BIRZEIT UNIVERSITY

Faculty of Engineering & Technology
Department of Electrical and Computer Engineering

ENC3310 – Advanced Digital Systems Design

Project Report

Prepared by:

Name: Aya Dahbour 1201738 ID #: 1201738

Instructor: Dr. Abdellatif Abu-Issa

Section: 1

Date: January-2024

Brief Introduction and Background

In this project, we will build a simple part of a microprocessor as shown below in Figure 1, by building two main blocks and then connecting them to get the system design, using Aldec Active-HDL student edition to write Verilog codes, simulate the system then check the output. In the first part, we will implement the Arithmetic Logic Unit (ALU) and the register file, and then in part two we will connect them and run a simple machine code program on them. machine code program on them.

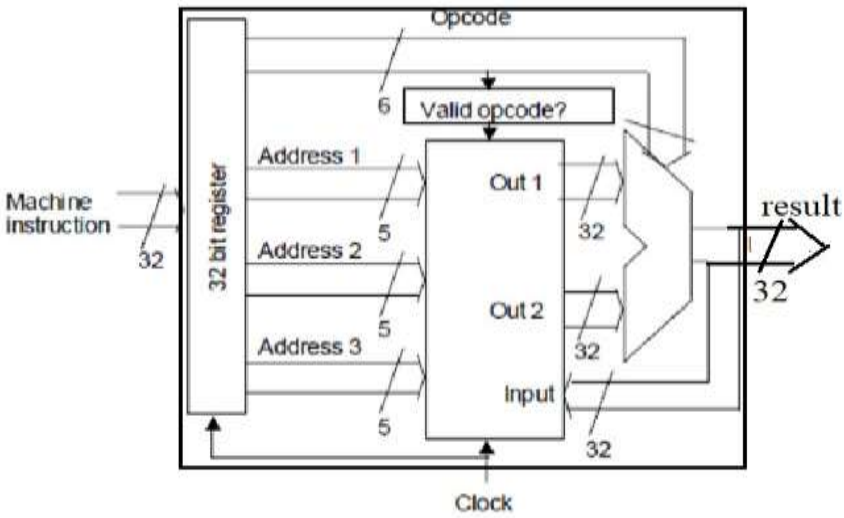

Figure 1: Simple part of a microprocessor

The ALU:

The basic component in the system is the ALU, that performs all the math and logic operations. With its 32-bit architecture, it takes in two numbers, crunches them according to the opcode, and spits out the result as shown in Figure 2, all while keeping an eye out for any calculation that goes overboard, known as an overflow.

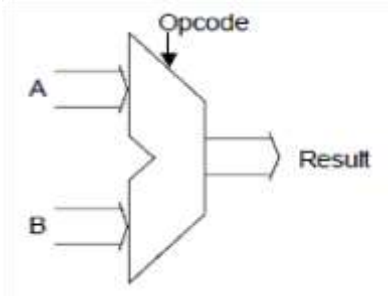

Figure 2: ALU module

I will use these opcodes according to the last digit of my ID, which is 8, to represent every process represented by the below:

- a + b = 1
- a - b = 6
- |a| = 13
- -a = 8
- max (a, b) = 7
- min (a, b) = 4
- avg(a,b) = 11
- not a = 15
- a or b = 3
- a and b = 5
- a xor b = 2

The Register File:

The register file is akin to a versatile cabinet within a microprocessor, where snippets of data are stashed momentarily for easy access. It's composed of multiple slots, each holding a 32-bit value, that have 3 addresses, two of them are for reading from the memory using the two outputs, and the third is for writing in it by the input. As follows in Figure 3.

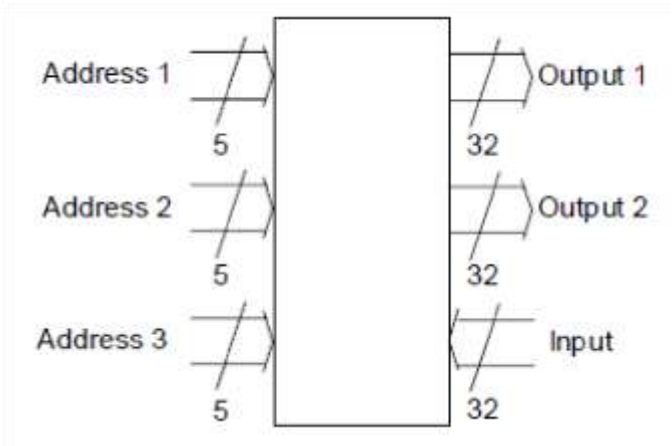

Figure 3: The register file module

The initial values stored in the register file are the values for number 3 determined by the following:

0	0	17	14102
1	12996	18	13200
2	11490	19	3264
3	7070	20	2368
4	6026	21	15846
5	3322	22	11710
6	10344	23	14736
7	6734	24	5338
8	15834	25	5544
9	15314	26	1852
10	6000	27	3898
11	12196	28	16252
12	11290	29	1048
13	13350	30	5642
14	2086	31	0
15	6734		
16	7430		

Figure 4: Initial values in the register file

Design Philosophy

ALU Design

In developing the ALU, I initially focused on writing and fine-tuning the Verilog code for the ALU module as shown in Figure 5. This involved meticulously ensuring the module was capable of handling a diverse array of arithmetic and logical operations, each identified by a unique opcode.

```
//Advance digital project - Aya Dahbour 1201738

`timescale 1ns / 1ps

module alu(
    input [5:0] opcode,
    input signed [31:0] a, b,
    output reg signed [31:0] result
);

    always @(*) begin
        case (opcode)
            6'b000001: result = a + b; // a + b //1
            6'b000110: result = a - b; // a - b //6
            6'b001101: result = a[31] ? -a : a; // |a| //13
            6'b001000: result = -a; // -a //8
            6'b000111: result = (a > b) ? a : b; // max(a, b) //7
            6'b000100: result = (a < b) ? a : b; // min(a, b) //4
            6'b001011: result = (a + b) >> 1; // avg(a, b) //11
            6'b001111: result = ~a; // not a //15
            6'b000011: result = a | b; // a or b //3
            6'b000101: result = a & b; // a and b //5
            6'b000010: result = a ^ b; // a xor b //2
            default: result = 32'd0;
        endcase
    end
endmodule
```

Figure 5: ALU module code

Then before proceeding to the next stage, determine the source of each error and resolve it. And to ensure that the code works correctly and to rigorously validate each function of the ALU. I developed a detailed test bench and then ran it to have a successful output as shown in Figure 6.

PS: The first test values is a = 15 b = 10, and the second test values is a = 20 b = 10.

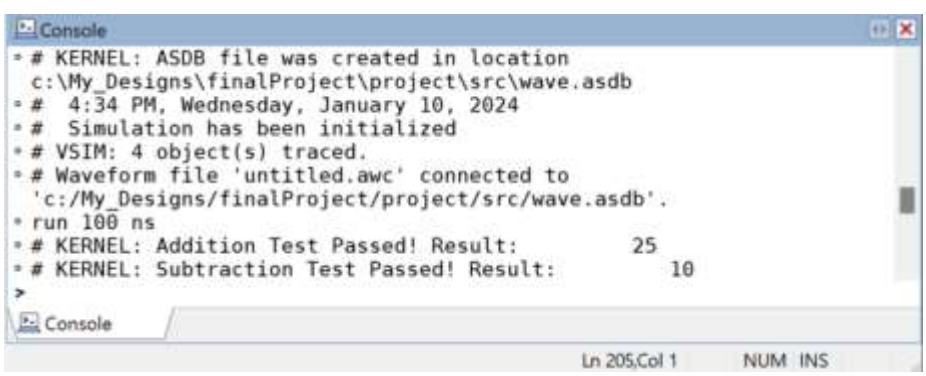

Figure 6: Test Bench output for ALU

The Register File Design

In this stage, I began by designing and implementing the Register File module in Verilog as shown below in Figure 7. The primary function of this module was to create a bank of registers, each capable of storing a 32-bit value. Either to read or write, depending on the address that calls for it.

```verilog
module reg_file(
    input clk,
    input valid_opcode,
    input [4:0] addr1, addr2, addr3,
    input [31:0] in,
    output [31:0] out1, out2
    );

    reg [31:0] registers [31:0];

    // Initialize the register file with the values
    initial begin
        registers[0] = 30;
        registers[1] = 32'd12996;
        registers[2] = 32'd11490;
        registers[3] = 32'd7070;
        registers[4] = 32'd6026;
        registers[5] = 32'd3322;
        registers[6] = 32'd10344;
        registers[7] = 32'd6734;
        registers[8] = 32'd15834;
        registers[9] = 32'd15314;
        registers[10] = 32'd6000;
        registers[11] = 32'd12196;
        registers[12] = 32'd11290;
        registers[13] = 32'd13350;
        registers[14] = 32'd2086;
        registers[15] = 32'd6734;
        registers[16] = 32'd7430;
        registers[17] = 32'd14102;
        registers[18] = 32'd13200;
        registers[19] = 32'd3264;
        registers[20] = 32'd2368;
        registers[21] = 32'd15846;
        registers[22] = 32'd11710;
        registers[23] = 32'd14736;
        registers[24] = 32'd5338;
        registers[25] = 32'd5544;
        registers[26] = 32'd1852;
        registers[27] = 32'd3898;
        registers[28] = 32'd16252;
        registers[29] = 32'd1048;
        registers[30] = 32'd5642;
        registers[31] = 32'd0;
    end

    //Read operations
    assign out1 = registers[addr1];
    assign out2 = registers[addr2];

    //Write operation
    always @(posedge clk) begin
        if (valid_opcode) begin
            registers[addr3] <= in;
        end
    end
endmodule
```

Figure 7: The register file module code

To check its functionality, I developed the test bench "reg_file_tb" to simulate numerous scenarios, such as writing data to specific registers and subsequently reading data from them. To ensure that the Register File correctly stored and retrieved data as intended. Also, this test was successful, where the observed outcomes matched the expected results as shown in its output in Figure 8.

```
Console
# 6:46 PM, Wednesday, January 10, 2024
# Simulation has been initialized
run 100 ns
# KERNEL: At 10000, it's a reading from Address 1. 12996 expected,
Got:     12996
# KERNEL: Performed write to Address 2 at 30000
# KERNEL: Reading from Address 2 at 40000 after write. Expected:
12345, Got:      12345
# KERNEL: Reading from Address 15 at 50000. Expected: 6734, Got:
    6734
>
```

Figure 8: The register file test bench output

The System Design

In the final stage, I meticulously constructed the core of this microprocessor system by seamlessly integrating predefined functions. Central to its operation is the handling of a singular 32-bit instruction input. This instruction is methodically dissected and allocated, with each segmented part corresponding to specific variables within the system. The outcome of this intricate process is a calculated result, derived from executing one of the approved operations as indicated by the opcodes. This is contingent upon the Opcode's validity and its adherence to the predefined group within the Arithmetic Logic Unit (ALU). The entire operation hinges on two key numbers, which are securely stored and retrieved from designated addresses in the register file.

This can be observed in the images of the code below.

```
module mp_top (clk, instruction, result);

    input clk;
    input [31:0] instruction;
    output reg [31:0] result;

    // Internal signals
    wire [5:0] opcode;
    wire [4:0] addr1, addr2, addr3;
    reg valid_opcode;
    wire [31:0] out1, out2;


    // Parse the instruction
    assign opcode = instruction[5:0];
    assign addr1 = instruction[10:6];
    assign addr2 = instruction[15:11];
    assign addr3 = instruction[20:16];
      always @(*) begin
        valid_opcode = (opcode == 6'd1)  || // a + b
                       (opcode == 6'd6)  || // a - b
                       (opcode == 6'd13) || // |a|
                       (opcode == 6'd8)  || // -a
                       (opcode == 6'd7)  || // max(a, b)
                       (opcode == 6'd4)  || // min(a, b)
                       (opcode == 6'd11) || // avg(a, b)
                       (opcode == 6'd15) || // not a
                       (opcode == 6'd3)  || // a or b
                       (opcode == 6'd5)  || // a and b
                       (opcode == 6'd2);    // a xor b
    end

    //Sample of the Register File based on valid_opcode
    reg_file reg_file_instance (
        .clk(clk),
        .valid_opcode(valid_opcode),
        .addr1(addr1),
        .addr2(addr2),
        .addr3(addr3),
        .in(valid_opcode ? result : 32'd0), //Don't write if opcode is not valid
        .out1(out1),
        .out2(out2)
    );
    //Sample of the ALU based on valid_opcode
    alu alu_instance (
        .opcode(valid_opcode ? opcode : 6'd0), //Disable ALU if opcode is not valid
        .a(out1),
        .b(out2),
        .result(result)
    );


endmodule
```

Figure 9: The system code

Then to verify the accuracy and effectiveness of the microprocessor's design, I developed the following test bench. This concise yet comprehensive part rigorously tests the system's functionality, ensuring seamless integration and reliable performance. An array of ten instructions, each testing a different mathematical operation, is deployed. These instructions, beginning with the Opcode followed by register numbers, are paired with an array of expected results. The system's success in each test is determined by comparing the actual output with the expected values, ensuring precise and reliable operation.

```
//System Test bench
module mp_top_tb;

    reg clk;
    reg [31:0] instructionss[0:11];
    reg [31:0] instruction;
    integer currInstruction;
    wire signed [31:0] result;

    reg signed [31:0] expResult;
    reg signed [31:0] expResults[0:10];

    initial begin

        expResults[0] = 24486 ;
        expResults[1] = 3274 ;
        expResults[2] = 5338 ;
        expResults[3] =-13200 ;
        expResults[4] = 12996 ;
        expResults[5] = 1852 ;
        expResults[6] = 4674 ;
        expResults[7] =-1049 ;
        expResults[8] = 16294 ;
        expResults[9] = 5376 ;
        expResults[10] = 4626 ;

    end

    initial
    begin
        clk = 0;
        currInstruction = 0;
        instructionss[0] = 32'b00000000000000000000100010000001; // r0 = r1+r2   //24486
        instructionss[1] = 32'b00000000000111110001100110000110; // r31 = r6-r3  //3274
        instructionss[2] = 32'b00000000000000000010111110000001101; // r0 = |r24| ,r23     //5338
        instructionss[3] = 32'b00000000000111110111010010001000; // r31 = -r18 ,r14   //-13200
        instructionss[4] = 32'b00000000000000000000110110000111; // r0 = max(r22,r1)    //12996
        instructionss[5] = 32'b00000000000111111101011011000100; // r31 = min(r27,r26)  //1852
        instructionss[6] = 32'b00000000000000000001010010000001011; // r0= avg(r4,r5)      //4674
        instructionss[7] = 32'b00000000000111110010111101001111; // r31 = -r29 ,r5     //-1048=-1049
        instructionss[8] = 32'b00000000000000000101110000000011; // r0 = r16 | r11     //7430|12196=16294
        instructionss[9] = 32'b00000000000111111000111001000101; // r31 = r25 & r17  //5544&14102=5376
        instructionss[10] = 32'b00000000000000001111011101000010; // r0 = r29 ^ r30     // 1048^5642=4626
        instructionss[11] = 32'b00000000000000001111011101001001;  //test for unvalid opcode (9)
    end

    always #10ns clk = ~clk;
```

```
    always #10ns clk = ~clk;

    always @(posedge clk) begin
        if (currInstruction < 12) begin
            instruction = instructionss[currInstruction];
            expResult = expResults[currInstruction];
            currInstruction = currInstruction + 1;
        end
        #10ns
            if (result == expResult)
              $display("Current Instruction = %h, Result = %d, Expected = %d, Pass", instruction, result, expResult);

            else
              $display("Current Instruction = %h, Result = %d, Expected = %d, Fail", instruction, result, expResult);

        if (currInstruction == 12)
            $finish;
    end

    //Sample of the system
    mp_top mpTop(clk, instruction, result);

endmodule
```

Figure 10:system test bench code

When the system test bench run, it gave us this output:

```
Console
# 11:10 PM, Sunday, January 21, 2024
# Simulation has been initialized
run
# KERNEL: Current Instruction = 00000881, Result =       24486, Expected =       24486, Pass
# KERNEL: Current Instruction = 001f1986, Result =        3274, Expected =        3274, Pass
# KERNEL: Current Instruction = 0000be0d, Result =        5338, Expected =        5338, Pass
# KERNEL: Current Instruction = 001f7488, Result =      -13200, Expected =      -13200, Pass
# KERNEL: Current Instruction = 00000d87, Result =       12996, Expected =       12996, Pass
# KERNEL: Current Instruction = 001fd6c4, Result =        1852, Expected =        1852, Pass
# KERNEL: Current Instruction = 0000290b, Result =        4674, Expected =        4674, Pass
# KERNEL: Current Instruction = 001f2f4f, Result =       -1049, Expected =       -1049, Pass
# KERNEL: Current Instruction = 00005c03, Result =       16294, Expected =       16294, Pass
# KERNEL: Current Instruction = 001f8e45, Result =        5376, Expected =        5376, Pass
# KERNEL: Current Instruction = 0000f742, Result =        4626, Expected =        4626, Pass
# KERNEL: Current Instruction = 0000f749, Result =           0, Expected =           x, Fail
# RUNTIME: Info: RUNTIME_0068_project_v (318): $finish called
```

It is clear that the entire system was completed successfully by testing the 10 operations defined in the ALU module, and the values were successfully verified after calculating them. Trying to put an instruction that contains an invalid opcode, and the system does not work on it, which indicates its success.