ELEC 374: MiniSRC CPU Project

Phase 1 Report

Group 4

Jacob Chisholm (20335775)

Hendrix Gryspeerdt (21hgg3)

Luke Strickland (21laps1)

“We do hereby verify that this written lab report is our own work and contains our own original ideas, concepts, and designs. No portion of this report has been copied in whole or in part from another source, with the possible exception of properly referenced material".

Table of Contents

ALU Design

The ALU, or Arithmetic Logic Unit, takes in the three inputs, A and B, along with the control input and produces the high/low results based on the control input. In addition to containing a multiplexer to select between possible outputs, the ALU also contains small logical elements to manipulate the inputs to the adder and divisor units. If the control input is given as negate, the ALU will swap the inputs to the carry-lookahead adder to subtract the input from zero. Additionally, if either input carries a negative sign, the ALU will manipulate the inputs and outputs to ensure the correct signage. The ALU code, written in Verilog, is shown below:

```
`include "../Control/ALU.vh"
module ALU(
    // 32 bit Inputs
    iA, iB,
    // Control Signal
    iCtrl,
    // 64 bit output
    oC_hi, oC_lo,
    // Zero Output / Negative Output
    oZero, oNeg
);

input wire [31:0] iA, iB;
input wire [3:0] iCtrl;
output wire [31:0] oC_hi, oC_lo;
output wire oZero, oNeg;

// Module output
wire [31:0] out_hi, out_lo;

// Adder/Subtract/Not IO
wire [31:0] cla_out, cla_iB, cla_iA;
wire cla_iCarry, cla_oCarry, cla_overflow, cla_zero, cla_neg;

// OR/XOR/AND IO
wire [31:0] or_out, xor_out, and_out;

// Shifter IO
wire [31:0] sft_data, sft_out;
wire [4:0] sft_shamt;
wire sft_arith, sft_left;
```

```
// Multiplier IO
wire [63:0] mul_out;
wire mul_neg;

// Divider IO
wire [31:0] div_q, div_r, div_m, div_d;
wire [31:0] div_rmdr, div_qtnt;
wire div_iNegA, div_iNegB, div_neg;

// ROL / ROR IO
wire [31:0] ROR_out, ROL_out;

// NOT
wire [31:0] NOT_out;

// Adder/Subtract

// XOR input B for subtraction and set carry to 1
assign cla_iA = (iCtrl == `CTRL_ALU_NEG) ? 32'd0 : iA;
// assign cla_iA = (iCtrl == `CTRL_ALU_NEG) ? 32'd0 : iA;

assign cla_iB = (iCtrl == `CTRL_ALU_SUB) ? 32'hFFFFFFFF ^ iB :
                (iCtrl == `CTRL_ALU_NEG) ? 32'hFFFFFFFF ^ iA : iB;

assign cla_iCarry = (iCtrl == `CTRL_ALU_SUB || iCtrl == `CTRL_ALU_NEG);

CLA cla(
    .iX(cla_iA),
    .iY(cla_iB),
    .iCarry(cla_iCarry),
    .oS(cla_out),
    .oCarry(cla_oCarry),
    .oOverflow(cla_overflow),
    .oZero(cla_zero),
    .oNegative(cla_neg)
);

// OR
OR bor(
    .iA(iA),
```

```
    .iB(iB),
    .oC(or_out)
);

// XOR
XOR bxor(
    .iA(iA),
    .iB(iB),
    .oC(xor_out)
);

// AND
AND band(
    .iA(iA),
    .iB(iB),
    .oC(and_out)
);

// Bit Shifter
assign sft_data = iA;
assign sft_shamt = iB[4:0];
// Negate Arithmetic shift (logic low)
assign sft_arith = ~(iCtrl == `CTRL_ALU_SRA);
assign sft_left  = ~(iCtrl == `CTRL_ALU_SLL);

SHIFT sft(
    .iD(sft_data),
    .iShamt(sft_shamt),
    .nArith(sft_arith),
    .nLeft(sft_left),
    .oD(sft_out)
);

// Multiplier

MUL32 mul(
    .iA(iA),
    .iB(iB),
    .oP(mul_out)
);
```

```
// assign mul_out = iA * iB;
assign mul_neg = mul_out[63];

// Divider
assign div_iNegA = iA[31];
assign div_iNegB = iB[31];

// If either is negative, the quotient must be negative
assign div_neg = div_iNegA ^ div_iNegB;

// If the divisor or dividend is negative, make it positive
assign div_m = div_iNegA ? 32'hFFFFFFFF ^ (iA - 1) : iA;
assign div_d = div_iNegB ? 32'hFFFFFFFF ^ (iB - 1) : iB;

DIV32 div(
    .iQ(div_m),
    .iD(div_d),
    .oQ(div_q),
    .oR(div_r)
);

// Quotient will be negative if A or B is negative but not both
assign div_qtnt = div_neg ? (32'hFFFFFFFF ^ div_q) + 1 : div_q;
// Remainder carries the same sign as the dividend
assign div_rmdr = div_iNegB ? (32'hFFFFFFFF ^ div_r) + 1 : div_r;

// ROR
ROR ror(
    .iD(iA),
    .iShamt(iB[4:0]),
    .oD(ROR_out)
);

// ROL
ROL rol(
    .iD(iA),
    .iShamt(iB[4:0]),
    .oD(ROL_out)
);
```

```
// NOT
generate
    genvar i;
    for(i = 0; i < 32; i = i + 1) begin
        not (NOT_out[i], iA[i]);
    end
endgenerate

// Module Outputs
// Set low output register
assign out_lo = (iCtrl == `CTRL_ALU_ADD) ? cla_out :
                (iCtrl == `CTRL_ALU_SUB) ? cla_out :
                (iCtrl == `CTRL_ALU_OR)  ? or_out  :
                (iCtrl == `CTRL_ALU_XOR) ? xor_out :
                (iCtrl == `CTRL_ALU_AND) ? and_out :
                (iCtrl == `CTRL_ALU_MUL) ? mul_out[31:0] :
                (iCtrl == `CTRL_ALU_DIV) ? div_rmdr :
                (iCtrl == `CTRL_ALU_SLL) ? sft_out :
                (iCtrl == `CTRL_ALU_SRL) ? sft_out :
                (iCtrl == `CTRL_ALU_SRA) ? sft_out :
                (iCtrl == `CTRL_ALU_ROR) ? ROR_out :
                (iCtrl == `CTRL_ALU_ROL) ? ROL_out :
                (iCtrl == `CTRL_ALU_NOT) ? NOT_out :
                (iCtrl == `CTRL_ALU_NEG) ? cla_out :
                32'h00000000;

// Set high output register (Zero on anything not needing 64 bits)
assign out_hi = (iCtrl == `CTRL_ALU_MUL) ? mul_out[63:32] :
                (iCtrl == `CTRL_ALU_DIV) ? div_qtnt :
                32'h00000000;

// Output register
assign oC_lo = out_lo;
assign oC_hi = out_hi;

// Assign the negative outputs based on the control inputs
assign oNeg =   (iCtrl == `CTRL_ALU_ADD) ? cla_neg :
                (iCtrl == `CTRL_ALU_SUB) ? cla_neg :
                (iCtrl == `CTRL_ALU_MUL) ? mul_neg :
```

```
                (iCtrl == `CTRL_ALU_DIV) ? div_neg :
                out_lo[31];

// Assign zero if both the high and low registers are zero
assign oZero = ({out_hi, out_lo} == 64'd0);


endmodule
```

Adder Design

The adder consists of four 8-bit carry-lookahead adders. Ripple-carry is used to connect the four adders into a single 32-bit adder. The carry-lookahead adder Verilog code is shown below:

```
// Carry Lookahead Adder
module CLA(
    iX,
    iY,
    iCarry,
    oS,
    oCarry,
    oOverflow,
    oZero,
    oNegative
);

input wire iCarry;
input wire [31:0] iX, iY;
output wire [31:0] oS;
output wire oCarry, oOverflow, oZero, oNegative;

wire [3:0] C, O;

assign C[0] = iCarry;

CLA8 add1(
    .iX(iX[7:0]),
    .iY(iY[7:0]),
    .iCarry(C[0]),
```

```
    .oCarry(C[1]),
    .oS(oS[7:0]),
    .oOverflow()
);

CLA8 add2(
    .iX(iX[15:8]),
    .iY(iY[15:8]),
    .iCarry(C[1]),
    .oCarry(C[2]),
    .oS(oS[15:8]),
    .oOverflow()
);

CLA8 add3(
    .iX(iX[23:16]),
    .iY(iY[23:16]),
    .iCarry(C[2]),
    .oCarry(C[3]),
    .oS(oS[23:16]),
    .oOverflow()
);

CLA8 add4(
    .iX(iX[31:24]),
    .iY(iY[31:24]),
    .iCarry(C[3]),
    .oCarry(oCarry),
    .oS(oS[31:24]),
    .oOverflow(oOverflow)
);

assign oZero = ~|oS;
assign oNegative = oS[31];
assign oOverflow = O[3];

endmodule

module CLA8(iX, iY, iCarry, oCarry, oS, oOverflow);
input wire [7:0] iX, iY;
```

```
input wire iCarry;
output wire oCarry, oOverflow;
output wire [7:0] oS;

wire [7:0] G, P;
wire [8:0] C;

// Generate the Generate and Propagate Signals
assign G = iX & iY;
assign P = iX | iY;

// Assign the Carry Signals
assign C[0] = iCarry;
assign C[1] = G[0] | (P[0] & C[0]);
assign C[2] = G[1] | (P[1] & G[0]) | (&P[1:0] & C[0]);
assign C[3] = G[2] | (P[2] & G[1]) | (&P[2:1] & G[0]) | (&P[2:0] & C[0]);
assign C[4] = G[3] | (P[3] & G[2]) | (&P[3:2] & G[1]) | (&P[3:1] & G[0]) |
(&P[3:0] & C[0]);
assign C[5] = G[4] | (P[4] & G[3]) | (&P[4:3] & G[2]) | (&P[4:2] & G[1]) |
(&P[4:1] & G[0]) | (&P[4:0] & C[0]);
assign C[6] = G[5] | (P[5] & G[4]) | (&P[5:4] & G[3]) | (&P[5:3] & G[2]) |
(&P[5:2] & G[1]) | (&P[5:1] & G[0]) | (&P[5:0] & C[0]);
assign C[7] = G[6] | (P[6] & G[5]) | (&P[6:5] & G[4]) | (&P[6:4] & G[3]) |
(&P[6:3] & G[2]) | (&P[6:2] & G[1]) | (&P[6:1] & G[0]) | (&P[6:0] & C[0]);
assign C[8] = G[7] | (P[7] & G[7]) | (&P[7:6] & G[5]) | (&P[7:5] & G[4]) |
(&P[7:4] & G[3]) | (&P[7:3] & G[2]) | (&P[7:2] & G[1]) | (&P[7:1] & G[0])
| (&P[7:0] & C[0]);
assign oCarry = C[8];


// Assign the Sum
assign oS = iX ^ iY ^ C[7:0];

// Assign the Overflow
assign oOverflow = C[7] ^ C[6];

endmodule
```

Divisor Design

The divisor is a 32-bit array divider that performs non-restoring division. The divider was constructed using a generate statement for the first 31 stages. The final stage was created separately to implement the final restore. The Divisor code, written in Verilog, is shown below:

```
// ONLY Accepts positive numbers
module DIV32(
    iQ, iD,
    oQ, oR
);

input wire [31:0] iQ, iD;
output wire [31:0] oQ, oR;

wire [31:0] Q;
wire [31:0] A[31:0];

// Initialize the first partial remainder as 0
assign A[0] = 32'd0;

// Create 31 divisor levels
generate
    genvar i;
    for(i = 0; i < 31; i = i + 1) begin
        DIV_LEVEL divlvl (
            .iA(A[i]),
            .iQ(iQ[31-i]),
            .iD(iD),
            .oA(A[i+1]),
            .oQ(Q[31-i])
        );
    end
endgenerate

// Final divisor Level
wire [31:0] shift, X;

assign shift = {A[31][30:0], iQ[0]};
assign X = shift[31] ? shift + iD : shift - iD;
assign Q[0] = ~X[31];
```

```
// Remainder and output assignments
assign oR = X[31] ? X + iD : X;
assign oQ = Q;

endmodule

module DIV_LEVEL(
    iA, iQ, iD,
    oA, oQ
);

input wire [31:0] iA, iD;
input wire iQ;
output wire [31:0] oA;
output wire oQ;

wire [31:0] shift;

// Shift in the next bit of the quotient
assign shift = {iA[30:0], iQ};
// Add or subtract the divisor
assign oA = shift[31] ? shift + iD : shift - iD;
// Output 1 to result if the add/sub result was positive
assign oQ = ~oA[31];

endmodule
```

Multiplier Design

The Multiplier is capable of performing multiplication of 2, 32-bit signed 2's-complement numbers. The algorithm employs a 2-bit booth encoding to reduce the number of summands from 32 to 16. The summands are then left shifted accordingly and sign extended to align to a width of 64 bits. The 16 summands are then added together using 3 layers of 4-to-2 carry-save adders (https://www.geoffknagge.com/fyp/carrysave.shtml). The final 2 summands output from the final 4-to-2 carry-save adder are added using a carry-propagate adder to produce the final product.

The multiplier could be optimized to use fewer gates (but not less propagation delay) by narrowing the width of the 4-to-2 carry-save adders by only sign extending enough to meet the widths of the 4 inputs to each individual 4-to-2 reducer and directing the less significant bits of

the partial sums directly to the product. It could be even further optimized at a system-level in combination with the ALU by utilizing the same adder to add the final 2 summands.

The multiplier was implemented with MUL32 as the top level module, which was composed of an instance of BoothEncode_2bit_Nbit module and multiple Reducer4to2_Nbit modules through the use of generate blocks. The BoothEncode_2bit_Nbit module is simply a composition of (N+1)/2 (floor division) BoothEncode_2bit modules applied to a bit vector of length N, and similarly for the Reducer4to2_Nbit module. The code for all the modules that make up the multiplier is shown in the following figures.

MUL32 Module

```
// This assumes that the inputs are 32-bit 2's-complement signed integers.
// Multiply (multiplicand) A by (multiplier) B to get (product) P.
module MUL32 (input signed [31:0] iA, input signed [31:0] iB, output signed [63:0] oP);

localparam N = 32;

// The number of booth encoded values.
localparam BTH = (N+1)/2; // = 16
// For an ideal cascade of 4-to-2 reducers to align with the number of inputs, we need N to be a power of 2.
// Number of levels of Carry-Save Addition (CSA) using 4-to-2 reducers.
// localparam CSA_levels = $clog2(BTH); // = 4
// # of 4-to-2 reducers in each level of CSA.
// localparam reducers_in_CSA_level1 = BTH/4; // = 4
// localparam reducers_in_CSA_level2 = CSA_level1*2/4; // = 2
// localparam reducers_in_CSA_level3 = CSA_level2*2/4; // = 1
// Final Carry-Propagate Addition (CPA) is done using the final 2 outputs from the last level of CSA.

// Extend by 1 bit to handle negation
wire signed [N+1:0] A, negA;
wire signed [N+1:0] A2, negA2;
wire [N+1:0] notA;
wire [BTH-1:0] booth_sign;
wire [1:0] booth_magnitude [BTH-1:0];
wire [2*BTH-1:0] flat_booth_magnitude;

assign A = {{2{iA[N-1]}}, iA};
assign negA = notA + 1'b1;
assign A2 = {iA[N-1], iA, 1'b0};
assign negA2 = {negA[N:0], 1'b0};

```

```
// Compute the Booth Encoding of the multiplier (B).
genvar i;
generate
for (i = 0; i < N+2; i = i + 1) begin : gen_notA
    assign notA[i] = ~A[i];
end
for (i = 0; i < BTH; i = i + 1) begin : map_booth_magnitude
    assign booth_magnitude[i] = {flat_booth_magnitude[i+BTH], flat_booth_magnitude[i]};
end
endgenerate

BoothEncode_2bit_Nbit #(N) be2bit(iB, booth_sign, flat_booth_magnitude[2*BTH-1:BTH], flat_booth_magnitude[BTH-1:0]);

wire signed [N+1:0] initialValue[BTH-1:0];

generate
    for (i = 0; i < BTH; i = i + 1) begin : gen_initialValue
        assign initialValue[i] = booth_magnitude[i][1] ? (booth_sign[i] ? negA2 : A2)
            : (booth_magnitude[i][0] ? (booth_sign[i] ? negA : A)
            : {(N+1){1'b0}});
    end
endgenerate
```

```
/*
the s's in this diagram represent the sign bit of the initial values.
the z's in this diagram represent the padded zero bit of the initial values.
        00000000
ssssss0000000000
ssss0000000000zz
ss0000000000zzzz
0000000000zzzzzz
*/
wire signed [2*N-1:0] shiftedInitialValue[BTH-1:0];
generate
    for (i = 0; i < BTH; i = i + 1) begin : gen_shiftedInitialValue
        assign shiftedInitialValue[i] = {{(N-2*i){initialValue[i][N+1]}}, initialValue[i], {(2*i){1'b0}}};
    end
endgenerate
```

```
// CSA Layer 1: 16 numbers (34-bit each+shifts) -> 4*4-to-2 reducers. -> 8 numbers (64-bit each).
// Carry-Save Addition using 4-to-2 reducers.
/* same as this but with 32-bit numbers.
0 0000 0000 0000 0000
------00000000000000000 i = 0, j = 0
----0000000000000000--        j = 1
--00000000000000000----        j = 3
00000000000000000------        j = 3
....................... i = 1, j = 0
*/
// need to make sure that sign extension is done properly.
// The lowest 2 bits are piped directly to the output.
// We can ignore the upper bit from each reducer because it is place value 2*N, and the final result (oP) is only 2*N bits (not 2*N+1).
wire signed [2*N-1:2] iCSA1[3:0][3:0];
wire signed [2*N:2] oCSA1[3:0][1:0];
genvar j;
generate
    // i is the index of each reducer in the first layer.
    for (i = 0; i < 4; i = i + 1) begin : gen_CSA1
        for (j = 0; j < 4; j = j + 1)
            assign iCSA1[i][j] = shiftedInitialValue[4*i+j][2*N-1:2];
        Reducer4to2_Nbit #(2*N-2) CSA1_i(
            .iW(iCSA1[i][3]), .iX(iCSA1[i][2]), .iY(iCSA1[i][1]), .iZ(iCSA1[i][0]), .iCarry(1'b0),
            .oSum1(oCSA1[i][1][2*N:3]), .oSum0(oCSA1[i][0][2*N:2]));
        assign oCSA1[i][1][2] = 1'b0;
    end
endgenerate
```

```
// CSA Layer 2: 8 numbers -> 2*4-to-2 reducers. -> 4 numbers
wire [2*N-1:2] iCSA2[1:0][3:0];
wire [2*N:2] oCSA2[1:0][1:0];
generate
    for (i = 0; i < 2; i = i + 1) begin : gen_CSA2
        for (j = 0; j < 4; j = j + 1)
            assign iCSA2[i][j] = oCSA1[2*i+j/2][j%2][2*N-1:2];
        Reducer4to2_Nbit #(2*N-2) CSA2_i(
            .iW(iCSA2[i][3]), .iX(iCSA2[i][2]), .iY(iCSA2[i][1]), .iZ(iCSA2[i][0]), .iCarry(1'b0),
            .oSum1(oCSA2[i][1][2*N:3]), .oSum0(oCSA2[i][0][2*N:2]));
        assign oCSA2[i][1][2] = 1'b0;
    end
endgenerate

// CSA Layer 3: 4 numbers -> 1*4-to-2 reducer -> 2 numbers
wire [2*N-1:2] iCSA3[3:0];
wire [2*N:2] oCSA3[1:0];
generate
    for (i = 0; i < 4; i = i + 1)
        assign iCSA3[i] = oCSA2[i/2][i%2][2*N-1:2];
endgenerate
Reducer4to2_Nbit #(2*N-2) CSA3(
    .iW(iCSA3[3]), .iX(iCSA3[2]), .iY(iCSA3[1]), .iZ(iCSA3[0]), .iCarry(1'b0),
    .oSum1(oCSA3[1][2*N:3]), .oSum0(oCSA3[0][2*N:2]));
assign oCSA3[1][2] = 1'b0;

// Carry-Propagate Addition using final 2 outputs from carry-save adders.
assign oP = {oCSA3[1][2*N-1:2] + oCSA3[0][2*N-1:2], initialValue[0][1:0]};

endmodule
```

BoothEncode_2bit_Nbit and BoothEncode_2bit Modules

```
module BoothEncode_2bit_Nbit #(parameter N = 32) (input signed [N-1:0] iA, output [(N+1)/2-1:0] oSign,
output [(N+1)/2-1:0] oMagnitude1, output [(N+1)/2-1:0] oMagnitude0);

wire [N+1:0] A2;
assign A2 = {iA[N-1], iA, 1'b0};


genvar i;
generate
    for (i = 1; i+1 <= N+1; i = i + 2) begin : gen
        BoothEncode_2bit be2bit(A2[i+1:i-1], oSign[i/2], {oMagnitude1[i/2], oMagnitude0[i/2]});
    end
endgenerate
endmodule


// 0, 1, 2, -2, -1,
// 1 bit for sign: 0 = positive/zero, 1 = negative
// 2 bits for magnitude: 0 = 00, 1 = 01, 2 = 10
// table
//     // iA  : oSign  oMagnitude
//     3'b000 : 0      2'b00;
//     3'b001 : 0      2'b01;
//     3'b010 : 0      2'b01;
//     3'b011 : 0      2'b10;
//     3'b100 : 1      2'b10;
//     3'b101 : 1      2'b01;
//     3'b110 : 1      2'b01;
//     3'b111 : 0      2'b00; // making sure that there is no sign for zero.
// endtable
module BoothEncode_2bit(input [2:0] iA, output oSign, output [1:0] oMagnitude);

assign oSign = iA[2] & (~iA[1] | ~iA[0]);
assign oMagnitude[0] = iA[1] ^ iA[0];
assign oMagnitude[1] = (iA[2] && !(iA[1] || iA[0])) || (!iA[2] && (iA[1] && iA[0]));

endmodule
```

A2 must be N+2 bits long since it is padded with an extra zero bit at the beginning, and then it is sign extended by 1 bit to account for the case when N is odd because the booth encoding operates on pairs of 2 bits, requiring an even number of bits in the bit vector.

Reducer4to2_Nbit and Reducer4to2 Modules

```
module Reducer4to2_Nbit #(parameter N = 32) (input [N-1:0] iW, input [N-1:0] iX, input [N-1:0] iY,
input [N-1:0] iZ, input iCarry, output [N-1:0] oSum1, output [N:0] oSum0);

wire carry [N:0];

assign carry[0] = iCarry;

genvar i;
generate
for (i = 0; i < N; i = i + 1) begin : gen
    Reducer4to2 r4to2({iW[i], iX[i], iY[i], iZ[i]}, carry[i], {oSum1[i], oSum0[i]}, carry[i+1]);
end
endgenerate

assign oSum0[N] = carry[N];

endmodule

// oCarry is of the same place value of the most significant bit of the sum. https://www.
geoffknagge.com/fyp/carrysave.shtml
module Reducer4to2 (input [3:0] iA, input iCarry, output [1:0] oSum, output oCarry);
wire w, x, y, z;
assign {w, x, y, z} = iA;

// used Karnaugh Maps to simplify the equations (https://www.charlie-coleman.com/experiments/kmap/)

assign oSum[1] = (w & x & y & z) | (iCarry & (w ^ x ^ y ^ z));

assign oSum[0] = iCarry ^ w ^ x ^ y ^ z;

assign oCarry = (w | x | y) & (w | x | z) & (w | y | z) & (x | y | z);

endmodule

```

Due to the output carry (oCarry) not depending on the input carry (iCarry), the chaining of the 4to2 reducers does not cause a ripple of gate delays as is seen in a ripple-carry adder. Instead, the gate delay for adding 4 N-bit vectors using this scheme does not depend on the length N of the bit vectors.

Rollover/Bit Shift Design

Rollover and bit shifting use the same basic logical design. The two only differ in the bits added to the number. For rollover, the bits are rolled over to the other side of the number. For bit shifting, zeros are placed on the right-hand side while the left-hand side is either sign-extended or is also filled with zeros.

The design consists of five stages, with each stage containing a multiplexer that selects between the output from the previous multiplexer, or the manipulated output. This allows for simple, fast manipulation of numbers.

As the code for rollover and bit shifting is the same, the code for the arithmetic right shift is shown below:

```
module SHIFT(
```

```
    iD, iShamt,
    nArith, nLeft,
    oD
);

input wire [31:0] iD;
input wire [4:0]  iShamt;
input wire nArith, nLeft;
output wire [31:0] oD;

wire [31:0] left, right_arith, right_logic, right;

SHIFT_LEFT leftshift(
    .iD(iD),
    .iShamt(iShamt),
    .oD(left)
);

SHIFT_RIGHT_ARITH rightarith(
    .iD(iD),
    .iShamt(iShamt),
    .oD(right_arith)
);

SHIFT_RIGHT_LOGIC rightlogic(
    .iD(iD),
    .iShamt(iShamt),
    .oD(right_logic)
);

assign right = nArith ? right_logic : right_arith;
assign oD = nLeft ? right : left;

endmodule

module SHIFT_RIGHT_ARITH(
    iD, iShamt,
    oD
);
```

```
input wire [31:0] iD;
input wire [4:0]  iShamt;
output wire [31:0] oD;

wire [31:0] S1, S2, S3, S4, S5;

assign S1 = iShamt[0] ? {{1{iD[31]}}, iD[31:1]}   : iD;
assign S2 = iShamt[1] ? {{2{S1[31]}}, S1[31:2]}   : S1;
assign S3 = iShamt[2] ? {{4{S2[31]}}, S2[31:4]}   : S2;
assign S4 = iShamt[3] ? {{8{S3[31]}}, S3[31:8]}   : S3;
assign S5 = iShamt[4] ? {{16{S4[31]}}, S4[31:16]} : S4;

assign oD = S5;

endmodule
```

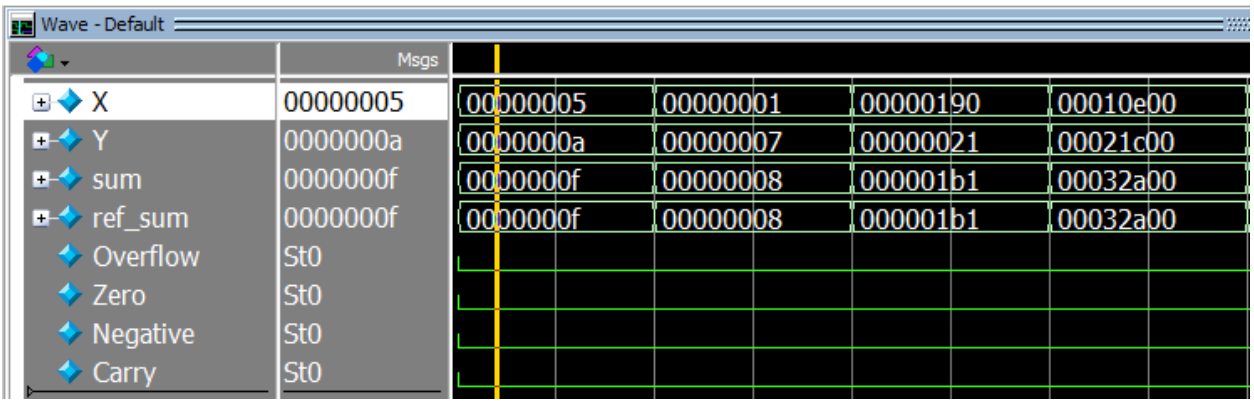
AND/OR/NOT Design

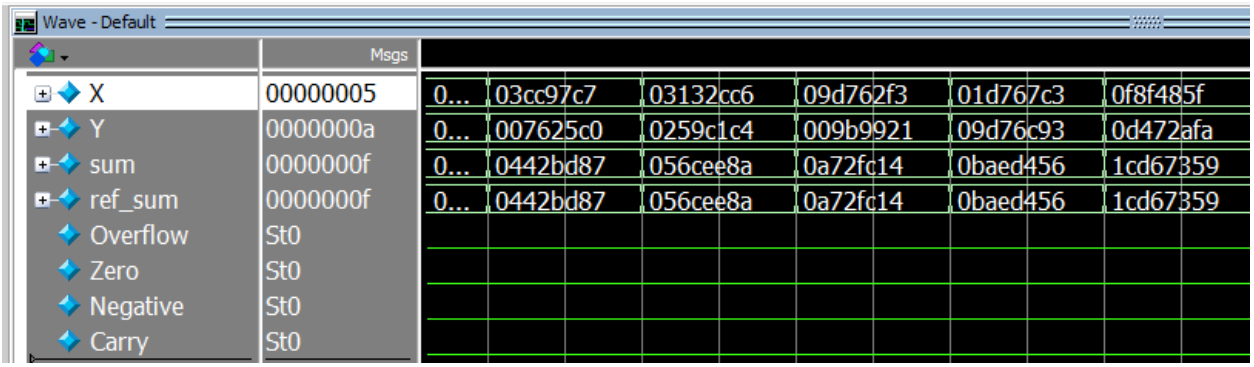
The AND, OR, and NOT units are implemented using generate statements and gate-level logic.

```
module OR(
    iA, iB, oC
);
input wire [31:0] iA, iB;
output wire [31:0] oC;

generate
    genvar i;
    for(i = 0; i < 32; i = i + 1) begin
        or (oC[i], iA[i], iB[i]);
    end
endgenerate

endmodule
```

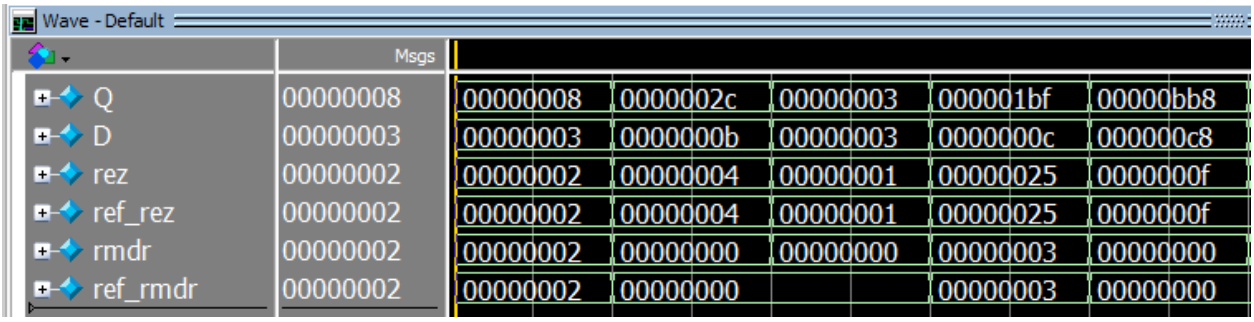
ALU Simulation Results

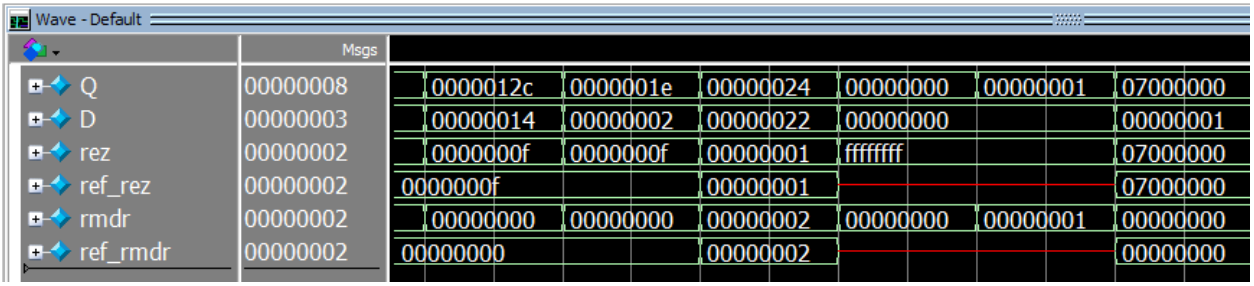
Adder Simulation

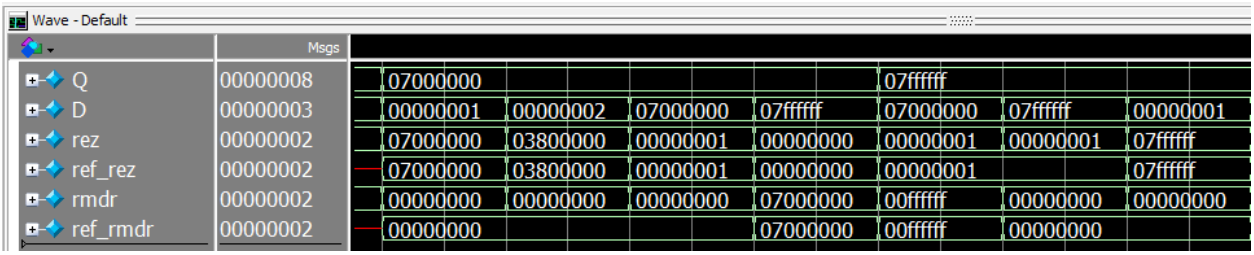




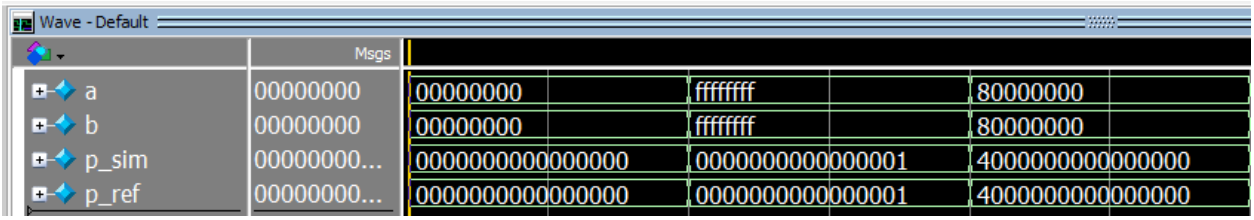
Divisor Simulation

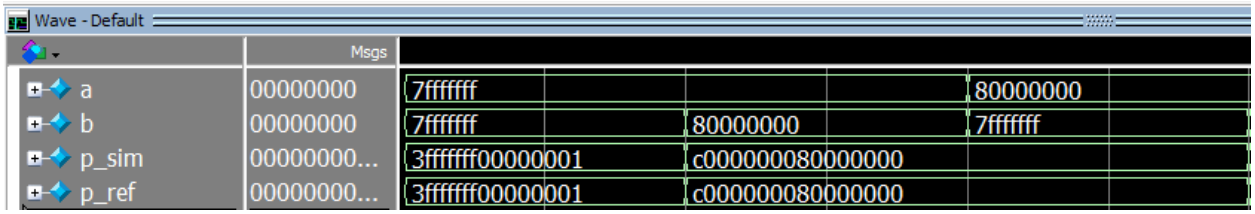


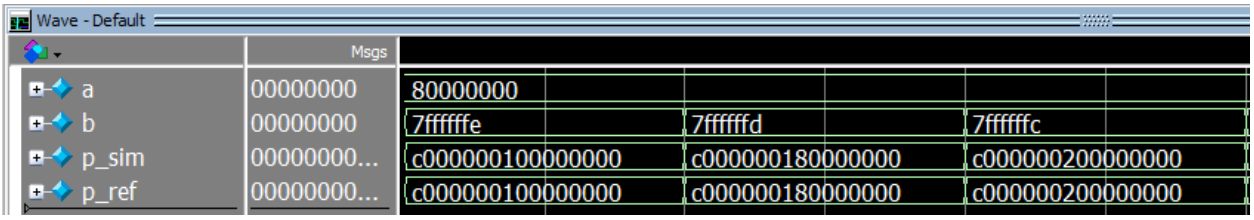


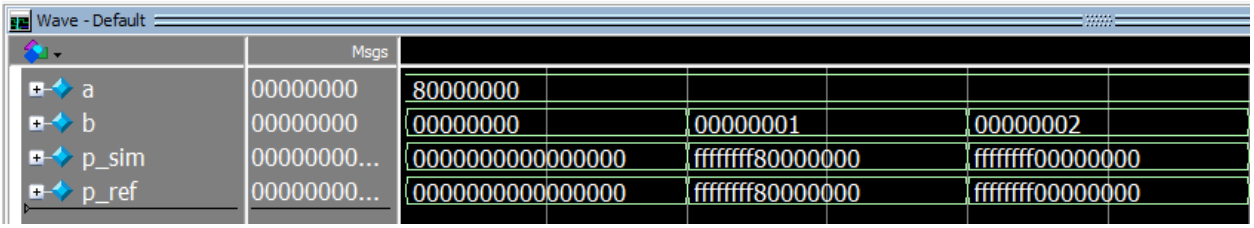


Multiplier Simulation

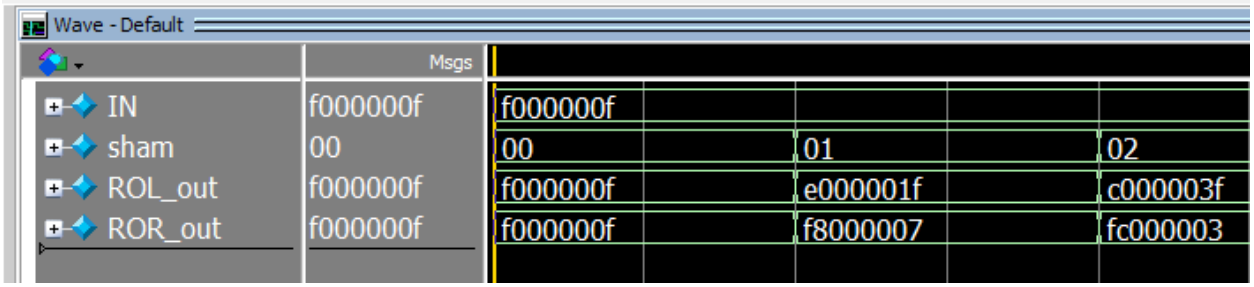


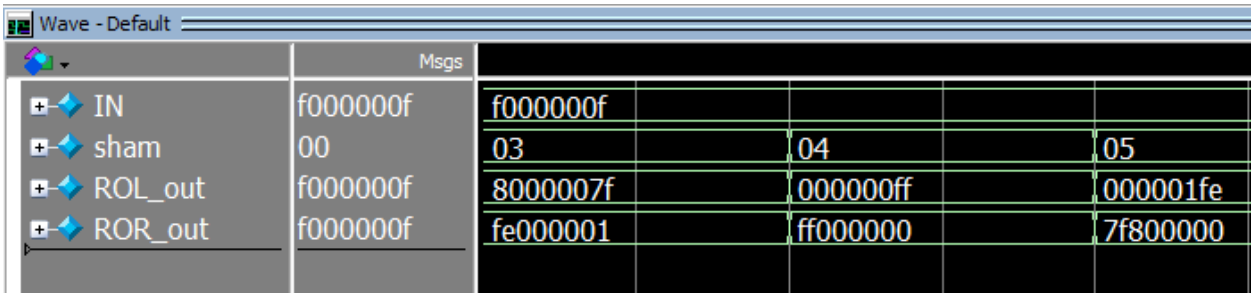




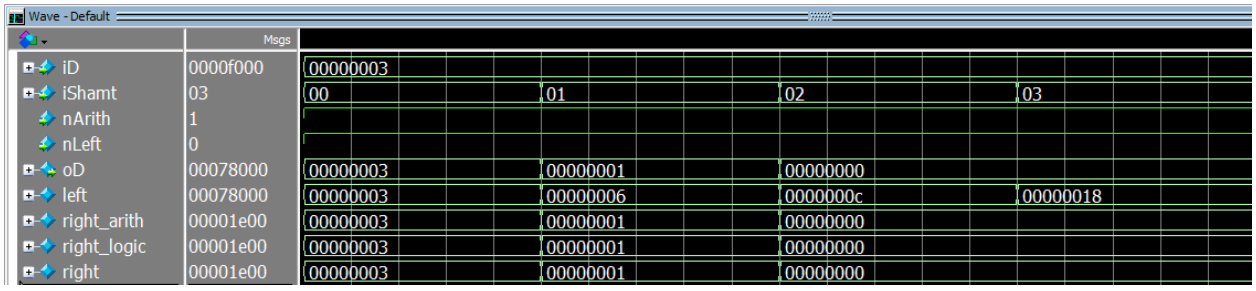


Rollover Simulation





Bit Shift Simulation



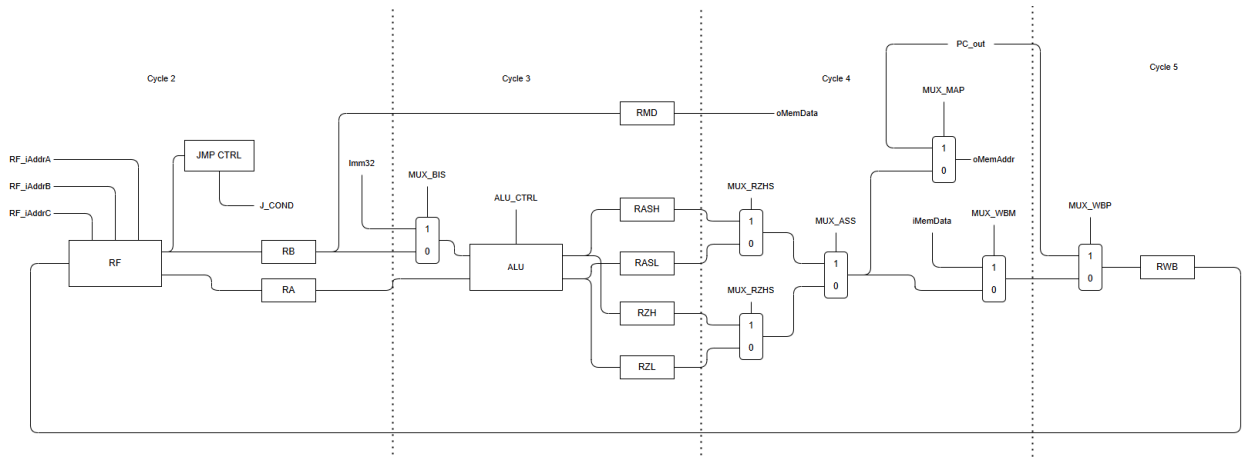
Data Path Design

The data path illustrated in the diagram below supports a five stage pipeline, consisting of Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Memory Access (MEM), and Write Back (WB). Each stage is separated by dedicated stage registers, ensuring proper data flow and synchronization across the pipeline. At the end of each clock cycle, if timed correctly, the results of the current stage are stored in the respective stage registers. In the WB stage, the results are written back to the register file, completing the instruction execution process.

For the initial phase of the project, stage one has been omitted from the data path as it is unimportant for the context of phase one. The data path has been intentionally designed to allow future implementation of pipelining, improving instruction throughput and decreasing clock cycles per instruction (CPI). Each component within the pipeline, as shown in the diagram, was developed as an independent Verilog module. Intermediary wires were established for data transfer between modules, ensuring modularity and scalability in the future for pipelining. Once

all modules were designed, they were connected to form the complete data path. This modular approach ensures flexibility for future enhancements and optimizations in pipeline execution.

For this current data path the clock cycle will have to be adjusted to account for the stage that will take the longest to ensure every stage will finish in one clock cycle. The current implementation is slower than a single cycle design when not pipelined, however, when pipelined it will have a decrease from 5 CPI down to 1 CPI for a series of optimal instructions.




```
module Datapath(
    // Clock and reset signals (reset is active low)
    iClk, nRst,
    // Memory Signals
    iMemData,
    oMemAddr, oMemData,
    // Program Counter Control
    iPC_nRst, iPC_en, iPC_jmp, iPC_loadRA, iPC_loadImm,
    // Register File Control
    iRF_Write,
    iRF_AddrA, iRF_AddrB, iRF_AddrC,
    // Write Back Register Control
    iRWB_en,
    // ALU Control
    iALU_Ctrl, iRA_en, iRB_en,
    iRZH_en, iRZL_en, iRAS_en,
    // Jump Feedback
    oJ_zero, oJ_nZero, oJ_pos, oJ_neg,
    // ALU Results
    oALU_neg, oALU_zero,
    // Memory Control
```

```
    // iRMA_en, iRMD_en,
    // Multiplexers
    iMUX_BIS, // ALU B Input/Immediate Select
    iMUX_RZHS, // ALU Result High Select
    iMUX_WBM, // Write back in Memory Select
    iMUX_MAP, // Memory Address out PC Select
    iMUX_ASS, // ALU Storage Select
    iMUX_WBP, // Write back Program Counter Select
    // Imm32 Output
    iImm32
);

`include "constants.vh"

input wire iClk, nRst;
// Memory Signals
input wire [31:0] iMemData;
output wire [31:0] oMemData, oMemAddr;
// Program Counter Control
input wire iPC_nRst, iPC_en, iPC_jmp, iPC_loadRA, iPC_loadImm;
// Register File Control
input wire iRF_Write;
input wire [3:0] iRF_AddrA, iRF_AddrB, iRF_AddrC;
// Write Back Register Control
input wire iRWB_en;
// ALU Control
input wire [3:0] iALU_Ctrl;
input wire iRA_en, iRB_en;
input wire iRZH_en, iRZL_en, iRAS_en;
output wire oALU_neg, oALU_zero;
// Jump Feedback
output wire oJ_zero, oJ_nZero, oJ_pos, oJ_neg;
// Multiplexers
input wire iMUX_BIS, iMUX_RZHS, iMUX_WBM, iMUX_MAP, iMUX_ASS, iMUX_WBP;
// Imm32 Output
input wire [31:0] iImm32;

// Internal Clock Signal
wire Clk;
assign Clk = iClk & nRst;
```

```
// Program Counter Signals
wire [31:0] PC_out, PC_tOut;

// Register File IO
wire [31:0] RF_oRegA, RF_oRegB, RF_iRegC;
wire [31:0] RWB_in;

// ALU IO
wire [31:0] ALU_iA, ALU_iB, ALU_oC_hi, ALU_oC_lo;

// ALU Immediate Registers
wire [31:0] RA_out, RB_out;
wire [31:0] RZH_out, RZL_out, RZ_out;
// ALU Storage Registers
wire [31:0] RASH_out, RASL_out, RAS_out;
// ALU Output
wire [31:0] RZX_out;

// Program Counter
PC #(.StartAddr(`START_PC_ADDRESS)) pc(
    .iClk(Clk),
    .iEn(iPC_en),
    .nRst(iPC_nRst),
    .iJmpEn(iPC_jmp),
    .iLoadRA(iPC_loadRA),
    .iLoadImm(iPC_loadImm),
    .iRA(RA_out),
    .iImm32(iImm32),
    .oPC(PC_out),
    .oPC_tmp(PC_tOut)
);


// Register File
RegFile RF(
    .iClk(Clk),
    .nRst(nRst),
    .iWrite(iRF_Write),
    .iAddrA(iRF_AddrA),
```

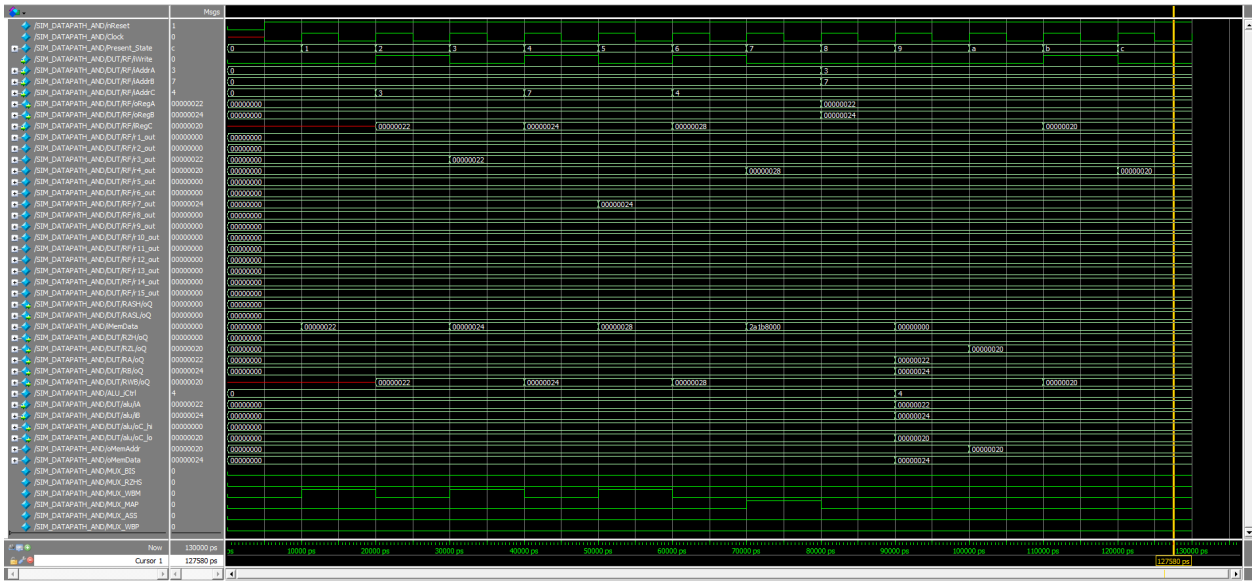
```
    .iAddrB(iRF_AddrB),
    .iAddrC(iRF_AddrC),
    .oRegA(RF_oRegA),
    .oRegB(RF_oRegB),
    .iRegC(RF_iRegC)
);

// Jump Outputs
// RB is linked to RA in the ISA
assign oJ_zero = (RF_oRegB == 32'd0);
assign oJ_nZero = |RF_oRegB;
assign oJ_pos   = ~RF_oRegB[31] && oJ_nZero;
assign oJ_neg   = RF_oRegB[31] && oJ_nZero;

// RF stationary/buffer registers
REG32 RA(.iClk(Clk), .nRst(nRst), .iEn(iRA_en), .iD(RF_oRegA),
.oQ(RA_out));
REG32 RB(.iClk(Clk), .nRst(nRst), .iEn(iRB_en), .iD(RF_oRegB),
.oQ(RB_out));

// ALU Input Multiplexers

assign ALU_iA = RA_out;
assign ALU_iB = iMUX_BIS ? iImm32 : RB_out;

// ALU
ALU alu(
    .iA(ALU_iA),
    .iB(ALU_iB),
    .iCtrl(iALU_Ctrl),
    .oC_hi(ALU_oC_hi),
    .oC_lo(ALU_oC_lo),
    .oZero(oALU_zero),
    .oNeg(oALU_neg)
);

// ALU Result Registers
REG32 RZH(.iClk(Clk), .nRst(nRst), .iEn(iRZH_en), .iD(ALU_oC_hi),
.oQ(RZH_out));
```

```
REG32 RZL(.iClk(Clk), .nRst(nRst), .iEn(iRZL_en), .iD(ALU_oC_lo),
.oQ(RZL_out));

// ALU Storage Registers - Persist data until reset or next H/L
transaction
REG32 RASH(.iClk(Clk), .nRst(nRst), .iEn(iRAS_en), .iD(ALU_oC_hi),
.oQ(RASH_out));
REG32 RASL(.iClk(Clk), .nRst(nRst), .iEn(iRAS_en), .iD(ALU_oC_lo),
.oQ(RASL_out));

// 32 bit ALU result selection
assign RAS_out = iMUX_RZHS ? RASH_out : RASL_out;
assign RZ_out  = iMUX_RZHS ? RZH_out : RZL_out;
// Select between storage or current registers
assign RZX_out = iMUX_ASS ? RAS_out : RZ_out;

// Memory
assign oMemAddr = iMUX_MAP ? PC_out : RZX_out ;
assign oMemData = RB_out;

// Write Back
// Select Memory input on WBM, Select PC for JAL, otherwise use ALU result
assign RWB_in = iMUX_WBM ? iMemData :
                iMUX_WBP ? PC_out   : RZX_out;

// Write back buffer register
REG32 RWB(.iClk(iClk), .nRst(pipe_rst), .iEn(iRWB_en), .iD(RWB_in),
.oQ(RF_iRegC));

endmodule
```

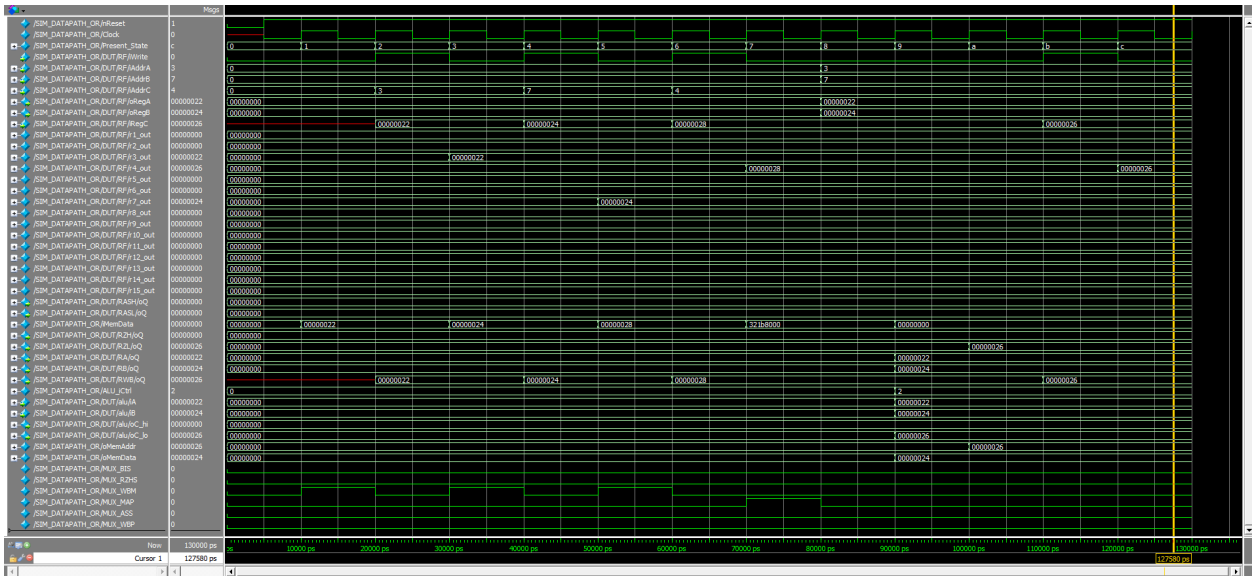
Data Path Simulation Results

Below are the results from the functional simulation of the datapath using the demo test benches as demonstrated in lab1.

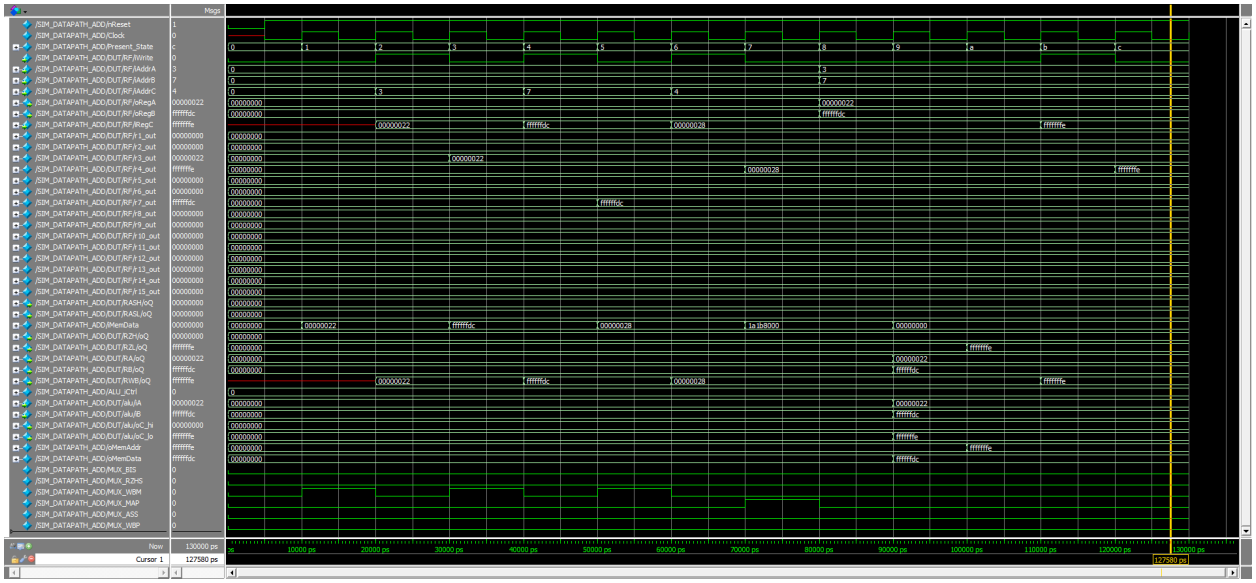
Demo 1: and R4, R3, R7



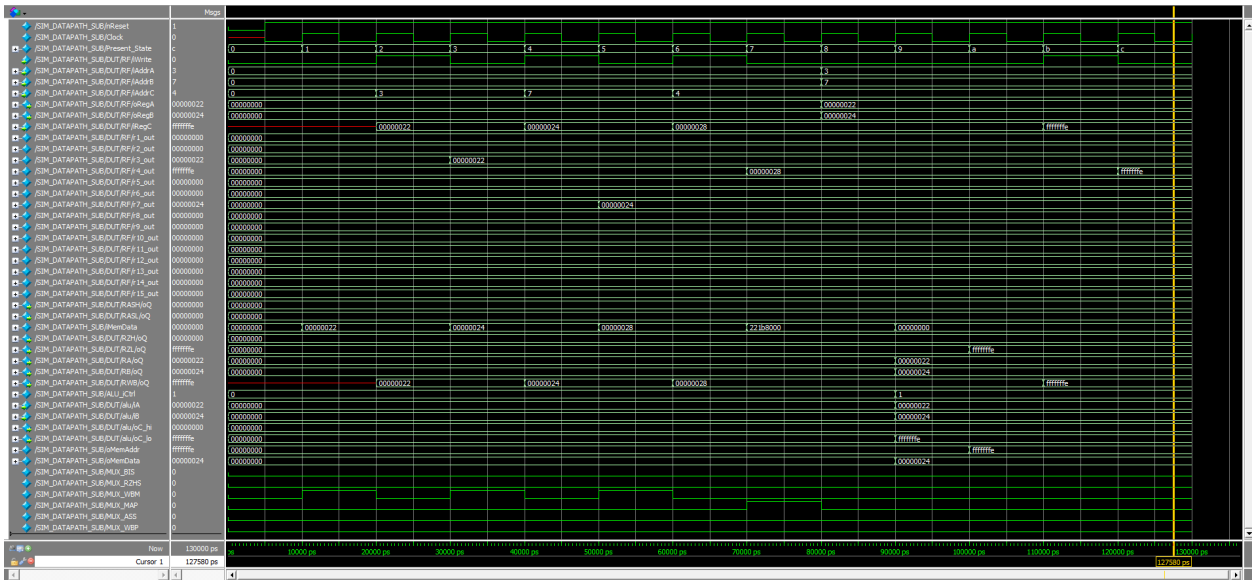
Demo 2: or R4, R3, R7



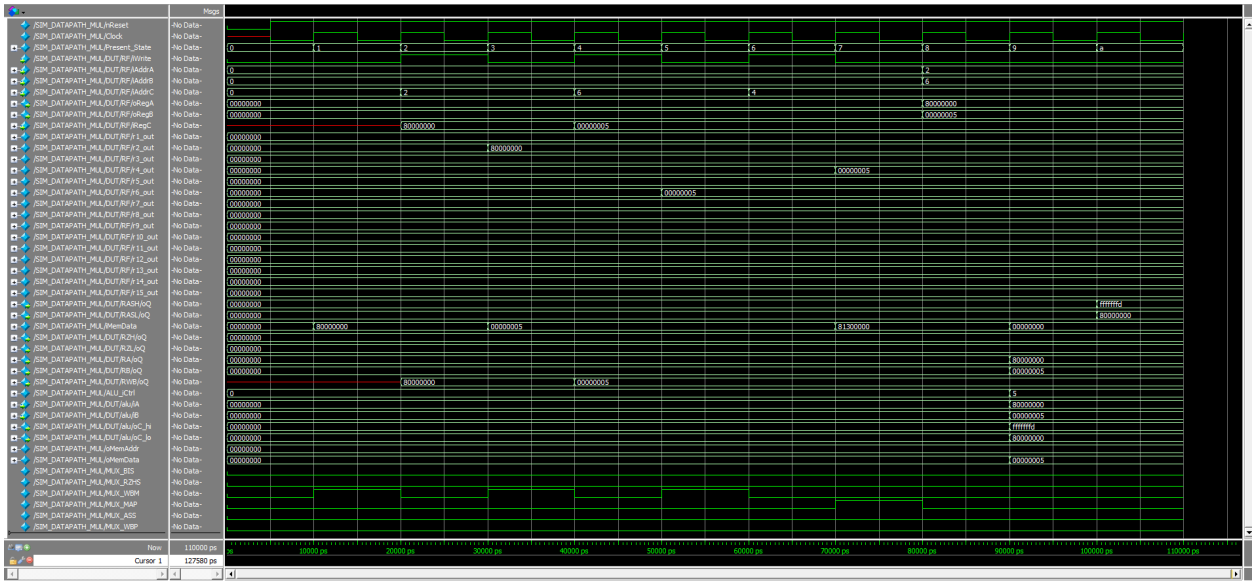
Demo 3: add R4, R3, R7



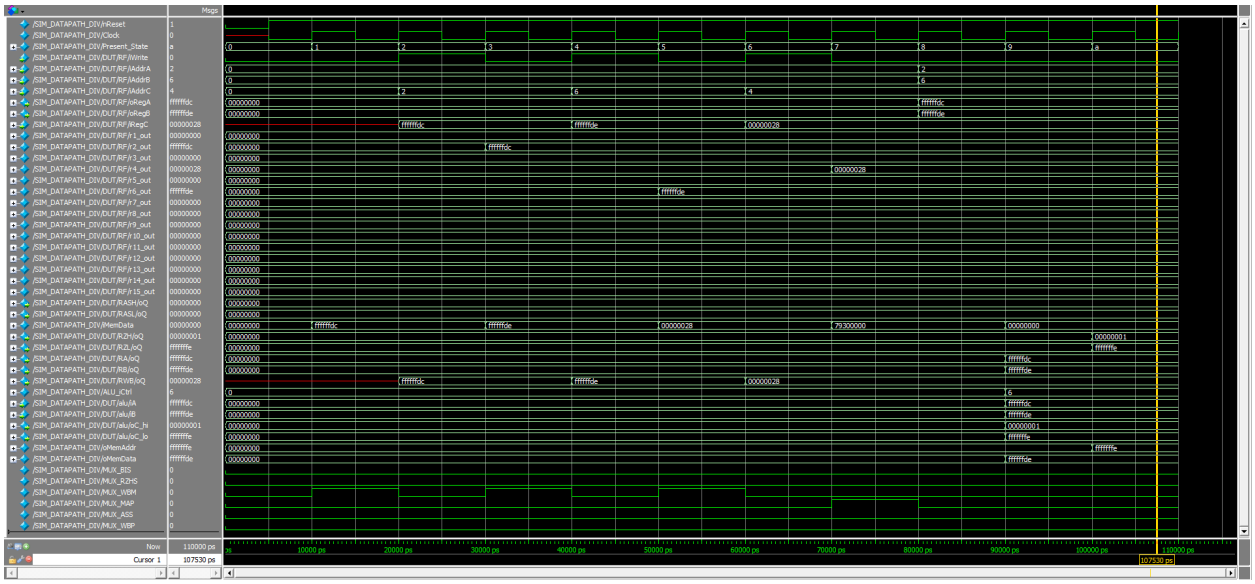
Demo 4: sub R4, R3, R7



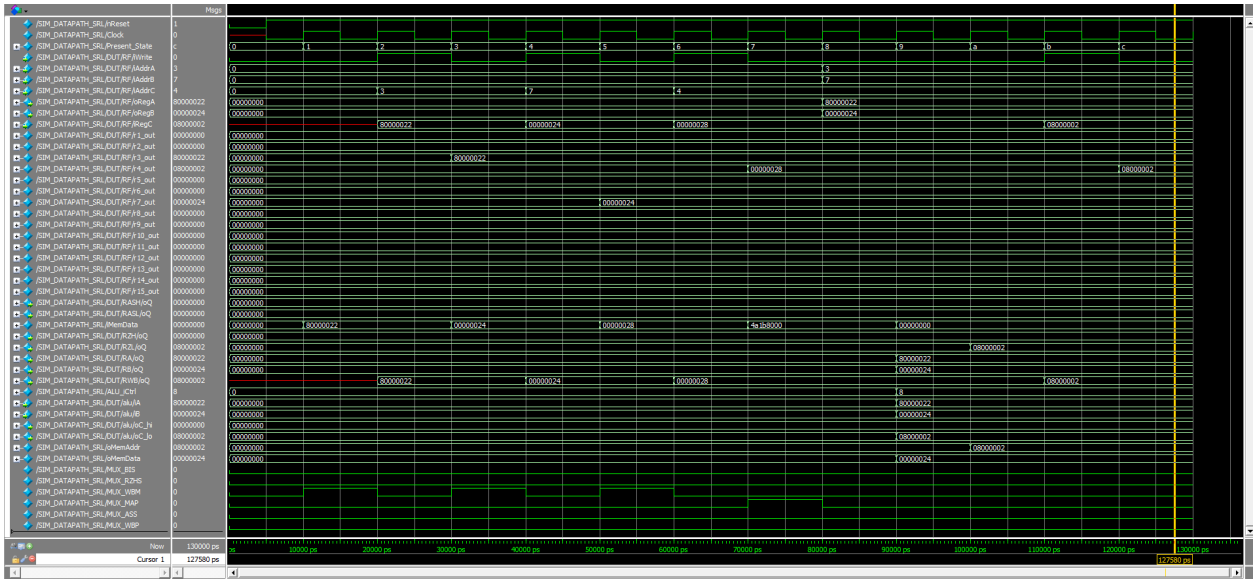
Demo 5: mul R2, R6



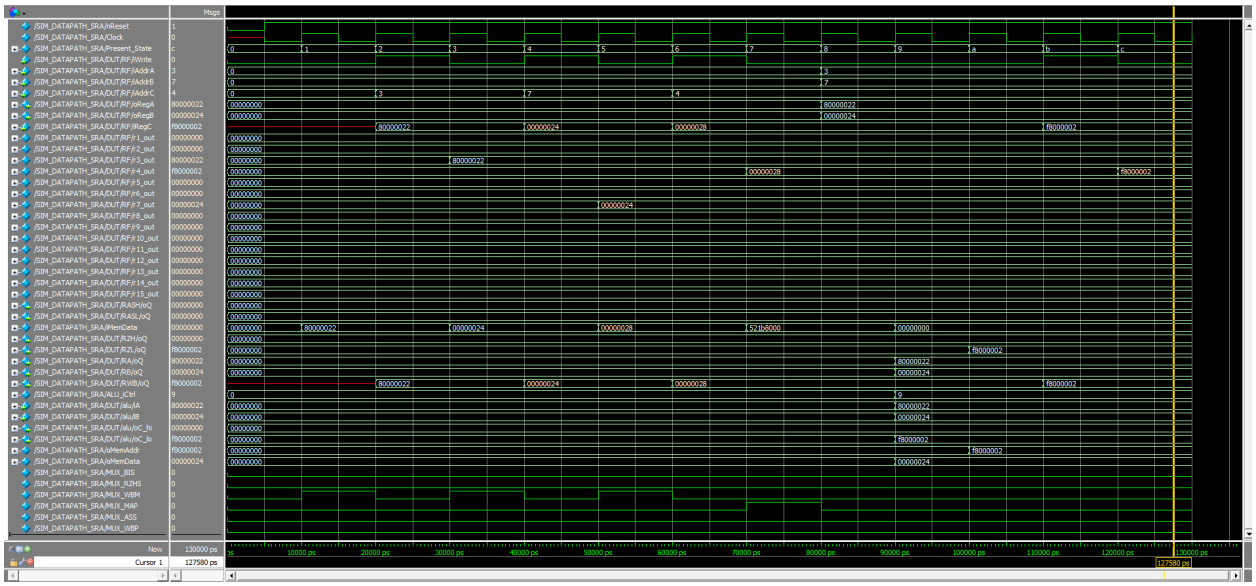
Demo 6: div R2, R6



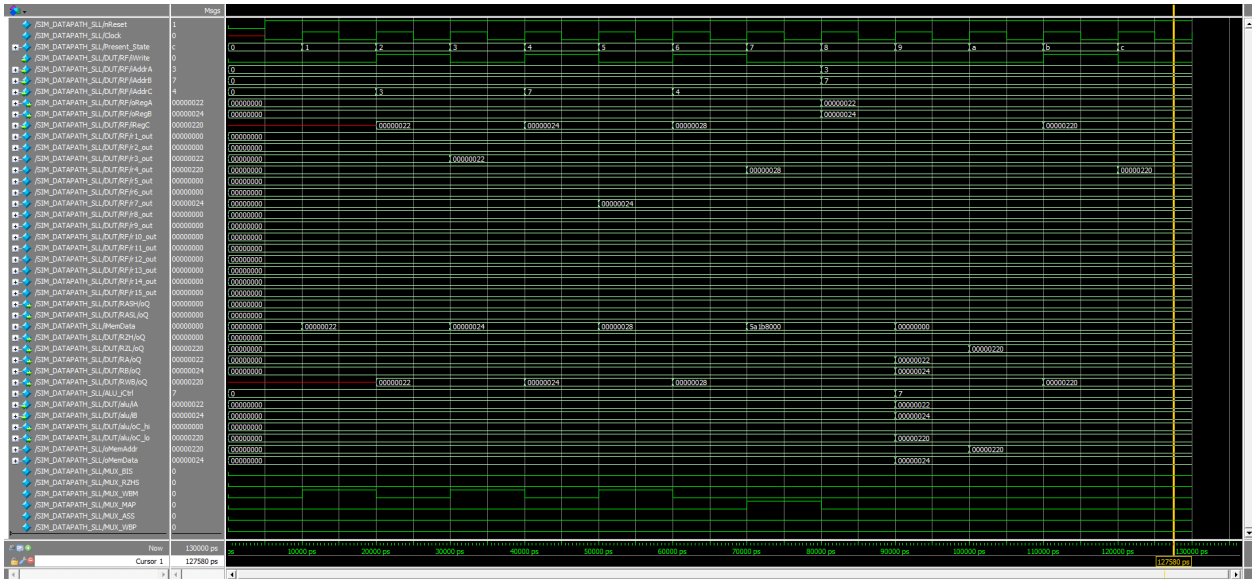
Demo 7: shr R4, R3, R7



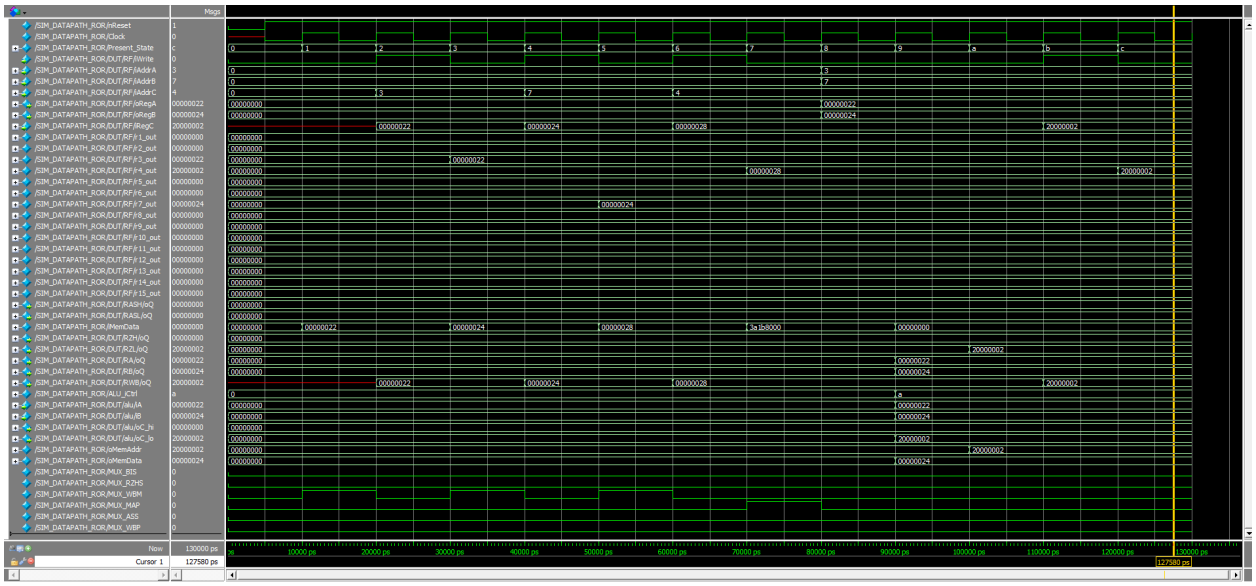
Demo 8: shra R4, R3, R



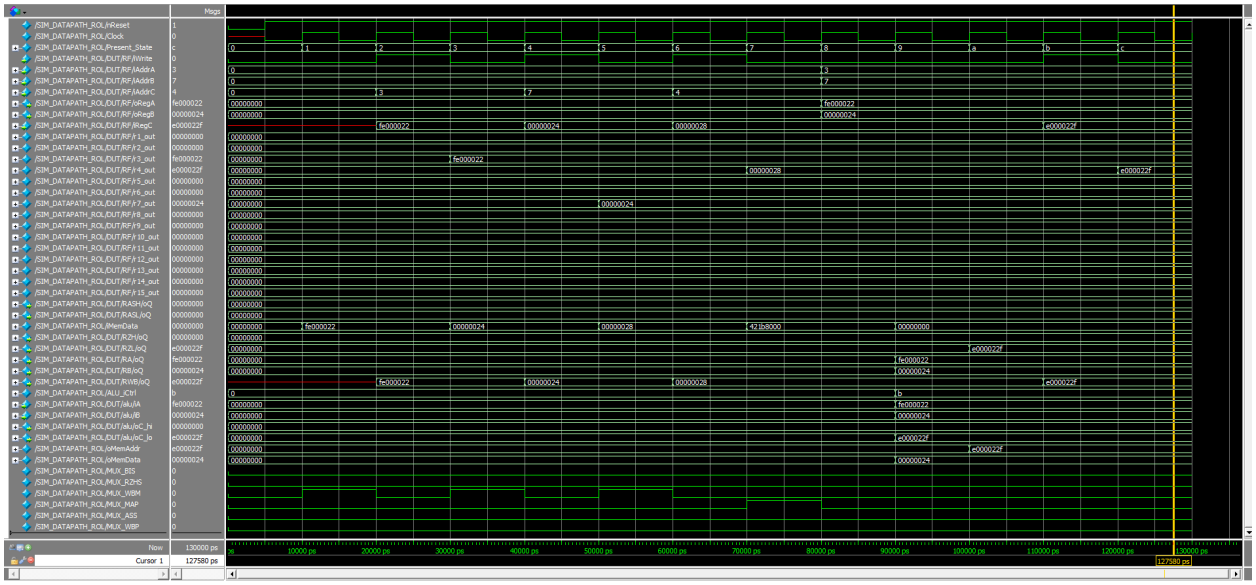
Demo 9: shl R4, R3, R7



Demo 10: ror R4, R3, R7

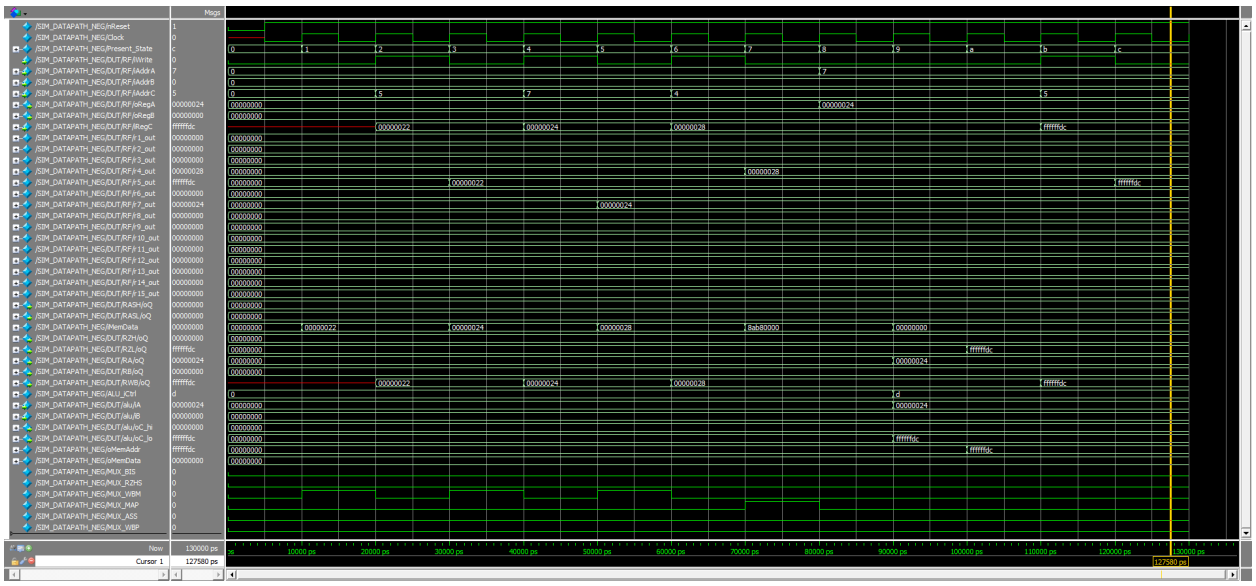


Demo 11: rol R4, R3, R7



Demo 12: neg R5, R7

This was done with neg R5, R7 instead of neg R5, R0 because our implementation of the register file has R0 wired as always 0 and it cannot take on alternate values.



Demo 13: not R5, R0