Lab 8

Jiasen Zhou, Jon Johnston

March 27, 2019

1 Executive Summary

The purpose of this lab is to design and simulate the execute stage of the datapath. The execute stage is made up of the primary ALU for executing instructions, another ALU for branching, and control modules for their operation. This stage is responsible for performing the necessary arithmetic to execute instructions on data in registers and memory as well as branching. Mostly, this lab pertained to connecting modules created in earlier labs. After running the test for the module and comparing it to expected results, the lab proved to be successful.

2 Test Report

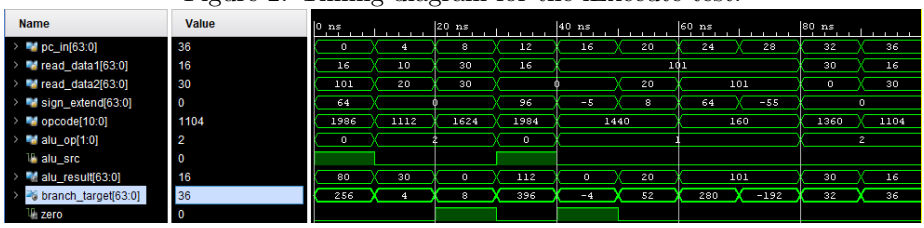
To verify operation of this module, this lab requires one test bench.

1. iExecute Test Bench

Figure 1: Expected Results of the iExecute test.

	Instruction 1	Instruction 2	Instruction 3	Instruction 4	Instruction 5	Instruction 6	Instruction 7	Instruction 8	Instruction 9	Instruction 10
Instruction	LDUR X9, [X22, #64]	ADD X10, X19, X9	SUB X11, X20, X10	STUR X11, [X22, #96]	CBZ X11, -5	CBZ X9, 8	B 64	B -55	ORR X9, X10, X21	AND X9, X22, X10
Machine Instruction (hex)	F84402C9	8B09026A	CB0A028B	F80602C8	B4FFFF6B	b4000109	14000040	17FFFFC9	AA150149	8A0A02C9
opcode (binary)	11111000010	10001011000	11001011000	11111000000	10110100	10110100	000101	000101	10101010000	10001010000
sign_extended_output (hex)	0000000000000040	0000000000000000	0000000000000000	0000000000000060	FFFFFFFFFFFFFFFB	0000000000000008	0000000000000040	FFFFFFFFFFFFFFC9	0000000000000000	0000000000000000
reg2_loc	0	0	0	1	1	1	0	0	0	0
branch	0	0	0	0	1	1	0	0	0	0
mem_read	1	0	0	0	0	0	0	0	0	0
mem_to_reg	1	0	0	0	0	0	0	0	0	0
alu_op	00	10	10	00	01	01	01	01	10	10
mem_write	0	0	0	1	0	0	0	0	0	0
alu_src	1	0	0	1	0	0	0	0	0	0
reg_write	1	1	1	0	0	0	0	0	1	1
uncondbranch	0	0	0	0	0	0	1	1	0	0
write_data (decimal)	20	30	0	X	X	X	X	X	30	16
read_data1 (decimal)	16	10	30	16	X	X	X	X	30	16
read_data2 (decimal)	X	20	30	0	0	20	X	X	0	30
branch_target	X	X	X	X	-20	52	28	256	-220	X
alu_result	80	30	0	112	0	20	X	X	30	16
zero	0	0	1	0	1	0	0	0	0	0

Figure 2: Timing diagram for the iExecute test.



3 Code Appendix

Listing 1: Verilog code for testing the iExecute module.

```
`include "definitions.vh"
module iExecute_test;
    reg [`WORD-1:0] pc_in, read_data1, read_data2, sign_extend;
    reg [10:0] opcode;
    reg [1:0] alu_op;
    reg alu_src;
    wire [`WORD-1:0] alu_result, branch_target;
    wire zero;

    iExecute execute_module(
        .pc_in(pc_in),
        .read_data1(read_data1),
        .read_data2(read_data2),
        .sign_extend(sign_extend),
        .opcode(opcode),
        .alu_op(alu_op),
        .alu_src(alu_src),
        .alu_result(alu_result),
```

```
        .branch_target(branch_target),
        .zero(zero));

initial
    begin
    // 1
    pc_in = 0;
    read_data1 = 16;
    read_data2 = 101;
    sign_extend = 64'h0000000000000040;
    opcode = 11'b11111000010;
    alu_op = 2'b00;
    alu_src = 1;
    #10;
    // 2
    pc_in = 4;
    read_data1 = 10;
    read_data2 = 20;
    sign_extend = 64'h0000000000000000;
    opcode = 11'b10001011000;
    alu_op = 2'b10;
    alu_src = 0;
    #10;
    // 3
    pc_in = 8;
    read_data1 = 30;
    read_data2 = 30;
    sign_extend = 64'h0000000000000000;
    opcode = 11'b11001011000;
    alu_op = 2'b10;
    alu_src = 0;
    #10;
    // 4
    pc_in = 12;
    read_data1 = 16;
    read_data2 = 0;
    sign_extend = 64'h0000000000000060;
    opcode = 11'b11111000000;
    alu_op = 2'b00;
    alu_src = 1;
    #10;
    // 5
    pc_in = 16;
    read_data1 = 101;
    read_data2 = 0;
    sign_extend = 64'hFFFFFFFFFFFFFFFB;
```

```
    opcode = 11'b10110100000;
    alu_op = 2'b01;
    alu_src = 0;
    #10;
    // 6
    pc_in = 20;
    read_data1 = 101;
    read_data2 = 20;
    sign_extend = 64'h0000000000000008;
    opcode = 11'b10110100000;
    alu_op = 2'b01;
    alu_src = 0;
    #10;
    // 7
    pc_in = 24;
    read_data1 = 101;
    read_data2 = 101;
    sign_extend = 64'h0000000000000040;
    opcode = 11'b00010100000;
    alu_op = 2'b01;
    alu_src = 0;
    #10;
    // 8
    pc_in = 28;
    read_data1 = 101;
    read_data2 = 101;
    sign_extend = 64'hFFFFFFFFFFFFFFC9;
    opcode = 11'b00010100000;
    alu_op = 2'b01;
    alu_src = 0;
    #10;
    // 9
    pc_in = 32;
    read_data1 = 30;
    read_data2 = 0;
    sign_extend = 64'h0000000000000000;
    opcode = 11'b10101010000;
    alu_op = 2'b10;
    alu_src = 0;
    #10;
    // 10
    pc_in = 36;
    read_data1 = 16;
    read_data2 = 30;
    sign_extend = 64'h0000000000000000;
    opcode = 11'b10001010000;
```

```
    alu_op = 2'b10;
    alu_src = 0;
    #10;
    $finish;
    end
endmodule
```

Listing 2: Verilog code for the iExecute module

```
`include "definitions.vh"
module iExecute(
    input [`WORD-1:0] pc_in,
    input [`WORD-1:0] read_data1,
    input [`WORD-1:0] read_data2,
    input [`WORD-1:0] sign_extend,
    input [10:0] opcode,
    input [1:0]alu_op,
    input alu_src,
    output [`WORD-1:0] alu_result,
    output zero,
    output [`WORD-1:0] branch_target
    );

    wire [`WORD-1:0] mux_out;
    wire [3:0] alu_ctrl_out;
    wire [`WORD-1:0] shift_result;

    mux MUX (
        .a_in(read_data2),
        .b_in(sign_extend),
        .control(alu_src),
        .mux_out(mux_out));

    ALU alu(
        .a_in(read_data1),
        .b_in(mux_out),
        .alu_control(alu_ctrl_out),
        .alu_result(alu_result),
        .zero_flag(zero)
        );

    alu_control alu_ctrl(
        .ALUOp(alu_op),
        .opcode(opcode),
        .ALU_control(alu_ctrl_out));
```

```
        //shift sign extend left by 2 = *4
    assign shift_result = sign_extend*4;

    adder pc_adder(
        .a_in(pc_in),
        .b_in(shift_result),
        .add_out(branch_target)
        );

endmodule
```