# CS224
# Lab No: 4
# Section No: 4

# Emine İrem Esendemir
# 22202103

b) Determine the assembly language equivalent of the machine codes given in the imem module in the "Complete MIPS model.txt" file posted on Moodle for this lab. In the given SystemVerilog module for imem, the hex values are the MIPS machine language instructions for a small test program. Disassemble these codes into the equivalent assembly language instructions and give a 3-column table for the program, with one line per instruction, containing its location, machine instruction (in hex) and its assembly language equivalent. [Note: you may disassemble by hand or use a program tool.]

| LOCATION | MACHINE INSTRUCTION | ASSEMBLY LANGUAGE |
|---|---|---|
| 00000000 | 20020005 | addi $2,$0,5 |
| 00000004 | 2003000c | addi $3,$0,12 |
| 00000008 | 2067fff7 | addi $7,$3,-9 |
| 0000000c | 00e22025 | or $4,$7,$2 |
| 00000010 | 00642824 | and $5,$3,$4 |
| 00000014 | 00a42820 | add $5,$5,$4 |
| 00000018 | 10a7000a | beq $5,$7,0x00000044 |
| 0000001c | 0064202a | slt $4,$3,$4 |
| 00000020 | 10800001 | beq $4,$0,0x00000028 |
| 00000024 | 20050000 | addi $5,$0,0 |
| 00000028 | 00e2202a | slt $4,$7,$2 |
| 0000002c | 00853820 | add $7,$4,$5 |
| 00000030 | 00e23822 | sub $7,$7,$2 |
| 00000034 | ac670044 | sw $7,68($3) |
| 00000038 | 8c020050 | lw $2,80($0) |
| 0000003c | 08000011 | j 0x00000044 |
| 00000040 | 20020001 | addi $2,$0,1 |
| 00000044 | ac020054 | sw $2,84($0) |
| 00000048 | 08000012 | j 0x00000048 |

c) Register Transfer Level (RTL) expressions for each of the new instructions that you are adding (see list below for your section), including the fetch and the updating of the PC.

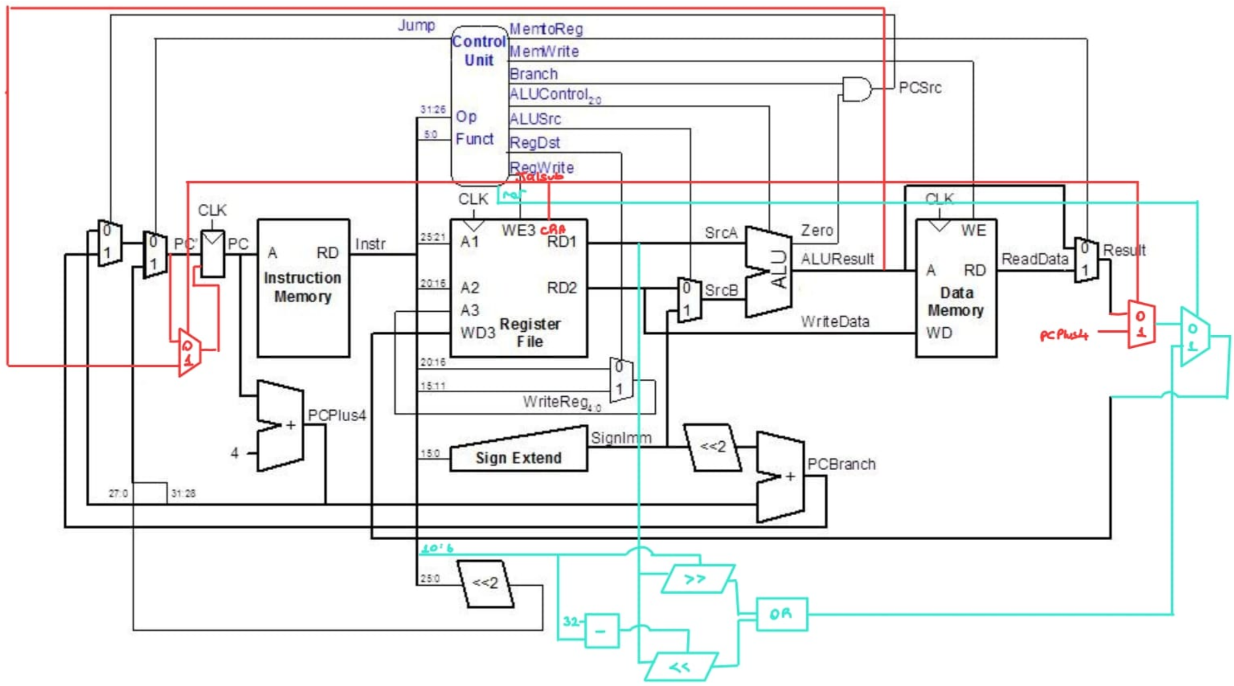
**jalsub (Jump and link with subtraction):** These R-type instructions jump to the locations calculated by addition or subtraction of the registers rs and rt. While jumping the 31st register ($ra) is changed to PC + 4 like a normal jal instruction. Usage: jalsub rs, rt (You can assume rd and shamt are zero in this instruction).
Examples: jalsub $a2, $a1 (when $a2 is 0x20 and $a1 is 0x04, $ra will be PC +4 and PC will be 0x18)

```
IM[PC];
RF[ra] ← PC +4;
PC ← RF[rs] - RF[rt];
```

**ror**: This R-type instruction right rotates RF[rs] by the amount specified by the shamt field of the instruction. Right rotation is similar to shifting a binary value right, except the bits that would be lost during shifting are placed to the left. The result is stored in RF[rd].
Example: ror $s0, $s1, 5.
Note: It is assumed that the rt field of the instruction is 0b00000 while assembling the test programs

```
IM[PC];
RF[rd] ← (RF[rs] >> shamt) | (RF[rs] << (32 - shamt));
PC ← PC + 4;
```

d) Make any additions or changes to the datapath which are needed in order to make the RTLs for the instructions possible. The base datapath should be in black, with changes marked in red and other colors (one color per new instruction). Make your changes on “Final Datapath.png” file.



e) [5 points] Make a new row in the main control table for each new instruction being added, and if necessary add new columns for any new control signals that are needed (input or output). Be sure to completely fill in the table—all values must be specified. If any changes are needed in the ALU decoder table, give this table in its new form (with new rows, columns, etc). Make your changes on Table 1: Main
Decoder for Original10. The base table should be in black, with changes marked in red and other colors. {Note: if you need new ALUOp bits to encode new values, you should also give a new version of Table 2, showing the new encodings}

Table 1: Main Decoder

| Instructio n | Opcode | RegWrite | RegDst | ALUSrc | Branch | MemWrit e | MemToR eg | ALUOp | Jump | jalsub | ror |
|---|---|---|---|---|---|---|---|---|---|---|---|
| R-type | 000000 | 1 | 1 | 0 | 0 | 0 | 0 | 10 | 0 | 0 | 0 |
| lw | 100011 | 1 | 0 | 1 | 0 | 0 | 1 | 00 | 0 | 0 | 0 |
| sw | 101011 | 0 | X | 1 | 0 | 1 | X | 00 | 0 | 0 | X |
| beq | 000100 | 0 | X | 0 | 1 | 0 | X | 01 | 0 | 0 | X |
| addi | 001000 | 1 | 0 | 1 | 0 | 0 | X | 00 | 0 | 0 | X |
| j | 000010 | 0 | X | X | 0 | 0 | X | XX | 1 | 0 | X |
| jalsub | 000000 | 0 | 1 | 0 | 1 | 0 | X | 01 | 1 | 1 | 0 |
| ror | 000000 | 1 | 1 | 0 | 0 | 0 | X | XX | 0 | 0 | 1 |

f) [10 points] Write a test program in MIPS assembly language, that will show whether the new instructions are working or not, and that will confirm that all existing old instructions still continue to work. Don't use any pseudo-instructions; use only real MIPS instructions that will be recognized by the new control unit.

```
.text
#add, sub, and, or, slt, lw, sw, beq, jalsub, ror, addi

addi $4, $0, 0x18
addi $5, $0, 0x08
jalsub $4, $5
addi $7, $0, 3 #should not be executed
addi $31, $31, 6
addi $7, $31, 0
addi $2, $zero, 0x0F0F
ror $3, $2, 3  #t1 = 0xE00001E1
add $2, $0, $0
addi $2, $0, 5
addi $3, $0, 12
addi $7, $3, -9
or $4, $7, $2
and $5, $3, $4
add $5, $5, $4
beq $5, $7, 17Later # does not executed as $5≠$7
slt $4, $3, $4
beq $4, $0, 1Later
addi $5, $zero, 0 # does not executed bc of branch
slt $4, $7, $2
add $7, $4, $5
sub $7, $7, $2
sw $7, 68($3)
lw $2, 80($zero)
j 0x68 #skip next instruction
addi $2, $zero, 1
sw $2, 84($zero)
j 0x6c #infinite loop
```

g) [20 points] Write a list of the SystemVerilog modules that will need changes in order to make these new instructions part of the single-cycle MIPS processor's instruction set. For each module in the list, determine the new SystemVerilog model that will be needed in order for the instructions to be added. Give the SystemVerilog code for each module that needs to be changed.

**MIPS:**

```
module mips (input  logic        clk, reset,
             output logic[31:0]  pc,
             input  logic[31:0]  instr,
             output logic        memwrite,
             output logic[31:0]  aluout, writedata,
             input  logic[31:0]  readdata);

  logic        memtoreg, pcsrc, zero, alusrc, regdst, regwrite, jump, jalsub,
ror;
  logic [2:0]  alucontrol;

  controller c (instr[31:26], instr[5:0], zero, memtoreg, memwrite, pcsrc,
alusrc, regdst, regwrite, jump, jalsub, ror, alucontrol);

  datapath dp (clk, reset, memtoreg, pcsrc, alusrc, regdst, regwrite, jump,
jalsub, ror, alucontrol, zero, pc, instr, aluout, writedata, readdata);

endmodule
```

**CONTROLLER:**

```
module controller(input  logic[5:0] op, funct,
                  input  logic     zero,
                  output logic     memtoreg, memwrite,
                  output logic     pcsrc, alusrc,
                  output logic     regdst, regwrite,
                  output logic     jump,
                  output logic     jalsub, ror,
                  output logic[2:0] alucontrol);

   logic [1:0] aluop;
   logic       branch;

   maindec md (op, funct, memtoreg, memwrite, branch, alusrc, regdst, regwrite,
jalsub, ror, jump,  aluop);

   aludec  ad (funct, aluop, alucontrol);

   assign pcsrc = branch & zero;

endmodule
```

**MAINDEC:**

```
module maindec (
    input logic[5:0] op, funct,
    output logic memtoreg, memwrite, branch,
    output logic alusrc, regdst, regwrite, jalsub, ror, jump,
    output logic [1:0] aluop
);
```

```
    logic [10:0] controls;

    assign {regwrite, regdst, alusrc, branch, memwrite,
             memtoreg, aluop, jump, jalsub, ror} = controls;

    always_comb begin
        controls = 11'b00000000000;
        case (op)
            6'b000010: controls = 11'b00000000100; // J
            6'b000000: begin // R-type instructions
                case (funct)
                    6'b100000: controls = 11'b11000010000; // ADD
                    6'b100010: controls = 11'b11000010000; // SUB
                    6'b100100: controls = 11'b11000010000; // AND
                    6'b100101: controls = 11'b11000010000; // OR
                    6'b101010: controls = 11'b11000010000; // SLT
                    6'b110000: controls = 11'b01010001110; // JALSUB
                    6'b111000: controls = 11'b11000000001; // ROR
                    default:   controls = 11'b00000000000; // Illegal funct
                endcase
            end
            6'b100011: controls = 11'b10100100000; // LW
            6'b101011: controls = 11'b00101000000; // SW
            6'b000100: controls = 11'b00010001000; // BEQ
            6'b001000: controls = 11'b10100000000; // ADDI
            default:   controls = 11'b00000000000; // Illegal opcode
        endcase
    end
endmodule
```

**DATAPATH:**

```
module datapath (input  logic clk, reset, memtoreg, pcsrc, alusrc, regdst,
                 input  logic regwrite, jump, jalsub, ror,
             input  logic[2:0]  alucontrol,
                 output logic zero,
             output logic[31:0] pc,
               input  logic[31:0] instr,
                 output logic[31:0] aluout, writedata,
               input  logic[31:0] readdata);

  logic [4:0]  writereg;
  logic [31:0] pcnext, pcnextbr, pcplus4, pcbranch, pcnextbeforejalsub;
  logic [31:0] signimm, signimmsh, srca, srcb, resultbeforepcplus4,
resultbeforeror, rorresult, result;

  // next PC logic
  flopr #(32) pcreg(clk, reset, pcnext, pc);
  adder       pcadd1(pc, 32'b100, pcplus4);
  sl2         immsh(signimm, signimmsh);
  adder       pcadd2(pcplus4, signimmsh, pcbranch);
  mux2 #(32)  pcbrmux(pcplus4, pcbranch, pcsrc,
                      pcnextbr);
  mux2 #(32)  pcmuxbeforejalsub(pcnextbr, {pcplus4[31:28],
                    instr[25:0], 2'b00}, jump, pcnextbeforejalsub);
  mux2 #(32) pcmux(pcnextbeforejalsub, aluout, jalsub, pcnext);

// register file logic
   regfile     rf (clk, regwrite, jalsub,instr[25:21], instr[20:16], writereg,
                   result, srca, writedata);

   mux2 #(5)    wrmux (instr[20:16], instr[15:11], regdst, writereg);
   mux2 #(32)  resmux (aluout, readdata, memtoreg, resultbeforepcplus4);
```

```
   mux2 #(32)  muxwithjalsub(resultbeforepcplus4, pcplus4, jalsub, 
resultbeforeror);
   assign rorresult = ((srca >> instr[10:6]) | (srca << (32 - instr[10:6])));
   mux2 #(32) muxwithror(resultbeforeror, rorresult, ror, result);
   signext         se (instr[15:0], signimm);

  // ALU logic
   mux2 #(32)  srcbmux (writedata, signimm, alusrc, srcb);
   alu         alu (srca, srcb, alucontrol, aluout, zero);

endmodule
```

**REGFILE:**

```
module regfile (input    logic clk, we3, changeRA,
                input    logic[4:0]  ra1, ra2, wa3,
                input    logic[31:0] wd3,
                output   logic[31:0] rd1, rd2);

  logic [31:0] rf [31:0];
  initial begin
        for (int i = 0; i < 32; i = i + 1)
            rf[i] = 32'h0;
    end

  // three ported register file: read two ports combinationally
  // write third port on rising edge of clock. Register0 hardwired to 0.

  always_ff@(posedge clk)
      if(changeRA && !we3) begin
        rf[31] <= wd3;
      end
     else if (we3) begin
         rf [wa3] <= wd3;
     end


  assign rd1 = (ra1 != 0) ? rf [ra1] : 0;
  assign rd2 = (ra2 != 0) ? rf[ ra2] : 0;

endmodule
```