Final Project

Brandon Howell
Mauro Alvizo

Group name: MB
Class: CS 3339 - Computer Architecture

April 22, 2024

1 Introduction

For the final step of the project, we combined all of the ALU functions developed during steps 1 and 2 into one control circuit. It processes inputs by taking two 4-bit inputs and a 3-bit opcode and selects the appropriate ALU module through a multiplexer based on the provided opcode. We then conducted circuit tests and generated simulation waveforms to ensure our circuit performs as expected.

2 Verilog Code and Test Benches

2.1 Verilog Code

– Binary logic functions: The following modules include the Or, And, and Xor gates

```
// 4-bit OR gate
module Or(
    input [3:0] A,
    input [3:0] B,
    output [3:0] Y
);
assign Y = A | B;
endmodule;

// 4-bit AND gate
module And(
    input [3:0] A,
    input [3:0] B,
    output [3:0] Y
);
assign Y = A & B;
endmodule

// 4-bit XOR gate
module Xor(
    input [3:0] A,
    input [3:0] B,
    output [3:0] Y
);
assign Y = A ^ B;
endmodule
```

– ALU arithmetic operations: The following modules include the addition, subtraction, multiplication and division operations.

```
    // 4-bit Adder with carry-in and carry-out
    module Adder(
        input [3:0] A,
        input [3:0] B,
        input Cin,
        output [3:0] Sum,
        output Cout
    );
        wire [4:0] full_sum;
        assign full_sum = A + B + Cin;
        assign Sum = full_sum[3:0];
        assign Cout = full_sum[4];
    endmodule

    // 4-bit Subtractor with Borrow-in and Borrow-out
        module Subtractor(
        input [3:0] A,
        input [3:0] B,
        output [3:0] Diff,
        output Borrow
    );
        assign Diff = A - B;
        assign Borrow = A < B;
    endmodule

    // 4-bit Multiplier
    module Multiplier(
        input [3:0] A,
        input [3:0] B,
        output [7:0] Product
    );
        assign Product = A * B;
    endmodule


    // 4-bit Divider
    module Divider(
        input enable,
        input [3:0] A,
        input [3:0] B,
        output reg [3:0] Quotient,
        output reg [3:0] Remainder
    );
        always @* begin
            if (enable && B != 0) begin
                Quotient = A / B;
                Remainder = A % B;
            end else begin
                Quotient = 0;
                Remainder = 0;
            end
        end
    endmodule
```

– Shifter module: The following module implements bitwise left and right shifts on a 4-bit input based on control signals.

```
// Module for a 4-bit shifter
module Shifter(
    input wire [3:0] A,
    input wire [3:0] B,
    output reg [3:0] Y
);
    integer i;
    reg [3:0] temp;
    wire shift_dir;
    wire fill_bit = 0;

    assign shift_dir = B[3];

    always @* begin
        temp = A;
        for (i = 0; i < B[2:0]; i = i + 1) begin
            if (shift_dir)
                temp = {temp[2:0], fill_bit};
            else
                temp = {fill_bit, temp[3:1]};
        end
        Y = temp;
    end
endmodule
```

– ALU module: The following module contains the circuits for arithmetic and logical operations, outputting results based on inputs A, B, and opcode.

```
module ALU(
    input [3:0] A,
    input [3:0] B,
    input [2:0] opcode,
    output reg [3:0] result,
    output reg [3:0] remainder,
    output zeroFlag,
    output overflowFlag
);
    wire [3:0] add_result, sub_result, and_result, or_result,
        xor_result, shifter_y, div_result, div_remainder;
    wire [7:0] mul_result;
    wire add_cout, sub_borrow;
    wire enable_adder, enable_subtractor, enable_multiplier,
        enable_and, enable_or, enable_xor, enable_shifter,
        enable_divider;

    ALU_Control control(
        .opcode(opcode),
        .enable_adder(enable_adder),
        .enable_subtractor(enable_subtractor),
        .enable_multiplier(enable_multiplier),
        .enable_divider(enable_divider),
        .enable_and(enable_and),
        .enable_or(enable_or),
        .enable_xor(enable_xor),
        .enable_shifter(enable_shifter)
    );

    Adder add(.A(A), .B(B), .Cin(1'b0), .Sum(add_result),
        .Cout(add_cout));
    Subtractor sub(.A(A), .B(B), .Diff(sub_result),
        .Borrow(sub_borrow));
    Multiplier mult(.A(A), .B(B), .Product(mul_result));
    Divider div(.enable(enable_divider), .A(A), .B(B),
        .Quotient(div_result), .Remainder(div_remainder));
    And and_gate(.A(A), .B(B), .Y(and_result));
    Or or_gate(.A(A), .B(B), .Y(or_result));
    Xor xor_gate(.A(A), .B(B), .Y(xor_result));
    Shifter shifter(.A(A), .B(B), .Y(shifter_y));

    always @(*) begin
        case (opcode)
            3'b000: result = enable_adder ? add_result :
                4'b0000;
            3'b001: result = enable_subtractor ? sub_result :
                4'b0000;
            3'b010: result = enable_multiplier ?
                mul_result[3:0] :4'b0000;
            3'b100: result = enable_and ? and_result :4'b0000;
            3'b101: result = enable_or ? or_result :4'b0000;
            3'b110: result = enable_xor ? xor_result :4'b0000;
            3'b011: result = enable_shifter ? shifter_y :
                4'b0000;
            3'b111: begin
               result = enable_divider ? div_result: 4'b0000;
               remainder = enable_divider? div_remainder:
                   4'b0000;
            end
            default: begin
               result = 4'b0000;
               remainder = 4'b0000;
            end
        endcase
    end

    assign zeroFlag = (result == 4'b0000);
    assign overflowFlag = (opcode == 3'b000 && add_cout) ||
        (opcode == 3'b001 && sub_borrow);

endmodule
```

– ALU control module: The following module decodes the opcode to activate the specific operation within the ALU.

```
    // Control module for the ALU
    module ALU_Control(
        input wire [2:0] opcode,
        output wire enable_adder,
        output wire enable_subtractor,
        output wire enable_multiplier,
        output wire enable_divider,
        output wire enable_and,
        output wire enable_or,
        output wire enable_xor,
        output wire enable_shifter
    );

        assign enable_adder = (opcode == 3'b000);
        assign enable_subtractor = (opcode == 3'b001);
        assign enable_multiplier = (opcode == 3'b010);
        assign enable_divider = (opcode == 3'b111);
        assign enable_and = (opcode == 3'b100);
        assign enable_or = (opcode == 3'b101);
        assign enable_xor = (opcode == 3'b110);
        assign enable_shifter = (opcode == 3'b011);

    endmodule
```

2.2 Test Benches

In order to maintain brevity and clear visualizations for our test benches later in the report, we separated our modules into 3 test benches: arithmetic operations, binary logic functions, and the shifter function.

– ALU Arithmetic Operations Test Bench

```
'timescale 1ns / 1ps

module operations_testbench;

    reg [3:0] A, B;
    reg [2:0] opcode;
    wire [3:0] result;
    wire [3:0] remainder;
    wire zeroFlag, overflowFlag;

    // Instantiate the ALU
    ALU alu (
        .A(A),
        .B(B),
        .opcode(opcode),
        .result(result),
        .remainder(remainder),
        .zeroFlag(zeroFlag),
        .overflowFlag(overflowFlag)
    );

    initial begin
        $dumpfile("operations_testbench.vcd");
        $dumpvars(0, operations_testbench);
        A = 0; B = 0; opcode = 3'bxxx;
        // Addition
        A = 4'b1001; B = 4'b0110; opcode = 3'b000; #10;
        A = 4'b1010; B = 4'b0011; opcode = 3'b000; #10;
        A = 4'b0011; B = 4'b0010; opcode = 3'b000; #10;
        A = 4'b1000; B = 4'b0010; opcode = 3'b000; #10;

        // Subtraction
        A = 4'b1001; B = 4'b0110; opcode = 3'b001; #10;
        A = 4'b1010; B = 4'b0011; opcode = 3'b001; #10;
        A = 4'b0011; B = 4'b0010; opcode = 3'b001; #10;
        A = 4'b1000; B = 4'b0010; opcode = 3'b001; #10;


        // Multiplication
        A = 4'b1001; B = 4'b0110; opcode = 3'b010; #10;
        A = 4'b1010; B = 4'b0011; opcode = 3'b010; #10;
        A = 4'b0011; B = 4'b0010; opcode = 3'b010; #10;
        A = 4'b1000; B = 4'b0010; opcode = 3'b010; #10;

        // Division
        A = 4'b1001; B = 4'b0110; opcode = 3'b111; #10;
        A = 4'b1010; B = 4'b0011; opcode = 3'b111; #10;
        A = 4'b0011; B = 4'b0010; opcode = 3'b111; #10;
        A = 4'b1000; B = 4'b0010; opcode = 3'b111; #10;

        $finish;
    end
endmodule
```

– Binary Logic Function Test Bench

```
`timescale 1ns / 1ps

module binary_testbench;
    reg [3:0] A, B;
    reg[2:0] opcode;
    wire [3:0] result;
    wire zeroFlag, overflowFlag;

    ALU alu (
        .A(A),
        .B(B),
        .opcode(opcode),
        .result(result),
        .zeroFlag(zeroFlag),
        .overflowFlag(overflowFlag)
    );

    initial begin
        $dumpfile("binary_testbench.vcd");
        $dumpvars(0, binary_testbench);

        // AND
        A = 4'b1101; B = 4'b0011; opcode = 3'b100; #10;
        A = 4'b0011; B = 4'b0101; opcode = 3'b100; #10;
        A = 4'b1101; B = 4'b1101; opcode = 3'b100; #10;

        // OR
        A = 4'b1101; B = 4'b0011; opcode = 3'b101; #10;
        A = 4'b0011; B = 4'b0101; opcode = 3'b101; #10;
        A = 4'b1101; B = 4'b1101; opcode = 3'b101; #10;

        // XOR
        A = 4'b1101; B = 4'b0011; opcode = 3'b110; #10;
        A = 4'b0011; B = 4'b0101; opcode = 3'b110; #10;
        A = 4'b1101; B = 4'b1101; opcode = 3'b110; #10;

        $finish;
    end
endmodule
```

– Shift Function Test Bench

```
`timescale 1ns / 1ps

module shifter_testbench;
    reg [3:0] A, B;
    reg [2:0] opcode;
    wire [3:0] result;
    wire zeroFlag, overflowFlag;

    ALU alu (
        .A(A),
        .B(B),
        .opcode(opcode),
        .result(result),
        .zeroFlag(zeroFlag),
        .overflowFlag(overflowFlag)
    );

    initial begin
        $dumpfile("shifter_testbench.vcd");
        $dumpvars(0, shifter_testbench);

        // Test 1: Left shift A=4'b1010 by 1
        A = 4'b1010; B = 4'b1001; opcode = 3'b011; #10;

        // Test 2: Right shift A=4'b1010 by 2 (B=4'b0010)
        A = 4'b1010; B = 4'b0010; opcode = 3'b011; #10;

        // Test 3: Left shift A=4'b1101 by 3 (B=4'b1111)
        A = 4'b1101; B = 4'b1111; opcode = 3'b011; #10;

        // Test 4: Right shift A=4'b1001 by 1 (B=4'b0001)
        A = 4'b1001; B = 4'b0001; opcode = 3'b011; #10;

        $finish;
    end
endmodule
```

3 Results and Explanation

3.1 ALU Arithmetic Operations

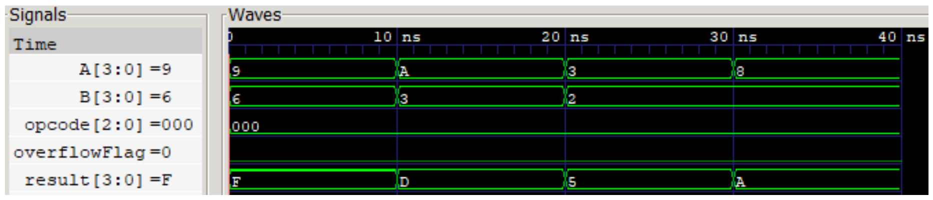

Fig. 1: Addition Waveform

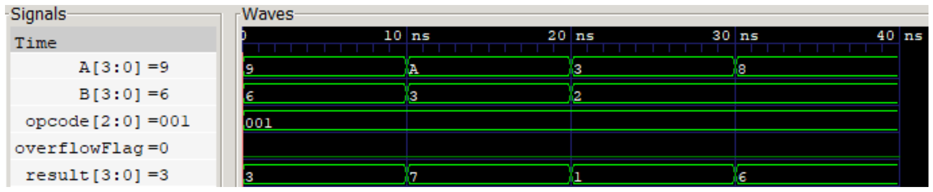

Fig. 2: Subtraction Waveform

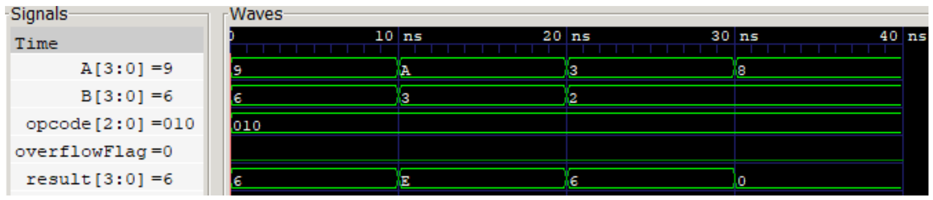

Fig. 3: Multiplication Waveform: due to the 4-bit output register in the ALU, only the least significant 4 bits of the 8-bit product are presented as the output, truncating the higher bits.

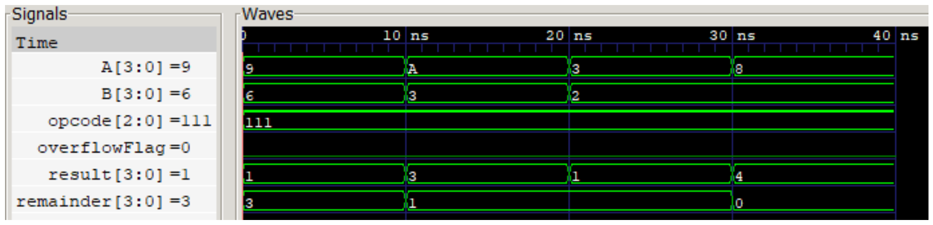

Fig. 4: Division Waveform

3.2 Binary Logic Functions

The result is a 4-bit value where each bit is the logical AND of the corresponding bits in the operands, outputting '1' only where both bits are '1'.

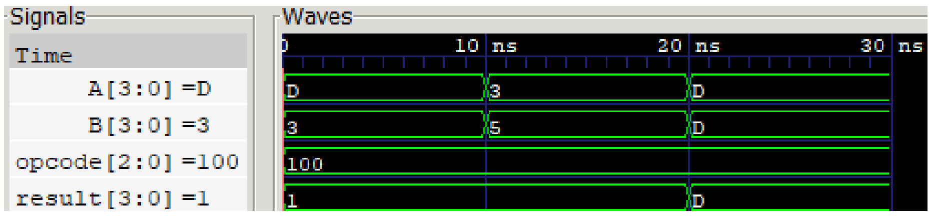

Fig. 5: And Gate: outputs only common set bits.

The result is a 4-bit value where each bit is the logical OR of the corresponding bits in the operands, set to '1' if either or both corresponding input bits are '1'.

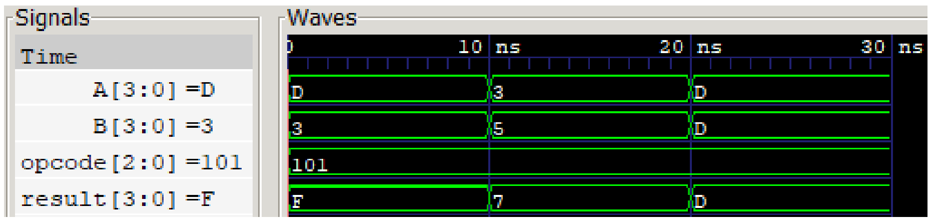

Fig. 6: Or Gate: merges set bits from either input

The result is a 4-bit value where each bit is the logical XOR of the corresponding bits in the operands, set to '1' only if the input bits from A and B differ.

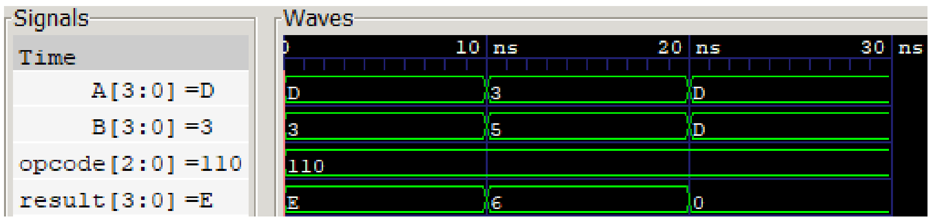

Fig. 7: Xor Gate: outputs '1' for differing bits from input

3.3 Shift Function

The shifter uses B[3] to determine the shift direction and B[2:0] to determine the number of positions A is shifted, modifying A's bit pattern based on B's bit pattern.

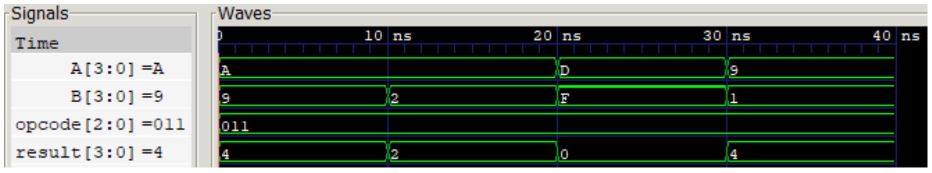

Fig. 8: Shifter Waveform

4 Conclusion

In summary, we finished a control circuit that handles binary, arithmetic, and shift functions depending on our testbench values. We then used GTKWaves to visually represent our results and how our control unit worked in real time. Overall we learned how the inner workings of a processor uses basic functions such as ADD, SUB, XOR etc. to create more complex algorithms and what those inner workings would look like in real time.