VLSI-2

Project Report-5

Deniz Zakir EROĞLU- 040200249

Şeyma ÇALIŞKAN- intern at GSTL

Deniz Zakir EROĞLU- 040200249
Şeyma ÇALIŞKAN

Function Unit (FU) Overview

In a RISC-V processor architecture, the **Function Unit (FU)** is a central component responsible for executing core computational tasks. It is an essential part of the **datapath**, acting as the execution engine that handles **arithmetic**, **logical**, **comparison**, and **shift** operations, all of which are fundamental to instruction execution.

The Function Unit designed in this project supports the RV32I base instruction set and includes the following submodules:

- **Arithmetic Module**: Performs addition and subtraction.
- **Logic Module**: Executes bitwise operations such as AND, OR, and XOR.
- **Comparison Module**: Evaluates conditions for branching and set-less-than operations.
- **Shift Module**: Handles logical and arithmetic shift instructions.

Each submodule is controlled by a selection signal, and only one submodule is active during each instruction cycle. The result of the selected operation is forwarded to the destination register (rd), and a status output is provided when required, particularly for branch decisions.

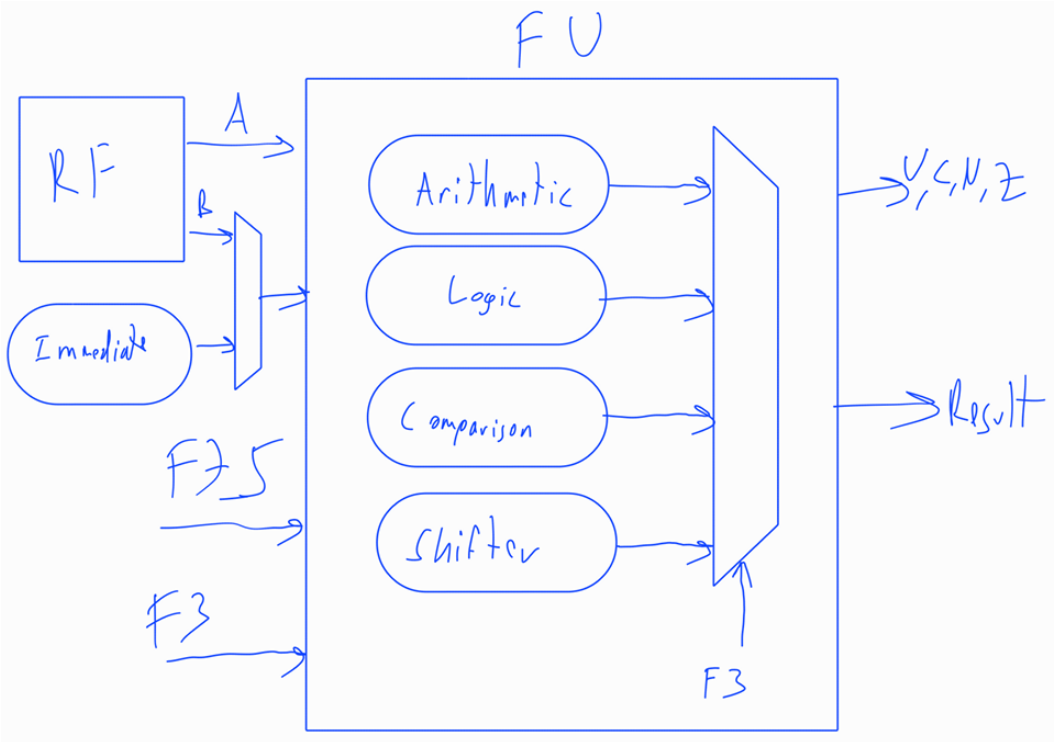

Figure 1: Function Unit Module

Deniz Zakir EROĞLU- 040200249
Şeyma ÇALIŞKAN

Submodule Implementations:

Arithmetic Module

The Arithmetic module is designed to support both addition and subtraction operations with minimal hardware. In RISC-V RV32I, these are among the most frequently used operations, and both rely on two operands (rs1 and rs2 or immediate). Instead of creating separate dedicated adders and subtractors, the module uses a **shared arithmetic unit** controlled by a single-bit selection signal.

- When sel = 0, the unit performs addition.
- When sel = 1, the same inputs are interpreted by the subtractor.

This approach minimizes logic duplication and ensures synthesis efficiency. The subtraction is directly implemented using Verilog's - operator, which internally performs two's complement addition. Since addition and subtraction differ only by inversion of the second operand and carry-in, using a control signal to switch behavior makes the design both compact and logically consistent with the ALU principles.

The code for Arithmetic Module is given below:

```
module Arithmetic(
    input [31:0] A,B,
    input add_sub_sel,
    output [31:0] add_res,
    output V,C,Z,S
  );
    wire signed [31:0] A_signed, B_signed;
    assign A_signed = A;
    assign B_signed = B;
    reg signed [32:0] add_res_wire;

    assign V = ((A_signed[31] == B_signed[31]) && (add_res_wire[31] !=
A_signed[31]));
    assign Z = ~(|add_res_wire);
    assign S = add_res_wire[31];
    assign C = add_res_wire[32];
    assign add_res = add_res_wire;

always @ * begin
    if (add_sub_sel == 1) begin
        add_res_wire <= A_signed - B_signed;
    end
    else begin
        add_res_wire <= A_signed + B_signed;
    end
end
endmodule
```

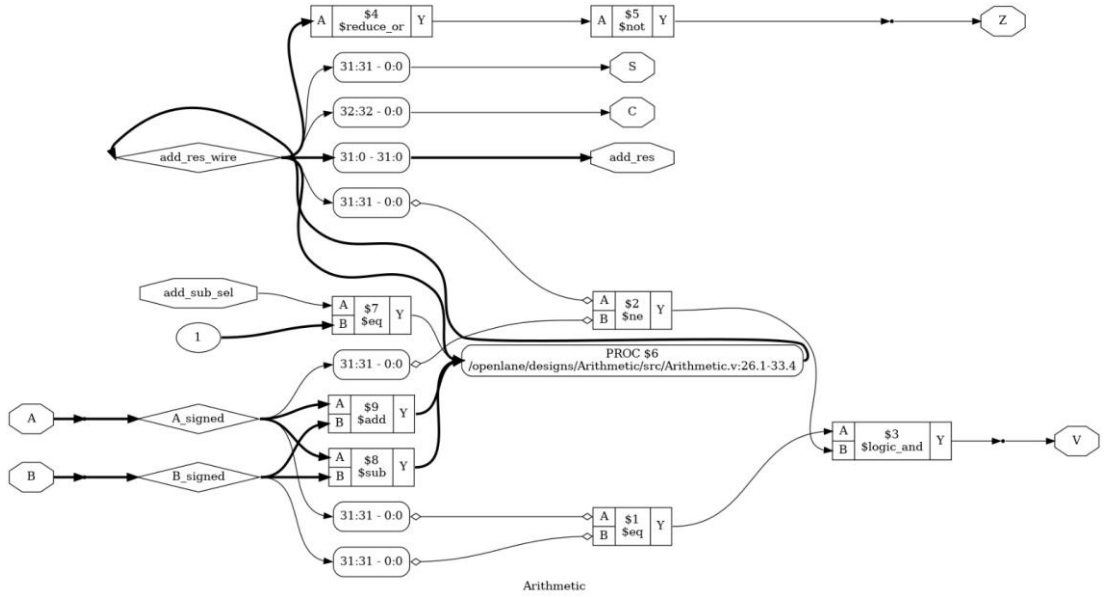

Figure 2: Dot Schematic: Arithmetic Module

```
VCD info: dumpfile Arithmetic_tb.vcd opened for output.
-----Adder Test-----
A:  215631123,B:   48965346,V:0,C:0,result:         19,expected:         19 ==> TRUE
A:   -1653165,B:   51381563,V:0,C:0,result:  264596469,expected:  264596469 ==> TRUE
A:   54185122,B:    -156342,V:0,C:0,result:   49728398,expected:   49728398 ==> TRUE
A:    -156342,B:  -15123442,V:0,C:0,result:   54028780,expected:   54028780 ==> TRUE
A:     -15156,B:  -15123442,V:0,C:1,result:  -15279784,expected:  -15279784 ==> TRUE
```

Figure 3 : Testbench Result : Arithmetic Module

Comparison Module

The Comparison module evaluates conditions required by both relational operations (like SLT) and branch instructions (like BEQ, BLT). The design centralizes all comparison types within a single module, using a 3-bit selector to choose between six possible comparisons.

To maintain compatibility with both **signed** and **unsigned** operations, the module uses Verilog's $signed() casting selectively. This ensures that arithmetic comparisons like BLT correctly interpret negative numbers, while logical comparisons like BLTU remain unsigned.

Implementing all comparisons in one module with a case-based selection structure not only reduces area overhead but also standardizes the output interface (1 for true, 0 for false), which is essential for controlling program flow via branching.

The code for Comparison Module is given below:

```
module Comparison(
input [31:0] A, B,
input [2:0] sel,
output reg [31:0] res,
input S,
input co
);
    always @ * begin
        case (sel)
            'b010: begin
                if(A[31]!=B[31]) res <= A[31];
                else res <= {S};
            end
            'b011: begin
                if(A[31]!=B[31]) res <= ~A[31];
                else res <= {~co};
            end
        endcase
    end

endmodule
```

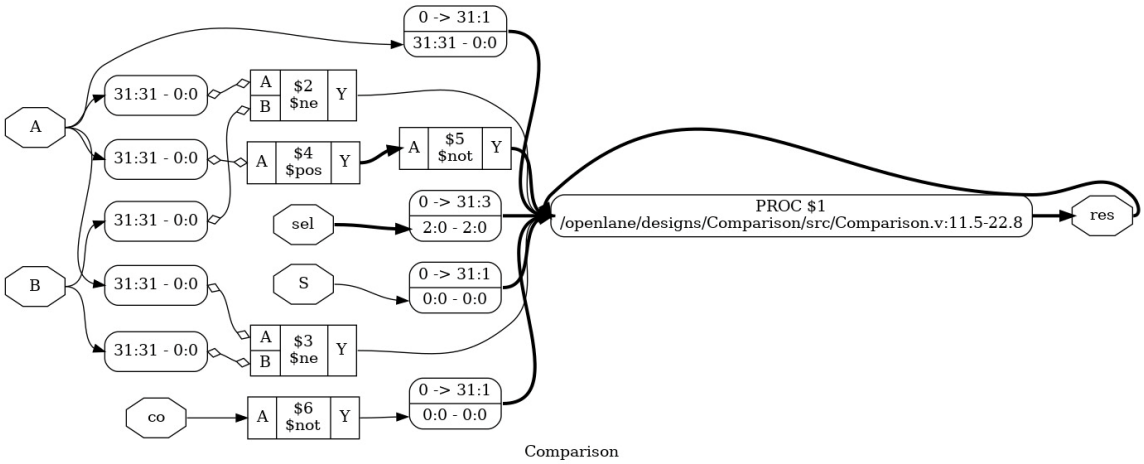

Figure 4: Dot Schematic: Comparison Module

```
VCD info: dumpfile Comparison_tb.vcd opened for output.
-----Comparison-----
10000   80000001        00000001        010     0       0       00000001
20000   00000002        00000001        010     1       1       00000001
30000   80000002        80000001        011     0       0       00000001
```

Figure 5: Testbench Result : Comparison Module

Deniz Zakir EROĞLU- 040200249
Şeyma ÇALIŞKAN

Logic Module

The Logic module executes core bitwise operations (AND, OR, XOR), which are stateless and parallel by nature. These operations are essential for flag manipulation, masking, and low-level data handling in RV32I.

The module design uses a **2-bit selector** to choose among the three supported logic operations. Since these are simple gate-level functions, each operation is implemented directly using Verilog's bitwise operators. There's no need for sequential logic, as the outcome depends only on the inputs at a given time.

This structure ensures minimal propagation delay and efficient synthesis, making the logic unit lightweight but effective in ALU integration.

The code for Logic Module is given below:

```
module Logic(
input [31:0] A, B,
input [2:0] sel,
output reg [31:0] res
);

always @ * begin
    case (sel)
            'b100: begin
                res <= A ^ B;
            end
            'b110: begin
                res <= A | B;
            end
            'b111: begin
                res <= A & B;
            end
            default: begin
                res <= A ^ B;
            end
    endcase
end
endmodule
```

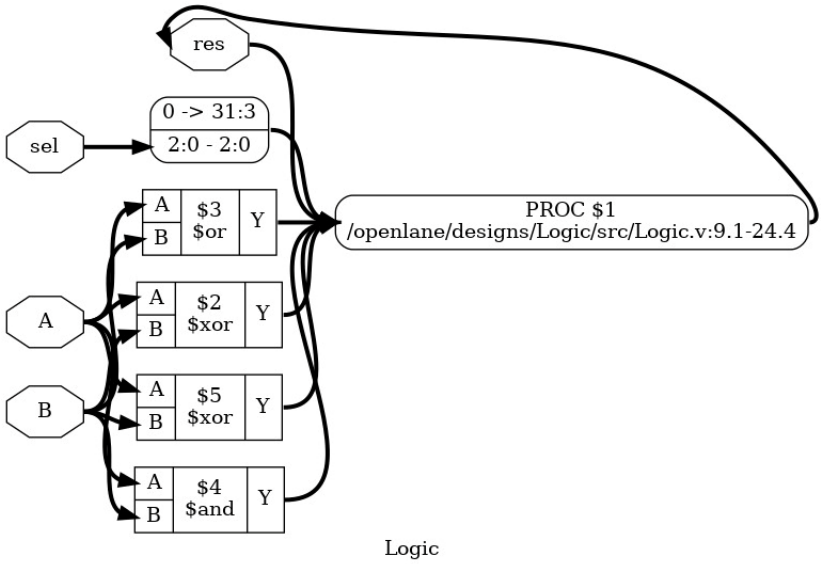

Figure 6: Dot Schematic: Logic Module

```
VCD info: dumpfile Logic_tb.vcd opened for output.
A=5555ffff B=aaaaaaaa sel=100 res=ffff5555
A=5555ffff B=aaaaaaaa sel=110 res=ffffffff
A=5555ffff B=aaaaaaaa sel=111 res=0000aaaa
```

Figure 7: Testbench Result :Logic Module

Shift Module

The Shift module is built to support both **logical** and **arithmetic** shift operations in the RV32I set. The module receives a value and a shift amount as input, and performs:

- Logical left shift (SLL)
- Logical right shift (SRL)
- Arithmetic right shift (SRA)

The need to distinguish between **signed and unsigned shifting** is handled by using Verilog's >>> operator for SRA, which preserves the sign bit. The selection signal (sel) ensures only one shift type is active at a time.

Instead of hardcoding immediate shift amounts (SLLI), the module uses a unified interface where the shift amount is an input, allowing the same hardware to serve both register-based and immediate-based shifts. This modularity simplifies control signal integration and improves reuse across the datapath.

The code for Shifter Module is given below:

```
module Shifter(
    input [31:0] A,
    input [31:0] B,
    input [2:0] select,
    input select_2,
    output reg [31:0] shifter_Res
    );
    always @ * begin
        case(select[2])
            'b0: begin
               shifter_Res <= A << B;
            end
            'b1: begin
                if(select_2) begin
                    shifter_Res <= $signed(A)>>>B;
                end
                else begin
                    shifter_Res <= A>>B;
                end
            end
        endcase
    end
endmodule
```

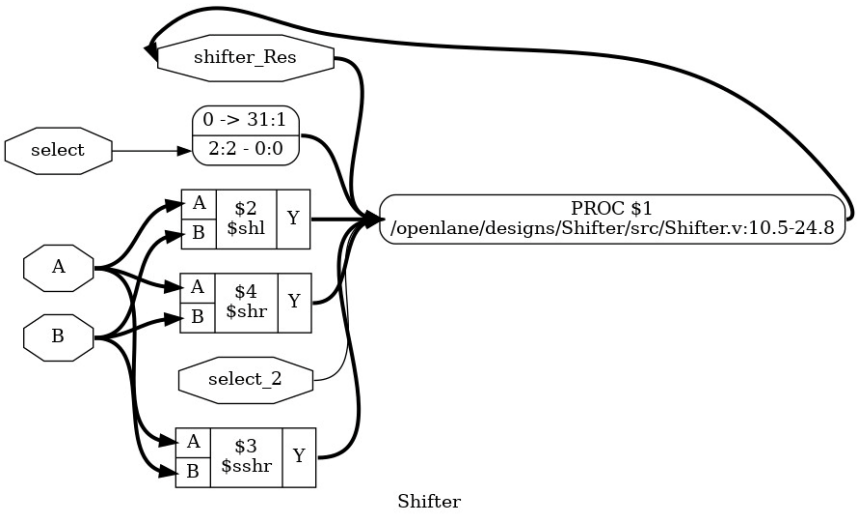

Figure 8: Dot Schematic: Shift Module

```
VCD info: dumpfile Shifter_tb.vcd opened for output.
Test1: A=0000000f B=00000002 select=000 select_2=0 res=0000003c
Test2: A=0000000f B=00000002 select=100 select_2=0 res=00000003
Test3: A=f0000000 B=00000002 select=100 select_2=1 res=fc000000
Shifter_tb.v:41: $finish called at 30000 (1ps)
```

Figure 9: Testbench Result: Shift Module

Function Unit (FU)- Top Module:

The FU takes as input the operand values (rs1, rs2, or immediate) and a set of control signals that are derived from the instruction's opcode, funct3, and funct7 fields. These control signals determine which operation will be performed and which internal submodule will be activated.

	op	F3	F7_5
Arithmetic	Add	000	0
	Sub	000	1
Logic	XOR	100	0
	OR	110	0
	AND	111	0
Comparison	SLT	010	0
	SLTU	011	0
Shifter	SLL	001	0
	SRL	101	0
	SRA	101	1

Figure 10: Function Table

The code for Function Unit is given below:

```
module FU(
    input [31:0] A,
    input [31:0] B,
    input [2:0] sel,
    input func_7,
    output reg [31:0] res,
    output V,
    output C,
    output Z,
    output reg S,
    input shift_sel
    );

    wire [31:0] arithmetic_res;
    wire [31:0] comparison_res;
    wire [31:0] logic_res;
    wire [31:0] shifter_res;
    wire co_arith;
    wire S_arith;
    reg co;
```

```
    Arithmetic
arithmetic_0(.A(A),.B(B),.add_sub_sel(func_7),.add_res(arithmetic_res),.V(V),.Z
(Z),.C(co_arith),.S(S_arith));
    Comparison comparison_0(.A(A), .B(B), .res(comparison_res), .sel(sel),
.S(S_arith), .co(co_arith));
    Logic logic_0(.A(A), .B(B), .res(logic_res), .sel(sel));
    Shifter shifter_0(.A(A), .B(B), .shifter_Res(shifter_res), .select(sel),
.select_2(func_7));

    assign C = co;

    always @ * begin
        if (shift_sel == 1) begin
            res <= shifter_res;
        end
        else begin
            casez (sel)
                'b000: begin
                    res <= arithmetic_res;
                    co <= co_arith;
                    S <= S_arith;
                end
                'b1??: begin
                    res <= logic_res;
                    co <= co_arith;
                    S <= S_arith;
                end
                'b01?: begin
                    res <= comparison_res;
                    co <= co_arith;
                    S <= S_arith;
                end
                default: begin
                    res <= logic_res;
                    co <= co_arith;
                    S <= S_arith;
                end
            endcase
        end
    end
endmodule
```

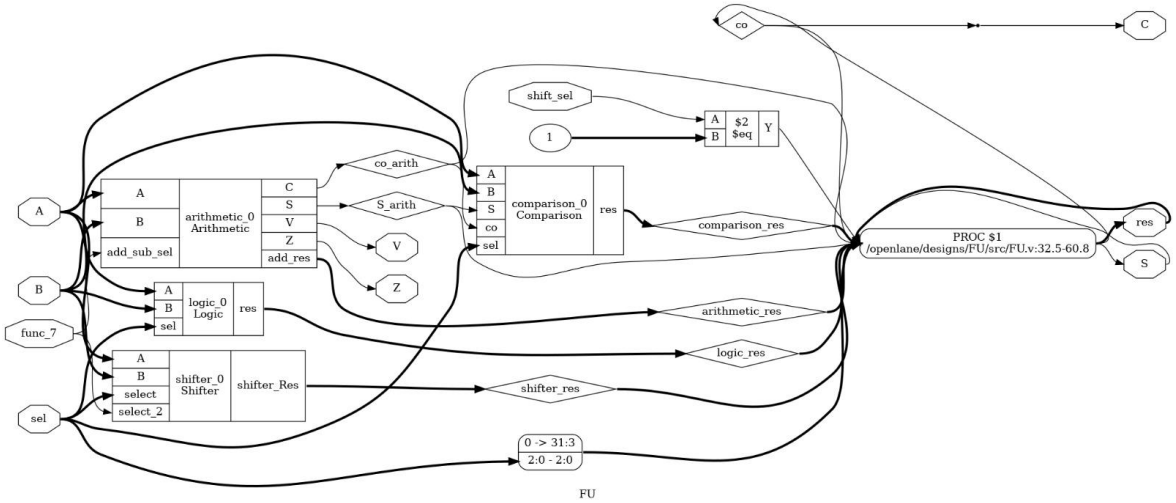

Figure 11: Dot Schematic: Function Unit (FU) Module

As shown in the Figure 10, the Function Unit (FU) successfully passed all test cases across its submodules. The results validate the correct functionality of arithmetic, logic, comparison, and shift operations.

Simulation of Function Unit

The testbench code is given below:

```
module FU_tb(
);
    reg [31:0] A,B;
    reg [2:0] select;
    reg select_2,select_ALU_Shifter;
    wire [31:0] out_FU;
    wire V,C,N,Z;

    FU uut(
        .A(A),.B(B),
        .sel(select),
        .func_7(select_2),
        .shift_sel(select_ALU_Shifter),
        .res(out_FU),
        .V(V),.C(C),.S(N),.Z(Z)
    );

    reg [31:0] temp;
    initial begin
        $dumpfile("FU_tb.vcd");
        $dumpvars(0, FU_tb);
        $display("-----Arithmetic Test-----");
```

```
        A = 21;
        B = 12;
        select = 0;
        select_2 = 0;
        select_ALU_Shifter = 0;
        #50;
        temp = $signed(A)+$signed(B);
        if ($signed(out_FU) == $signed(temp)) begin
        $display("A:%d,B:%d,V:%d,C:%d,result:%d,expected:%d ==>
TRUE",$signed(A),$signed(B),V,C,$signed(out_FU),$signed(temp));
        end
        else $display("A:%d,B:%d,V:%d,C:%d,result:%d,expected:%d ==>
FALSE",$signed(A),$signed(B),V,C,$signed(out_FU),$signed(temp));
        select_2 = 1;
        A = 6425345;
        B = 4572695;
        #50;
        temp = $signed(A)-$signed(B);
        if ($signed(out_FU) == $signed(temp)) begin
        $display("A:%d,B:%d,V:%d,C:%d,result:%d,expected:%d ==>
TRUE",$signed(A),$signed(B),V,C,$signed(out_FU),$signed(temp));
        end
        else $display("A:%d,B:%d,V:%d,C:%d,result:%d,expected:%d ==>
FALSE",$signed(A),$signed(B),V,C,$signed(out_FU),$signed(temp));
        select_2 = 0;
        $display("-----Logic Test-----");
        select = 3'b100;
        A = 32'b1001010;
        B = 32'b1010101;
        #50;
        temp = A ^ B;
        if ($signed(out_FU) == $signed(temp)) begin
        $display("A:%d,B:%d,V:%d,C:%d,result:%d,expected:%d ==>
TRUE",$signed(A),$signed(B),V,C,$signed(out_FU),$signed(temp));
        end
        else $display("A:%d,B:%d,V:%d,C:%d,result:%d,expected:%d ==>
FALSE",$signed(A),$signed(B),V,C,$signed(out_FU),$signed(temp));
        $finish;
    end
endmodule
```

The Post Synthesis Result verified and ready for synthesis and layout steps within the OpenLane design flow. The Post Synthesis Result can be seen in Figure 12.

```
VCD info: dumpfile FU_tb.vcd opened for output.
-----Arithmetic-----
Add: A=21 B=12 Res=33 Exp=33 PASS
Sub: A=6425345 B=4572695 Res=1852650 Exp=1852650 PASS
-----Logic-----
XOR: A=00000000000000000000000001001010 B=00000000000000000000000001010101 Res=00000000000000000000000000011111 Exp=00000000000000000000000000011111 PASS
OR:  A=00000000000000000000000001001010 B=00000000000000000000000001010101 Res=00000000000000000000000001011111 Exp=00000000000000000000000001011111 PASS
AND: A=00000000000000000000000001001010 B=00000000000000000000000001010101 Res=00000000000000000000000001000000 Exp=00000000000000000000000001000000 PASS
-----Comparison-----
LT1: Res=1 Exp=1 PASS
LT2: Res=1 Exp=1 PASS
-----Shifter-----
SHL: Res=000000f0 Exp=000000f0 PASS
SRL: Res=0f000000 Exp=0f000000 PASS
SRA: Res=ff000000 Exp=ff000000 PASS
```

Figure 12: Post Synthesis Result: Function Unit (FU) Module

Synthesis of Function Unit

As shown in the Figure 13 below, the final physical layout of the Function Unit (FU) was successfully generated using the OpenLane flow.

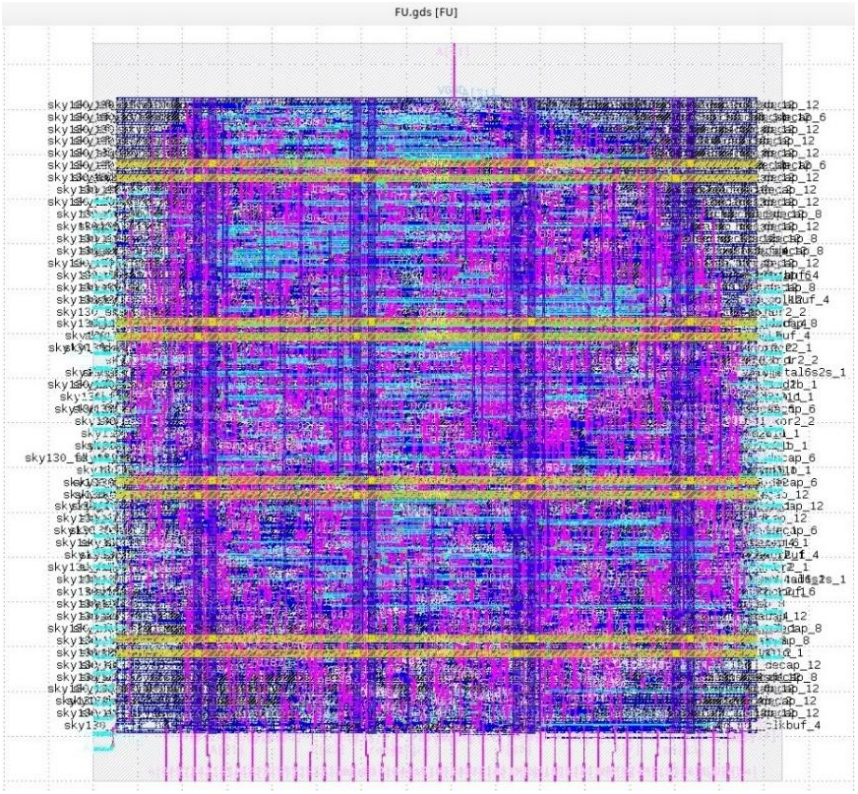

Figure 13: Klayout Function Unit (FU)

The power analysis can be seen in Figure 14.

```
 1
 2 ****************************************************************************
 3  report_power
 4 ****************************************************************************
 5 ======================= Typical Corner ====================================
 6
 7 Group                  Internal  Switching    Leakage      Total
 8                           Power      Power      Power      Power (Watts)
 9 ----------------------------------------------------------------
10 Sequential             3.41e-06   7.46e-07   3.19e-11   4.15e-06   0.8%
11 Combinational          2.34e-04   2.69e-04   4.59e-09   5.03e-04  99.2%
12 Clock                  0.00e+00   0.00e+00   3.37e-10   3.37e-10   0.0%
13 Macro                  0.00e+00   0.00e+00   0.00e+00   0.00e+00   0.0%
14 Pad                    0.00e+00   0.00e+00   0.00e+00   0.00e+00   0.0%
15 ----------------------------------------------------------------
16 Total                  2.38e-04   2.70e-04   4.96e-09   5.07e-04 100.0%
17                           46.8%      53.2%       0.0%
18
```

Figure 14: Function Unit (FU) Report Power

The timing summary can be seen in Figure 15.

```
 1
 2 ****************************************************************************
 3 report_tns
 4 ****************************************************************************
 5 tns 0.00
 6
 7 ****************************************************************************
 8 report_wns
 9 ****************************************************************************
10 wns 0.00
11
12 ****************************************************************************
13 report_worst_slack -max (Setup)
14 ****************************************************************************
15 worst slack 0.80
16
17 ****************************************************************************
18 report_worst_slack -min (Hold)
19 ****************************************************************************
20 worst slack 4.25
```

Figure 15: Function Unit (FU) Timing Summary

The suggested Clock Frequency can be seen in Figure 16.

suggested_clock_frequency
69,34812760055479

Figure 16: Function Unit (FU) Suggested Clock Frequency

The Core Area can be seen in Figure 17.

CoreArea_um^2
20169,343999999997

Figure 17: Function Unit (FU) Core Are

Datapath Module:

Datapath have to include register file, function unit and some muxes to handle the data flow. The muxes for example, immediate selector or memory/function unit data out selector can be controlled by using control word. Also data like Register file (RF) addresses are inputs. Not only control signals but also immediate data or data memory output is the inputs for datapath.

The code for Datapath is given below:

```
module Datapath (input clk,rst_ni,
input [4:0] rs1,rs2,rd,
input mw,we, //memory_write,rf_we
input mb, //B_data_select; RF or imm
input [2:0] func3,
input select_2,//func3_extension func7
input md,// mem_out or FU out
input [31:0] data_in, imm_extend, // memory data in, immediate
input mf, // fu out sel
output [31:0] out_FU,
output V,C,Z,N  //flags for status
   );
    wire [31:0] rf_data_A, rf_data_B;
    wire [31:0] rf_data_in;

    RF#(.DEPTH(32),.DATA_WIDTH(32)) RF(
    .clk(clk),
    .rstn(rst_ni),
    .rd_addr0(rs1),
    .wr_addr0(rd),
    .rd_addr1(rs2),
    .wr_din0(rf_data_in),
    .rd_dout0(rf_data_A),
    .rd_dout1(rf_data_B),
    .we0(we));

    wire [31:0] B_fu;
    mux_2 mux_MB(.i0(rf_data_B),.i1(imm_extend),.sel(mb),.o(B_fu));

    FU FU(
    .A(rf_data_A),
    .B(B_fu),
    .sel(func3),
    .func_7(select_2),
    .shift_sel(mf),
    .res(out_FU),
    .V(V),.C(C),.S(N),.Z(Z));

    mux_2 mux_MD(.i0(out_FU),.i1(data_in),.sel(md),.o(rf_data_in));
endmodule
```

Deniz Zakir EROĞLU- 040200249
Şeyma ÇALIŞKAN

Control Words For Algorithm

The values of **A** and **N** are initially stored in the data memory. Since RISC architectures do not support direct arithmetic operations between memory and registers, the program begins by **loading A and N into the 2nd and 3rd registers** of the Register File (RF). This step ensures that all arithmetic operations can be performed within the register set.

Next, the value **C is initialized to 1** and stored in the 1st register (x1) of the RF. The subsequent instruction performs the operation **C = C + C**, which requires all three register addresses (rs1, rs2, and rd) to be set to 0x1. This means that the value of register 1 is doubled and the result is stored back into the same register.

All control signal configurations and register mappings used for this operation sequence are summarized in **Table 1**.

OP		RS1	RS2	RD	MW	WE	MB	FUNC3	FUNC7_5	MD	MF	BC
RF[2]=A	IMM=2	0	0	2	0	1	1	0	0	1	X	X
RF[3]=N	IMM=3	0	0	3	0	1	1	0	0	1	X	X
C=1	IMM=1	0	0	1	0	1	1	0	0	0	0	X
C=C+C	IMM=X	1	1	1	0	1	0	0	0	0	0	X
C=C-N	IMM=X	1	3	1	0	1	0	0	1	0	0	X
C=C+A	IMM=X	1	2	1	0	1	0	0	0	0	0	X
C=C-N	IMM=X	1	3	1	0	1	0	0	1	0	0	X

Tablo 1: Control Words for Project

Simulation of Datapath:

The testbench code is given below:

```
module datapath_tb(
);
reg clk,rst_ni;
reg [4:0] rs1,rs2,rd;
reg mw,we; // memory_write,rf_we
reg mb; // B_data_select; RF or imm
reg [2:0] func3;
reg func7; // func3_extension func7
reg md; // mem_out or FU out
reg [31:0] data_in, imm_extend; // memory data in, immediate
reg mf; // fu out sel
wire [31:0] out_FU;
wire V,C,Z,N;  // flags for status
wire [31:0] data_from_mem;
```

```
    Datapath uut(
        .clk(clk),.rst_ni(rst_ni),
        .rs1(rs1),.rs2(rs2),.rd(rd),
        .mw(mw),.we(we), //memory_write,rf_we
        .mb(mb), // B_data_select; RF or imm
        .func3(func3),
        .func7(func7), // func3_extension func7
        .md(md), // mem_out or FU out
        .data_in(data_from_mem),
        .imm_extend(imm_extend), // memory data in, immediate
        .mf(mf), // fu out sel
        .out_FU(out_FU),
        .V(V),.C(C),.Z(Z),.N(N));  // flags for status
    data_mem DM(
    .clk(clk),.rstn(rst_ni),
    .wr_strb(),
    .rd_addr0(out_FU),.wr_addr0(),
    .wr_din0(),
    .rd_dout0(data_from_mem),
    .we0());

    initial begin
        $dumpfile("datapath_tb.vcd");
        $dumpvars(0,datapath_tb);
        clk = 0;
        rst_ni = 0;
        #20;
        rst_ni = 1;
        #20; // data_load A
        mb=1;
        imm_extend=1;
        mf=0;
        rs1=0;
        rd = 2;
        we=1;
        func3=0;
        func7=0;
        md=1;
        #100; // data_load N
        mb=1;
        imm_extend=2;
        mf=0;
        rs1=0;
        rd = 3;
        we=1;
        func3=0;
        func7=0;
        md=1;
        #100; // C initalize by 1
```

```
        mb=1;
        imm_extend=1;
        mf=0;
        rs1=0;
        rd = 1;
        we=1;
        func3=0;
        func7=0;
        md=0;
        #100; // C = C + C
        mb=0;
        imm_extend=1;
        mf=0;
        rs1=1;
        rs2=1;
        rd = 1;
        we=1;
        func3=0;
        func7=0;
        md=0;
        #100; // C = C - N
        mb=0;
        imm_extend=1;
        mf=0;
        rs1=1;
        rs2=3;
        rd = 1;
        we=1;
        func3=0;
        func7=1;
        md=0;
        #100; // C = C + A
        mb=0;
        imm_extend=1;
        mf=0;
        rs1=1;
        rs2=2;
        rd = 1;
        we=1;
        func3=0;
        func7=0;
        md=0;
        #100; // C = C - N
        mb=0;
        imm_extend=1;
        mf=0;
        rs1=1;
        rs2=3;
        rd = 1;
        we=1;
        func3=0;
        func7=1;
```

```
        md=0;
        #100; // NOP
        mb=1;
        imm_extend=0;
        mf=0;
        rs1=0;
        rs2=3;
        rd = 0;
        we=1;
        func3=0;
        func7=1;
        md=0;
        #100;
        $finish;
    end

 always @(datapath_tb.uut.RF.mem[1])
 $display("RF[1] (C value) :%d",$signed(datapath_tb.uut.RF.mem[1]));
 always @(datapath_tb.uut.RF.mem[2])
 $display("RF[2] (A value) :%d",$signed(datapath_tb.uut.RF.mem[2]));
 always @(datapath_tb.uut.RF.mem[3])
 $display("RF[3] (N value) :%d",$signed(datapath_tb.uut.RF.mem[3]));

    always #50 clk = ~clk;
endmodule
```

The zero result can be seen in Figure 18.

```
RF[1] (C value) :           0
RF[2] (A value) :           0
RF[3] (N value) :           0
RF[2] (A value) :           8
RF[3] (N value) :           5
RF[1] (C value) :           1
RF[1] (C value) :           2
RF[1] (C value) :          -3
RF[1] (C value) :           5
RF[1] (C value) :           0
```

Figure 18: The output Datapath for 0 Result

The positive result can be seen in Figure 19.

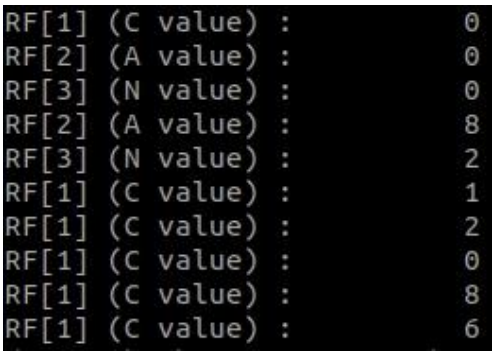

Figure 19: The output Datapath for Positive Result

The negative result can be seen in Figure 20.

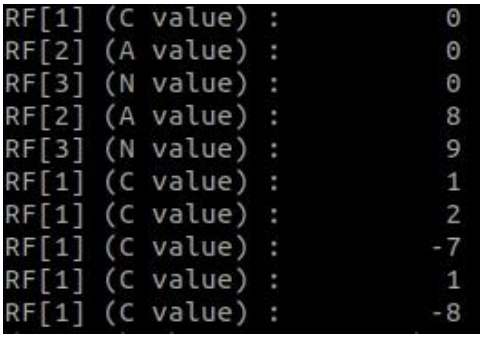

Figure 20: The Output Datapath for Negative Result