آوا میرمحمدمهدی 810199501 نسا عباسی مقدم 810199457

Register File:

رجیستر فایل از یک آرایه‌ی دوبعدی تشکیل شده است که شامل 16 رجیستر 32 بیتی است و در ابتدا باید هر خانه‌ی آن را برابر با شماره‌ی خانه‌اش مقداردهی اولیه کنیم که این کار در initial begin انجام می‌شود؛ همچنین در یک always که به لبه‌ی بالارونده‌ی clk و rst حساس است (rst به صورت async است) درصورتی که سیگنال writeBackEn فعال باشد مقدار ورودی (result_WB) را در ایندکس dest_wb رجیستر فایل می‌ریزیم؛ مقدار خروجی‌های reg1 و reg2 را نیز برابر با ایندکس src1 و src2 از رجیسترفایل قرار می‌دهیم.

```
module Register_File(reg1, reg2, result_WB, src1, src2, dest_wb, writeBackEn, rst, clk);
    input [`WORD_WIDTH-1:0] result_WB;
    input [`REG_FILE_ADDRESS_LEN-1:0] src1, src2, dest_wb;
    input clk, rst, writeBackEn;
    output [`WORD_WIDTH-1:0] reg1, reg2;
    reg [`WORD_WIDTH-1:0] registerFile [0:`REG_FILE_SIZE-1];
    integer i;

    initial begin
        for(i = 0; i < `REG_FILE_SIZE; i = i + 1)
            registerFile[i] <= i;
    end

    always@(posedge clk, posedge rst)begin
        if(rst)
            for(i = 0; i < `REG_FILE_SIZE; i = i + 1)
                registerFile[i] <= i;
        else if(writeBackEn)
                registerFile[dest_wb] <= result_WB;
    end

    assign reg1 = registerFile[src1];
    assign reg2 = registerFile[src2];
endmodule
```

Control Unit:

```
module Control_Unit(S, mode, op_code, executeCommand, mem_read, mem_write, WB_Enable, B, status_update);
    input S;
    input [1:0] mode;
    input [3:0] op_code;
    output reg [3:0] executeCommand;
    output reg  mem_read, mem_write, WB_Enable, B;
    output status_update;

    always @(*) begin
        {mem_read, mem_write, WB_Enable, B} = 4'd0;
        case (mode)
            `MODE_BRANCH: begin
                B = 1'b1;
```

```
            `MODE_MEM: begin
                case (S)
                    0: begin
                        executeCommand = `EX_STR;
                        mem_write = 1'b1;
                    end
                    1: begin
                        executeCommand = `EX_LDR;
                        mem_read = 1'b1;
                        WB_Enable = 1'b1;
                    end
                endcase
            end

            `MODE_ARITHMETIC: begin
                case (op_code)
                    `OP_MOV: begin
                        executeCommand = `EX_MOV;
                        WB_Enable = 1'b1;
                    end

                    `OP_MVN: begin
                        executeCommand = `EX_MVN;
                        WB_Enable = 1'b1;
                    end

                    `OP_ADD: begin
                        executeCommand = `EX_ADD;
                        WB_Enable = 1'b1;
                    end

                    `OP_ADC: begin
                        executeCommand = `EX_ADC;
                        WB_Enable = 1'b1;
                    end

                    `OP_SUB: begin
                        executeCommand = `EX_SUB;
                        WB_Enable = 1'b1;
                    end

                    `OP_SBC: begin
                        executeCommand = `EX_SBC;
                        WB_Enable = 1'b1;
                    end
                    `OP_AND: begin
                        executeCommand = `EX_AND;
                        WB_Enable = 1'b1;
                    end

                    `OP_ORR: begin
                        executeCommand = `EX_ORR;
                        WB_Enable = 1'b1;
                    end

                    `OP_EOR: begin
                        executeCommand = `EX_EOR;
                        WB_Enable = 1'b1;
                    end

                    `OP_CMP: begin
                        executeCommand = `EX_CMP;
                        WB_Enable = 1'b0;
                    end

                    `OP_TST: begin
                        executeCommand = `EX_TST;
                        WB_Enable = 1'b0;
                    end
                endcase
            end
        endcase
    end

    assign status_update = S;
endmodule
```

دستورات را طبق دو بیت 27 و 26 شان (Mode) به سه دسته‌ی memory،branch و arithmetic تقسیم می‌شوند؛ تنها یک دستور در دسته‌ی branch قرار دارد و سیگنال B را برای آن برابر 1 قرار می‌دهیم؛ طبق بیت 20 ام دستورات memory (S) می‌توان تصمیم گرفت که مربوط به دستور load یا store هستند، در دستور load بیت S برابر با 1 است چون state register را تغییر می‌دهد ولی در store اینگونه نیست؛ برای دستور store، mem_write را برابر 1 می‌کنیم تا بتواند در حافه بنویسد و دستور mem_read،load را برابر 1 می‌کند تا از

حافظه بخواند، همچنین WB_enable را برایی نوشتن در رجیستر فعال می‌کند؛ دستورات arithmetic را بر حسب بیت‌های 24:21 شان (OP-Code) تقسیم‌بندی می‌کنیم؛ WB_En برای تمامی این دستورات به جز Compare و Test برابر 1 است چون تمامی دستورات جز این دو، مقداری را در رجیستر می‌نویسند. در تمامی دستورات مقدار execute_command را طبق جدول 5 موجود در گزارش کار مقداردهی می‌کنیم.

:Condition_Check

```
module Condition_Check(cond, status_register, condition_state);
    input  [3:0] cond, status_register;
    output reg   condition_state;
    assign {Z, C, N, V} = status_register;

    parameter [3:0] EQ = 4'b0000,
                    NE = 4'b0001,
                    CS_HS = 4'b0010,
                    CC_LO = 4'b0011,
                    MI = 4'b0100,
                    PL = 4'b0101,
                    VS = 4'b0110,
                    VC = 4'b0111,
                    HI = 4'b1000,
                    LS = 4'b1001,
                    GE = 4'b1010,
                    LT = 4'b1011,
                    GT = 4'b1100,
                    LE = 4'b1101,
                    AL = 4'b1110;

        always @(cond, Z, C, N, V)begin
            case(cond)
                EQ:    condition_state = Z;
                NE:    condition_state = ~Z;
                CS_HS: condition_state = C;
                CC_LO: condition_state = ~C;
                MI:    condition_state = N;
                PL:    condition_state = ~N;
                VS:    condition_state = V;
                VC:    condition_state = ~V;
                HI:    condition_state = C & ~Z;
                LS:    condition_state = ~C & Z;
                GE:    condition_state = (N & V) | (~N & ~V);
                LT:    condition_state = (N & ~V) | (~N & V);
                GT:    condition_state = ~Z & ((N & V) | (~N & ~V));
                LE:    condition_state = Z & ((N & ~V) | (~N & V));
                AL:    condition_state = 1'b1;
            endcase
        end
endmodule
```

طبق 4 بیت cond، condition_state را برای حالات مختلف EQ، NE و... با توجه به بیت‌های Z و C و N و V تعیین می‌کنیم که در جدول 3 نوشته شده است.

ID Stage:

```
module ID_Stage (clk, rst, freeze, reg_file_enable,  pc_in, instruction_in, reg_file_result_WB, reg_file_dest_wb, status_register,
                                pcOut, val_Rn, val_Rm, signed_immediate, shifter_operand, reg_file_src1,
                                reg_file_src2, reg_file_dst, execute_cmd_out, mem_read_out, mem_write_out, WB_en_out, Imm_out, B_out, status_update_out);


    input clk, rst, freeze, reg_file_enable;
    input [`WORD_WIDTH-1:0] pc_in, instruction_in, reg_file_result_WB;
    input [3:0] reg_file_dest_wb, status_register;

    output [`WORD_WIDTH-1:0] pcOut, val_Rn, val_Rm;
    output [`SIGNED_IMM_WIDTH-1:0] signed_immediate;
    output [`SHIFTER_OPERAND_WIDTH-1:0] shifter_operand;
    output [`REG_FILE_ADDRESS_LEN-1:0] reg_file_src1, reg_file_src2, reg_file_dst, execute_cmd_out;
    output mem_read_out, mem_write_out, WB_en_out, Imm_out, B_out, status_update_out;

    wire [8:0] ctrl_unit_mux_in, ctrl_unit_mux_out;
    wire[3:0] executeCommand;
    wire mem_read, mem_write, WB_Enable, B, status_update, condition_state;
    wire ctrl_unit_mux_enable;
    Mux2To1 #(4) muxRegFile(.out(reg_file_src2),.in1(instruction_in[3:0]), .in2(instruction_in[15:12]), .sel(mem_write));
    Register_File regFile(.reg1(val_Rn), .reg2(val_Rm), .result_WB(reg_file_result_WB), .src1(reg_file_src1), .src2(reg_file_src2),
                          .dest_wb(reg_file_dest_wb), .writeBackEn(reg_file_enable), .rst(rst), .clk(clk));
    Mux2To1 #(9) muxCtrlUnit(.out(ctrl_unit_mux_out),.in1(ctrl_unit_mux_in), .in2(9'd0), .sel(ctrl_unit_mux_enable));
    Control_Unit ctrlUnit(.S(instruction_in[20]), .mode(instruction_in[27:26]), .op_code(instruction_in[24:21]), .executeCommand(executeCommand),
                          .mem_read(mem_read), .mem_write(mem_write), .WB_Enable(WB_Enable), .B(B), .status_update(status_update));
    Condition_Check condCheck(.cond(instruction_in[31:28]), .status_register(status_register), .condition_state(condition_state));

    assign pcOut = pc_in;
    assign ctrl_unit_mux_in = {status_update, B, executeCommand, mem_write, mem_read, WB_Enable};
    assign {status_update_out, B_out, execute_cmd_out, mem_write_out, mem_read_out, WB_en_out} = ctrl_unit_mux_out;

    assign ctrl_unit_mux_enable = (~condition_state) | freeze;

    assign shifter_operand = instruction_in[11:0];
    assign reg_file_dst = instruction_in[15:12];
    assign reg_file_src1 = instruction_in[19:16];
    assign signed_immediate = instruction_in[23:0];
    assign Imm_out = instruction_in[25];

endmodule
```

در این ماژول از ماژول‌های Register_File، Control_Unit و Consition_Check نمونه‌گیری می‌کنیم؛ بر سر ورودی src2 رجیستر فایل یک مولتی پلکسر 4 بیتی وجود دارد که select آن mem_write است و بین بیت‌های 3:0 و 15:12 دستور انتخاب می‌کند؛ همچنین یک مولتی پلکسر 9 بیتی برای سیگنال‌های کنترلی تعریف می‌کنیم که سلکتور آن درصورتی 1 می‌شود که freeze 1 باشد و یا condition_state برابر 0 باشد، در این صورت سیگنال‌های کنترلی که در Control_Unit مقداردهی شده‌اند به مرحله‌ی بعد منتقل می‌شوند و در غیر اینصورت مقادیر تمامی سیگنال‌های کنترلی برابر با 0 می‌شود. در این ماژول مقادیر shifter_operand, reg_file_des, signed_immediate و Imm را نیز طبق جدول با ورودی instruction مقداردهی می‌کنیم.

:ID_Stage_Register

```
module ID_Stage_Reg #(parameter N = 32)(clk, rst, flush, mem_read_in, mem_write_in, WB_Enable_in, Imm_in, B_in, status_update_in,
                                        reg_file_dst_in, executeCommand_in, status_register_in, reg_file_src1_in, reg_file_src2_in,
                                        shifter_operand_in, signed_immediate_in, pc_in, val_Rn_in, val_Rm_in, mem_read_out,
                                        mem_write_out, WB_Enable_out, Imm_out, B_out, status_update_out, reg_file_dst_out, execute_cmd_out,
                                        status_register_out, reg_file_src1_out, reg_file_src2_out, shifter_operand_out, signed_immediate_out,
                                        pc_out, val_Rn_out, val_Rm_out);

    input clk, rst, flush, mem_read_in, mem_write_in, WB_Enable_in, Imm_in, B_in, status_update_in;
    input [3:0] reg_file_dst_in, executeCommand_in, status_register_in, reg_file_src1_in, reg_file_src2_in;
    input [`SHIFTER_OPERAND_WIDTH-1:0] shifter_operand_in;
    input [`SIGNED_IMM_WIDTH-1:0] signed_immediate_in;
    input [`WORD_WIDTH-1:0] pc_in, val_Rn_in, val_Rm_in;

    output reg mem_read_out, mem_write_out, WB_Enable_out, Imm_out, B_out, status_update_out;
    output reg [3:0] reg_file_dst_out, execute_cmd_out, status_register_out, reg_file_src1_out, reg_file_src2_out;
    output reg [`SHIFTER_OPERAND_WIDTH-1:0] shifter_operand_out;
    output reg [`SIGNED_IMM_WIDTH-1:0] signed_immediate_out;
    output reg [`WORD_WIDTH-1:0] pc_out, val_Rn_out, val_Rm_out;



   always @(posedge clk, posedge rst) begin
    if (rst)
        begin
            pc_out <= 0;
            reg_file_dst_out <= 0;
            val_Rn_out <= 0;
            val_Rm_out <=0;
            signed_immediate_out <= 0;
            shifter_operand_out <= 0;
            execute_cmd_out <= 0;
            status_register_out <= 0;
            mem_read_out <= 0;
            mem_write_out <= 0;
            WB_Enable_out <= 0;
            Imm_out <= 0;
            B_out <= 0;
            status_update_out <= 0;
            reg_file_src1_out <= 0;
            reg_file_src2_out <= 0;
        end
    if (flush)
        begin
            pc_out <= 0;
            reg_file_dst_out <= 0;
            val_Rn_out <= 0;
            val_Rm_out <=0;
            signed_immediate_out <= 0;
            shifter_operand_out <= 0;
            execute_cmd_out <= 0;
            status_register_out <= 0;
            mem_read_out <= 0;
            mem_write_out <= 0;
            WB_Enable_out <= 0;
            Imm_out <= 0;
            B_out <= 0;
            status_update_out <= 0;
            reg_file_src1_out <= 0;
            reg_file_src2_out <= 0;
        end
    else
        begin
            pc_out <= pc_in;
            reg_file_dst_out <= reg_file_dst_in;
            val_Rn_out <= val_Rn_in;
            val_Rm_out <= val_Rm_in;
            signed_immediate_out <= signed_immediate_in;
            shifter_operand_out <= shifter_operand_in;
            execute_cmd_out <= executeCommand_in;
            status_register_out <= status_register_in;
            mem_read_out <= mem_read_in;
            mem_write_out <= mem_write_in;
            WB_Enable_out <= WB_Enable_in;
            Imm_out <= Imm_in;
            B_out <= B_in;
            status_update_out <= status_update_in;
            reg_file_src1_out <= reg_file_src1_in;
            reg_file_src2_out <= reg_file_src2_in;
        end
  end
endmodule
```

در این رجیستر مقادیر بدست آمده در مرحله‌ی قبل با لبه‌ی بالارونده‌ی clk آپدیت می‌شوند و درصورتی که flush برابر با 1 باشد مقادیر تمامی آنها برابر با 0 می‌شود.

Arm Top module:

```
module ARM(clk, rst, out);
    input clk, rst;
    output [31:0] out;
    wire freeze, branch_taken;
    //IF Stage
    wire  [`WORD_WIDTH-1:0] branchAddr, IFStagePcOut, IFRegPcOut, IFStageInstructionOut, IFRegInstructionOut;

    //ID Stage
    wire  [`WORD_WIDTH-1:0] ID_pc_out, ID_val_Rn, ID_val_Rm;
    wire [`SIGNED_IMM_WIDTH-1:0] ID_signed_immediate;
    wire [`SHIFTER_OPERAND_WIDTH-1:0] ID_shifter_operand;
    wire [`REG_FILE_ADDRESS_LEN-1:0] ID_regFile_src1, ID_regFile_src2, ID_regFile_dst;
    wire [3:0] ID_execute_cmd_out, status_reg_out;
    wire ID_mem_read_out, ID_mem_write_out, ID_WB_en_out, ID_imm_out, ID_B_out, ID_status_update_out;

    //ID Reg
    wire IDreg_mem_read_out, IDreg_mem_write_out, IDreg_WB_en_out, IDreg_imm_out, IDreg_B_out, IDreg_status_update_out;
    wire [`REG_FILE_ADDRESS_LEN-1:0] IDreg_regFile_dst;
    wire [3:0] IDreg_execute_cmd_out, IDreg_status_out;
    wire [`REG_FILE_ADDRESS_LEN-1:0] IDreg_regFile_src1, IDreg_regFile_src2;
    wire [`SHIFTER_OPERAND_WIDTH-1:0] IDreg_shifter_operand;
    wire [`SIGNED_IMM_WIDTH-1:0] IDreg_signed_immediate;
    wire  [`WORD_WIDTH-1:0] IDreg_pc_out, IDreg_val_Rn, IDreg_val_Rm;
    //////////
    wire [31:0]  execOut, exMemRegOut, memOut, memWbRegOut, IFIDregOut;

    //EXE Stage
    wire EXE_B_out;

    //WB Stage
    wire WB_WB_en_out;
    wire [`WORD_WIDTH-1:0] WB_Value;
    wire [`REG_FILE_ADDRESS_LEN-1:0] WB_dst_out;


    IF_Stage IF_stage(.clk(clk), .rst(rst), .freeze(freeze), .branch_taken(branch_taken), .branchAddr(branchAddr), .pc(IFStagePcOut),
                    .instruction(IFStageInstructionOut));
    IF_Stage_Reg IF_stage_reg(.clk(clk), .rst(rst), .freeze(freeze), .flush(branch_taken), .pcIn(IFStagePcOut), .instructionIn(IFStageInstructionOut),
                            .pcOut(IFRegPcOut), .instruction(IFRegInstructionOut));

    ID_Stage ID_stage(.clk(clk), .rst(rst), .freeze(1'b0), .reg_file_enable(/*WB_WB_en_out*/1'b0),  .pc_in(IFRegPcOut), .instruction_in(IFStageInstructionOut),
                    .reg_file_result_WB(/*WB_value*/32'd0), .reg_file_dest_wb(/*WB_dst_out*/ 4'd0), .status_register(/*status_reg_out*/4'd0), .pcOut(ID_pc_out),
                    .val_Rn(ID_val_Rn), .val_Rm(ID_val_Rm), .signed_immediate(ID_signed_immediate), .shifter_operand(ID_shifter_operand),
                    .reg_file_src1(ID_regFile_src1), .reg_file_src2(ID_regFile_src2), .reg_file_dst(ID_regFile_dst),
                    .execute_cmd_out(ID_execute_cmd_out), .mem_read_out(ID_mem_read_out), .mem_write_out(ID_mem_write_out),
                    .WB_en_out(ID_WB_en_out), .Imm_out(ID_imm_out), .B_out(ID_B_out), .status_update_out(ID_status_update_out));

    ID_Stage_Reg ID_stage_reg(.clk(clk), .rst(rst), .flush(EXE_B_out), .mem_read_in(ID_mem_read_out), .mem_write_in(ID_mem_write_out),
                            .WB_Enable_in(ID_WB_en_out), .Imm_in(ID_imm_out), .B_in(ID_B_out), .status_update_in(ID_status_update_out),
                            .reg_file_dst_in(ID_regFile_dst), .executeCommand_in(ID_execute_cmd_out), .status_register_in(/*status_reg_out*/4'd0),
                            .reg_file_src1_in(ID_regFile_src1), .reg_file_src2_in(ID_regFile_src2), .shifter_operand_in(ID_shifter_operand),
                            .signed_immediate_in(ID_signed_immediate), .pc_in(ID_pc_out), .val_Rn_in(ID_val_Rn), .val_Rm_in(ID_val_Rm),
                            .mem_read_out(IDreg_mem_read_out), .mem_write_out(IDreg_mem_write_out), .WB_Enable_out(IDreg_WB_en_out),
                            .Imm_out(IDreg_imm_out), .B_out(IDreg_B_out), .status_update_out(IDreg_status_update_out),
                            .reg_file_dst_out(IDreg_regFile_dst), .execute_cmd_out(IDreg_execute_cmd_out), .status_register_out(IDreg_status_out),
                            .reg_file_src1_out(IDreg_regFile_src1), .reg_file_src2_out(IDreg_regFile_src2), .shifter_operand_out(IDreg_shifter_operand),
                            .signed_immediate_out(IDreg_signed_immediate), .pc_out(IDreg_pc_out),
                            .val_Rn_out(IDreg_val_Rn), .val_Rm_out(IDreg_val_Rm));
    Exec_Stage Exec_stage(.clk(clk), .rst(rst), .in(IDreg_pc_out), .out(execOut));
    Exec_Stage_Reg Exec_stage_reg(.clk(clk), .rst(rst), .in(execOut), .out(exMemRegOut));
    Mem_Stage Mem_stage(.clk(clk), .rst(rst), .in(exMemRegOut), .out(memOut));
    Mem_Stage_Reg Mem_stage_reg(.clk(clk), .rst(rst), .in(memOut), .out(memWbRegOut));
    WB_Stage WB_stage(.clk(clk), .rst(rst), .in(memWbRegOut), .out(out));
endmodule
```

به ازای wire هایی که هنوز پیاده‌سازی نشده‌اند 0 قرار می‌دهیم و خروجی ID_Stage را به ورودی ID_Stage_reg وصل می‌کنیم.

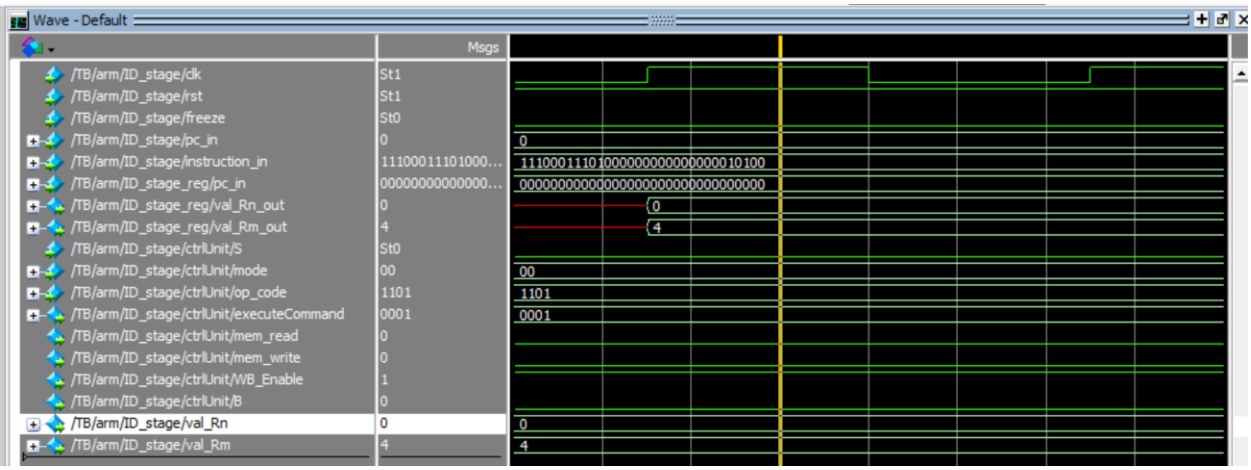
همانطور که تصویر زیر دیده می‌شود 18 دستور اول را در instruction memory قرار میدهیم.

```
`include "defines.v"
module Instruction_Memory(rst, addr, read_instruction);
    input [`INSTRUCTION_LEN - 1 : 0] addr;
    input rst;
    output [`INSTRUCTION_LEN - 1 : 0] read_instruction;

    reg[7 : 0] instruction[0:`INSTRUCTION_MEM_SIZE - 1];
    assign read_instruction = {instruction[addr], instruction[addr + 1], instruction[addr + 2], instruction[addr + 3]};

    always @(posedge rst) begin
        if (rst)
        begin
            {instruction[0], instruction[1], instruction[2], instruction[3]} <= `INSTRUCTION_LEN'b1110_00_1_1101_0_0000_0000_000000010100;      //  MOV R0 = 20          R0 = 20
            {instruction[4], instruction[5], instruction[6], instruction[7]} <= `INSTRUCTION_LEN'b1110_00_1_1101_0_0000_0001_101000000001;      //  MOV R1 ,#4096        R1 = 4096
            {instruction[8], instruction[9], instruction[10], instruction[11]} <= `INSTRUCTION_LEN'b1110_00_1_1101_0_0000_0010_000100000011;   //  MOV R2 ,#0xC0000000    R2 = -1073741824
            {instruction[12], instruction[13], instruction[14], instruction[15]} <= `INSTRUCTION_LEN'b1110_00_0_0100_1_0010_0011_000000000010; //  ADDS R3 ,R2,R2         R3 = -2147483648
            {instruction[16], instruction[17], instruction[18], instruction[19]} <=  `INSTRUCTION_LEN'b1110_00_0_0101_0_0000_0100_000000000000; //ADC R4 ,R0,R0 //R4 = 41
            {instruction[20], instruction[21], instruction[22], instruction[23]} <= `INSTRUCTION_LEN'b1110_00_0_0010_0_0100_0101_000100000100; //SUB R5 ,R4,R4,LSL #2 //R5 = -123
            {instruction[24], instruction[25], instruction[26], instruction[27]} <= `INSTRUCTION_LEN'b1110_00_0_0110_0_0000_0110_000010100000; //SBC R6 ,R0,R0,LSR #1//R6 = 10
            {instruction[28], instruction[29], instruction[30], instruction[31]} <= `INSTRUCTION_LEN'b1110_00_0_1100_0_0101_0111_000101000010; //ORR    R7 ,R5,R2,ASR #2//R7 = -123
            {instruction[32], instruction[33], instruction[34], instruction[35]} <= `INSTRUCTION_LEN'b1110_00_0_0000_0_0111_1000_000000000011; //AND R8 ,R7,R3   R8 = -2147483648
            {instruction[36], instruction[37], instruction[38], instruction[39]} <= `INSTRUCTION_LEN'b1110_00_0_1111_0_0000_1001_000000000110; //MVNR9 ,R6//R9 = -11
            {instruction[40], instruction[41], instruction[42], instruction[43]} <= `INSTRUCTION_LEN'b1110_00_0_0001_0_0100_1010_000000000101; //EOR R10,R4,R5 //R10 = -84
            {instruction[44], instruction[45], instruction[46], instruction[47]} <= `INSTRUCTION_LEN'b1110_00_0_1010_1_1000_0000_000000000110; //CMPR8 ,R6
            {instruction[48], instruction[49], instruction[50], instruction[51]} <= `INSTRUCTION_LEN'b0001_00_0_0100_0_0001_0001_000000000001; //ADDNER1 ,R1,R1//R1 = 8192
            {instruction[52], instruction[53], instruction[54], instruction[55]} <= `INSTRUCTION_LEN'b1110_00_0_1000_1_1001_0000_000000001000; //TSTR9 ,R8
            {instruction[56], instruction[57], instruction[58], instruction[59]} <= `INSTRUCTION_LEN'b0000_00_0_0100_0_0010_0010_000000000010; //ADDEQ R2 ,R2,R2   //R2 = -1073741824
            {instruction[60], instruction[61], instruction[62], instruction[63]} <= `INSTRUCTION_LEN'b1110_00_1_1101_0_0000_0000_101100000001; //MOVR0 ,#1024//R0 = 1024
            {instruction[64], instruction[65], instruction[66], instruction[67]} <= `INSTRUCTION_LEN'b1110_01_0_0100_0_0000_0001_000000000000; //STRR1 ,[R0],#0//MEM[1024] = 8192
            {instruction[68], instruction[69], instruction[70], instruction[71]} <= `INSTRUCTION_LEN'b1110_01_0_0100_1_0000_1011_000000000000; //LDRR11,[R0],#0//R11 = 8192
        end
    end

endmodule
```

درستی سیگنال‌های خروجی Control_Unit و Rn و Rm:

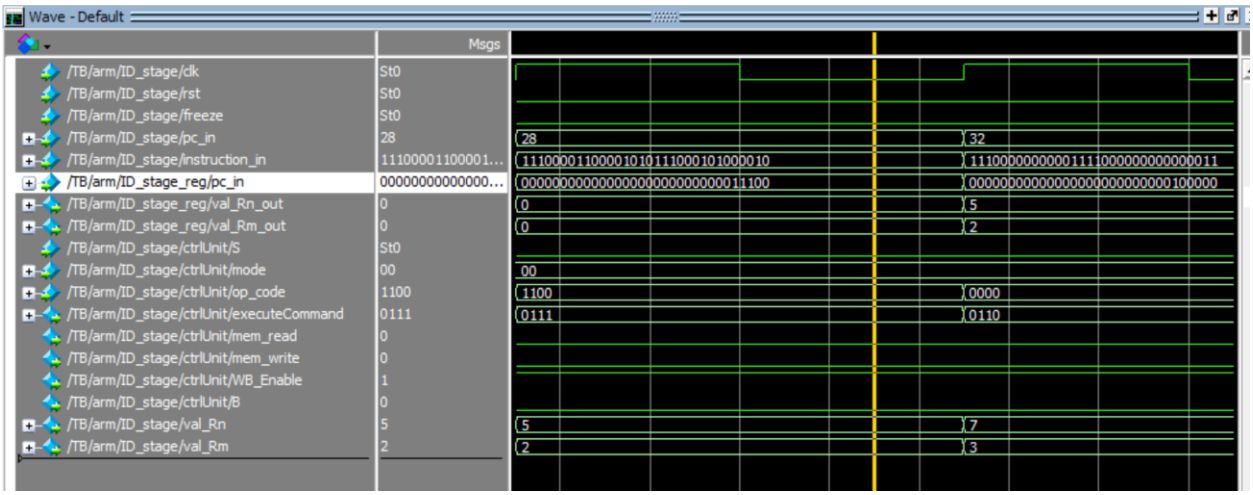
چهار مورد از 18 دستور اول را انتخاب کرده و به توضیح درستی آنها می‌پردازیم.

دستور //MOV R0 ,#20 //R0 = 20 :



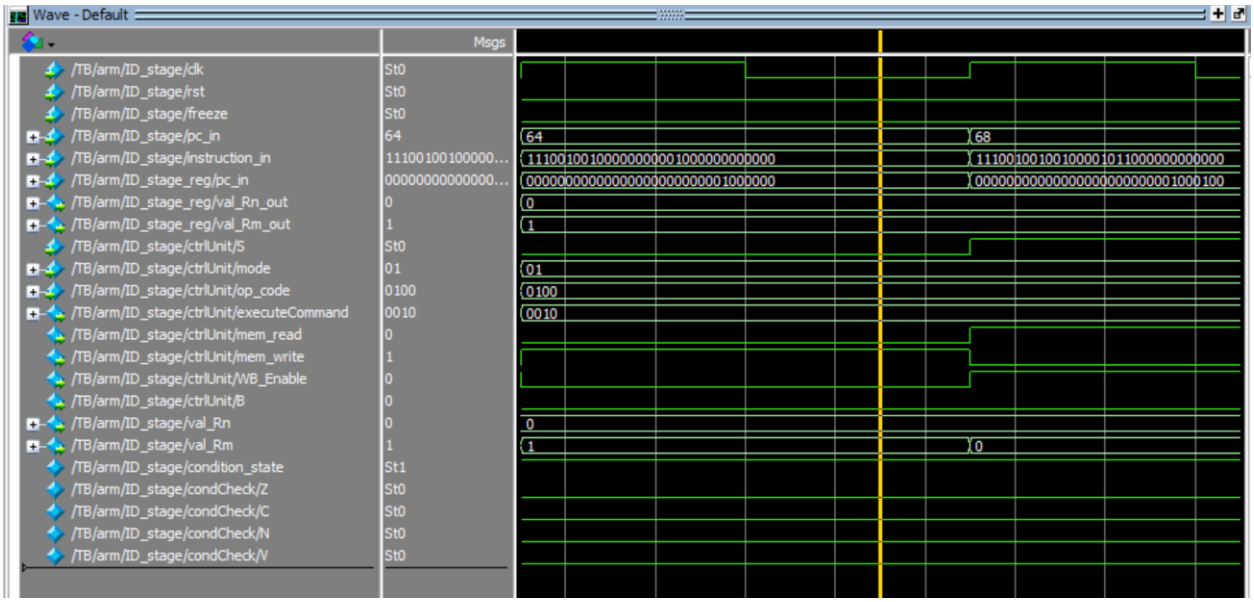
همانطور که می‌بینیم تنها مقدار سیگنال کنترلی WB_Enable برابر با 1 است که همانگونه است که انتظارش را داشتیم و مقادیر Rn و Rm باید برابر با اندیسشان باشد در اینجا بیت‌های 3:0 دستور برابر با 0100 است پس مقدار Rm برابر با 4 می‌شود و بیت‌های 19:16 برابر با 0000 است پس مقدار Rn برابر با 0 است. همچنین executeCommand مربوط به آن برابر با 0001 است.

دستور //ORR R7 ,R5,R2,ASR #2 //R7 = -123 :



همانطور که می‌بینیم تنها مقدار سیگنال کنترلی WB_Enable برابر با 1 است که همانگونه است که انتظارش را داشتیم و مقادیر Rn و Rm باید برابر با اندیسشان باشد در اینجا بیت‌های 3:0 دستور برابر با 0010 است پس مقدار Rm برابر با 2 می‌شود و بیت‌های 19:16 برابر با 0101 است پس مقدار Rn برابر با 5 است. توجه شود که مقادیر Rm و Rn رجیستر پس از ID Stage به اندازه یک کلاک عقب است و مقادیر ذکر شده مربوط به Rn و Rm در مرحله‌ی ID است. همچنین executeCommand مربوط به آن برابر با 0111 است.

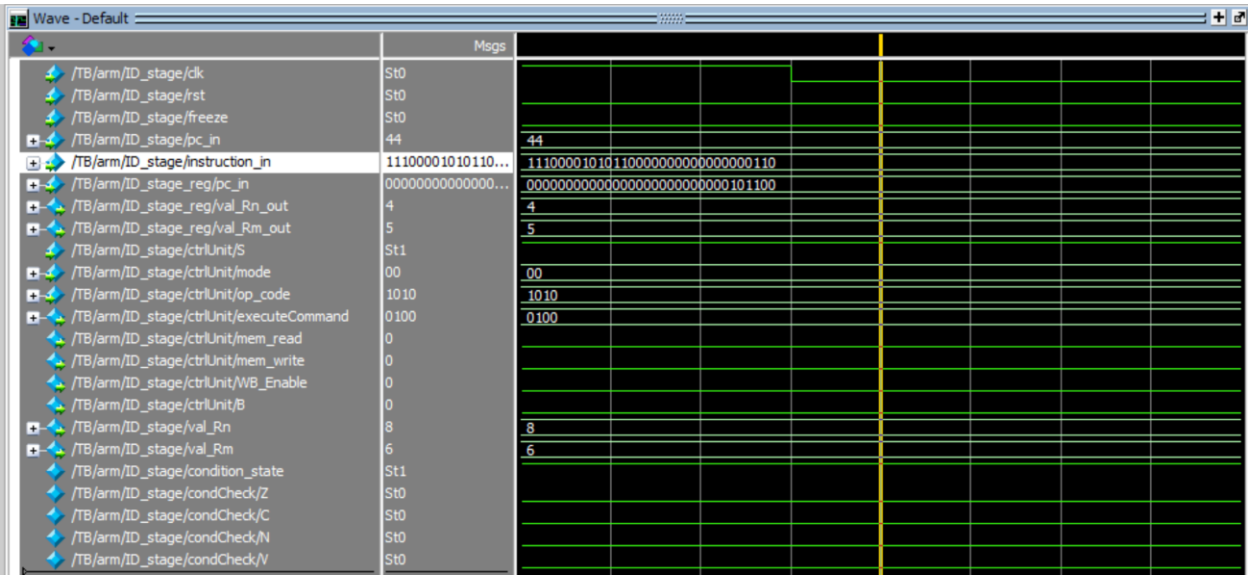
دستور //STR R1 ,[R0],#0 //MEM[1024] = 8192:



همانطور که می‌بینیم تنها مقدار سیگنال کنترلی mem_write برابر با 1 است زیرا دستور store باید در حافظه بنویسد و مقادیر Rn و Rm باید برابر با اندیسشان باشد در اینجا بیت‌های 3:0 دستور برابر با 0001 است پس

مقدار Rm برابر با 1 می‌شود و بیت‌های 19:16 برابر با 0000 است پس مقدار Rn برابر با 0 است. همچنین executeCommand مربوط به آن برابر با 0010 است. همچنین بیت S در اینجا برابر با 0 است چون نیازی به آپدیت کردن status register نیست. Condition_state برابر با 1 است چون cond برابر با 1110 است. executeCommand مربوط به آن برابر با 0010 است.

دستور //CMP R8 ,R6:



همانطور که دیده می‌شود در این دستور WB_Enable برابر با 0 است چون نیازی به ذخیره‌ی نتیجه در رجیستر نیست؛ بیت‌های 3:0 دستور برابر با 0110 است پس مقدار Rm برابر با 6 می‌شود و بیت‌های 19:16 برابر با 1000 است پس مقدار Rn برابر با 8 است. Condition_state برابر با 1 است چون cond برابر با 1110 است. executeCommand مربوط به آن برابر با 0100 است. S نیز برابر با 1 است چون status register باید آپدیت شود.