آوا میرمحمدمهدی 810199501 نسا عباسی مقدم 810199457

:ALU

```
module ALU ( val1, val2, cin, EX_command, ALU_out, SR);
    input [`WORD_WIDTH-1:0] val1, val2;
    input cin;
    input [3:0] EX_command;
    output reg [`WORD_WIDTH-1:0] ALU_out;
    output [3:0] SR;

    reg V, C;
    wire N, Z;

    always @(*) begin
        {V, C} = 2'd0;
        case (EX_command)
            `EX_MOV: begin
                ALU_out = val2;
            end
            `EX_MVN: begin
                ALU_out = ~val2;
            end
            `EX_ADD: begin
                {C, ALU_out} = val1 + val2;
                V = ((val1[`WORD_WIDTH - 1] == val2[`WORD_WIDTH - 1]) & (ALU_out[`WORD_WIDTH - 1] != val1[`WORD_WIDTH - 1]));
            end
            `EX_ADC: begin
                {C, ALU_out} = val1 + val2 + cin;
                V = ((val1[`WORD_WIDTH - 1] == val2[`WORD_WIDTH - 1]) & (ALU_out[`WORD_WIDTH - 1] != val1[`WORD_WIDTH - 1]));
            end
            `EX_SUB: begin
               {C, ALU_out} = val1 - val2;
                V = ((val1[`WORD_WIDTH - 1] == val2[`WORD_WIDTH - 1]) & (ALU_out[`WORD_WIDTH - 1] != val1[`WORD_WIDTH - 1]));

            end
            `EX_SBC: begin
               {C, ALU_out} = val1 - val2 -1 + cin;
                V = ((val1[`WORD_WIDTH - 1] == val2[`WORD_WIDTH - 1]) & (ALU_out[`WORD_WIDTH - 1] != val1[`WORD_WIDTH - 1]));
            end
            `EX_AND: begin
                ALU_out = val1 & val2;
            end
            `EX_ORR: begin
                ALU_out = val1 | val2;
            end
            `EX_EOR: begin
                ALU_out = val1 ^ val2;
            end
            `EX_CMP: begin
                {C, ALU_out} = {val1[`WORD_WIDTH-1], val1} - {val2[`WORD_WIDTH-1], val2};
                V = ((val1[`WORD_WIDTH - 1] == val2[`WORD_WIDTH - 1]) & (ALU_out[`WORD_WIDTH - 1] != val1[`WORD_WIDTH - 1]));
            end
            `EX_TST: begin
                ALU_out = val1 & val2;
            end
            `EX_LDR: begin
                ALU_out = val1 + val2;
            end
            `EX_STR: begin
                ALU_out = val1 + val2;
            end

        endcase
    end


    assign SR = {Z, C, N, V};
    assign N = ALU_out[`WORD_WIDTH-1];
    assign Z = |ALU_out ? 1'b0:1'b1;

endmodule
```

طبق 4 بیت execute_command مشخص می‌شود که واحد arithmetic باید یه عملیاتی انجام دهد؛ برای عملیات جمع یا تفریق، سمت چپ‌ترین بیت نتیجه را به عنوان بیت carry در نظر می‌گیریم؛ همچنین بیت overflow(V) زمانی برابر با 1 است که اولا دو عددی که روی آنها عملیات انجام می‌دهیم هم علامت باشند (یعنی سمت چپ‌ترین بیتشان باهم برابر باشد) و ثانیا عدد حاصل شده با این اعداد هم علامت نباشد. درصورتی که or تمامی بیت‌های نتیجه برابر با 0 بود یعنی عدد برابر با 0 است پس Z = 0 می‌شود و در غیر اینصورت برابر با 1 است.

همچنین اگر سمت چپ‌ترین بیت نتیجه برابر با 1 باشد یعنی عدد منفی است و N = 1 می‌شود. در نهایت چهار بیت نشان‌دهنده‌ی وضعیت را در SR ذخیره می‌کنیم.

:Val2Generator

```
module Val2Generator(val_Rm, shift_operand, immediate, is_mem_cmd, val2_out);
    input [`WORD_WIDTH-1:0] val_Rm;
    input [`SHIFTER_OPERAND_WIDTH-1:0] shift_operand;
    input immediate, is_mem_cmd;
    output reg [`WORD_WIDTH-1:0] val2_out;

    reg [2 * (`WORD_WIDTH) - 1 : 0] tmp;
    integer i;
    always @(val_Rm, shift_operand, immediate, is_mem_cmd) begin
        val2_out = `WORD_WIDTH'b0;
        tmp = 0;
        if (is_mem_cmd == 1'b0) begin

            if (immediate == 1'b1) begin
                val2_out = {24'b0 ,shift_operand[7 : 0]};
                tmp = {val2_out, val2_out} >> (({{2'b0},shift_operand[11 : 8]}) << 1);
                val2_out = tmp[`WORD_WIDTH - 1 : 0];

            end
            else if(immediate == 1'b0 && shift_operand[4] == 0) begin
                case(shift_operand[6:5])
                    `LSL_SHIFT : begin
                        val2_out = val_Rm << shift_operand[11 : 7];
                    end
                    `LSR_SHIFT : begin
                        val2_out = val_Rm >> shift_operand[11 : 7];
                    end
                    `ASR_SHIFT : begin
                        val2_out = val_Rm >>> shift_operand[11 : 7];
                    end
                    `ROR_SHIFT : begin
                        tmp = {val_Rm, val_Rm} >> (shift_operand[11 : 7]);
                        val2_out = tmp[`WORD_WIDTH - 1 : 0];
                    end
                endcase
            end
        end
        else //is mem_command
        begin
            val2_out = { {20{shift_operand[11]}} , shift_operand[11 : 0]};
        end

    end
endmodule
```

در این قسمت مقدار ورودی دوم ALU محاسبه می‌شود. در ابتدا دستورات را بر حسب اینکه مربوط به حافظه هستند یا نه به دو دسته تقسیم می‌کنیم. mem_read_en یا mem_write_en در دستوراتی که مربوط به حافظه‌اند برابر با 1 است. در این حالت عدد 12 بیتی shift_operand را sign_extend می‌کنیم تا 32 بیت شود و آن را به عنوان مقدار ورودی دوم ALU قرار می‌دهیم. درصورتی که دستور مربوط به حافظه باشد و بیت I در instruction برابر با 1 باشد، 8 بیت سمت راست instruction را برمی‌داریم و در یک ظرف 32 بیتی می‌ریزیم یعنی به سمت چپ آن 24 بیت صفر اضافه می‌کنیم سپس آن را به اندازه‌ی دو برابر rotate_imm (instruction[11:8] به سمت راست می‌چرخانیم؛ برای اینکار از یک متغیر reg کمکی استفاده می‌کنیم و {val2_out, val2_out} را در آن می‌ریزیم و سپس آن را به مقدار گفته شده شیفت می‌دهیم تا هنگام شیفت به راست بیتی از بین نرود. در نهایت اگر بیت I برابر با 0 و بیت چهارم دستورالعمل نیز برابر با 0 باشد، عملوند دوم را بر اساس دو بیتی که تعیین کننده‌ی نوع شیفت است، شیفت می‌دهیم.

:Status_Register

```
module Status_Register(clk, rst, ld, in, out);
    input clk;
    input rst;
    input ld;
    input [3:0] in;
    output reg [3:0] out;
    always@(posedge clk, posedge rst)
    begin
        if (rst)
            out <= 0;
        else
        if (ld)
            out <= in;
    end
endmodule
```

Status_register مانند یک رجیستر معمولی است که چهار بیت وضعیت Z, C, N, V را ذخیره می‌کند و load آن را نیز به بیت S متصل می‌کنیم تا در صورت نیاز مقادیر بیت‌های وضعیت آپدیت شود.

:EXE_Stage

```
module Exec_Stage(clk, rst, pc_in, mem_read_in, mem_write_in, S, B, WB_en_in, execute_cmd_in, immediate_in, signed_immediate_24_in, shift_operand_in, val_Rn_in,
                val_Rm_in, SR_in, dst_in, alu_out, branch_address, status_reg_out, branch_taken_out, WB_en_out, mem_read_out, mem_write_out, dst_out, val_Rm_out);

    input clk, rst;
    input [`WORD_WIDTH-1: 0] pc_in;
    input mem_read_in, mem_write_in, S, B, WB_en_in;
    input [3:0] execute_cmd_in;
    input immediate_in;
    input [`SIGNED_IMM_WIDTH-1:0] signed_immediate_24_in;
    input [`SHIFTER_OPERAND_WIDTH - 1 : 0] shift_operand_in;
    input [`WORD_WIDTH - 1: 0] val_Rn_in, val_Rm_in;
    input [3:0] SR_in;
    input [`REG_FILE_ADDRESS_LEN-1:0] dst_in;

    output [`WORD_WIDTH - 1 : 0] alu_out;
    output [`WORD_WIDTH - 1 : 0] branch_address;
    output [3:0] status_reg_out;
    output branch_taken_out, WB_en_out, mem_read_out, mem_write_out;
    output [`REG_FILE_ADDRESS_LEN-1:0] dst_out;
    output [`WORD_WIDTH - 1 : 0] val_Rm_out;


    wire [3:0] status_bits;
    wire [`WORD_WIDTH - 1 : 0] val2out;

    ALU alu(.val1(val_Rn_in), .val2(val2out), .cin(SR_in[2]), .EX_command(execute_cmd_in), .ALU_out(alu_out), .SR(status_bits));

    wire is_mem_cmd;
    assign is_mem_cmd = mem_read_in | mem_write_in;

    assign dst_out = dst_in;
    assign branch_taken_out = B;
    assign mem_read_out = mem_read_in;
    assign mem_write_out = mem_write_in;
    assign WB_en_out = WB_en_in;

    Status_Register status_ref(.clk(clk), .rst(rst), .ld(S), .in(status_bits), .out(status_reg_out));

    assign val_Rm_out = val_Rm_in;
    Val2Generator v2g(.val_Rm(val_Rm_in), .shift_operand(shift_operand_in), .immediate(immediate_in), .is_mem_cmd(is_mem_cmd), .val2_out(val2out));

    wire [`WORD_WIDTH - 1 : 0] sign_immediate_extended = { {8{signed_immediate_24_in[23]}}, signed_immediate_24_in};
    wire [`WORD_WIDTH - 1 : 0] sign_immediate_extended_xfour = sign_immediate_extended << 2;
    Adder adderPc(.a(pc_in), .b(sign_immediate_extended_xfour), .res(branch_address));

endmodule
```

همانطور که در کد مشخص است برای بدست آوردن آدرس پرش ابتدا عدد 24 بیتی را sign_extend می‌کنیم و سپس آن را دو واحد به چپ شیفت می‌دهیم تا عدد مضرب 4 شود (چون pc را در هر مرحله با 4 جمع می‌کنیم). در مرحله EXE از ماژول‌های ALU، Val2Generator و Status_Register نمونه‌گیری می‌کنیم و مقدار خروجی Val2Generator را به عنوان ورودی دوم به ALU می‌دهیم.

:EXE_Stage_reg

```
module Exec_Stage_Reg (clk, rst, dst_in, mem_read_in, mem_write_in, WB_en_in, val_Rm_in, ALU_res_in, dst_out, ALU_res_out, val_Rm_out, mem_read_out, mem_write_out, WB_en_out);
    input clk, rst;
    input  [`REG_FILE_ADDRESS_LEN-1:0] dst_in;
    input  mem_read_in, mem_write_in, WB_en_in;
    input  [`WORD_WIDTH-1:0] val_Rm_in;
    input [`WORD_WIDTH-1:0] ALU_res_in;

    output reg [`REG_FILE_ADDRESS_LEN-1:0] dst_out;
    output reg [`WORD_WIDTH-1:0] ALU_res_out;
    output reg [`WORD_WIDTH-1:0] val_Rm_out;
    output reg mem_read_out, mem_write_out, WB_en_out;

    always @(posedge clk, posedge rst) begin
        if(rst) begin
        dst_out <= 0;
        ALU_res_out <= 0;
        val_Rm_out <= 0;
        mem_read_out <= 0;
        mem_write_out <= 0;
        WB_en_out <= 0;
        end
        else begin
        dst_out <= dst_in;
        ALU_res_out <= ALU_res_in;
        val_Rm_out <= val_Rm_in;
        mem_read_out <= mem_read_in;
        mem_write_out <= mem_write_in;
        WB_en_out <= WB_en_in;
        end
    end
endmodule
```

همانطور که در کد دیده می‌شود این رجیستر مقادیر dest و WB_En و حاصل ALU، مقدار Rm، mem_read و mem_write ذخیره می‌شود.

:Data_Memory

```
module Data_Memory(clk, rst, addr, write_data, mem_read, mem_write, read_data);
    input [`INSTRUCTION_LEN - 1 : 0] addr, write_data;
    input clk, rst, mem_read, mem_write;
    output [`INSTRUCTION_LEN - 1 : 0] read_data;

    reg[7 : 0] data[0:`INSTRUCTION_MEM_SIZE - 1];
    integer i;

    assign read_data = mem_read ?  {data[addr], data[addr + 1], data[addr + 2], data[addr + 3]} : `INSTRUCTION_LEN'b0;

    wire [`INSTRUCTION_LEN - 1 : 0] new_addr;
    assign new_addr = { {addr[`INSTRUCTION_LEN - 1 : 2]}, {2'b00} };

    always@(posedge clk, posedge rst)begin
        if(rst)
            for(i = 0; i < `INSTRUCTION_MEM_SIZE; i = i + 1)
                data[i] <= 8'd0;
        if (mem_write)begin
            {data[new_addr], data[new_addr + 1], data[new_addr + 2], data[new_addr + 3]} = write_data;
        end
    end

endmodule
```

از آنجایی که instruction ها 32 بیتی است ولی هر خانه از حافظه 8 بیت گنجایش دارد، زمانی که mem_write برابر با 1 است مقداری که باید در حافظه نوشته شود را در چهار خانه‌ی کنارهم از حافظه می‌نویسیم. زمانی که mem_read برابر با 1 است نیز مقدار چهار خانه‌ی کنار هم از حافظه را در read_data می‌ریزیم و در غیر اینصورت مقدار این خروجی برابر با 0 است؛ خانه‌های حافظه نیز در ابتدا با 0 مقداردهی اولیه شده‌اند.

Mem_Stage:

```
module Mem_Stage (clk, rst, dst, ALU_res, val_Rm, mem_read, mem_write, WB_en, dst_out, ALU_res_out, mem_out, mem_read_out, WB_en_out);
    input clk, rst;
    input [`REG_FILE_ADDRESS_LEN-1:0] dst;
    input [`WORD_WIDTH-1:0] ALU_res;
    input [`WORD_WIDTH-1:0] val_Rm;
    input mem_read, mem_write, WB_en;

    output [`REG_FILE_ADDRESS_LEN-1:0] dst_out;
    output [`WORD_WIDTH-1:0] ALU_res_out;
    output [`WORD_WIDTH-1:0] mem_out;
    output mem_read_out, WB_en_out;

    assign dst_out = dst;
    assign mem_read_out = mem_read;
    assign WB_en_out = WB_en;
    assign ALU_res_out = ALU_res;

    Data_Memory data_mem(.clk(clk), .rst(rst), .addr(ALU_res), .write_data(val_Rm), .mem_read(mem_read), .mem_write(mem_write), .read_data(mem_out));

endmodule
```

در این ماژول از Data_Memory نمونه‌گیری می‌کنیم و مقادیر dest، mem_read، WB_en و ALU_res با این مقادیر در ورودی ماژول برابر است.

Mem_Stage_reg:

```
module Mem_Stage_Reg (clk, rst, dst, ALU_res, mem_data, mem_read, WB_en, dst_out, ALU_res_out, mem_data_out, mem_read_out, WB_en_out);
    input clk, rst;
    input [`REG_FILE_ADDRESS_LEN-1:0] dst;
    input [`WORD_WIDTH-1:0] ALU_res;
    input [`WORD_WIDTH-1:0] mem_data;
    input mem_read, WB_en;

    output reg [`REG_FILE_ADDRESS_LEN-1:0] dst_out;
    output reg [`WORD_WIDTH-1:0] ALU_res_out;
    output reg [`WORD_WIDTH-1:0] mem_data_out;
    output reg mem_read_out, WB_en_out;

    always @(posedge clk, posedge rst) begin
        if(rst) begin
            dst_out <= 0;
            ALU_res_out <= 0;
            mem_data_out <= 0;
            mem_read_out <= 0;
            WB_en_out <= 0;
        end
        else begin
            dst_out <= dst;
            ALU_res_out <= ALU_res;
            mem_data_out <= mem_data;
            mem_read_out <= mem_read;
            WB_en_out <= WB_en;
        end
    end
endmodule
```

در این رجیستر مقادیر dest، ALU_res، mem_data،mem_read و WB_en نگهداری می‌شود.

:WB_Stage

```
module WB_Stage(clk, rst, dst, ALU_res, mem_data, mem_read, WB_en, WB_dst, WB_en_out, WB_value);

    input clk, rst;
    input [`REG_FILE_ADDRESS_LEN-1:0] dst;
    input [`WORD_WIDTH-1:0] ALU_res;
    input [`WORD_WIDTH-1:0] mem_data;
    input mem_read, WB_en;

    output [`REG_FILE_ADDRESS_LEN-1:0] WB_dst;
    output WB_en_out;
    output [`WORD_WIDTH-1:0] WB_value;

    assign WB_dst = dst;
    assign WB_en_out = WB_en;

    Mux2To1 #(`WORD_WIDTH) MUX_2_to_1_Reg_File (.out(WB_value), .in1(ALU_res), .in2(mem_data), .sel(mem_read));

endmodule
```

در این مرحله یک موبت پلکسر وجود دارد که تعیین می‌کند مقدار خروجی از حافظه و یا مقدار خروجی ALU در رجیستر نوشته شود؛ قسمت select این موبتی پلکسر mem_read_enable است.

:ARM

```
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_S_out;
    wire [`REG_FILE_ADDRESS_LEN-1:0] IDreg_regFile_dst;
    wire [3:0] IDreg_execute_cmd_out, IDreg_status_reg_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;


    //EXE Stage
    wire EXE_wb_en_out, EXE_mem_read_out, EXE_mem_write_out;
    wire [`WORD_WIDTH-1:0] EXE_alu_out, EXE_val_Rm_out;
    wire [`REG_FILE_ADDRESS_LEN-1:0] EXE_dst_out;

    //EXE Reg
    wire EXEreg_wb_en_out, EXEreg_mem_read_out, EXEreg_mem_write_out;
    wire [`WORD_WIDTH-1:0] EXEreg_alu_out, EXEreg_val_Rm_out;
    wire [`REG_FILE_ADDRESS_LEN-1:0] EXEreg_dst_out;

    //MEM Stage
    wire MEM_wb_en_out, MEM_mem_read_out;
    wire [`WORD_WIDTH-1:0] MEM_alu_out;
    wire [`REG_FILE_ADDRESS_LEN-1:0] MEM_dst_out;
    wire [`WORD_WIDTH-1:0] mem_out;

    //MEM Reg
    wire MEMreg_wb_en_out, MEMreg_mem_read_out;
    wire [`WORD_WIDTH-1:0] MEMreg_alu_out;
    wire [`REG_FILE_ADDRESS_LEN-1:0] MEMreg_dst_out;
    wire [`WORD_WIDTH-1:0] MEMreg_mem_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),  .pc_in(IFRegPcOut), .instruction_in(IFStageInstructionOut),
                      .reg_file_result_WB(WB_value), .reg_file_dest_wb(WB_dst_out), .status_register(status_reg_out), .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(branch_taken), .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),
                              .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_S_out),
                              .reg_file_dst_out(IDreg_regFile_dst), .execute_cmd_out(IDreg_execute_cmd_out), .status_register_out(IDreg_status_reg_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), .pc_in(IDreg_pc_out), .mem_read_in(IDreg_mem_read_out), .mem_write_in(IDreg_mem_write_out), .S(IDreg_S_out), .B(IDreg_B_out),
                          .WB_en_in(IDreg_WB_en_out), .execute_cmd_in(IDreg_execute_cmd_out), .immediate_in(IDreg_imm_out), .signed_immediate_24_in(IDreg_signed_immediate),
                          .shift_operand_in(IDreg_shifter_operand), .val_Rn_in(IDreg_val_Rn), .val_Rm_in(IDreg_val_Rm), .SR_in(IDreg_status_reg_out), .dst_in(IDreg_regFile_dst),
                          .alu_out(EXE_alu_out), .branch_address(branchAddr), .status_reg_out(status_reg_out), .branch_taken_out(branch_taken), .WB_en_out(EXE_wb_en_out),
                          .mem_read_out(EXE_mem_read_out), .mem_write_out(EXE_mem_write_out), .dst_out(EXE_dst_out), .val_Rm_out(EXE_val_Rm_out));

    Exec_Stage_Reg Exec_stage_reg(.clk(clk), .rst(rst), .dst_in(EXE_dst_out), .mem_read_in(EXE_mem_read_out), .mem_write_in(EXE_mem_write_out), .WB_en_in(EXE_wb_en_out),
                                  .val_Rm_in(EXE_val_Rm_out), .ALU_res_in(EXE_alu_out), .dst_out(EXEreg_dst_out), .ALU_res_out(EXEreg_alu_out), .val_Rm_out(EXEreg_val_Rm_out),
                                  .mem_read_out(EXEreg_mem_read_out), .mem_write_out(EXEreg_mem_write_out), .WB_en_out(EXEreg_wb_en_out));

    Mem_Stage Mem_stage(.clk(clk), .rst(rst), .dst(EXEreg_dst_out), .ALU_res(EXEreg_alu_out), .val_Rm(EXEreg_val_Rm_out), .mem_read(EXEreg_mem_read_out), .mem_write(EXEreg_mem_write_out),
                        .WB_en(EXEreg_wb_en_out), .dst_out(MEM_dst_out), .ALU_res_out(MEM_alu_out), .mem_out(mem_out), .mem_read_out(MEM_mem_read_out), .WB_en_out(MEM_wb_en_out));

    Mem_Stage_Reg Mem_stage_reg(.clk(clk), .rst(rst), .dst(MEM_dst_out), .ALU_res(MEM_alu_out), .mem_data(mem_out), .mem_read(MEM_mem_read_out), .WB_en(MEM_wb_en_out),
                                .dst_out(MEMreg_dst_out), .ALU_res_out(MEMreg_alu_out), .mem_data_out(MEMreg_mem_out), .mem_read_out(MEMreg_mem_read_out), .WB_en_out(MEMreg_wb_en_out));

    WB_Stage wb_stage(.clk(clk), .rst(rst), .dst(MEMreg_dst_out), .ALU_res(MEMreg_alu_out), .mem_data(MEMreg_mem_out), .mem_read(MEMreg_mem_read_out), .WB_en(MEMreg_wb_en_out),
                      .WB_dst(WB_dst_out), .WB_en_out(WB_WB_en_out), .WB_value(WB_value));
endmodule
```

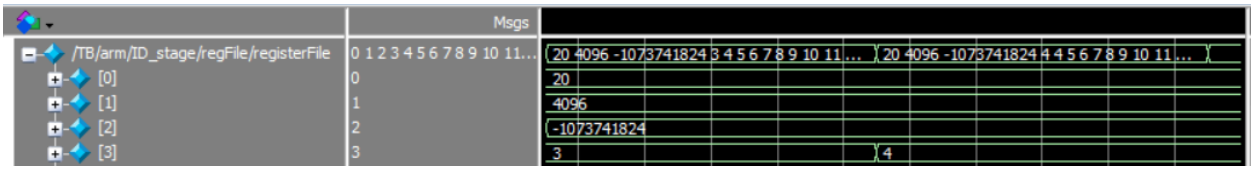
در اینجا wire های مربوط را به ماژول‌های نمونه‌گیری شده وصل می‌کنیم، برای مثال خروجی branch_taken به ورودی flush دو رجیستر اول و select مولتی پلکسر موجود در مرحله IF وصل می‌شود. همچنین WB_Dest، WB_value و WB_WB_EN از مرحله WB به رجیستر فایل که در مرحله ID است متصل می‌شود.

توضیح 4 دستور برنامه:

- دستور چهارم:

1110_00_0_0100_1_0010_0011_000000000010

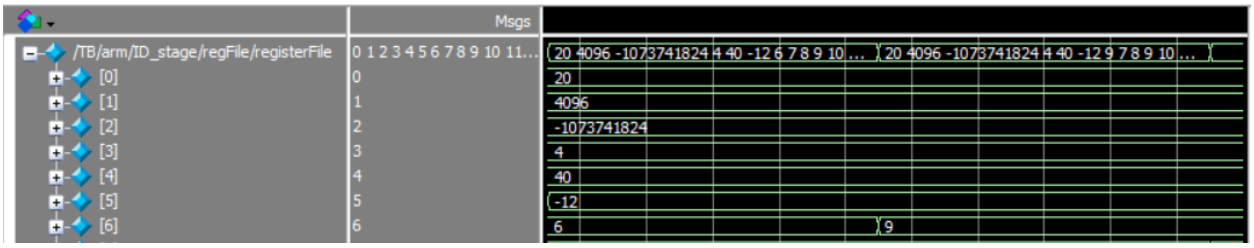
دستور چهارم، دستور ADD می‌باشد که مقدار رجیستر دوم را با خودش جمع کرده و در رجیستر سوم ذخیره می‌کند. از آنجایی که مقدار رجیستر دوم 2 است، بنابراین پاسخی که در رجیستر سوم ذخیره میشود برابر 4 خواهد بود. در تصویر زیر مقدار رجیستر را قبل و بعد از تغییر مشاهده می‌کنیم.



- دستور هفتم:

1110_00_0_0110_0_0000_0110_000010100000

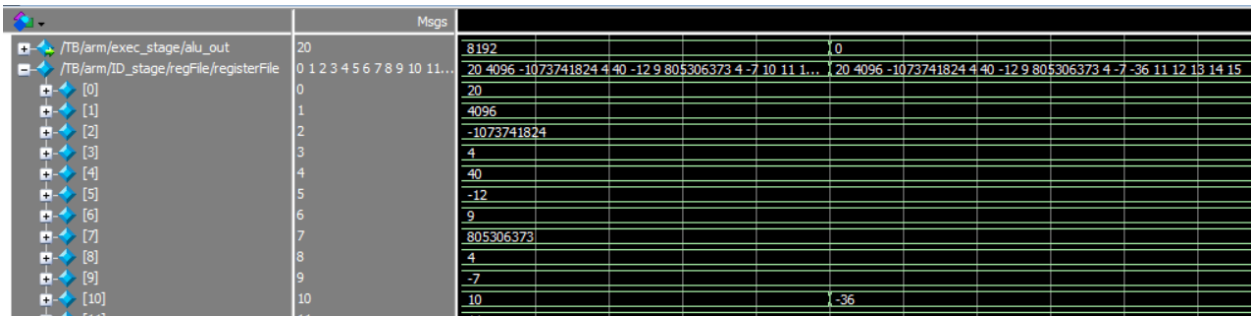
دستور هفتم، دستور SBC می‌باشد که مقدار in1 - in2 – ~C را در رجیستر ذخیره می‌کند. در اینجا اما یک Logical Shift Right داریم، که یعنی باید رجیستر صفرم را 1 بیت به راست شیفت داده و سپس عملیات را انجام دهیم. مقدار رجیستر صفرم برابر 20 است، که با شیفت دادن مقدار آن برابر 10 می‌شود، سپس با انجام عملیات مقدار نهایی آن برابر 9 خواهد شد.



- دستور یازدهم:

1110_00_0_0001_0_0100_1010_000000000101

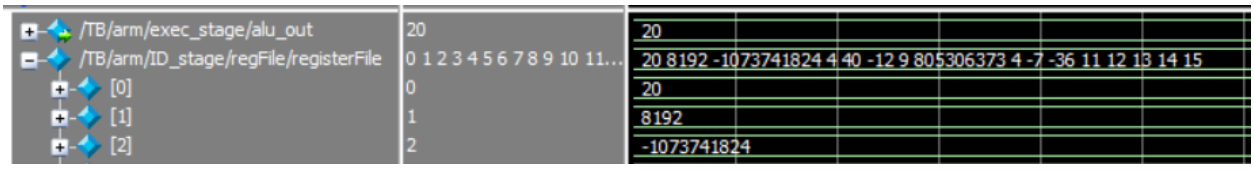
دستور یازدهم، دستور EOR می‌باشد که بیت های رجیستر هارا با هم xor می‌کند. در اینجا بیت های دو رجیستر 4 ام و 5 ام را با هم xor کرده و در رجیستر دهم ذخیره می‌کنیم، از آنجایی که مقدار این رجیسترها به ترتیب 40 و 12- است، بنابراین نتیجه نهایی برابر 36- می‌شود که در شکل قابل مشاهده است.



- دستور پانزدهم:

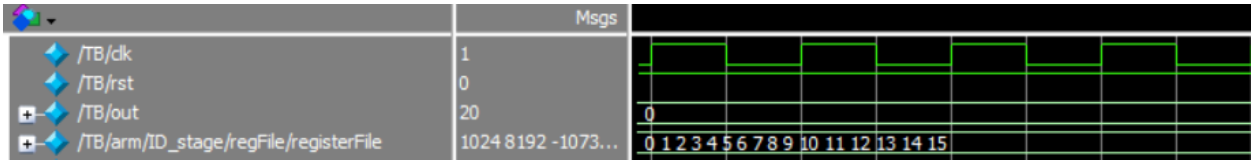
0000_00_0_0100_0_0010_0010_000000000010

دستور پانزدهم، دستور ADDEQ می‌باشد که درصورتیکه شرط EQ برقرار باشد، رجیسترهای داده شده را با هم جمع کرده و نتیجه را در رجیستر سوم ذخیره می‌کند. در اینجا شرط نیز در صورت برقراری شرط، نتیجه را در رجیستر دوم ذخیره می‌کند.



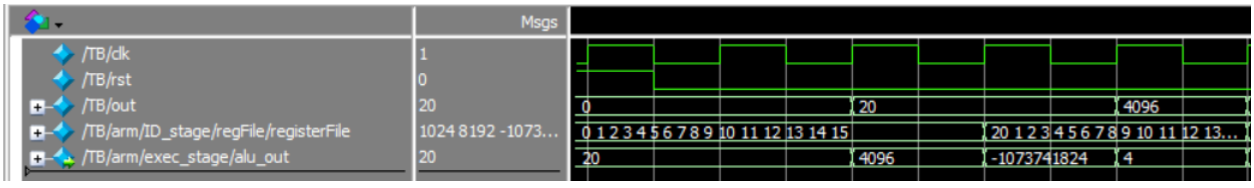
خروجی 18 دستور:

مقادیر اولیه رجیسترها:



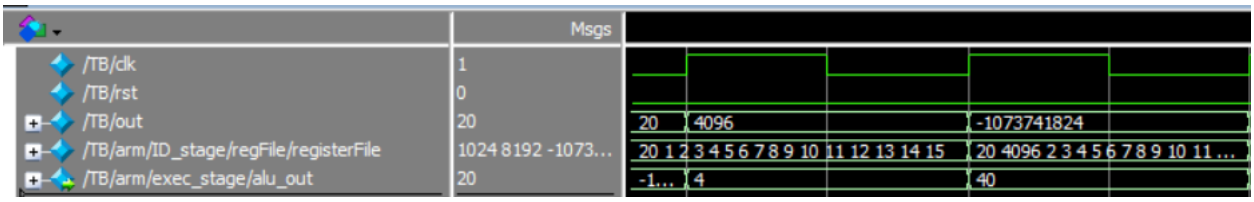
دستور اول:

1110_00_1_1101_0_0000_0000_000000010100 → MOV R0 = 20



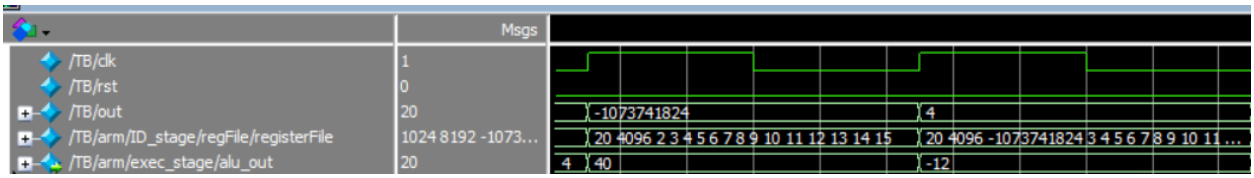
دستور دوم:

1110_00_1_1101_0_0000_0001_101000000001 → MOV R1, #4096



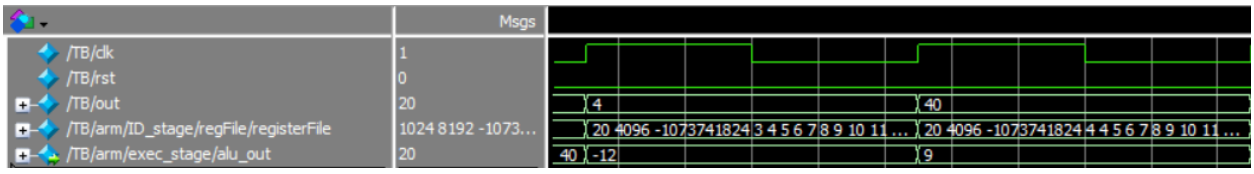
دستور سوم:

1110_00_1_1101_0_0000_0010_000100000011 → MOV R2, #0xC0000000



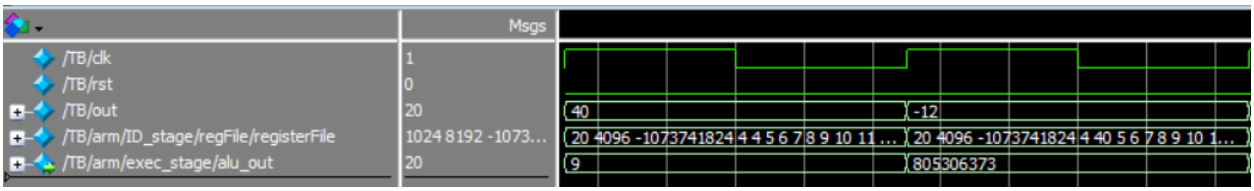
دستور چهارم:

1110_00_0_0100_1_0010_0011_000000000010 → ADDS R3, R2, R2



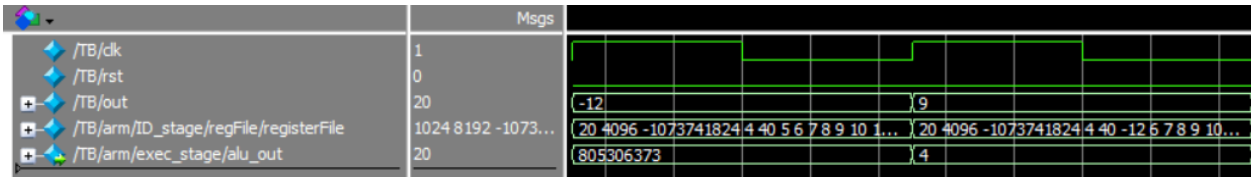
دستور پنجم:

1110_00_0_0101_0_0000_0100_000000000000 → ADC R4, R0, R0



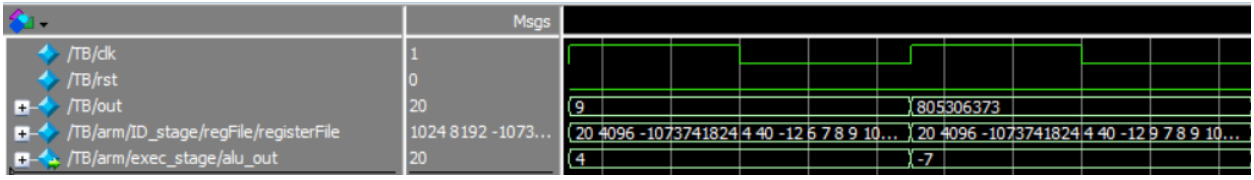
دستور ششم:

1110_00_0_0010_0_0100_0101_000100000100 → SUB R5, R4, R4, LSL #2



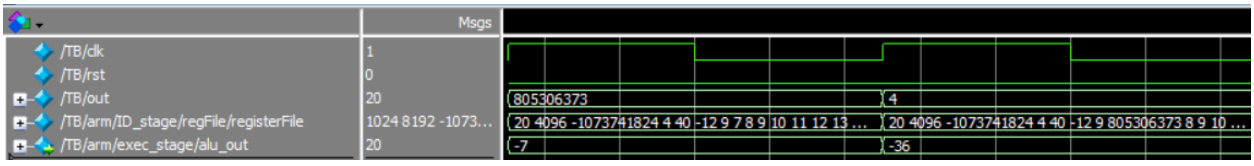
دستور هفتم:

1110_00_0_0110_0_0000_0110_000010100000 → SBC R6, R0, R0, LSR #1



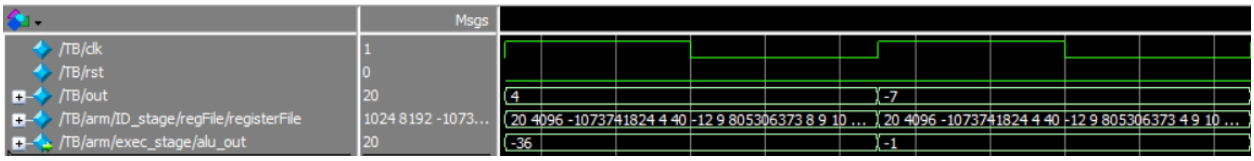
دستور هشتم:

1110_00_0_1100_0_0101_0111_000101000010 → ORR R7, R5, R2, ASR #2



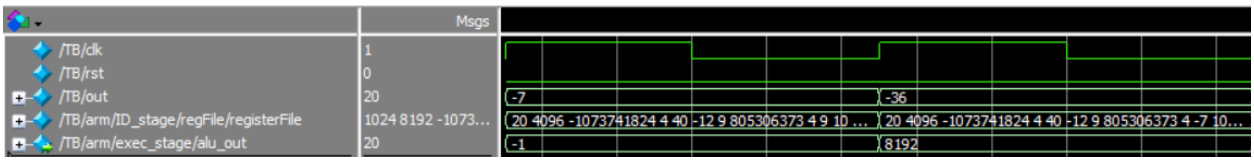
دستور نهم:

1110_00_0_0000_0_0111_1000_000000000011 → AND R8, R7, R3



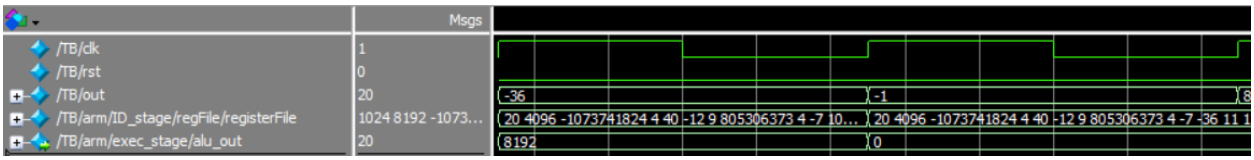
دستور دهم:

1110_00_0_1111_0_0000_1001_000000000110 → MVN R9, R6



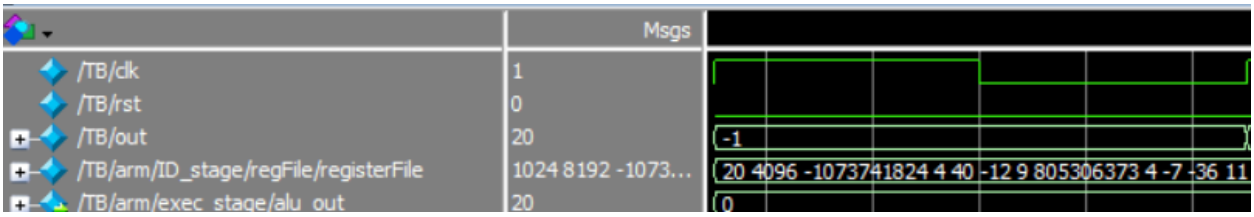
دستور یازدهم:

1110_00_0_0001_0_0100_1010_000000000101 → EOR R10, R4, R5



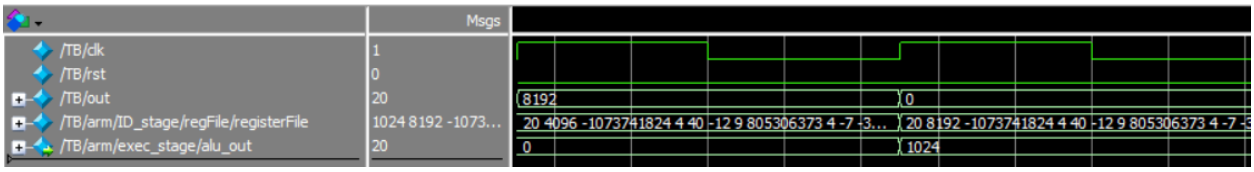
دستور دوازدهم:

1110_00_0_1010_1_1000_0000_000000000110 → CMP R8, R6



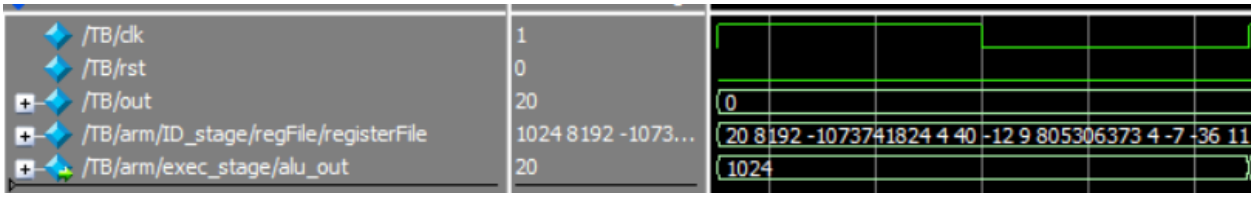
دستور سیزدهم:

0001_00_0_0100_0_0001_0001_000000000001 → ADDNE R1, R1, R1



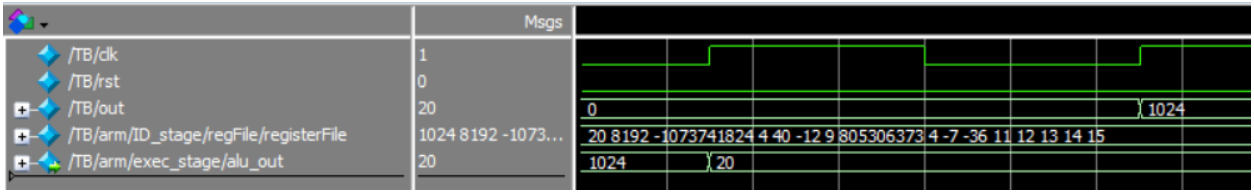
دستور چهاردهم:

1110_00_0_1000_1_1001_0000_000000001000 → TST R9, R8



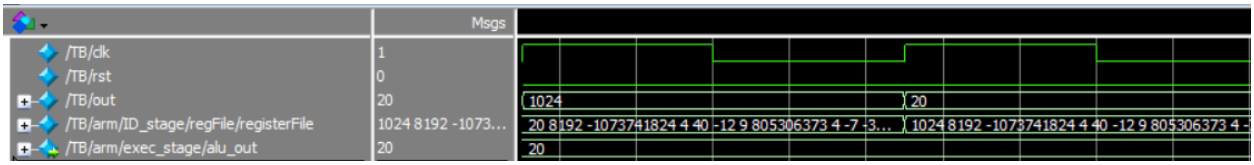
دستور پانزدهم:

0000_00_0_0100_0_0010_0010_000000000010 → ADDEQ R2, R2, R2



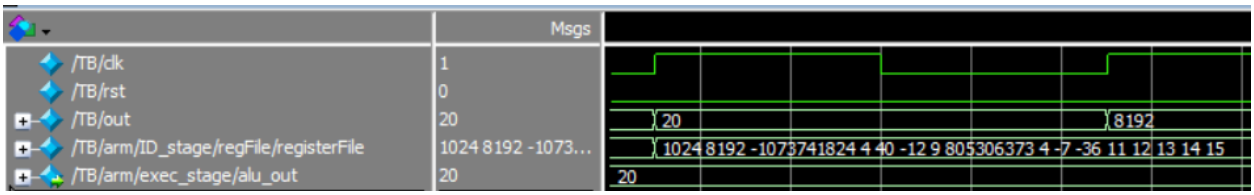
دستور شانزدهم:

1110_00_1_1101_0_0000_0000_101100000001 → MOV R0, #1024



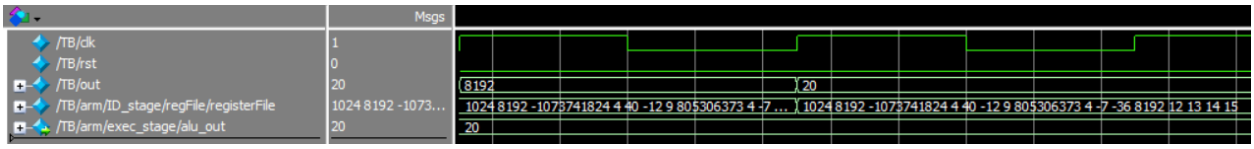
دستور هفدهم:

1110_01_0_0100_0_0000_0001_000000000000 → STR R1, [R0], #0



دستور هجدهم:

1110_01_0_0100_1_0000_1011_000000000000 → LDR R11, [R0], #0



خروجی نهایی: