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 reg [`INSTRUCTION_LEN - 1 : 0] read_data;

    reg[`INSTRUCTION_LEN - 1 : 0] data[0:`DATA_MEM_SIZE - 1];

    integer i;
    wire [31:0] dataAdr, adr;
    assign dataAdr = addr - 32'd1024;
    assign adr = {2'b00, dataAdr[31:2]};

    always @(mem_read or adr) begin
        if (mem_read)
            read_data = data[adr];
    end

    always@(negedge clk, posedge rst)begin
        if(rst)
            for(i = 0; i < `DATA_MEM_SIZE; i = i + 1)
                data[i] <= 32'd0;
        if (mem_write)begin
            data[adr] = write_data;
        end
    end

endmodule
```

همانطور که در توضیحات گفته شده و در کد نیز مشخص است، خواندن بصورت Combinational و نوشتن بصورت Sequential است. addr ای که دیتا در آن نوشته و یا از آن خوانده می‌شود، عدد بدست آمده از ALU در مرحله قبل است. از آنجایی که فضای آدرس‌دهی حافظه از ۱۰۲۴ شروع می‌شود، باید ۱۰۲۴ را از آدرس بدست آمده کم کنیم. زمانی که mem_write برابر با ۱ است ، داده را در خانه ای از حافظه که توسط adr مشخص شده می‌نویسیم. زمانی که mem_read برابر با ۱ است مقدار موجود در آدرس adr حافظه را در read_data می‌ریزیم.

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 تنها یک multiplexer وجود دارد که از بین خروجی ALU و خروجی حافظه، مقداری که باید در رجیستر فایل نوشته بشود را مشخص می‌کند. select این multiplexer سیگنال mem_read یا همان خواندن از حافظه است، چرا که در صورتی که دستور مرحله قبل خواندن از حافظه باشد، خروجی حافظه و در غیر این صورت خروجی ALU انتخاب می‌شود.

```
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));
```

بنابراین پس از کامل کردن این دو قسمت، بخش WriteBack و Memory پردازنده تکمیل می‌شود. در این مرحله از این دو بخش درTop module ARM نمونه گرفته و ورودی و خروجی های مربوطه را به آن متصل می‌کنیم.

Hazard_Detector:

```
module Hazard_Detector (src1, src2, exe_wb_dest, mem_wb_dest, two_src,  exe_wb_enable, mem_wb_enable, hazard);

    input [`REG_FILE_ADDRESS_LEN - 1:0] src1, src2;
    input [`REG_FILE_ADDRESS_LEN - 1:0] exe_wb_dest;
    input [`REG_FILE_ADDRESS_LEN - 1:0] mem_wb_dest;
    input two_src, exe_wb_enable, mem_wb_enable;

    output reg hazard;
    always @(*)
    begin
        if ((src1 == exe_wb_dest) && (exe_wb_enable == 1'b1))
            hazard = 1'b1;
        else
        if ((src1 == mem_wb_dest) && (mem_wb_enable == 1'b1))
            hazard = 1'b1;
        else
        if ((src2 == exe_wb_dest) && (exe_wb_enable == 1'b1) && (two_src == 1'b1))
            hazard = 1'b1;
        else
        if ((src2 == mem_wb_dest) && (mem_wb_enable == 1'b1) && (two_src == 1'b1))
            hazard = 1'b1;
        else
            hazard = 1'b0;
    en

endmodule
```

ماژول Hazard_detector وظیفه تشخیص مخاطره داده‌ها در پردازنده را به عهده دارد. مخاطره داده زمانی رخ می دهد که یک دستورالعمل به نتیجه دستورالعمل قبلی که هنوز کامل نشده است بستگی داشته باشد. این می تواند منجر به نتایج نادرست یا توقف در خط لوله شود که عملکرد را کاهش می دهد. Hazard_detector، مخاطرات بین دستورالعمل فعلی و دو دستورالعمل قبلی را در خط لوله بررسی می کند (یعنی دستورالعملی که در حال حاضر در مرحله اجرا است و دستورالعمل فعلی در مرحله حافظه). اگر بین دستورالعمل فعلی و هر یک از این دستورالعمل ها خطر داده وجود داشته باشد، خروجی خطر روی بالا تنظیم می شود. توجه داشته باشید که این ماژول فقط خطرات ناشی از وابستگی داده ها بین دستورالعمل ها را تشخیص می دهد. انواع دیگر خطرات، مانند خطرات کنترلی یا مخاطرات ساختاری، با استفاده از مکانیسم های دیگر شناسایی می شوند.

ورودی های این ماژول شامل src1, src2, exe_wb_dest, mem_wb_dest, two_src, exe_wb_enable, mem_wb_enable و خروجی hazard می‌باشد و این خروجی در ۴ حالتی که در ادامه گفته می‌شود، یک خواهد شد:

- برابری src1 با مقصد exe_wb_dest در صورت یک بودن exe_wb_enable در مرحله اجرا
- برابری src1 با مقصد mem_wb_dest در صورت یک بودن mem_wb_enable در مرحله حافظه
- برابری src2 با مقصد exe_wb_dest در صورت یک بودن exe_wb_enable در مرحله اجرا و دو منبعی بودن دستور
- برابری src2 با مقصد mem_wb_dest در صورت یک بودن mem_wb_enable در مرحله حافظه و دو منبعی بودن دستور

```
assign two_src = ID_mem_write_out | ~ID_imm_out;
Hazard_Detector hazard_detector(.src1(ID_regFile_src1), .src2(ID_regFile_src2), .exe_wb_dest(EXE_dst_out),
                                .mem_wb_dest(MEM_dst_out), .two_src(two_src),  .exe_wb_enable(EXE_wb_en_out),
                                .mem_wb_enable(MEM_wb_en_out), .hazard(freeze));
```

پس از پیاده‌سازی ماژول های جدید، ماژول های قبلی را با توجه به قابلیت های جدید تغییر می‌دهیم. قابلیت Freeze را به رجیستر PC در مرحله IF و رجیسترهای بعد از IF اضافه نمایید و خروجی واحد تشخیص هازارد را به آن متصل می‌کنیم.

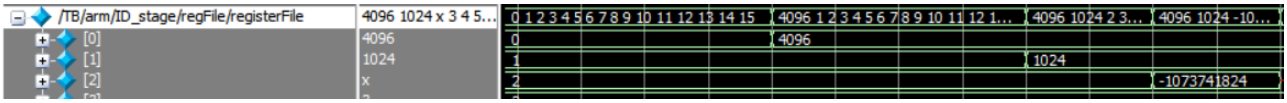
```
module IF_Stage(clk, rst, freeze, branch_taken, branchAddr, pc, instruction);
    input clk, rst, freeze, branch_taken;
    input [31:0] branchAddr;
    output [31:0] pc, instruction;

    wire [31:0] adderOut;
    wire [31:0] PCout;
    wire [31:0] muxIFout;

    Adder adder(.a(PCout), .b(32'd4), .res(adderOut));
    Instruction_Memory instMem(.rst(rst), .addr(PCout), .read_instruction(instruction));
    Mux2To1 muxIF(.out(muxIFout),.in1(adderOut), .in2(branchAddr), .sel(branch_taken));
    PCRegister PCReg(.out(PCout), .in(muxIFout), .rst(rst), .freeze(freeze), .clk(clk));

    assign pc = adderOut;

endmodule
```

همچنین برای ایجاد حباب در خط لوله، باید سیگنال های کنترلی خروجی از واحد کنترل را صفر کنیم. برای اینکار خروجی ماژول مخاطره (Hazard_detector) را با خروجی بررسی شرط دستورات (Conditon_check) or می‌کنیم.

```
assign ctrl_unit_mux_enable = (~condition_state) | freeze;
```

چند دستور بدون داشتن hazard را بررسی می‌کنیم.

```
{instruction[0], instruction[1], instruction[2], instruction[3]} <= `INSTRUCTION_LEN'b1110_00_1_1101_0_0000_0000_101000000001;
{instruction[4], instruction[5], instruction[6], instruction[7]} <= `INSTRUCTION_LEN'b1110_00_1_1101_0_0000_0001_101100000001;
{instruction[8], instruction[9], instruction[10], instruction[11]} <= `INSTRUCTION_LEN'b1110_00_1_1101_0_0000_0010_000100000011;
```

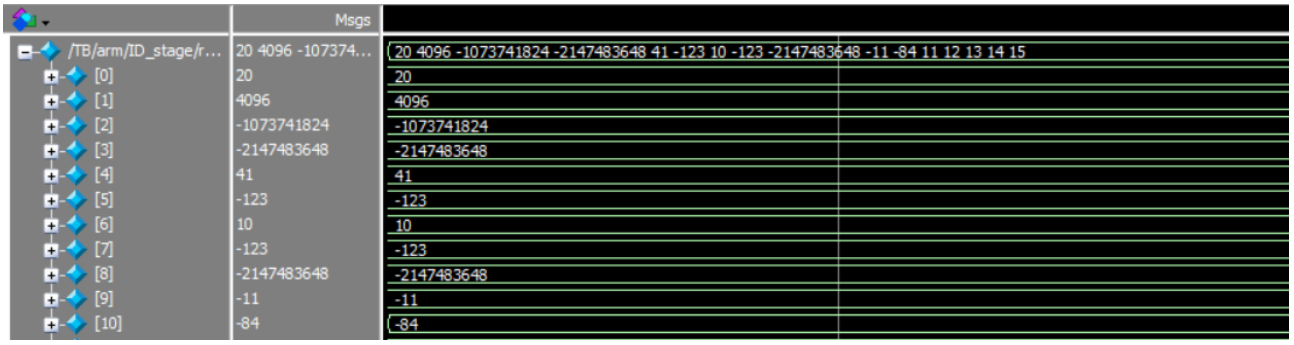


سپس دستورات ۱۸ تایی instruction را کامل کرده و برنامه را تست می‌کنیم.

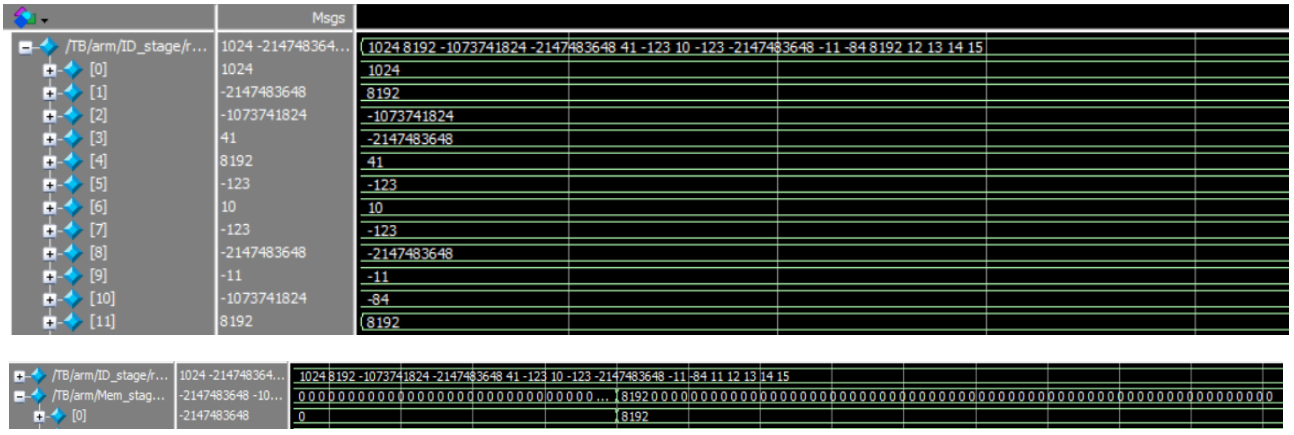
```
module Instruction_Memory(rst, addr, read_instruction);
    input [`INSTRUCTION_LEN - 1 : 0] addr;
    input rst;
    output [`INSTRUCTION_LEN - 1 : 0] read_instruction;

    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],   instructi.n[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,  #0xC00000000  -> 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_000101100000; // 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; // MVN   R9,  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; // CMP   R8,  R6            -> z = 0
            {instruction[48],  instruction[49],  instruction[50],  instruction[51]}  <= `INSTRUCTION_LEN'b0001_00_0_0100_0_0001_0001_000000000001; // ADDNE R1,  R1, R1        -> R1 = 8192
            {instruction[52],  instruction[53],  instruction[54],  instruction[55]}  <= `INSTRUCTION_LEN'b1110_00_0_1000_1_1001_0000_000000001000; // TST   R9,  R8            -> z = 0
            {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; // MOV   R0,  #1024         -> R0 = 1024
            {instruction[64],  instruction[65],  instruction[66],  instruction[67]}  <= `INSTRUCTION_LEN'b1110_01_0_0100_0_0000_0001_000000000000; // STR   R1,  [R0], #0      -> MEM[1024] = 8192
            {instruction[68],  instruction[69],  instruction[70],  instruction[71]}  <= `INSTRUCTION_LEN'b1110_01_0_0100_1_0000_1011_000000000000; // LDR   R11, [R0], #0      -> R11 = 8192
            {instruction[72],  instruction[73],  instruction[74],  instruction[75]}  <= `INSTRUCTION_LEN'b1110_01_0_0100_0_0000_0010_000000000100; // STR   R2,  [R0], #4      -> MEM[1028] = -1073741824
            {instruction[76],  instruction[77],  instruction[78],  instruction[79]}  <= `INSTRUCTION_LEN'b1110_01_0_0100_0_0000_0011_000000001000; // STR   R3,  [R0], #8      -> MEM[1032] = -2147483648
            {instruction[80],  instruction[81],  instruction[82],  instruction[83]}  <= `INSTRUCTION_LEN'b1110_01_0_0100_0_0000_0100_000000001101; // STR   R4,  [R0], #13     -> MEM[1036] = 41
            {instruction[84],  instruction[85],  instruction[86],  instruction[87]}  <= `INSTRUCTION_LEN'b1110_01_0_0100_0_0000_0101_000000010000; // STR   R5,  [R0], #16     -> MEM[1040] = -123
            {instruction[88],  instruction[89],  instruction[90],  instruction[91]}  <= `INSTRUCTION_LEN'b1110_01_0_0100_0_0000_0110_000000010100; // STR   R6,  [R0], #20     -> MEM[1044] = 10
            {instruction[92],  instruction[93],  instruction[94],  instruction[95]}  <= `INSTRUCTION_LEN'b1110_01_0_0100_1_0000_1010_000000000100; // LDR   R10, [R0], #4      -> R10 = -1073741824
            {instruction[96],  instruction[97],  instruction[98],  instruction[99]}  <= `INSTRUCTION_LEN'b1110_01_0_0100_0_0000_0111_000000011000; // STR   R7,  [R0], #24     -> MEM[1048] = -123
            {instruction[100], instruction[101], instruction[102], instruction[103]} <= `INSTRUCTION_LEN'b1110_00_1_1101_0_0000_0001_000000000100; // MOV   R1,  #4            -> R1 = 4
            {instruction[104], instruction[105], instruction[106], instruction[107]} <= `INSTRUCTION_LEN'b1110_00_1_1101_0_0000_0010_000000000000; // MOV   R2,  #0            -> R2 = 0
            {instruction[108], instruction[109], instruction[110], instruction[111]} <= `INSTRUCTION_LEN'b1110_00_1_1101_0_0000_0011_000000000000; // MOV   R3,  #0            -> R3 = 0
            {instruction[112], instruction[113], instruction[114], instruction[115]} <= `INSTRUCTION_LEN'b1110_00_0_0100_0_0000_0100_000100000011; // ADD   R4,  R0, R3, LSL #2 -> R4 = 1024
            {instruction[116], instruction[117], instruction[118], instruction[119]} <= `INSTRUCTION_LEN'b1110_01_0_0100_1_0100_0101_000000000000; // LDR   R5,  [R4], #0      -> R5 = 8192
            {instruction[120], instruction[121], instruction[122], instruction[123]} <= `INSTRUCTION_LEN'b1110_01_0_0100_1_0100_0110_000000000100; // LDR   R6,  [R4], #4      -> R6 = -1073741824
            {instruction[124], instruction[125], instruction[126], instruction[127]} <= `INSTRUCTION_LEN'b1110_00_0_1010_1_0101_0000_000000000110; // CMP   R5,  R6            -> z = 0, n = 0, v = 0
            {instruction[128], instruction[129], instruction[130], instruction[131]} <= `INSTRUCTION_LEN'b1100_01_0_0100_0_0100_0110_000000000000; // STRGT R6,  [R4], #0      -> MEM[1024] = -1073741824
            {instruction[132], instruction[133], instruction[134], instruction[135]} <= `INSTRUCTION_LEN'b1100_01_0_0100_0_0100_0101_000000000100; // STRGT R5,  [R4], #4      -> MEM[1028] = 8192
            {instruction[136], instruction[137], instruction[138], instruction[139]} <= `INSTRUCTION_LEN'b1110_00_1_0100_0_0011_0011_000000000001; // ADD   R3,  R3,   #1      -> R3 = 1
            {instruction[140], instruction[141], instruction[142], instruction[143]} <= `INSTRUCTION_LEN'b1110_00_1_1010_1_0011_0000_000000000011; // CMP   R3,  #3            -> z = 0, n = 1, v = 0
            {instruction[144], instruction[145], instruction[146], instruction[147]} <= `INSTRUCTION_LEN'b1011_10_1_0_11111111111111111110111;    // BLT   #-9                -> PC = 32'd112
            {instruction[148], instruction[149], instruction[150], instruction[151]} <= `INSTRUCTION_LEN'b1110_00_1_0100_0_0010_0010_000000000001; // ADD   R2,  R2,   #1      -> R2 = -2147483648
            {instruction[152], instruction[153], instruction[154], instruction[155]} <= `INSTRUCTION_LEN'b1110_00_0_1010_1_0010_0000_000000000001; // CMP   R2,  R1            -> z = 0, n = 1, v = 0
            {instruction[156], instruction[157], instruction[158], instruction[159]} <= `INSTRUCTION_LEN'b1011_10_1_0_11111111111111111110011;    // BLT   #-13               -> PC = 32'd112
            {instruction[160], instruction[161], instruction[162], instruction[163]} <= `INSTRUCTION_LEN'b1110_01_0_0100_1_0000_0001_000000000000; // LDR   R1,  [R0], #0      -> R1 = -2147483648
            {instruction[164], instruction[165], instruction[166], instruction[167]} <= `INSTRUCTION_LEN'b1110_01_0_0100_1_0000_0010_000000000100; // LDR   R2,  [R0], #4      -> R2 = -1073741824
            {instruction[168], instruction[169], instruction[170], instruction[171]} <= `INSTRUCTION_LEN'b1110_01_0_0100_1_0000_0011_000000001000; // LDR   R3,  [R0], #8      -> R3 = 41
            {instruction[172], instruction[173], instruction[174], instruction[175]} <= `INSTRUCTION_LEN'b1110_01_0_0100_1_0000_0100_000000001100; // LDR   R4,  [R0], #12     -> R4 = 8192
            {instruction[176], instruction[177], instruction[178], instruction[179]} <= `INSTRUCTION_LEN'b1110_01_0_0100_1_0000_0101_000000010000; // LDR   R5,  [R0], #16     -> R5 = -123
            {instruction[180], instruction[181], instruction[182], instruction[183]} <= `INSTRUCTION_LEN'b1110_01_0_0100_1_0000_0110_000000010100; // LDR   R6,  [R0], #20     -> R4 = 10
            {instruction[184], instruction[185], instruction[186], instruction[187]} <= `INSTRUCTION_LEN'b1110_10_1_0_11111111111111111111111;    // B   #-1                -> PC = 32'd184 (infinite loop)
        end
    end

endmodule
```

نتایج بدست آمده در waveform را نمایش داده و با جواب های موجود چک میکنیم:

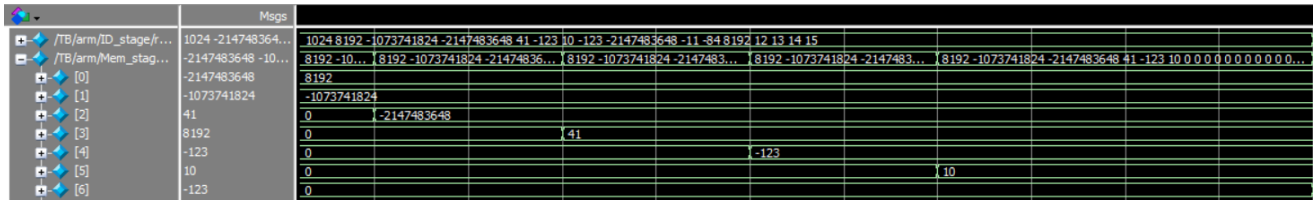
```
`INSTRUCTION_LEN'b1110_00_1_1101_0_0000_0000_000000010100; // MOV   R0,  #20            -> R0 = 20
`INSTRUCTION_LEN'b1110_00_1_1101_0_0000_0001_101000000001; // MOV   R1,  #4096          -> R1 = 4096
`INSTRUCTION_LEN'b1110_00_1_1101_0_0000_0010_000100000011; // MOV   R2,  #0xC06000000   -> R2 = -1073741824
`INSTRUCTION_LEN'b1110_00_0_0100_1_0010_0011_000000000010; // ADDS  R3,  R2, R2         -> R3 = -2147483648
`INSTRUCTION_LEN'b1110_00_0_0101_0_0000_0100_000000000000; // ADC   R4,  R0, R0         -> R4 = 41
`INSTRUCTION_LEN'b1110_00_0_0010_0_0100_0101_000100000100; // SUB   R5,  R4, R4, LSL #2 -> R5= -123
`INSTRUCTION_LEN'b1110_00_0_0110_0_0000_0110_000010100000; // SBC   R6,  R0, R0, LSR #1 -> R6 = 10
`INSTRUCTION_LEN'b1110_00_0_1100_0_0101_0111_000101000010; // ORR   R7,  R5, R2, ASR #2 -> R7 = -123
`INSTRUCTION_LEN'b1110_00_0_0000_0_0111_1000_000000000011; // AND   R8,  R7, R3         -> R8 = -2147483648
`INSTRUCTION_LEN'b1110_00_0_1111_0_0000_1001_000000000110; // MVN   R9,  R6             -> R9 = -11
`INSTRUCTION_LEN'b1110_00_0_0001_0_0100_1010_000000000101; // EOR   R10, R4, R5         -> R10 = -84
```



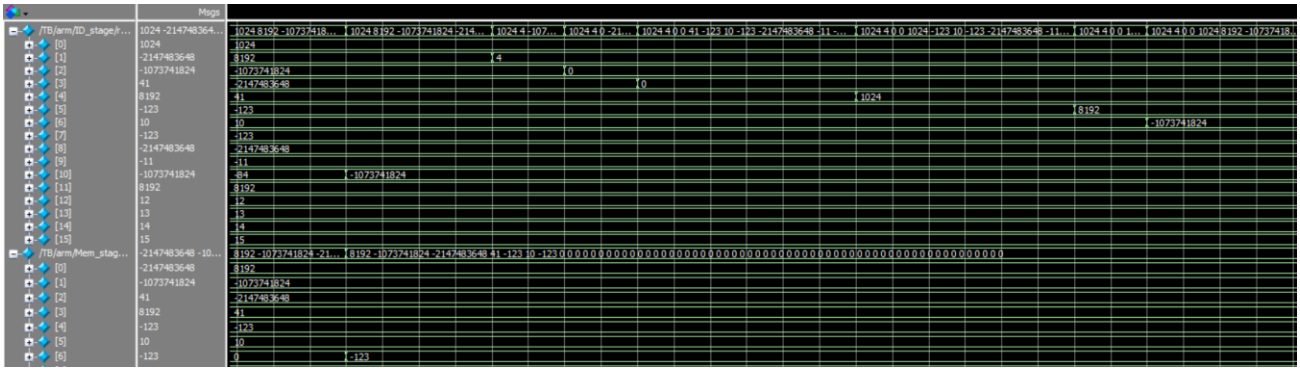
```
`INSTRUCTION_LEN'b1110_00_0_1010_1_1000_0000_000000000110; // CMP   R8,  R6             -> z = 0
`INSTRUCTION_LEN'b0001_00_0_0100_0_0001_0001_000000000001; // ADDNE R1,  R1, R1         -> R1 = 8192
`INSTRUCTION_LEN'b1110_00_0_1000_1_1001_0000_000000001000; // TST   R9,  R8             -> z = 0
`INSTRUCTION_LEN'b0000_00_0_0100_0_0010_0010_000000000010; // ADDEQ R2,  R2, R2         -> R2 = -1073741824
`INSTRUCTION_LEN'b1110_00_1_1101_0_0000_0000_101100000001; // MOV   R0,  #1024          -> R0 = 1024
`INSTRUCTION_LEN'b1110_01_0_0100_0_0000_0001_000000000000; // STR   R1,  [R0], #0       -> MEM[1024] = 8192
`INSTRUCTION_LEN'b1110_01_0_0100_1_0000_1011_000000000000; // LDR   R11, [R0], #0       -> R11 = 8192
```



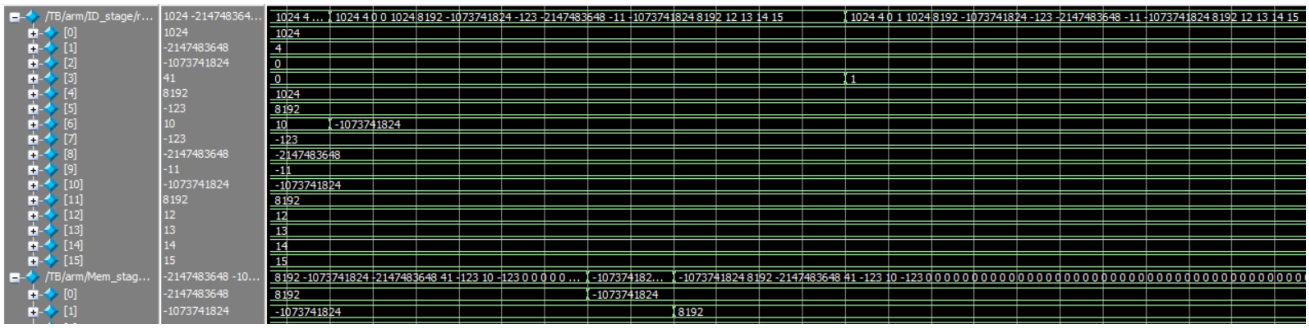
```
`INSTRUCTION_LEN'b1110_01_0_0100_0_0000_0010_000000000100; // STR     R2,  [R0], #4       -> MEM[1028] = -1073741824
`INSTRUCTION_LEN'b1110_01_0_0100_0_0000_0011_000000001000; // STR     R3,  [R0], #8       -> MEM[1032] = -2147483648
`INSTRUCTION_LEN'b1110_01_0_0100_0_0000_0100_000000001101; // STR     R4,  [R0], #13      -> MEM[1036] = 41
`INSTRUCTION_LEN'b1110_01_0_0100_0_0000_0101_000000010000; // STR     R5,  [R0], #16      -> MEM[1040] = -123
`INSTRUCTION_LEN'b1110_01_0_0100_0_0000_0110_000000010100; // STR     R6,  [R0], #20      -> MEM[1044] = 10
```



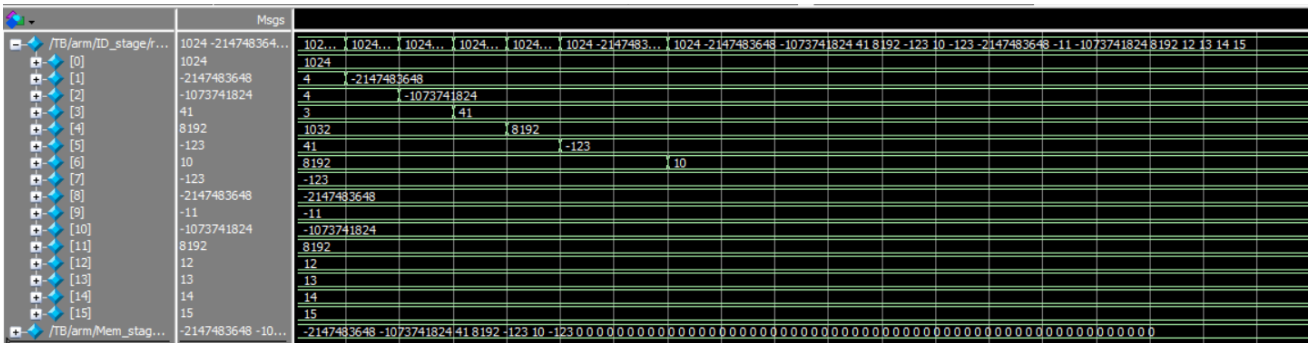
```
`INSTRUCTION_LEN'b1110_01_0_0100_1_0000_1010_000000000100; // LDR     R10, [R0], #4       -> R10 = -1073741824
`INSTRUCTION_LEN'b1110_01_0_0100_0_0000_0111_000000011000; // STR     R7,  [R0], #24      -> MEM[1048] = -123
`INSTRUCTION_LEN'b1110_00_1_1101_0_0000_0001_000000000100; // MOV     R1,  #4             -> R1 = 4
`INSTRUCTION_LEN'b1110_00_1_1101_0_0000_0010_000000000000; // MOV     R2,  #0             -> R2 = 0
`INSTRUCTION_LEN'b1110_00_1_1101_0_0000_0011_000000000000; // MOV     R3,  #0             -> R3 = 0
`INSTRUCTION_LEN'b1110_00_0_0100_0_0000_0100_000100000011; // ADD     R4,  R0, R3, LSL #2 -> R4 = 1024
`INSTRUCTION_LEN'b1110_01_0_0100_1_0100_0101_000000000000; // LDR     R5,  [R4], #0       -> R5 = 8192
`INSTRUCTION_LEN'b1110_01_0_0100_1_0100_0110_000000000100; // LDR     R6,  [R4], #4       -> R6 = -1073741824
```



```
`INSTRUCTION_LEN'b1100_01_0_0100_0_0100_0110_000000000000; // STRGT R6,  [R4], #0       -> MEM[1024] = -1073741824
`INSTRUCTION_LEN'b1100_01_0_0100_0_0100_0101_000000000100; // STRGT R5,  [R4], #4       -> MEM[1028] = 8192
`INSTRUCTION_LEN'b1110_00_1_0100_0_0011_0011_000000000001; // ADD     R3,  R3,   #1       -> R3 = 1
```

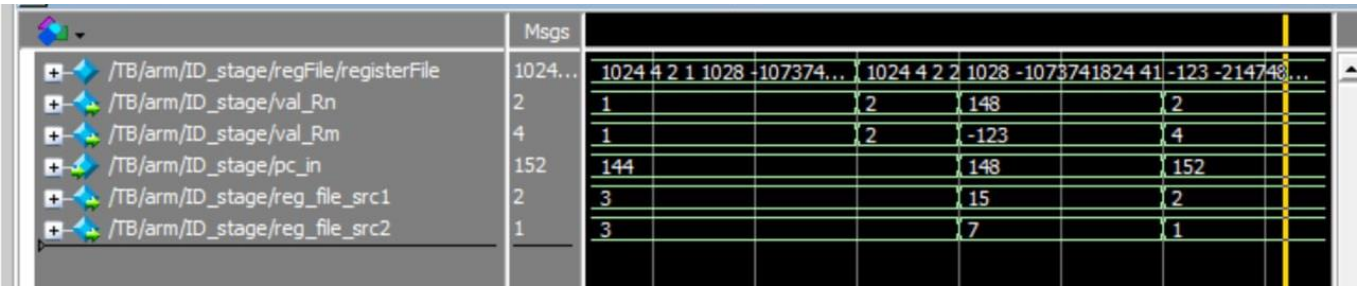


```
`INSTRUCTION_LEN'b1110_00_1_0100_0_0010_0010_000000000001; // ADD     R2,  R2,   #1       -> R2 = -2147483648
`INSTRUCTION_LEN'b1110_00_0_1010_1_0010_0000_000000000001; // CMP     R2,  R1            -> z = 0, n = 1, v = 0
`INSTRUCTION_LEN'b1011_10_1_0_11111111111111111110011;    // BLT     #-13               -> PC = 32'd112
`INSTRUCTION_LEN'b1110_01_0_0100_1_0000_0001_000000000000; // LDR     R1,  [R0], #0      -> R1 = -2147483648
`INSTRUCTION_LEN'b1110_01_0_0100_1_0000_0010_000000000100; // LDR     R2,  [R0], #4      -> R2 = -1073741824
`INSTRUCTION_LEN'b1110_01_0_0100_1_0000_0011_000000001000; // LDR     R3,  [R0], #8      -> R3 = 41
`INSTRUCTION_LEN'b1110_01_0_0100_1_0000_0100_000000001100; // LDR     R4,  [R0], #12     -> R4 = 8192
`INSTRUCTION_LEN'b1110_01_0_0100_1_0000_0101_000000010000; // LDR     R5,  [R0], #16     -> R5 = -123
`INSTRUCTION_LEN'b1110_01_0_0100_1_0000_0110_000000010100; // LDR     R6,  [R0], #20     -> R4 = 10
```



برای انجام بخش امتیازی نیز باید دو فایل ID_Stage و Register_file را تغییر دهیم، به اینصورت که دو مولتی پلکسر در مسیر خروجی Val_Rm و Val_Rn میگذاریم که ورودی هر یک به ترتیب Val_Rm و pc_in و Val_Rn و pc_in است.

```
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-2];
    integer i;

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

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

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



پیش از انجام این تغییرات، نتیجه این قسمت برابر x بود، زیرا وقتی src1 برابر ۱۵ می‌شد چون رجیسترفایل آیتم ۱۵ را نداشت، Val_Rm برابر x می‌شد اما اضافه کردن دو مولتی پلکسر در این مسیر، سبب می‌شود که وقتی مقدار برابر ۱۵

بود، مقدار pc را به عنوان خروجی بدهد و همانطور که در waveform هم قابل مشاهده است، وقتی مقدار src1 برابر ۱۵ شده است، مقدار pc_in در Val_Rn قرار می‌گیرد.