Gate Level Behavior Model of a 32-Bit Processor Using Verilog

Abdurrahman Mohammad
San Jose State University Computer Science Department
San Jose State University Santa Clara, United States
abdurrahman.mohammad@sjsu.edu

***Abstract* – This paper covers the gate level design and implementation of an ALU and a 32x32 register file. The language used to model this processor is Verilog. This report explains how the listed components work together to implement an ALU and register file specific to "CS 147DV" with extensive testing.**

I. Introduction

The goal of this project is to implement an ALU and register file for the instruction set of CS 147DV at the gate level. To do this, we need to design the lower level components involved in the functioning of the ALU and register file. The design for this implementation has been completed in Verilog using ModelSim by Mentor Graphics. The design for this implementation has been completed and tested in Verilog using ModelSim by Mentor Graphics.

II. General Information

Tools used:

a) An adequate computer
b) Windows 10 by Microsoft
c) ModelSim by Mentor Graphics
d) Starter code and guidelines provided by SJSU Professor Kaushik Patra
e) Designs of ALU, register file, and their lower level components by Professor Kaushik Patra

III. Components

The ALU and register file can be further broken down into smaller components on the gate level. Below are some general components used in this project:

32-Bit Logic Components

- 32-bit INV – Inverts a block of 32-bits
- 32-bit AND – Performs the AND operation on 32-bits
- 32-bit OR - Performs the OR operation on 32-bits
- 32-bit NOR - Performs the NOR operation on 32-bits

Multiplexers

- 1-bit multiplexer
- 32-bit 2x1 multiplexer
- 32-bit 4x1 multiplexer
- 32-bit 8x4 multiplexer
- 32-bit 16x1 multiplexer
- 32x32 multiplexer
- 64-bit 2x1 multiplexer

ALU Components

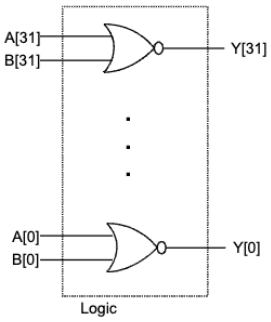
- Ripple carry adder and subtractor
    - Full adder – made from 2 half adders
        - Half adder – made form an XOR gate
- Shifter
    - Barrel shifter
        - Left barrel shifter
        - Right barrel shifter
- Multiplier
- Two's complement 64-bit
    - Two's complement 32-bit

Register File Components

- Components required in the ALU
- 5x32 decoder
- 4x16 decoder
- 3x8 decoder
- 2x4 decoder
- D-Flip-Flop
- D-Latch
- SR-Latch
- Register 1-bit
- Register 32-bit

IV. 32-Bit Logic Components Design

Below are the designs of basic 32-bit logic components created from basic bit level logic gates. Most of these components were derived from the diagram. The 32-bit NOR is made from a 32-bit OR and a 32-bit INV.



32-bit INV – Inverts a block of 32-bits

```
// 32-bit inverter
module INV32_1x1(Y,A);
//output
output [31:0] Y;
//input
input [31:0] A;

// My work below
genvar i;
generate
for (i = 0; i < 32; i = i + 1) begin : inv32_gen_loop
        not not_inst(Y[i], A[i]);
end
endgenerate
endmodule
```

32-bit AND – Performs the AND operation on 32-bits

```
// 32-bit AND
module AND32_2x1(Y,A,B);
//output
output [31:0] Y;
//input
input [31:0] A;
input [31:0] B;
// My work below
genvar i;
generate
for (i = 0; i < 32; i = i + 1) begin : and32_gen_loop
       and and_inst(Y[i], A[i], B[i]);
end
endgenerate
endmodule
```

32-bit OR - Performs the OR operation on 32-bits

```
// 32-bit OR
module OR32_2x1(Y,A,B);
//output
output [31:0] Y;
//input
input [31:0] A;
input [31:0] B;

// My work below
genvar i;
generate
for (i = 0; i < 32; i = i + 1) begin : or32_gen_loop
        or or_inst(Y[i], A[i], B[i]);
end
endgenerate
endmodule
```

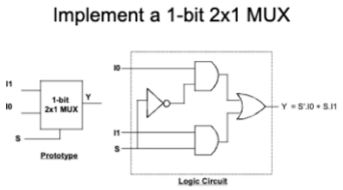
32-bit NOR - Performs the NOR operation on 32-bits

```
// 32-bit NOR
module NOR32_2x1(Y,A,B);
//output
output [31:0] Y;
//input
input [31:0] A;
input [31:0] B;

wire [31:0] or_w; // Wire to transfer the result of 32-bit or
OR32_2x1 or32_inst(.Y(or_w), .A(A), .B(B)); // OR A
INV32_1x1 inv32_inst(.Y(Y), .A(or_w)); // INV (OR A)

endmodule
```

V. Multiplexers

Below are the designs and implementation of the different multiplexers used in this project:

1-bit multiplexer: This basic mux is made out of 1 NOT gate, 2 AND gates, and 1 OR gate. The design in the picture has been translated into Verilog. **With this mux, you can implement bigger multiplexers like the ones that follow**.



File : mux.v

```
// 1-bit mux
module MUX1_2x1(Y,I0, I1, S);
//output list
output Y;
//input list
input I0, I1, S;

// Task 1: Implement a 1-bit mux
wire not_w, and_1_w, and_2_w;
not not_1(not_w, S);
and and_1(and_1_w, I0, not_w);
and and_2(and_2_w, I1, S);
or or_1(Y, and_1_w, and_2_w);
endmodule
```

32-bit 2x1 multiplexer

```
// 32-bit mux
module MUX32_2x1(Y, I0, I1, S);
// output list
output [31:0] Y;
//input list
input [31:0] I0;
input [31:0] I1;
input S;

// Task1: Generate 32 1 bit 2x1 multiplexers
genvar i;
generate
for (i = 0; i < 32; i = i + 1) begin : mux_gen_loop // Definition of the loo
       MUX1_2x1 mux2x1_inst(Y[i], I0[i], I1[i], S); // Instantiate and conn
end
endgenerate
endmodule
```

32-bit 4x1 multiplexer

```
// 32-bit 4x1 mux
module MUX32_4x1(Y, I0, I1, I2, I3, S);
// output list
output [31:0] Y;
//input list
input [31:0] I0;
input [31:0] I1;
input [31:0] I2;
input [31:0] I3;
input [1:0] S;

wire [31:0] mux_1_w, mux_2_w; // Task 1: Create and attach 2 2x1 multiplexers
MUX32_2x1 mux_1(.Y(mux_1_w), .I0(I0), .I1(I1), .S(S[0]));
MUX32_2x1 mux_2(.Y(mux_2_w), .I0(I2), .I1(I3), .S(S[0]));
MUX32_2x1 mux_3(.Y(Y), .I0(mux_1_w), .I1(mux_2_w), .S(S[1])); // Task 2: Create
endmodule
```

32-bit 8x4 multiplexer

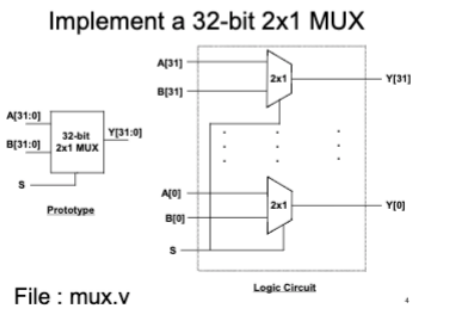
```
// 32-bit 8x1 mux
module MUX32_8x1(Y, I0, I1, I2, I3, I4, I5, I6, I7, S);
// output list
output [31:0] Y;
//input list
input [31:0] I0;
input [31:0] I1;
input [31:0] I2;
input [31:0] I3;
input [31:0] I4;
input [31:0] I5;
input [31:0] I6;
input [31:0] I7;
input [2:0] S;

// Task 1: Create and attach 2 4x1 multiplexers
wire [31:0] mux_1_w, mux_2_w;
MUX32_4x1 mux_1(.Y(mux_1_w), .I0(I0), .I1(I1), .I2(I2), .I3(I3), .S(S[1:0]));
MUX32_4x1 mux_2(.Y(mux_2_w), .I0(I4), .I1(I5), .I2(I6), .I3(I7), .S(S[1:0]));
// Task 2: Create a 2x1 multiplexer to choose between the above 2
MUX32_2x1 mux_3(.Y(Y), .I0(mux_1_w), .I1(mux_2_w), .S(S[2]));
endmodule
```

32-bit 16x1 multiplexer

```
// 32-bit 16x1 mux
module MUX32_16x1(Y, I0, I1, I2, I3, I4, I5, I6, I7,
                  I8, I9, I10, I11, I12, I13, I14, I15, S);
// output list
output [31:0] Y;
//input list
input [31:0] I0;
input [31:0] I1;
input [31:0] I2;
input [31:0] I3;
input [31:0] I4;
input [31:0] I5;
input [31:0] I6;
input [31:0] I7;
input [31:0] I8;
input [31:0] I9;
input [31:0] I10;
input [31:0] I11;
input [31:0] I12;
input [31:0] I13;
input [31:0] I14;
input [31:0] I15;
input [3:0] S;

// Task 1: Create and attach 2 8x1 multiplexers
wire [31:0] mux_1_w, mux_2_w;
MUX32_8x1 mux_1(.Y(mux_1_w), .I0(I0), .I1(I1), .I2(I2), .I3(I3), .I4(I4), .I5(I5), .I6(I6), .I7(I7), .S(S[2:0]));
MUX32_8x1 mux_2(.Y(mux_2_w), .I0(I8), .I1(I9), .I2(I10), .I3(I11), .I4(I12), .I5(I13), .I6(I14), .I7(I15), .S(S[2:0]));
// Task 2: Create a 2x1 multiplexer to choose between the above 2
MUX32_2x1 mux_3(.Y(Y), .I0(mux_1_w), .I1(mux_2_w), .S(S[3]));
endmodule
```

32-bit multiplexer 32x1

```
// 32-bit mux
module MUX32_32x1(Y, I0, I1, I2, I3, I4, I5, I6, I7,
                  I8, I9, I10, I11, I12, I13, I14, I15,
                  I16, I17, I18, I19, I20, I21, I22, I23,
                  I24, I25, I26, I27, I28, I29, I30, I31, S);
// output list
output [31:0] Y; // Result
//input list
input [31:0] I0, I1, I2, I3, I4, I5, I6, I7;
input [31:0] I8, I9, I10, I11, I12, I13, I14, I15;
input [31:0] I16, I17, I18, I19, I20, I21, I22, I23;
input [31:0] I24, I25, I26, I27, I28, I29, I30, I31;
input [4:0] S; // Control

// Task 1: Create and attach 2 16x1 multiplexers
wire [31:0] mux_1_w, mux_2_w;
MUX32_16x1 mux_1(.Y(mux_1_w), .I0(I0), .I1(I1), .I2(I2), .I3(I3), .I4(I4), .I5(I5), .I6(I6), .I7(I7),
                 .I8(I8), .I9(I9), .I10(I10), .I11(I11), .I12(I12), .I13(I13), .I14(I14), .I15(I15), .S(S[3:0])); // mux 1
MUX32_16x1 mux_2(.Y(mux_2_w), .I0(I16), .I1(I17), .I2(I18), .I3(I19), .I4(I20), .I5(I21),
                 .I6(I22), .I7(I23), .I8(I24), .I9(I25), .I10(I26), .I11(I27), .I12(I28), .I13(I29), .I14(I30), .I15(I31), .S(S[3:0]));
// Task 2: Create a 2x1 multiplexer to choose between the above 2
MUX32_2x1 mux_3(.Y(Y), .I0(mux_1_w), .I1(mux_2_w), .S(S[4]));

endmodule
```



File : mux.v

64-bit 2x1 multiplexer

```
module MUX64_2x1(Y, I0, I1, S);
// output list
output [63:0] Y;
//input list
input [63:0] I0;
input [63:0] I1;
input S;

genvar i;
generate
for(i = 0; i < 64; i = i + 1) begin : mux64_loop //generate 64-bit 2x1 mux
        MUX1_2x1 mux64_inst(Y[i], I0[i], I1[i], S);
end
endgenerate
endmodule
```

Multiplexer Test Bench

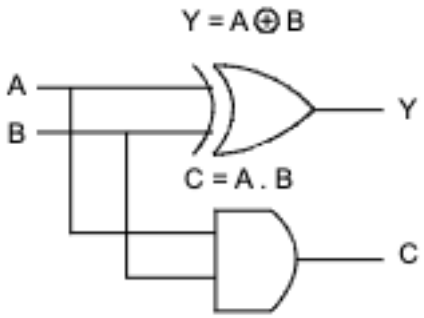
```
VSIM 17> run -all
#
# OP1 = 0, OP2 = 0,   S = 0 ==> Result = 0
# OP1 = 1, OP2 = 0,   S = 0 ==> Result = 1
# OP1 = 0, OP2 = 1,   S = 0 ==> Result = 0
# OP1 = 1, OP2 = 1,   S = 0 ==> Result = 1
# OP1 = 0, OP2 = 0,   S = 1 ==> Result = 0
# OP1 = 1, OP2 = 0,   S = 1 ==> Result = 0
# OP1 = 0, OP2 = 1,   S = 1 ==> Result = 1
OP1 = 1, OP2 = 1,   S = 1 ==> Result = 1
VSIM 18>
```

VI. ALU Design and Testing

Below are the designs and test results of the essential components involved in the ALU:

Half adder – made form 1 XOR and 1 AND gate

Design



Implementation in Verilog

```
//-----------------------------------------------------------------
`include "prj_definition.v"

module HALF_ADDER(Y,C,A,B);
output Y,C;
input A,B;

xor xor_gate(Y, A, B); // Task 1: Half adder adds with XOR
and and_gate(C, A, B); // Task 2: Calculate carry

endmodule
```

Test Bench File

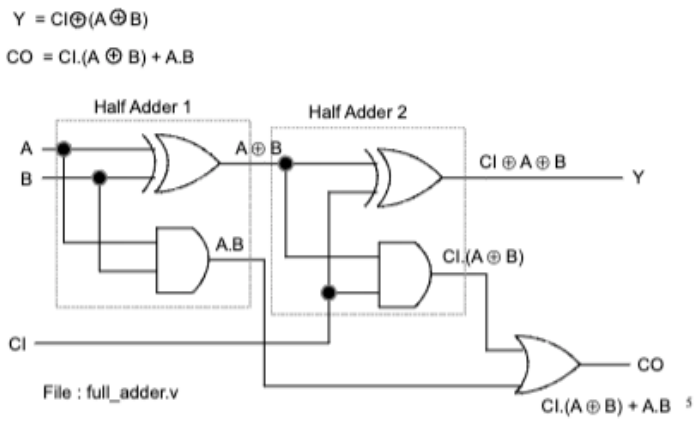
```
`timescale 1ns/1ps
module half_adder_tb;
        reg A, B;
        wire Y, C;
        HALF_ADDER ha_inst(.Y(Y), .C(C), .A(A), .B(B));
        initial begin
                #5 A=0; B=0; // Initial values of OP1 and OP2
                #5 $write("\nOP1 = %d,\tOP2 = %d,\t==> OP1 + OP2 = %d,\tCO = %d", A, B, Y, C);
                #5 A=0; B=1;
                #5 $write("\nOP1 = %d,\tOP2 = %d,\t==> OP1 + OP2 = %d,\tCO = %d", A, B, Y, C);
                #5 A=1; B=0;
                #5 $write("\nOP1 = %d,\tOP2 = %d,\t==> OP1 + OP2 = %d,\tCO = %d", A, B, Y, C);
                #5 A=1; B=1;
                #5 $write("\nOP1 = %d,\tOP2 = %d,\t==> OP1 + OP2 = %d,\tCO = %d", A, B, Y, C);
                #5;
        end
endmodule
```

Test Bench Results

```
# Loading work.HALF_ADDER
VSIM 2> run -all
#
# OP1 = 0, OP2 = 0, ==> OP1 + OP2 = 0, CO = 0
# OP1 = 0, OP2 = 1, ==> OP1 + OP2 = 1, CO = 0
# OP1 = 1, OP2 = 0, ==> OP1 + OP2 = 1, CO = 0
OP1 = 1, OP2 = 1, ==> OP1 + OP2 = 0, CO = 1
VSIM 3>
```

Full adder – made from 2 connected half adders and 1 OR gate to calculate carry out values

Design



Implementation in Verilog

```
''
`include "prj_definition.v"

module FULL_ADDER(S,CO,A,B, CI);
output S,CO;
input A,B, CI;

// Task 1: Create wires
wire hal_result; // Half adder 1 operand
wire hal_co; // Half adder 1 carry out
wire ha2_co; // Half adder 2 carry out
// Task 2: Instantiate 2 half adders and connect them
HALF_ADDER ha_inst_1(.Y(hal_result), .C(hal_co), .A(A), .B(B));
HALF_ADDER ha_inst_2(.Y(S), .C(ha2_co), .A(hal_result), .B(CI));
// Task 3: Calculate carry out: CO = (hal_co || ha2_co)
or or_inst(CO, hal_co, ha2_co);
endmodule
```

Test Bench File

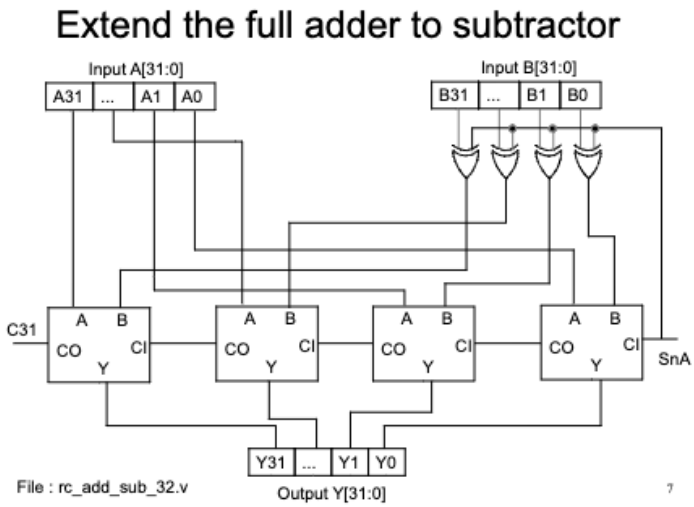
```
`timescale 1ns/1ps
module full_adder_tb;
        reg operand1, operand2, carryIn;
        wire sum, carryOut;
        FULL_ADDER fa_inst(.S(sum), .CO(carryOut), .A(operand1), .B(operand2), .CI(carryIn));
        // OP1 = A, OP2 = B, Carry In = CI, SUM = S, Carry Out = CO
        initial begin
                #5 operand1 = 0; operand2 = 0; carryIn = 0;  // Initial values of OP1 and OP2
                #5 $write("\nOP1 = %d,\tOP2 = %d,\tCI = %d,\t==> OP1 + OP2 = %d,\tCO = %d", operand1, operand2, carryIn, sum, carryOut);
                #5 operand1=1; operand2=0; carryIn=0;
                #5 $write("\nOP1 = %d,\tOP2 = %d,\tCI = %d,\t==> OP1 + OP2 = %d,\tCO = %d", operand1, operand2, carryIn, sum, carryOut);
                #5 operand1=0; operand2=1; carryIn=0;
                #5 $write("\nOP1 = %d,\tOP2 = %d,\tCI = %d,\t==> OP1 + OP2 = %d,\tCO = %d", operand1, operand2, carryIn, sum, carryOut);
                #5 operand1=1; operand2=1; carryIn=0;
                #5 $write("\nOP1 = %d,\tOP2 = %d,\tCI = %d,\t==> OP1 + OP2 = %d,\tCO = %d", operand1, operand2, carryIn, sum, carryOut);
                #5 operand1=0; operand2=0; carryIn=1;
                #5 $write("\nOP1 = %d,\tOP2 = %d,\tCI = %d,\t==> OP1 + OP2 = %d,\tCO = %d", operand1, operand2, carryIn, sum, carryOut);
                #5 operand1=1; operand2=0; carryIn=1;
                #5 $write("\nOP1 = %d,\tOP2 = %d,\tCI = %d,\t==> OP1 + OP2 = %d,\tCO = %d", operand1, operand2, carryIn, sum, carryOut);
                #5 operand1=0; operand2=1; carryIn=1;
                #5 $write("\nOP1 = %d,\tOP2 = %d,\tCI = %d,\t==> OP1 + OP2 = %d,\tCO = %d", operand1, operand2, carryIn, sum, carryOut);
                #5 operand1=1; operand2=1; carryIn=1;
                #5 $write("\nOP1 = %d,\tOP2 = %d,\tCI = %d,\t==> OP1 + OP2 = %d,\tCO = %d", operand1, operand2, carryIn, sum, carryOut);
                #5;
        end
endmodule
```

Test Bench Results

```
VSIM 5> run -all
#
# OP1 = 0, OP2 = 0, CI = 0, ==> OP1 + OP2 = 0, CO = 0
# OP1 = 1, OP2 = 0, CI = 0, ==> OP1 + OP2 = 1, CO = 0
# OP1 = 0, OP2 = 1, CI = 0, ==> OP1 + OP2 = 1, CO = 0
# OP1 = 1, OP2 = 1, CI = 0, ==> OP1 + OP2 = 0, CO = 1
# OP1 = 0, OP2 = 0, CI = 1, ==> OP1 + OP2 = 1, CO = 0
# OP1 = 1, OP2 = 0, CI = 1, ==> OP1 + OP2 = 0, CO = 1
# OP1 = 0, OP2 = 1, CI = 1, ==> OP1 + OP2 = 0, CO = 1
OP1 = 1, OP2 = 1, CI = 1, ==> OP1 + OP2 = 1, CO = 1
VSIM 6>
```

Ripple carry adder and subtractor – made form the number of input bits n XOR gates and n adders

Design



32-bit implementation

```
module RC_ADD_SUB_32(Y, CO, A, B, SnA);
// output list
output [31:0] Y;
output CO;
// input list
input [31:0] A;
input [31:0] B;
input SnA;

wire [31:0] xor_result_w, co_w; // Make wires for the xor result (if we are adding or subtracting) and for
genvar i;
generate
for (i = 0; i < 32; i = i + 1) begin : rc_add_sub_31_loop // Loop 32 times for the 64 bits
        xor xor_inst(xor_result_w[i], SnA, B[i]); // xor every bit of the input: xor_result = SNA xor B[]
        if(i == 0) begin // For the first bit of the input: i = 0
                FULL_ADDER fa_inst(.S(Y[i]), .CO(co_w[i]), .A(A[i]), .B(xor_result_w[i]), .CI(SnA));
        end else if(i != 31 && i!= 0) begin // For all other bits other than first and last
                FULL_ADDER fa_inst(.S(Y[i]), .CO(co_w[i]), .A(A[i]), .B(xor_result_w[i]), .CI(co_w[i - 1]))
        end else if(i == 31) begin // For the last bit
                FULL_ADDER fa_inst(.S(Y[i]), .CO(CO), .A(A[i]), .B(xor_result_w[i]), .CI(co_w[i - 1])); //
        end
        end
endgenerate
endmodule
```

64-bit implementation

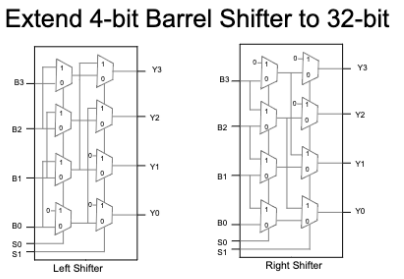
```
module RC_ADD_SUB_64(Y, CO, A, B, SnA);
// output list
output [63:0] Y;
output CO;
// input list
input [63:0] A;
input [63:0] B;
input SnA;

wire [63:0] xor_result_w, co_w; // Make wires for the xor result (if we are adding or subtracti
genvar i;
generate
for (i = 0; i < 64; i = i + 1) begin : rc_add_sub_64_loop // Loop 64 times for the 64 bits
        xor xor_inst(xor_result_w[i], SnA, B[i]); // xor every bit of the input: xor_result = S
        if(i == 0) begin // For the first bit of the input: i = 0
                FULL_ADDER fa_inst(.S(Y[i]), .CO(co_w[i]), .A(A[i]), .B(xor_result_w[i]), .CI(S
        end else if(i != 63 && i!= 0) begin // For all other bits other than first and last
                FULL_ADDER fa_inst(.S(Y[i]), .CO(co_w[i]), .A(A[i]), .B(xor_result_w[i]), .CI(C
        end else if(i == 63) begin // For the last bit
                FULL_ADDER fa_inst(.S(Y[i]), .CO(CO), .A(A[i]), .B(xor_result_w[i]), .CI(co_w[3
        end
        end
endgenerate
endmodule
```

Test Bench Results

```
VSIM 23> run -all
#
# OP1 =           0, OP2 =           0, SnA = 0  ==> OP1 + OP2 =           0, CO = 0
# OP1 =           0, OP2 =           0, SnA = 1  ==> OP1 + OP2 =           0, CO = 1
# OP1 =          10, OP2 =           5, SnA = 0  ==> OP1 + OP2 =          15, CO = 0
# OP1 =          10, OP2 =           5, SnA = 1  ==> OP1 + OP2 =           5, CO = 1
# OP1 =           5, OP2 =          10, SnA = 0  ==> OP1 + OP2 =          15, CO = 0
# OP1 =           5, OP2 =          10, SnA = 1  ==> OP1 + OP2 = 4294967291, CO = 0
OP1 =          0, OP2 =          1, SnA = 1  ==> OP1 + OP2 = 4294967295, CO = 0
VSIM 24>
```

Left barrel shifter – made of a series of interconnected multiplexers

Design:



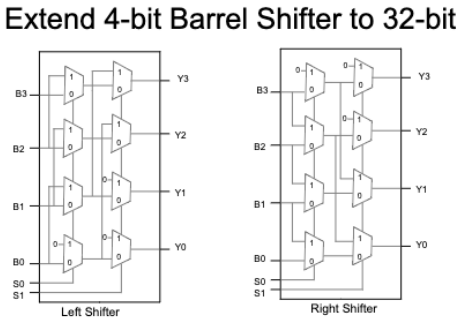
File: barrel_shifter.v

Implementation

```
// Left shifter
module SHIFT32_L(Y,D,S);
// output list
output [31:0] Y;
// input list
input [31:0] D;
input [4:0] S;

// Wires between the multiplexers
wire [31:0] wire0; // Shift 1-bit Y/N Result
wire [31:0] wire1; // Shift 2-bit Y/N Result
wire [31:0] wire2; // Shift 4-bit Y/N Result
wire [31:0] wire3; // Shift 8-bit Y/N Result

MUX32_2x1 mux_inst0(wire0, D , {D[30:0],1'b0}, S[0]); // (Shift 1-bit <<1) Choose between D and shifted D
MUX32_2x1 mux_inst1(wire1, wire0, {wire0[29:0],2'b0}, S[1]); // (Shift 2-bit <<2) Choose between D and shifted D
MUX32_2x1 mux_inst2(wire2, wire1, {wire1[27:0],4'b0}, S[2]); // (Shift 4-bit <<4) Choose between D and shifted D
MUX32_2x1 mux_inst3(wire3, wire2, {wire2[23:0],8'b0}, S[3]); // (Shift 8-bit <<8) Choose between D and shifted D
MUX32_2x1 mux_inst4(Y, wire3, {wire3[15:0],16'b0}, S[4]); // (Shift 16-bit <<16) Choose between D and shifted D
endmodule
```

Right barrel shifter - made of a series of interconnected multiplexers

Design:



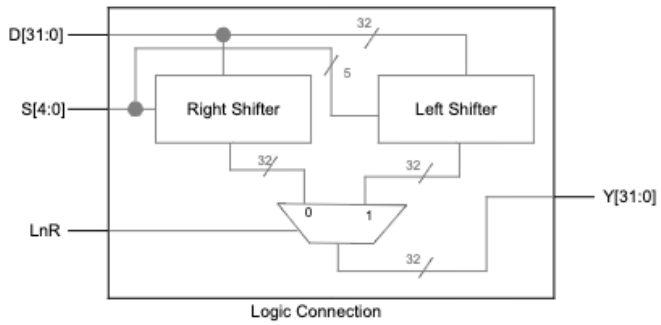
File: barrel_shifter.v

Implementation

```
// Right shifter
module SHIFT32_R(Y,D,S);
// output list
output [31:0] Y;
// input list
input [31:0] D;
input [4:0] S;

// Wires between the multiplexers
wire [31:0] wire0; // Shift 1-bit Y/N Result
wire [31:0] wire1; // Shift 2-bit Y/N Result
wire [31:0] wire2; // Shift 4-bit Y/N Result
wire [31:0] wire3; // Shift 8-bit Y/N Result

MUX32_2x1 mux_inst0(wire0, D, {1'b0,D[31:1]}, S[0]); // (Shift 1-bit <<1) Choose between D and shifted D
MUX32_2x1 mux_inst1(wire1, wire0, {2'b0,wire0[31:2]}, S[1]);// (Shift 2-bit <<2) Choose between D and shifted D
MUX32_2x1 mux_inst2(wire2, wire1, {4'b0,wire1[31:4]}, S[2]); // (Shift 4-bit <<4) Choose between D and shifted D
MUX32_2x1 mux_inst3(wire3, wire2, {8'b0,wire2[31:8]}, S[3]); // (Shift 8-bit <<8) Choose between D and shifted D
MUX32_2x1 mux_inst4(Y, wire3, {16'b0,wire3[31:16]}, S[4]); // (Shift 16-bit <<16) Choose between D and shifted D
endmodule
```

Barrel shifter – made from 1 left shifter, 1 right shifter, and 1 multiplexer

Design



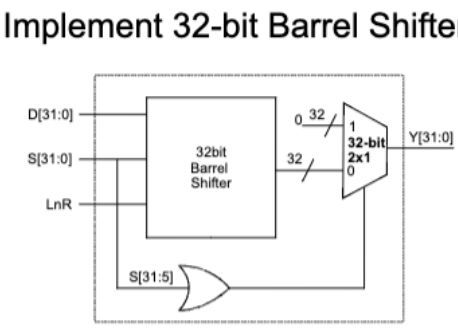
File: barrel_shifter.v

Implementation

```
// Shift with control L or R shift
module BARREL_SHIFTER32(Y,D,S, LnR);
// output list
output [31:0] Y;
// input list
input [31:0] D;
input [4:0] S;
input LnR;

wire [31:0] left_w; //result from LShift
wire [31:0] right_w; //result from RShift
SHIFT32_R r_shifter_inst(right_w, D, S);
SHIFT32_L l_shifter_inst(left_w, D, S);
MUX32_2x1 mux32_inst(Y, right_w, left_w, LnR);
endmodule
```

32-bit Shifter – Made from 1 barrel shifter, 1 32-bit 2x1 mux, and a 32-bit OR.

Design



File: barrel_shifter.v

Implementation

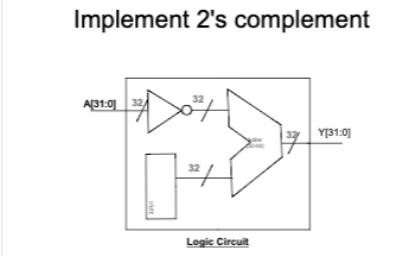
```
// 32-bit shift amount shifter
module SHIFT32(Y,D,S, LnR);
// output list
output [31:0] Y;
// input list
input [31:0] D;
input [31:0] S;
input LnR;

wire [31:0] shift_result_w; //Shifter result wire
wire or_w; // Or result wire
BARREL_SHIFTER32 barrelShifter32_inst(shift_result_w, D, S[4:0], LnR);
or or_inst(or_w, S[5], S[6], S[7], S[8], S[9], S[10], S[11], S[12], S[13],
        S[14], S[15], S[16], S[17], S[18], S[19], S[20], S[21], S[22],
        S[23], S[24], S[25], S[26], S[27], S[28], S[29], S[30], S[31]);
MUX32_2x1 mux32_inst(Y, shift_result_w, 32'h0, or_w);
endmodule
```

Test Bench

```
VSIM 11> run -all
#         1 <<          1 =          0 ==> got          0[PASSED]
# Run barrel shift TB
# 2147483648 <<          1 = 1073741824 ==> got 1073741824[PASSED]
# Run barrel shift TB
#         2 <<          2 =          0 ==> got          0[PASSED]
# Run barrel shift TB
#         4 <<          2 =          1 ==> got          1[PASSED]
# Run barrel shift TB
#         4 <<          3 =          0 ==> got          0[PASSED]
# Run barrel shift TB
#        24 <<          3 =          3 ==> got          3[PASSED]
# Run barrel shift TB
# 2147483648 <<          1 = 1073741824 ==> got 1073741824[PASSED]
# Run barrel shift TB
#         1 <<          5 =          0 ==> got          0[PASSED]
# Run barrel shift TB
# 2147483648 <<          1 = 1073741824 ==> got 1073741824[PASSED]
# Run barrel shift TB
#         2 <<         16 =          0 ==> got          0[PASSED]
# Run barrel shift TB
#         4 <<          2 =          1 ==> got          1[PASSED]
# Run barrel shift TB
#         4 <<          3 =          0 ==> got          0[PASSED]
# Run barrel shift TB
#        24 <<          3 =          3 ==> got          3[PASSED]
```

32-bit Two's Complement

Design – 32 NOT gates, register of 1's, and a ripple carry adder subtractor



Implementation

```
// 32-bit two's complement
module TWOSCOMP32(Y,A);
//output list
output [31:0] Y;
//input list
input [31:0] A;

// --- My work below ---
wire [31:0] not_w ;
wire empty;
reg addZero = 0;
reg [31:0] addOne = 1;

genvar i;
generate
for(i = 0; i < 32; i = i + 1) begin
        not not_inst(not_w[i], A[i]);
end
endgenerate
RC_ADD_SUB_32 rc_add_sub_32_inst(Y, empty, not_w, addOne, addZero);
endmodule
```

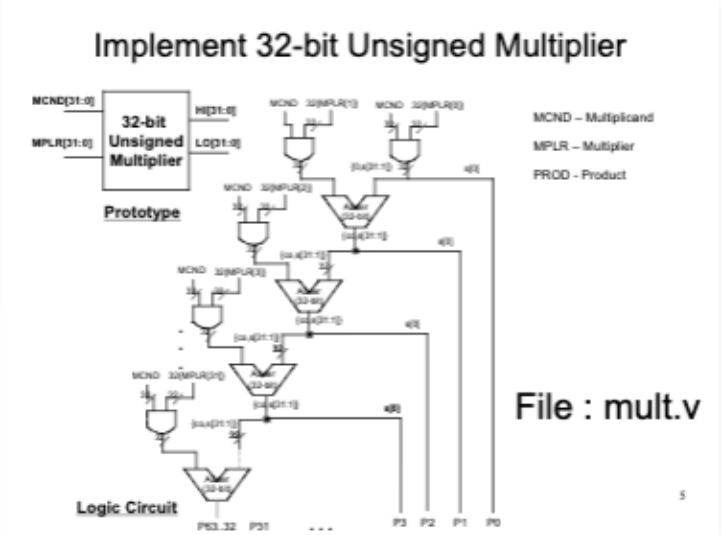
32-bit Two's Complement – consists of 2 32-bit two's complement

Implementation

```
// 64-bit two's complement
module TWOSCOMP64(Y,A);
//output list
output [63:0] Y;
//input list
input [63:0] A;

// --- My work below ---
wire[63:0] not_w;
wire empty;
reg add = 0;
reg [63:0] adding = 1;

genvar i;
generate
for(i = 0; i < 64; i = i + 1) begin
        not not_inst(not_w[i], A[i]);
end
endgenerate
RC_ADD_SUB_64 rc_add_sub_32_inst(Y, empty, not_w, adding, add);
endmodule
```

Unsigned Multiplier – 32 AND gates and 32 adders

Design

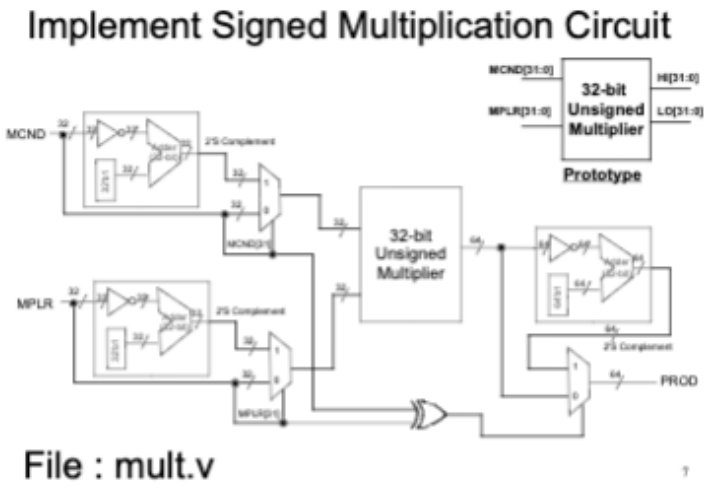


Implementation

```
module MULT32_U(HI, LO, A, B);
// output list
output [31:0] HI;
output [31:0] LO;
// input list
input [31:0] A;
input [31:0] B;

// Unsigned Multiplication: Multiplicand = A, Multiplier = B
// Unsigned multiplication's diagram is implemented below
wire [31:0] carry_out_w;
wire [31:0] remainder_w [31:0];
AND32_2x1 and32_inst_int(remainder_w[0], A, {32{B[0]}});
buf buf_1(carry_out_w[0], 1'b0); // Wipe out carry_out_w
buf buf_2(LO[0], remainder_w[0][0]); // LO[0] = operand_2_wire[0][0]
genvar i;
generate
for (i = 1; i < 32; i = i + 1) begin : mul_U_32_loop
        wire [31:0] operand_w;
        AND32_2x1 and32_inst(operand_w, A, {32{B[i]}});
        RC_ADD_SUB_32 adder32_inst(remainder_w[i], carry_out_w[i], operand_w,
                {carry_out_w[i - 1], {remainder_w[i - 1][31:1]}}, 1'b0);
        buf buf_inst(LO[i], remainder_w[i][0]);
end
endgenerate
// Store carry out and remainder in HI
BUF32x32 buff_inst_last(HI, {carry_out_w[31],
        {remainder_w[31][31:1]}}); // Located in logic32bit file
endmodule
```

Signed Multiplier

Design



Implementation

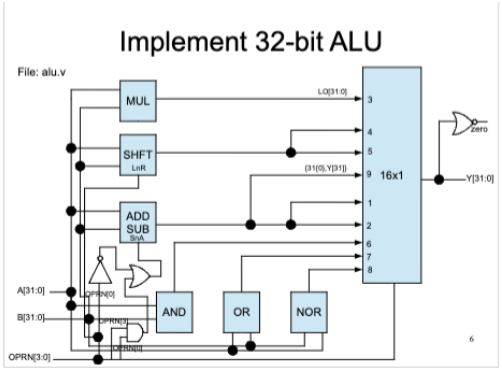
```
module MULT32(HI, LO, A, B);
// output list
output [31:0] HI;
output [31:0] LO;
// input list
input [31:0] A; // Multiplicand
input [31:0] B; // Multiplier
// Task 1: Create wires to store/transfer intemediate results
wire [31:0] signed_A_w; // Wire for signed A
wire [31:0] signed_B_w; // Wire for signed B
wire [31:0] choice_A_w; // Wire for signed A
wire [31:0] choice_B_w; // Wire for signed B
wire [63:0] unsigned_output_w; // Make a 64-bit wire for unsigned output
wire [63:0] twos_complement_w; // Make a 64-bit wire for signed output/2's complement of the output
wire [63:0] mux_output_w; // Make a 64-bit wire for the 64-bit multiplexer
wire xor_output_w; // Make a wire for the result of the xor gate's result

// Task 2: Calculate the sign of the output
xor xor_inst(xor_output_w, A[31], B[31]); // Calculate if the result is + or - based on the MSB's
// Task 3: Find 2's complement of A, B, and their product
TWOSCOMP32 twoscomp32_inst_1(signed_A_w, A); // Find the 2's complement of A and store it
MUX32_2x1 m1(choice_A_w, A, signed_A_w, A[31]); // Create a 32-bit 2x1 mux to choose signed or unsigned
TWOSCOMP32 twoscomp32_inst_2(signed_B_w, B); // Find the 2's complement of B and store it
MUX32_2x1 m2(choice_B_w, B, signed_B_w, B[31]); // Create a 32-bit 2x1 mux to choose signed or unsigned
MULT32_U multiplierU_inst(unsigned_output_w[63:32], unsigned_output_w[31:0], choice_A_w, choice_B_w); //
TWOSCOMP64 twoscomp64_inst(twos_complement_w, unsigned_output_w); // Find the 2's complement of the unsi
// Task4: Instantiate a 64-bit 2x1 mux. Choose between unsigned or signed result based on xor calculatio
MUX64_2x1 mux64(mux_output_w, unsigned_output_w, twos_complement_w, xor_output_w);
BUF32x32 buff_1(LO, mux_output_w[31:0]); // First portion of 64 bit mux [0, 31] are LO portion of the pr
BUF32x32 buff_2(HI, mux_output_w[63:32]); // Second portion of 64 bit mux [32, 63] are HI portion of the
endmodule
```

Test Bench Results

```
VSIM 20> run -all
# [         0 *         0 =         0 ,         0] ==> got         0,         0[PASSED]
# Run mult TB
# [         1 *         0 =         0 ,         0] ==> got         0,         0[PASSED]
# Run mult TB
# [         0 *         1 =         0 ,         0] ==> got         0,         0[PASSED]
# Run mult TB
# [         1 *         1 =         0 ,         1] ==> got         0,         1[PASSED]
# Run mult TB
# [         3 *         1 =         0 ,         3] ==> got         0,         3[PASSED]
# Run mult TB
# [2147483647 *         1 =         0 ,2147483647] ==> got         0,2147483647[PASSED]
# Run mult TB
# [4294967295 *         1 = 4294967295 ,4294967295] ==> got 4294967295,4294967295[PASSED]
# Run mult TB
# [         2 *         2 =         0 ,         4] ==> got         0,         4[PASSED]
# Run mult TB
# [         3 *         3 =         0 ,         9] ==> got         0,         9[PASSED]
# Run mult TB
```

ALU Design

Design



Implementation

Step 1: make the necessary wires

```
// Task 1: Create wires
wire [31:0] HI, LO; // For multiplication
wire [`DATA_INDEX_LIMIT:0] shift_result; // For shift
wire [`DATA_INDEX_LIMIT:0] add_sub_result; // For addition and subtraction
wire [31:0] and_result; // For AND operation
wire [31:0] or_result; // For OR operation
wire [31:0] nor_result; // For NOR operation
wire add_sub_co; // Carry out from addition and subtraction
wire SnA_or_w; // SnA or
wire SnA_not_w; // SnA not
wire SnA_and_w; // SnA and
```

Step 2: implement the operations

```
MULT32 mult32_inst(HI, LO, OP1, OP2); // Multiplcation
SHIFT32 shift32_inst(shift_result, OP1, OP2, OPRN[0]); // Shifting

// Addition / Subtraction
not inv_inst(SnA_not_w, OPRN[0]);
and and_inst(SnA_and_w, OPRN[0], OPRN[3]);
or or_inst(SnA_or_w, SnA_not_w, SnA_and_w);
RC_ADD_SUB_32 rc_add_sub_inst(add_sub_result, add_sub_co, OP1, OP2, SnA_or_w);

// 32-bit logical operations: AND, OR, NOR
AND32_2x1 and32_inst(.Y(and_result), .A(OP1), .B(OP2));
OR32_2x1 or32_inst(.Y(or_result), .A(OP1), .B(OP2));
NOR32_2x1 nor32_inst(.Y(nor_result), .A(OP1), .B(OP2));
```

Step 3: Choose the output with a mux

```
// Choose the result. Indices of mux determined by diagram. Set the output of mux to the output of the
MUX32_16x1 mux16x1_inst(.Y(OUT), .I0(32'h00000000), .I1(add_sub_result), .I2(add_sub_result),
        .I3(LO), .I4(shift_result), .I5(shift_result), .I6(and_result), .I7(or_result), .I8(nor_result),
        .I9({31'b0, add_sub_result[31]}), .I10(32'h00000000), .I11(32'h00000000), .I12(32'h00000000),
        .I13(32'h00000000), .I14(32'h00000000), .I15(32'h00000000), .S({OPRN[3:0]}));
```

Step 4: calculate zero flag

```
// Calculate the ZERO flag by NOR-ing the output bits
nor nor_Z_flag(ZERO, OUT[0], OUT[1], OUT[2], OUT[3], OUT[4], OUT[5], OUT[6], OUT[7], OUT[8], OUT[09], OUT[10], OUT[11], OUT[12],
        OUT[13], OUT[14], OUT[15], OUT[16], OUT[17], OUT[18], OUT[19], OUT[20], OUT[21], OUT[22], OUT[23], OUT[24],
        OUT[25], OUT[26], OUT[27], OUT[28], OUT[29], OUT[30], OUT[31]);
```

ALU Test Bench Results

```
VSIM 8> run -all
# [TEST] 15 + 5 = 20 , got 20. (Zero flag is = 0) ... [PASSED]
# [TEST] 15 - 5 = 10 , got 10. (Zero flag is = 0) ... [PASSED]
# [TEST] 5 * 3 = 15 , got 15. (Zero flag is = 0) ... [PASSED]
# [TEST] 8 >> 2 = 2 , got 2. (Zero flag is = 0) ... [PASSED]
# [TEST] 2 << 2 = 8 , got 8. (Zero flag is = 0) ... [PASSED]
# [TEST] 15 & 5 = 5 , got 5. (Zero flag is = 0) ... [PASSED]
# [TEST] 15 | 5 = 15 , got 15. (Zero flag is = 0) ... [PASSED]
# [TEST] 1 ~| 1 = 4294967294 , got 4294967294. (Zero flag is = 0) ... [PASSED]
# [TEST] 8 < 12 = 1 , got 1. (Zero flag is = 0) ... [PASSED]
# [TEST] 15 - 15 = 0 , got 0. (Zero flag is = 1) ... [PASSED]
#
#       Total number of tests          10
#       Total number of pass           10
#
```

Conclusion

These are essential components in making an ALU for our instruction set. All our operations can be implemented with these components. Many of these components will also be used in the register file as well.

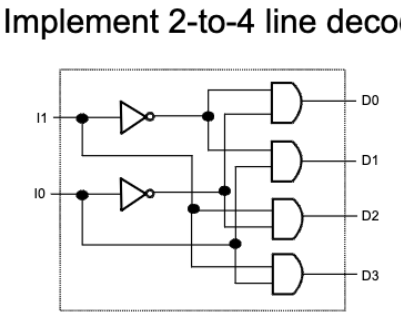
VII. Register File Design and Testing

Below are the designs and test results of the essential components involved in the register file:

Components required in the ALU – refer to ALU

2x4 decoder

Design

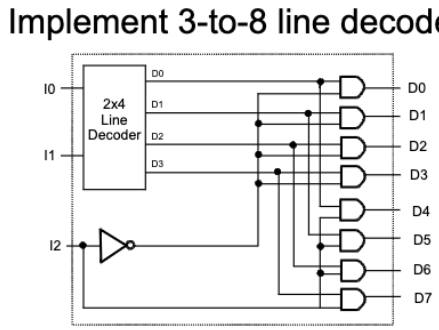


Implementation

```
// 2x4 Line decoder
module DECODER_2x4(D,I);
// output
output [3:0] D;
// input
input [1:0] I;

// --- My work below ---
wire [1:0] inv_I;
// The 2 NOT gates
not not_0(inv_I[0], I[0]); // NOT I0
not not_1(inv_I[1], I[1]); // NOT I1
// The 4 AND gates
and and_0(D[0], inv_I[0], inv_I[1]); // D0 = ~I1 AND ~I0
and and_1(D[1], inv_I[1], I[0]);  // D1 = ~I1 AND I0
and and_2(D[2], inv_I[0], I[1]); // D2 = I1 AND ~I0
and and_3(D[3], I[0], I[1]); // D3 = I1 AND I0
endmodule
```

3x8 decoder

Design

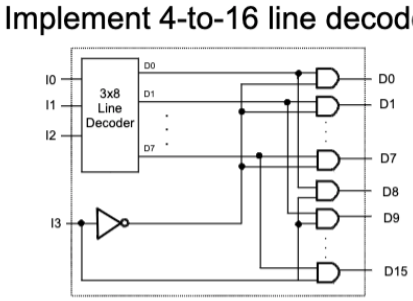


Implementation

```
// 3x8 Line decoder
module DECODER_3x8(D,I);
// output
output [7:0] D;
// input
input [2:0] I;

// --- My work below ---
wire [4:0] inv_I;
not not_inst(inv_I[4], I[2]);
DECODER_2x4 decoder2x4_inst(inv_I[3:0], I[1:0]);
genvar i;
generate
for(i = 0; i < 4; i = i + 1) begin : decoder2x4_loop
        and and_inst0(D[i], inv_I[i], inv_I[4]);
        and and_inst1(D[i + 4], inv_I[i], I[2]);
end
endgenerate
endmodule
```

4x16 decoder

Design

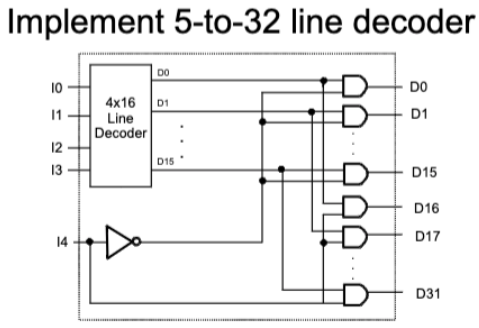


Implementation

```
// 4x16 Line decoder
module DECODER_4x16(D,I);
// output
output [15:0] D;
// input
input [3:0] I;

// --- My work below ---
wire [8:0] inv_I;
DECODER_3x8 decoder3x8_inst(inv_I[7:0], I[2:0]);
not not_inst(inv_I[8], I[3]);
genvar i;
generate
for(i = 0; i < 8; i = i + 1) begin : decoder3x8_loop
        and and_inst0(D[i], inv_I[i], inv_I[8]); // D[0] to D[
        and and_inst1(D[i + 8], inv_I[i], I[3]); // D[8] to D[
end
endgenerate
endmodule
```

5x32 decoder

Design

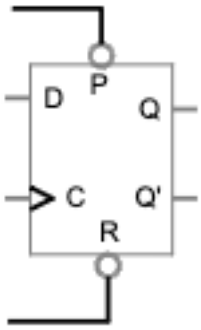


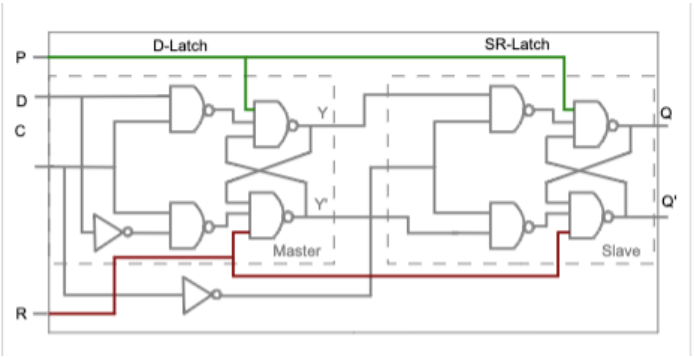
Implementation

```
// 5x32 Line decoder
module DECODER_5x32(D,I);
// output
output [31:0] D;
// input
input [4:0] I;

// --- My work below ---
wire [16:0] inv_I;
DECODER_4x16 decoder4x16_inst(inv_I[15:0], I[3:0]);
not not_inst(inv_I[16], I[4]);
genvar i;
generate
for(i = 0; i < 16; i = i + 1) begin : decoder4x16_loop
        and and_inst0(D[i], inv_I[i], inv_I[16]);
        and and_inst1(D[i + 16], inv_I[i], I[4]);
end
endgenerate
endmodule
```

D-Flip-Flop

Design



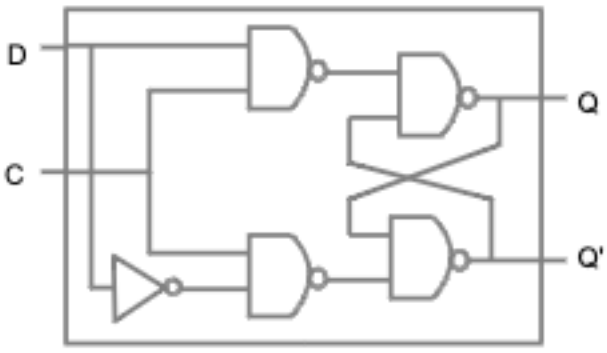


Implementation

```
// 1 bit flipflop +ve edge,
// Preset on nP=0, nR=1, reset on nP=1, nR=0;
// Undefined nP=0, nR=0
// normal operation nP=1, nR=1
module D_FF(Q, Qbar, D, C, nP, nR);
input D, C;
input nP, nR;
output Q,Qbar;

// --- My work below ---
wire Y, Ybar, inv_C;
not not_inst(inv_C, C);
D_LATCH dlatch_inst(.Q(Y), .Qbar(Ybar), .D(D), .C(in
SR_LATCH srlatch_inst(.Q(Q), .Qbar(Qbar), .S(Y), .R(
endmodule
```

D-Latch

Design

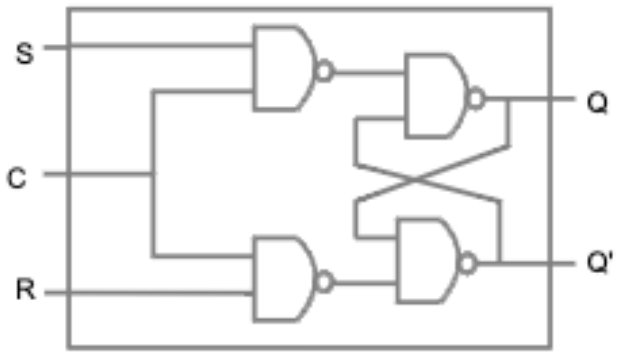


Implementation

```
// 1 bit D latch
// Preset on nP=0, nR=1, reset on nP=1, nR=0;
// Undefined nP=0, nR=0
// normal operation nP=1, nR=1
module D_LATCH(Q, Qbar, D, C, nP, nR);
input D, C;
input nP, nR;
output Q,Qbar;

// --- My work below ---
wire inv_D_w, and_1_w, and_2_w;
not not_inst(inv_D_w, D);
nand nand_inst1(and_1_w, D, C);
nand nand_inst2(and_2_w, inv_D_w, C);
nand nand_inst3(Q, Qbar, and_1_w, nP); // nR
nand nand_inst4(Qbar, Q, and_2_w, nR); // nP
endmodule
```

SR-Latch

Design

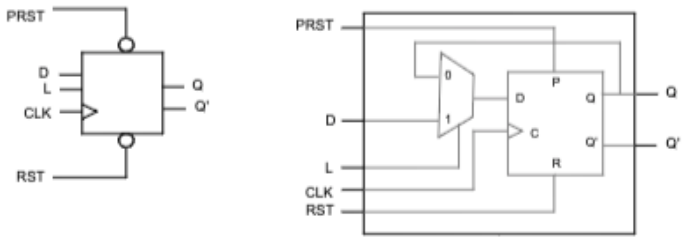


Implementation

```
// 1 bit SR latch
// Preset on nP=0, nR=1, reset on nP=1, nR=0;
// Undefined nP=0, nR=0
// normal operation nP=1, nR=1
module SR_LATCH(Q,Qbar, S, R, C, nP, nR);
input S, R, C;
input nP, nR;
output Q,Qbar;

// --- My work below ---
wire and_1_w, and_2_w;
nand nand_inst1(and_1_w, S, C);
nand nand_inst2(and_2_w, R, C);
nand nand_inst3(Q, Qbar, and_1_w, nP);
nand nand_inst4(Qbar, Q, and_2_w, nR);
```

Register 1-bit

Design

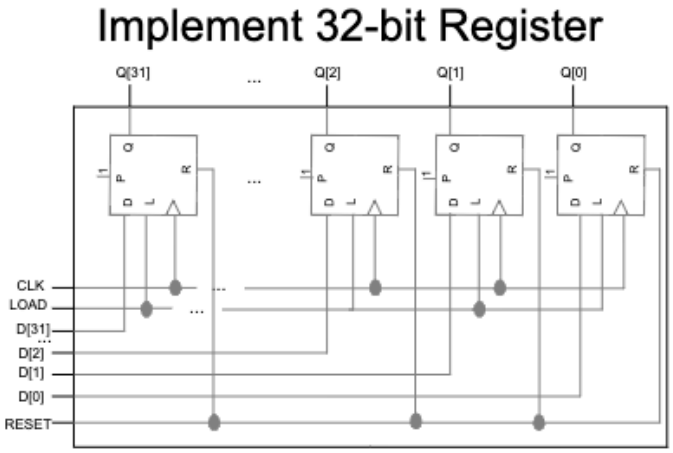


Implementation

```
// 1 bit register +ve edge,
// Preset on nP=0, nR=1, reset on nP=1, nR=0;
// Undefined nP=0, nR=0
// normal operation nP=1, nR=1
module REG1(Q, Qbar, D, L, C, nP, nR);
input D, C, L;
input nP, nR;
output Q,Qbar;

// --- My work below ---
wire mux_w;
MUX1_2x1 mux_inst(mux_w, Q, D, L);
D_FF dff(.Q(Q), .Qbar(Qbar), .D(mux_w), .C(C), .nP(nP), .nR(nR));
endmodule
```

Register 32-bit

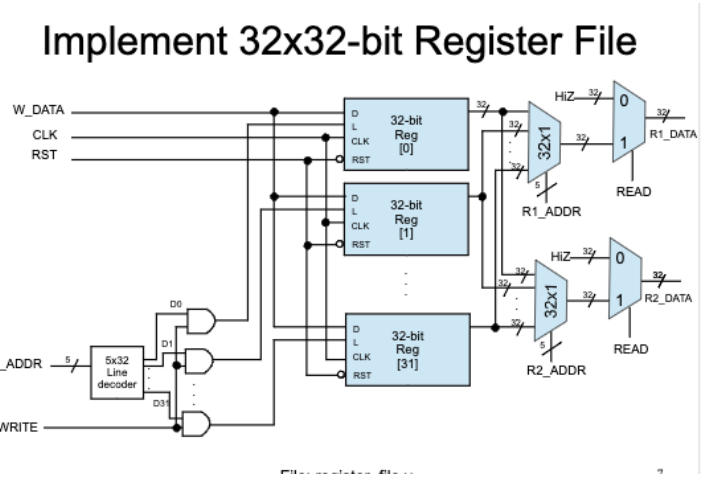
Design



Implementation

```
// 32-bit registere +ve edge, Reset on RESET=0
module REG32(Q, D, LOAD, CLK, RESET);
output [31:0] Q;
input CLK, LOAD;
input [31:0] D;
input RESET;

// --- My work below ---
genvar i;
generate
for(i = 0; i < 32; i = i + 1)begin : reg32_loop
        wire Qbar;
        REG1 reg1_inst(.Q(Q[i]), .Qbar(Qbar), .D(D[i]),
        .L(LOAD), .C(CLK), .nP(1'b1), .nR(RESET));
end
endgenerate
endmodule
```

Register File Design



Register File Implementation

```
// --- My work below ---
wire inv_RST; // Stores the inverse of RST
wire [31:0] decoder_result_w, and_result_w, reg_1_data, reg_2_data;
wire [31:0] REGISTERS [31:0]; // wire fo register file

DECODER_5x32 decoder5x32_inst(decoder_result_w, ADDR_W); // Decoder for the ac
not not_inst(inv_RST, RST); // Calculate the inverse of RST

genvar i;
for (i = 0; i < 32; i = i + 1) begin : reg32_gen_loop // Generate 32 32-bit re
        and and_inst(and_result_w[i], decoder_result_w[i], WRITE); // AND each
        REG32 reg32_inst(REGISTERS[i], DATA_W, and_result_w[i], CLK, inv_RST);
end
// The 4 multiplexers in the diagram
 // Access data at ADDR_R1
MUX32_32x1 mux_inst_1(reg_1_data, REGISTERS[0], REGISTERS[1], REGISTERS[2], RE
        REGISTERS[8], REGISTERS[9],  REGISTERS[10], REGISTERS[11], REGISTERS[1
        REGISTERS[17], REGISTERS[18], REGISTERS[19], REGISTERS[20], REGISTERS[
        REGISTERS[26], REGISTERS[27], REGISTERS[28], REGISTERS[29], REGISTERS[
// Access data at ADDR_R2
MUX32_32x1 mux_inst_2(reg_2_data, REGISTERS[0], REGISTERS[1], REGISTERS[2], RE
        REGISTERS[8], REGISTERS[9],  REGISTERS[10], REGISTERS[11], REGISTERS[1
        REGISTERS[17], REGISTERS[18], REGISTERS[19], REGISTERS[20], REGISTERS[
        REGISTERS[26], REGISTERS[27], REGISTERS[28], REGISTERS[29], REGISTERS[
MUX32_2x1 mux_inst_3(DATA_R1, 32'hzzzzzzzz, reg_1_data, READ); // Return data
MUX32_2x1 mux_inst_4(DATA_R2, 32'hzzzzzzzz, reg_2_data, READ); // Return data

endmodule
```

Register File Test Bench Results

```
VSIM 29> run -all
#
#       Total number of tests          32
#       Total number of pass           32
#
```

Conclusion

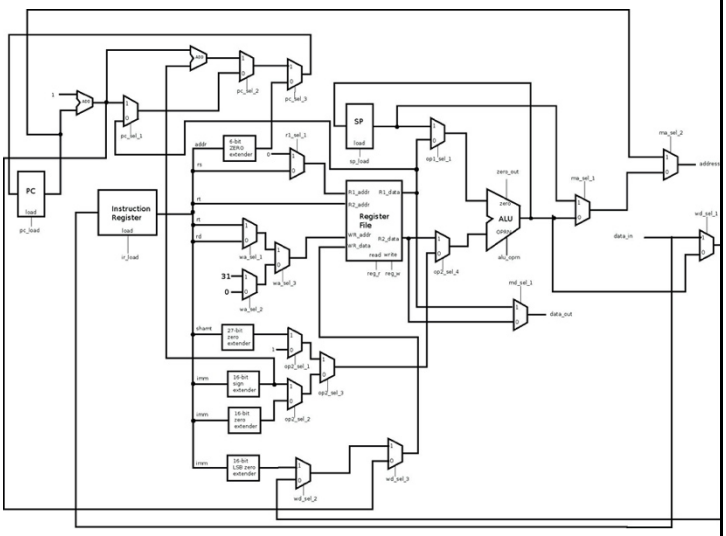
These are essential components in making a register file for our instruction set. The register file will behave according to the inputs from its user (READ and WRITE signals), sent by the control unit. 1-bit registers are also used in data path and control unit.

VIII. Datapath

In order to implement the data path for our instruction set, we must analyze the design diagram and count all the components involved. First, instantiate all the components which are not multiplexers, such as

ALU, instruction register, program counter, stack pointer, the two adders, and the register file. Then proceed to instantiate all the multiplexers and connect their corresponding wires. For each multiplexer, create a wire. For each output from other components such as ALU, register file, pc register value added one, and others, create additional intermediate wires. Trace the diagram and carefully connect all the components.

Design



Implementation

Step 1: Create the corresponding wires. Let there be an output wire for each component. For simplicity, I have copied the table in lecture eleven. Not all the wires are used. Create intermediate wires – listed in the second portion of the picture.

```
// Copy of the table in lecture 11s
// Wires involved in datapath
wire [31:0] pc_load;
wire [31:0] pc_sel_1;
wire [31:0] pc_sel_2;
wire [31:0] pc_sel_3;
wire [31:0] mem_r;
wire [31:0] mem_w;
wire [31:0] r1_sel_1;
wire [31:0] reg_r;
wire [31:0] reg_w;
wire [31:0] wa_sel_1;
wire [31:0] wa_sel_2;
wire [31:0] wa_sel_3;
wire [31:0] wd_sel_1;
wire [31:0] wd_sel_2;
wire [31:0] wd_sel_3;
wire [31:0] sp_load;
wire [31:0] op1_sel_1;
wire [31:0] op2_sel_1;
wire [31:0] op2_sel_2;
wire [31:0] op2_sel_3;
wire [31:0] op2_sel_4;
wire [31:0] alu_oprn;
wire [31:0] ma_sel_1;
wire [31:0] dmem_r;
wire [31:0] dmem_w;
wire [31:0] md_sel_1;
// Additional intermediate wires
wire [31:0] OUT;
wire [31:0] program_ctr;
wire [31:0] stack_ptr;
wire [31:0] pc_immediate;
wire [31:0] pc_plus_1;
wire [31:0] R1_data,R2_data;
```

Step 2: Instantiate the additional components (not multiplexers) and connect. These include ALU, register file, program counter, stack pointer, and the two adders.

```
// ***** Define the additional components first and connect *****
// ALU
ALU alu_inst(.OUT(OUT), .ZERO(ZERO), .OP1(op1_sel_1), .OP2(op2_sel_4), .OPRN( CTRL[27:22]  ) );

// Register File
REGISTER_FILE_32x32 REGISTERS(.DATA_R1(R1_data), .DATA_R2(R2_data),
        .ADDR_R1(r1_sel_1[4:0] ), .ADDR_R2(INSTRUCTION[20:16]) ,
        .DATA_W(wd_sel_3), .ADDR_W(wa_sel_3[4:0]), .READ(CTRL[8]),
        .WRITE(CTRL[9]), .CLK(CLK), .RST(RST) );

// Program Counter
REG32 pc_reg(.Q(program_ctr ), .D( pc_sel_3  ), .LOAD( CTRL[0] ), .CLK(CLK), .RESET(RST));

// Stack Pointer
REG32 sp_reg(.Q(stack_ptr), .D(OUT), .LOAD( CTRL[16] ), .CLK(CLK), .RESET(RST));

// Instruction Register
REG32 instruction_reg(.Q(INSTRUCTION), .D(DATA_IN), .LOAD(CTRL[4]), .CLK(CLK), .RESET(RST));

// Adder 1
RC_ADD_SUB_32 adder_1(.Y(pc_plus_1) , .CO(blank_w1) , .A(program_ctr), .B(32'b1), .SnA(1'b0));

// Adder 2
RC_ADD_SUB_32 adder_2(.Y(pc_immediate) , .CO(blank_w2) , .A(pc_plus_1), .B( {{16{INSTRUCTION[15]}} ,INSTRUCTION[15:0]}), .SnA(1'b0));
```

Step 3: Instantiate the multiplexers and make the connections according to the design diagram.

```
// *** Connections for each mux in datapath ***

// pc_load - Done in CU

// pc_sel_1
MUX32_2x1 mux_pc_sel_1_inst(.Y(pc_sel_1) , .I0(R1_data), .I1(pc_plus_1), .S(CTRL[1]));
// pc_sel_2
MUX32_2x1 mux_pc_sel_2_inst(.Y(pc_sel_2) , .I0(pc_sel_1), .I1(pc_immediate), .S(CTRL[2]));
// pc_sel_3
MUX32_2x1 mux_pc_sel_3_inst(.Y(pc_sel_3) , .I0({{6'b0} ,INSTRUCTION[25:0]}), .I1(pc_sel_2), .S(CTRL[3]));
// mem_r - Done in CU

// mem_w - Done in CU

// r1_sel_1
MUX32_2x1 mux_r1_sel_1_inst(.Y(r1_sel_1) , .I0({{27{1'b0}}, INSTRUCTION[25:21]}), .I1(32'b00000), .S(CTRL[7]));
// reg_r - Done in CU

// reg_w - Done in CU

// wa_sel_1
MUX32_2x1 mux_wa_sel_1_inst(.Y(wa_sel_1), .I0({{27{1'b0}}, INSTRUCTION[15:11]}), .I1({{27{1'b0}}, INSTRUCTION[20:16]}), .S(CTRL[10]));
// wa_sel_2
MUX32_2x1 mux_wa_sel_2_inst(.Y(wa_sel_2), .I0(32'h00000000), .I1(32'b11111), .S(CTRL[11]));
// wa_sel_3
MUX32_2x1 mux_wa_sel_3_inst(.Y(wa_sel_3), .I0(wa_sel_2), .I1(wa_sel_1), .S(CTRL[12]));
// wd_sel_1
MUX32_2x1 mux_wd_sel_1_inst(.Y(wd_sel_1), .I0(OUT), .I1(DATA_IN), .S(CTRL[13]));
// wd_sel_2
MUX32_2x1 mux_wd_sel_2_inst(.Y(wd_sel_2), .I0(wd_sel_1), .I1({INSTRUCTION[15:0],{16'b0}}), .S(CTRL[14]));
// wd_sel_3
MUX32_2x1 wd_sel_3_inst(.Y(wd_sel_3), .I0(pc_plus_1), .I1(wd_sel_2), .S(CTRL[15]));
// sp_load - Done in CU

// op1_sel_1
MUX32_2x1 mux_op1_sel_1_inst(.Y(op1_sel_1) , .I0(R1_data), .I1(stack_ptr), .S(CTRL[17]));
// op2_sel_1
MUX32_2x1 mux_op2_sel_1_inst(.Y(op2_sel_1) , .I0(32'b1), .I1({{27'b0} ,INSTRUCTION[10:6]}), .S(CTRL[18]));
// op2_sel_2
MUX32_2x1 mux_op2_sel_2_inst(.Y(op2_sel_2) , .I0({{16'b0} ,INSTRUCTION[15:0]}), .I1({{16{INSTRUCTION[15]}} ,INSTRUCTION[15:0]}), .S(  CTRL[19]));
// op2_sel_3
MUX32_2x1 mux_op2_sel_3_inst(.Y(op2_sel_3) , .I0(op2_sel_2), .I1(op2_sel_1), .S(CTRL[20]));
// op2_sel_4
MUX32_2x1 mux_op2_sel_4_inst(.Y(op2_sel_4) , .I0(op2_sel_3), .I1(R2_data), .S(CTRL[21]));
// alu_oprn - Done in CU

// ma_sel_1
MUX32_2x1 mux_ma_sel_1_inst(.Y(ma_sel_1) , .I0(OUT), .I1(stack_ptr), .S(CTRL[28]));
// ma_sel_2
MUX32_2x1 mux_ma_sel_2_inst(.Y(ADDR) , .I0(ma_sel_1), .I1(program_ctr), .S(CTRL[29]));
// dmem_r - Done in CU

// dmem_w - Done in CU

// md_sel_1
MUX32_2x1 mux_md_sel_1_inst(.Y(DATA_OUT) , .I0(R2_data), .I1(R1_data), .S(CTRL[30]));
```

IX. Conclusion

By completing this project, I have demonstrated how an ALU is designed at the gate level with all of its components and how registers are designed at the gate level with their various components. By doing so, we can implement a processor for our instruction set that uses these components to perform various operations. By implementing ALU and register file at the gate level, I have demonstrated my thorough understanding of the design and organization of the ALU and register file and the design of their various components and logical subcomponents by translating my understanding of the various schematics above in Verilog