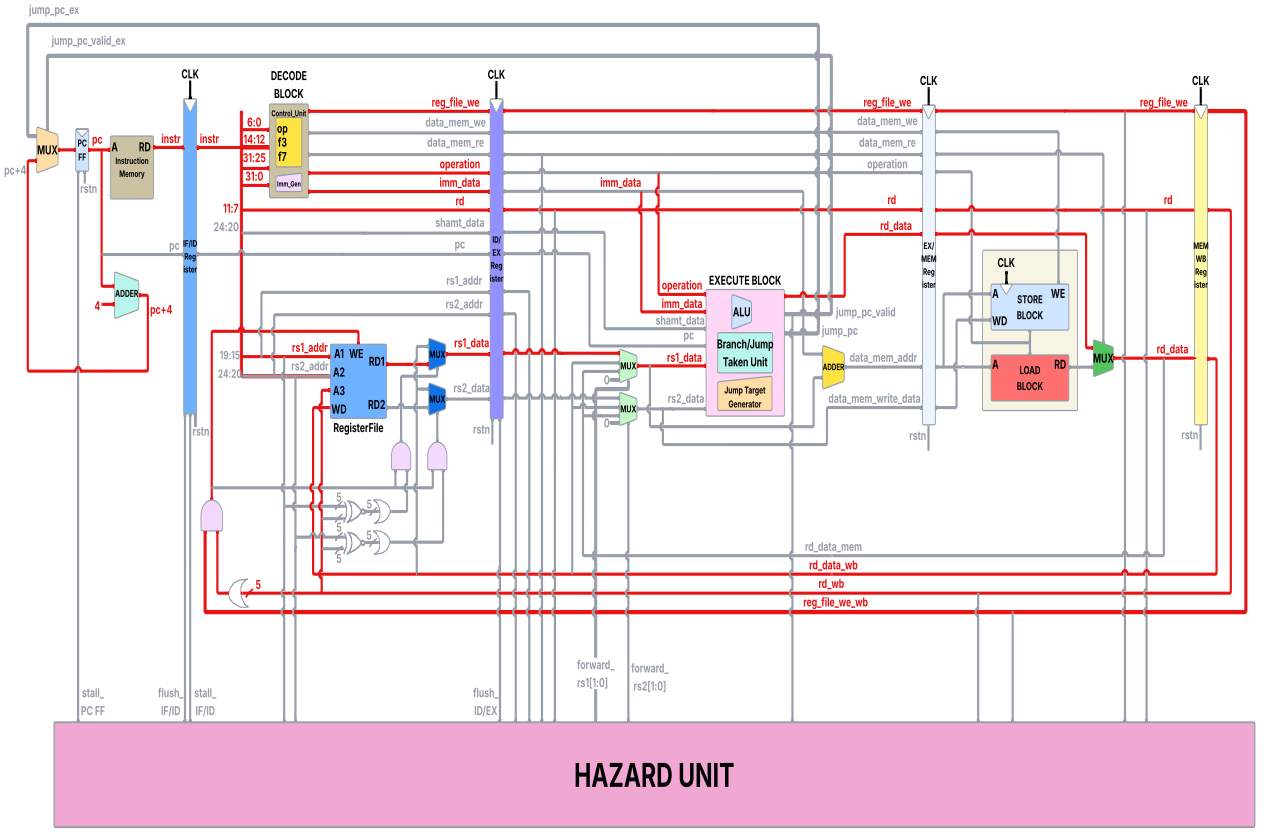

Figure. I-TYPE OperationImm Instruction Data Flow

S-TYPE INSTRUCTION

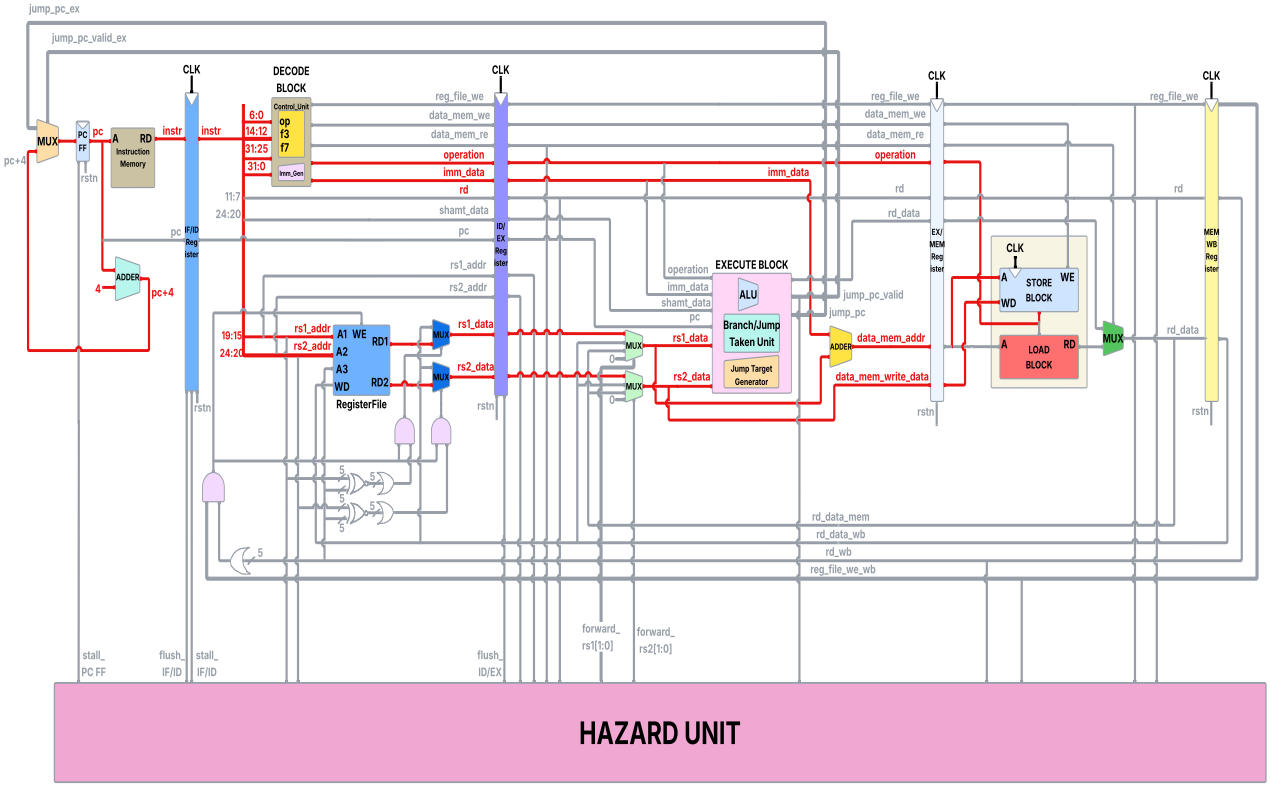

Figure. S-TYPE Instruction Data Flow

B-TYPE INSTRUCTION

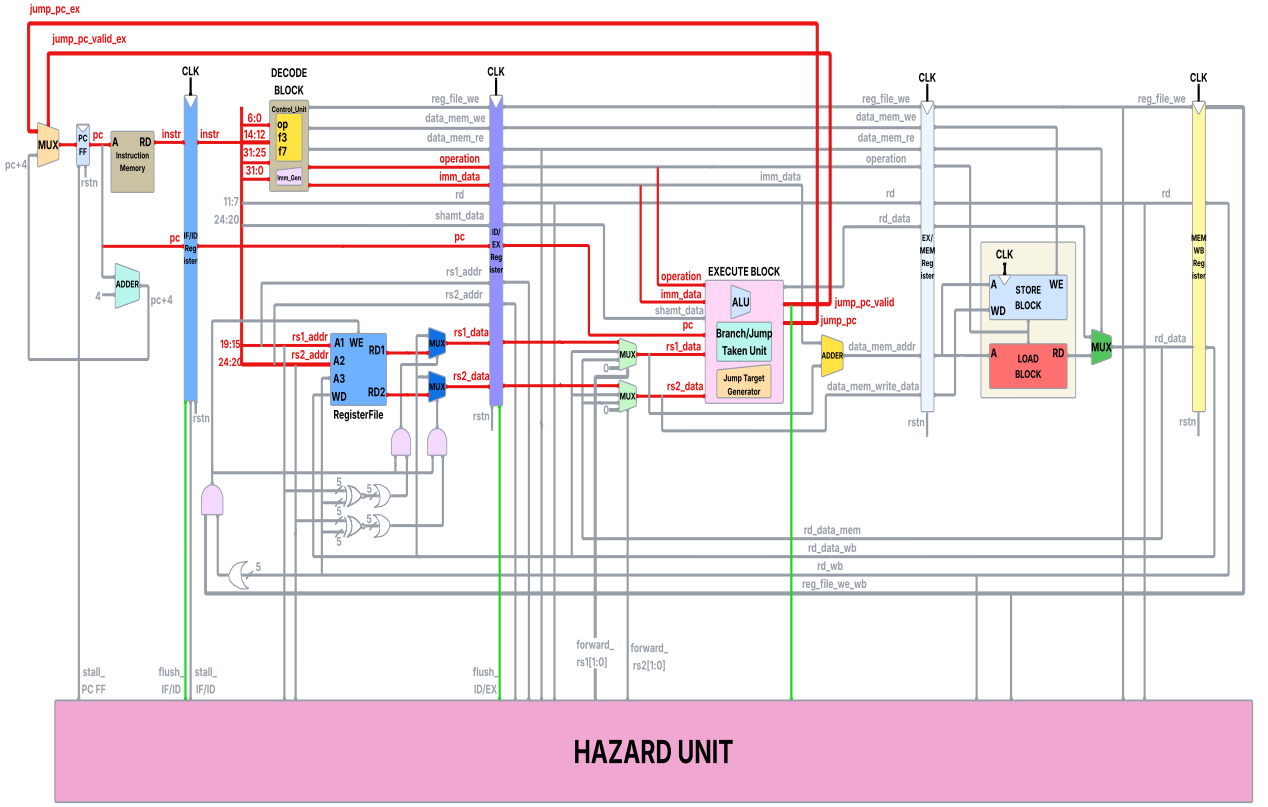

Figure. B-TYPE Instruction Data Flow

J-TYPE INSTRUCION

JAL INSTRUCTION

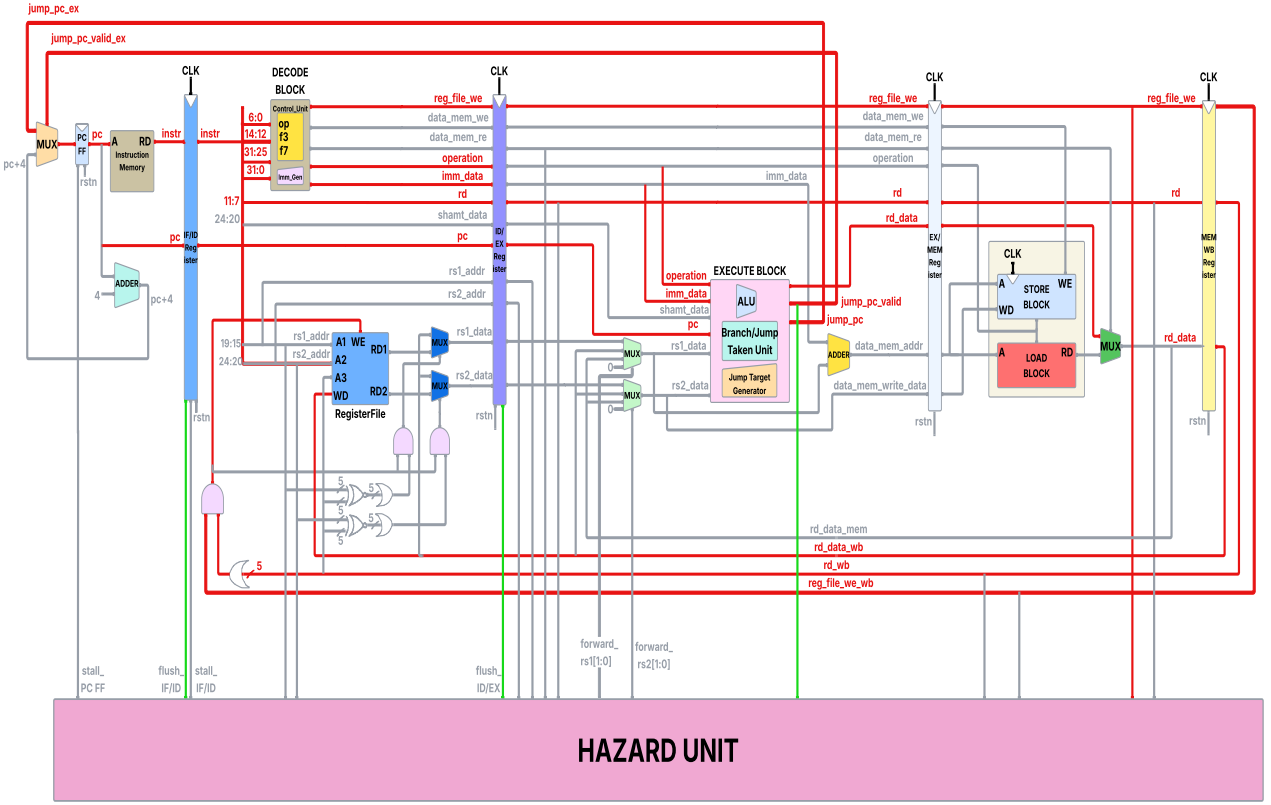

Figure. JAL Instruction Data Flow

J-TYPE INSTRUCION

JALR INSTRUCTION

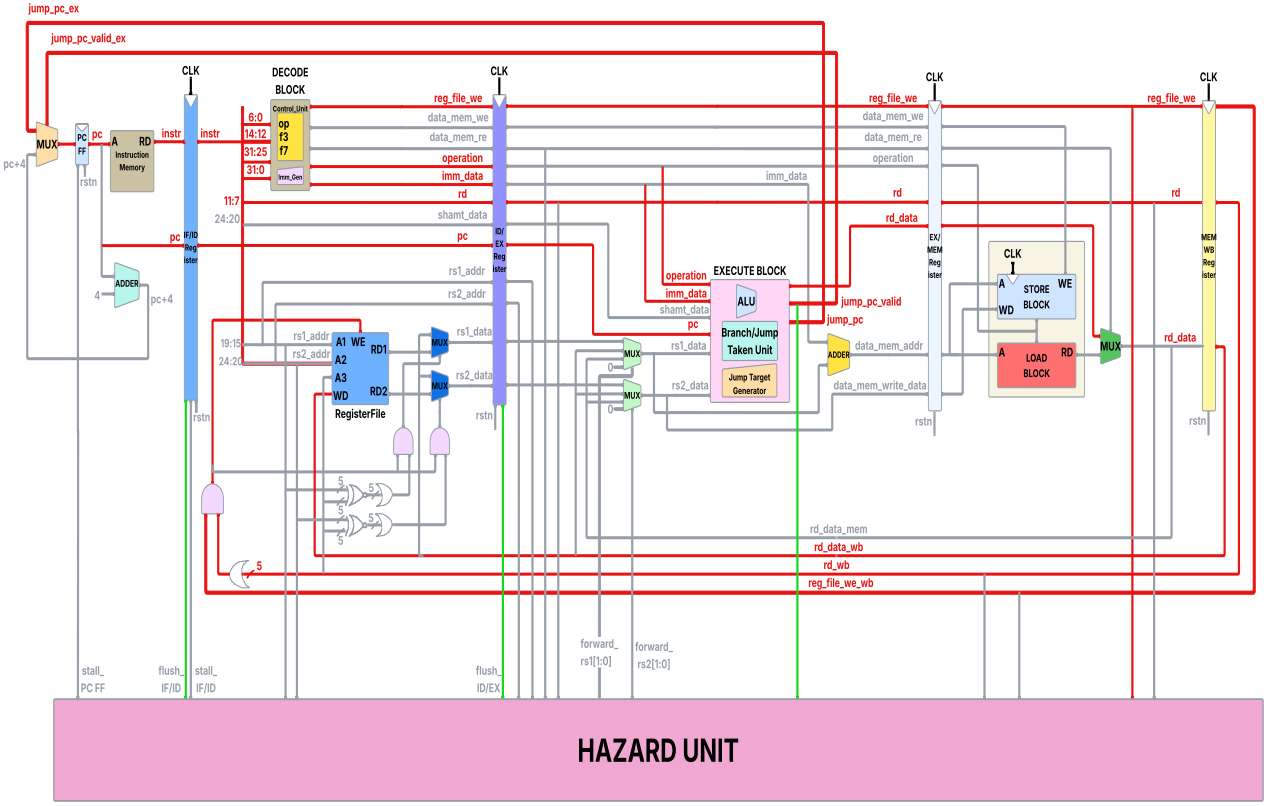

Figure. JALR Instruction Data Flow

U-TYPE INSTRUCTION

LUI INSTRUCTION

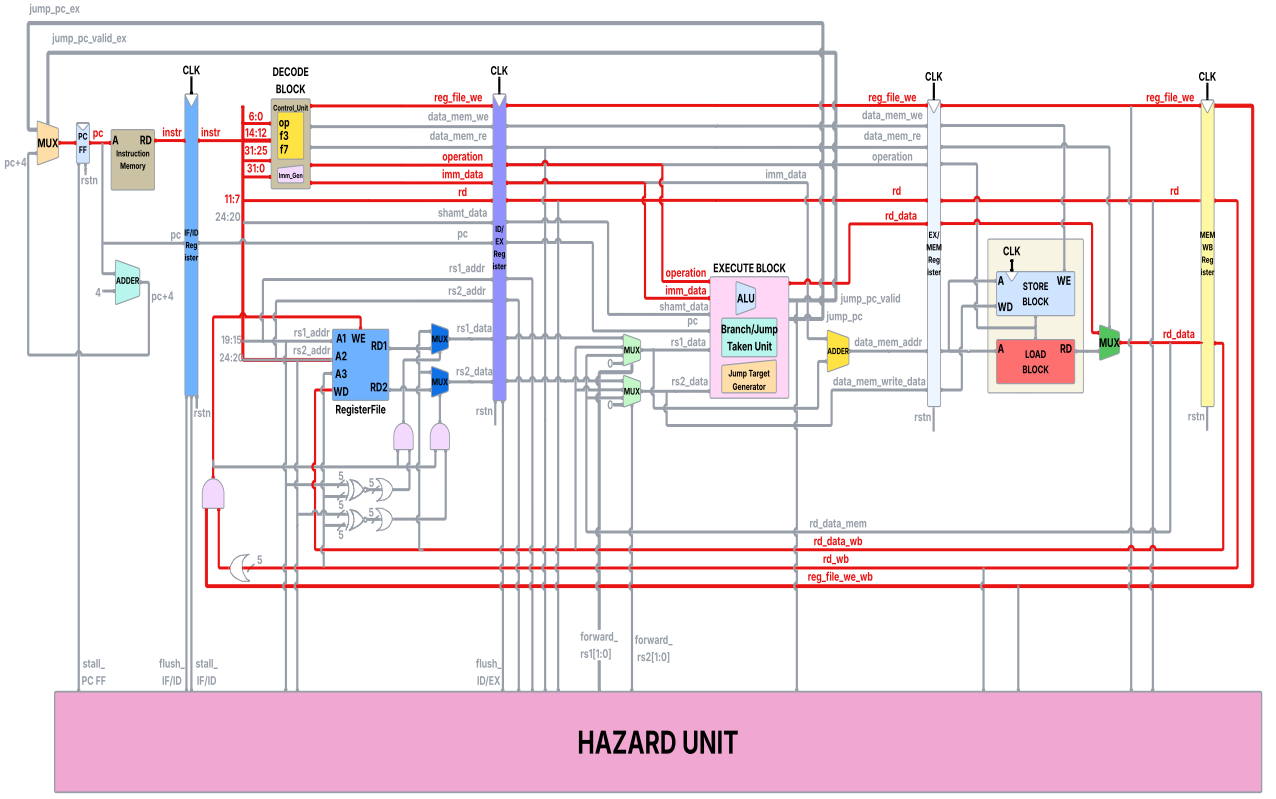

Figure. LUI Instruction Data Flow

U-TYPE INSTRUCTION

AUIPC INSTRUCTION

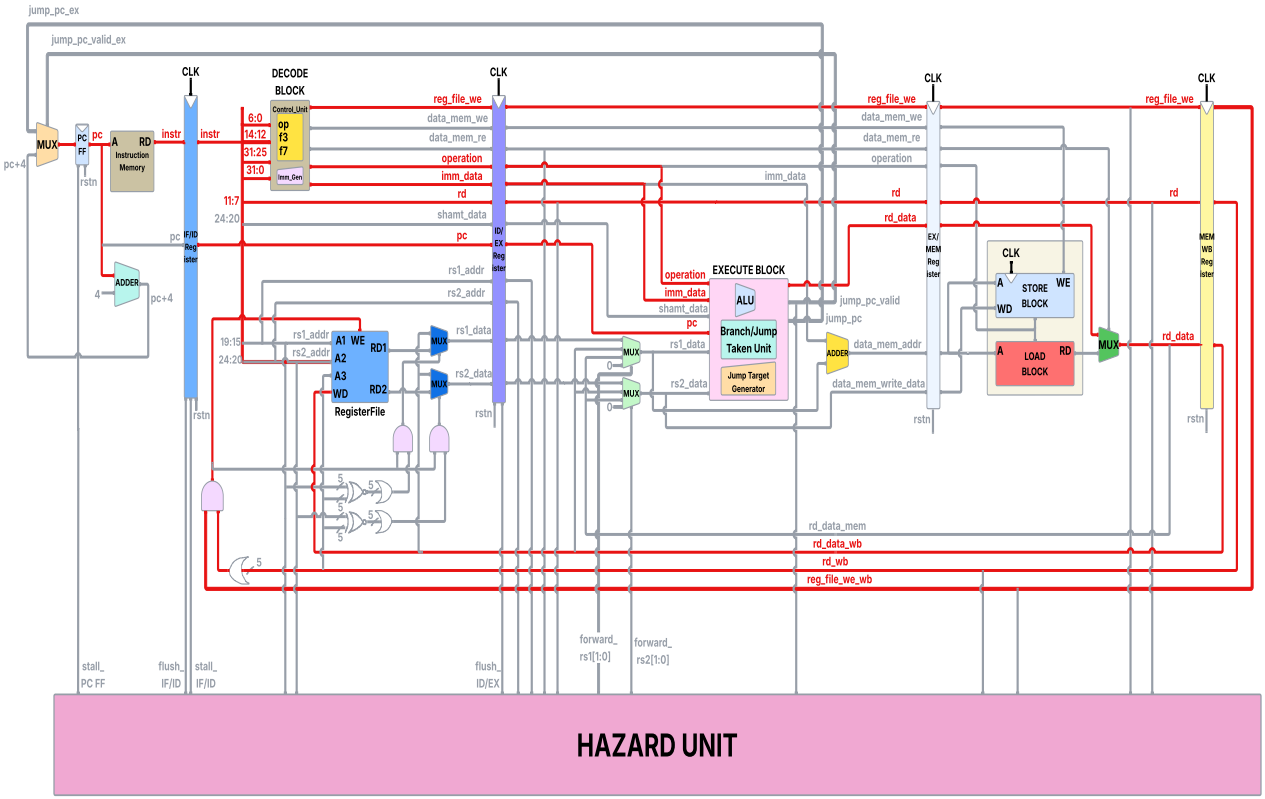

Figure. AUIPC Instruction Data Flow

Part III

RISC-V Base Instruction Set Entegrasyon Testleri

ALU

ALU Test Assembly Kodu

```
.global _start
_start:
    nop
_init:
    nop
main:
    li   x5, 5
    li   x3, 10
    li   x10, -7
    li   x2, -3
############## ADD #############
    add  x6,x5,x3    # iki pozitif sayı toplamı x6 = 5 + 10 = 15(0x0000000F)
    add  x7,x5,x2    # bir pozitif bir negatif sayı toplamı x7 = 5 + (-3) = 2(0x00000002)
    add  x8,x10,x2   # iki negatif sayı toplamı x8 = (-7) + (-3) = -10(0xFFFFFFF6)
############## SUB #############
    sub  x9,x5,x3   # iki pozitif sayı, sonuc negatif, x9 = 5 - 10 = -5(0xFFFFFFFB)
    sub  x11,x3,x5  # iki pozitif sayı, sonuc pozitif, x11 = 10 - 5 = 5
    sub  x12,x10,x2 # iki negatif sayı, sonuc negatif, x12 = -7 - (-3) = -4
    sub  x13,x2,x10 # iki negatif sayı, sonuc pozitif, x13 = -3 - (-7) = 4
######### SLL-SRL-SRA #########
    sll  x14,x5,x7  # veri logic olarak sola kaydırılır, x7 register'ı içinde 2 var veri
    2 defa sola kaydırılacak, boşalan bitlere 0 gelecek.
    srl  x15,x5,x7  # veri logic olarak sağa kaydırılır, x7 register'ı içinde 2 var veri
    2 defa sağa kaydırılacak, boşalan bitlere 0 gelicek
    sra  x16,x10,x7 # veri aritmetik olarak sağa kaydırılır, x7 register'ı içinde 2 var
    veri 2 defa sağa kaydırılacak, boşalan bitlere işaret biti gelicek.
########## SLT-SLTU ###########
    slt  x9,x5,x2   # x5 register'ı içindeki veri x2 register'ı içindeki veriden büyükse
    x9 register'ına 1 atanacak değilse 0 atanacak. İşarete hassasiyetlidir.
    sltu x6,x5,x3   # x5 register'ı içindeki veri x2 register'ı içindeki veriden büyükse
    x9 register'ına 1 atanacak değilse 0 atanacak.
######### XOR-OR-AND ##########
    xor x8,x9,x6    # 1 xor 0 = 1
    or  x13,x9,x6   # 1 or 0  = 1
    and x14,x9,x6   # 1 and 0 = 0

    j test_end
test_end:
    j test_end

```

ALU

ALU Test Log Karşılaştırması ve Simülasyon Özeti

ALU Test Log

```
    0x80000000 (0x00000013)
    0x80000004 (0x00000013)
    0x80000008 (0x00500293) x5  0x00000005
    0x8000000c (0x00a00193) x3  0x0000000a
    0x80000010 (0xff900513) x10 0xfffffff9
    0x80000014 (0xffd00113) x2  0xfffffffd
    0x80000018 (0x00328333) x6  0x0000000f
    0x8000001c (0x002283b3) x7  0x00000002
    0x80000020 (0x00250433) x8  0xfffffff6
    0x80000024 (0x403284b3) x9  0xfffffffb
    0x80000028 (0x405185b3) x11 0x00000005
    0x8000002c (0x40250633) x12 0xfffffffc
    0x80000030 (0x40a106b3) x13 0x00000004
    0x80000034 (0x00729733) x14 0x00000014
    0x80000038 (0x0072d7b3) x15 0x00000001
    0x8000003c (0x40755833) x16 0xfffffffe
    0x80000040 (0x0022a4b3) x9  0x00000000
    0x80000044 (0x0032b333) x6  0x00000001
    0x80000048 (0x0064c433) x8  0x00000001
    0x8000004c (0x0064e6b3) x13 0x00000001
    0x80000050 (0x0064f733) x14 0x00000000
    0x80000054 (0x0040006f)
```

Golden Model Test Log

```
    0x80000000 (0x00000013)
    0x80000004 (0x00000013)
    0x80000008 (0x00500293) x5  0x00000005
    0x8000000c (0x00a00193) x3  0x0000000a
    0x80000010 (0xff900513) x10 0xfffffff9
    0x80000014 (0xffd00113) x2  0xfffffffd
    0x80000018 (0x00328333) x6  0x0000000f
    0x8000001c (0x002283b3) x7  0x00000002
    0x80000020 (0x00250433) x8  0xfffffff6
    0x80000024 (0x403284b3) x9  0xfffffffb
    0x80000028 (0x405185b3) x11 0x00000005
    0x8000002c (0x40250633) x12 0xfffffffc
    0x80000030 (0x40a106b3) x13 0x00000004
    0x80000034 (0x00729733) x14 0x00000014
    0x80000038 (0x0072d7b3) x15 0x00000001
    0x8000003c (0x40755833) x16 0xfffffffe
    0x80000040 (0x0022a4b3) x9  0x00000000
    0x80000044 (0x0032b333) x6  0x00000001
    0x80000048 (0x0064c433) x8  0x00000001
    0x8000004c (0x0064e6b3) x13 0x00000001
    0x80000050 (0x0064f733) x14 0x00000000
    0x80000054 (0x0040006f)
```

ALU Simulation Summary (Verilator)

```
                ------------------------------------------------------------
                SUCCESS: Test End Reached (PC: 0x80000058)
                ------------------------------------------------------------
                - tb/tb.sv:67: Verilog $finish
                0x80000058 (0x0000006f)
                ------------------------------------------------------------
                Total Cycles: 29
                Committed Instructions: 22
                CPI: 1.3182
                ------------------------------------------------------------
                - S i m u l a t i o n   R e p o r t: Verilator 5.037 devel
                - Verilator: $finish at 78ps; walltime 0.001 s; speed 77.335 ns/s
                - Verilator: cpu 0.001 s on 1 threads; alloced 121 MB
```

Figure. ALU Test Log Karşılaştırması ve Simulasyon Özeti

ALUIMM

ALUIMM Test Assembly Kodu

```
.global _start

_start:
    nop
_init:
    nop

main:
    li    x10, 554
    li    x11, 376
    li    x12, -812
    li    x13, -611
    li    x25, 1

############## ADDI ##############
    addi  x1, x10, 16   # iki pozitif sayı
    addi  x2, x10, -14  # bir pozitif bir negatif sayı
    addi  x3, x12, -8   # iki negatif sayı
########### SLTI/SLTIU ###########
    slti  x4,  x10, 500  # x4 = 0 set edilmeyecek
    slti  x5,  x11, 378  # x5 = 1 set edilecek
    slti  x6,  x12, -810 # x6 = 1 set edilecek
    slti  x7,  x13, -612 # x7 = 0 set edilmeyecek
    slti  x8,  x12, 10   # x8 = 1 set edilecek
    sltiu x9,  x12, 10   # x9 = 0 set edilmeyecek çünkü unsigned olduğu için x12 = -812
    değil aslında çok büyük bir sayı
######### XORI/ORI/ANDI #########
    xori  x14,x11,123
    ori   x15,x10,999
    andi  x16,x11, 777
######### SLLI/SRLI/SRAI #########
    slli  x17,x10,2  # x17 = 2216
    srli  x18,x11,2  # x18 = 94
    srai  x19,x12,1  # x19 = -406 işaret korunmalı.
    slli  x20,x10,0  # x20 = 554, sayı değişmemeli
    slli  x21,x25,31 # x21 = 8'h80000000
    srai  x22,x12,0  # x22 = x12, sayı değişmemeli
    srai  x23,x12,31 # x23 = 8'hFFFFFFFF
    srli  x24,x11,31 # x24 = 32'b0
test_end:
    j test_end


```

ALUIMM

ALUIMM Test Log Karşılaştırması ve Simülasyon Özeti

ALUIMM Test Log

```
    0x80000000 (0x00000013)
    0x80000004 (0x00000013)
    0x80000008 (0x22a00513) x10 0x0000022a
    0x8000000c (0x17800593) x11 0x00000178
    0x80000010 (0xcd400613) x12 0xfffffcd4
    0x80000014 (0xd9d00693) x13 0xfffffd9d
    0x80000018 (0x00100c93) x25 0x00000001
    0x8000001c (0x01050093) x1  0x0000023a
    0x80000020 (0xff250113) x2  0x0000021c
    0x80000024 (0xff860193) x3  0xfffffccc
    0x80000028 (0x1f452213) x4  0x00000000
    0x8000002c (0x17a5a293) x5  0x00000001
    0x80000030 (0xcd662313) x6  0x00000001
    0x80000034 (0xd9c6a393) x7  0x00000000
    0x80000038 (0x00a62413) x8  0x00000001
    0x8000003c (0x00a63493) x9  0x00000000
    0x80000040 (0x07b5c713) x14 0x00000103
    0x80000044 (0x3e756793) x15 0x000003ef
    0x80000048 (0x3095f813) x16 0x00000108
    0x8000004c (0x00251893) x17 0x000008a8
    0x80000050 (0x0025d913) x18 0x0000005e
    0x80000054 (0x40165993) x19 0xfffffe6a
    0x80000058 (0x00051a13) x20 0x0000022a
    0x8000005c (0x01fc9a93) x21 0x80000000
    0x80000060 (0x40065b13) x22 0xfffffcd4
    0x80000064 (0x41f65b93) x23 0xffffffff
    0x80000068 (0x01f5dc13) x24 0x00000000
```

ALUIMM Golden Model Test Log

```
    0x80000000 (0x00000013)
    0x80000004 (0x00000013)
    0x80000008 (0x22a00513) x10 0x0000022a
    0x8000000c (0x17800593) x11 0x00000178
    0x80000010 (0xcd400613) x12 0xfffffcd4
    0x80000014 (0xd9d00693) x13 0xfffffd9d
    0x80000018 (0x00100c93) x25 0x00000001
    0x8000001c (0x01050093) x1  0x0000023a
    0x80000020 (0xff250113) x2  0x0000021c
    0x80000024 (0xff860193) x3  0xfffffccc
    0x80000028 (0x1f452213) x4  0x00000000
    0x8000002c (0x17a5a293) x5  0x00000001
    0x80000030 (0xcd662313) x6  0x00000001
    0x80000034 (0xd9c6a393) x7  0x00000000
    0x80000038 (0x00a62413) x8  0x00000001
    0x8000003c (0x00a63493) x9  0x00000000
    0x80000040 (0x07b5c713) x14 0x00000103
    0x80000044 (0x3e756793) x15 0x000003ef
    0x80000048 (0x3095f813) x16 0x00000108
    0x8000004c (0x00251893) x17 0x000008a8
    0x80000050 (0x0025d913) x18 0x0000005e
    0x80000054 (0x40165993) x19 0xfffffe6a
    0x80000058 (0x00051a13) x20 0x0000022a
    0x8000005c (0x01fc9a93) x21 0x80000000
    0x80000060 (0x40065b13) x22 0xfffffcd4
    0x80000064 (0x41f65b93) x23 0xffffffff
    0x80000068 (0x01f5dc13) x24 0x00000000
```

ALUIMM Simulation Summary (Verilator)

```
                ----------------------------------------------------------
                SUCCESS: Test End Reached (PC: 0x8000006c)
                ----------------------------------------------------------
                - tb/tb.sv:67: Verilog $finish
                0x8000006c (0x0000006f)
                ----------------------------------------------------------
                Total Cycles: 32
                Committed Instructions: 27
                CPI: 1.1852
                ----------------------------------------------------------
                - S i m u l a t i o n   R e p o r t: Verilator 5.037 devel
                - Verilator: $finish at 84ps; walltime 0.002 s; speed 48.291 ns/s
                - Verilator: cpu 0.002 s on 1 threads; alloced 121 MB
```

Figure. ALUIMM Test Log Karşılaştırması ve Simülasyon Özeti

ALU HAZARDS

ALU Hazards Test Assembly Kodu

```
.global _start

_start:
    nop
_init:
    nop

main:
    li x1, 10        # Test değeri 1
    li x2, 20        # Test değeri 2
    li x3, 5         # Shift miktarı vb.

# SENARYO 1 MEMORY AŞAMASINDAN EXECUTE AŞAMASINA FORWARD
    add x4, x1, x2  # x4 = 10 + 20 = 30 (Bu komut EX'teyken...)
    sub x5, x4, x3  # x5 = 30 - 5 = 25  (...bu komut ID'den yeni çıktı)
                    # Hazard Unit, x4'ün değerini x4 Register File'a yazılmadan
                    # havada yakalayıp sub işlemine vermeli.
# SENARYO 2 WB AŞAMASINDAN EXECUTE AŞAMASINA FORWARD
    add x6, x1, x1  # x6 = 20
    nop             # Araya bir boşluk (veya alakasız komut)
    xor x7, x6, x2  # x6 burada kullanılacak.
                    # x6 şu an MEM/WB sınırında, x7 ise EX aşamasında.
# SENARYO 3 RS1 RS2 AYNI ANDA BAĞIMLI
    addi x8, x0, 100 # x8 = 100
    add  x9, x8, x8  # x9 = 100 + 100 = 200
                     # Burada hem rs1 (x8) hem rs2 (x8) bir önceki komuta bağımlı!
                     # Forwarding Unit'in iki bacağı da (ForwardA ve ForwardB) aktif
    olmalı.
# SENARYO 4
    or   x10, x4, x5 # x4 ve x5 yukarılardan geliyor (Uzak mesafe testi)
    andi x11, x10, 0x0FF # x10'a bağımlı yani memory aşamasından forwarding

    slli x12, x11, 2 # x11'e bağımlı yani memory aşamasından forwarding lazım
    add  x13, x12, x11 # x12 ve x11'e bağımlı yani hem wb aşamasından hem memory aşaması
    ndan forwarding lazım
test_end:
    j test_end

```

ALU HAZARDS

ALU Hazards Test Log Karşılaştırması ve Simülasyon Özeti

ALU Hazards Test Log

```
0x80000000 (0x00000013)
0x80000004 (0x00000013)
0x80000008 (0x00a00093) x1  0x0000000a
0x8000000c (0x01400113) x2  0x00000014
0x80000010 (0x00500193) x3  0x00000005
0x80000014 (0x00208233) x4  0x0000001e
0x80000018 (0x403202b3) x5  0x00000019
0x8000001c (0x00108333) x6  0x00000014
0x80000020 (0x00000013)
0x80000024 (0x002343b3) x7  0x00000000
0x80000028 (0x06400413) x8  0x00000064
0x8000002c (0x008404b3) x9  0x000000c8
0x80000030 (0x00526533) x10 0x0000001f
0x80000034 (0x0ff57593) x11 0x0000001f
0x80000038 (0x00259613) x12 0x0000007c
0x8000003c (0x00b606b3) x13 0x0000009b
```

ALU Hazards Golden Model Test Log

```
0x80000000 (0x00000013)
0x80000004 (0x00000013)
0x80000008 (0x00a00093) x1  0x0000000a
0x8000000c (0x01400113) x2  0x00000014
0x80000010 (0x00500193) x3  0x00000005
0x80000014 (0x00208233) x4  0x0000001e
0x80000018 (0x403202b3) x5  0x00000019
0x8000001c (0x00108333) x6  0x00000014
0x80000020 (0x00000013)
0x80000024 (0x002343b3) x7  0x00000000
0x80000028 (0x06400413) x8  0x00000064
0x8000002c (0x008404b3) x9  0x000000c8
0x80000030 (0x00526533) x10 0x0000001f
0x80000034 (0x0ff57593) x11 0x0000001f
0x80000038 (0x00259613) x12 0x0000007c
0x8000003c (0x00b606b3) x13 0x0000009b
```

ALU Hazards Simulation Summary (Verilator)

```
------------------------------------------------------------
SUCCESS: Test End Reached (PC: 0x80000040)
------------------------------------------------------------
- tb/tb.sv:67: Verilog $finish
0x80000040 (0x0000006f)
------------------------------------------------------------
Total Cycles: 21
Committed Instructions: 16
CPI: 1.3125
------------------------------------------------------------
- S i m u l a t i o n   R e p o r t: Verilator 5.037 devel
- Verilator: $finish at 62ps; walltime 0.002 s; speed 39.730 ns/s
- Verilator: cpu 0.002 s on 1 threads; alloced 121 MB
```

Figure. ALU Hazards Test Log Karşılaştırması ve Simülasyon Özeti

BEQ

BEQ Test Assembly Kodu

```
.global _start

_start:
    nop
_init:
    nop

main:
    li x1, 2
    li x2, 2
    li x3, -1
    li x4, 0
    li x5, 0
    li x8, 0
    li x9, 0
######### BRANCH TAKEN #########
    beq x1,x2,label_taken_success
    li x8, 1 # burası atlanmalı
    j test_not_taken
label_taken_success:
    li x9,1
test_not_taken:
######## BRANCH NOT TAKEN ########
    beq x2,x3,label_not_taken_fail # aşağıdaki kod bloğu atlanmamalı, çünkü x2 ve x3 eşit
     değil
    li x4, 1
    j test_end
label_not_taken_fail:
    li x5, 1
test_end:
    j test_end


```

BEQ

BEQ Test Log Karşılaştırması ve Simülasyon Özeti

BEQ Test Log

```
    0x80000000 (0x00000013)
    0x80000004 (0x00000013)
    0x80000008 (0x00200093) x1  0x00000002
    0x8000000c (0x00200113) x2  0x00000002
    0x80000010 (0xfff00193) x3  0xffffffff
    0x80000014 (0x00000213) x4  0x00000000
    0x80000018 (0x00000293) x5  0x00000000
    0x8000001c (0x00000413) x8  0x00000000
    0x80000020 (0x00000493) x9  0x00000000
    0x80000024 (0x00208663)
    0x80000030 (0x00100493) x9  0x00000001
    0x80000034 (0x00310663)
    0x80000038 (0x00100213) x4  0x00000001
    0x8000003c (0x0080006f)
```

BEQ Golden Model Test Log

```
    0x80000000 (0x00000013)
    0x80000004 (0x00000013)
    0x80000008 (0x00200093) x1  0x00000002
    0x8000000c (0x00200113) x2  0x00000002
    0x80000010 (0xfff00193) x3  0xffffffff
    0x80000014 (0x00000213) x4  0x00000000
    0x80000018 (0x00000293) x5  0x00000000
    0x8000001c (0x00000413) x8  0x00000000
    0x80000020 (0x00000493) x9  0x00000000
    0x80000024 (0x00208663)
    0x80000030 (0x00100493) x9  0x00000001
    0x80000034 (0x00310663)
    0x80000038 (0x00100213) x4  0x00000001
    0x8000003c (0x0080006f)
```

BEQ Simulation Summary (Verilator)

```
                ------------------------------------------------------------
                SUCCESS: Test End Reached (PC: 0x80000044)
                ------------------------------------------------------------
                - tb/tb.sv:67: Verilog $finish
                0x80000044 (0x0000006f)
                ------------------------------------------------------------
                Total Cycles: 23
                Committed Instructions: 14
                CPI: 1.6429
                ------------------------------------------------------------
                - S i m u l a t i o n   R e p o r t: Verilator 5.037 devel
                - Verilator: $finish at 66ps; walltime 0.002 s; speed 41.254 ns/s
                - Verilator: cpu 0.002 s on 1 threads; alloced 121 MB
```

Figure. BEQ Test Log Karşılaştırması ve Simülasyon Özeti

BGE

BGE Test Assembly Kodu

```
.global _start

_start:
    nop
_init:
    nop

main:
    li x1, 3
    li x2, 2
    li x3, -1
    li x4, 0
    li x5, 0
    li x8, 0
    li x9, 0

######### BRANCH TAKEN #########
    bge x1,x2,label_taken_success
    li x8, 1 # burası atlanmalı
    j test_not_taken  # eğer burayı koymazsam, işlemci bozuk olsa bile kod doğrusal aşağı aktığı için x9'a her halükarda 1 set edilecek ve hatayı anlayamıcam.
label_taken_success:
    li x9,1
test_not_taken:
######## BRANCH NOT TAKEN ########
    bge x2,x1,label_not_taken_fail # aşağıdaki kod bloğu atlanmamalı, çünkü x2 ve x1'den büyük eşit değil
    li x4, 1
    j test_end
label_not_taken_fail:
    li x5, 1
test_end:
    j test_end


```

BGE

BGE Test Log Karşılaştırması ve Simülasyon Özeti

BGE Test Log

```
    0x80000000 (0x00000013)
    0x80000004 (0x00000013)
    0x80000008 (0x00300093) x1  0x00000003
    0x8000000c (0x00200113) x2  0x00000002
    0x80000010 (0xfff00193) x3  0xffffffff
    0x80000014 (0x00000213) x4  0x00000000
    0x80000018 (0x00000293) x5  0x00000000
    0x8000001c (0x00000413) x8  0x00000000
    0x80000020 (0x00000493) x9  0x00000000
    0x80000024 (0x0020d663)
    0x80000030 (0x00100493) x9  0x00000001
    0x80000034 (0x00115663)
    0x80000038 (0x00100213) x4  0x00000001
    0x8000003c (0x0080006f)
```

BGE Golden Model Test Log

```
    0x80000000 (0x00000013)
    0x80000004 (0x00000013)
    0x80000008 (0x00300093) x1  0x00000003
    0x8000000c (0x00200113) x2  0x00000002
    0x80000010 (0xfff00193) x3  0xffffffff
    0x80000014 (0x00000213) x4  0x00000000
    0x80000018 (0x00000293) x5  0x00000000
    0x8000001c (0x00000413) x8  0x00000000
    0x80000020 (0x00000493) x9  0x00000000
    0x80000024 (0x0020d663)
    0x80000030 (0x00100493) x9  0x00000001
    0x80000034 (0x00115663)
    0x80000038 (0x00100213) x4  0x00000001
    0x8000003c (0x0080006f)
```

BGE Simulation Summary (Verilator)

```
            ------------------------------------------------------------
            SUCCESS: Test End Reached (PC: 0x80000044)
            ------------------------------------------------------------
            - tb/tb.sv:67: Verilog $finish
            0x80000044 (0x0000006f)
            ------------------------------------------------------------
            Total Cycles: 23
            Committed Instructions: 14
            CPI: 1.6429
            ------------------------------------------------------------
            - S i m u l a t i o n   R e p o r t: Verilator 5.037 devel
            - Verilator: $finish at 66ps; walltime 0.002 s; speed 38.111 ns/s
            - Verilator: cpu 0.002 s on 1 threads; alloced 121 MB
```

Figure. BGE Test Log Karşılaştırması ve Simülasyon Özeti

BGEU

BGEU Test Assembly Kodu

```
.global _start

_start:
    nop
_init:
    nop

main:
    li x1, 122
    li x2, 98
    li x3, -71
    li x4, 0
    li x5, 0
    li x8, 0
    li x9, 0

######### BRANCH TAKEN #########
    bgeu x1,x2,label_taken_success
    li x8, 1 # burası atlanmalı
    j test_not_taken  # eğer burayı koymazsam, işlemci bozuk olsa bile kod doğrusal aşağı aktığı için x9'a her halükarda 1 set edilecek ve hatayı anlayamıcam.
label_taken_success:
    li x9,1
test_not_taken:
######## BRANCH NOT TAKEN ########
    bgeu x1,x3,label_not_taken_fail # aşağıdaki kod bloğu atlanmamalı, çünkü x1 ve x3'den büyük eşit değil. bgeu(unsigned) oldugu için x3(0xFFFFFFB9) = -71 değil 4294967225 olarak ifade edilir.
    li x4, 1

    j test_end

label_not_taken_fail:
    li x5, 1
test_end:
    j test_end


```

BGEU

BGEU Test Log Karşılaştırması ve Simülasyon Özeti

BGEU Test Log

```
    0x80000000 (0x00000013)
    0x80000004 (0x00000013)
    0x80000008 (0x07a00093) x1  0x0000007a
    0x8000000c (0x06200113) x2  0x00000062
    0x80000010 (0xfb900193) x3  0xfffffffb9
    0x80000014 (0x00000213) x4  0x00000000
    0x80000018 (0x00000293) x5  0x00000000
    0x8000001c (0x00000413) x8  0x00000000
    0x80000020 (0x00000493) x9  0x00000000
    0x80000024 (0x0020f663)
    0x80000030 (0x00100493) x9  0x00000001
    0x80000034 (0x0030f663)
    0x80000038 (0x00100213) x4  0x00000001
    0x8000003c (0x0080006f)
```

BGEU Golden Model Test Log

```
    0x80000000 (0x00000013)
    0x80000004 (0x00000013)
    0x80000008 (0x07a00093) x1  0x0000007a
    0x8000000c (0x06200113) x2  0x00000062
    0x80000010 (0xfb900193) x3  0xfffffffb9
    0x80000014 (0x00000213) x4  0x00000000
    0x80000018 (0x00000293) x5  0x00000000
    0x8000001c (0x00000413) x8  0x00000000
    0x80000020 (0x00000493) x9  0x00000000
    0x80000024 (0x0020f663)
    0x80000030 (0x00100493) x9  0x00000001
    0x80000034 (0x0030f663)
    0x80000038 (0x00100213) x4  0x00000001
    0x8000003c (0x0080006f)
```

BGEU Simulation Summary (Verilator)

```
                ------------------------------------------------------------
                SUCCESS: Test End Reached (PC: 0x80000044)
                ------------------------------------------------------------
                - tb/tb.sv:67: Verilog $finish
                0x80000044 (0x0000006f)
                ------------------------------------------------------------
                Total Cycles: 23
                Committed Instructions: 14
                CPI: 1.6429
                ------------------------------------------------------------
                - S i m u l a t i o n   R e p o r t: Verilator 5.037 devel
                - Verilator: $finish at 66ps; walltime 0.002 s; speed 43.379 ns/s
                - Verilator: cpu 0.002 s on 1 threads; alloced 121 MB
```

Figure. BGEU Test Log Karşılaştırması ve Simülasyon Özeti

BLT

BLT Test Assembly Kodu

```
.global _start

_start:
    nop
_init:
    nop

main:
    li x1, 35
    li x2, 20
    li x3, -30
    li x4, 0
    li x5, 0
    li x8, 0
    li x9, 0

######### BRANCH TAKEN #########
    blt x3,x1,label_taken_success
    li x8, 1 # burası atlanmalı
    j test_not_taken
label_taken_success:
    li x9,1
test_not_taken:
######## BRANCH NOT TAKEN ########
    blt x1,x2,label_not_taken_fail # aşağıdaki kod bloğu atlanmamalı, çünkü x1 x2'den küç
    ük değil.
    li x4, 1

    j test_end

label_not_taken_fail:
    li x5, 1

test_end:
    j test_end

```

BLT

BLT Test Log Karşılaştırması ve Simülasyon Özeti

BLT Test Log

```
    0x80000000 (0x00000013)
    0x80000004 (0x00000013)
    0x80000008 (0x02300093) x1  0x00000023
    0x8000000c (0x01400113) x2  0x00000014
    0x80000010 (0xfe200193) x3  0xfffffffe2
    0x80000014 (0x00000213) x4  0x00000000
    0x80000018 (0x00000293) x5  0x00000000
    0x8000001c (0x00000413) x8  0x00000000
    0x80000020 (0x00000493) x9  0x00000000
    0x80000024 (0x0011c663)
    0x80000030 (0x00100493) x9  0x00000001
    0x80000034 (0x0020c663)
    0x80000038 (0x00100213) x4  0x00000001
    0x8000003c (0x0080006f)
```

BLT Golden Model Test Log

```
    0x80000000 (0x00000013)
    0x80000004 (0x00000013)
    0x80000008 (0x02300093) x1  0x00000023
    0x8000000c (0x01400113) x2  0x00000014
    0x80000010 (0xfe200193) x3  0xfffffffe2
    0x80000014 (0x00000213) x4  0x00000000
    0x80000018 (0x00000293) x5  0x00000000
    0x8000001c (0x00000413) x8  0x00000000
    0x80000020 (0x00000493) x9  0x00000000
    0x80000024 (0x0011c663)
    0x80000030 (0x00100493) x9  0x00000001
    0x80000034 (0x0020c663)
    0x80000038 (0x00100213) x4  0x00000001
    0x8000003c (0x0080006f)
```

BLT Simulation Summary (Verilator)

```
                ------------------------------------------------------------
                SUCCESS: Test End Reached (PC: 0x80000044)
                ------------------------------------------------------------
                - tb/tb.sv:67: Verilog $finish
                0x80000044 (0x0000006f)
                ------------------------------------------------------------
                Total Cycles: 23
                Committed Instructions: 14
                CPI: 1.6429
                ------------------------------------------------------------
                - S i m u l a t i o n   R e p o r t: Verilator 5.037 devel
                - Verilator: $finish at 66ps; walltime 0.001 s; speed 64.751 ns/s
                - Verilator: cpu 0.001 s on 1 threads; alloced 121 MB
```

Figure. BLT Test Log Karşılaştırması ve Simülasyon Özeti

BLTU

BLTU Test Assembly Kodu

```
.global _start

_start:
    nop
_init:
    nop

main:
    li x1, 99
    li x2, 82
    li x3, -30
    li x4, 0
    li x5, 0
    li x8, 0
    li x9, 0
######### BRANCH TAKEN #########
    bltu x2,x1,label_taken_success
    li x8, 1 # burası atlanmalı
    j test_not_taken

label_taken_success:
    li x9,1

test_not_taken:
######## BRANCH NOT TAKEN ########
    bltu x3,x2,label_not_taken_fail # aşağıdaki kod bloğu atlanmamalı, çünkü bltu(
    unsigned) olduğu için x3(0xFFFFFFE2) = -30 değil 4294967266 sayısına eşittir.
    li x4, 1

    j test_end
label_not_taken_fail:
    li x5, 1

test_end:
    j test_end


```

BLTU

BLTU Test Log Karşılaştırması ve Simülasyon Özeti

BLTU Test Log

```
    0x80000000 (0x00000013)
    0x80000004 (0x00000013)
    0x80000008 (0x06300093) x1  0x00000063
    0x8000000c (0x05200113) x2  0x00000052
    0x80000010 (0xfe200193) x3  0xfffffffe2
    0x80000014 (0x00000213) x4  0x00000000
    0x80000018 (0x00000293) x5  0x00000000
    0x8000001c (0x00000413) x8  0x00000000
    0x80000020 (0x00000493) x9  0x00000000
    0x80000024 (0x00116663)
    0x80000030 (0x00100493) x9  0x00000001
    0x80000034 (0x0021e663)
    0x80000038 (0x00100213) x4  0x00000001
    0x8000003c (0x0080006f)
```

BLTU Golden Model Test Log

```
    0x80000000 (0x00000013)
    0x80000004 (0x00000013)
    0x80000008 (0x06300093) x1  0x00000063
    0x8000000c (0x05200113) x2  0x00000052
    0x80000010 (0xfe200193) x3  0xfffffffe2
    0x80000014 (0x00000213) x4  0x00000000
    0x80000018 (0x00000293) x5  0x00000000
    0x8000001c (0x00000413) x8  0x00000000
    0x80000020 (0x00000493) x9  0x00000000
    0x80000024 (0x00116663)
    0x80000030 (0x00100493) x9  0x00000001
    0x80000034 (0x0021e663)
    0x80000038 (0x00100213) x4  0x00000001
    0x8000003c (0x0080006f)
```

BLTU Simulation Summary (Verilator)

```
                ----------------------------------------------------------
                SUCCESS: Test End Reached (PC: 0x80000044)
                ----------------------------------------------------------
                - tb/tb.sv:67: Verilog $finish
                0x80000044 (0x0000006f)
                ----------------------------------------------------------
                Total Cycles: 23
                Committed Instructions: 14
                CPI: 1.6429
                ----------------------------------------------------------
                - S i m u l a t i o n   R e p o r t: Verilator 5.037 devel
                - Verilator: $finish at 66ps; walltime 0.002 s; speed 40.381 ns/s
                - Verilator: cpu 0.002 s on 1 threads; alloced 121 MB
```

Figure. BLTU Test Log Karşılaştırması ve Simülasyon Özeti

BNE

BNE Test Assembly Kodu

```
.global _start

_start:
    nop
_init:
    nop

main:
    li x1, 2
    li x2, 2
    li x3, -1
    li x4, 0
    li x5, 0
    li x8, 0
    li x9, 0
######### BRANCH TAKEN #########
    bne x1,x3,label_taken_success
    li x8, 1 # burası atlanmalı
    j test_not_taken

label_taken_success:
    li x9,1

test_not_taken:
######## BRANCH NOT TAKEN ########
    bne x1,x2,label_not_taken_fail # aşağıdaki kod bloğu atlanmamalı, çünkü x2 ve x3 eşit
    li x4, 1

    j test_end

label_not_taken_fail:
    li x5, 1

test_end:
    j test_end


```

BNE Test Log

```
    0x80000000 (0x00000013)
    0x80000004 (0x00000013)
    0x80000008 (0x00200093) x1  0x00000002
    0x8000000c (0x00200113) x2  0x00000002
    0x80000010 (0xfff00193) x3  0xffffffff
    0x80000014 (0x00000213) x4  0x00000000
    0x80000018 (0x00000293) x5  0x00000000
    0x8000001c (0x00000413) x8  0x00000000
    0x80000020 (0x00000493) x9  0x00000000
    0x80000024 (0x00309663)
    0x80000030 (0x00100493) x9  0x00000001
    0x80000034 (0x00209663)
    0x80000038 (0x00100213) x4  0x00000001
    0x8000003c (0x0080006f)
```

BNE Golden Model Test Log

```
    0x80000000 (0x00000013)
    0x80000004 (0x00000013)
    0x80000008 (0x00200093) x1  0x00000002
    0x8000000c (0x00200113) x2  0x00000002
    0x80000010 (0xfff00193) x3  0xffffffff
    0x80000014 (0x00000213) x4  0x00000000
    0x80000018 (0x00000293) x5  0x00000000
    0x8000001c (0x00000413) x8  0x00000000
    0x80000020 (0x00000493) x9  0x00000000
    0x80000024 (0x00309663)
    0x80000030 (0x00100493) x9  0x00000001
    0x80000034 (0x00209663)
    0x80000038 (0x00100213) x4  0x00000001
    0x8000003c (0x0080006f)
```

BNE Simulation Summary (Verilator)

```
                ------------------------------------------------------------
                SUCCESS: Test End Reached (PC: 0x80000044)
                ------------------------------------------------------------
                - tb/tb.sv:67: Verilog $finish
                0x80000044 (0x0000006f)
                ------------------------------------------------------------
                Total Cycles: 23
                Committed Instructions: 14
                CPI: 1.6429
                ------------------------------------------------------------
                - S i m u l a t i o n   R e p o r t: Verilator 5.037 devel
                - Verilator: $finish at 66ps; walltime 0.001 s; speed 53.200 ns/s
                - Verilator: cpu 0.001 s on 1 threads; alloced 121 MB
```

Figure. BNE Test Log Karşılaştırması ve Simülasyon Özeti

NESTED BRANCH

NESTED BRANCH Test Assembly Kodu

```
.global _start

_start:
    nop
_init:
    nop

main:
    li x20,10  # iç döngü sınırı
    li x21,10  # dış döngü sınırı

    li x10,0 # iç döngü sayacı (j)
    li x11,0 # dış döngü sayacı (i)
    li x12,0 # toplam değeri(başlangıçta sıfır)

    li x30,0 # hata sayacı olarak kullanıcaz.

outer_loop:
    li x10,0 # iç döngü sayacı(j) dış döngü başlangıcında her seferinde sıfırlanır.

    inner_loop:
        addi x12,x12,1 # her döngüde +1 ekliyoruz toplama
        addi x10,x10,1 # iç döngü sayacını her defasın 1 artırıyoruz j++
        addi x30,x30,1
        bne  x10,x20,inner_loop

        #addi x30,x30,1 # iç döngü bitmeden buraya inmemesi lazım, nop enjekte ederek kaç defa buraya girmiş onu ölçücez.

    addi x11,x11,1 # dış döngü sayacını her defasında 1 artırıyoruz i++
    bne  x11,x21,outer_loop

    #addi x30,x30,1 # dış döngü tamamen bitmeden buraya inmemesi lazım nop sayısını ölçerek doğruluğu kontrol edicez yine.

test_end:
    # son durumda x30 = 100 olmalı düşük ya da fazlaysa flush mekanizması hatalı.
    j test_end

```

NESTED BRANCH

NESTED-BRANCH Test Log Karşılaştırması ve Simülasyon Özeti

NESTED BRANCH Test Log

```
    0x80000000 (0x00000013)
    0x80000004 (0x00000013)
    0x80000008 (0x00a00a13) x20 0x0000000a
    0x8000000c (0x00a00a93) x21 0x0000000a
    0x80000010 (0x00000513) x10 0x00000000
    0x80000014 (0x00000593) x11 0x00000000
    0x80000018 (0x00000613) x12 0x00000000
    0x8000001c (0x00000f13) x30 0x00000000
    0x80000020 (0x00000513) x10 0x00000000
    0x80000024 (0x00160613) x12 0x00000001
    0x80000028 (0x00150513) x10 0x00000001
    0x8000002c (0x001f0f13) x30 0x00000001
    0x80000030 (0xff451ae3)
    .
    .
    .
    0x80000024 (0x00160613) x12 0x00000063
    0x80000028 (0x00150513) x10 0x00000009
    0x8000002c (0x001f0f13) x30 0x00000063
    0x80000030 (0xff451ae3)
    0x80000024 (0x00160613) x12 0x00000064
    0x80000028 (0x00150513) x10 0x0000000a
    0x8000002c (0x001f0f13) x30 0x00000064
    0x80000030 (0xff451ae3)
    0x80000034 (0x00158593) x11 0x0000000a
    0x80000038 (0xff5594e3)
```

NESTED BRANCH Golden Model Test Log

```
    0x80000000 (0x00000013)
    0x80000004 (0x00000013)
    0x80000008 (0x00a00a13) x20 0x0000000a
    0x8000000c (0x00a00a93) x21 0x0000000a
    0x80000010 (0x00000513) x10 0x00000000
    0x80000014 (0x00000593) x11 0x00000000
    0x80000018 (0x00000613) x12 0x00000000
    0x8000001c (0x00000f13) x30 0x00000000
    0x80000020 (0x00000513) x10 0x00000000
    0x80000024 (0x00160613) x12 0x00000001
    0x80000028 (0x00150513) x10 0x00000001
    0x8000002c (0x001f0f13) x30 0x00000001
    0x80000030 (0xff451ae3)
    .
    .
    .
    0x80000024 (0x00160613) x12 0x00000063
    0x80000028 (0x00150513) x10 0x00000009
    0x8000002c (0x001f0f13) x30 0x00000063
    0x80000030 (0xff451ae3)
    0x80000024 (0x00160613) x12 0x00000064
    0x80000028 (0x00150513) x10 0x0000000a
    0x8000002c (0x001f0f13) x30 0x00000064
    0x80000030 (0xff451ae3)
    0x80000034 (0x00158593) x11 0x0000000a
    0x80000038 (0xff5594e3)
```

NESTED BRANCH Simulation Summary (Verilator)

```
    ------------------------------------------------------------
    SUCCESS: Test End Reached (PC: 0x8000003c)
    ------------------------------------------------------------
    - tb/tb.sv:67: Verilog $finish
    0x8000003c (0x0000006f)
    ------------------------------------------------------------
    Total Cycles: 641
    Committed Instructions: 438
    CPI: 1.4635
    ------------------------------------------------------------
    - S i m u l a t i o n   R e p o r t: Verilator 5.037 devel
    - Verilator: $finish at 1ns; walltime 0.002 s; speed 731.146 ns/s
    - Verilator: cpu 0.002 s on 1 threads; alloced 121 MB
```

Figure. NESTED BRANCH Test Log Karşılaştırması ve Simülasyon Özeti

JAL-JALR

JAL-JALR Test Assembly Kodu

```
.global _start

_start:
    nop
_init:
    nop

main:
    li x4, 0 # JAL Kontrol sinyali
    li x5, 0 # JALR Kontrol sinyali
    li x8, 10
    li x9, 5
############# JAL(Jump And Link) #############
    jal x1, funct_add   # toplama fonksiyonuna gidiyoruz, giderken x1(return address register) register'ına mevcut pc değerini kaydediyoruz.
return_point_jal:
    li x4, 1 # jal return point'e ulaştıysak jal doğru çalışıyor.
######## JALR(Jump And Link Register) #######

    la x6, funct_sub # hedef fonlsiyonun adresini x6 regiter'ına yükledik, bakalım jalr ile dinamik atlama yapabiliyor muyuz test edicez.
    jalr x1,x6,0     # x6'daki adrese yani funct_sub'a atlıyoruz ve şu anki pc değerini x1 register'ına kaydediyoruz geri dönüş için.
return_point_jalr:
    li x5,1 # jalr return point'e ulaştıysak jalr doğru çalışıyor.
    j test_end
funct_add:
    add  x14,x8,x9
    jalr x0,x1,0 # jalr ile x1 register'ı içindeki kaydedilmiş pc değerine geri dönüyoruz, offset olarak 0 kullanıyoruz, mevcut pc değerini ise x0 register'ına
                 # yazıyoruz, x0 register'ına yazma yapılamadığı için aslında çöpe atıyoruz çünkü geri dönerken pc değerini kaydetmemize gerek yok.
funct_sub:
    sub x15,x8,x9
    jalr x0,x1,0
test_end:
    j test_end


```

JAL-JALR Test Log

```
    0x80000000 (0x00000013)
    0x80000004 (0x00000013)
    0x80000008 (0x00000213) x4  0x00000000
    0x8000000c (0x00000293) x5  0x00000000
    0x80000010 (0x00a00413) x8  0x0000000a
    0x80000014 (0x00500493) x9  0x00000005
    0x80000018 (0x01c000ef) x1  0x8000001c
    0x80000034 (0x00940733) x14 0x0000000f
    0x80000038 (0x00008067)
    0x8000001c (0x00100213) x4  0x00000001
    0x80000020 (0x00000317) x6  0x80000020
    0x80000024 (0x01c30313) x6  0x8000003c
    0x80000028 (0x000300e7) x1  0x8000002c
    0x8000003c (0x409407b3) x15 0x00000005
    0x80000040 (0x00008067)
    0x8000002c (0x00100293) x5  0x00000001
    0x80000030 (0x0140006f)
```

JAL-JALR Golden Model Test Log

```
    0x80000000 (0x00000013)
    0x80000004 (0x00000013)
    0x80000008 (0x00000213) x4  0x00000000
    0x8000000c (0x00000293) x5  0x00000000
    0x80000010 (0x00a00413) x8  0x0000000a
    0x80000014 (0x00500493) x9  0x00000005
    0x80000018 (0x01c000ef) x1  0x8000001c
    0x80000034 (0x00940733) x14 0x0000000f
    0x80000038 (0x00008067)
    0x8000001c (0x00100213) x4  0x00000001
    0x80000020 (0x00000317) x6  0x80000020
    0x80000024 (0x01c30313) x6  0x8000003c
    0x80000028 (0x000300e7) x1  0x8000002c
    0x8000003c (0x409407b3) x15 0x00000005
    0x80000040 (0x00008067)
    0x8000002c (0x00100293) x5  0x00000001
    0x80000030 (0x0140006f)
```

JAL-JALR Simulation Summary (Verilator)

```
    ------------------------------------------------------------
    SUCCESS: Test End Reached (PC: 0x80000044)
    ------------------------------------------------------------
    - tb/tb.sv:67: Verilog $finish
    0x80000044 (0x0000006f)
    ------------------------------------------------------------
    Total Cycles: 32
    Committed Instructions: 17
    CPI: 1.8824
    ------------------------------------------------------------
    - S i m u l a t i o n   R e p o r t: Verilator 5.037 devel
    - Verilator: $finish at 84ps; walltime 0.002 s; speed 52.535 ns/s
    - Verilator: cpu 0.002 s on 1 threads; alloced 121 MB
```

Figure. JAL-JALR Test Log Karşılaştırması ve Simülasyon Özeti

STORE-LOAD

STORE-LOAD Test Assembly Kodu

```
.global _start
_start:
    nop
_init:
    nop
main:
    li  x1,0x80001000   # base address değeri olarak belirledik, bu adrese göre yazma okuma yapıcaz adresler çakışmasın diye.
test_signed_unsigned:
    li  x5,-1 # x5 register'ına 0xFFFFFFFF yazdık.
    sb  x5,0(x1) # x1 register'ı içerisindeki base adres değeri üzerine 0 offsetini ekleyerek yani adresin kendisine x5 register'ı içindeki veriyi yazdık.
    lb  x6,0(x1) # x1 register'ı içindeki base adres değerinin kendisini kullanarak load yaptık ve veriyi x6'ya yükledik.  x6 = 0xFFFFFFFF olmalı.
    lbu x7,0(x1) # x7 = 0x000000FF olmalı çünkü lbu unsigned olduğu için veriyi 0 ile genişletir. lb veriyi işaret biti ile genişletiyor.
test_little_endian_write_read:
    li x11,0x11223344
    sw x11,4(x1) # x11 register'ı içerisindeki veriyi base address + offset(4) adresine yazdık yani 4 byte ilerledik 1 satır atlamış olduk, dizinin 2. elemanı gibi düşünelim .
                 # dikkat edelim, yazma işlemi little-endian yani sağdan sola doğru yapılır bu durumda adrese veri 44 33 22 11 şeklinde yerleşir.

    lb x12,4(x1) # x12 = 0x44 olmalı
    lb x13,5(x1) # x13 = 0x33 olmalı
    lb x14,6(x1) # x14 = 0x22 olmalı
    lb x15,7(x1) # x15 = 0x11 olmalı
test_parca_parca_yazma:
    li x20,0xAA
    sb x20,16(x1)
    li x21,0xBB
    sb x21,17(x1)
    li x22,0xCC
    sb x22,18(x1)
    li x23,0xDD
    sb x23,19(x1)
    lw x24,16(x1) # beklenen değer 0xDDCCBBAA
test_end:
    j test_end

```

STORE-LOAD

STORE-LOAD Test Log Karşılaştırması ve Simülasyon Özeti

STORE-LOAD Test Log

```
0x80000008 (0x800010b7) x1  0x80001000
0x8000000c (0xfff00293) x5  0xffffffff
0x80000010 (0x00508023) mem 0x80001000 0xff
0x80000014 (0x00008303) x6  0xffffffff mem 0
     x80001000
0x80000018 (0x0000c383) x7  0x000000ff mem 0
     x80001000
0x8000001c (0x112235b7) x11 0x11223000
0x80000020 (0x34458593) x11 0x11223344
0x80000024 (0x00b0a223) mem 0x80001004 0
     x11223344
0x80000028 (0x00408603) x12 0x00000044 mem 0
     x80001004
0x8000002c (0x00508683) x13 0x00000033 mem 0
     x80001005
0x80000030 (0x00608703) x14 0x00000022 mem 0
     x80001006
0x80000034 (0x00708783) x15 0x00000011 mem 0
     x80001007
0x80000038 (0x0aa00a13) x20 0x000000aa
0x8000003c (0x01408823) mem 0x80001010 0xaa
0x80000040 (0x0bb00a93) x21 0x000000bb
0x80000044 (0x015088a3) mem 0x80001011 0xbb
0x80000048 (0x0cc00b13) x22 0x000000cc
0x8000004c (0x01608923) mem 0x80001012 0xcc
0x80000050 (0x0dd00b93) x23 0x000000dd
0x80000054 (0x017089a3) mem 0x80001013 0xdd
0x80000058 (0x0100ac03) x24 0xddccbbaa mem 0
     x80001010
```

STORE-LOAD Golden Model Test Log

```
0x80000008 (0x800010b7) x1  0x80001000
0x8000000c (0xfff00293) x5  0xffffffff
0x80000010 (0x00508023) mem 0x80001000 0xff
0x80000014 (0x00008303) x6  0xffffffff mem 0
     x80001000
0x80000018 (0x0000c383) x7  0x000000ff mem 0
     x80001000
0x8000001c (0x112235b7) x11 0x11223000
0x80000020 (0x34458593) x11 0x11223344
0x80000024 (0x00b0a223) mem 0x80001004 0
     x11223344
0x80000028 (0x00408603) x12 0x00000044 mem 0
     x80001004
0x8000002c (0x00508683) x13 0x00000033 mem 0
     x80001005
0x80000030 (0x00608703) x14 0x00000022 mem 0
     x80001006
0x80000034 (0x00708783) x15 0x00000011 mem 0
     x80001007
0x80000038 (0x0aa00a13) x20 0x000000aa
0x8000003c (0x01408823) mem 0x80001010 0xaa
0x80000040 (0x0bb00a93) x21 0x000000bb
0x80000044 (0x015088a3) mem 0x80001011 0xbb
0x80000048 (0x0cc00b13) x22 0x000000cc
0x8000004c (0x01608923) mem 0x80001012 0xcc
0x80000050 (0x0dd00b93) x23 0x000000dd
0x80000054 (0x017089a3) mem 0x80001013 0xdd
0x80000058 (0x0100ac03) x24 0xddccbbaa mem 0
     x80001010
```

STORE-LOAD Simulation Summary (Verilator)

```
----------------------------------------------------------------
SUCCESS: Test End Reached (PC: 0x8000005c)
----------------------------------------------------------------
- tb/tb.sv:67: Verilog $finish
0x8000005c (0x0000006f)
----------------------------------------------------------------
Total Cycles: 28
Committed Instructions: 23
CPI: 1.2174
----------------------------------------------------------------
- S i m u l a t i o n   R e p o r t: Verilator 5.037 devel
- Verilator: $finish at 76ps; walltime 0.001 s; speed 65.702 ns/s
- Verilator: cpu 0.001 s on 1 threads; alloced 121 MB
```

Figure. STORE-LOAD Test Log Karşılaştırması ve Simülasyon Özeti

UPPER IMMEDIATE

UPPER IMMEDIATE TEST ASSEMBLY KODU

```
.global _start

_start:
    nop
_init:
    nop

main:
    li x10,0 # LUI Kontrol Sinyali
    li x11,0 # AUIPC Kontrol Sinyali

#////////////////////////////////////// LUI TEST ////////////////////////////////////////

    lui x1, 0x12345 # x1'in üst 20 bitine 0x123456 değerini ekledik, şu anda x1 register'ı 0x12345000 olmalı.
    li  x2, 0x12345000 # x2 değerine manuel olarak 0x12345000 değerini yükledik x1 ile karşılaştırıcaz.

    bne x1,x2,lui_fail # eğer eşit değillerse lui doğru çalışmamıştır ve lui_fail'e atlar.
    li x10,1 # eğer bne çalışmadıysa lui doğru çalışmıştır ve x10 register'ını 1 yaparak bunu doğruluyoruz.
    j test_auipc

lui_fail:
    li x10,2 # eğer lui fail'e atlandıysa x10 register'ını 2 yapıyoruz. 2 yapmamızın sebebi baştaki 0 değeri ile çakışma olmasın ki kesin atlandığını anlayalım.

#////////////////////////////////////// AUIPC TEST ////////////////////////////////////////

test_auipc:

    auipc_label: # bu senaryoda pc'yi 0 offset ile toplucaz yani direkt pc'nin kendisini register'a yüklicez.

        auipc x3,0 # bu komut mevcut pc(program counter) değerini 0 offset ile toplar ve x3 register'ına yazar yani direkt pc'nin kendisi x3'e yazılır.
        la    x4,auipc_label # bu komut ile assembly tarafından hesaplanan auipc_label adresini x4'e yüklüyoruz, x3 ile karşılaştırıcaz.

        bne x3,x4,auipc_fail

    auipc_offset_label: # bu senaryoda ise pc'yi herhangi bir offset değeri ile toplucaz bakalım auipc toplama işini de doğru yapıyor mu?

        auipc x5,1 # burada mevcut pc değerini 0x1000(4096) ile toplayıp x5 register'ına yazdık. 1 vermemizin sebebi auipc ve lui komutlarının mantığı ile alakalı.
                   # auipc ve lui komutları verdiğimiz offset değerini "12 bit sola kaydırarak" register'a yazar. 1 verdiğimizde aslında ...1 0000 0000 0000
                   # yapmış oluyoruz.

        la    x6,auipc_offset_label # auipc_offset_label adresinin assembly tarafından hesaplanmış halini la ile yükledik.
        li    x7,0x1000 # x7 register'ına 0x1000(4096) immediate değerini yüklüyoruz.
        add   x8,x6,x7  # x8 register'ına yüklediğimiz değer x5 register'ına auipc ile yüklediğimiz değerle karşılaştırıcaz, bakalım aynı mı

        bne x5,x8,auipc_fail # eğer eşit değillerse auipc_fail'e atlıcak bu durumda auipc doğru çalışmamış demektir.
        li  x11,1 # eğer bne komutu çalışmadıysa buraya gelir bu durumda auipc doğru çalışmıştır, x11 değerini 1 yapıyoruz.
        j test_end

    auipc_fail:
        li x11,2 # eğer buraya düştüysek auipc yanlış çalışmıştır x11 register'ını 2 yapıyoruz.

test_end:
    j test_end


```

UPPER IMMEDIATE

UPPER IMMEDIATE Test Log Karşılaştırması ve Simülasyon Özeti

UPPER IMMEDIATE Test Log

```
0x80000000 (0x00000013)
0x80000004 (0x00000013)
0x80000008 (0x00000513) x10 0x00000000
0x8000000c (0x00000593) x11 0x00000000
0x80000010 (0x123450b7) x1  0x12345000
0x80000014 (0x12345137) x2  0x12345000
0x80000018 (0x00209663)
0x8000001c (0x00100513) x10 0x00000001
0x80000020 (0x0080006f)
0x80000028 (0x00000197) x3  0x80000028
0x8000002c (0x00000217) x4  0x8000002c
0x80000030 (0xffc20213) x4  0x80000028
0x80000034 (0x02419263)
0x80000038 (0x00001297) x5  0x80001038
0x8000003c (0x00000317) x6  0x8000003c
0x80000040 (0xffc30313) x6  0x80000038
0x80000044 (0x000013b7) x7  0x00001000
0x80000048 (0x00730433) x8  0x80001038
0x8000004c (0x00829663)
0x80000050 (0x00100593) x11 0x00000001
0x80000054 (0x0080006f)
```

UPPER IMMEDIATE Golden Model Test Log

```
0x80000000 (0x00000013)
0x80000004 (0x00000013)
0x80000008 (0x00000513) x10 0x00000000
0x8000000c (0x00000593) x11 0x00000000
0x80000010 (0x123450b7) x1  0x12345000
0x80000014 (0x12345137) x2  0x12345000
0x80000018 (0x00209663)
0x8000001c (0x00100513) x10 0x00000001
0x80000020 (0x0080006f)
0x80000028 (0x00000197) x3  0x80000028
0x8000002c (0x00000217) x4  0x8000002c
0x80000030 (0xffc20213) x4  0x80000028
0x80000034 (0x02419263)
0x80000038 (0x00001297) x5  0x80001038
0x8000003c (0x00000317) x6  0x8000003c
0x80000040 (0xffc30313) x6  0x80000038
0x80000044 (0x000013b7) x7  0x00001000
0x80000048 (0x00730433) x8  0x80001038
0x8000004c (0x00829663)
0x80000050 (0x00100593) x11 0x00000001
0x80000054 (0x0080006f)
```

UPPER IMMEDIATE Simulation Summary (Verilator)

```
------------------------------------------------------------
SUCCESS: Test End Reached (PC: 0x8000005c)
------------------------------------------------------------
- tb/tb.sv:67: Verilog $finish
0x8000005c (0x0000006f)
------------------------------------------------------------
Total Cycles: 30
Committed Instructions: 21
CPI: 1.4286
------------------------------------------------------------
- S i m u l a t i o n   R e p o r t: Verilator 5.037 devel
- Verilator: $finish at 80ps; walltime 0.002 s; speed 49.299 ns/s
- Verilator: cpu 0.002 s on 1 threads; alloced 121 MB
```

Figure. UPPER IMMEDIATE Test Log Karşılaştırması ve Simülasyon Özeti

Bibliography I

[1] D. M. Harris and S. L. Harris, *Digital Design and Computer Architecture: RISC-V Edition*. Morgan Kaufmann, 2021.

[2] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 6th. Morgan Kaufmann, 2017.

[3] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design RISC-V Edition: The Hardware Software Interface*, 2nd. Morgan Kaufmann, 2020.

[4] Python Software Foundation, ***Python programming language***, `https://www.python.org`, Accessed: 2025, 1991.

[5] Verilator Development Team, ***Verilator: Open-source systemverilog simulator***, `https://www.veripool.org/verilator`, Accessed: 2025, 2003.

[6] RISC-V International, ***Spike: Risc-v isa simulator***, `https://github.com/riscv-software-src/riscv-isa-sim`, Accessed: 2025, 2014.

[7] RISC-V GNU Toolchain Contributors, ***Risc-v gnu compiler toolchain***, `https://github.com/riscv-collab/riscv-gnu-toolchain`, Accessed: 2025, 2014.

[8] T. Bybell, ***Gtkwave: Waveform viewer***, `http://gtkwave.sourceforge.net`, Accessed: 2025, 1999.