Lab 9 - iExecute and Datapath

Breana Leal and Matthew Carrano

March 25, 2018

1 Introduction

The goal of this lab is completing the iExecute stage and integrating the iExecute stage into the overall datapath. Once constructed, the outputs are verified through simulation and an expected results table.

2 Interface

The iExecute module has inputs pc_in, read_data1, read_data2, sign_extended_output, alu_op, opcode, and alu_src. It has outputs branch_target, alu_result, and zero. The next five modules are part of the iExecute module.

The inputs of the mux are read_data2 and sign_extended_output. The control signal is alu_src, this signal chooses which of the two above signals to output. The output is mux_out. The shifter module left shifts the bits of its input, sign_extended_out, and outputs shift_result. The inputs of the adder module are pc_in and shift_result. These two values are added and the result, and therefore output, is branch_target. The alu_control module inputs are alu_op and opcode. The output is control. The ALU module has inputs read_data1, mux_out, and control. The outputs are alu_result and zero. The final module is the ALU. This module has inputs control, read_data1, and mux_out. It's outputs are alu_result and zero.

3 Design

iExecute is one unit with five internal modules. The modules are shifter (alu_shifter), mux (alu_mux), adder (branch_adder),alu_control (ex_alu_con), and ALU (ex_alu). The following process applies to instructions with branch addressing. The iExecute uses the adder and shifter to left shift the sign extended address add the address with the current PC to provide the execute's branch target. Additionally, it uses the ALU control to determine the operations of the ALU. The ALU does arithmetic operations on its inputs and outputs the results as well as a flag which is 1 when the result is 0 and 0 otherwise. Overall, iExecute provides the current branch targeted address, a zero flag, and ALU operation results.

The datapath.v links the iExecute, iDecode, and iFetch modules. As described previously, the iDecode stage will process the outputted instruction from the Fetch stage. Similarly, iExecute engages the read_data1, read_data2, and iDecode's extended address target to output the current branch target address, ALU result, and zero flag.

4 Implementation

The following modules are part of the iExecute module.

The mux module is implemented using the ? operator. If the control signal is 0, mux_out is set to a_in. If the control signal is 1, mux_out is set to b_in. In the iExecute module, signals read_data2, sign_extended_output, alu_src, and mux_out, correspond to signals a_in, b_in, control, and mux_out of the mux module, respectively. In this way the control module from the iDecode stage can control which signal goes to the ALU module. The code can be found in Listing 8 on page 9.

The shifter module is implemented by left shifting the input (offset) by 2 using the << operator and assigning the result to "result". This is done inside an always@(*) block (sequential logic) and therefore a non-blocking assignment is used. In this way the result is set at any change to offset. In the iExecute module, signals sign_extended_output and shift_result, correspond to signals offset and result of the shifter module, respectively. The code can be found in Listing 3 on page 3.

The adder module is implemented using the + operator to add the inputs a_in and b_in. The result is assigned to add_out using a continuous assignment. In the iExecute module, signals pc_in, shift_result, and branch_target, correspond to signals a_in, b_in, and add_out of the adder module, respectively. In this configuration the adder outputs the next address of the program counter by adding the current pc with the left shifted sign extended address. The code can be found in Listing 2 on page 3.

The alu_control module was implemented using a case statement inside of an always@(*) block. The case statement is constructed using the alu_op signal as the switch variable and macros for the possible values of alu_op as the cases. Each case sets the control_bits to a specific 4-bit value using a non-blocking assignment. These values will control the operation of the ALU. The default case is the d-type control signal because this is a fast ALU operation. Because of the always block, the control_bits signal will be set with any change to alu_op or opcode. In the iExecute module the signals are named the same except control in corresponds to control_bits in the alu_control module. The code can be found in Listing 4 on page 4.

Like the alu_control module, the ALU module is implemented using a case statement inside of an always@(*) module. The case statement is constructed with control as the switch variable, and the control signals from the alu_control module as the cases. The result signal is set in each of the case blocks using a non-blocking assignment. If the control signal is signal for an add instruction, the result is the sum of data_1 and data_2. If the control signal is the signal for a conditional branch, the result is set to data_2. The rest of the cases can be seen in Listing 5 on page 5. Lastly, the flag output signal is set to the result of the Boolean logic statement, (result==0). This means that flag is 1 if the result is 0, and 0 if the result is not 0. With this module the datapath can do the necessary mathematical operations on all of its signals. The code can be found in Listing 5 on page 5.

The iExecute module was implemented simply by instantiating each of the five modules above and connecting them as described above. This can be seen further in Listing 6 on page 6. This module completes the execute stage of the processor.

Listing 1: Verilog code for implementing the mux.

```
`include "definitions.vh"

module mux#(
    parameter SIZE=8)(
    input [SIZE-1:0] a_in,
    input [SIZE-1:0] b_in,
    input control,
    output [SIZE-1:0] mux_out
    );
    assign mux_out = control?b_in:a_in;
endmodule
```

Listing 2: Verilog code for implementing the adder.

```
`timescale 1ns / 1ps
`include "definitions.vh"

module adder(
    input [`WORD-1:0] a_in,
    input [`WORD-1:0] b_in,
    output [`WORD-1:0] add_out
    );

    assign add_out = a_in + b_in;

endmodule
```

Listing 3: Verilog code for implementing the shifter.

```
`include "definitions.vh"


module shifter(
    input [`WORD-1:0] offset,
    output reg [`WORD-1:0] result
    );

   always @(*) begin
      result <= offset << 2;
   end

endmodule
```

Listing 4: Verilog code for implementing the ALU control.

```
`include "definitions.vh"

module alu_control(
    input [1:0] alu_op,
    input [10:0] opcode,
    output reg [3:0] control_bits
    );

    always @(*) begin
        case(alu_op)

        default: begin
            control_bits <= 4'b0010;
        end

        `ALUOp_DTYPE: begin
            control_bits <= 4'b0010;
        end

        `ALUOp_RTYPE: begin
            control_bits[3] <= 0;
            control_bits[2] <= opcode[9];
            control_bits[1] <= opcode[3];
            control_bits[0] <= opcode[8];
        end

        `ALUOp_BRANCH: begin
            control_bits <= 4'b0111;
        end

        endcase

    end
```

```


endmodule
```

Listing 5: Verilog code for implementing the ALU.

```
`include "definitions.vh"

module ALU(

    input  [`WORD-1:0] data_1, data_2,
    input  [3:0] control,
    output reg [`WORD-1:0] result,
    output reg flag
    );

    always @(*)begin
            case(control)

            default: begin
                result <= data_1 & data_2;
            end

            `alu_add: begin
                result <= data_1 + data_2;
            end

            `alu_cbz: begin
                result <= data_2;
            end

            `alu_and: begin
                 result <= data_1 & data_2;
            end

            `alu_orr: begin
                 result <= data_1 | data_2;
            end

            `alu_sub: begin
                 result <= data_1 - data_2;
            end

            endcase

        flag <= (result == 0);
    end

endmodule
```

Listing 6: Verilog code for implementing 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_extended_output,
    output [`WORD-1:0] branch_target,
    output [`WORD-1:0] alu_result,
    output zero,
    input [1:0] alu_op,
    input [10:0] opcode,
    input alu_src
    );

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

mux#(`WORD) alu_mux(
    .a_in(read_data2),
    .b_in(sign_extended_output),
    .control(alu_src),
    .mux_out(mux_out)
    );

shifter alu_shifter(
    .offset(sign_extended_output),
    .result(shift_result)
    );

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

alu_control ex_alu_con(
    .alu_op(alu_op),
    .opcode(opcode),
    .control_bits(control)
    );

ALU ex_alu(
    .data_1(read_data1),
    .data_2(mux_out),
    .control(control),
    .result(alu_result),
    .flag(zero)
    );

endmodule
```

5 Test Bench Design

The iExecute test bench creates regs for all inputs and wires for all outputs. Zero and alu_src are 1 bit, alu_op is 2 bits, opcode is 11 bits, and all other signals are 64 bits. An iExecute module is instantiated. Values are set in the initial section every cycle (10ns). In the first cycle a value is set for all inputs.

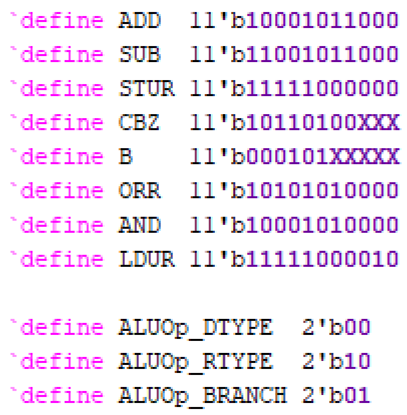
The iExecute test bench underwent 14 operations each requiring either subtraction, addition, bit-wise OR, load, store, CBZ comparison, or branching operations. Before testing, our definitions included each operation as a separate macro for its specific instruction and instruction type. Each instruction has 11 bits, CBZ and B have non-applicable Xs for unnecessary bits. The three defined types, R-type, D-type, and Branch are two bit macros. Figure 1 demonstrates the header changes. The initial values for read_data1 and read_2 are 10 and 15. They are redefined after the ninth, twelfth, and thirteenth cycle to 15 and 15, 15 and 0, 15 and 15 respectively (since there is no write data signal). The connecting wires are associated with its three external pieces, branch target, zero, and ALU result.

Listing 7: Verilog code for testing the Execute stage.

```
`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_extended_output,
    output [`WORD-1:0] branch_target,
    output [`WORD-1:0] alu_result,
    output zero,
    input [1:0] alu_op,
    input [10:0] opcode,
    input alu_src
    );

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

mux#(`WORD) alu_mux(
    .a_in(read_data2),
    .b_in(sign_extended_output),
    .control(alu_src),
    .mux_out(mux_out)
    );

shifter alu_shifter(
    .offset(sign_extended_output),
    .result(shift_result)
    );

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

alu_control ex_alu_con(
    .alu_op(alu_op),
    .opcode(opcode),
    .control_bits(control)
    );

ALU ex_alu(
    .data_1(read_data1),
    .data_2(mux_out),
    .control(control),
    .result(alu_result),
    .flag(zero)
    );

endmodule
```

Datapath is the top hierarchy testbench for debugging and connecting stages together. iExecute is an additional stage to bridge towards a single completed ARM datapath. With the iExecute module implemented it now only needs

Figure 1: Updated Definitions Header.



```
`define ADD  11'b10001011000
`define SUB  11'b11001011000
`define STUR 11'b11111000000
`define CBZ  11'b10110100XXX
`define B    11'b000101XXXXX
`define ORR  11'b10101010000
`define AND  11'b10001010000
`define LDUR 11'b11111000010

`define ALUOp_DTYPE  2'b00
`define ALUOp_RTYPE  2'b10
`define ALUOp_BRANCH 2'b01
```

initial values to start the program counter, as well as write_data values to store alu results in the correct registers.

Listing 8: Verilog code for testing the Updated Datapath.

```
`include "definitions.vh"

module datapath;

   wire clk, read_clk, write_clk;
   reg rst;
   reg pc_src;
   reg [`WORD-1:0] branch_target;
   reg [`WORD-1:0] write_data;
   wire [`WORD-1:0] read_data1;
   wire [`WORD-1:0] read_data2;
   wire [`WORD-1:0] incremented_pc;
   wire [`WORD-1:0] cur_pc;
   wire uncondbranch;
   wire branch;
   wire mem_read;
   wire mem_to_reg;
   wire [1:0] alu_op;
   wire mem_write;
   wire alu_src;
   wire [`WORD-1:0] ext_addr;
   wire [`INSTR_LEN-1:0] instruct;
   wire [`WORD-1:0] execute_branch_target;
   wire [`WORD-1:0] alu_result;
   wire zero;
```

```
oscillator clk_gen(clk);

iFetch fetch(
.clk(clk),
.reset(rst),
.pc_src(pc_src),
.branch_target(branch_target),
.incremented_pc(incremented_pc),
.instruction(instruct), //The only wire that connects iFetch and iDecode
.cur_pc(cur_pc)
);

iDecode decode(
.clk(clk_plus_2),
.read_clk(clk_plus_3),
.write_clk(clk_plus_7),
.instr(instruct),
.write_data(write_data),
.read_data1(read_data1),
.read_data2(read_data2),
.uncondbranch(uncondbranch),
.branch(branch),
.mem_read(mem_read),
.mem_to_reg(mem_to_reg),
.alu_op(alu_op),
.mem_write(mem_write),
.alu_src(alu_src),
.ext_addr(ext_addr)
);

iExecute execute(
  .pc_in(cur_pc),
  .read_data1(read_data1),
  .read_data2(read_data2),
  .sign_extended_output(ext_addr),
  .branch_target(execute_branch_target),
  .alu_result(alu_result),
  .zero(zero),
  .alu_op(alu_op),
  .opcode(instruct[31:21]),
  .alu_src(alu_src)
 );

delay clk_delay_1(
    .a(clk),
        .a_delayed(clk_plus_1)
    );

    delay clk_delay_2(
        .a(clk_plus_1),
        .a_delayed(clk_plus_2)
    );

    delay clk_delay_3(
        .a(clk_plus_2),
        .a_delayed(clk_plus_3)
    );

        delay clk_delay_4(
        .a(clk_plus_3),
        .a_delayed(clk_plus_4)
    );

    delay clk_delay_5(
        .a(clk_plus_4),
        .a_delayed(clk_plus_5)
    );

    delay clk_delay_6(
        .a(clk_plus_5),
        .a_delayed(clk_plus_6)
    );

        delay clk_delay_7(
        .a(clk_plus_6),
        .a_delayed(clk_plus_7)
    );

initial
        begin
            // wait for the rising clock edge
            #(`CYCLE/2);
            rst <= 1;
            pc_src <= 0;
            branch_target <= 0;
            #(`CYCLE/2);
            rst <= 0;
            write_data <= `WORD'd20;
            #(`CYCLE);
            write_data <= `WORD'd30; // Addition result
            #(`CYCLE);
            write_data <= `WORD'd0; // Subtraction Result
            #(`CYCLE);
            write_data <= `WORD'd0; // N/A
            #(`CYCLE);
            write_data <= `WORD'd0; // N/A
            #(`CYCLE);
            write_data <= `WORD'd0; // N/A
            #(`CYCLE);
            write_data <= `WORD'd0; // N/A
            #(`CYCLE);
            write_data <= `WORD'd0; // N/A
            #(`CYCLE);
            write_data <= `WORD'd30; // ORR result
            #(`CYCLE);
            write_data <= `WORD'd14; // Writing

        end

endmodule
```

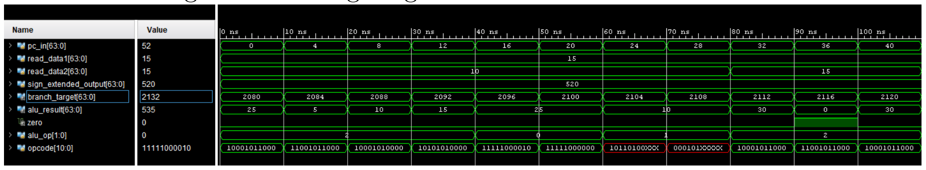
6 Simulation

The timing diagrams in Figure 2 and 3 verify that iExecute stage is functioning properly within the datapath and has outputs corresponding to our expected results table Figure 4.

7 Conclusions

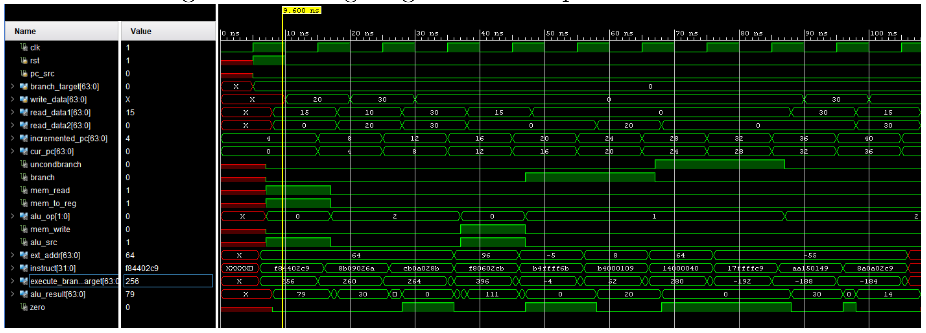
The iExecute module and datapath testbench were created and updated. iExecute is our third stage that provides an updated brancg target, ALU results, and zero flag. Its input are the iDecode's wire connecting outputs. The iFetch, iDecode, and iExecute are linked in the datapath testbench. The testbench demonstrates the ARM's increments fetched at proper times and providing correctly executed results.

Figure 2: Timing diagram for iExecute module test.



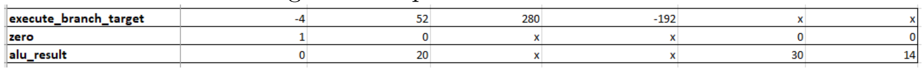
Name	Value
pc_in[63:0]	52
read_data1[63:0]	15
read_data2[63:0]	15
sign_extended_output[63:0]	520
branch_target[63:0]	2132
alu_result[63:0]	535
zero	0
alu_op[1:0]	0
opcode[10:0]	11111000010

Figure 3: Timing diagram for datapath module test.



Name	Value
clk	1
rst	1
pc_src	0
branch_target[63:0]	0
write_data[63:0]	X
read_data1[63:0]	15
read_data2[63:0]	0
incremented_pc[63:0]	4
cur_pc[63:0]	0
uncondbranch	0
branch	0
mem_read	1
mem_to_reg	1
alu_op[1:0]	0
mem_write	0
alu_src	1
ext_addr[63:0]	64
instruct[31:0]	f84402c9
execute_bran...arget[63:0	256
alu_result[63:0]	79
zero	0

Figure 4: Expected Results Table.



execute_branch_target	-4	52	280	-192	x	x
zero	1	0	x	x	0	0
alu_result	0	20	x	x	30	14