Universiteti i Prishtinës
Fakulteti i Inxhinierisë Elektrike dhe Kompjuterike
Arkitektura e Kompjuterëve, 2023/2024
Prof. Valon Raça & Synim Selimi

Shpat Mjeku

Detyra e dytë

Dizajnimi i një CPU 16-bitëshe (Single-Cycle)

1. Hyrje

Qëllimi i detyrës është dizajnimi i një CPU 16-bitëshe (Single-Cycle) në Verilog, ku secili detaj i sistemit së CPU-së është i listuar në kërkesat e detyrës. CPU duhet të mbështet instruksione të formatit R dhe I. Nuk do të mbështet instruksione të formatit J, si jump.

Dizajnimi i CPU-së 16-bitëshe single-cycle përbëhet nga dy pjesë kryesore:

1. **Datapath (DP)**: Pjesa që përmban elementet harduerike të nevojshme për të ekzekutuar instruksionet dhe për të manipuluar të dhënat, siç janë ALU, regjistrat, dhe linjat e bus-it.
2. **Control Unit (CU)**: Pjesa që interpreton opcode-in e instruksionit dhe prodhon sinjale të kontrollit për të dirigjuar rrjedhën e të dhënave dhe operacionet e DP.

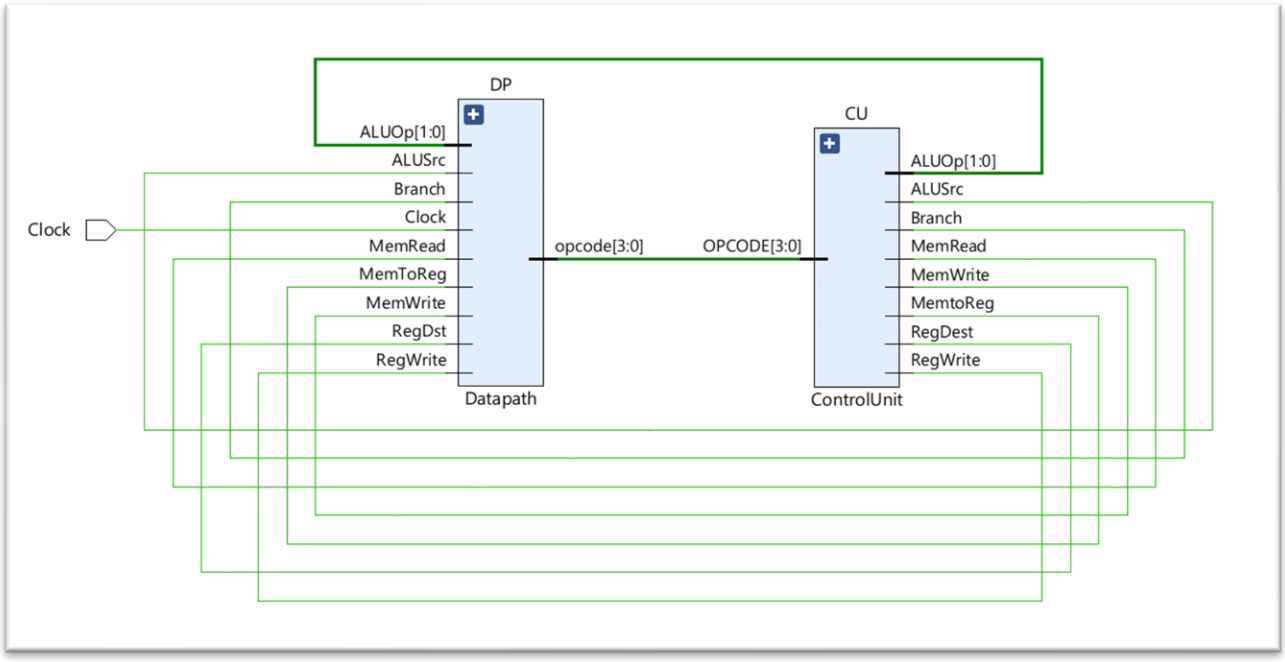

Figura 1. Diagrami ilustron arkitekturën e CPU-së që kam dizajnuar, përfshirë njësinë Datapath dhe njësinë Control-Unit.

2. Dizajni

Specifikimi i formatit të instrusioneve të mbështetura:

Formati R

OPCODE				RS		RT		RD		SHAMT				FUNCT	
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Instruksionet e formatit R OPCODE do ta kenë 0XXX (XXX -> 000-111). Jo të gjitha instruksionet e formatit R do ta kenë OPCODE-in e njejtë.

Formati I

OPCODE				RS		RT		IMMEDIATE OR ADDRESS							
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Instruksionet e formatit I OPCODE e kanë 1XXX (XXX -> 000-111).

Instruksionet e formatit I, secila vec e vec do te ketë OPCODE të ndryshëm.

Komponentët

ALU 16 Bitëshe : ALU 16-bitëshe (Arithmetic Logic Unit) është pjesa e procesorit që kryen operacione aritmetike dhe logjike (si mbledhja dhe zbritja) dhe operacione logjike (si AND, OR, XOR) mbi operandët që janë 16 bitë në gjerësi. ALU-ja 16-bitëshe formohet nga 16 ALU-ja 1-bitëshe të lidhura njëra pas tjetrës përmes metodës Ripple-Carry. Përmes 16-të ALU-ve të lidhura 1-bitëshe implementohen operacionet për instruksionet And, Or, Xor, dhe operacionet që kryhen përmes një mbledhësi (Add, Sub, Addi, Subi). Pjesërisht është i implementuar edhe operacion SLTI (set less than immidiate) por që funksioni total implementohet në vetë në ALU16bit përmes një moduli special. Po ashtu kam dhe dy module tjera speciale për ti implementuar operacionet SLL dhe SRA. Në ALU është një cast statement i cili zgjedh bazë Op-it se rezultati a të jetë rezultati nga alu-të një bitëshe apo nga modulet speciale të SLL dhe SRA.

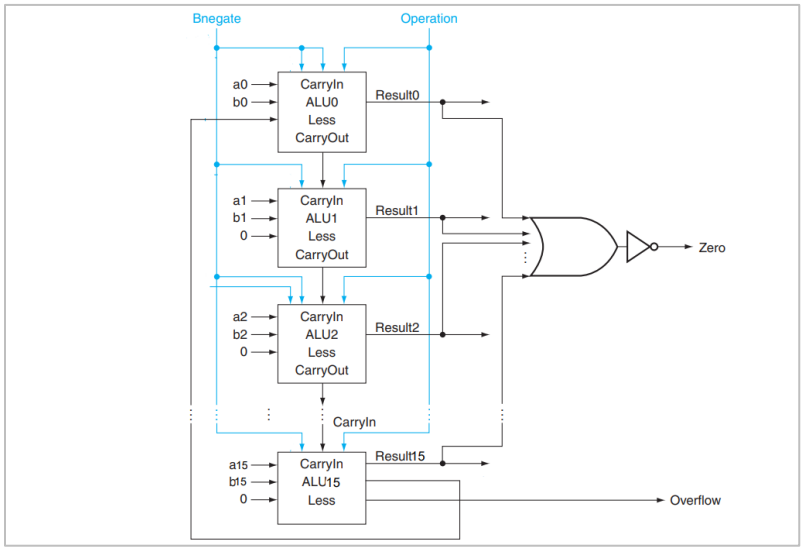

Figura 2. Dizajni i përafertë i ALU 16-bitëshe

ALU Control Unit: Bazë të ALU Op-it i cili vie nga njësia kontrolluese dhe OPCODE dhe Funct përcaktojmë përmes if-else statements në ALU se cilin operacion dëshirojmë ta kryejmë.

```
module ALUControl(
    input [1:0] ALUOp,
    input [1:0] Funct,
    input [3:0] OPCODE,
    output reg [3:0] ALUCtrlTeli
);

always @(ALUOp, Funct, OPCODE)
begin
    if (ALUOp == 2'b00) begin
      ALUCtrlTeli = 4'b0100; // LW+SW
    end
    else if (ALUOp == 2'b01) begin
        ALUCtrlTeli = 4'b1100; // BEQ
    end
    else if (ALUOp == 2'b10) begin // R-Format
      if (OPCODE == 4'b0000) begin // Kqyre OPCODE 0000 tani Funct
            if (Funct == 2'b00) ALUCtrlTeli = 4'b0000; // AND
            else if (Funct == 2'b01) ALUCtrlTeli = 4'b0010; // OR
            else if (Funct == 2'b10) ALUCtrlTeli = 4'b0011; // XOR
        end
        else if (OPCODE == 4'b0001) begin // Kqyre OPCODE 0001 tani Funct
            if (Funct == 2'b00) ALUCtrlTeli = 4'b0100; // ADD
            else if (Funct == 2'b01) ALUCtrlTeli = 4'b1100; // SUB
        end
        else if (OPCODE == 4'b0010) begin // Kqyre OPCODE 0010 tani Funct
            if (Funct == 2'b00) ALUCtrlTeli = 4'b0110; // SLL
            else if (Funct == 2'b01) ALUCtrlTeli = 4'b0111; // SRA
        end
    end
    else if (ALUOp == 2'b11) begin // I-format, Kqyre OPCODE
            if (OPCODE == 4'b1001) ALUCtrlTeli = 4'b0100; //  ADDI
            else if (OPCODE == 4'b1010) ALUCtrlTeli = 4'b1100; //  SUBI
            else if (OPCODE == 4'b1011) ALUCtrlTeli = 4'b0001; //  SLTI
    end
end
endmodule
```

Figura 3. Kodi në verilog i modulit së ALU Control Unit.

Datamemory : Data memory është 128 byte dhe funksionon në mënyrë që të bëjë leximin dhe shkrimin e të dhënave në dhe nga memoria. Ka katër hyrje, një për adresën e vendndodhjes së memories hyrëse, një për të dhënat qe do shkruhen dhe MemWrite dhe MemRead. Dalje është e dhëna e lexuar nga memoria sipas adreses që po e përcaktojmë. Leximet dhe shkrimet kontrollohen nga MemRead dhe MemWrite. Datamemory është big - endian.

Register File: Përmban katër regjistra, regjistri $zero (adresa 00), $r1 (01), $2 (10) dhe regjistri $r3 me adresen (11). Ka tre hyrje dy bitëshe për përcaktimin e regjistrave RS, RT, RD. Ka një hyrje 16 bitëshe për të shkruar në regjistrin RD dhe ka dy dajle 16 bitëshe për të lexuar të dhënat nga regjistrat e përcaktuar në RS dhe RT.

Regjistrin $zero përmes një if-statement nuk e lejojmë të mbishkruhet, ngjajshëm sikur në MIPS.

```
reg[15:0] Registers[3:0];
//Reseto te gjithe regjistrat ne 0
integer i;
initial
begin
   for(i = 0; i < 4; i = i + 1)
      Registers[i] <= 16'd0;
end

//Shkruaj ne regjister
always @(posedge Clock)
  begin
   if (RegWrite && RD != 2'b00) begin
      Registers[RD] <= WriteData;
  end
end

//lexo regjistrat ReadData1, ReadData2
assign ReadRS = Registers[RS];
assign ReadRT = Registers[RT];
```

Fragment 1. Pjesë e kodit së modulit së Register File

Instrucion Memory: Gjithashtu njejtë sikur tek Data Memory është 128 byte. Adresa 0 deri në 9 byte është e rezervuar prandaj edhe tek program-counteri duhet të fillojme nga adresa e 10-të ti lexojmë instruksionet që dëshirojmë ti ekzekutojmë. Instruction Memory është vetëm Read-Only. Hyrje 16 bitëshe nga PC për përcaktimin e adresës së instruksionit. Dalje 16 bitëshe për leximin e instruksionit.

```
$readmemb("InstructionMemory.mem", instrMem);

assign Instruction[15:8] = instrMem[PCAddress];
assign Instruction[7:0] = instrMem[PCAddress + 16'd1];
```

Figura 4. Leximi i instruksioneve nga Instruction Memory ku një instruction është 16-bit.

Multiplekserët : Janë disa lloje të ndryshme të multiplekserëve të cilët duhen për selektim ndërmjet sinjalëve. Janë dy multiplekserë në Alu 1 bitëshe, njëri mux 2në1 duhet kur kemi zbritje ndërmjet A-së dhe B-së ndërsa muxi tjetër 8në1 duhet për selektimin e operacionit And, Or, Xor, Add/Addi/Sub/Subi dhe Less për SLTI instruksionin.

Është multiplekserin mux2në1 2-bit i cili shërben për përcaktimin nëse RD është RD (te R-formati) apo

RD = RT (te I-formati). Po ashtu është dhe multiplekserin mux 2në1 16-bit prej të cilit formohen tri instanca në datapath për selektim të sinjaleve si njëri prej tyre mux M2 bën përcaktimi nëse hyrja e para ALU eshte Regjstri 2 i RF apo vlera imediate e instruksionit.

Mbledhësit: Janë disa lloje. Është mbledhësin e plotë i cili shfrytëzohet në alu 1 bitëshe, dhe përmes tij kryhen operacionet e mbledhjes dhe zbritjes.

Është mbledhësin mbledhësiCLA (carry look ahead) të cilin e kam përdor si mbledhës të vecantë për program-counter, në rastin kur kemi me mbledhë me 2 byte për adresën e ardhshme për të gjitha instruksionet përvec beq për të cilin po ashtu krijohet një instance e tijë për të kalukulu adresën në qoftë se kemi instrukionin BEQ.

Mbledhës tjetër është RippleCarryAdder të cilin e përdorim për implementimin e instruksionit SLTI.

Control Unit: Njësia e kontrollit si hyrje ka opcodin nga instruksioni bitat [15:12], ku pastaj ajo dirigjon të gjithë procedorin varësisht nga opcode që merr, duke i aktivizu qarqet që duhen për instruksionin i cili zbatohet. Përmes case-statement nga opcodi, përcakton daljet RegDest, Branch, MemRead, MemtoReg, [1:0] ALUOp, MemWrite, ALUSrc, RegWrite.

SLTI moduli: E instancion një RippleCarryAdder ku përmes tij kryen zbritjen ndërmjet operandëve A dhe B. Dalje ka rezulatin SLTI 1 bit, i cili është most significant bit pas zbritjes, por nese ka overflow kam implementu logjikën që siguron rezultatin e saktë. Pastaj në Alu 16 bit, ruhet rezultati përmes alu-ve 1 bit që least significant bit të jetë rezultati i SLTI, dhe 15 bitat tjerë zero.

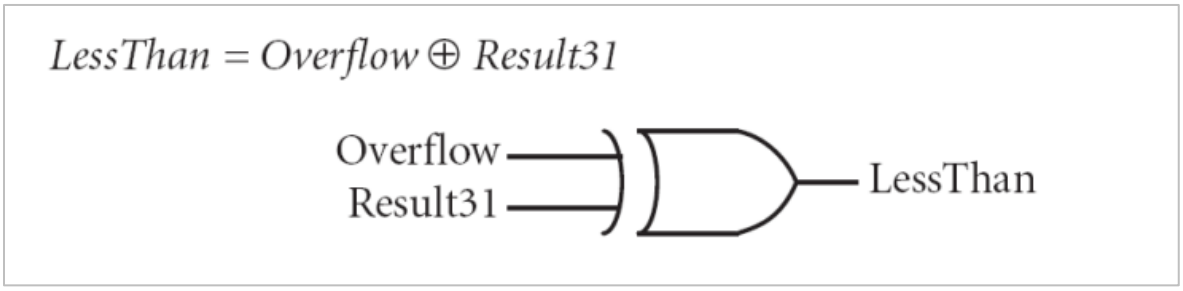

Figura 5. Rezultati nga slti, mund të jetë gabim nëse ka overflow zbritja prandaj e sigurojm të jetë i saktë përmes kësaj logjike në figurë.

BarrelShifterSLL, **BarrelShifterSRA:** Modulet për implementimin e instruksioneve bonus të shtyerjes. Kem shtyrjen majtas logjike, dhe shtyrjen djathtas aritmetike. **SLL** – shtyrja logjike majtas është sikur shumëzimi me 2. **SRA** - shtyrja aritmetike djathtas, bitat e ri e ruajnë vlerën e njejtë të sign bit dhe është pothuajse sikur pjestimi me 2.

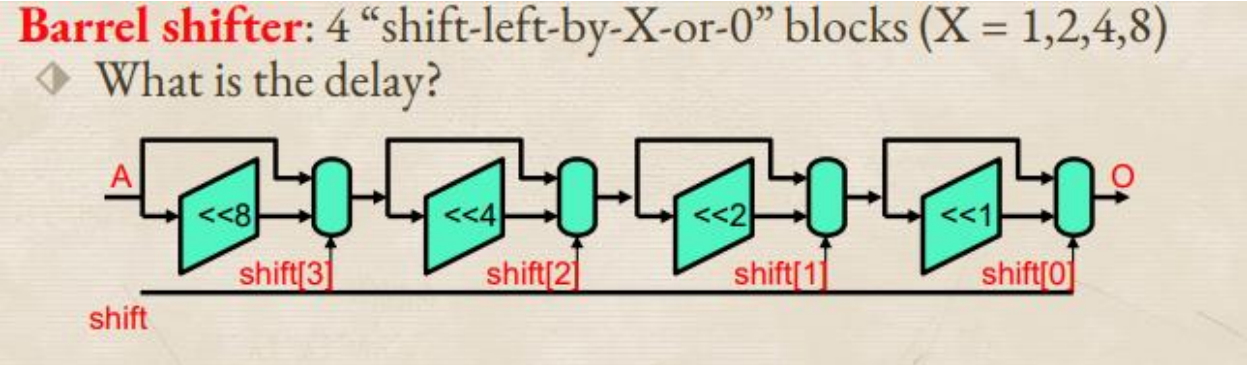

Figura 6. Barrelshifter (marrë nga Computer Architechture slides).

Figura 7. SLL DHE SRA

BarrelShifterSLL mundëson shtyrjen prej 0 der ne 15 pozita majtas. Shembull për fazën e parë:

Faza 1 (faza1): Nëse në instruksion shamt[0] është vendosur vlera 1, hyrja A zhvendoset majtas për 1 pozicion (e barabartë me shumëzimin me 2). Least signficiant bit bëhet 0. Kur në instruksion shamt[0] dhe shamt[1] janë 1, (A) do të shtyhet majtas për një total prej 3 pozicioneve (1 + 2), dhe bitat e ri least significant do të bëhen zero. Rezultati do të jetë i barabartë me shumëzimin e A me 2^3, ose 8.

BarrelShifterSRA mundëson shtyrjen prej 0 deri në 15 pozita djathtas. Shembull për fazën e parë:

Faza 1 (faza1): Nëse në instruksion shamt[0] është vendosur vlera 1, hyrja A zhvendoset djathtas për 1 pozicion dhe most significant bit (biti i shenjës) ruhet në pozicionin e lartë. Në këtë mënyrë, biti i shenjës kopjohet në pozicionin e ri bosh, duke krijuar një shtyrje aritmetike djathtas. Nëse shamt[0] dhe shamt[1] janë të dyja 1, A do të shtyhet djathtas për një total prej 3 pozicioneve (1 + 2), dhe bitat e ri least significant mbushen me bitin e shenjës signBit. Rezultati është i barabartë me zhvendosjen e A djathtas për 2^3, ose 8 pozicione, dhe gjithmonë duke ruajtur bitin e shenjës për të mos ndrruar vlerën e shenjës së numrit origjinal. Ngjajshëm për fazat tjera për të dy instruksionet e shtyrjes.

```
module barrelShifterSLL(
  input [15:0] A,
  input [3:0] shamt,
  output reg [15:0] O
);
    // telat per lidhjen e fazave te shifterit
    wire [15:0] faza1, faza2, faza3, faza4;

  // faza 1: boje Shift per 1 nese shamt[0] eshte set
    assign faza1 = shamt[0] ? {A[14:0], 1'b0} : A;

    // faza 2: boje Shift per 2 nese shamt[1] eshte set
    assign faza2 = shamt[1] ? {faza1[13:0], 2'b00} : faza1;

    // faza 3: boje Shift per 4 nese shamt[2] eshte set
    assign faza3 = shamt[2] ? {faza2[11:0], 4'b0000} : faza2;

    // faza 4: boje Shift per 8 nese shamt[3] eshte set
    assign faza4 = shamt[3] ? {faza3[7:0], 8'b00000000} : faza3;

    // rezultati
    always @* begin
        O = faza4;
    end
endmodule
```

Fragment 2. Moduli barrelShifterSLL

Datapath: Pjesa më e rëndësishme sepse përmes telave lidh dhe fut në funksion të gjithë komponentët duke mundësu lëvizjen e të dhënave.

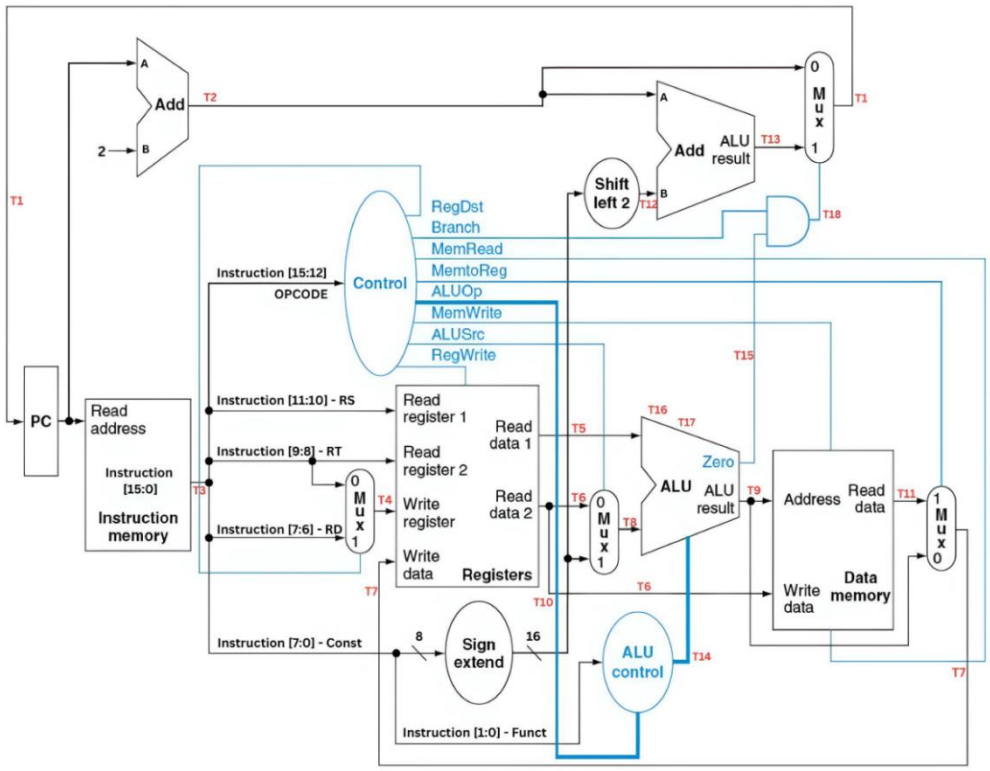

Figura 8. Pamja e dizajnit së Datapath-it

```
wire [15:0]  pc_next;        // TELI: T1
wire [15:0]  pc2;           // TELI: T2
wire [15:0]  instruction;     // TELI: T3
wire [1:0]   mux_regfile;     // TELI: T4
wire [15:0]  readData1;       // TELI: T5
wire [15:0]  readData2;       // TELI: T6
wire [15:0]  writeData;       // TELI: T7
wire [15:0]  mux_ALU;         // TELI: T8
wire [15:0]  ALU_Out;         // TELI: T9
wire [15:0]  Zgjerimi;        // TELI: T10
wire [15:0]  memToMux;        // TELI: T11
wire [15:0]  shifter2beq;     // TELI: T12
wire [15:0]  beqAddress;      // TELI: T13
wire [3:0]   ALUCtrl;         // TELI: T14

wire zerof;                 // TELI: T15
wire overflow;              // TELI: T16
wire carryout;              // TELI: T17
wire andMuxBranch;          // TELI: T18
```

Fragment 3. Telat që shfrytëzohen në datapath module, shiko gjithashtu figurën 8.

CPU moduli: Input ka clock-un. Instancon datapath-in dhe njësin e kontrollit.

Kemi dhe modulet për AND, Or, XOR, të cilat i kam bërë pasi që është specifikuar në detyre që të krijohet për secilin funksion të krijohet moduli i posaçëm strukturor apo behavior-ist

```
sub $r1, $zero, $zero
lw $r2, 4($r1)
xor $r1, $r2, $r3
beq $r2, $zero, kercimi
and $r3, $r1, $r2
kercimi: sw $r3, 4($r2)
slti $r2, $r3, 2   #vetëm nëse është implementuar instruksioni nga bonus
sll $r1, $r2, 3    #vetëm nëse është implementuar instruksioni nga bonus
sra $r2, $r1, 9    #vetëm nëse është implementuar instruksioni nga bonus
```

Figura 9. Instruksionet që duhen ekzekutuar.

Në $r1 pas instruksionit të zbritjes ruhet zero. Pastaj shkojmë në adresën 0+4 në datamemory e bëjm load word atë vlerë në regjistrin $r2 (ku duhet ta kqyrim adresën 4,5 në datameory pasi që sistemi është 16 bit dhe vlerën e caktojmë vet). E vendosim shembull vlerën 15 . Pastaj përmes xor ruhet në $r1 vlera 15. Instruksioni beq, nuk kalon tek kërcimi pasi që $r2 != zero. Përmes instrksionit And ruhet vlera 15 në $r3. Bëhet store word $r3 në datamemory në adresën 4+15 =19 pas ekzekutimit do nodhet vlera 15. Tek instruksionit SLTI, $r3 = 15 > 2, prandaj në $r2 ruhet zero. Me SLL të $r2 që është zero rezultati është zero prandaj në $r1 ruhet vlera zero. Njejtë tani me SRA të regjistrit $r1 që është tani zero, ruhet vlera zero në $r2. Prandaj për të parë ndyshimin me përdorim të instruksioneve SLL dhe SRA kam krijuar një instruction memory file tjetër përmes të cilit kur të ekzekutohet shofim rezultate të instruksioneve SLL dhe SRA.

3. Ekzekutimi

Detyrën e kam realizuar në EDA Playground dhe Vivado. Në linkun më poshtë gjendet në EDAPlaygorund fajlli i testbench CPU, dhe kur e klikojmë run e kam setup ashtu që të hapet DataMemory file. Shihen ndryshimet në DatamMemory pas ekzekutimit së testit së CPU-së.

Linku i parë ka kodin e saktë me instruksione sikur të kërkesave të detyrës.

https://www.edaplayground.com/x/c_UM

Linku i dytë ka fajllin e instruction memory me instruksione më ndryshe që kur të ekzekutohet, shfaqen ndryshimet në data memory nga përdorimi i instruksioneve storeword për regjistrat që marrin vlerë ndryshe pas përdorimit të instruksioneve SLL dhe SRA.

https://www.edaplayground.com/x/PKLW