ELEC 374 Project Phase 1 Report

Group 42

- Alex O'Neill, 20043101, 16amon
- Jared McGrath, 12345678, 16mickyd

Table of Contents

1. Design Overview

An overview of our design, and testing infrastructure

2. Testbench Waveforms

Testbenches, including results

3. Code

All code, and perhaps a short explanation

File: hdl/adder_subtractor.v

```
module adder_subtractor(
    input [31:0] a,
```

```
    input [31:0] b,
    output [31:0] sum,
    input sub, // 0 = Add, 1 = Subtract
    output c_out // Carry out (overflow detection)
);
    wire [31:0] b_in;
    assign b_in = b ^ {32{sub}};

    // Inner adder used by the adder/subtractor is a CLA (two level, with one RCA
to connect the 16-bit adders together)
    carry_lookahead_adder #( .BITS16(2) ) _cla ( .a(a), .b(b_in), .sum(sum),
.c_in(sub), .c_out(c_out) );

endmodule


`timescale 1ns/100ps
module adder_subtractor_test;

    reg [31:0] a, b;
    reg c_in, sub;
    wire [31:0] sum;
    wire c_out;

    integer i;

    adder_subtractor target ( .a(a), .b(b), .sum(sum), .sub(sub), .c_out(c_out) );

    initial begin
        // Addition
        sub <= 1'b0;
        for (i = 0; i < 100; i = i + 1) begin
            a <= $random;
            b <= $random;
            #1 $display("Test | add %d + %d | %d | %d", a, b, a + b, sum);
        end

        // Subtraction
        sub <= 1'b1;
        for (i = 0; i < 100; i = i + 1) begin
            a <= $random;
            b <= $random;
            #1 $display("Test | sub %d - %d | %d | %d", a, b, a - b, sum);
        end

        $finish;
    end
endmodule
```

File: hdl/alu.v

```
module alu(
    input [31:0] a,
    input [31:0] b,
    input [11:0] select, // {alu_not, alu_neg, alu_div, alu_mul, alu_or, alu_and,
alu_rol, alu_ror, alu_shl, alu_shr, alu_sub, alu_add}
    output reg [31:0] z, // Outputs for all other instructions
    output reg [31:0] hi, // Outputs for div, mul
    output reg [31:0] lo
);
    // ALU Operations (by select index)
    // 0 = Add
    // 1 = Sub
    // 2 = Shift Right
    // 3 = Shift Left
    // 4 = Rotate Right
    // 5 = Rotate Left
    // 6 = And
    // 7 = Or
    // 8 = Multiply
    // 9 = Divide
    // A = Negate
    // B = Not

    wire [31:0] z_add_sub, z_shift_right, z_shift_left, z_and, z_or, z_not;
    // pre-MUX outputs for mul/div
    wire [31:0] hi_mul, lo_mul, hi_div, lo_div;

    // ALU Operations

    // Add / Subtract / Negate
    wire add_sub_c_out; // todo: overflow flag from carry out?
    adder_subtractor add_sub ( .a(select[10] ? 32'b0 : a), .b(select[10] ? a : b),
.sum(z_add_sub), .sub(select[1] | select[10]), .c_out(add_sub_c_out) );

    // Shift / Rotate
    right_shift_32b _shr ( .in(a), .shift(b), .out(z_shift_right),
.is_rotate(select[4]) );
    left_shift_32b  _shl ( .in(a), .shift(b), .out(z_shift_left),
.is_rotate(select[5]) );
    // todo: rotate right
    // todo: rotate left

    assign z_and = a & b; // and
    assign z_or = a | b; // or

    // Multiplication
    booth_bit_pair_multiplier mul ( .multiplicand(a), .multiplier(b),
.product({hi_mul, lo_mul}) );

    // Division
    array_div #( .BITS(32) ) div ( .dividend(a), .divisor(b), .quotient(lo_div),
.remainder(hi_div) );
```

```
    assign z_not = ~a; // not

    // Multiplex the outputs together
    always @(*) begin
        case (select)
            12'b000000000001 : z = z_add_sub;
            12'b000000000010 : z = z_add_sub;
            12'b000000000100 : z = z_shift_right;
            12'b000000001000 : z = z_shift_left;
            12'b000000010000 : z = z_shift_right;
            12'b000000100000 : z = z_shift_left;
            12'b000001000000 : z = z_and;
            12'b000010000000 : z = z_or;
            // Multiply / Divide output to hi/lo
            12'b010000000000 : z = z_add_sub;
            12'b100000000000 : z = z_not;
            default : z = 32'b0;
        endcase
    end

    always @(*) begin
        case (select)
            12'b000100000000 : {hi, lo} <= {hi_mul, lo_mul};
            12'b001000000000 : {hi, lo} <= {hi_div, lo_div};
            default : {hi, lo} <= 64'b0;
        endcase
    end

endmodule


`timescale 1ns/100ps
module alu_test;

    reg [31:0] a, b;
    reg [11:0] select;
    wire [31:0] z, hi, lo;
    wire signed [31:0] sz;

    assign sz = z; // For reading signed outputs

    alu _alu ( .a(a), .b(b), .select(select), .z(z), .hi(hi), .lo(lo) );

    initial begin

        a <= 32'h7C; // 124
        b <= 32'h7; // 7

        select <= 12'b000000000001; // Add
        #1 $display("Test | add | 124 + 7 = 131 | %0d + %0d = %0d", a, b, z);

        select <= 12'b000000000010; // Subtract
        #1 $display("Test | sub | 124 - 7 = 117 | %0d - %0d = %0d", a, b, z);
```

```
        select <= 12'b000000000100; // Shift Right
        #1 $display("Test | shift right | 0000007c >> 00000007 = 00000000 | %h >>
%h = %h", a, b, z);

        select <= 12'b000000001000; // Shift Left
        #1 $display("Test | shift left | 0000007c << 00000007 = 00003e00 | %h <<
%h = %h", a, b, z);

        select <= 12'b000000010000; // Rotate Right
        #1 $display("Test | rotate right | 0000007c >>R 00000007 = f8000000 | %h
>>R %h = %h", a, b, z);

        select <= 12'b000000100000; // Rotate Left
        #1 $display("Test | rotate left | 0000007c R<< 00000007 = 00003e00 | %h
R<< %h = %h", a, b, z);

        select <= 12'b000001000000; // And
        #1 $display("Test | and | 0000007c & 00000007 = 00000004 | %h & %h = %h",
a, b, z);

        select <= 12'b000010000000; // Or
        #1 $display("Test | or | 0000007c or 00000007 = 0000007f | %h or %h = %h",
a, b, z);

        select <= 12'b000100000000; // Multiply
        #1 $display("Test | mul | 124 * 7 = (lo 868, hi 0) | %0d * %0d = (lo %0d,
hi %0d)", a, b, lo, hi);

        select <= 12'b001000000000; // Divide
        #1 $display("Test | div | 124 / 7 = (lo 17, hi 5) | %0d / %0d = (lo %0d,
hi %0d)", a, b, lo, hi);

        select <= 12'b010000000000; // Negate
        #1 $display("Test | neg | -(124) = -124 | -(%0d) = %0d", a, sz);

        select <= 12'b100000000000; // Not
        #1 $display("Test | not | ~0000007c = ffffff83 | ~%h = %h", a, z);

        $finish;

    end

endmodule
```

File: hdl/array_div.v

```
module array_div #(
    parameter BITS = 32
) (
    input [BITS - 1:0] dividend,
    input [BITS - 1:0] divisor,
    output [BITS - 1:0] quotient,
```

```
    output [BITS - 1:0] remainder
);

    // Signed complement stuff
    wire [BITS - 1:0] dividend_comp, divisor_comp, quotient_comp, remainder_comp,
dividend_u, divisor_u, quotient_u, remainder_u;
    // q_pos indicates quotient should be positive, r_pos indicates remainder
should be positive
    wire q_pos, r_pos;

    // Internal mode, sum, and b_out lines (between layers)
    wire [BITS - 1:-1] layer_mode_out;
    wire [BITS - 1:0] layer_sum [-1:BITS - 1];
    // layer_b_out is not really necessary. but keeping it true to the original
circuit diagram
    wire [BITS - 1:0] layer_b_out [-1:BITS - 1];
    // Remainder sum
    wire [BITS - 1:0] r_sum;
    // Not needed, but it makes Quartus happy
    wire r_c_out;

    // Instantiate signed_compliment
    signed_compliment #( .BITS(BITS) ) sc_dividend ( .in(dividend),
.out(dividend_comp) );
    signed_compliment #( .BITS(BITS) ) sc_divisor ( .in(divisor),
.out(divisor_comp) );
    // Assign the correct (unsigned) versions of dividend/divisor
    assign dividend_u = dividend[BITS - 1] ? dividend_comp : dividend;
    assign divisor_u = divisor[BITS - 1] ? divisor_comp : divisor;
    // Expression derived from C division/remainder behaviour
    assign q_pos = ~(dividend[BITS - 1] ^ divisor[BITS - 1]);
    assign r_pos = ~dividend[BITS - 1];

    // Assign constant 1 to first layer's mode_in
    assign layer_mode_out[-1] = 1'b1;
    // a input to most significant 31 cellsin first layer are 0, followed by msb
of dividend
    assign layer_sum[-1] = { {(BITS - 1){1'b0}}, dividend_u[BITS - 1] };
    // making the part-select explicit so there's no ambiguity
    assign layer_b_out[-1] = divisor_u;

    genvar layer, node;
    generate
        for (layer = 0; layer < BITS; layer = layer + 1) begin : gen_div_layer
            // Declare internal wires to carry mode_in/out and c_in/out between
cells
            wire [BITS:0] mode_in;
            wire [BITS:0] carry_in;
            // Also declare sum, so we can manipulate before assigning to
layer_sum
            wire [BITS - 1:0] sum;

            // Assign input mode for the first cell in layer
            assign mode_in[BITS] = layer_mode_out[layer - 1];
```

```
            // Assign last mode to last carry
            assign carry_in[0] = mode_in[0];

            div_cell dc [BITS - 1:0] (
                    .a( layer_sum[layer-1] ),
                    .b_in( layer_b_out[layer-1] ),
                    .mode_in( mode_in[BITS:1] ),
                    .mode_out( mode_in[BITS - 1:0] ),
                    .c_in( carry_in[BITS - 1:0] ),
                    .c_out( carry_in[BITS:1] ),
                    .b_out( layer_b_out[layer] ),
                    .sum( sum )
                );

            // Make assignments to set up next layer
            // This concatenation provides the shifting on all except the last
layer
            assign layer_sum[layer] = layer == BITS - 1 ? sum : {sum[BITS - 2:0],
dividend_u[(BITS - 2) - layer]};
            assign layer_mode_out[layer] = carry_in[BITS - 1];
            // Assign quotient here
            assign quotient_u[(BITS - 1) - layer] = carry_in[BITS - 1];
        end
    endgenerate

    // generate remainder
    ripple_carry_adder #( .BITS(BITS) ) _rca ( .a(layer_sum[BITS - 1]),
.b(layer_b_out[BITS - 1]), .sum(r_sum), .c_in(1'b0), .c_out(r_c_out) );
    assign remainder_u = quotient_u[0] ? layer_sum[BITS - 1] : r_sum;

    // Handle complements of outputs now
    signed_compliment #( .BITS(BITS) ) sc_quotient ( .in(quotient_u),
.out(quotient_comp) );
    signed_compliment #( .BITS(BITS) ) sc_remainder ( .in(remainder_u),
.out(remainder_comp) );

    // Assign final outputs based on q_pos, r_pos
    assign quotient = q_pos ? quotient_u : quotient_comp;
    assign remainder = r_pos ? remainder_u : remainder_comp;

endmodule

/**
 * div_cell
 * Represents a single 'cell' in the division array.
 * This is essentially a full adder + XOR. Also, b_in -> b_out and mode_in ->
mode_out
 * are just pass throughs. They maintain the 'structure' of the circuit in
 * Computer_Arithmetic slide 54, but provide no benefit over the alternative.
 */
module div_cell(
    input a,
    input b_in,
    output b_out,
```

```
    input mode_in,
    output mode_out,
    input c_in,
    output c_out,
    output sum
);

    // instantiate full adder
    full_adder _fa ( .a(b_in ^ mode_in), .b(a), .sum(sum), .c_in(c_in),
.c_out(c_out) );

    // other assignments (for propagation)
    assign mode_out = mode_in;
    assign b_out = b_in;

endmodule

/**
 * Testbench
 * Lots of test cases because division is hard
 */
`timescale 1ns/100ps
module array_div_test;

    reg signed [31:0] a, b;
    wire signed [31:0] quotient, remainder;

    integer i;

    array_div #( .BITS(32) ) _div ( .dividend(a[31:0]), .divisor(b[31:0]),
.quotient(quotient), .remainder(remainder) );

    initial begin
        // Regressions
        a <= 97;
        b <= 7;
        #1 $display("Test | divide %0d / %0d | %0d, %0d | %0d, %0d", a, b, a / b,
a % b, quotient, remainder);

        for (i = 0; i < 100; i = i + 1) begin
            a <= $random % 1000000;
            b <= $random % 1000;
            if (b != 0)
                #1 $display("Test | divide %0d / %0d | %0d, %0d | %0d, %0d", a, b,
a / b, a % b, quotient, remainder);
        end

        $finish;
    end
endmodule
```

File: hdl/booth_bit_pair_multiplier.v

```
module booth_bit_pair_multiplier(
    input [31:0] multiplicand,
    input [31:0] multiplier,
    output [63:0] product
);
    // Booth Recoding
    // Bit n | Bit n-1 | Multiplier (ones, sign)
    // 0     | 0       | 0   (0, ?)
    // 0     | 1       | +1  (1, 1)
    // 1     | 0       | -1  (1, 0)
    // 1     | 1       | 0   (0, ?)
    // Bit -1 is implicitly 0

    wire [31:0] booth_ones, booth_signs;

    assign booth_ones = multiplier[31:0] ^ {multiplier[30:0], 1'b0}; // bit n !=
bit n-1
    assign booth_signs = {multiplier[30:0], 1'b0}; // bit n-1

    // Bit-Pair Recoding
    // Examine pairs of booth ones, and perform multiplications by -2, -1, 0, 1 or
2
    // This produces 16 partial products as opposed to 32
    wire [32:0] pp0, pp1, pp2, pp3, pp4, pp5, pp6, pp7, pp8, pp9, pp10, pp11,
pp12, pp13, pp14, pp15;

    partial_product _pp0  ( booth_ones[1:0],   booth_signs[1:0],   multiplicand,
pp0 );
    partial_product _pp1  ( booth_ones[3:2],   booth_signs[3:2],   multiplicand,
pp1 );
    partial_product _pp2  ( booth_ones[5:4],   booth_signs[5:4],   multiplicand,
pp2 );
    partial_product _pp3  ( booth_ones[7:6],   booth_signs[7:6],   multiplicand,
pp3 );
    partial_product _pp4  ( booth_ones[9:8],   booth_signs[9:8],   multiplicand,
pp4 );
    partial_product _pp5  ( booth_ones[11:10], booth_signs[11:10], multiplicand,
pp5 );
    partial_product _pp6  ( booth_ones[13:12], booth_signs[13:12], multiplicand,
pp6 );
    partial_product _pp7  ( booth_ones[15:14], booth_signs[15:14], multiplicand,
pp7 );
    partial_product _pp8  ( booth_ones[17:16], booth_signs[17:16], multiplicand,
pp8 );
    partial_product _pp9  ( booth_ones[19:18], booth_signs[19:18], multiplicand,
pp9 );
    partial_product _pp10 ( booth_ones[21:20], booth_signs[21:20], multiplicand,
pp10 );
    partial_product _pp11 ( booth_ones[23:22], booth_signs[23:22], multiplicand,
pp11 );
    partial_product _pp12 ( booth_ones[25:24], booth_signs[25:24], multiplicand,
pp12 );
    partial_product _pp13 ( booth_ones[27:26], booth_signs[27:26], multiplicand,
```

```
pp13 );
    partial_product _pp14 ( booth_ones[29:28], booth_signs[29:28], multiplicand,
pp14 );
    partial_product _pp15 ( booth_ones[31:30], booth_signs[31:30], multiplicand,
pp15 );

    // Wallace Tree (Carry-Save Adders + 1 Hiearchical Carry Lookahead Adder)
    // Input partial products are all either (left) sign extended or (right) zero
padded to be 64-bit summands.
    // Quartus should be clever enough to eliminate the dead logic this creates
(and in testing, it does, within statistically insignifigant LE counts)

    // Level 0
    wire [63:0] sum00, sum01, sum02, sum03, sum04, carry00, carry01, carry02,
carry03, carry04, pass01;

    carry_save_adder #( .BITS(64) ) _csa00 ( .a({{31{pp0[32]}}, pp0        }),
.b({{29{pp1[32]}},  pp1,  2'b0 }), .c({{27{pp2[32]}}, pp2,  4'b0 }), .sum(sum00),
.carry(carry00) );
    carry_save_adder #( .BITS(64) ) _csa01 ( .a({{25{pp3[32]}}, pp3,  6'b0 }),
.b({{23{pp4[32]}},  pp4,  8'b0 }), .c({{21{pp5[32]}}, pp5,  10'b0}), .sum(sum01),
.carry(carry01) );
    carry_save_adder #( .BITS(64) ) _csa02 ( .a({{19{pp6[32]}}, pp6,  12'b0}),
.b({{17{pp7[32]}},  pp7,  14'b0}), .c({{15{pp8[32]}}, pp8,  16'b0}), .sum(sum02),
.carry(carry02) );
    carry_save_adder #( .BITS(64) ) _csa03 ( .a({{13{pp9[32]}}, pp9,  18'b0}),
.b({{11{pp10[32]}}, pp10, 20'b0}), .c({{9{pp11[32]}}, pp11, 22'b0}), .sum(sum03),
.carry(carry03) );
    carry_save_adder #( .BITS(64) ) _csa04 ( .a({{7{pp12[32]}}, pp12, 24'b0}),
.b({{5{pp13[32]}},  pp13, 26'b0}), .c({{3{pp14[32]}}, pp14, 28'b0}), .sum(sum04),
.carry(carry04) );

    assign pass01 = {{1{pp15[32]}}, pp15, 30'b0};

    // Level 1
    wire [63:0] sum10, sum11, sum12, carry10, carry11, carry12, pass10, pass11;

    carry_save_adder #( .BITS(64) ) _csa10 ( .a(sum00), .b({carry00[62:0], 1'b0}),
.c(sum01), .sum(sum10), .carry(carry10) );
    carry_save_adder #( .BITS(64) ) _csa11 ( .a({carry01[62:0], 1'b0}), .b(sum02),
.c({carry02[62:0], 1'b0}), .sum(sum11), .carry(carry11) );
    carry_save_adder #( .BITS(64) ) _csa12 ( .a(sum03), .b({carry03[62:0], 1'b0}),
.c(sum04), .sum(sum12), .carry(carry12) );

    assign pass10 = {carry04[62:0], 1'b0};
    assign pass11 = pass01;

    // Level 2
    wire [63:0] sum20, sum21, carry20, carry21, pass20, pass21;

    carry_save_adder #( .BITS(64) ) _csa20 ( .a(sum10), .b({carry10[62:0], 1'b0}),
.c(sum11), .sum(sum20), .carry(carry20) );
    carry_save_adder #( .BITS(64) ) _csa21 ( .a({carry11[62:0], 1'b0}), .b(sum12),
.c({carry12[62:0], 1'b0}), .sum(sum21), .carry(carry21) );
```

```
    assign pass20 = pass10;
    assign pass21 = pass11;

    // Level 3
    wire [63:0] sum30, sum31, carry30, carry31;

    carry_save_adder #( .BITS(64) ) _csa30 ( .a(sum20), .b({carry20[62:0], 1'b0}),
.c(sum21), .sum(sum30), .carry(carry30) );
    carry_save_adder #( .BITS(64) ) _csa31 ( .a({carry21[62:0], 1'b0}),
.b(pass20), .c(pass21), .sum(sum31), .carry(carry31) );

    // Level 4
    wire [63:0] sum4, carry4, pass4;

    carry_save_adder #( .BITS(64) ) _csa4 ( .a(sum30), .b({carry30[62:0], 1'b0}),
.c(sum31), .sum(sum4), .carry(carry4) );

    assign pass4 = {carry31[62:0], 1'b0};

    // Level 5

    wire [63:0] sum5, carry5;

    carry_save_adder #( .BITS(64) ) _csa5 ( .a(sum4), .b({carry4[62:0], 1'b0}),
.c(pass4), .sum(sum5), .carry(carry5) );

    // Level 6 (Top Level), Carry Lookahead Adder

    wire cla_c_out;

    carry_lookahead_adder #( .BITS16(4) ) _cla6 ( .a(sum5), .b({carry5[62:0],
1'b0}), .sum(product), .c_in(1'b0), .c_out(cla_c_out) );

endmodule

/**
 * Partial product calculator, using the booth encoded ones and signs
 * Computes a 33-bit partial product due to the maximum range of bit-pair
encoding.
 */
module partial_product(
    input [1:0] booth_ones,
    input [1:0] booth_signs,
    input [31:0] multiplicand,
    output [32:0] product
);

    wire [32:0] result_c, result_p; // Compliment of result
    reg [32:0] result;
    reg negative;

    assign result_p = result;
    signed_compliment #( .BITS(33) ) sc ( .in(result_p), .out(result_c) );
```

```
    // Product selects between the compliment and result value, based on the
negative flag
    assign product = negative ? result_c : result;

    always @(*) begin

        // Choose cases based on the booth ones + signs
        // Multiply by +/- 2: Shift left by one
        // Multiply by +/- 1: Sign extend by one
        // Otherwise: set to zero
        casez ({booth_ones, booth_signs})
            4'b00?? : //  0  0 ->  0
                begin
                    result = 33'b0;
                    negative = 1'b0;
                end
            4'b100? : // -1  0 -> -2
                begin
                    result = {multiplicand, 1'b0};
                    negative = 1'b1;
                end
            4'b101? : // +1  0 -> +2
                begin
                    result = {multiplicand, 1'b0};
                    negative = 1'b0;
                end
            4'b01?0 : //  0 -1 -> -1
                begin
                    result = {multiplicand[31], multiplicand};
                    negative = 1'b1;
                end
            4'b01?1 : //  0 +1 -> +1
                begin
                    result = {multiplicand[31], multiplicand};
                    negative = 1'b0;
                end
            // 4'b1100   -1 -1 -> Booth recoding will not have adjacent equal
signs
            4'b1101 : // -1 +1 -> -1
                begin
                    result = {multiplicand[31], multiplicand};
                    negative = 1'b1;
                end
            4'b1110 : // +1 -1 -> +1
                begin
                    result = {multiplicand[31], multiplicand};
                    negative = 1'b0;
                end
            // 4'b1111 : +1 +1 -> Booth recoding will not have adjacent equal
signs
            default :
                begin
                    result = 33'b0;
```

```
                    negative = 1'b0;
                end
        endcase
    end

endmodule


/**
 * Testbench
 * Lots of test cases because multiplication is hard
 */
`timescale 1ns/100ps
module booth_bit_pair_multiplier_test;

    reg signed [63:0] a, b;
    wire signed [63:0] product;

    integer i;

    booth_bit_pair_multiplier _mul ( .multiplicand(a[31:0]), .multiplier(b[31:0]),
.product(product) );

    assign ai = {{32{a[31]}}, a};
    assign bi = {{32{b[31]}}, b};

    initial begin
        // Regressions
        a <= 1062902654;
        b <= -309493541;
        #1 $display("Test | multiply regression %0d * %0d | %0d | %0d", a, b, a *
b, product);

        for (i = 0; i < 1000; i = i + 1) begin
            a <= $random;
            b <= $random;
            #1 $display("Test | multiply %0d * %0d | %0d | %0d", a, b, a * b,
product);
        end

        $finish;
    end

endmodule
```

File: hdl/carry_lookahead_adder.v

```
/**
 * A 16N-Bit RCA, implemented with two levels of nested CLAs.
 * The parameter specifies the bits in multiples of 16 bits.
 */
module carry_lookahead_adder #(
```

```
    parameter BITS16 = 2
) (
    input [(BITS16 * 16) - 1:0] a,
    input [(BITS16 * 16) - 1:0] b,
    output [(BITS16 * 16) - 1:0] sum,
    input c_in,
    output c_out
);

    // Internal carry line
    wire [BITS16:0] carry;
    wire [BITS16 - 1:0] gi, pi;

    // Connect to c_in and c_out
    assign carry[0] = c_in;
    assign c_out = carry[BITS16];

    genvar i;
    generate
        for (i = 0; i < BITS16; i = i + 1) begin : gen_adder
            carry_lookahead_adder_16b cla (
                .a(a[(16 * i) + 15:16 * i]),
                .b(b[(16 * i) + 15:16 * i]),
                .sum(sum[(16 * i) + 15:16 * i]),
                .c_in(carry[i]),
                .c_out(carry[i + 1])
            );
        end
    endgenerate
endmodule


`timescale 1ns/100ps
module carry_lookahead_adder_test;

    reg [31:0] a, b;
    wire [32:0] sum, a32, b32;
    reg c_in;

    integer i;

    assign a32 = {1'b0, a};
    assign b32 = {1'b0, b};

    carry_lookahead_adder #( .BITS16(2) ) cla ( .a(a), .b(b), .sum(sum[31:0]),
.c_in(c_in), .c_out(sum[32]) );

    initial begin
        for (i = 0; i < 1000; i = i + 1) begin
            a <= $random;
            b <= $random;
            c_in <= $random;
            #1 $display("Test | add %0d + %0d + %0d | %0d | %0d", a, b, c_in, a32
+ b32 + c_in, sum);
```

```
        end

        $finish;
    end

endmodule
```

File: hdl/carry_lookahead_adder_16b.v

```
module carry_lookahead_adder_16b(
    input [15:0] a,
    input [15:0] b,
    output [15:0] sum,
    input c_in,
    output c_out
);
    wire g0, g1, g2, g3, p0, p1, p2, p3, c0, c1, c2, c01, c02, c03;

    // Carry Lookahead
    assign c2 = g2 | (p2 & g1) | (p2 & p1 & g0) | (p2 & p1 & p0 & c_in);
    assign c1 = g1 | (p1 & g0) | (p1 & p0 & c_in);
    assign c0 = g0 | (p0 & c_in);

    // Sum + Carry
    carry_lookahead_adder_4b _cla0 ( .a(a[3:0]), .b(b[3:0]), .sum(sum[3:0]),
.c_in(c_in), .c_out(c01), .p_out(p0), .g_out(g0) );
    carry_lookahead_adder_4b _cla1 ( .a(a[7:4]), .b(b[7:4]), .sum(sum[7:4]),
.c_in(c0), .c_out(c02), .p_out(p1), .g_out(g1) );
    carry_lookahead_adder_4b _cla2 ( .a(a[11:8]), .b(b[11:8]), .sum(sum[11:8]),
.c_in(c1), .c_out(c03), .p_out(p2), .g_out(g2) );
    carry_lookahead_adder_4b _cla3 ( .a(a[15:12]), .b(b[15:12]), .sum(sum[15:12]),
.c_in(c2), .c_out(c_out), .p_out(p3), .g_out(g3) );

endmodule


`timescale 1ns/100ps
module carry_lookahead_adder_16b_test;

    reg [16:0] a, b;
    reg c_in;
    wire [16:0] sum;

    integer i;

    carry_lookahead_adder_16b cla ( .a(a[15:0]), .b(b[15:0]), .sum(sum[15:0]),
.c_in(c_in), .c_out(sum[16]) );

    initial begin
        for (i = 0; i < 1000; i = i + 1) begin
            a <= $urandom % 65536;
            b <= $urandom % 65536;
```

```
            c_in <= $urandom;
            #1 $display("Test | add %0d + %0d + %0d | %0d | %0d", a, b, c_in, a +
b + c_in, sum);
        end
    end
endmodule
```

File: hdl/carry_lookahead_adder_4b.v

```
module carry_lookahead_adder_4b(
    input [3:0] a,
    input [3:0] b,
    output [3:0] sum,
    input c_in,
    output c_out,
    output g_out,
    output p_out
);
    wire g0, g1, g2, g3, p0, p1, p2, p3, c0, c1, c2;

    assign {g3, g2, g1, g0} = a & b; // gi = ai & bi
    assign {p3, p2, p1, p0} = a | b; // pi = ai | bi

    // Carry Lookahead
    assign c2 = g2 | (p2 & g1) | (p2 & p1 & g0) | (p2 & p1 & p0 & c_in);
    assign c1 = g1 | (p1 & g0) | (p1 & p0 & c_in);
    assign c0 = g0 | (p0 & c_in);

    // Generate + Propogate out
    assign p_out = p3 & p2 & p1 & p0;
    assign g_out = g3 | (p3 & g2) | (p3 & p2 & g1) | (p3 & p2 & p1 & g0);

    // Sum + Carry
    assign sum = a ^ b ^ {c2, c1, c0, c_in};
    assign c_out = g_out | (p_out & c_in);

endmodule


`timescale 1ns/100ps
module carry_lookahead_adder_4b_test;

    reg [3:0] a, b;
    reg c_in;
    wire [4:0] sum;
    wire p_out, g_out;

    integer i, j, k;

    carry_lookahead_adder_4b cla ( .a(a), .b(b), .sum(sum[3:0]), .c_in(c_in),
.c_out(sum[4]), .p_out(p_out), .g_out(g_out) );
```

```
    initial begin
        c_in <= 0;
        for (i = 0; i < 16; i = i + 1) begin
            for (j = 0; j < 16; j = j + 1) begin
                for (k = 0; k <= 1; k = k + 1) begin
                    a <= i;
                    b <= j;
                    c_in <= k;
                    #1 $display("Test | add %0d + %0d + %0d | %0d | %0d", i, j, k,
i + j + k, sum);
                end
            end
        end
    end
endmodule
```

File: hdl/carry_save_adder.v

```
module carry_save_adder #(
    parameter BITS = 32
) (
    input [BITS - 1:0] a,
    input [BITS - 1:0] b,
    input [BITS - 1:0] c,
    output [BITS - 1:0] sum,
    output [BITS - 1:0] carry
);

    genvar i;
    generate
        for (i = 0; i < BITS; i = i + 1) begin : gen_fa
            full_adder fa ( .a(a[i]), .b(b[i]), .sum(sum[i]), .c_in(c[i]),
.c_out(carry[i]) );
        end
    endgenerate

endmodule


`timescale 1ns/100ps
module carry_save_adder_test;

    // 4-Bit RCA : Small enough to test all possible inputs exhaustively (16 * 16
* 16 = 4096 inputs)
    reg [4:0] a, b, c;
    wire [6:0] sum, carry;

    carry_save_adder #( .BITS(5) ) csa ( .a(a), .b(b), .c(c), .sum(sum[4:0]),
.carry(carry[4:0]) );

    assign sum[6:5] = 2'b0;
    assign carry[6:5] = 2'b0;
```

```
    integer i, j, k;

    initial begin
        for (i = 0; i < 16; i = i + 1) begin
            for (j = 0; j < 16; j = j + 1) begin
                for (k = 0; k < 16; k = k + 1) begin
                    a <= i;
                    b <= j;
                    c <= k;
                    #1 $display("Test | add %0d + %0d + %0d | %0d | %0d", a, b, c,
i + j + k, (carry << 1) + sum);
                end
            end
        end

        $finish;
    end
endmodule
```

File: hdl/cpu.v

```
module cpu(
    // Control Signals
    input ir_en,
    input pc_increment, input pc_in_alu, input pc_in_rf_a,
    input ma_in_pc, input ma_in_alu,
    input md_in_memory, input md_in_rf_b,
    input alu_a_in_rf, input alu_a_in_pc,
    input alu_b_in_rf, input alu_b_in_constant,
    input lo_en,
    input hi_en,
    input rf_in_alu, input rf_in_hi, input rf_in_lo, input rf_in_md,

    input [3:0] rf_a_addr,
    input [3:0] rf_b_addr,
    input [3:0] rf_z_addr,

    input [11:0] alu_select, // {alu_add, alu_sub, alu_shr, alu_shl, alu_ror,
alu_rol, alu_and, alu_or, alu_mul, alu_div, alu_neg, alu_not}
    input [31:0] constant_c, // Sign extended to 32-bit

    // Memory Interface
    input [31:0] data_from_memory,
    output [31:0] address_to_memory,
    output [31:0] data_to_memory,

    // To Control Logic
    input [31:0] ir_out,

    // Standard
    input clk,
```

```
    input clr
);

    datapath _datapath (
        .ir_en(ir_en),
        .pc_increment(pc_increment), .pc_in_alu(pc_in_alu),
.pc_in_rf_a(pc_in_rf_a),
        .ma_in_pc(ma_in_pc), .ma_in_alu(ma_in_alu),
        .md_in_memory(md_in_memory), .md_in_rf_b(md_in_rf_b),
        .alu_a_in_rf(alu_a_in_rf), .alu_a_in_pc(alu_a_in_pc),
        .alu_b_in_rf(alu_b_in_rf), .alu_b_in_constant(alu_b_in_constant),
        .lo_en(lo_en), .hi_en(hi_en),
        .rf_in_alu(rf_in_alu), .rf_in_hi(rf_in_hi), .rf_in_lo(rf_in_lo),
.rf_in_md(rf_in_md),
        .rf_a_addr(rf_a_addr), .rf_b_addr(rf_b_addr), .rf_z_addr(rf_z_addr),
        .alu_select(alu_select), .constant_c(constant_c),
        .data_from_memory(data_from_memory), .data_to_memory(data_to_memory),
.address_to_memory(address_to_memory),
        .ir_out(ir_out), .clk(clk), .clr(clr)
    );

endmodule


module cpu_test;

    initial begin
        $finish;
    end
endmodule
```

File: hdl/datapath.v

```
module datapath(
    // Control Signals
    input ir_en,
    input pc_increment, input pc_in_alu, input pc_in_rf_a,
    input ma_in_pc, input ma_in_alu,
    input md_in_memory, input md_in_rf_b,
    input alu_a_in_rf, input alu_a_in_pc,
    input alu_b_in_rf, input alu_b_in_constant,
    input lo_en,
    input hi_en,
    input rf_in_alu, input rf_in_hi, input rf_in_lo, input rf_in_md,

    input [3:0] rf_a_addr,
    input [3:0] rf_b_addr,
    input [3:0] rf_z_addr,

    input [11:0] alu_select,
    input [31:0] constant_c, // Sign extended to 32-bit
```

```
    // Memory Interface
    input [31:0] data_from_memory,
    output [31:0] address_to_memory,
    output [31:0] data_to_memory,

    // To Control Logic
    input [31:0] ir_out,

    // Standard
    input clk,
    input clr
);
    // Based on the 3-Bus Architecture
    // We can exclude the A, B, Y and Z registers
    wire [31:0] pc_out, ma_out, md_out, hi_out, lo_out, rf_a_out, rf_b_out,
alu_z_out, alu_lo_out, alu_hi_out;
    reg [31:0] pc_in, ma_in, md_in, alu_a_in, alu_b_in, rf_in;
    wire pc_en, ma_en, md_en, rf_en;

    // Additional register connections

    // PC Increment Logic
    // Control Signals: pc_increment, pc_in_alu, pc_in_rf_a
    // Inputs: PC + 4, PC + C, rX
    wire [31:0] pc_plus_4;
    wire pc_cout;

    ripple_carry_adder _pc_adder ( .a(pc_out), .b(32'b100), .sum(pc_plus_4),
.c_in(1'b0), .c_out(pc_cout) ); // PC + 4

    assign pc_en = pc_increment | pc_in_alu | pc_in_rf_a;

    always @(*) begin
        case ({pc_increment, pc_in_alu, pc_in_rf_a})
            3'b001 : pc_in <= rf_a_out;
            3'b010 : pc_in <= alu_z_out;
            3'b100 : pc_in <= pc_plus_4;
            default : pc_in <= 32'b0;
        endcase
    end

    // MA Register
    // Control Signals: ma_in_pc, ma_in_alu
    assign ma_en = ma_in_pc | ma_in_alu;
    assign address_to_memory = ma_out;

    always @(*) begin
        case ({ma_in_pc, ma_in_alu})
            2'b01 : ma_in <= alu_z_out;
            2'b10 : ma_in <= pc_out;
            default : ma_in <= 32'b0;
        endcase
    end
```

```
    // MD Register
    // Control Signals: md_in_memory, md_in_rf_b
    // Inputs: Memory[MA], rX
    assign md_en = md_in_memory | md_in_rf_b;
    assign data_to_memory = md_out;

    always @(*) begin
        case ({md_in_memory, md_in_rf_b})
            2'b01 : md_in <= rf_b_out;
            2'b10 : md_in <= data_from_memory;
            default : md_in <= 32'b0;
        endcase
    end

    // ALU Inputs
    // Left input can be PC or rY
    // Right input can be rX or constant C
    // Control Signals: alu_a_in_rf, alu_a_in_pc, alu_b_in_rf, alu_b_in_constant
    always @(*) begin
        case ({alu_a_in_rf, alu_a_in_pc})
            2'b01 : alu_a_in <= pc_out;
            2'b10 : alu_a_in <= rf_a_out;
            default : alu_a_in <= 32'b0;
        endcase

        case ({alu_b_in_rf, alu_b_in_constant})
            2'b01 : alu_b_in <= constant_c;
            2'b10 : alu_b_in <= rf_b_out;
            default : alu_b_in <= 32'b0;
        endcase
    end

    // RF Inputs
    // Input can by any of ALU Z, HI, LO, MD
    assign rf_en = rf_in_alu | rf_in_hi | rf_in_lo | rf_in_md;

    always @(*) begin
        case ({rf_in_alu, rf_in_hi, rf_in_lo, rf_in_md})
            4'b0001 : rf_in <= md_out;
            4'b0010 : rf_in <= lo_out;
            4'b0100 : rf_in <= hi_out;
            4'b1000 : rf_in <= alu_z_out;
            default : rf_in <= 32'b0;
        endcase
    end

    register_file _rf (
        .write_data(rf_in),
        .write_addr(rf_en ? rf_z_addr : 4'b0), // When not enabled, writes go to
r0 (noop)
        .read_addr_a(rf_a_addr),
        .read_addr_b(rf_b_addr),
        .data_a(rf_a_out),
        .data_b(rf_b_out),
```

```
        .clk(clk),
        .clr(clr)
    );

    register _pc ( .q(pc_in),      .d(pc_out), .en(pc_en), .clk(clk), .clr(clr) );
    register _ir ( .q(md_out),     .d(ir_out), .en(ir_en), .clk(clk), .clr(clr) );
// IR in = MDR out
    register _ma ( .q(ma_in),      .d(ma_out), .en(ma_en), .clk(clk), .clr(clr) );
    register _md ( .q(md_in),      .d(md_out), .en(md_en), .clk(clk), .clr(clr) );
    register _hi ( .q(alu_hi_out), .d(hi_out), .en(hi_en), .clk(clk), .clr(clr) );
// HI and LO in = ALU out
    register _lo ( .q(alu_lo_out), .d(lo_out), .en(lo_en), .clk(clk), .clr(clr) );

    alu _alu ( .a(alu_a_in), .b(alu_b_in), .z(alu_z_out), .hi(alu_hi_out),
.lo(alu_lo_out), .select(alu_select) );

endmodule


/**
 * Testbench
 * Simulates various instructions by manually wiring in the correct control
signals
 */
`timescale 1ns/100ps
module datapath_test;

    // Control Signals
    reg ir_en;
    reg pc_increment, pc_in_alu, pc_in_rf_a;
    reg ma_in_pc, ma_in_alu;
    reg md_in_memory, md_in_rf_b;
    reg alu_a_in_rf, alu_a_in_pc;
    reg alu_b_in_rf, alu_b_in_constant;
    reg lo_en, hi_en;
    reg rf_in_alu, rf_in_hi, rf_in_lo, rf_in_md;

    reg [3:0] rf_a_addr, rf_b_addr, rf_z_addr;

    reg alu_not, alu_neg, alu_div, alu_mul, alu_or, alu_and, alu_rol, alu_ror,
alu_shl, alu_shr, alu_sub, alu_add;
    reg [31:0] constant_c, data_from_memory;
    wire [31:0] address_to_memory, data_to_memory, ir_out;

    reg clk, clr;

    datapath _dp (
        .ir_en(ir_en),
        .pc_increment(pc_increment), .pc_in_alu(pc_in_alu),
.pc_in_rf_a(pc_in_rf_a),
        .ma_in_pc(ma_in_pc), .ma_in_alu(ma_in_alu),
        .md_in_memory(md_in_memory), .md_in_rf_b(md_in_rf_b),
        .alu_a_in_rf(alu_a_in_rf), .alu_a_in_pc(alu_a_in_pc),
        .alu_b_in_rf(alu_b_in_rf), .alu_b_in_constant(alu_b_in_constant),
```

```
        .lo_en(lo_en), .hi_en(hi_en),
        .rf_in_alu(rf_in_alu), .rf_in_hi(rf_in_hi), .rf_in_lo(rf_in_lo),
.rf_in_md(rf_in_md),
        .rf_a_addr(rf_a_addr), .rf_b_addr(rf_b_addr), .rf_z_addr(rf_z_addr),
        .alu_select({alu_not, alu_neg, alu_div, alu_mul, alu_or, alu_and, alu_rol,
alu_ror, alu_shl, alu_shr, alu_sub, alu_add}), .constant_c(constant_c),
        .data_from_memory(data_from_memory), .data_to_memory(data_to_memory),
.address_to_memory(address_to_memory),
        .ir_out(ir_out), .clk(clk), .clr(clr)
    );

    // Clock
    initial begin
        clk <= 1'b1;
        forever #5 clk <= ~clk;
    end

    initial begin
        // Zero all inputs
        ir_en <= 1'b0;
        pc_increment <= 1'b0; pc_in_alu <= 1'b0; pc_in_rf_a <= 1'b0;
        ma_in_pc <= 1'b0; ma_in_alu <= 1'b0;
        md_in_memory <= 1'b0; md_in_rf_b <= 1'b0;
        alu_a_in_rf <= 1'b0; alu_a_in_pc <= 1'b0;
        alu_b_in_rf <= 1'b0; alu_b_in_constant <= 1'b0;
        lo_en <= 1'b0; hi_en <= 1'b0;
        rf_in_alu <= 1'b0; rf_in_hi <= 1'b0; rf_in_lo <= 1'b0; rf_in_md <= 1'b0;
        rf_a_addr <= 4'b0; rf_b_addr <= 4'b0; rf_z_addr <= 4'b0;
        {alu_not, alu_neg, alu_div, alu_mul, alu_or, alu_and, alu_rol, alu_ror,
alu_shl, alu_shr, alu_sub, alu_add} <= 12'b0; constant_c <= 32'b0;
        data_from_memory <= 32'b0;
        clr <= 1'b0;
        #10

        // Start
        #1 clr <= 1'b1;

        // Initialize some values in RF

        // Load MD
        md_in_memory = 1'b1; data_from_memory <= 53;
        #10

        // Load r2
        md_in_memory = 1'b0; data_from_memory <= 0;
        rf_in_md <= 1'b1; rf_z_addr <= 4'b10; // r2 = 53
        #10 $display("Test | init r2 | r2=53 | r2=%0d", _dp._rf.data[2]);

        // Load MD
        rf_in_md <= 1'b0; rf_z_addr <= 4'b0;
        md_in_memory = 1'b1; data_from_memory <= 28;
        #10

        // Load r4
```

```
        md_in_memory = 1'b0; data_from_memory <= 0;
        rf_in_md <= 1'b1; rf_z_addr <= 4'b100; // r4 = 28
        #10 $display("Test | init r4 | r2=28 | r2=%0d", _dp._rf.data[4]);

        // Clear
        rf_in_md <= 1'b0; rf_z_addr <= 4'b0;
        #10

        // Instruction Simulation
        // Program (In Memory)
        // 00 and r5 r2 r4
        // 04 or r5 r2 r4
        // 08 add r5 r2 r4
        // 12 sub r5 r2 r4
        // 16 shr r5 r2 r4
        // 20 shl r5 r2 r4
        // 24 ror r5 r2 r4
        // 32 mul r2 r4
        // 36 div r2 r4
        // 40 neg r5 r2
        // 44 not r5 r2

        // and r5, r2, r4

        // T0
        pc_increment <= 1'b1; ma_in_pc <= 1'b1;
        #10 $display("Test | and r5 r2 r4 @ T0 | pc=4, ma=0 | pc=%0d, ma=%0d",
_dp._pc.d, _dp._ma.d);

        // T1
        pc_increment <= 1'b0; ma_in_pc <= 1'b0;
        md_in_memory <= 1'b1; data_from_memory <=
32'b01001_0101_0010_0100_000000000000000;
        #10 $display("Test | and r5 r2 r4 @ T1 | md=0x4a920000 | md=0x%h",
_dp._md.d);

        // T2
        md_in_memory <= 1'b0; data_from_memory <= 32'b0;
        ir_en <= 1'b1;
        #10 $display("Test | and r5 r2 r4 @ T2 | ir=0x4a920000 | ir=0x%h",
_dp._ir.d);

        // T3
        ir_en <= 1'b0;
        rf_z_addr <= 4'b0101; rf_a_addr <= 4'b0010; rf_b_addr <= 4'b0100;
alu_a_in_rf <= 1'b1; alu_b_in_rf <= 1'b1; rf_in_alu <= 1'b1; alu_and <= 1'b1;
        #10 $display("Test | add r5 r2 r4 @ T3 | a=53, b=28, z=20, r5=20 | a=%0d,
b=%0d, z=%0d, r5=%0d", _dp._alu.a, _dp._alu.b, _dp._alu.z, _dp._rf.data[5]);

        // or r5, r2, r4

        // T0
        rf_z_addr <= 4'b0; rf_a_addr <= 4'b0; rf_b_addr <= 4'b0; alu_a_in_rf <=
1'b0; alu_b_in_rf <= 1'b0; rf_in_alu <= 1'b0; alu_and <= 1'b0;
```

```
        pc_increment <= 1'b1; ma_in_pc <= 1'b1;
        #10 $display("Test | or r5 r2 r4 @ T0 | pc=8, ma=4 | pc=%0d, ma=%0d",
_dp._pc.d, _dp._ma.d);

        // T1
        pc_increment <= 1'b0; ma_in_pc <= 1'b0;
        md_in_memory <= 1'b1; data_from_memory <=
32'b01010_0101_0010_0100_000000000000000;
        #10 $display("Test | or r5 r2 r4 @ T1 | md=0x52920000 | md=0x%h",
_dp._md.d);

        // T2
        md_in_memory <= 1'b0; data_from_memory <= 32'b0;
        ir_en <= 1'b1;
        #10 $display("Test | or r5 r2 r4 @ T2 | ir=0x52920000 | ir=0x%h",
_dp._ir.d);

        // T3
        ir_en <= 1'b0;
        rf_z_addr <= 4'b0101; rf_a_addr <= 4'b0010; rf_b_addr <= 4'b0100;
alu_a_in_rf <= 1'b1; alu_b_in_rf <= 1'b1; rf_in_alu <= 1'b1; alu_or <= 1'b1;
        #10 $display("Test | or r5 r2 r4 @ T3 | a=53, b=28, z=61, r5=61 | a=%0d,
b=%0d, z=%0d, r5=%0d", _dp._alu.a, _dp._alu.b, _dp._alu.z, _dp._rf.data[5]);

        // add r5, r2, r4

        // T0
        rf_z_addr <= 4'b0; rf_a_addr <= 4'b0; rf_b_addr <= 4'b0; alu_a_in_rf <=
1'b0; alu_b_in_rf <= 1'b0; rf_in_alu <= 1'b0; alu_or <= 1'b0;
        pc_increment <= 1'b1; ma_in_pc <= 1'b1;
        #10 $display("Test | add r5 r2 r4 @ T0 | pc=12, ma=8 | pc=%0d, ma=%0d",
_dp._pc.d, _dp._ma.d);

        // T1
        pc_increment <= 1'b0; ma_in_pc <= 1'b0;
        md_in_memory <= 1'b1; data_from_memory <=
32'b00011_0101_0010_0100_000000000000000;
        #10 $display("Test | add r5 r2 r4 @ T1 | md=0x1a920000 | md=0x%h",
_dp._md.d);

        // T2
        md_in_memory <= 1'b0; data_from_memory <= 32'b0;
        ir_en <= 1'b1;
        #10 $display("Test | add r5 r2 r4 @ T2 | ir=0x1a920000 | ir=0x%h",
_dp._ir.d);

        // T3
        ir_en <= 1'b0;
        rf_z_addr <= 4'b0101; rf_a_addr <= 4'b0010; rf_b_addr <= 4'b0100;
alu_a_in_rf <= 1'b1; alu_b_in_rf <= 1'b1; rf_in_alu <= 1'b1; alu_add <= 1'b1;
        #10 $display("Test | add r5 r2 r4 @ T3 | a=53, b=28, z=81, r5=81 | a=%0d,
b=%0d, z=%0d, r5=%0d", _dp._alu.a, _dp._alu.b, _dp._alu.z, _dp._rf.data[5]);
```

```
        // sub r5, r2, r4

        // T0
        rf_z_addr <= 4'b0; rf_a_addr <= 4'b0; rf_b_addr <= 4'b0; alu_a_in_rf <=
1'b0; alu_b_in_rf <= 1'b0; rf_in_alu <= 1'b0; alu_add <= 1'b0;
        pc_increment <= 1'b1; ma_in_pc <= 1'b1;
        #10 $display("Test | sub r5 r2 r4 @ T0 | pc=16, ma=12 | pc=%0d, ma=%0d",
_dp._pc.d, _dp._ma.d);

        // T1
        pc_increment <= 1'b0; ma_in_pc <= 1'b0;
        md_in_memory <= 1'b1; data_from_memory <=
32'b00100_0101_0010_0100_000000000000000;
        #10 $display("Test | sub r5 r2 r4 @ T1 | md=0x22920000 | md=0x%h",
_dp._md.d);

        // T2
        md_in_memory <= 1'b0; data_from_memory <= 32'b0;
        ir_en <= 1'b1;
        #10 $display("Test | sub r5 r2 r4 @ T2 | ir=0x22920000 | ir=0x%h",
_dp._ir.d);

        // T3
        ir_en <= 1'b0;
        rf_z_addr <= 4'b0101; rf_a_addr <= 4'b0010; rf_b_addr <= 4'b0100;
alu_a_in_rf <= 1'b1; alu_b_in_rf <= 1'b1; rf_in_alu <= 1'b1; alu_sub <= 1'b1;
        #10 $display("Test | sub r5 r2 r4 @ T3 | a=53, b=28, z=25, r5=25 | a=%0d,
b=%0d, z=%0d, r5=%0d", _dp._alu.a, _dp._alu.b, _dp._alu.z, _dp._rf.data[5]);

        // shr r5, r2, r4

        // T0
        rf_z_addr <= 4'b0; rf_a_addr <= 4'b0; rf_b_addr <= 4'b0; alu_a_in_rf <=
1'b0; alu_b_in_rf <= 1'b0; rf_in_alu <= 1'b0; alu_sub <= 1'b0;
        pc_increment <= 1'b1; ma_in_pc <= 1'b1;
        #10 $display("Test | shr r5 r2 r4 @ T0 | pc=20, ma=16 | pc=%0d, ma=%0d",
_dp._pc.d, _dp._ma.d);

        // T1
        pc_increment <= 1'b0; ma_in_pc <= 1'b0;
        md_in_memory <= 1'b1; data_from_memory <=
32'b00101_0101_0010_0100_000000000000000;
        #10 $display("Test | shr r5 r2 r4 @ T1 | md=0x2a920000 | md=0x%h",
_dp._md.d);

        // T2
        md_in_memory <= 1'b0; data_from_memory <= 32'b0;
        ir_en <= 1'b1;
        #10 $display("Test | shr r5 r2 r4 @ T2 | ir=0x2a920000 | ir=0x%h",
_dp._ir.d);

        // T3
        ir_en <= 1'b0;
        rf_z_addr <= 4'b0101; rf_a_addr <= 4'b0010; rf_b_addr <= 4'b0100;
```

```
alu_a_in_rf <= 1'b1; alu_b_in_rf <= 1'b1; rf_in_alu <= 1'b1; alu_shr <= 1'b1;
        #10 $display("Test | shr r5 r2 r4 @ T3 | a=53, b=28, z=0, r5=0 | a=%0d,
b=%0d, z=%0d, r5=%0d", _dp._alu.a, _dp._alu.b, _dp._alu.z, _dp._rf.data[5]);


        // shl r5, r2, r4

        // T0
        rf_z_addr <= 4'b0; rf_a_addr <= 4'b0; rf_b_addr <= 4'b0; alu_a_in_rf <=
1'b0; alu_b_in_rf <= 1'b0; rf_in_alu <= 1'b0; alu_shr <= 1'b0;
        pc_increment <= 1'b1; ma_in_pc <= 1'b1;
        #10 $display("Test | shl r5 r2 r4 @ T0 | pc=24, ma=20 | pc=%0d, ma=%0d",
_dp._pc.d, _dp._ma.d);

        // T1
        pc_increment <= 1'b0; ma_in_pc <= 1'b0;
        md_in_memory <= 1'b1; data_from_memory <=
32'b00110_0101_0010_0100_000000000000000;
        #10 $display("Test | shl r5 r2 r4 @ T1 | md=0x32920000 | md=0x%h",
_dp._md.d);

        // T2
        md_in_memory <= 1'b0; data_from_memory <= 32'b0;
        ir_en <= 1'b1;
        #10 $display("Test | shl r5 r2 r4 @ T2 | ir=0x32920000 | ir=0x%h",
_dp._ir.d);

        // T3
        ir_en <= 1'b0;
        rf_z_addr <= 4'b0101; rf_a_addr <= 4'b0010; rf_b_addr <= 4'b0100;
alu_a_in_rf <= 1'b1; alu_b_in_rf <= 1'b1; rf_in_alu <= 1'b1; alu_shl <= 1'b1;
        #10 $display("Test | shl r5 r2 r4 @ T3 | a=53, b=28, z=1342177280,
r5=1342177280 | a=%0d, b=%0d, z=%0d, r5=%0d", _dp._alu.a, _dp._alu.b, _dp._alu.z,
_dp._rf.data[5]);

        // ror r5, r2, r4

        // T0
        rf_z_addr <= 4'b0; rf_a_addr <= 4'b0; rf_b_addr <= 4'b0; alu_a_in_rf <=
1'b0; alu_b_in_rf <= 1'b0; rf_in_alu <= 1'b0; alu_shl <= 1'b0;
        pc_increment <= 1'b1; ma_in_pc <= 1'b1;
        #10 $display("Test | ror r5 r2 r4 @ T0 | pc=28, ma=24 | pc=%0d, ma=%0d",
_dp._pc.d, _dp._ma.d);

        // T1
        pc_increment <= 1'b0; ma_in_pc <= 1'b0;
        md_in_memory <= 1'b1; data_from_memory <=
32'b00111_0101_0010_0100_000000000000000;
        #10 $display("Test | ror r5 r2 r4 @ T1 | md=0x3a920000 | md=0x%h",
_dp._md.d);

        // T2
        md_in_memory <= 1'b0; data_from_memory <= 32'b0;
        ir_en <= 1'b1;
```

```
        #10 $display("Test | ror r5 r2 r4 @ T2 | ir=0x3a920000 | ir=0x%h",
_dp._ir.d);

        // T3
        ir_en <= 1'b0;
        rf_z_addr <= 4'b0101; rf_a_addr <= 4'b0010; rf_b_addr <= 4'b0100;
alu_a_in_rf <= 1'b1; alu_b_in_rf <= 1'b1; rf_in_alu <= 1'b1; alu_ror <= 1'b1;
        #10 $display("Test | ror r5 r2 r4 @ T3 | a=53, b=28, z=848, r5=848 |
a=%0d, b=%0d, z=%0d, r5=%0d", _dp._alu.a, _dp._alu.b, _dp._alu.z,
_dp._rf.data[5]);

        // rol r5, r2, r4

        // T0
        rf_z_addr <= 4'b0; rf_a_addr <= 4'b0; rf_b_addr <= 4'b0; alu_a_in_rf <=
1'b0; alu_b_in_rf <= 1'b0; rf_in_alu <= 1'b0; alu_ror <= 1'b0;
        pc_increment <= 1'b1; ma_in_pc <= 1'b1;
        #10 $display("Test | rol r5 r2 r4 @ T0 | pc=32, ma=28 | pc=%0d, ma=%0d",
_dp._pc.d, _dp._ma.d);

        // T1
        pc_increment <= 1'b0; ma_in_pc <= 1'b0;
        md_in_memory <= 1'b1; data_from_memory <=
32'b01000_0101_0010_0100_000000000000000;
        #10 $display("Test | rol r5 r2 r4 @ T1 | md=0x42920000 | md=0x%h",
_dp._md.d);

        // T2
        md_in_memory <= 1'b0; data_from_memory <= 32'b0;
        ir_en <= 1'b1;
        #10 $display("Test | rol r5 r2 r4 @ T2 | ir=0x42920000 | ir=0x%h",
_dp._ir.d);

        // T3
        ir_en <= 1'b0;
        rf_z_addr <= 4'b0101; rf_a_addr <= 4'b0010; rf_b_addr <= 4'b0100;
alu_a_in_rf <= 1'b1; alu_b_in_rf <= 1'b1; rf_in_alu <= 1'b1; alu_rol <= 1'b1;
        #10 $display("Test | rol r5 r2 r4 @ T3 | a=53, b=28, z=1342177283,
r5=1342177283 | a=%0d, b=%0d, z=%0d, r5=%0d", _dp._alu.a, _dp._alu.b, _dp._alu.z,
_dp._rf.data[5]);


        // mul r2, r4

        // T0
        rf_z_addr <= 4'b0; rf_a_addr <= 4'b0; rf_b_addr <= 4'b0; alu_a_in_rf <=
1'b0; alu_b_in_rf <= 1'b0; rf_in_alu <= 1'b0; alu_rol <= 1'b0;
        pc_increment <= 1'b1; ma_in_pc <= 1'b1;
        #10 $display("Test | mul r2 r4 @ T0 | pc=36, ma=32 | pc=%0d, ma=%0d",
_dp._pc.d, _dp._ma.d);

        // T1
        pc_increment <= 1'b0; ma_in_pc <= 1'b0;
        md_in_memory <= 1'b1; data_from_memory <=
```

```
32'b01110_0000_0010_0100_000000000000000;
        #10 $display("Test | mul r2 r4 @ T1 | md=0x70120000 | md=0x%h",
_dp._md.d);

        // T2
        md_in_memory <= 1'b0; data_from_memory <= 32'b0;
        ir_en <= 1'b1;
        #10 $display("Test | mul r2 r4 @ T2 | ir=0x70120000 | ir=0x%h",
_dp._ir.d);

        // T3
        ir_en <= 1'b0;
        rf_a_addr <= 4'b0010; rf_b_addr <= 4'b0100; alu_a_in_rf <= 1'b1;
alu_b_in_rf <= 1'b1; alu_mul = 1'b1; hi_en = 1'b1; lo_en = 1'b1;
        #10 $display("Test | mul r2 r4 @ T3 | a=53, b=28, hi=0, lo=1484 | a=%0d,
b=%0d, hi=%0d, lo=%0d", _dp._alu.a, _dp._alu.b, _dp._hi.d, _dp._lo.d);


        // div r2, r4

        // T0
        rf_a_addr <= 4'b0; rf_b_addr <= 4'b0; alu_a_in_rf <= 1'b0; alu_b_in_rf <=
1'b0; alu_mul = 1'b0; hi_en = 1'b0; lo_en = 1'b0;
        pc_increment <= 1'b1; ma_in_pc <= 1'b1;
        #10 $display("Test | div r2 r4 @ T0 | pc=40, ma=36 | pc=%0d, ma=%0d",
_dp._pc.d, _dp._ma.d);

        // T1
        pc_increment <= 1'b0; ma_in_pc <= 1'b0;
        md_in_memory <= 1'b1; data_from_memory <=
32'b01111_0000_0010_0100_000000000000000;
        #10 $display("Test | div r2 r4 @ T1 | md=0x78120000 | md=0x%h",
_dp._md.d);

        // T2
        md_in_memory <= 1'b0; data_from_memory <= 32'b0;
        ir_en <= 1'b1;
        #10 $display("Test | div r2 r4 @ T2 | ir=0x78120000 | ir=0x%h",
_dp._ir.d);

        // T3
        ir_en <= 1'b0;
        rf_a_addr <= 4'b0010; rf_b_addr <= 4'b0100; alu_a_in_rf <= 1'b1;
alu_b_in_rf <= 1'b1; alu_div = 1'b1; hi_en = 1'b1; lo_en = 1'b1;
        #10 $display("Test | div r2 r4 @ T3 | a=53, b=28, hi=25, lo=1 | a=%0d,
b=%0d, hi=%0d, lo=%0d", _dp._alu.a, _dp._alu.b, _dp._hi.d, _dp._lo.d);

        // neg r5, r2

        rf_a_addr <= 4'b0; rf_b_addr <= 4'b0; alu_a_in_rf <= 1'b0; alu_b_in_rf <=
1'b0; alu_div = 1'b0; hi_en = 1'b0; lo_en = 1'b0;
        pc_increment <= 1'b1; ma_in_pc <= 1'b1;
        #10 $display("Test | neg r5 r2 @ T0 | pc=44, ma=40 | pc=%0d, ma=%0d",
_dp._pc.d, _dp._ma.d);
```

```
        // T1
        pc_increment <= 1'b0; ma_in_pc <= 1'b0;
        md_in_memory <= 1'b1; data_from_memory <=
32'b10000_0101_0010_0000_000000000000000;
        #10 $display("Test | neg r5 r2 @ T1 | md=0x82900000 | md=0x%h",
_dp._md.d);

        // T2
        md_in_memory <= 1'b0; data_from_memory <= 32'b0;
        ir_en <= 1'b1;
        #10 $display("Test | neg r5 r2 @ T2 | ir=0x82900000 | ir=0x%h",
_dp._ir.d);

        // T3
        ir_en <= 1'b0;
        rf_z_addr <= 4'b0101; rf_a_addr <= 4'b0010; alu_a_in_rf <= 1'b1; rf_in_alu
<= 1'b1; alu_neg <= 1'b1;
        #10 $display("Test | neg r5 r2 @ T3 | a=53, z=-53, r5=-53 | a=%0d, z=%0d,
r5=%0d", _dp._alu.a, $signed(_dp._alu.z), $signed(_dp._rf.data[5]));

        // not r5, r2

        rf_z_addr <= 4'b0; rf_a_addr <= 4'b0; alu_a_in_rf <= 1'b0; rf_in_alu <=
1'b0; alu_neg <= 1'b0;
        pc_increment <= 1'b1; ma_in_pc <= 1'b1;
        #10 $display("Test | not r5 r2 @ T0 | pc=48, ma=44 | pc=%0d, ma=%0d",
_dp._pc.d, _dp._ma.d);

        // T1
        pc_increment <= 1'b0; ma_in_pc <= 1'b0;
        md_in_memory <= 1'b1; data_from_memory <=
32'b10001_0101_0010_0000_000000000000000;
        #10 $display("Test | not r5 r2 @ T1 | md=0x8a900000 | md=0x%h",
_dp._md.d);

        // T2
        md_in_memory <= 1'b0; data_from_memory <= 32'b0;
        ir_en <= 1'b1;
        #10 $display("Test | not r5 r2 @ T2 | ir=0x8a900000 | ir=0x%h",
_dp._ir.d);

        // T3
        ir_en <= 1'b0;
        rf_z_addr <= 4'b0101; rf_a_addr <= 4'b0010; alu_a_in_rf <= 1'b1; rf_in_alu
<= 1'b1; alu_not <= 1'b1;
        #10 $display("Test | not r5 r2 @ T3 | a=0x00000035, z=0xffffffca,
r5=0xffffffca | a=0x%h, z=0x%h, r5=0x%h", _dp._alu.a, _dp._alu.z,
_dp._rf.data[5]);

    end
endmodule
```

File: hdl/full_adder.v

```
module full_adder(
    input a,
    input b,
    output sum,
    input c_in,
    output c_out
);

    assign sum = a ^ b ^ c_in;
    assign c_out = (a & b) | (a & c_in) | (b & c_in);

endmodule

`timescale 1ns/100ps
module full_adder_test;

    reg a, b, c_in;
    wire sum, c_out;

    full_adder target ( .a(a), .b(b), .sum(sum), .c_in(c_in), .c_out(c_out) );

    initial begin
        // All input combinations
        a <= 0; b <= 0; c_in <= 0; #1 $display("Test | add0 | 0 -> 0 + 0 = 0 -> 0
| %d -> %d + %d = %d -> %d", c_in, a, b, sum, c_out);
        a <= 1; b <= 0; c_in <= 0; #1 $display("Test | add1 | 0 -> 1 + 0 = 1 -> 0
| %d -> %d + %d = %d -> %d", c_in, a, b, sum, c_out);
        a <= 0; b <= 1; c_in <= 0; #1 $display("Test | add2 | 0 -> 0 + 1 = 1 -> 0
| %d -> %d + %d = %d -> %d", c_in, a, b, sum, c_out);
        a <= 1; b <= 1; c_in <= 0; #1 $display("Test | add3 | 0 -> 1 + 1 = 0 -> 1
| %d -> %d + %d = %d -> %d", c_in, a, b, sum, c_out);
        a <= 0; b <= 0; c_in <= 1; #1 $display("Test | add4 | 1 -> 0 + 0 = 1 -> 0
| %d -> %d + %d = %d -> %d", c_in, a, b, sum, c_out);
        a <= 1; b <= 0; c_in <= 1; #1 $display("Test | add5 | 1 -> 1 + 0 = 0 -> 1
| %d -> %d + %d = %d -> %d", c_in, a, b, sum, c_out);
        a <= 0; b <= 1; c_in <= 1; #1 $display("Test | add6 | 1 -> 0 + 1 = 0 -> 1
| %d -> %d + %d = %d -> %d", c_in, a, b, sum, c_out);
        a <= 1; b <= 1; c_in <= 1; #1 $display("Test | add7 | 1 -> 1 + 1 = 1 -> 1
| %d -> %d + %d = %d -> %d", c_in, a, b, sum, c_out);

        $finish;
    end
endmodule
```

File: hdl/left_shift_32b.v

```
module left_shift_32b (
    input [31:0] in,
```

```
    input [31:0] shift,
    output [31:0] out,
    input is_rotate
);
    wire [31:0] shift_1, shift_2, shift_4, shift_8, shift_16;

    // Shift amount is a 32-bit unsigned value
    // For shifts >= 32, any of the upper bits = 0 indicates the result is zero
(this has no effect on rotates).
    wire is_zero;
    assign is_zero = !is_rotate && (| shift[31:5]);

    assign shift_1 = shift[0] ? { in[30:0], (is_rotate ? in[31] : 1'b0) } : in;
    assign shift_2 = shift[1] ? { shift_1[29:0], (is_rotate ? shift_1[31:30] :
2'b0) } : shift_1;
    assign shift_4 = shift[2] ? { shift_2[27:0], (is_rotate ? shift_2[31:28] :
4'b0) } : shift_2;
    assign shift_8 = shift[3] ? { shift_4[23:0], (is_rotate ? shift_4[31:24] :
8'b0) } : shift_4;
    assign shift_16 = shift[4] ? { shift_8[15:0], (is_rotate ? shift_8[31:16] :
16'b0) } : shift_8;
    assign out = is_zero ? 32'b0 : shift_16;

endmodule


// Testbench
`timescale 1ns/100ps
module left_shift_32b_test;

    reg [31:0] in;
    reg [31:0] shift;
    wire [31:0] out_shift, out_rotate;

    left_shift_32b _ls ( .in(in), .shift(shift), .out(out_shift), .is_rotate(1'b0)
);
    left_shift_32b _lr ( .in(in), .shift(shift), .out(out_rotate),
.is_rotate(1'b1) );

    integer i;

    initial begin
        // Shift values between [0, 32)
        for (i = 0; i < 100; i = i + 1) begin
            in <= $urandom;
            shift <= $urandom % 32;
            #1 $display("Test | left shift 0x%h << %0d | 0x%h | 0x%h", in, shift,
in << shift, out_shift);
            #1 $display("Test | left rotate 0x%h << %0d | 0x%h | 0x%h", in, shift,
(in << shift) | (in >> (32 - shift)), out_rotate);
        end

        // Shift values (generally) >32
        for (i = 0; i < 100; i = i + 1) begin
```

```
            in <= $urandom;
            shift <= $urandom;
            #1 $display("Test | left shift large 0x%h << %0d | 0x%h | 0x%h", in,
shift, in << shift, out_shift);
            #1 $display("Test | left rotate large 0x%h << %0d | 0x%h | 0x%h", in,
shift, (in << (shift % 32)) | (in >> (32 - (shift % 32))), out_rotate);
        end

        #1;
        $finish;
    end
endmodule
```

File: hdl/register.v

```
module register #(
    parameter BITS = 32
) (
    input [BITS - 1:0] q,
    output reg [BITS - 1:0] d,
    input clk,
    input clr, // active-low, asynchronous clear
    input en // write enable
);
    always @(posedge clk, negedge clr) begin
        if (clr == 1'b0)
            d <= {BITS{1'b0}};
        else if (en == 1'b1)
            d <= q;
        else
            d <= d;
    end
endmodule


`timescale 1ns/100ps
module register_test;

    reg [31:0] q;
    wire [31:0] d;

    reg clk, clr, en;

    register r ( .q(q), .d(d), .clk(clk), .clr(clr), .en(en) );

    // Clock
    initial begin
        clk <= 1'b0;
        forever #5 clk = ~clk;
    end

    // Test
```

```
    initial begin
        #7 // Offset so we're in the middle of the positive clock signal

        // Verify async clear
        en <= 1'b1;
        clr <= 1'b0;
        q <= 134;
        #1 $display("Test | async clear 0 after 1ns | en=1, clr=0, q=134, d=0 |
en=%d, clr=%d, q=%0d, d=%0d", en, clr, q, d);
        #9 $display("Test | async clear 0 after 10ns | en=1, clr=0, q=134, d=0 |
en=%d, clr=%d, q=%0d, d=%0d", en, clr, q, d);

        clr <= 1'b1;
        q <= 57;
        #1 $display("Test | async clear 1 after 1ns | en=1, clr=1, q=57, d=0 |
en=%d, clr=%d, q=%0d, d=%0d", en, clr, q, d);
        #9 $display("Test | async clear 1 after 10ns | en=1, clr=1, q=57, d=57 |
en=%d, clr=%d, q=%0d, d=%0d", en, clr, q, d);

        clr <= 1'b0;
        #1 $display("Test | async clear 0 after 1ns | en=1, clr=0, q=57, d=0 |
en=%d, clr=%d, q=%0d, d=%0d", en, clr, q, d);
        #9 $display("Test | async clear 0 after 10ns | en=1, clr=0, q=57, d=0 |
en=%d, clr=%d, q=%0d, d=%0d", en, clr, q, d);

        // Verify write enable
        q <= 183;
        clr <= 1'b1;
        #1 $display("Test | write enable 1 after 1ns | en=1, clr=1, q=183, d=0 |
en=%d, clr=%d, q=%0d, d=%0d", en, clr, q, d);
        #9 $display("Test | write enable 1 after 10ns | en=1, clr=1, q=183, d=183
| en=%d, clr=%d, q=%0d, d=%0d", en, clr, q, d);

        q <= 89;
        en <= 1'b0;
        #1 $display("Test | write enable 0 after 1ns | en=0, clr=1, q=89, d=183 |
en=%d, clr=%d, q=%0d, d=%0d", en, clr, q, d);
        #9 $display("Test | write enable 0 after 10ns | en=0, clr=1, q=89, d=183 |
en=%d, clr=%d, q=%0d, d=%0d", en, clr, q, d);

        q <= 555;
        en <= 1'b1;
        #1 $display("Test | write enable 0 after 1ns | en=1, clr=1, q=555, d=183 |
en=%d, clr=%d, q=%0d, d=%0d", en, clr, q, d);
        #9 $display("Test | write enable 0 after 10ns | en=1, clr=1, q=555, d=555
| en=%d, clr=%d, q=%0d, d=%0d", en, clr, q, d);

        $finish;
    end
endmodule
```

File: hdl/register_file.v

```
module register_file (
    input [31:0] write_data,
    input [3:0] write_addr,
    input [3:0] read_addr_a,
    input [3:0] read_addr_b,
    output [31:0] data_a,
    output [31:0] data_b,
    input clk,
    input clr
);
    // 16x 32-Bit Register File
    // Dual ported: each register connects to two multiplexers which determine the
output
    wire [31:0] data [15:0];

    // r0
    assign data[0] = 32'b0;

    genvar i;
    generate
        // Ignore r0, as it has a no-op (zero) connection
        for (i = 1; i < 16; i = i + 1) begin : gen_r
            register ri ( .q(write_data), .d(data[i]), .clk(clk), .clr(clr),
.en(write_addr == i) );
        end
    endgenerate

    assign data_a = data[read_addr_a];
    assign data_b = data[read_addr_b];

endmodule


`timescale 1ns/100ps
module register_file_test;

    reg [31:0] z;
    reg [3:0] addr_z, addr_a, addr_b;
    wire [31:0] a, b;
    reg clk, clr;

    register_file rf ( .write_data(z), .write_addr(addr_z), .read_addr_a(addr_a),
.read_addr_b(addr_b), .data_a(a), .data_b(b), .clk(clk), .clr(clr) );

    // Clock
    initial begin
        clk <= 1'b0;
        forever #5 clk = ~clk;
    end

    // Test
    initial begin
        #7 // Offset so we're in the middle of the positive clock signal
```

```
        clr <= 1'b1; // clr is assumed to work as it's only connected to the
internal register's clr line

        // Write some data
        z <= 853;
        addr_z <= 11;
        #10;
        z <= 124;
        addr_z <= 4;
        #10;
        z <= 888;
        addr_z <= 15;
        #10;
        z <= 999;
        addr_z <= 0;
        #10;

        addr_z <= 0;

        // Read data back again, checking both channels
        addr_a <= 11;
        addr_b <= 15;
        #1;
        $display("Test | register file read a1 | a=853 | a=%0d", a);
        $display("Test | register file read b1 | b=888 | b=%0d", b);

        #9;
        addr_a <= 4;
        addr_b <= 0;
        #1;
        $display("Test | register file read a2 | a=124 | a=%0d", a);
        $display("Test | register file read b2 | b=0 | b=%0d", b);

        $finish;
    end
endmodule
```

File: hdl/right_shift_32b.v

```
module right_shift_32b (
    input [31:0] in,
    input [31:0] shift,
    output [31:0] out,
    input is_rotate
);
    wire [31:0] shift_1, shift_2, shift_4, shift_8, shift_16;

    // Shift amount is a 32-bit unsigned value
    // For shifts >= 32, any of the upper bits = 0 indicates the result is zero
(this has no effect on rotates).
    wire is_zero;
    assign is_zero = !is_rotate && (| shift[31:5]);
```

```
    assign shift_1 = shift[0] ? { (is_rotate ? in[0] : 1'b0), in[31:1] } : in;
    assign shift_2 = shift[1] ? { (is_rotate ? shift_1[1:0] : 2'b0), shift_1[31:2]
} : shift_1;
    assign shift_4 = shift[2] ? { (is_rotate ? shift_2[3:0] : 4'b0), shift_2[31:4]
} : shift_2;
    assign shift_8 = shift[3] ? { (is_rotate ? shift_4[7:0] : 8'b0), shift_4[31:8]
} : shift_4;
    assign shift_16 = shift[4] ? { (is_rotate ? shift_8[15:0] : 16'b0),
shift_8[31:16] } : shift_8;
    assign out = is_zero ? 32'b0 : shift_16;

endmodule


// Testbench
`timescale 1ns/100ps
module right_shift_32b_test;

    reg [31:0] in;
    reg [31:0] shift;
    wire [31:0] out_shift, out_rotate;

    right_shift_32b _rs ( .in(in), .shift(shift), .out(out_shift),
.is_rotate(1'b0) );
    right_shift_32b _rr ( .in(in), .shift(shift), .out(out_rotate),
.is_rotate(1'b1) );

    integer i;

    initial begin
        // Shift values between [0, 32])
        for (i = 0; i < 100; i = i + 1) begin
            in <= $urandom;
            shift <= $urandom % 32;
            #1 $display("Test | right shift 0x%h >> %0d | 0x%h | 0x%h", in, shift,
in >> shift, out_shift);
            #1 $display("Test | right rotate 0x%h >> %0d | 0x%h | 0x%h", in,
shift, (in >> shift) | (in << (32 - shift)), out_rotate);
        end

        // Shift values (generally) >32
        for (i = 0; i < 100; i = i + 1) begin
            in <= $urandom;
            shift <= $urandom;
            #1 $display("Test | right shift large 0x%h >> %0d | 0x%h | 0x%h", in,
shift, in >> shift, out_shift);
            #1 $display("Test | right rotate large 0x%h >> %0d | 0x%h | 0x%h", in,
shift, (in >> (shift % 32)) | (in << (32 - (shift % 32))), out_rotate);
        end

        $finish;
    end
endmodule
```

File: hdl/ripple_carry_adder.v

```
/**
 * N-Bit Ripple Carry Adder (RCA)
 */
module ripple_carry_adder #(
    parameter BITS = 32
) (
    input [BITS - 1:0] a,
    input [BITS - 1:0] b,
    output [BITS - 1:0] sum,
    input c_in,
    output c_out
);

    // Internal carry line
    wire [BITS:0] carry;

    // Connect to c_in and c_out
    assign carry[0] = c_in;
    assign c_out = carry[BITS];

    genvar i;
    generate
        for (i = 0; i < BITS; i = i + 1) begin : gen_adder
            full_adder fa ( .a(a[i]), .b(b[i]), .sum(sum[i]), .c_in(carry[i]),
.c_out(carry[i + 1]) );
        end
    endgenerate
endmodule


`timescale 1ns/100ps
module ripple_carry_adder_test;

    // 5-Bit RCA : Small enough to test all possible inputs exhaustively (32 * 32
* 2 = 2048 inputs)
    reg [4:0] a, b;
    reg c_in;
    wire [5:0] sum;

    ripple_carry_adder #( .BITS(5) ) rca ( .a(a), .b(b), .sum(sum[4:0]),
.c_in(c_in), .c_out(sum[5]) );

    integer i, j, k;

    initial begin
        for (i = 0; i < 32; i = i + 1) begin
            for (j = 0; j < 32; j = j + 1) begin
                for (k = 0; k <= 1; k = k + 1) begin
                    a <= i;
```

```
                    b <= j;
                    c_in <= k;
                    #1 $display("Test | add %0d + %0d | %0d | %0d", a, b, i + j +
k, sum);
                end
            end
        end

        $finish;
    end
endmodule
```

File: hdl/signed_compliment.v

```
/**
 * N-Bit Signed 2's Compliment Negation x -> (-x)
 */
module signed_compliment #(
    parameter BITS = 32
) (
    input [BITS - 1:0] in,
    output [BITS - 1:0] out
);

    // Negation = Invert all bits and add one
    // Addition is performed with a single carry chain, similar to a RCA if b = 0
    wire [BITS - 1:0] carry;

    assign carry[0] = 1'b1; // c_in = 0
    assign carry[BITS - 1:1] = (~in[BITS - 2:0]) & carry[BITS - 2:0]; // carry
chain, ci+1 = xi & ci
    assign out = (~in) ^ carry; // summation, s = xi ^ ci

endmodule


`timescale 1ns/100ps
module signed_compliment_test;

    reg signed [4:0] in;
    wire signed [4:0] out;

    integer i;

    signed_compliment #( .BITS(5) ) sc ( .in(in), .out(out) );

    initial begin
        for (i = 0; i < 32; i = i + 1) begin
            in <= i;
            #1 $display("Test | compliment -%0d | %0d | %0d", in, out, -in);
        end
    end
```

```
endmodule
```