CMPE 374 | Digital Systems Engineering

Phase 2 Submission

April 5th, 2020

Daniyal Maniar | 20064993

Hermann Krohn | 20057435

Group: 7

Professor: Ahmad Afsahi, P.Eng

Teaching Assistant: Yiltan Temucin

Introduction

Our design follows the project specifications that were provided. Some improvements included:

- A completely modular design that could facilitate:
  - Any 2^N bit size design (16, 32, 64, 128, etc)
  - Any amount of registers (as long as the number of registers isn't above the bit threshold)
  - Any size of RAM
- A dual bus lines system for HI/LO registers to test higher bit count operations
  - Interchange system to switch values between bus lines

Code, Wave forms, & Memory Dumps

Register

```
// Module for single register
module register #(parameter BITS = 32)(
        input clk, input clear, input loadEnable,
        input [BITS-1:0] inputD,
        output reg [BITS-1:0] outputQ);

// To check whether clock or clear is selected
wire sen;
assign sen = clk || clear;

// At ever positive edge do this
always @ (posedge sen)
        // If clear else put value
        if (clear == 1)
                outputQ <= {BITS{1'b0}};
        else if (loadEnable == 1)
                outputQ <= inputD;
endmodule
```

Register File

```
// Register File

module registerFile #(parameter BITS = 32, REGISTERS = 16)(
                        input [BITS-1:0] busMuxOut,
                        input clk,
                        input [REGISTERS-1:0] clr, loadEnable,
                        output [BITS * REGISTERS - 1 : 0] registerStream
);
        generate
        genvar i;
                for (i = 0; i < REGISTERS; i = i + 1) begin: gen_registers
                        register #(.BITS(BITS)) generalRegister(clk, clr[i], loadEnable[i], busMuxOut, registerStream[(i+1) * BITS - 1 : i*BITS]);
                end
        endgenerate


endmodule
```

Register Multiplexer

```
// Module for register file

// Register stream ordering
// Indexes for registers shown below
/*
0 - r0
1 - r1
2 - r2
3 - r3
4 - r4
5 - r5
6 - r6
7 - r7
8 - r8
9 - r9
10 - r10
11 - r11
12 - r12
13 - r13
14 - r14
15 - r15
16 - pc
17 - rZ (LOW/HIGH)
18 - LO/HI
19 - MDR
20 - INPUT
21 - c_sign_extended
22 - INTERHI/INTERLO
*/

module registerSelect #(parameter BITS = 32, REGISTERS = 23)(
                        input [BITS * REGISTERS - 1 : 0] registerStream,
                        input [REGISTERS - 1 : 0] registerSelect,
                        input BAOut,
                        output [BITS - 1 : 0] busMuxOut
);

        generate
        genvar i;
        for (i = 0; i < REGISTERS; i = i + 1) begin : register_selector
                assign busMuxOut = registerSelect[i] ? ((BAOut && (i == 0)) ? {BITS{1'b0}} : registerStream[(i+1)*BITS-1: i*BITS]) : {BITS{1'bz}};
        end
        endgenerate

endmodule
```

ALU

```
// ALU

// ctrl_signal is a one-hot encoding
// Indexes for ctrl_signal shown below
/*
0 - add
1 - subtract
2 - multiply
3 - divide
4 - shift right
5 - shift left
6 - rotate right
7 - rotate left
8 - logical and
9 - logical or
10 - negate (two's compliment)
11 - Not (one's compliment)
12 - incPC
*/

module alu #(parameter BITS = 32, SIG_COUNT = 13)(
        input [SIG_COUNT-1:0] ctrl_signal,
        input [BITS-1:0] X, Y,
        output [(BITS*2)-1:0] operationResult
);

        wire signed [BITS - 1:0] add_result, sub_result;
        wire signed [(BITS*2) - 1:0] mul_result, div_result;
        wire[BITS - 1:0] shiftR_result, shiftL_result;
        wire[BITS - 1:0] rotateR_result, rotateL_result;
        wire[BITS - 1:0] and_result, or_result;
        wire[BITS - 1:0] negate_result, not_result;
        wire [BITS - 1:0] pc_result;

        aluResultSelector #(.BITS(BITS), .SIG_COUNT(SIG_COUNT), .OUT_BITS(BITS)) alu_out_select_LO(
                {pc_result, not_result, negate_result, or_result, and_result, rotateL_result, rotateR_result, shiftL_result, shiftR_result, div_result[BITS-1:0], mul_result[BITS-1:0], sub_result, add_result},
                ctrl_signal,
                operationResult[BITS-1:0]
        );
        aluResultSelector #(.BITS(BITS), .SIG_COUNT(SIG_COUNT), .OUT_BITS(BITS)) alu_out_select_HI(
                {{BITS{pc_result[BITS-1]}}, {BITS{not_result[BITS-1]}}, {BITS{negate_result[BITS-1]}}, {BITS{or_result[BITS-1]}}, {BITS{and_result[BITS-1]}}, {BITS{rotateL_result[BITS-1]}}, {BITS{rotateR_result[BITS-1]}}, {BITS{shiftL_result[BITS-1]}},
{BITS{shiftR_result[BITS-1]}}, div_result[(2*BITS)-1:BITS], mul_result[(2*BITS)-1:BITS], {BITS{sub_result[BITS-1]}}, {BITS{add_result[BITS-1]}}},
                ctrl_signal,
                operationResult[(BITS*2)-1:BITS]
        );


        carryLookAheadAdder #(.BITS(BITS)) cla_inst(X, Y, add_result);
        subtract #(.BITS(BITS)) sub_inst(X, Y, sub_result);
        multiply #(.BITS(BITS)) mul_inst(X, Y, mul_result);
        division #(.BITS(BITS)) div_inst(X, Y, div_result);
        shiftR #(.BITS(BITS)) shiftR_inst(X, Y, shiftR_result);
        shiftL #(.BITS(BITS)) shiftL_inst(X, Y, shiftL_result);
        rotateR #(.BITS(BITS)) rotateR_inst(X, Y, rotateR_result);
        rotateL #(.BITS(BITS)) rotateL_inst(X, Y, rotateL_result);
        assign and_result =  X && Y;
        assign or_result = X || Y;
        negate #(.BITS(BITS))  negate_inst(X, negate_result);
        notBits #(.BITS(BITS))  notBits_inst(X, not_result);
        carryLookAheadAdder #(.BITS(BITS)) PC_INC_ADDER ({BITS{1'b0}}+4, Y, pc_result);

endmodule
```

ALU Result Selector

```
// Operation result selector
// aluResultSelector

module aluResultSelector #(parameter BITS=32, SIG_COUNT=13, OUT_BITS=BITS)(
        input [(OUT_BITS*SIG_COUNT)-1:0] resultStream,
        input [SIG_COUNT-1:0] ctrl_signal,
        output [OUT_BITS-1:0] selectedResult
);

        genvar i;
        generate
                for(i = 0; i < SIG_COUNT; i = i + 1)begin : selectResult
                        assign selectedResult = ctrl_signal[i] ? resultStream[((i+1)*OUT_BITS)-1:i*OUT_BITS] : {OUT_BITS{1'bz}};
                end
        endgenerate

endmodule
```

CLA Adder

```
// Carry lookahead adder

module carryLookAheadAdder #(parameter BITS=32)(
        input [BITS-1:0] summand1_32_bits,
        input [BITS-1:0] summand2_32_bits,
        output [BITS-1:0] outputSum
);

        wire [BITS:0] w_C;
        wire [BITS-1:0] w_G, w_P, w_Sum;

        // Create the full adders
        genvar i;
        generate
                for(i=0; i<BITS; i=i+1) begin : genFullAdders
                        fulladder_cla full_adder_inst(
                                .summand1(summand1_32_bits[i]),
                                .summand2(summand2_32_bits[i]),
                                .inCarry(w_C[i]),
                                .sum(w_Sum[i])
                        );
                end
        endgenerate


        // Generate the Generate terms and Propagate terms and Carry Terms
        genvar j;
        generate
                for(j=0; j<BITS; j=j+1) begin : genTerms
                                assign w_G[j] = summand1_32_bits[j] & summand2_32_bits[j];
                                assign w_P[j] = summand1_32_bits[j] | summand2_32_bits[j];
                                assign w_C[j+1] = w_G[j] |(w_P[j] & w_C[j]);
                        end
        endgenerate

        assign w_C[0] = 1'b0; // no carry input to the first full adder

        //assign outputSum = {w_C[BITS], w_Sum}; // concatenation
        assign outputSum = w_Sum; // concatenation

endmodule
```

CLA Full Adder

```
// Fulladders for carry-lookahead adder

module fulladder_cla(
        input summand1,
        input summand2,
        input inCarry,
        output sum
);

        assign sum = ((summand1 ^ summand2) ^ inCarry);

endmodule
```

Subtract

```
// Subtract

module subtract #(parameter BITS=32)(
        input [BITS-1:0] X,
        input [BITS-1:0] Y,
        output [BITS-1:0] outputSub
);

        wire [BITS-1:0] negatedY;

        negate #(.BITS(BITS)) neg_inst(Y, negatedY);

        carryLookAheadAdder #(.BITS(BITS)) cla_inst(X, negatedY, outputSub);

endmodule
```

Negate

```
// Negate (two's compliment)

module negate #(parameter BITS=32)(
        input [BITS-1:0] in,
        output wire [BITS-1:0] out
);

        wire [BITS-1:0] notResult;
        notBits #(.BITS(BITS)) not_inst(in, notResult);

        carryLookAheadAdder #(.BITS(BITS)) cla_inst(notResult, 1, out);

endmodule
```

Not

```
// Not (one's compliment)

module notBits #(parameter BITS=32)(
        input [BITS-1:0] in,
        output wire [BITS-1:0] out
);

        assign out = (in ^ {BITS{1'b1}});

endmodule
```

Multiply

```
// Multiply.v

module multiply #(parameter BITS=32)(
        input [BITS-1:0] multiplicand,
        input [BITS-1:0] multiplier,
        output reg [(BITS*2)-1:0] outputMul
);


        wire [BITS-1:0] negMul;
        negate #(.BITS(BITS)) neg_inst(multiplicand, negMul);
        reg [2:0] bitGroupings [(BITS/2)-1:0];  // Array of 3 bit groupings
        reg signed [BITS:0] currentAddition;                            // current derived value from bit grouping
        reg signed [(BITS*2)-1:0] shiftedCurrentAddition;
        integer i, j;

        always @ (*)
        begin
                outputMul = 0;
                bitGroupings[0] = {multiplier[1], multiplier[0], 1'b0};

                for(i = 1;i<(BITS/2);i=i+1) begin
                        bitGroupings[i] = {multiplier[(2*i)+1],multiplier[2*i],multiplier[(2*i)-1]};
                end

                for(j=0;j<(BITS/2);j=j+1) begin
                        case(bitGroupings[j])
                                3'b001 , 3'b010 : currentAddition = {multiplicand[BITS-1], multiplicand};
                                3'b011 : currentAddition = {multiplicand, 1'b0};
                                3'b100 : currentAddition = {negMul, 1'b0};
                                3'b101 , 3'b110 : currentAddition = {negMul[BITS-1], negMul};
                                default : currentAddition = 0;
                        endcase
                        shiftedCurrentAddition = currentAddition << (2*j);
                        outputMul = (outputMul + shiftedCurrentAddition);
                end
        end
endmodule
```

Division

```
// division.v

module division #(parameter BITS=32)(
        input [BITS-1:0] dividend,
        input [BITS-1:0] divisor,
        output reg [(BITS*2)-1:0] result
);

        reg [BITS:0] a;
        reg [BITS-1:0] q;
        reg [BITS:0] m;
        reg negate_flag;

        integer i;

        always @ *
        begin
                a = 0;

          //  Make dividend and divisor are positive
          if (dividend[BITS - 1] == 1)
                        q = -dividend;
          else
                        q = dividend;

          if (divisor[BITS - 1] == 1)
                        m = {1'b0, -divisor};
          else
                        m = {1'b0, divisor};

          // Determine if result needs to be negated
          if ((dividend[BITS - 1] ^ divisor[BITS - 1]) == 1)
                        negate_flag = 1'b1;
          else
                        negate_flag = 1'b0;

                for(i = 0; i < BITS; i = i + 1) begin
                        a = a << 1;
                        a[0] = q[BITS-1];
                        q = q << 1;
                        if(a[BITS] == 1)
                                a = a + m;
                        else
                                a = a - m;
                        if(a[BITS] == 1)
                                q[0] = 1'b0;
                        else
                                q[0] = 1'b1;
                end
                if(a[BITS] == 1) begin
                        a = a + m;
                end

                if (negate_flag == 1)
                        q = -q;

                result[(2*BITS)-1:BITS] = a[BITS - 1: 0];       // Quotient
                result[BITS-1:0] = q;                                                   // Remainder

        end
endmodule
```

Shift Right

```
// Shift Right

module shiftR #(parameter BITS=32)(
        input [BITS-1:0] in,
        input [BITS-1:0] shiftAmount,
        output wire [BITS-1:0] out
);
        assign out = (in >> shiftAmount);
endmodule
```

Shift left

```
// Shift left

module shiftL #(parameter BITS=32)(
         input [BITS-1:0] in,
         input [BITS-1:0] shiftAmount,
         output wire [BITS-1:0] out
);
         assign out = (in << shiftAmount);
endmodule
```

Rotate Right

```
// Rotate Right

module rotateR #(parameter BITS=32)(
        input [BITS-1:0] in,
        input [BITS-1:0] rotateAmount,
        output wire [BITS-1:0] out
);

        assign out = ((in >> (rotateAmount % BITS))|(in << (BITS-(rotateAmount % BITS))));
endmodule
```

Rotate Left

```
// Rotate Left

module rotateL #(parameter BITS=32)(
        input [BITS-1:0] in,
        input [BITS-1:0] rotateAmount,
        output wire [BITS-1:0] out
);

        assign out = ((in << (rotateAmount % BITS))|(in >> (BITS-(rotateAmount % BITS))));
endmodule
```

Datapath

```
// Datapath

module datapath #(parameter BITS=32, REGISTERS=16, RAMSIZE=512, TOT_REGISTERS=REGISTERS+7, SIG_COUNT=13)(
        input reset, clk, rClk,
        input CONin, PCin, IRin, RYin, RZin, MARin, HILOin, OUTPUTin, INTERin, MDRin, Read, Write, INPUTout, MDRout, HILOout, RZout, PCout, Cout, INTERout,
        input BAout, Gra, Grb, Grc, Rout, Rin,
        input ADD, SUB, MUL, DIV, SHR, SHL, ROR, ROL, AND, OR, NEGATE, NOT, IncPC,
        input [BITS-1:0] INPUTUnit,
        output wire [(BITS*TOT_REGISTERS)-1:0] regSelectStreamLO, regSelectStreamHI,
        output wire [BITS-1:0] busLO, busHI,
        output wire [BITS-1:0] MARVal,
        output wire [(BITS*2)-1:0] RZVal,
        output wire [BITS-1:0] IRVal,
        output wire [BITS-1:0] LOVal, HIVal,
        output wire [BITS-1:0] OUTPUTUnit,
        output wire [BITS-1:0] c_sign_extended,
        output wire [BITS-1:0] MDRVal,
        output wire [BITS-1:0] INTERHIVal, INTERLOVal,
        output wire CON
);
        localparam ADDR = $clog2(RAMSIZE);

        wire [(BITS*REGISTERS)-1:0] genRegisterStream;
        wire [BITS-1:0] PCVal;
        wire [BITS-1:0] RYVal;
        wire [(BITS*2)-1:0] operationResult;
        wire [SIG_COUNT-1:0] alu_ctrl_signal;
        wire [BITS-1:0] INVal;
        wire [REGISTERS-1:0] GPRin, GPRout;
        wire [BITS-1:0] MDataIn;

        register #(.BITS(BITS)) PC(clk, reset, PCin, busLO, PCVal);
        register #(.BITS(BITS)) IR(clk, reset, IRin, busLO, IRVal);
        register #(.BITS(BITS)) RY(clk, reset, RYin, busLO, RYVal);
        register #(.BITS(BITS*2)) RZ(clk, reset, RZin, operationResult, RZVal);
        register #(.BITS(BITS)) MAR(clk, reset, MARin, busLO, MARVal);
        register #(.BITS(BITS)) HI(clk, reset, HILOin, busHI, HIVal);
        register #(.BITS(BITS)) LO(clk, reset, HILOin, busLO, LOVal);
        register #(.BITS(BITS)) OUTPUT(clk, reset, OUTPUTin, busLO, OUTPUTUnit);
        register #(.BITS(BITS)) INPUT(clk, reset, 1'b1, INPUTUnit, INVal);
        register #(.BITS(BITS)) INTERHI(clk, reset, INTERin, busLO, INTERHIVal);
        register #(.BITS(BITS)) INTERLO(clk, reset, INTERin, busHI, INTERLOVal);
        mdr #(.BITS(BITS)) MDR(busLO, MDataIn, Read, clk, reset, MDRin, MDRVal);

        ram #(.BITS(BITS), .RAMSIZE(RAMSIZE), .ADDR(ADDR))RAM(MDRVal, Read, Write, MARVal[ADDR-1:0], rClk, MDataIn); // Maybe a fan out warning but should be fine //Not sure if t

        //assign regSelectStream = {INVal, MDRVal, LOVal, HIVal, RZVal[(BITS*2)-1:BITS], RZVal[BITS-1:0], PCVal, genRegisterStream};
        assign regSelectStreamLO = {INTERLOVal, c_sign_extended, INVal, MDRVal, LOVal, RZVal[BITS-1:0], PCVal, genRegisterStream};
        assign regSelectStreamHI = {INTERHIVal, {BITS{1'bz}}, {BITS{1'bz}}, {BITS{1'bz}}, HIVal, RZVal[(2*BITS)-1:BITS], {BITS{1'bz}}, {(BITS*REGISTERS){1'bz}}};
        assign alu_ctrl_signal = {IncPC, NOT, NEGATE, OR, AND, ROL, ROR, SHL, SHR, DIV, MUL, SUB, ADD};

        selectAndEncode #(.BITS(BITS), .REGISTERS(REGISTERS)) selectAndEncode_inst(IRVal, Gra, Grb, Grc, Rin, Rout, BAout, GPRin, GPRout, c_sign_extended);
        registerFile #(.BITS(BITS), .REGISTERS(REGISTERS)) genPurposeRegs(busLO, clk, {REGISTERS{reset}}, GPRin, genRegisterStream);
        registerSelect #(.BITS(BITS), .REGISTERS(TOT_REGISTERS)) regSelectLO(regSelectStreamLO, {INTERout, Cout, INPUTout, MDRout, HILOout, RZout, PCout, GPRout}, BAout, busLO);
        registerSelect #(.BITS(BITS), .REGISTERS(TOT_REGISTERS)) regSelectHI(regSelectStreamHI, {INTERout, Cout, INPUTout, MDRout, HILOout, RZout, PCout, GPRout}, BAout, busHI);
        alu #(.BITS(BITS), .SIG_COUNT(SIG_COUNT)) alu_inst(alu_ctrl_signal, RYVal, busLO, operationResult);
        con_ff #(.BITS(BITS)) con_ff_inst(busLO, IRVal[20:19], CONin, CON);

endmodule
```

MDR

```
// MDR (Memory Data Register) with MUX to select between input from the bus or the mem chip

module mdr #(parameter BITS=32)(
        input [BITS-1:0] busMuxOut, MDataIn,
        input read, clk, clear, enable,
        output [BITS-1:0] MDRout
);

        reg [BITS-1:0] muxOut;

        always @ (busMuxOut, MDataIn, read) begin
                if(read == 1)
                        muxOut <= MDataIn;
                else
                        muxOut <= busMuxOut;

        end

        register #(.BITS(BITS)) MDR_reg(clk, clear, enable, muxOut, MDRout);

endmodule
```

CON_FF

```
// con_ff.v

module con_ff  #(parameter BITS = 32)(
    input [BITS-1:0] bus,
    input [1:0] IR_C2,
    input CON_enable,
    output reg Q
);

        reg con_d;

        initial begin
                Q <= 0;
        end

   always @ (bus, IR_C2) begin
                case(IR_C2)
                        2'b00: con_d <= (bus == 32'h00000000)? 1'b1 : 1'b0;
                        2'b01: con_d <= (bus != 32'h00000000)? 1'b1 : 1'b0;
                        2'b10: con_d <= (bus[BITS-1] == 1'b0)? 1'b1 : 1'b0;
                        2'b11: con_d <= (bus[BITS-1] == 1'b1)? 1'b1 : 1'b0;
                        default: con_d <= 0;
                endcase
        end

        always @(posedge CON_enable) begin
                Q <= con_d;
        end
endmodule
```

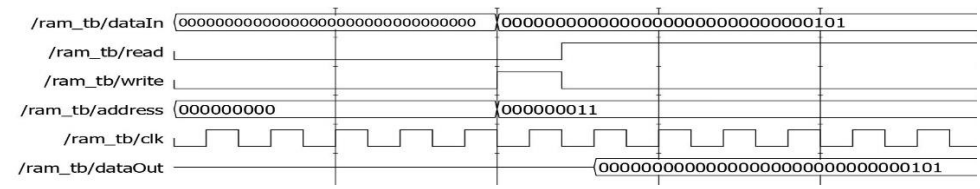
RAM

```
// RAM

module ram #(parameter BITS=32, RAMSIZE=512, ADDR=$clog2(RAMSIZE))(
        input [BITS-1:0] dataIn,
        input read,
        input write,
        input [ADDR-1:0] address,
        input clk,
        output reg [BITS-1:0] dataOut
);

        reg [BITS-1:0] RAM [0:RAMSIZE-1];


        initial begin
                // case 1: load
                //RAM[0] <= 'b00000_0000_0000_00000000000000000;
                //RAM[4] <= 'b00000_0001_0000_00000000000001010101;
                //RAM[85] <= 'h0000f7f7;

                // case 2: load (DELETE THIS?)
                // RAM[0] <= 'b00000_0001_0000_00000000000000000101;
                // RAM[4] <= 'b00000_0000_0001_00000000000000100011;
                // RAM[40] <= '0000hf7f7

                // case 3: load imm
                //RAM[0] <= 'b00000_0001_0000_00000000000000000000;
                //RAM[4] <= 'b00001_0001_0000_00000000000001010101;

                // case 4: load imm
                // RAM[0] <= 'b00000_0001_0000_00000000000000000101;
                // RAM[4] <= 'b00001_0000_0001_00000000000000100011;

                // Store Case 1
                //RAM[0] <= 'b00000_0001_0000_00000000000001010101;
                //RAM[4] <= 'b00010_0001_0000_00000000000001011010;

                // Store Case 2
                //RAM[0] <= 'b00000_0001_0000_00000000000001010101;
                //RAM[4] <= 'b00010_0001_0001_00000000000001011010;

                // addi case 1
                //RAM[0] <= 'b00000_0001_0000_00000000000000001010;
                //RAM[4] <= 'b01011_0010_0001_11111111111111111011;

                // andi case 2
                //RAM[0] <= 'b00000_0001_0000_00000000000000000000;
                //RAM[4] <= 'b01011_0010_0001_00000000000000011010;

                // ori case 2
                //RAM[0] <= 'b00000_0001_0000_00000000000000000000;
                //RAM[4] <= 'b01011_0010_0001_00000000000000011010;

                // BRANCH zr
                //RAM[0] <= 'b00000_0010_0000_00000000000000000000;
                //RAM[4] <= 'b10010_0010_0000_00000000000000100011;

                // BRANCH nz
                //RAM[0] <= 'b00000_0010_0000_00000000000000000001;
                //RAM[4] <= 'b10010_0010_0001_00000000000000100011;

                // BRANCH pl
                //RAM[0] <= 'b00000_0010_0000_00000000000000000001;
                //RAM[4] <= 'b10010_0010_0010_00000000000000100011;

                //BRANCH brmi
                //RAM[0] <= 'b00000_0010_0000_10000000000000000001;
                //RAM[4] <= 'b10010_0010_0011_00000000000000100011;

                // mfhi/mflo R2
                //RAM[0] <= 'b00000_0010_0000_00000000000000000111;
                //RAM[4] <= 'b00000_0010_0000_00000000000000000000;

                // Jump
                //RAM[0] <= 'b00000_0001_0000_00000000000000001100;
                //RAM[4] <= 'b10011_0001_0000_00000000000000000000;

                // jump jal
                //RAM[0] <= 'b00000_0001_0000_00000000000000001100;
                //RAM[4] <= 'b10100_0001_1111_00000000000000000001;

                // Input
                //RAM[0] <= 'b10101_0001_0000_00000000000000000000;

                // Output
                RAM[0] <= 'b00000_0001_0000_00000000000000001111;
                RAM[4] <= 'b10110_0001_0000_00000000000000000000;
        end

        always @ (posedge clk) begin
                if(write == 1 && read == 0) begin
                        RAM[address] = dataIn;
                end
                if (read == 1 && write == 0) begin
                        dataOut = RAM[address];
                end
        end

endmodule
```

RAM Testbench

```
// RAM tb

`timescale 1ns/10ps

module ram_tb;
        parameter BITS=32, RAMSIZE=512, ADDR=$clog2(RAMSIZE);


        reg [BITS-1:0] dataIn;
        reg read;
        reg write;
        reg [ADDR-1:0] address;
        reg clk;
        wire [BITS-1:0] dataOut;

        ram2 #(.BITS(BITS), .ADDR(ADDR), .RAMSIZE(RAMSIZE)) ram_inst(dataIn, read, write, address, clk, dataOut);

        initial
    begin
        clk = 0;
        forever #10 clk = ~clk;
    end

         initial begin
                // Initialize values to default
                dataIn <= 'h0;
                address <= 'h0;
                write <= 0;
                read <= 0;
                #100;
                dataIn <= 'h5;
                address <= 'h3;
                write <= 1;
                read <= 0;
                #20;
                dataIn <= 'h5;
                write <= 0;
                read <= 1;
         end

endmodule
```

RAM Waveform



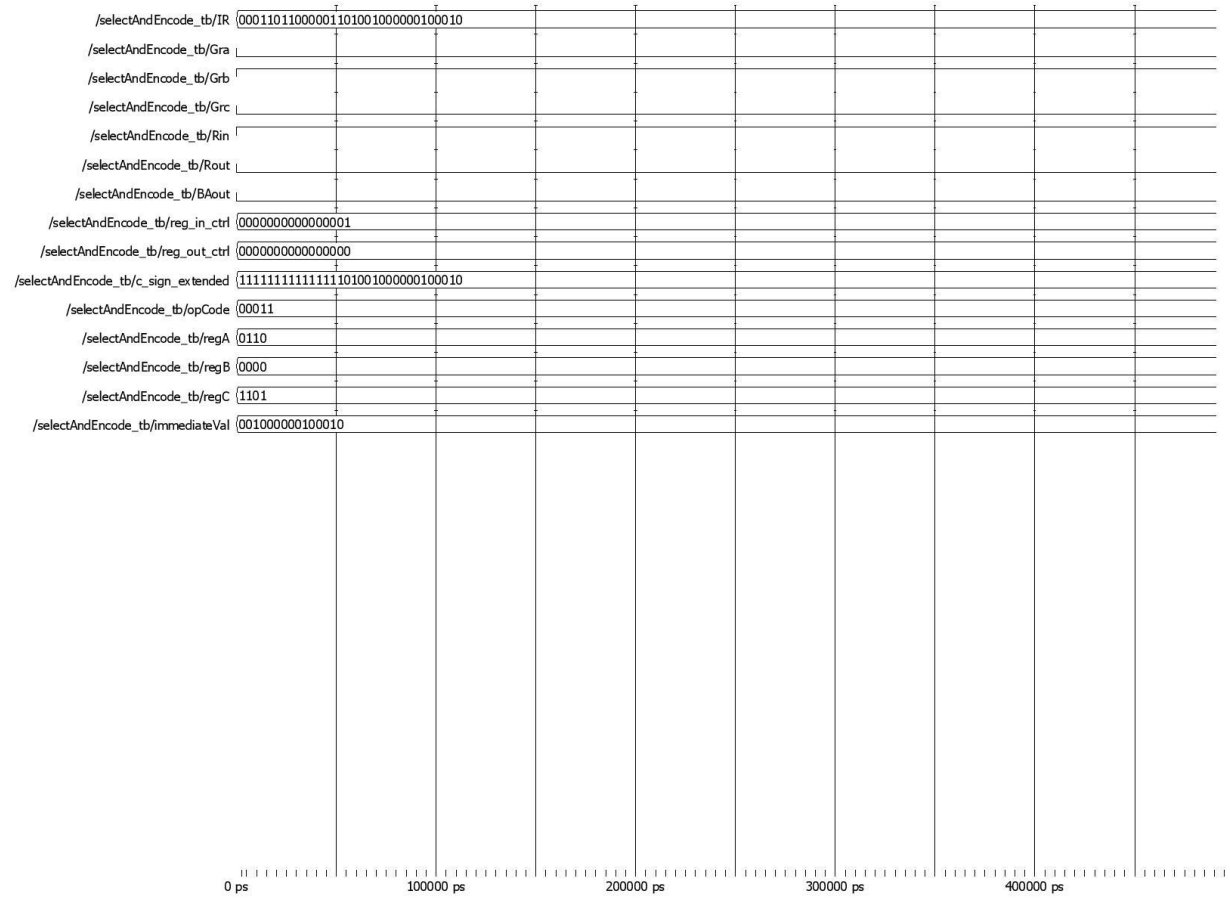
Select and Encode

```
// Select and Encode

module selectAndEncode #(parameter BITS = 32, REGISTERS = 16, REGISTER_BITS = $clog2(REGISTERS))(
    input [BITS-1:0] IR,
    input Gra, Grb, Grc, Rin, Rout, BAout,
    output [REGISTERS-1:0] reg_in_ctrl, reg_out_ctrl,
    output [BITS-1:0] c_sign_extended
);

    // Generate decoder input and c_sign_extended
         localparam regA = BITS-5-1;
         localparam regB = regA-REGISTER_BITS;
         localparam regC = regB-REGISTER_BITS;
         localparam Unused = regC-REGISTER_BITS;

         // Realized the only time this is used is when the instruction is in
         // I-Format, so we always assume Immediate bits end where regC is suppose
         // to end. Do not read the sign extended output unless in I Format
    //assign c_sign_extended = {{(BITS-Unused){IR[Unused]}}, IR[Unused:0]};
         assign c_sign_extended = {{(BITS-1-regC){IR[regC]}}, IR[regC:0]};

    wire [REGISTER_BITS-1:0] decoder_in;
         assign decoder_in = (IR[regA:regB+1]&{REGISTER_BITS{Gra}}) | (IR[regB:regC+1]&{REGISTER_BITS{Grb}}) | (IR[regC:Unused+1]&{REGISTER_BITS{Grc}});

         genvar i;
    generate
        for(i = 0; i < REGISTERS; i = i + 1) begin : decode
                                assign reg_in_ctrl[i] = (decoder_in == i) & Rin;
            assign reg_out_ctrl[i] = (decoder_in == i) & (Rout|BAout);
        end
    endgenerate


endmodule
```

Select and Encode Testbench

```
// selectAndEncode tb

`timescale 1ns/10ps

module selectAndEncode_tb;
        parameter BITS = 32;
        parameter REGISTERS = 16;
        parameter REGISTER_BITS = $clog2(REGISTERS);
        localparam immediateValLen = (BITS-5-(REGISTER_BITS*3));

        wire [BITS-1:0] IR;
        reg Gra, Grb, Grc, Rin, Rout, BAout;
        wire [REGISTERS-1:0] reg_in_ctrl, reg_out_ctrl;
        wire [BITS-1:0] c_sign_extended;

        reg [4:0] opCode;
        reg [REGISTER_BITS-1:0] regA, regB, regC;
        reg [immediateValLen-1:0] immediateVal;
        assign IR = {opCode, regA, regB, regC, immediateVal};

        selectAndEncode #(.BITS(BITS), .REGISTERS(REGISTERS), .REGISTER_BITS(REGISTER_BITS)) sAndE_inst(IR, Gra, Grb, Grc, Rin, Rout, BAout, reg_in_ctrl, reg_out_ctrl, c_sign_extended);

        initial begin
                // R format
                opCode <= 5'b00011;
                regA <= ({REGISTER_BITS{1'b0}} + 6);
                regB <= ({REGISTER_BITS{1'b0}} + 0);
                regC <= ({REGISTER_BITS{1'b0}} + 13);
                immediateVal <= ({immediateValLen{1'b0}}+4130);
                Gra <= 0;
                Grb <= 1;
                Grc <= 0;
                Rin <= 1;
                Rout <= 0;
                BAout <= 0;
        end
endmodule
```

Select and Encode Waveform



LD Testbench

```
// Load tb

`timescale 1ns/10ps
module load_tb;

	parameter BITS=32, REGISTERS=16, TOT_REGISTERS=REGISTERS+7, RAMSIZE=512;

	reg INPUTout, MDRout, HILOout, RZout, PCout, Cout, INTERout, BAout, Gra, Grb, Grc, Rout, Rin; // add any other signals to see in your simulation
	reg CONin, PCin, IRin, RYin, RZin, MARin, HILOin, OUTPUTin, INTERin, MDRin;
	reg Read, Write, ADD, SUB, MUL, DIV, SHR, SHL, ROR, ROL, AND, OR, NEGATE, NOT, IncPC;
	reg clk, rClk;
	reg [BITS-1:0] INPUTUnit;
	reg reset;
	wire [(BITS*TOT_REGISTERS)-1:0] regSelectStreamLO, regSelectStreamHI;
	wire [BITS-1:0] busLO, busHI;
	wire [BITS-1:0] MARVal;
	wire [(BITS*2)-1:0] RZVal;
	wire [BITS-1:0] IRVal;
	wire [BITS-1:0] OUTPUTUnit;
	wire [BITS-1:0] c_sign_extended;
	wire [BITS-1:0] MDRVal;
	wire [BITS-1:0] INTERHIVal, INTERLOVal;
	wire CON;
	wire [BITS-1:0] R0Val, R1Val, LOVal, HIVal;
	assign R0Val = regSelectStreamLO[(1*BITS)-1:BITS*0];
	assign R1Val = regSelectStreamLO[(2*BITS)-1:BITS*1];

	parameter Default = 4'b0000, T0 = 4'b0001, T1 = 4'b0010, T2 = 4'b0011,
			T3 = 4'b0100, T4 = 4'b0101, T5 = 4'b0110, T6 = 4'b0111,
			T7 = 4'b1000, T8 = 4'b1001, T9 = 4'b1010;

	reg [3:0] Present_state = Default;
	reg preload_reg = 1'b0;

	datapath #(.BITS(BITS), .REGISTERS(REGISTERS), .RAMSIZE(RAMSIZE)) DUT(
			reset, clk, rClk,
			CONin, PCin, IRin, RYin, RZin, MARin, HILOin, OUTPUTin, INTERin, MDRin, Read, Write, INPUTout, MDRout, HILOout, RZout, PCout, Cout, INTERout,
			BAout, Gra, Grb, Grc, Rout, Rin,
			ADD, SUB, MUL, DIV, SHR, SHL, ROR, ROL, AND, OR, NEGATE, NOT, IncPC,
			INPUTUnit,
			regSelectStreamLO, regSelectStreamHI,
			busLO, busHI,
			MARVal,
			RZVal,
			IRVal,
			LOVal, HIVal,
			OUTPUTUnit,
			c_sign_extended,
			MDRVal,
			INTERHIVal, INTERLOVal,
			CON);


	initial begin
		clk = 0;
		forever #10 clk = ~clk;
	end

	 initial begin
			rClk = 1;
			forever #5 rClk = ~rClk;
	 end

	always @(posedge clk) // finite state machine; if clock rising-edge
	begin
		case (Present_state)
			Default : Present_state = T0;
			T0 : Present_state = T1;
			T1 : Present_state = T2;
			T2 : begin
				if(preload_reg == 1)
					Present_state = T5;
				else
					Present_state = T3;
			end
			T3 : Present_state = T4;
			T4 : Present_state = T1;
				T5 : Present_state = T6;
				T6 : Present_state = T7;
				T7 : Present_state = T8;
				T8 : Present_state = T9;
		endcase
	end

	always @(Present_state) // do the required job in each state
	begin
		case (Present_state) // assert the required signals in each clock cycle
			Default: begin
				reset <= 1;
				CONin <= 0; PCin <= 0; IRin <= 0; RYin <= 0; RZin <= 0; MARin <= 0; HILOin <= 0; MDRin <= 0; OUTPUTin <= 0; INTERin <= 0;
				Read <= 0; Write <= 0;
				INPUTout <= 0; MDRout <= 0; HILOout <= 0; RZout <= 0; PCout <= 0; Cout <= 0; INTERout <= 0;
				BAout <= 0; Gra <= 0; Grb <= 0; Grc <= 0; Rout <= 0; Rin <= 0;
				ADD <= 0; SUB <= 0; MUL <= 0; DIV <= 0; SHR <= 0; SHL <= 0; ROR <= 0; ROL <= 0; AND <= 0; OR <= 0; NEGATE <= 0; NOT <= 0; IncPC <= 0;
				INPUTUnit <= {BITS{1'b0}};
				#5 reset <= 0;
			end
			T0: begin
				MARin <= 1; IncPC <= 1; RZin <= 1; PCout <= 1;
			end
			T1: begin
				Read <= 1;
				PCin <= 1; MDRin <= 1;
				#5 RZout <= 1; PCout <= 0; MARin <= 0; IncPC <= 0; RZin <= 0;
			end
			T2: begin
				IRin <= 1;
				#5 PCin <= 0; RZout <= 0; MDRout <= 1; MDRin <= 0; Read <= 0;
			end
			T3: begin
				Gra <= 1; Rin <= 1;
				#5 MDRout <= 0; IRin <= 0;Cout <= 1;
			end
			T4: begin
				MARin <= 1; IncPC <= 1; RZin <= 1;
				#5 Gra <= 0; Rin <= 0; Cout <= 0; PCout <= 1; preload_reg <= 1;
			end
			T5: begin
				RYin <= 1; Grb <= 1;
				#5 IRin <= 0; MDRout <= 0; BAout <= 1;
			end
			T6: begin
				ADD <= 1; RZin <= 1;
				#5 RYin <= 0; Grb <= 0; BAout <= 0; Cout <= 1;
			end
			T7: begin
				MARin <= 1;
				#5 ADD <= 0; RZin <= 0; RZout <= 1; Cout <= 0;
			end
			T8: begin
				Read <= 1; MDRin <= 1;
				#5 MARin <= 0; RZout <= 0;
			end
			T9: begin
				Gra <= 1; Rin <= 1;
				#5 MDRin <= 0; MDRout <= 1; Read <= 0;
			end
		endcase
	end
endmodule
```

LD Waveform

LD Case 1

Memory dump before/after LD Case 1

```
// memory data file (do not edit the following line - required for mem load use)
// instance=/load_tb/DUT/RAM/RAM
// format=mti addressradix=h dataradix=h version=1.0 wordsperline=8
  0: 00000000 xxxxxxxx xxxxxxxx xxxxxxxx 00800055 xxxxxxxx xxxxxxxx xxxxxxxx
  8: xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx
*
 50: xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx 0000f7f7 xxxxxxxx xxxxxxxx
 58: xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx
*
1f8: xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx
```

LD Case 2


/load_tb/CONin
/load_tb/PCin
/load_tb/IRin
/load_tb/RYin
/load_tb/RZin
/load_tb/MARin
/load_tb/HILOin
/load_tb/OUTPUTin
/load_tb/INTERin
/load_tb/MDRin
/load_tb/Read
/load_tb/Write
/load_tb/ADD
/load_tb/SUB
/load_tb/MUL
/load_tb/DIV
/load_tb/SHR
/load_tb/SHL
/load_tb/ROR
/load_tb/ROL
/load_tb/AND
/load_tb/OR
/load_tb/NEGATE
/load_tb/NOT
/load_tb/IncPC
/load_tb/clk
/load_tb/rClk
/load_tb/INPUTUnit
/load_tb/reset
/load_tb/regSelectStreamLO
/load_tb/regSelectStreamHI
/load_tb/busLO
/load_tb/busHI
/load_tb/MARVal
/load_tb/RZVal
/load_tb/IRVal
/load_tb/OUTPUTUnit
/load_tb/c_sign_extended
/load_tb/MDRVal
/load_tb/INTERHIVal
/load_tb/INTERLOVal
/load_tb/CON
/load_tb/R0Val
/load_tb/R1Val
/load_tb/LOVal
/load_tb/HIVal
/load_tb/Present_state
/load_tb/preload_reg
00000000000000000000000000000000
00000000
0000f7f7
00000000
00000004
00000028
0000000000000004
0000000000000008
0000000000000028
00800005
00080023
00000005
00000023
0000f7f7
0001
0010
0011
0100
0101
0010
0011
0110
0111
1000
1001
1010
0 ps
100000 ps
200000 ps
300000 ps
400000 ps
500000 ps

Memory dump before/after LD Case 2

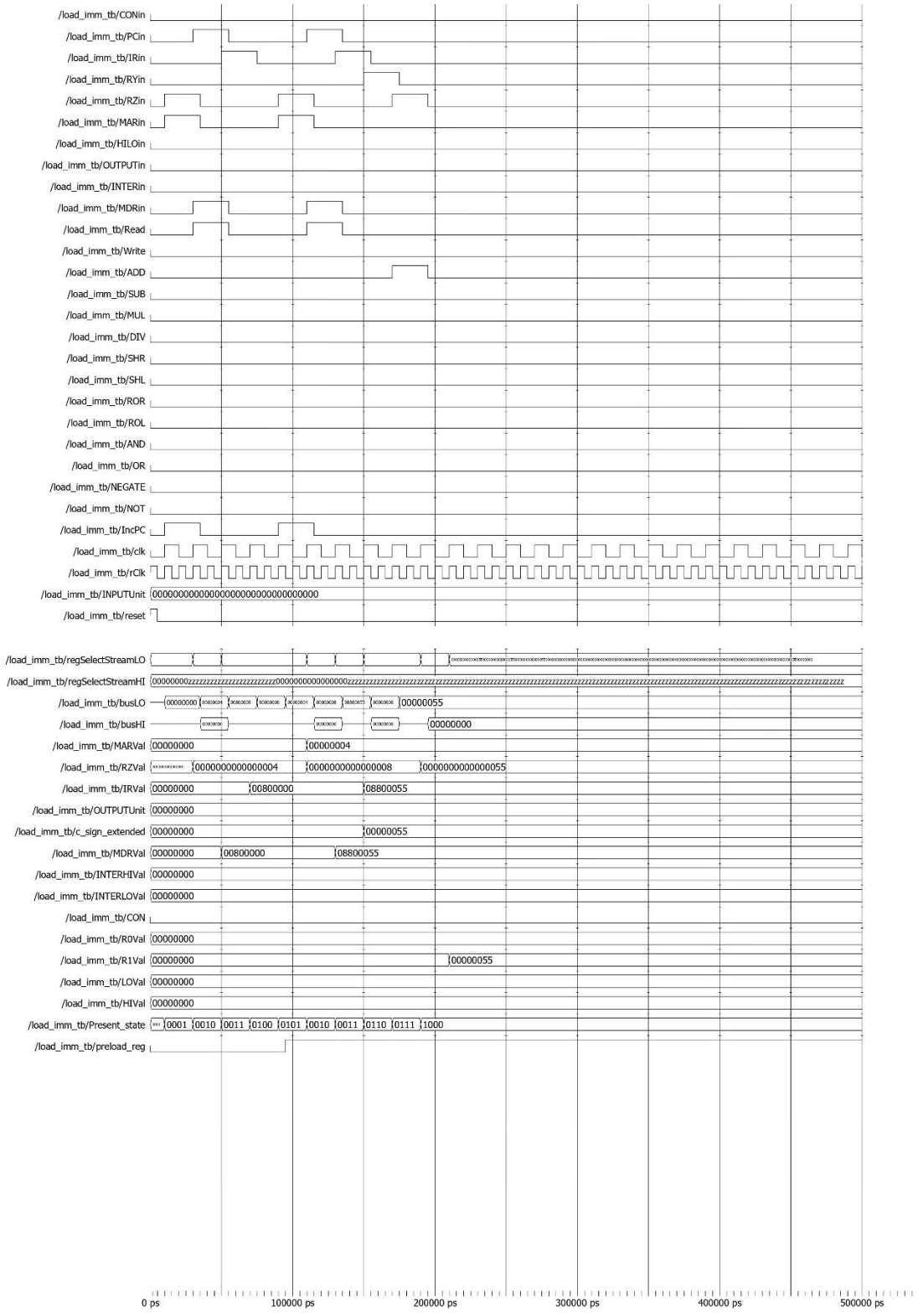
```
// memory data file (do not edit the following line - required for mem load use)
// instance=/load_tb/DUT/RAM/RAM
// format=mti addressradix=h dataradix=h version=1.0 wordsperline=8
  0: 00800005 xxxxxxxx xxxxxxxx xxxxxxxx 00080023 xxxxxxxx xxxxxxxx xxxxxxxx
  8: xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx
*
 28: 0000f7f7 xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx
 30: xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx
*
1f8: xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx
```

LDI Testbench

```
// Load_imm tb


`timescale 1ns/10ps
module load_imm_tb;

        parameter BITS=32, REGISTERS=16, TOT_REGISTERS=REGISTERS+7, RAMSIZE=512;

        reg INPUTout, MDRout, HILOout, RZout, PCout, Cout, INTERout, BAout, Gra, Grb, Grc, Rout, Rin; // add any other signals to see in your simulation
        reg CONin, PCin, IRin, RYin, RZin, MARin, HILOin, OUTPUTin, INTERin, MDRin;
        reg Read, Write, ADD, SUB, MUL, DIV, SHR, SHL, ROR, ROL, AND, OR, NEGATE, NOT, IncPC;
        reg clk, rClk;
        reg [BITS-1:0] INPUTUnit;
        reg reset;
        wire [(BITS*TOT_REGISTERS)-1:0] regSelectStreamLO, regSelectStreamHI;
        wire [BITS-1:0] busLO, busHI;
        wire [BITS-1:0] MARVal;
        wire [(BITS*2)-1:0] RZVal;
        wire [BITS-1:0] IRVal;
        wire [BITS-1:0] OUTPUTUnit;
        wire [BITS-1:0] c_sign_extended;
        wire [BITS-1:0] MDRVal;
        wire [BITS-1:0] INTERHIVal, INTERLOVal;
        wire CON;
        wire [BITS-1:0] R0Val, R1Val, LOVal, HIVal;
        assign R0Val = regSelectStreamLO[(1*BITS)-1:BITS*0];
        assign R1Val = regSelectStreamLO[(2*BITS)-1:BITS*1];

        parameter Default = 4'b0000, T0 = 4'b0001, T1 = 4'b0010, T2 = 4'b0011,
                  T3 = 4'b0100, T4 = 4'b0101, T5 = 4'b0110, T6 = 4'b0111,
                  T7 = 4'b1000, T8 = 4'b1001, T9 = 4'b1010;

        reg [3:0] Present_state = Default;
        reg preload_reg = 1'b0;

        datapath #(.BITS(BITS), .REGISTERS(REGISTERS), .RAMSIZE(RAMSIZE)) DUT(
                  reset, clk, rClk,
                  CONin, PCin, IRin, RYin, RZin, MARin, HILOin, OUTPUTin, INTERin, MDRin, Read, Write, INPUTout, MDRout, HILOout, RZout, PCout, Cout, INTERout,
                  BAout, Gra, Grb, Grc, Rout, Rin,
                  ADD, SUB, MUL, DIV, SHR, SHL, ROR, ROL, AND, OR, NEGATE, NOT, IncPC,
                  INPUTUnit,
                  regSelectStreamLO, regSelectStreamHI,
                  busLO, busHI,
                  MARVal,
                  RZVal,
                  IRVal,
                  LOVal, HIVal,
                  OUTPUTUnit,
                  c_sign_extended,
                  MDRVal,
                  INTERHIVal, INTERLOVal,
                  CON);


        initial begin
                clk = 0;
                forever #10 clk = ~clk;
        end

        initial begin
                rClk = 1;
                forever #5 rClk = ~rClk;
        end

        always @(posedge clk) // finite state machine; if clock rising-edge
        begin
                case (Present_state)
                        Default : Present_state = T0;
                        T0 : Present_state = T1;
                        T1 : Present_state = T2;
                        T2 : begin
                                if(preload_reg == 1)
                                        Present_state = T5;
                                else
                                        Present_state = T3;
                        end
                        T3 : Present_state = T4;
                        T4 : Present_state = T1;
                        T5 : Present_state = T6;
                        T6 : Present_state = T7;
                endcase
        end

        always @(Present_state) // do the required job in each state
        begin
                case (Present_state) // assert the required signals in each clock cycle
                        Default: begin
                                reset <= 1;
                                CONin <= 0; PCin <= 0; IRin <= 0; RYin <= 0; RZin <= 0; MARin <= 0; HILOin <= 0; MDRin <= 0; OUTPUTin <= 0; INTERin <= 0;
                                Read <= 0; Write <= 0;
                                INPUTout <= 0; MDRout <= 0; HILOout <= 0; RZout <= 0; PCout <= 0; Cout <= 0; INTERout <= 0;
                                BAout <= 0; Gra <= 0; Grb <= 0; Grc <= 0; Rout <= 0; Rin <= 0;
                                ADD <= 0; SUB <= 0; MUL <= 0; DIV <= 0; SHR <= 0; SHL <= 0; ROR <= 0; ROL <= 0; AND <= 0; OR <= 0; NEGATE <= 0; NOT <= 0; IncPC <= 0;
                                INPUTUnit <= {BITS{1'b0}};
                                #5 reset <= 0;
                        end
                        T0: begin
                                MARin <= 1; IncPC <= 1; RZin <= 1; PCout <= 1;
                        end
                        T1: begin
                                Read <= 1;
                                PCin <= 1; MDRin <= 1;
                                #5 RZout <= 1; PCout <= 0; MARin <= 0; IncPC <= 0; RZin <= 0;
                        end
                        T2: begin
                                IRin <= 1;
                                #5 PCin <= 0; RZout <= 0; MDRout <= 1; MDRin <= 0; Read <= 0;
                        end
                        T3: begin
                                Gra <= 1; Rin <= 1;
                                #5 MDRout <= 0; IRin <= 0;Cout <= 1;
                        end
                        T4: begin
                                MARin <= 1; IncPC <= 1; RZin <= 1;
                                #5 Gra <= 0; Rin <= 0; Cout <= 0; PCout <= 1; preload_reg <= 1;
                        end
                        T5: begin
                                RYin <= 1; Grb <= 1;
                                #5 IRin <= 0; MDRout <= 0; BAout <= 1;
                        end
                        T6: begin
                                ADD <= 1; RZin <= 1;
                                #5 RYin <= 0; Grb <= 0; BAout <= 0; Cout <= 1;
                        end
                        T7: begin
                                Gra <= 1; Rin <= 1;
                                #5 ADD <= 0; RZin <= 0; RZout <= 1; Cout <= 0;
                        end
                endcase
        end
endmodule
```

LDI Waveform

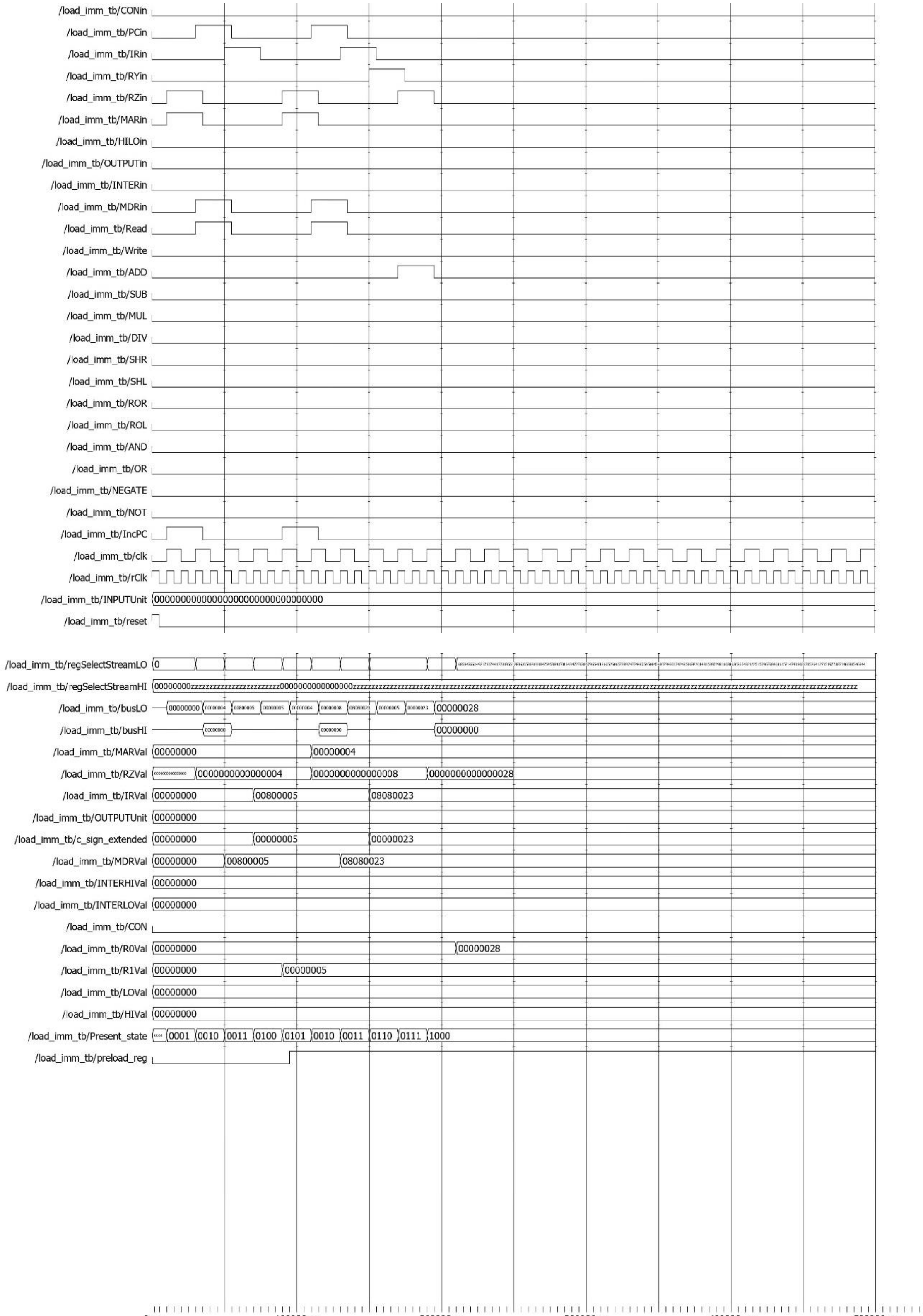
LDI Case 1



Memory dump before/after LDI Case 1

```
// memory data file (do not edit the following line - required for mem load use)
// instance=/load_imm_tb/DUT/RAM/RAM
// format=mti addressradix=h dataradix=h version=1.0 wordsperline=8
  0: 00800000 xxxxxxxx xxxxxxxx xxxxxxxx 08800055 xxxxxxxx xxxxxxxx xxxxxxxx
  8: xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx
*
1f8: xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx
```

LDI Case 2


/load_imm_tb/CONin
/load_imm_tb/PCin
/load_imm_tb/IRin
/load_imm_tb/RYin
/load_imm_tb/RZin
/load_imm_tb/MARin
/load_imm_tb/HILOin
/load_imm_tb/OUTPUTin
/load_imm_tb/INTERin
/load_imm_tb/MDRin
/load_imm_tb/Read
/load_imm_tb/Write
/load_imm_tb/ADD
/load_imm_tb/SUB
/load_imm_tb/MUL
/load_imm_tb/DIV
/load_imm_tb/SHR
/load_imm_tb/SHL
/load_imm_tb/ROR
/load_imm_tb/ROL
/load_imm_tb/AND
/load_imm_tb/OR
/load_imm_tb/NEGATE
/load_imm_tb/NOT
/load_imm_tb/IncPC
/load_imm_tb/clk
/load_imm_tb/rClk
/load_imm_tb/INPUTUnit 00000000000000000000000000000000
/load_imm_tb/reset
/load_imm_tb/regSelectStreamLO 0
/load_imm_tb/regSelectStreamHI 00000000
/load_imm_tb/busLO 00000000 00000028
/load_imm_tb/busHI 00000000
/load_imm_tb/MARVal 00000000 00000004
/load_imm_tb/RZVal 0000000000000004 0000000000000008 0000000000000028
/load_imm_tb/IRVal 00000000 00800005 08080023
/load_imm_tb/OUTPUTUnit 00000000
/load_imm_tb/c_sign_extended 00000000 00000005 00000023
/load_imm_tb/MDRVal 00000000 00800005 08080023
/load_imm_tb/INTERHIVal 00000000
/load_imm_tb/INTERLOVal 00000000
/load_imm_tb/CON
/load_imm_tb/R0Val 00000000 00000028
/load_imm_tb/R1Val 00000000 00000005
/load_imm_tb/LOVal 00000000
/load_imm_tb/HIVal 00000000
/load_imm_tb/Present_state 0001 0010 0011 0100 0101 0010 0011 0110 0111 1000
/load_imm_tb/preload_reg
0 ps
100000 ps
200000 ps
300000 ps
400000 ps
500000 ps

Memory dump before/after LD Case 2

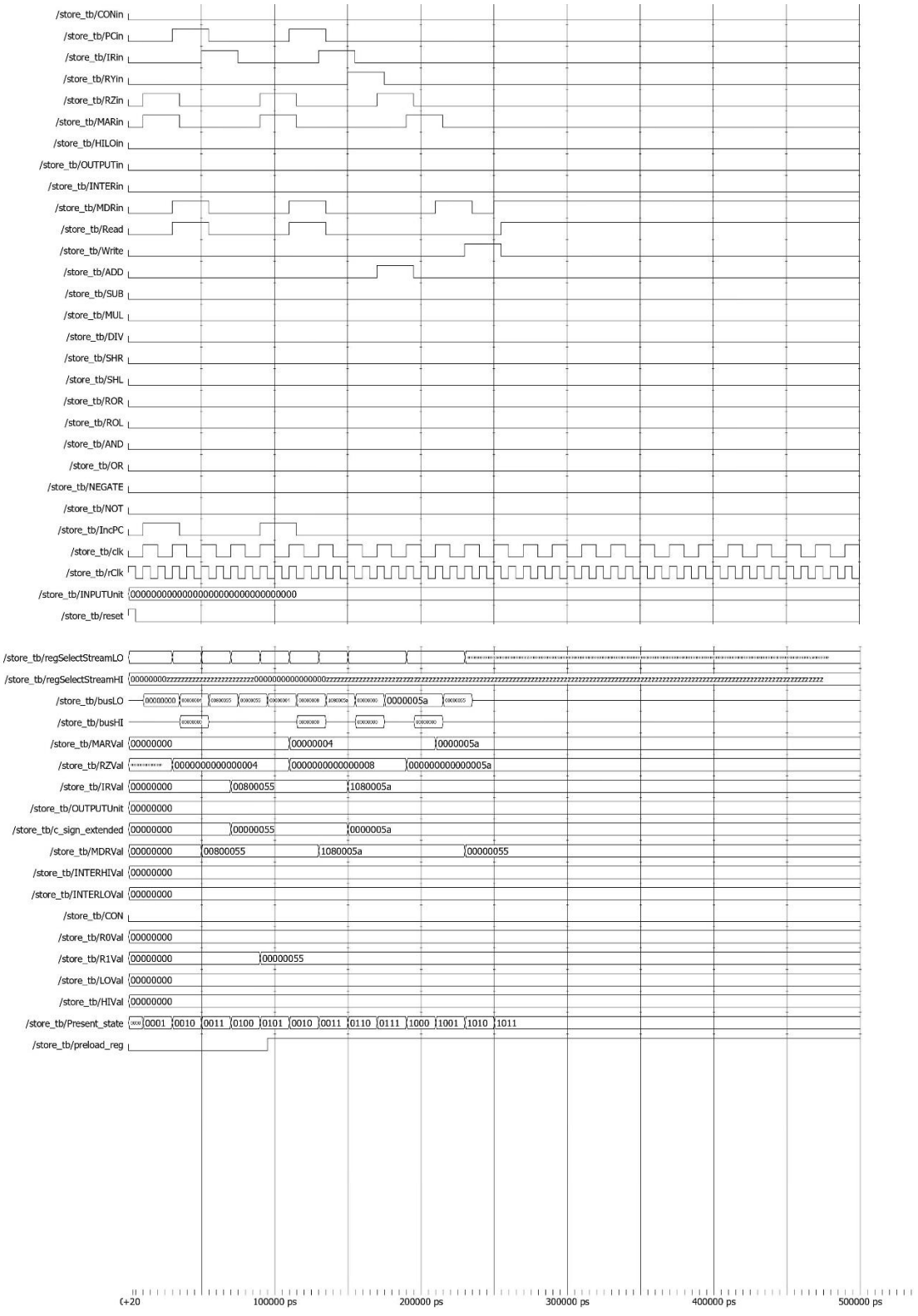
```
// memory data file (do not edit the following line - required for mem load use)
// instance=/load_imm_tb/DUT/RAM/RAM
// format=mti addressradix=h dataradix=h version=1.0 wordsperline=8
  0: 00800005 xxxxxxxx xxxxxxxx xxxxxxxx 08080023 xxxxxxxx xxxxxxxx xxxxxxxx
  8: xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx
*
1f8: xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx
```

ST Testbench

```
// Store tb

`timescale 1ns/10ps
module store_tb;

	parameter BITS=32, REGISTERS=16, TOT_REGISTERS=REGISTERS+7, RAMSIZE=512;

	reg INPUTout, MDRout, HILOout, RZout, PCout, Cout, INTERout, BAout, Gra, Grb, Grc, Rout, Rin; // add any other signals to see in your simulation
	reg CONin, PCin, IRin, RYin, RZin, MARin, HILOin, OUTPUTin, INTERin, MDRin;
	reg Read, Write, ADD, SUB, MUL, DIV, SHR, SHL, ROR, ROL, AND, OR, NEGATE, NOT, IncPC;
	reg clk, rClk;
	reg [BITS-1:0] INPUTUnit;
	reg reset;
	wire [(BITS*TOT_REGISTERS)-1:0] regSelectStreamLO, regSelectStreamHI;
	wire [BITS-1:0] busLO, busHI;
	wire [BITS-1:0] MARVal;
	wire [(BITS*2)-1:0] RZVal;
	wire [BITS-1:0] IRVal;
	wire [BITS-1:0] OUTPUTUnit;
	wire [BITS-1:0] c_sign_extended;
	wire [BITS-1:0] MDRVal;
	wire [BITS-1:0] INTERHIVal, INTERLOVal;
	wire CON;
	wire [BITS-1:0] R0Val, R1Val, LOVal, HIVal;
	assign R0Val = regSelectStreamLO[(1*BITS)-1:BITS*0];
	assign R1Val = regSelectStreamLO[(2*BITS)-1:BITS*1];

	parameter Default = 4'b0000, T0 = 4'b0001, T1 = 4'b0010, T2 = 4'b0011,
			T3 = 4'b0100, T4 = 4'b0101, T5 = 4'b0110, T6 = 4'b0111,
			T7 = 4'b1000, T8 = 4'b1001, T9 = 4'b1010, T10 = 4'b1011;

	reg [3:0] Present_state = Default;
	reg preload_reg = 1'b0;

	datapath #(.BITS(BITS), .REGISTERS(REGISTERS), .RAMSIZE(RAMSIZE)) DUT(
			reset, clk, rClk,
			CONin, PCin, IRin, RYin, RZin, MARin, HILOin, OUTPUTin, INTERin, MDRin, Read, Write, INPUTout, MDRout, HILOout, RZout, PCout, Cout, INTERout,
			BAout, Gra, Grb, Grc, Rout, Rin,
			ADD, SUB, MUL, DIV, SHR, SHL, ROR, ROL, AND, OR, NEGATE, NOT, IncPC,
			INPUTUnit,
			regSelectStreamLO, regSelectStreamHI,
			busLO, busHI,
			MARVal,
			RZVal,
			IRVal,
			LOVal, HIVal,
			OUTPUTUnit,
			c_sign_extended,
			MDRVal,
			INTERHIVal, INTERLOVal,
			CON);

	initial begin
		clk = 0;
		forever #10 clk = ~clk;
	end

	initial begin
		rClk = 1;
		forever #5 rClk = ~rClk;
	end

	always @(posedge clk) // finite state machine; if clock rising-edge
	begin
		case (Present_state)
			Default : Present_state = T0;
			T0 : Present_state = T1;
			T1 : Present_state = T2;
			T2 : begin
				if(preload_reg == 1)
					Present_state = T5;
				else
					Present_state = T3;
			end
			T3 : Present_state = T4;
			T4 : Present_state = T1;
			T5 : Present_state = T6;
			T6 : Present_state = T7;
			T7 : Present_state = T8;
			T8 : Present_state = T9;
			T9 : Present_state = T10;
		endcase
	end

	always @(Present_state) // do the required job in each state
	begin
		case (Present_state) // assert the required signals in each clock cycle
			Default: begin
				reset <= 1;
				CONin <= 0; PCin <= 0; IRin <= 0; RYin <= 0; RZin <= 0; MARin <= 0; HILOin <= 0; MDRin <= 0; OUTPUTin <= 0; INTERin <= 0;
				Read <= 0; Write <= 0;
				INPUTout <= 0; MDRout <= 0; HILOout <= 0; RZout <= 0; PCout <= 0; Cout <= 0; INTERout <= 0;
				BAout <= 0; Gra <= 0; Grb <= 0; Grc <= 0; Rout <= 0; Rin <= 0;
				ADD <= 0; SUB <= 0; MUL <= 0; DIV <= 0; SHR <= 0; SHL <= 0; ROR <= 0; ROL <= 0; AND <= 0; OR <= 0; NEGATE <= 0; NOT <= 0; IncPC <= 0;
				INPUTUnit <= {BITS{1'b0}};
				#5 reset <= 0;
			end
			T0: begin
				MARin <= 1; IncPC <= 1; RZin <= 1; PCout <= 1;
			end
			T1: begin
				Read <= 1;
				PCin <= 1; MDRin <= 1;
				#5 RZout <= 1; PCout <= 0; MARin <= 0; IncPC <= 0; RZin <= 0;
			end
			T2: begin
				IRin <= 1;
				#5 PCin <= 0; RZout <= 0; MDRout <= 1; MDRin <= 0; Read <= 0;
			end
			T3: begin
				Gra <= 1; Rin <= 1;
				#5 MDRout <= 0; IRin <= 0;Cout <= 1;
			end
			T4: begin
				MARin <= 1; IncPC <= 1; RZin <= 1;
				#5 Gra <= 0; Rin <= 0; Cout <= 0; PCout <= 1; preload_reg <= 1;
			end
			T5: begin
				RYin <= 1; Grb <= 1;
				#5 IRin <= 0; MDRout <= 0; BAout <= 1;
			end
			T6: begin
				ADD <= 1; RZin <= 1;
				#5 RYin <= 0; Grb <= 0; BAout <= 0; Cout <= 1;
			end
			T7: begin
				MARin <= 1;
				#5 Cout <= 0; ADD <= 0; RZin <= 0; RZout <= 1;
			end
				T8: begin
				Gra <= 1; MDRin <= 1;
				#5 RZout <= 0; MARin <= 0; Rout <= 1;
			end
			T9: begin
				Write <= 1;
				#5 Gra <= 0; MDRin <= 0; Rout <= 0;
			end
			T10: begin
				MDRin <= 1;
				#5 Write <= 0; Read <= 1;
			end
		endcase
	end
endmodule
```

ST Waveform

ST Case 1



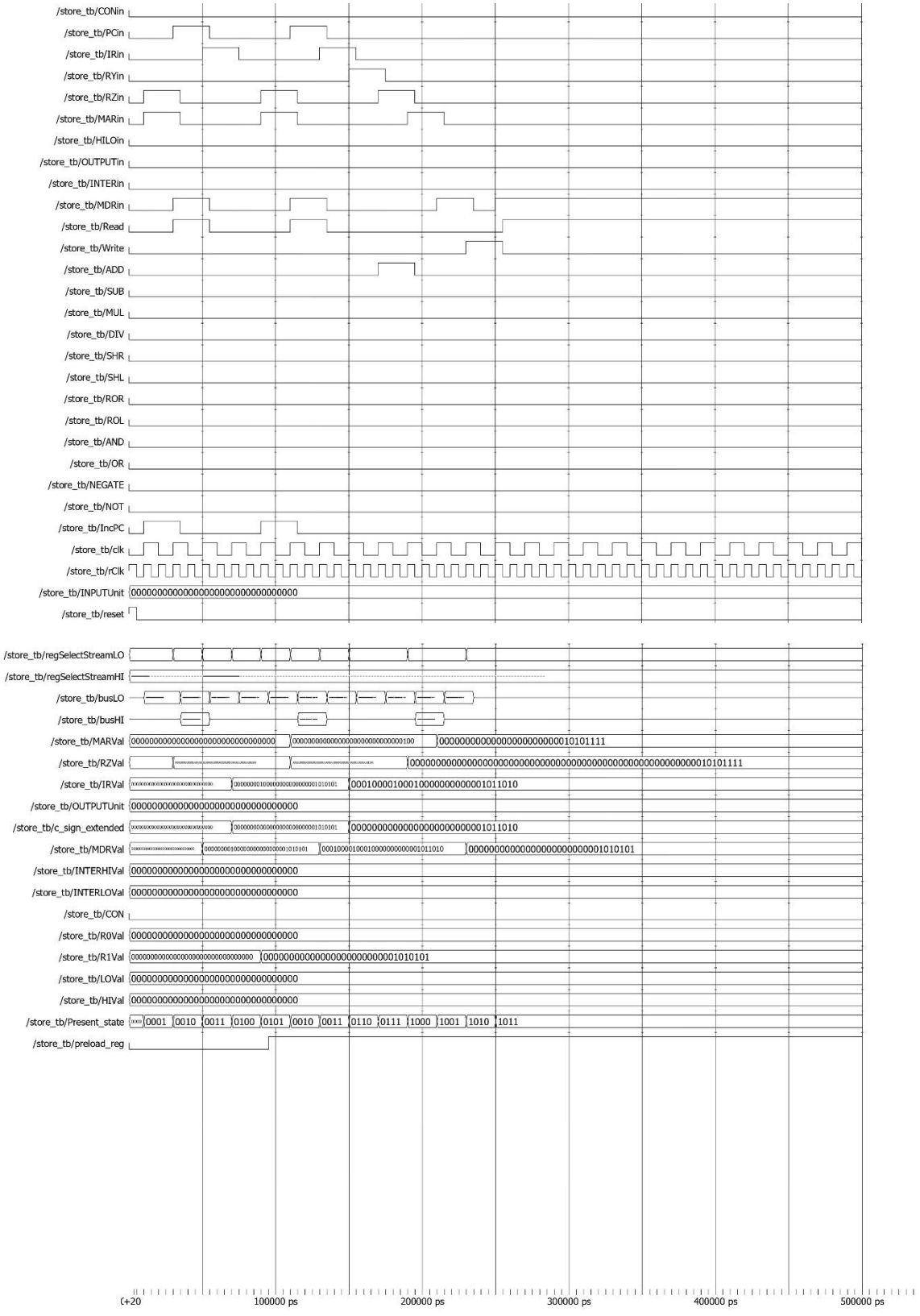
Memory dump before ST Case 1

```
// memory data file (do not edit the following line - required for mem load use)
// instance=/store_tb/DUT/RAM/RAM
// format=mti addressradix=h dataradix=h version=1.0 wordsperline=8
  0: 00800055 xxxxxxxx xxxxxxxx xxxxxxxx 1080005a xxxxxxxx xxxxxxxx xxxxxxxx
  8: xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx
*
 58: xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx
 60: xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx
*
1f8: xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx
```

Memory dump after ST Case 1

```
// memory data file (do not edit the following line - required for mem load use)
// instance=/store_tb/DUT/RAM/RAM
// format=mti addressradix=h dataradix=h version=1.0 wordsperline=8
  0: 00800055 xxxxxxxx xxxxxxxx xxxxxxxx 1080005a xxxxxxxx xxxxxxxx xxxxxxxx
  8: xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx
*
 58: xxxxxxxx xxxxxxxx 00000055 xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx
 60: xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx
*
1f8: xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx
```

ST Case 2


/store_tb/CONin
/store_tb/PCin
/store_tb/IRin
/store_tb/RYin
/store_tb/RZin
/store_tb/MARin
/store_tb/HILOin
/store_tb/OUTPUTin
/store_tb/INTERin
/store_tb/MDRin
/store_tb/Read
/store_tb/Write
/store_tb/ADD
/store_tb/SUB
/store_tb/MUL
/store_tb/DIV
/store_tb/SHR
/store_tb/SHL
/store_tb/ROR
/store_tb/ROL
/store_tb/AND
/store_tb/OR
/store_tb/NEGATE
/store_tb/NOT
/store_tb/IncPC
/store_tb/clk
/store_tb/rClk
/store_tb/INPUTUnit
/store_tb/reset
/store_tb/regSelectStreamLO
/store_tb/regSelectStreamHI
/store_tb/busLO
/store_tb/busHI
/store_tb/MARVal
/store_tb/RZVal
/store_tb/IRVal
/store_tb/OUTPUTUnit
/store_tb/c_sign_extended
/store_tb/MDRVal
/store_tb/INTERHIVal
/store_tb/INTERLOVal
/store_tb/CON
/store_tb/R0Val
/store_tb/R1Val
/store_tb/LOVal
/store_tb/HIVal
/store_tb/Present_state
0001
0010
0011
0100
0101
0010
0011
0110
0111
1000
1001
1010
1011
/store_tb/preload_reg
(+20
100000 ps
200000 ps
300000 ps
400000 ps
500000 ps

Memory dump before ST Case 2

```
// memory data file (do not edit the following line - required for mem load use)
// instance=/store_tb/DUT/RAM/RAM
// format=mti addressradix=h dataradix=h version=1.0 wordsperline=8
  0: 00800055 xxxxxxxx xxxxxxxx xxxxxxxx 1080005a xxxxxxxx xxxxxxxx xxxxxxxx
  8: xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx
*
 58: xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx
 60: xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx
*
1f8: xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx
```

Memory dump after ST Case 2

```
// memory data file (do not edit the following line - required for mem load use)
// instance=/store_tb/DUT/RAM/RAM
// format=mti addressradix=h dataradix=h version=1.0 wordsperline=8
  0: 00800055 xxxxxxxx xxxxxxxx xxxxxxxx 1088005a xxxxxxxx xxxxxxxx xxxxxxxx
  8: xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx
*
 a8: xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx 00000055
 b0: xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx
*
1f8: xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx
```

ADDI Testbench

```
// addi tb

`timescale 1ns/10ps
module addi_tb;

	parameter BITS=32, REGISTERS=16, TOT_REGISTERS=REGISTERS+7, RAMSIZE=512;

	reg INPUTout, MDRout, HILOout, RZout, PCout, Cout, INTERout, BAout, Gra, Grb, Grc, Rout, Rin; // add any other signals to see in your simulation
	reg CONin, PCin, IRin, RYin, RZin, MARin, HILOin, OUTPUTin, INTERin, MDRin;
	reg Read, Write, ADD, SUB, MUL, DIV, SHR, SHL, ROR, ROL, AND, OR, NEGATE, NOT, IncPC;
	reg clk, rClk;
	reg [BITS-1:0] INPUTUnit;
	reg reset;
	wire [(BITS*TOT_REGISTERS)-1:0] regSelectStreamLO, regSelectStreamHI;
	wire [BITS-1:0] busLO, busHI;
	wire [BITS-1:0] MARVal;
	wire [(BITS*2)-1:0] RZVal;
	wire [BITS-1:0] IRVal;
	wire [BITS-1:0] OUTPUTUnit;
	wire [BITS-1:0] c_sign_extended;
	wire [BITS-1:0] MDRVal;
	wire [BITS-1:0] INTERHIVal, INTERLOVal;
	wire CON;
	wire [BITS-1:0] R0Val, R1Val, R2Val, LOVal, HIVal;
	assign R0Val = regSelectStreamLO[(1*BITS)-1:BITS*0];
	assign R1Val = regSelectStreamLO[(2*BITS)-1:BITS*1];
	assign R2Val = regSelectStreamLO[(3*BITS)-1:BITS*2];

	parameter Default = 4'b0000, T0 = 4'b0001, T1 = 4'b0010, T2 = 4'b0011,
			T3 = 4'b0100, T4 = 4'b0101, T5 = 4'b0110, T6 = 4'b0111,
			T7 = 4'b1000;

	reg [3:0] Present_state = Default;
	reg preload_reg = 1'b0;

	datapath #(.BITS(BITS), .REGISTERS(REGISTERS), .RAMSIZE(RAMSIZE)) DUT(
			reset, clk, rClk,
			CONin, PCin, IRin, RYin, RZin, MARin, HILOin, OUTPUTin, INTERin, MDRin, Read, Write, INPUTout, MDRout, HILOout, RZout, PCout, Cout, INTERout,
			BAout, Gra, Grb, Grc, Rout, Rin,
			ADD, SUB, MUL, DIV, SHR, SHL, ROR, ROL, AND, OR, NEGATE, NOT, IncPC,
			INPUTUnit,
			regSelectStreamLO, regSelectStreamHI,
			busLO, busHI,
			MARVal,
			RZVal,
			IRVal,
			LOVal, HIVal,
			OUTPUTUnit,
			c_sign_extended,
			MDRVal,
			INTERHIVal, INTERLOVal,
			CON);


	initial begin
		clk = 0;
		forever #10 clk = ~clk;
	end

	initial begin
		rClk = 1;
		forever #5 rClk = ~rClk;
	end

	always @(posedge clk) // finite state machine; if clock rising-edge
	begin
		case (Present_state)
			Default : Present_state = T0;
			T0 : Present_state = T1;
			T1 : Present_state = T2;
			T2 : begin
				if(preload_reg == 1)
					Present_state = T5;
				else
					Present_state = T3;
			end
			T3 : Present_state = T4;
			T4 : Present_state = T1;
			T5 : Present_state = T6;
			T6 : Present_state = T7;
		endcase
	end

	always @(Present_state) // do the required job in each state
	begin
		case (Present_state) // assert the required signals in each clock cycle
			Default: begin
				reset <= 1;
				CONin <= 0; PCin <= 0; IRin <= 0; RYin <= 0; RZin <= 0; MARin <= 0; HILOin <= 0; MDRin <= 0; OUTPUTin <= 0; INTERin <= 0;
				Read <= 0; Write <= 0;
				INPUTout <= 0; MDRout <= 0; HILOout <= 0; RZout <= 0; PCout <= 0; Cout <= 0; INTERout <= 0;
				BAout <= 0; Gra <= 0; Grb <= 0; Grc <= 0; Rout <= 0; Rin <= 0;
				ADD <= 0; SUB <= 0; MUL <= 0; DIV <= 0; SHR <= 0; SHL <= 0; ROR <= 0; ROL <= 0; AND <= 0; OR <= 0; NEGATE <= 0; NOT <= 0; IncPC <= 0;
				INPUTUnit <= {BITS{1'b0}};
				#5 reset <= 0;
			end
			T0: begin
				MARin <= 1; IncPC <= 1; RZin <= 1; PCout <= 1;
			end
			T1: begin
				Read <= 1;
				PCin <= 1; MDRin <= 1;
				#5 RZout <= 1; PCout <= 0; MARin <= 0; IncPC <= 0; RZin <= 0;
			end
			T2: begin
				IRin <= 1;
				#5 PCin <= 0; RZout <= 0; MDRout <= 1; MDRin <= 0; Read <= 0;
			end
			T3: begin
				Gra <= 1; Rin <= 1;
				#5 MDRout <= 0; IRin <= 0;Cout <= 1;
			end
			T4: begin
				MARin <= 1; IncPC <= 1; RZin <= 1;
				#5 Gra <= 0; Rin <= 0; Cout <= 0; PCout <= 1; preload_reg <= 1;
			end
			T5: begin
				Grb <= 1; RYin <= 1;
				#5 MDRout <= 0; IRin <= 0; Rout <=1;
			end
			T6: begin
				ADD <= 1; RZin <= 1;
				#5 Rout <= 0; Grb <= 0; RYin <= 0; Cout <= 1;
			end
			T7: begin
				Gra <= 1; Rin <= 1;
				#5 Cout <= 0; ADD <= 0; RZin <= 0; RZout <= 1;
			end
		endcase
	end
endmodule
```

ADDI Waveform

ANDI Testbench

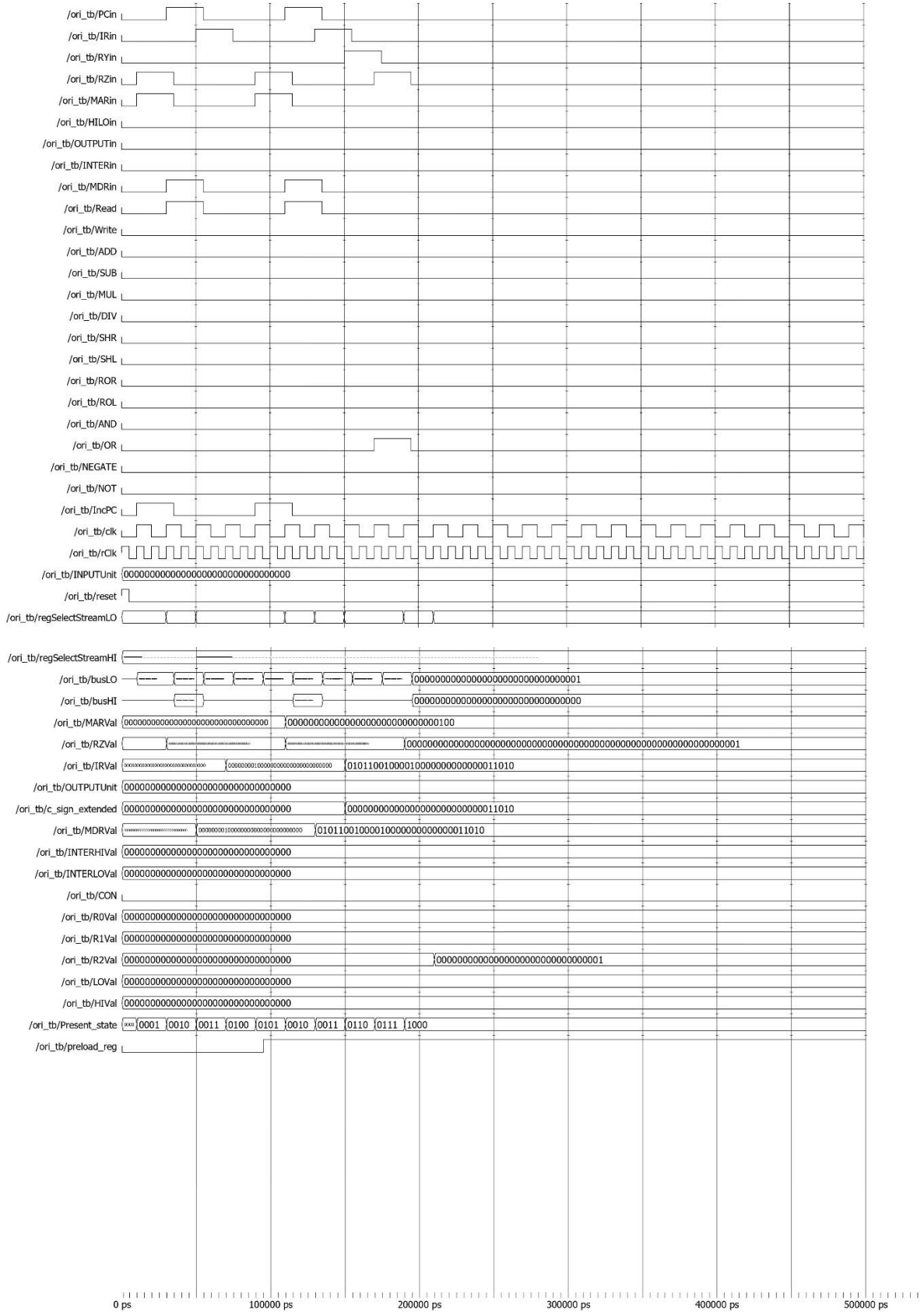
```
// andi tb

`timescale 1ns/10ps
module andi_tb;

	parameter BITS=32, REGISTERS=16, TOT_REGISTERS=REGISTERS+7, RAMSIZE=512;

	reg INPUTout, MDRout, HILOout, RZout, PCout, Cout, INTERout, BAout, Gra, Grb, Grc, Rout, Rin; // add any other signals to see in your simulation
	reg CONin, PCin, IRin, RYin, RZin, MARin, HILOin, OUTPUTin, INTERin, MDRin;
	reg Read, Write, ADD, SUB, MUL, DIV, SHR, SHL, ROR, ROL, AND, OR, NEGATE, NOT, IncPC;
	reg clk, rClk;
	reg [BITS-1:0] INPUTUnit;
	reg reset;
	wire [(BITS*TOT_REGISTERS)-1:0] regSelectStreamLO, regSelectStreamHI;
	wire [BITS-1:0] busLO, busHI;
	wire [BITS-1:0] MARVal;
	wire [(BITS*2)-1:0] RZVal;
	wire [BITS-1:0] IRVal;
	wire [BITS-1:0] OUTPUTUnit;
	wire [BITS-1:0] c_sign_extended;
	wire [BITS-1:0] MDRVal;
	wire [BITS-1:0] INTERHIVal, INTERLOVal;
	wire CON;
	wire [BITS-1:0] R0Val, R1Val, R2Val, LOVal, HIVal;
	assign R0Val = regSelectStreamLO[(1*BITS)-1:BITS*0];
	assign R1Val = regSelectStreamLO[(2*BITS)-1:BITS*1];
	assign R2Val = regSelectStreamLO[(3*BITS)-1:BITS*2];

	parameter Default = 4'b0000, T0 = 4'b0001, T1 = 4'b0010, T2 = 4'b0011,
			T3 = 4'b0100, T4 = 4'b0101, T5 = 4'b0110, T6 = 4'b0111,
			T7 = 4'b1000;

	reg [3:0] Present_state = Default;
	reg preload_reg = 1'b0;

	datapath #(.BITS(BITS), .REGISTERS(REGISTERS), .RAMSIZE(RAMSIZE)) DUT(
			reset, clk, rClk,
			CONin, PCin, IRin, RYin, RZin, MARin, HILOin, OUTPUTin, INTERin, MDRin, Read, Write, INPUTout, MDRout, HILOout, RZout, PCout, Cout, INTERout,
			BAout, Gra, Grb, Grc, Rout, Rin,
			ADD, SUB, MUL, DIV, SHR, SHL, ROR, ROL, AND, OR, NEGATE, NOT, IncPC,
			INPUTUnit,
			regSelectStreamLO, regSelectStreamHI,
			busLO, busHI,
			MARVal,
			RZVal,
			IRVal,
			LOVal, HIVal,
			OUTPUTUnit,
			c_sign_extended,
			MDRVal,
			INTERHIVal, INTERLOVal,
			CON);

	initial begin
		clk = 0;
		forever #10 clk = ~clk;
	end

	initial begin
		rClk = 1;
		forever #5 rClk = ~rClk;
	end

	always @(posedge clk) // finite state machine; if clock rising-edge
	begin
		case (Present_state)
			Default : Present_state = T0;
			T0 : Present_state = T1;
			T1 : Present_state = T2;
			T2 : begin
				if(preload_reg == 1)
					Present_state = T5;
				else
					Present_state = T3;
			end
			T3 : Present_state = T4;
			T4 : Present_state = T1;
			T5 : Present_state = T6;
			T6 : Present_state = T7;
		endcase
	end

	always @(Present_state) // do the required job in each state
	begin
		case (Present_state) // assert the required signals in each clock cycle
			Default: begin
				reset <= 1;
				CONin <= 0; PCin <= 0; IRin <= 0; RYin <= 0; RZin <= 0; MARin <= 0; HILOin <= 0; MDRin <= 0; OUTPUTin <= 0; INTERin <= 0;
				Read <= 0; Write <= 0;
				INPUTout <= 0; MDRout <= 0; HILOout <= 0; RZout <= 0; PCout <= 0; Cout <= 0; INTERout <= 0;
				BAout <= 0; Gra <= 0; Grb <= 0; Grc <= 0; Rout <= 0; Rin <= 0;
				ADD <= 0; SUB <= 0; MUL <= 0; DIV <= 0; SHR <= 0; SHL <= 0; ROR <= 0; ROL <= 0; AND <= 0; OR <= 0; NEGATE <= 0; NOT <= 0; IncPC <= 0;
				INPUTUnit <= {BITS{1'b0}};
				#5 reset <= 0;
			end
			T0: begin
				MARin <= 1; IncPC <= 1; RZin <= 1; PCout <= 1;
			end
			T1: begin
				Read <= 1;
				PCin <= 1; MDRin <= 1;
				#5 RZout <= 1; PCout <= 0; MARin <= 0; IncPC <= 0; RZin <= 0;
			end
			T2: begin
				IRin <= 1;
				#5 PCin <= 0; RZout <= 0; MDRout <= 1; MDRin <= 0; Read <= 0;
			end
			T3: begin
				Gra <= 1; Rin <= 1;
				#5 MDRout <= 0; IRin <= 0;Cout <= 1;
			end
			T4: begin
				MARin <= 1; IncPC <= 1; RZin <= 1;
				#5 Gra <= 0; Rin <= 0; Cout <= 0; PCout <= 1; preload_reg <= 1;
			end
			T5: begin
				Grb <= 1; RYin <= 1;
				#5 MDRout <= 0; IRin <= 0; Rout <=1;
			end
			T6: begin
				AND <= 1; RZin <= 1;
				#5 Rout <= 0; Grb <= 0; RYin <= 0; Cout <= 1;
			end
			T7: begin
				Gra <= 1; Rin <= 1;
				#5 Cout <= 0; AND <= 0; RZin <= 0; RZout <= 1;
			end
		endcase
	end
endmodule
```

ANDI Waveform

ORI Testbench

```
// ori tb

`timescale 1ns/10ps
module ori_tb;

	parameter BITS=32, REGISTERS=16, TOT_REGISTERS=REGISTERS+7, RAMSIZE=512;

	reg INPUTout, MDRout, HILOout, RZout, PCout, Cout, INTERout, BAout, Gra, Grb, Grc, Rout, Rin; // add any other signals to see in your simulation
	reg CONin, PCin, IRin, RYin, RZin, MARin, HILOin, OUTPUTin, INTERin, MDRin;
	reg Read, Write, ADD, SUB, MUL, DIV, SHR, SHL, ROR, ROL, AND, OR, NEGATE, NOT, IncPC;
	reg clk, rClk;
	reg [BITS-1:0] INPUTUnit;
	reg reset;
	wire [(BITS*TOT_REGISTERS)-1:0] regSelectStreamLO, regSelectStreamHI;
	wire [BITS-1:0] busLO, busHI;
	wire [BITS-1:0] MARVal;
	wire [(BITS*2)-1:0] RZVal;
	wire [BITS-1:0] IRVal;
	wire [BITS-1:0] OUTPUTUnit;
	wire [BITS-1:0] c_sign_extended;
	wire [BITS-1:0] MDRVal;
	wire [BITS-1:0] INTERHIVal, INTERLOVal;
	wire CON;
	wire [BITS-1:0] R0Val, R1Val, R2Val, LOVal, HIVal;
	assign R0Val = regSelectStreamLO[(1*BITS)-1:BITS*0];
	assign R1Val = regSelectStreamLO[(2*BITS)-1:BITS*1];
	assign R2Val = regSelectStreamLO[(3*BITS)-1:BITS*2];

	parameter Default = 4'b0000, T0 = 4'b0001, T1 = 4'b0010, T2 = 4'b0011,
			T3 = 4'b0100, T4 = 4'b0101, T5 = 4'b0110, T6 = 4'b0111,
			T7 = 4'b1000;

	reg [3:0] Present_state = Default;
	reg preload_reg = 1'b0;

	datapath #(.BITS(BITS), .REGISTERS(REGISTERS), .RAMSIZE(RAMSIZE)) DUT(
			reset, clk, rClk,
			CONin, PCin, IRin, RYin, RZin, MARin, HILOin, OUTPUTin, INTERin, MDRin, Read, Write, INPUTout, MDRout, HILOout, RZout, PCout, Cout, INTERout,
			BAout, Gra, Grb, Grc, Rout, Rin,
			ADD, SUB, MUL, DIV, SHR, SHL, ROR, ROL, AND, OR, NEGATE, NOT, IncPC,
			INPUTUnit,
			regSelectStreamLO, regSelectStreamHI,
			busLO, busHI,
			MARVal,
			RZVal,
			IRVal,
			LOVal, HIVal,
			OUTPUTUnit,
			c_sign_extended,
			MDRVal,
			INTERHIVal, INTERLOVal,
			CON);


	initial begin
		clk = 0;
		forever #10 clk = ~clk;
	end

	initial begin
		rClk = 1;
		forever #5 rClk = ~rClk;
	end

	always @(posedge clk) // finite state machine; if clock rising-edge
	begin
		case (Present_state)
			Default : Present_state = T0;
			T0 : Present_state = T1;
			T1 : Present_state = T2;
			T2 : begin
				if(preload_reg == 1)
					Present_state = T5;
				else
					Present_state = T3;
			end
			T3 : Present_state = T4;
			T4 : Present_state = T1;
			T5 : Present_state = T6;
			T6 : Present_state = T7;
		endcase
	end

	always @(Present_state) // do the required job in each state
	begin
		case (Present_state) // assert the required signals in each clock cycle
			Default: begin
				reset <= 1;
				CONin <= 0; PCin <= 0; IRin <= 0; RYin <= 0; RZin <= 0; MARin <= 0; HILOin <= 0; MDRin <= 0; OUTPUTin <= 0; INTERin <= 0;
				Read <= 0; Write <= 0;
				INPUTout <= 0; MDRout <= 0; HILOout <= 0; RZout <= 0; PCout <= 0; Cout <= 0; INTERout <= 0;
				BAout <= 0; Gra <= 0; Grb <= 0; Grc <= 0; Rout <= 0; Rin <= 0;
				ADD <= 0; SUB <= 0; MUL <= 0; DIV <= 0; SHR <= 0; SHL <= 0; ROR <= 0; ROL <= 0; AND <= 0; OR <= 0; NEGATE <= 0; NOT <= 0; IncPC <= 0;
				INPUTUnit <= {BITS{1'b0}};
				#5 reset <= 0;
			end
			T0: begin
				MARin <= 1; IncPC <= 1; RZin <= 1; PCout <= 1;
			end
			T1: begin
				Read <= 1;
				PCin <= 1; MDRin <= 1;
				#5 RZout <= 1; PCout <= 0; MARin <= 0; IncPC <= 0; RZin <= 0;
			end
			T2: begin
				IRin <= 1;
				#5 PCin <= 0; RZout <= 0; MDRout <= 1; MDRin <= 0; Read <= 0;
			end
			T3: begin
				Gra <= 1; Rin <= 1;
				#5 MDRout <= 0; IRin <= 0;Cout <= 1;
			end
			T4: begin
				MARin <= 1; IncPC <= 1; RZin <= 1;
				#5 Gra <= 0; Rin <= 0; Cout <= 0; PCout <= 1; preload_reg <= 1;
			end
			T5: begin
				Grb <= 1; RYin <= 1;
				#5 MDRout <= 0; IRin <= 0; Rout <=1;
			end
			T6: begin
				OR <= 1; RZin <= 1;
				#5 Rout <= 0; Grb <= 0; RYin <= 0; Cout <= 1;
			end
			T7: begin
				Gra <= 1; Rin <= 1;
				#5 Cout <= 0; OR <= 0; RZin <= 0; RZout <= 1;
			end
		endcase
	end
endmodule
```

ORI Waveform

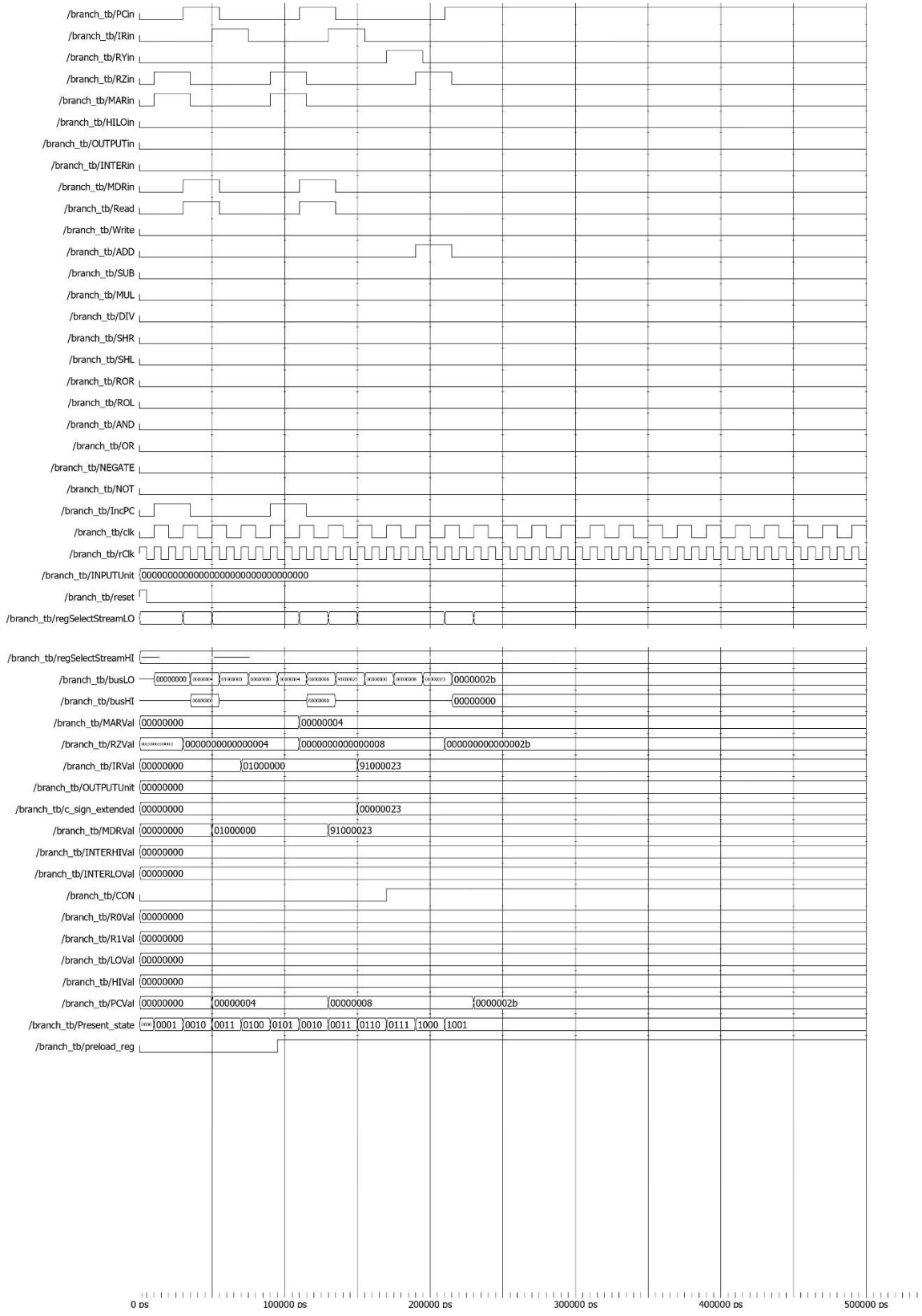


Branch Testbench

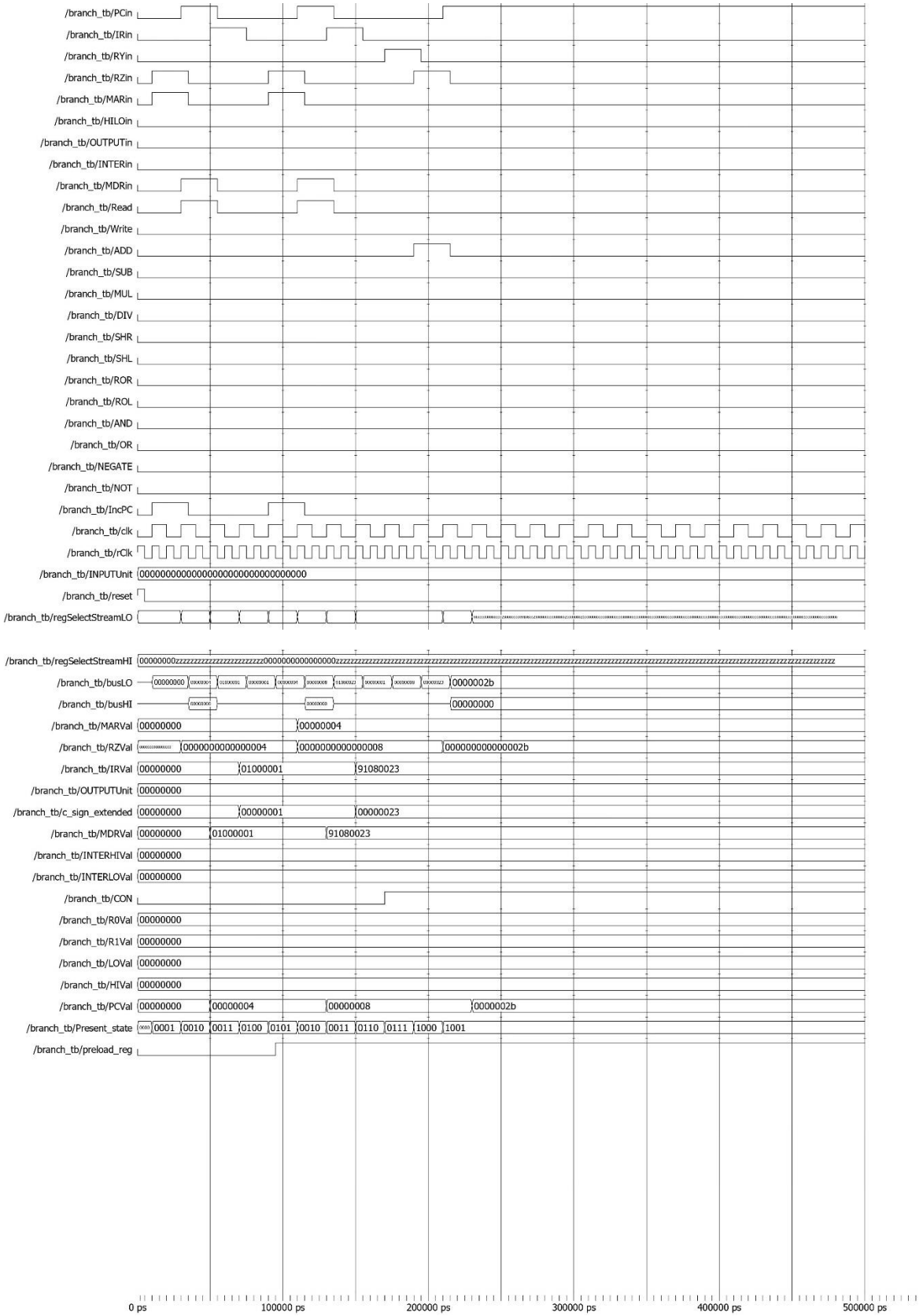
```
// branch tb

`timescale 1ns/10ps
module branch_tb;

	parameter BITS=32, REGISTERS=16, TOT_REGISTERS=REGISTERS+7, RAMSIZE=512;

	reg INPUTout, MDRout, HILOout, RZout, PCout, Cout, INTERout, BAout, Gra, Grb, Grc, Rout, Rin; // add any other signals to see in your simulation
	reg CONin, PCin, IRin, RYin, RZin, MARin, HILOin, OUTPUTin, INTERin, MDRin;
	reg Read, Write, ADD, SUB, MUL, DIV, SHR, SHL, ROR, ROL, AND, OR, NEGATE, NOT, IncPC;
	reg clk, rClk;
	reg [BITS-1:0] INPUTUnit;
	reg reset;
	wire [(BITS*TOT_REGISTERS)-1:0] regSelectStreamLO, regSelectStreamHI;
	wire [BITS-1:0] busLO, busHI;
	wire [BITS-1:0] MARVal;
	wire [(BITS*2)-1:0] RZVal;
	wire [BITS-1:0] IRVal;
	wire [BITS-1:0] OUTPUTUnit;
	wire [BITS-1:0] c_sign_extended;
	wire [BITS-1:0] MDRVal;
	wire [BITS-1:0] INTERHIVal, INTERLOVal;
	wire CON;
	wire [BITS-1:0] R0Val, R1Val, LOVal, HIVal, PCVal;
	assign R0Val = regSelectStreamLO[(1*BITS)-1:BITS*0];
	assign R1Val = regSelectStreamLO[(2*BITS)-1:BITS*1];
	assign PCVal = regSelectStreamLO[(17*BITS)-1:BITS*16];

	parameter Default = 4'b0000, T0 = 4'b0001, T1 = 4'b0010, T2 = 4'b0011,
			T3 = 4'b0100, T4 = 4'b0101, T5 = 4'b0110, T6 = 4'b0111, T7 = 4'b1000, T8 = 4'b1001;

	reg [3:0] Present_state = Default;
	reg preload_reg = 1'b0;

	datapath #(.BITS(BITS), .REGISTERS(REGISTERS), .RAMSIZE(RAMSIZE)) DUT(
			reset, clk, rClk,
			CONin, PCin, IRin, RYin, RZin, MARin, HILOin, OUTPUTin, INTERin, MDRin, Read, Write, INPUTout, MDRout, HILOout, RZout, PCout, Cout, INTERout,
			BAout, Gra, Grb, Grc, Rout, Rin,
			ADD, SUB, MUL, DIV, SHR, SHL, ROR, ROL, AND, OR, NEGATE, NOT, IncPC,
			INPUTUnit,
			regSelectStreamLO, regSelectStreamHI,
			busLO, busHI,
			MARVal,
			RZVal,
			IRVal,
			LOVal, HIVal,
			OUTPUTUnit,
			c_sign_extended,
			MDRVal,
			INTERHIVal, INTERLOVal,
			CON);

	initial begin
		clk = 0;
		forever #10 clk = ~clk;
	end

	initial begin
		rClk = 1;
		forever #5 rClk = ~rClk;
	end

	always @(posedge clk) // finite state machine; if clock rising-edge
	begin
		case (Present_state)
			Default : Present_state = T0;
			T0 : Present_state = T1;
			T1 : Present_state = T2;
			T2 : begin
				if(preload_reg == 1)
					Present_state = T5;
				else
					Present_state = T3;
			end
			T3 : Present_state = T4;
			T4 : Present_state = T1;
			T5 : Present_state = T6;
			T6 : Present_state = T7;
			T7 : Present_state = T8;
		endcase
	end

	always @(Present_state) // do the required job in each state
	begin
		case (Present_state) // assert the required signals in each clock cycle
			Default: begin
				reset <= 1;
				CONin <= 0; PCin <= 0; IRin <= 0; RYin <= 0; RZin <= 0; MARin <= 0; HILOin <= 0; MDRin <= 0; OUTPUTin <= 0; INTERin <= 0;
				Read <= 0; Write <= 0;
				INPUTout <= 0; MDRout <= 0; HILOout <= 0; RZout <= 0; PCout <= 0; Cout <= 0; INTERout <= 0;
				BAout <= 0; Gra <= 0; Grb <= 0; Grc <= 0; Rout <= 0; Rin <= 0;
				ADD <= 0; SUB <= 0; MUL <= 0; DIV <= 0; SHR <= 0; SHL <= 0; ROR <= 0; ROL <= 0; AND <= 0; OR <= 0; NEGATE <= 0; NOT <= 0; IncPC <= 0;
				INPUTUnit <= {BITS{1'b0}};
				#5 reset <= 0;
			end
			T0: begin
				MARin <= 1; IncPC <= 1; RZin <= 1; PCout <= 1;
			end
			T1: begin
				Read <= 1;
				PCin <= 1; MDRin <= 1;
				#5 RZout <= 1; PCout <= 0; MARin <= 0; IncPC <= 0; RZin <= 0;
			end
			T2: begin
				IRin <= 1;
				#5 PCin <= 0; RZout <= 0; MDRout <= 1; MDRin <= 0; Read <= 0;
			end
			T3: begin
				Gra <= 1; Rin <= 1;
				#5 MDRout <= 0; IRin <= 0; Cout <= 1;
			end
			T4: begin
				MARin <= 1; IncPC <= 1; RZin <= 1;
				#5 Gra <= 0; Rin <= 0; Cout <= 0; PCout <= 1; preload_reg <= 1;
			end
			T5: begin
				Gra <= 1;
				#5 MDRout <= 0; IRin <= 0; Rout <= 1;
			end
			T6: begin
				RYin <= 1; CONin <= 1;
				#5 Gra <= 0; CONin <= 0; Rout <= 0; PCout <= 1;
			end
			T7: begin
				ADD <= 1; RZin <= 1;
				#5 RYin <= 0; PCout <= 0; Cout <= 1;
			end
			T8: begin
				PCin <= CON;
				#5 Cout <= 0; ADD <= 0; RZin <= 0; RZout <= 1;
			end
		endcase
	end
endmodule
```

Branch Waveform

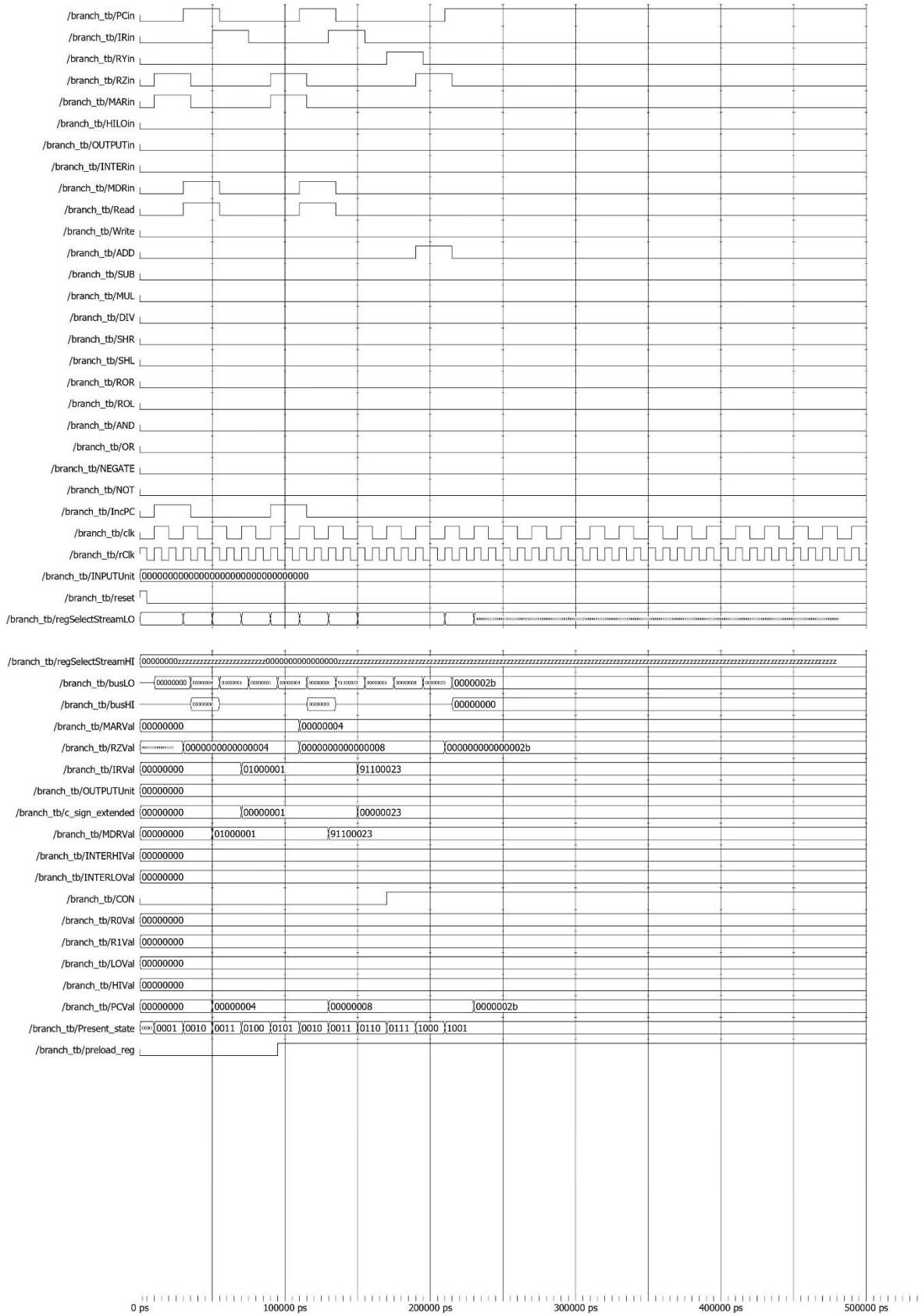
Branch Case 1



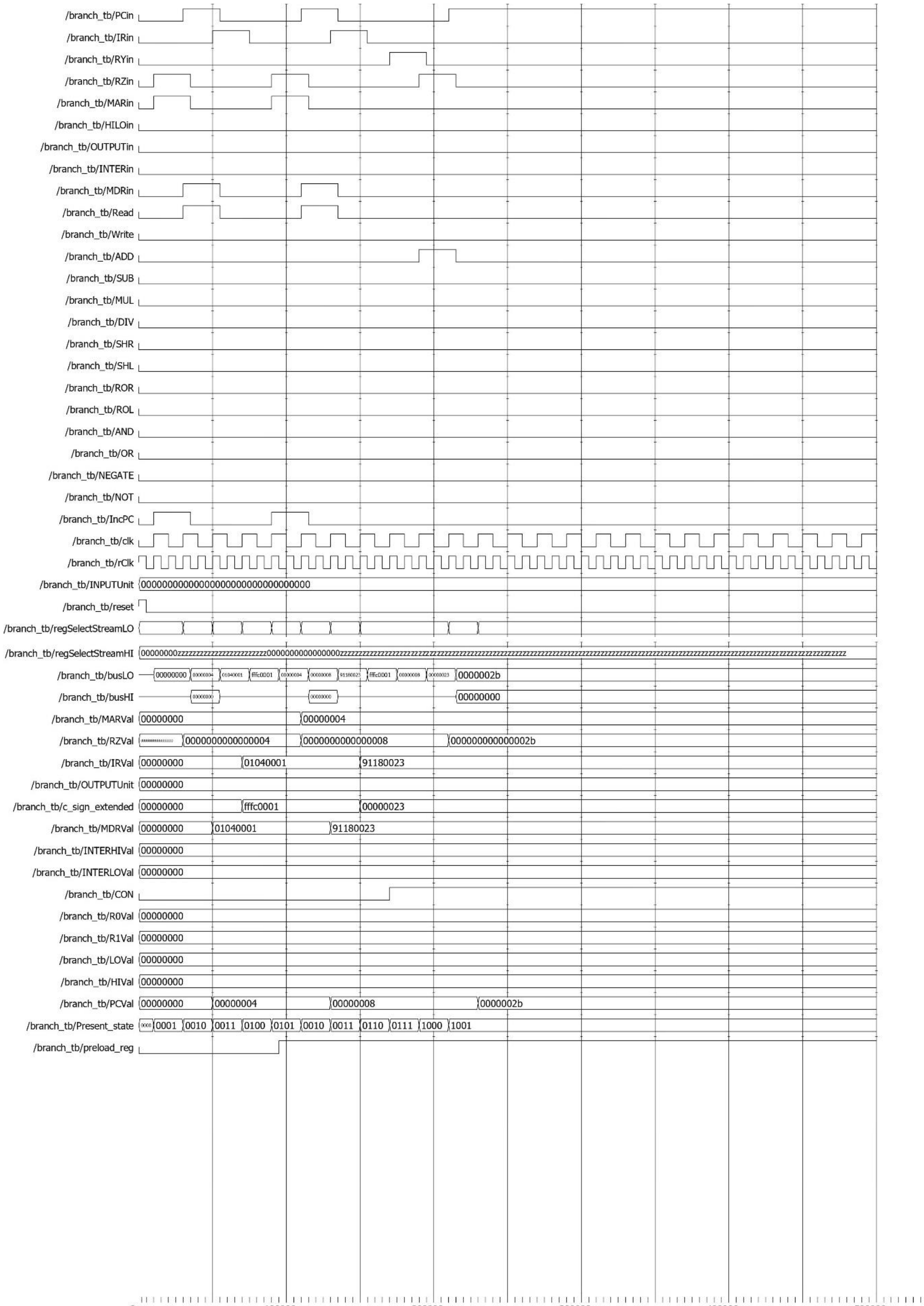
Branch Case 2



Branch Case 3



Branch Case 4



JR Testbench

```
// jump tb

`timescale 1ns/10ps
module jump_tb;

	parameter BITS=32, REGISTERS=16, TOT_REGISTERS=REGISTERS+7, RAMSIZE=512;

	reg INPUTout, MDRout, HILOout, RZout, PCout, Cout, INTERout, BAout, Gra, Grb, Grc, Rout, Rin; // add any other signals to see in your simulation
	reg CONin, PCin, IRin, RYin, RZin, MARin, HILOin, OUTPUTin, INTERin, MDRin;
	reg Read, Write, ADD, SUB, MUL, DIV, SHR, SHL, ROR, ROL, AND, OR, NEGATE, NOT, IncPC;
	reg clk, rClk;
	reg [BITS-1:0] INPUTUnit;
	reg reset;
	wire [(BITS*TOT_REGISTERS)-1:0] regSelectStreamLO, regSelectStreamHI;
	wire [BITS-1:0] busLO, busHI;
	wire [BITS-1:0] MARVal;
	wire [(BITS*2)-1:0] RZVal;
	wire [BITS-1:0] IRVal;
	wire [BITS-1:0] OUTPUTUnit;
	wire [BITS-1:0] c_sign_extended;
	wire [BITS-1:0] MDRVal;
	wire [BITS-1:0] INTERHIVal, INTERLOVal;
	wire CON;
	wire [BITS-1:0] R0Val, R1Val, R2Val, PCVal, LOVal, HIVal;
	assign R0Val = regSelectStreamLO[(1*BITS)-1:BITS*0];
	assign R1Val = regSelectStreamLO[(2*BITS)-1:BITS*1];
	assign R2Val = regSelectStreamLO[(3*BITS)-1:BITS*2];
	assign PCVal = regSelectStreamLO[(17*BITS)-1:BITS*16];

	parameter Default = 4'b0000, T0 = 4'b0001, T1 = 4'b0010, T2 = 4'b0011,
			T3 = 4'b0100, T4 = 4'b0101, T5 = 4'b0110;

	reg [3:0] Present_state = Default;
	reg preload_reg = 1'b0;

	datapath #(.BITS(BITS), .REGISTERS(REGISTERS), .RAMSIZE(RAMSIZE)) DUT(
			reset, clk, rClk,
			CONin, PCin, IRin, RYin, RZin, MARin, HILOin, OUTPUTin, INTERin, MDRin, Read, Write, INPUTout, MDRout, HILOout, RZout, PCout, Cout, INTERout,
			BAout, Gra, Grb, Grc, Rout, Rin,
			ADD, SUB, MUL, DIV, SHR, SHL, ROR, ROL, AND, OR, NEGATE, NOT, IncPC,
			INPUTUnit,
			regSelectStreamLO, regSelectStreamHI,
			busLO, busHI,
			MARVal,
			RZVal,
			IRVal,
			LOVal, HIVal,
			OUTPUTUnit,
			c_sign_extended,
			MDRVal,
			INTERHIVal, INTERLOVal,
			CON);

	initial begin
		clk = 0;
		forever #10 clk = ~clk;
	end

	initial begin
		rClk = 1;
		forever #5 rClk = ~rClk;
	end

	always @(posedge clk) // finite state machine; if clock rising-edge
	begin
		case (Present_state)
			Default : Present_state = T0;
			T0 : Present_state = T1;
			T1 : Present_state = T2;
			T2 : begin
				if(preload_reg == 1)
					Present_state = T5;
				else
					Present_state = T3;
			end
			T3 : Present_state = T4;
			T4 : Present_state = T1;
		endcase
	end

	always @(Present_state) // do the required job in each state
	begin
		case (Present_state) // assert the required signals in each clock cycle
			Default: begin
				reset <= 1;
				CONin <= 0; PCin <= 0; IRin <= 0; RYin <= 0; RZin <= 0; MARin <= 0; HILOin <= 0; MDRin <= 0; OUTPUTin <= 0; INTERin <= 0;
				Read <= 0; Write <= 0;
				INPUTout <= 0; MDRout <= 0; HILOout <= 0; RZout <= 0; PCout <= 0; Cout <= 0; INTERout <= 0;
				BAout <= 0; Gra <= 0; Grb <= 0; Grc <= 0; Rout <= 0; Rin <= 0;
				ADD <= 0; SUB <= 0; MUL <= 0; DIV <= 0; SHR <= 0; SHL <= 0; ROR <= 0; ROL <= 0; AND <= 0; OR <= 0; NEGATE <= 0; NOT <= 0; IncPC <= 0;
				INPUTUnit <= {BITS{1'b0}};
				#5 reset <= 0;
			end
			T0: begin
				MARin <= 1; IncPC <= 1; RZin <= 1; PCout <= 1;
			end
			T1: begin
				Read <= 1;
				PCin <= 1; MDRin <= 1;
				#5 RZout <= 1; PCout <= 0; MARin <= 0; IncPC <= 0; RZin <= 0;
			end
			T2: begin
				IRin <= 1;
				#5 PCin <= 0; RZout <= 0; MDRout <= 1; MDRin <= 0; Read <= 0;
			end
			T3: begin
				Gra <= 1; Rin <= 1;
				#5 MDRout <= 0; IRin <= 0;Cout <= 1;
			end
			T4: begin
				MARin <= 1; IncPC <= 1; RZin <= 1;
				#5 Gra <= 0; Rin <= 0; Cout <= 0; PCout <= 1; preload_reg <= 1;
			end
			T5: begin
				Gra <= 1; PCin <= 1;
				#5 MDRout <= 0; IRin <= 0; Rout <= 1;
			end
		endcase
	end
endmodule
```

JR Waveform


/jump_tb/IRin
/jump_tb/RYin
/jump_tb/RZin
/jump_tb/MARin
/jump_tb/HILOin
/jump_tb/OUTPUTin
/jump_tb/INTERin
/jump_tb/MDRin
/jump_tb/Read
/jump_tb/Write
/jump_tb/ADD
/jump_tb/SUB
/jump_tb/MUL
/jump_tb/DIV
/jump_tb/SHR
/jump_tb/SHL
/jump_tb/ROR
/jump_tb/ROL
/jump_tb/AND
/jump_tb/OR
/jump_tb/NEGATE
/jump_tb/NOT
/jump_tb/IncPC
/jump_tb/clk
/jump_tb/rClk
/jump_tb/INPUTUnit
/jump_tb/reset
/jump_tb/regSelectStreamLO
/jump_tb/regSelectStreamHI
/jump_tb/busLO
/jump_tb/busHI
/jump_tb/MARVal
/jump_tb/RZVal
/jump_tb/IRVal
/jump_tb/OUTPUTUnit
/jump_tb/c_sign_extended
/jump_tb/MDRVal
/jump_tb/INTERHIVal
/jump_tb/INTERLOVal
/jump_tb/CON
/jump_tb/R0Val
/jump_tb/R1Val
/jump_tb/R2Val
/jump_tb/PCVal
/jump_tb/LOVal
/jump_tb/HIVal
/jump_tb/Present_state
/jump_tb/preload_reg
0000000000000000000000000001100
0000000000000000000000000000100
0000000000000000000000000000000000000000000000000000000000001000
10011000100000000000000000000
0000000000000000000000000000000
0001
0010
0011
0100
0101
0010
0011
0110
(+20
100000 ps
200000 ps
300000 ps
400000 ps
500000 ps

JAL Testbench

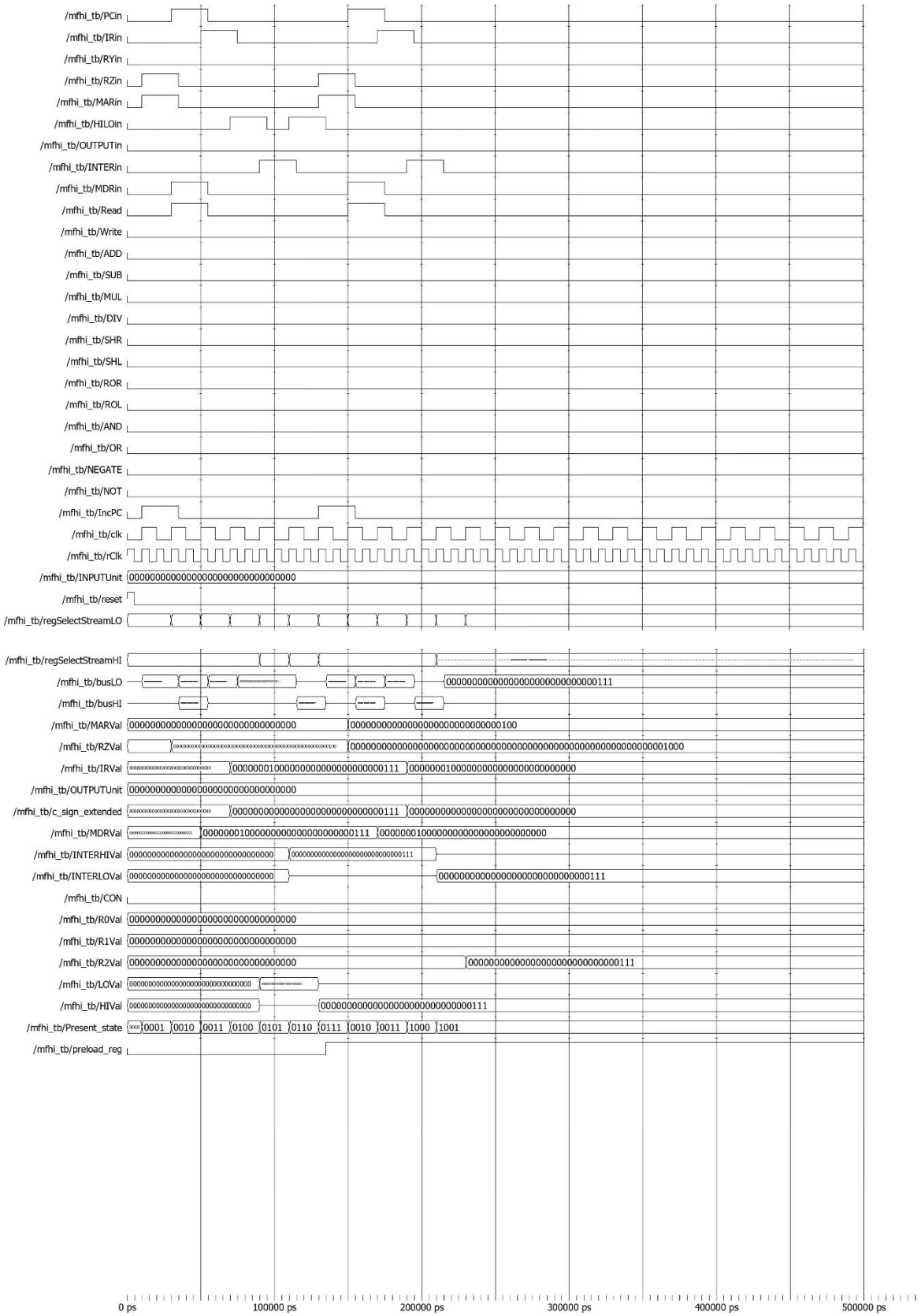
```
// jump_al_tb

`timescale 1ns/10ps
module jump_al_tb;

        parameter BITS=32, REGISTERS=16, TOT_REGISTERS=REGISTERS+7, RAMSIZE=512;

        reg INPUTout, MDRout, HILOout, RZout, PCout, Cout, INTERout, BAout, Gra, Grb, Grc, Rout, Rin; // add any other signals to see in your simulation
        reg CONin, PCin, IRin, RYin, RZin, MARin, HILOin, OUTPUTin, INTERin, MDRin;
        reg Read, Write, ADD, SUB, MUL, DIV, SHR, SHL, ROR, ROL, AND, OR, NEGATE, NOT, IncPC;
        reg clk, rClk;
        reg [BITS-1:0] INPUTUnit;
        reg reset;
        wire [(BITS*TOT_REGISTERS)-1:0] regSelectStreamLO, regSelectStreamHI;
        wire [BITS-1:0] busLO, busHI;
        wire [BITS-1:0] MARVal;
        wire [(BITS*2)-1:0] RZVal;
        wire [BITS-1:0] IRVal;
        wire [BITS-1:0] OUTPUTUnit;
        wire [BITS-1:0] c_sign_extended;
        wire [BITS-1:0] MDRVal;
        wire [BITS-1:0] INTERHIVal, INTERLOVal;
        wire CON;
        wire [BITS-1:0] R0Val, R1Val, R15Val, PCVal, LOVal, HIVal;
        assign R0Val = regSelectStreamLO[(1*BITS)-1:BITS*0];
        assign R1Val = regSelectStreamLO[(2*BITS)-1:BITS*1];
        assign R15Val = regSelectStreamLO[(16*BITS)-1:BITS*15];
        assign PCVal = regSelectStreamLO[(17*BITS)-1:BITS*16];

        parameter Default = 4'b0000, T0 = 4'b0001, T1 = 4'b0010, T2 = 4'b0011,
                  T3 = 4'b0100, T4 = 4'b0101, T5 = 4'b0110, T6 = 4'b0111, T7 = 4'b1000, T8 = 4'b1001;

        reg [3:0] Present_state = Default;
        reg preload_reg = 1'b0;

        datapath #(.BITS(BITS), .REGISTERS(REGISTERS), .RAMSIZE(RAMSIZE)) DUT(
                  reset, clk, rClk,
                  CONin, PCin, IRin, RYin, RZin, MARin, HILOin, OUTPUTin, INTERin, MDRin, Read, Write, INPUTout, MDRout, HILOout, RZout, PCout, Cout, INTERout,
                  BAout, Gra, Grb, Grc, Rout, Rin,
                  ADD, SUB, MUL, DIV, SHR, SHL, ROR, ROL, AND, OR, NEGATE, NOT, IncPC,
                  INPUTUnit,
                  regSelectStreamLO, regSelectStreamHI,
                  busLO, busHI,
                  MARVal,
                  RZVal,
                  IRVal,
                  LOVal, HIVal,
                  OUTPUTUnit,
                  c_sign_extended,
                  MDRVal,
                  INTERHIVal, INTERLOVal,
                  CON);

        initial begin
                clk = 0;
                forever #10 clk = ~clk;
        end

        initial begin
                rClk = 1;
                forever #5 rClk = ~rClk;
        end

        always @(posedge clk) // finite state machine; if clock rising-edge
        begin
                case (Present_state)
                        Default : Present_state = T0;
                        T0 : Present_state = T1;
                        T1 : Present_state = T2;
                        T2 : begin
                                if(preload_reg == 1)
                                        Present_state = T5;
                                else
                                        Present_state = T3;
                        end
                        T3 : Present_state = T4;
                        T4 : Present_state = T1;
                        T5 : Present_state = T6;
                        T6 : Present_state = T7;
                        T7 : Present_state = T8;
                endcase
        end

        always @(Present_state) // do the required job in each state
        begin
                case (Present_state) // assert the required signals in each clock cycle
                        Default: begin
                                reset <= 1;
                                CONin <= 0; PCin <= 0; IRin <= 0; RYin <= 0; RZin <= 0; MARin <= 0; HILOin <= 0; MDRin <= 0; OUTPUTin <= 0; INTERin <= 0;
                                Read <= 0; Write <= 0;
                                INPUTout <= 0; MDRout <= 0; HILOout <= 0; RZout <= 0; PCout <= 0; Cout <= 0; INTERout <= 0;
                                BAout <= 0; Gra <= 0; Grb <= 0; Grc <= 0; Rout <= 0; Rin <= 0;
                                ADD <= 0; SUB <= 0; MUL <= 0; DIV <= 0; SHR <= 0; SHL <= 0; ROR <= 0; ROL <= 0; AND <= 0; OR <= 0; NEGATE <= 0; NOT <= 0; IncPC <= 0;
                                INPUTUnit <= {BITS{1'b0}};
                                #5 reset <= 0;
                        end
                        T0: begin
                                MARin <= 1; IncPC <= 1; RZin <= 1; PCout <= 1;
                        end
                        T1: begin
                                Read <= 1;
                                PCin <= 1; MDRin <= 1;
                                #5 RZout <= 1; PCout <= 0; MARin <= 0; IncPC <= 0; RZin <= 0;
                        end
                        T2: begin
                                        IRin <= 1;
                                        #5 PCin <= 0; RZout <= 0; MDRout <= 1; MDRin <= 0; Read <= 0;
                        end
                        T3: begin
                                Gra <= 1; Rin <= 1;
                                #5 MDRout <= 0; IRin <= 0;Cout <= 1;
                        end
                        T4: begin
                                MARin <= 1; IncPC <= 1; RZin <= 1;
                                #5 Gra <= 0; Rin <= 0; Cout <= 0; PCout <= 1; preload_reg <= 1;
                        end
                        T5: begin
                                RYin <= 1;
                                #5 MDRout <= 0; IRin <= 0; PCout <= 1;
                        end
                        T6: begin
                                ADD <= 1; RZin <= 1;
                                #5 RYin <= 0; PCout <= 0; Cout <= 1;
                        end
                        T7: begin
                                Grb <= 1; Rin <= 1;
                                #5 ADD <= 0; RZin <= 0; Cout <= 0; RZout <= 1;
                        end
                        T8: begin
                                Gra <= 1; PCin <= 1;
                                #5 Grb <= 0; Rin <= 0; RZout <= 0; Rout <= 1;
                        end
                endcase
        end
endmodule
```

JAL Waveform

MFHI Testbench

```
// mfhi tb

`timescale 1ns/10ps
module mfhi_tb;

	parameter BITS=32, REGISTERS=16, TOT_REGISTERS=REGISTERS+7, RAMSIZE=512;

	reg INPUTout, MDRout, HILOout, RZout, PCout, Cout, INTERout, BAout, Gra, Grb, Grc, Rout, Rin; // add any other signals to see in your simulation
	reg CONin, PCin, IRin, RYin, RZin, MARin, HILOin, OUTPUTin, INTERin, MDRin;
	reg Read, Write, ADD, SUB, MUL, DIV, SHR, SHL, ROR, ROL, AND, OR, NEGATE, NOT, IncPC;
	reg clk, rClk;
	reg [BITS-1:0] INPUTUnit;
	reg reset;
	wire [(BITS*TOT_REGISTERS)-1:0] regSelectStreamLO, regSelectStreamHI;
	wire [BITS-1:0] busLO, busHI;
	wire [BITS-1:0] MARVal;
	wire [(BITS*2)-1:0] RZVal;
	wire [BITS-1:0] IRVal;
	wire [BITS-1:0] OUTPUTUnit;
	wire [BITS-1:0] c_sign_extended;
	wire [BITS-1:0] MDRVal;
	wire [BITS-1:0] INTERHIVal, INTERLOVal;
	wire CON;
	wire [BITS-1:0] R0Val, R1Val, R2Val, LOVal, HIVal;
	assign R0Val = regSelectStreamLO[(1*BITS)-1:BITS*0];
	assign R1Val = regSelectStreamLO[(2*BITS)-1:BITS*1];
	assign R2Val = regSelectStreamLO[(3*BITS)-1:BITS*2];

	parameter Default = 4'b0000, T0 = 4'b0001, T1 = 4'b0010, T2 = 4'b0011,
			T3 = 4'b0100, T4 = 4'b0101, T5 = 4'b0110, T6 = 4'b0111, T7 = 4'b1000, T8 = 4'b1001;

	reg [3:0] Present_state = Default;
	reg preload_reg = 1'b0;

	datapath #(.BITS(BITS), .REGISTERS(REGISTERS), .RAMSIZE(RAMSIZE)) DUT(
			reset, clk, rClk,
			CONin, PCin, IRin, RYin, RZin, MARin, HILOin, OUTPUTin, INTERin, MDRin, Read, Write, INPUTout, MDRout, HILOout, RZout, PCout, Cout, INTERout,
			BAout, Gra, Grb, Grc, Rout, Rin,
			ADD, SUB, MUL, DIV, SHR, SHL, ROR, ROL, AND, OR, NEGATE, NOT, IncPC,
			INPUTUnit,
			regSelectStreamLO, regSelectStreamHI,
			busLO, busHI,
			MARVal,
			RZVal,
			IRVal,
			LOVal, HIVal,
			OUTPUTUnit,
			c_sign_extended,
			MDRVal,
			INTERHIVal, INTERLOVal,
			CON);

	initial begin
		clk = 0;
		forever #10 clk = ~clk;
	end

	initial begin
		rClk = 1;
		forever #5 rClk = ~rClk;
	end

	always @(posedge clk) // finite state machine; if clock rising-edge
	begin
		case (Present_state)
			Default : Present_state = T0;
			T0 : Present_state = T1;
			T1 : Present_state = T2;
			T2 : begin
				if(preload_reg == 1)
					Present_state = T7;
				else
					Present_state = T3;
			end
			T3 : Present_state = T4;
			T4 : Present_state = T5;
			T5 : Present_state = T6;
			T6 : Present_state = T1;
			T7 : Present_state = T8;
		endcase
	end

	always @(Present_state) // do the required job in each state
	begin
		case (Present_state) // assert the required signals in each clock cycle
			Default: begin
				reset <= 1;
				CONin <= 0; PCin <= 0; IRin <= 0; RYin <= 0; RZin <= 0; MARin <= 0; HILOin <= 0; MDRin <= 0; OUTPUTin <= 0; INTERin <= 0;
				Read <= 0; Write <= 0;
				INPUTout <= 0; MDRout <= 0; HILOout <= 0; RZout <= 0; PCout <= 0; Cout <= 0; INTERout <= 0;
				BAout <= 0; Gra <= 0; Grb <= 0; Grc <= 0; Rout <= 0; Rin <= 0;
				ADD <= 0; SUB <= 0; MUL <= 0; DIV <= 0; SHR <= 0; SHL <= 0; ROR <= 0; ROL <= 0; AND <= 0; OR <= 0; NEGATE <= 0; NOT <= 0; IncPC <= 0;
				INPUTUnit <= {BITS{1'b0}};
				#5 reset <= 0;
			end
			T0: begin
				MARin <= 1; IncPC <= 1; RZin <= 1; PCout <= 1;
			end
			T1: begin
				Read <= 1; PCin <= 1; MDRin <= 1;
				#5 RZout <= 1; PCout <= 0; MARin <= 0; IncPC <= 0; RZin <= 0;
			end
			T2: begin
				IRin <= 1;
				#5 PCin <= 0; RZout <= 0; MDRout <= 1; MDRin <= 0; Read <= 0;
			end
			T3: begin
				HILOin <= 1;
				#5 MDRout <= 0; IRin <= 0; Cout <= 1;
			end
			T4: begin
				INTERin <= 1;
				#5 HILOin <= 0; Cout <= 0; HILOout <= 1;
			end
			T5: begin
				HILOin <= 1;
				#5 INTERin <= 0; INTERout <= 1; HILOout <= 0;
			end
			T6: begin
				MARin <= 1; IncPC <= 1; RZin <= 1;
				#5 HILOin <= 0; PCout <= 1; preload_reg <= 1; INTERout <= 0;
			end
			T7: begin
				INTERin <= 1;
				#5 MDRout <= 0; IRin <= 0; HILOout <= 1;  PCout <= 0; MARin <= 0;
			end
			T8: begin
				Gra <= 1; Rin <= 1;
				#5 INTERin <= 0; HILOout <= 0; INTERout <= 1;
			end
		endcase
	end
endmodule
```

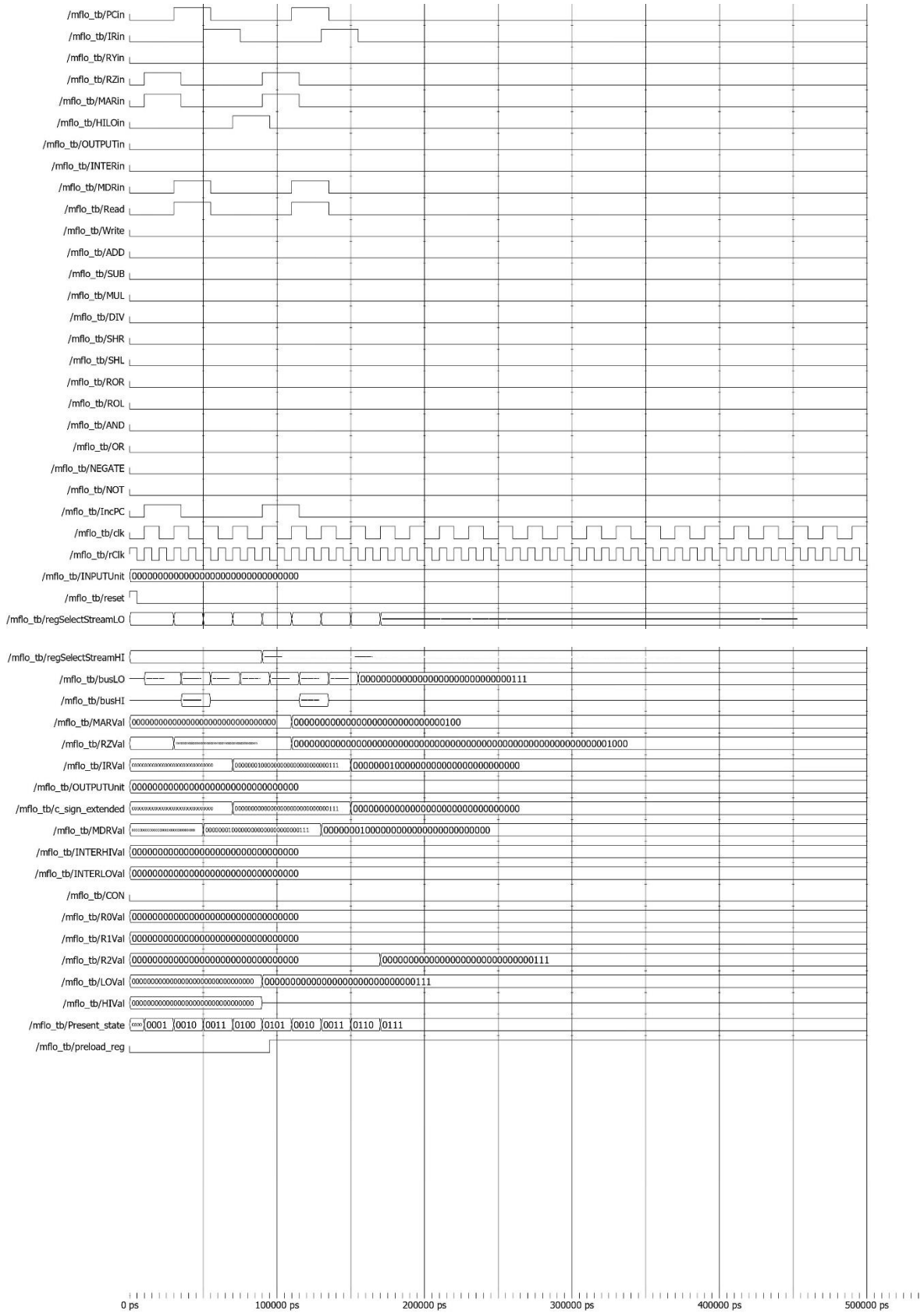
MFHI Waveform



MFLO Testbench

```
// mflo tb

`timescale 1ns/10ps
module mflo_tb;

	parameter BITS=32, REGISTERS=16, TOT_REGISTERS=REGISTERS+7, RAMSIZE=512;

	reg INPUTout, MDRout, HILOout, RZout, PCout, Cout, INTERout, BAout, Gra, Grb, Grc, Rout, Rin; // add any other signals to see in your simulation
	reg CONin, PCin, IRin, RYin, RZin, MARin, HILOin, OUTPUTin, INTERin, MDRin;
	reg Read, Write, ADD, SUB, MUL, DIV, SHR, SHL, ROR, ROL, AND, OR, NEGATE, NOT, IncPC;
	reg clk, rClk;
	reg [BITS-1:0] INPUTUnit;
	reg reset;
	wire [(BITS*TOT_REGISTERS)-1:0] regSelectStreamLO, regSelectStreamHI;
	wire [BITS-1:0] busLO, busHI;
	wire [BITS-1:0] MARVal;
	wire [(BITS*2)-1:0] RZVal;
	wire [BITS-1:0] IRVal;
	wire [BITS-1:0] OUTPUTUnit;
	wire [BITS-1:0] c_sign_extended;
	wire [BITS-1:0] MDRVal;
	wire [BITS-1:0] INTERHIVal, INTERLOVal;
	wire CON;
	wire [BITS-1:0] R0Val, R1Val, R2Val, LOVal, HIVal;
	assign R0Val = regSelectStreamLO[(1*BITS)-1:BITS*0];
	assign R1Val = regSelectStreamLO[(2*BITS)-1:BITS*1];
	assign R2Val = regSelectStreamLO[(3*BITS)-1:BITS*2];

	parameter Default = 4'b0000, T0 = 4'b0001, T1 = 4'b0010, T2 = 4'b0011,
			T3 = 4'b0100, T4 = 4'b0101, T5 = 4'b0110, T6 = 4'b0111;

	reg [3:0] Present_state = Default;
	reg preload_reg = 1'b0;

	datapath #(.BITS(BITS), .REGISTERS(REGISTERS), .RAMSIZE(RAMSIZE)) DUT(
			reset, clk, rClk,
			CONin, PCin, IRin, RYin, RZin, MARin, HILOin, OUTPUTin, INTERin, MDRin, Read, Write, INPUTout, MDRout, HILOout, RZout, PCout, Cout, INTERout,
			BAout, Gra, Grb, Grc, Rout, Rin,
			ADD, SUB, MUL, DIV, SHR, SHL, ROR, ROL, AND, OR, NEGATE, NOT, IncPC,
			INPUTUnit,
			regSelectStreamLO, regSelectStreamHI,
			busLO, busHI,
			MARVal,
			RZVal,
			IRVal,
			LOVal, HIVal,
			OUTPUTUnit,
			c_sign_extended,
			MDRVal,
			INTERHIVal, INTERLOVal,
			CON);

	initial begin
		clk = 0;
		forever #10 clk = ~clk;
	end

	initial begin
		rClk = 1;
		forever #5 rClk = ~rClk;
	end

	always @(posedge clk) // finite state machine; if clock rising-edge
	begin
		case (Present_state)
			Default : Present_state = T0;
			T0 : Present_state = T1;
			T1 : Present_state = T2;
			T2 : begin
				if(preload_reg == 1)
					Present_state = T5;
				else
					Present_state = T3;
			end
			T3 : Present_state = T4;
			T4 : Present_state = T1;
			T5 : Present_state = T6;
		endcase
	end

	always @(Present_state) // do the required job in each state
	begin
		case (Present_state) // assert the required signals in each clock cycle
			Default: begin
				reset <= 1;
				CONin <= 0; PCin <= 0; IRin <= 0; RYin <= 0; RZin <= 0; MARin <= 0; HILOin <= 0; MDRin <= 0; OUTPUTin <= 0; INTERin <= 0;
				Read <= 0; Write <= 0;
				INPUTout <= 0; MDRout <= 0; HILOout <= 0; RZout <= 0; PCout <= 0; Cout <= 0; INTERout <= 0;
				BAout <= 0; Gra <= 0; Grb <= 0; Grc <= 0; Rout <= 0; Rin <= 0;
				ADD <= 0; SUB <= 0; MUL <= 0; DIV <= 0; SHR <= 0; SHL <= 0; ROR <= 0; ROL <= 0; AND <= 0; OR <= 0; NEGATE <= 0; NOT <= 0; IncPC <= 0;
				INPUTUnit <= {BITS{1'b0}};
				#5 reset <= 0;
			end
			T0: begin
				MARin <= 1; IncPC <= 1; RZin <= 1; PCout <= 1;
			end
			T1: begin
				Read <= 1;
				PCin <= 1; MDRin <= 1;
				#5 RZout <= 1; PCout <= 0; MARin <= 0; IncPC <= 0; RZin <= 0;
			end
			T2: begin
				IRin <= 1;
				#5 PCin <= 0; RZout <= 0; MDRout <= 1; MDRin <= 0; Read <= 0;
			end
			T3: begin
				HILOin <= 1;
				#5 MDRout <= 0; IRin <= 0; Cout <= 1;
			end
			T4: begin
				MARin <= 1; IncPC <= 1; RZin <= 1;
				#5 HILOin <= 0; Cout <= 0; PCout <= 1; preload_reg <= 1;
			end
			T5: begin
				Gra <= 1; Rin <= 1;
				#5 MDRout <= 0; IRin <= 0; HILOout <= 1;  PCout <= 0; MARin <= 0;
			end
		endcase
	end
endmodule
```

MFLO Waveform



Input Testbench

```
// input_tb

`timescale 1ns/10ps
module input_tb;

	parameter BITS=32, REGISTERS=16, TOT_REGISTERS=REGISTERS+7, RAMSIZE=512;

	reg INPUTout, MDRout, HILOout, RZout, PCout, Cout, INTERout, BAout, Gra, Grb, Grc, Rout, Rin; // add any other signals to see in your simulation
	reg CONin, PCin, IRin, RYin, RZin, MARin, HILOin, OUTPUTin, INTERin, MDRin;
	reg Read, Write, ADD, SUB, MUL, DIV, SHR, SHL, ROR, ROL, AND, OR, NEGATE, NOT, IncPC;
	reg clk, rClk;
	reg [BITS-1:0] INPUTUnit;
	reg reset;
	wire [(BITS*TOT_REGISTERS)-1:0] regSelectStreamLO, regSelectStreamHI;
	wire [BITS-1:0] busLO, busHI;
	wire [BITS-1:0] MARVal;
	wire [(BITS*2)-1:0] RZVal;
	wire [BITS-1:0] IRVal;
	wire [BITS-1:0] OUTPUTUnit;
	wire [BITS-1:0] c_sign_extended;
	wire [BITS-1:0] MDRVal;
	wire [BITS-1:0] INTERHIVal, INTERLOVal;
	wire CON;
	wire [BITS-1:0] R0Val, R1Val, R15Val, PCVal, LOVal, HIVal;
	assign R0Val = regSelectStreamLO[(1*BITS)-1:BITS*0];
	assign R1Val = regSelectStreamLO[(2*BITS)-1:BITS*1];
	assign R15Val = regSelectStreamLO[(16*BITS)-1:BITS*15];
	assign PCVal = regSelectStreamLO[(17*BITS)-1:BITS*16];
	parameter Default = 4'b0000, T0 = 4'b0001, T1 = 4'b0010, T2 = 4'b0011, T3 = 4'b0100, T4 = 4'b0101;
	reg [3:0] Present_state = Default;

	datapath #(.BITS(BITS), .REGISTERS(REGISTERS), .RAMSIZE(RAMSIZE)) DUT(
			reset, clk, rClk,
			CONin, PCin, IRin, RYin, RZin, MARin, HILOin, OUTPUTin, INTERin, MDRin, Read, Write, INPUTout, MDRout, HILOout, RZout, PCout, Cout, INTERout,
			BAout, Gra, Grb, Grc, Rout, Rin,
			ADD, SUB, MUL, DIV, SHR, SHL, ROR, ROL, AND, OR, NEGATE, NOT, IncPC,
			INPUTUnit,
			regSelectStreamLO, regSelectStreamHI,
			busLO, busHI,
			MARVal,
			RZVal,
			IRVal,
			LOVal, HIVal,
			OUTPUTUnit,
			c_sign_extended,
			MDRVal,
			INTERHIVal, INTERLOVal,
			CON);

	initial begin
		clk = 0;
		forever #10 clk = ~clk;
	end

	initial begin
		rClk = 1;
		forever #5 rClk = ~rClk;
	end

	always @(posedge clk) // finite state machine; if clock rising-edge
	begin
		case (Present_state)
			Default : Present_state = T0;
			T0 : Present_state = T1;
			T1 : Present_state = T2;
			T2 : Present_state = T3;
			T3 : Present_state = T4;
		endcase
	end

	always @(Present_state) // do the required job in each state
	begin
		case (Present_state) // assert the required signals in each clock cycle
			Default: begin
				reset <= 1;
				CONin <= 0; PCin <= 0; IRin <= 0; RYin <= 0; RZin <= 0; MARin <= 0; HILOin <= 0; MDRin <= 0; OUTPUTin <= 0; INTERin <= 0;
				Read <= 0; Write <= 0;
				INPUTout <= 0; MDRout <= 0; HILOout <= 0; RZout <= 0; PCout <= 0; Cout <= 0; INTERout <= 0;
				BAout <= 0; Gra <= 0; Grb <= 0; Grc <= 0; Rout <= 0; Rin <= 0;
				ADD <= 0; SUB <= 0; MUL <= 0; DIV <= 0; SHR <= 0; SHL <= 0; ROR <= 0; ROL <= 0; AND <= 0; OR <= 0; NEGATE <= 0; NOT <= 0; IncPC <= 0;
				INPUTUnit <= {BITS{1'b0}};
				#5 reset <= 0;
			end
			T0: begin
				MARin <= 1; IncPC <= 1; RZin <= 1; PCout <= 1;
			end
			T1: begin
				Read <= 1; PCin <= 1; MDRin <= 1;
				#5 RZout <= 1; PCout <= 0; MARin <= 0; IncPC <= 0; RZin <= 0;
			end
			T2: begin
				IRin <= 1;
				#5 PCin <= 0; RZout <= 0; MDRout <= 1; MDRin <= 0; Read <= 0;
			end
			T3: begin
				INPUTUnit <= 30;
				#5 MDRout <= 0; IRin <= 0;
			end
			T4: begin
				Gra <= 1; Rin <= 1;
				#5 INPUTout <= 1;
			end
		endcase
	end
endmodule
```

Input Waveform

Output Testbench

```
// output_tb

`timescale 1ns/10ps
module output_tb;

	parameter BITS=32, REGISTERS=16, TOT_REGISTERS=REGISTERS+7, RAMSIZE=512;

	reg INPUTout, MDRout, HILOout, RZout, PCout, Cout, INTERout, BAout, Gra, Grb, Grc, Rout, Rin; // add any other signals to see in your simulation
	reg CONin, PCin, IRin, RYin, RZin, MARin, HILOin, OUTPUTin, INTERin, MDRin;
	reg Read, Write, ADD, SUB, MUL, DIV, SHR, SHL, ROR, ROL, AND, OR, NEGATE, NOT, IncPC;
	reg clk, rClk;
	reg [BITS-1:0] INPUTUnit;
	reg reset;
	wire [(BITS*TOT_REGISTERS)-1:0] regSelectStreamLO, regSelectStreamHI;
	wire [BITS-1:0] busLO, busHI;
	wire [BITS-1:0] MARVal;
	wire [(BITS*2)-1:0] RZVal;
	wire [BITS-1:0] IRVal;
	wire [BITS-1:0] OUTPUTUnit;
	wire [BITS-1:0] c_sign_extended;
	wire [BITS-1:0] MDRVal;
	wire [BITS-1:0] INTERHIVal, INTERLOVal;
	wire CON;
	wire [BITS-1:0] R0Val, R1Val, R15Val, PCVal, LOVal, HIVal;
	assign R0Val = regSelectStreamLO[(1*BITS)-1:BITS*0];
	assign R1Val = regSelectStreamLO[(2*BITS)-1:BITS*1];
	assign R15Val = regSelectStreamLO[(16*BITS)-1:BITS*15];
	assign PCVal = regSelectStreamLO[(17*BITS)-1:BITS*16];
	parameter Default = 4'b0000, T0 = 4'b0001, T1 = 4'b0010, T2 = 4'b0011, T3 = 4'b0100, T4 = 4'b0101, T5 = 4'b0110;
	reg [3:0] Present_state = Default;
	reg preload_reg = 1'b0;

	datapath #(.BITS(BITS), .REGISTERS(REGISTERS), .RAMSIZE(RAMSIZE)) DUT(
			reset, clk, rClk,
			CONin, PCin, IRin, RYin, RZin, MARin, HILOin, OUTPUTin, INTERin, MDRin, Read, Write, INPUTout, MDRout, HILOout, RZout, PCout, Cout, INTERout,
			BAout, Gra, Grb, Grc, Rout, Rin,
			ADD, SUB, MUL, DIV, SHR, SHL, ROR, ROL, AND, OR, NEGATE, NOT, IncPC,
			INPUTUnit,
			regSelectStreamLO, regSelectStreamHI,
			busLO, busHI,
			MARVal,
			RZVal,
			IRVal,
			LOVal, HIVal,
			OUTPUTUnit,
			c_sign_extended,
			MDRVal,
			INTERHIVal, INTERLOVal,
			CON);


	initial begin
		clk = 0;
		forever #10 clk = ~clk;
	end

	initial begin
		rClk = 1;
		forever #5 rClk = ~rClk;
	end

	always @(posedge clk) // finite state machine; if clock rising-edge
	begin
		case (Present_state)
			Default : Present_state = T0;
			T0 : Present_state = T1;
			T1 : Present_state = T2;
			T2 : begin
				if(preload_reg == 1)
					Present_state = T5;
				else
					Present_state = T3;
			end
			T3 : Present_state = T4;
			T4 : Present_state = T1;
		endcase
	end

	always @(Present_state) // do the required job in each state
	begin
		case (Present_state) // assert the required signals in each clock cycle
			Default: begin
				reset <= 1;
				CONin <= 0; PCin <= 0; IRin <= 0; RYin <= 0; RZin <= 0; MARin <= 0; HILOin <= 0; MDRin <= 0; OUTPUTin <= 0; INTERin <= 0;
				Read <= 0; Write <= 0;
				INPUTout <= 0; MDRout <= 0; HILOout <= 0; RZout <= 0; PCout <= 0; Cout <= 0; INTERout <= 0;
				BAout <= 0; Gra <= 0; Grb <= 0; Grc <= 0; Rout <= 0; Rin <= 0;
				ADD <= 0; SUB <= 0; MUL <= 0; DIV <= 0; SHR <= 0; SHL <= 0; ROR <= 0; ROL <= 0; AND <= 0; OR <= 0; NEGATE <= 0; NOT <= 0; IncPC <= 0;
				INPUTUnit <= {BITS{1'b0}};
				#5 reset <= 0;
			end
			T0: begin
				MARin <= 1; IncPC <= 1; RZin <= 1; PCout <= 1;
			end
			T1: begin
				Read <= 1;
				PCin <= 1; MDRin <= 1;
				#5 RZout <= 1; PCout <= 0; MARin <= 0; IncPC <= 0; RZin <= 0;
			end
			T2: begin
				IRin <= 1;
				#5 PCin <= 0; RZout <= 0; MDRout <= 1; MDRin <= 0; Read <= 0;
			end
			T3: begin
				Gra <= 1; Rin <= 1;
				#5 MDRout <= 0; IRin <= 0;Cout <= 1;
			end
			T4: begin
				MARin <= 1; IncPC <= 1; RZin <= 1;
				#5 Gra <= 0; Rin <= 0; Cout <= 0; PCout <= 1; preload_reg <= 1;
			end
			T5: begin
				Gra <= 1; OUTPUTin <= 1;
				#5 MDRout <= 0; IRin <= 0; Rout <= 1;
			end
		endcase
	end
endmodule
```

Output Waveform