ELEC 374 CPU Design – Phase 2

David-Alexandre Edwards (20099471),
Elan Bibas (20099396),
Hannah Berthiaume (20114482)

Verilog Code

DataPath

```
//added write signal to DataPath
module DataPath(
    input PCout, Zlowout, MDRout, MARin, Zin, PCin, MDRin, IRin, Yin, Read, Write
,
     input [4:0] aluControl,
     input clock, clear,
    input Gra, Grb, Grc, Rin, Rout, BAout, Cout, ConIn, HIout, incPC
);


//registers run behavior every positive clock edge, bus will update BusMuxOUt whe
n a change in register output is detected
//define signals that will connect regiters to bus
wire [31:0] BusMuxOut;
wire [31:0] BusMuxInR0;
wire [31:0] BusMuxInR1;
wire [31:0] BusMuxInR2;
wire [31:0] BusMuxInR3;
wire [31:0] BusMuxInR4;
wire [31:0] BusMuxInR5;
wire [31:0] BusMuxInR6;
wire [31:0] BusMuxInR7;
wire [31:0] BusMuxInR8;
wire [31:0] BusMuxInR9;
wire [31:0] BusMuxInR10;
wire [31:0] BusMuxInR11;
wire [31:0] BusMuxInR12;
wire [31:0] BusMuxInR13;
wire [31:0] BusMuxInR14;
wire [31:0] BusMuxInR15;
wire [31:0] BusMuxInZHi;
wire [31:0] BusMuxInZLo;
wire [31:0] BusMuxInPC;
wire [31:0] BusMuxInMDR;
wire [31:0] BusMuxInRinP;
wire [31:0] BusMuxInCSign;
wire [31:0] BusMuxInRY;
wire [31:0] ALULoOut;
wire [31:0] ALUHiOut;
//phase 2
```

```
wire [8:0] Address;
wire [31:0] ramOut;
wire [31:0] BusMuxInIR;
wire r0out;
wire r1out;
wire R2out;
wire r3out;
wire R4out;
wire r5out;
wire r6out;
wire r7out;
wire r8out;
wire r9out;
wire r10out;
wire r11out;
wire r12out;
wire r13out;
wire r14out;
wire r15out;
wire [31:0] BusMuxInRoutP;
wire [31:0] BusMuxInRHi;
// phase 3
wire branch;

//instantiate all register
 register R1(clock, clear, R1in, BusMuxOut, BusMuxInR1);
 register R2(clock, clear, R2in, BusMuxOut, BusMuxInR2);
 register R3(clock, clear, R3in, BusMuxOut, BusMuxInR3);
 register R4(clock, clear, R4in, BusMuxOut, BusMuxInR4);
 register R5(clock, clear, R5in, BusMuxOut, BusMuxInR5);
 register R6(clock, clear, R6in, BusMuxOut, BusMuxInR6);
 register R7(clock, clear, R7in, BusMuxOut, BusMuxInR7);
 register R8(clock, clear, R8in, BusMuxOut, BusMuxInR8);
 register R9(clock, clear, R9in, BusMuxOut, BusMuxInR9);
 register R10(clock, clear, R10in, BusMuxOut, BusMuxInR10);
 register R11(clock, clear, R11in, BusMuxOut, BusMuxInR11);
 register R12(clock, clear, R12in, BusMuxOut, BusMuxInR12);
 register R13(clock, clear, R13in, BusMuxOut, BusMuxInR13);
 register R14(clock, clear, R14in, BusMuxOut, BusMuxInR14);
 register R15(clock, clear, R15in, BusMuxOut, BusMuxInR15);
 register RHi(clock, clear, RHiIn, BusMuxOut, BusMuxInRHi);
 register RLO(clock, clear, RLOIn, BusMuxOut, BusMuxInRLo);
 register RZHi(clock, clear, Zin, ALUHiOut, BusMuxInZHi);
 register RZLO(clock, clear, Zin, ALULoOut, BusMuxInZLo);
 //register PC(clock, clear, PCIn, BusMuxOut, BusMuxInPC);
```

```
 register RInP(clock, clear, RInPIn, BusMuxOut, BusMuxInRInP);
 //register RCSign(clock, clear, RCSignIn, BusMuxOut, BusMuxInCSign);
 register RoutP(clock, clear, RoutPIn, BusMuxOut, BusMuxInRoutP);
 register RY(clock, clear, Yin, BusMuxOut, BusMuxInRY);


//phase 2 Select and Encode
 SelectEncode selectencode(BusMuxInIR, Gra, Grb, Grc, Rin, Rout, BAout, BusMuxInC
Sign, R0in, R1in, R2in, R3in, R4in, R5in, R6in, R7in, R8in, R9in, R10in, R11in, R
12in, R13in, R14in, R15in, R0out, R1out, R2out, R3out, R4out, R5out, R6out, R7out
, R8out, R9out, R10out, R11out, R12out, R13out, R14out, R15out);

 //phase 2 Memory subsystem
 register IR(clock, clear, IRin, BusMuxOut, BusMuxInIR);
 registerR0 R0(clock, clear, R0in, BAout, BusMuxOut, BusMuxInR0);
 MAR mar(clock, clear, MARin, BusMuxOut,Address);
 ram1 ram(Address,clock,BusMuxInMDR, Write, ramOut);
 MDR mdr(clock, clear, MDRin, Read, BusMuxOut, ramOut, BusMuxInMDR);
 pc PC(clock, clear, PCin, incPC, branch, BusMuxOut, BusMuxInPC); //incPC to PCin
 to '1

 //phase 2 conff
conff CONFF(BusMuxOut, BusMuxInIR, ConIn, branch);

//instantiate bus
bus cpuBUS(
    BusMuxInR0,BusMuxInR1,BusMuxInR2, BusMuxInR3,BusMuxInR4,BusMuxInR5,BusMuxInR6
,BusMuxInR7,BusMuxInR8,BusMuxInR9,BusMuxInR10,BusMuxInR11,BusMuxInR12,BusMuxInR13
,BusMuxInR14,BusMuxInR15,
    BusMuxInRHi,BusMuxInLo,BusMuxInZHi,BusMuxInZLo,BusMuxInPC,BusMuxInMDR,BusMuxI
nRInP,BusMuxInCSign,
    R0out,R1out,R2out,R3out,R4out,R5out,R6out,R7out,R8out,R9out,R10out,R11out,R12
out,R13out,R14out,R15out,
    HIout,LOout,Zhighout,Zlowout,PCout,MDRout,INportout,Cout,BusMuxOut
);


ALU alu(aluControl, BusMuxInRY, BusMuxOut, ALULoOut, ALUHiOut);


endmodule

//we essentially need to perform an action anytime one of the control signals cha
nges
```

```
// testbench changes state every rising-
edge of clock, does the job associated with state for each
//#10 = time delay of 10 before instruction is ran, same concept with #15
// wires are used to create connection between to ports (input and output port),
they do not store data, only drive data
// MDRout = 1, updates BusMuxOut with value of BusMuxInMDR, since R2in  =1, R2 wi
ll hold value (write to register)
// the always @ block is used for cyclic behavior
```

Select & Encode

```
module SelectEncode(
    input [31:0] IR,
    input Gra, Grb, Grc, Rin, Rout, BAout,
    output [31:0] C_sign,
    output R0in, R1in, R2in, R3in, R4in, R5in, R6in, R7in, R8in, R9in, R10in, R11in, R12in,
R13in, R14in, R15in,
    output R0out, R1out, R2out, R3out, R4out, R5out, R6out, R7out, R8out, R9out, R10out,
R11out, R12out, R13out, R14out, R15out
);


//have two outputs instead
reg [3:0] OpCode, Ra, Rb, Rc;
reg [3:0] DecoderInput;
reg [15:0] DecoderOutput;
reg [15:0] Out;
reg [31:0] C_sign_extended;
reg [15:0] In;
reg temp;
integer i;



always @ (*) begin
    // Gra or Grb or Grc
     OpCode = IR[31:27];
     Ra = IR[26:23];
     Rb = IR[22:19];
     Rc = IR[18:15];


    if(Gra == 1) begin //Add
        DecoderInput = Ra;
     end
    else if(Grb == 1) begin //Sub
```

```
    DecoderInput = Rb;
 end
else if(Grc == 1) begin //Shift Right
    DecoderInput = Rc;
 end


if(DecoderInput == 4'b0000) begin
    DecoderOutput = 16'b0000000000000001;
end
else if(DecoderInput == 4'b0001) begin
    DecoderOutput = 16'b0000000000000010;
end
else if(DecoderInput == 4'b0010) begin
    DecoderOutput = 16'b0000000000000100;
end
else if(DecoderInput == 4'b0011) begin
    DecoderOutput = 16'b0000000000001000;
end
else if(DecoderInput == 4'b0100) begin
    DecoderOutput = 16'b0000000000010000;
end
else if(DecoderInput == 4'b0101) begin
    DecoderOutput = 16'b0000000000100000;
end
else if(DecoderInput == 4'b0110) begin
    DecoderOutput = 16'b0000000001000000;
end
else if(DecoderInput == 4'b0111) begin
    DecoderOutput = 16'b0000000010000000;
end
else if(DecoderInput == 4'b1000) begin
    DecoderOutput = 16'b0000000100000000;
end
else if(DecoderInput == 4'b1001) begin
    DecoderOutput = 16'b0000001000000000;
end
else if(DecoderInput == 4'b1010) begin
    DecoderOutput = 16'b0000010000000000;
end
else if(DecoderInput == 4'b1011) begin
    DecoderOutput = 16'b0000100000000000;
end
else if(DecoderInput == 4'b1100) begin
    DecoderOutput = 16'b0001000000000000;
end
```

```
    else if(DecoderInput == 4'b1101) begin
        DecoderOutput = 16'b0010000000000000;
    end
    else if(DecoderInput == 4'b1110) begin
        DecoderOutput = 16'b0100000000000000;
    end
    else if(DecoderInput == 4'b1111) begin
        DecoderOutput = 16'b1000000000000000;
    end
     else begin
        DecoderOutput = 16'b0000000000000000;
     end
    for (i = 0 ; i < 16 ; i = i + 1)
        In[i] = DecoderOutput[i] & Rin;
     temp = Rout | BAout;
    for (i = 0 ; i < 16 ; i = i + 1)
        Out[i] = DecoderOutput[i] & temp;
    C_sign_extended = {{13{IR[18]}}, {IR[18:0]}};

end
assign {R15in, R14in, R13in, R12in, R11in, R10in, R9in, R8in, R7in, R6in, R5in, R4in, R3in,
R2in, R1in, R0in} = In;
assign {R15out, R14out, R13out, R12out, R11out, R10out, R9out, R8out, R7out, R6out, R5out,
R4out, R3out, R2out, R1out, R0out} = Out;
assign C_sign = C_sign_extended;


endmodule
```

Conff

```
module conff(
    input [31:0] BusMuxIn,
    input [31:0] IR,  // C2 = IR[22:19]
    input ConIn, // phase 3
    output branch
);

/*
make input BusMuxInIR 32 bits and isolate the 4 required bits
output name should be controlOut
you should have an always block that runs anytime inputs change
assign statement will need to be outside of always block (may need a temp reg var
iable and assign output to that outside always block)

*/
```

```
reg temp;
reg temp2;
reg [3:0] BusMuxInIR;
integer i;



always @ (ConIn) begin

    BusMuxInIR = IR[22:19];

    if (BusMuxInIR[0] == 1'b1) begin

         temp = BusMuxInIR[0] & ~(|BusMuxIn);     // IR[0] AND Bus
         // decoder[0] & (|bus)

         if (temp == 1'b1) begin             // if equals 0
              temp2 = 1'b1;
         end
    end
    else if (BusMuxInIR[1] == 1'b1) begin

         temp = BusMuxInIR[1] & (|BusMuxIn);            // IR[1] AND (NOT Bus)

         if (temp != 1'b0) begin                // if not equals 0
                                                          // not working, c
heck with TA
              temp2 = temp;
         end
    end
    else if (BusMuxInIR[2] == 1'b1) begin
         temp = BusMuxInIR[2] & ~BusMuxIn[31];          // IR[2] AND (NOT Bus[31]
)

                                                                    // if gre
ater than equal to 0
         temp2 = temp;

    end
    else if (BusMuxInIR[3] == 1'b1) begin
         temp = BusMuxInIR[3] & BusMuxIn[31];           // IR[3] AND Bus[31]
                                                                 // if less th
an 0
         temp2 = temp;
```

```
    end

    if (ConIn == 1'b0) begin
        temp2 = 1'b0;
    end

end

assign branch = temp2; // (((temp[0] | temp[1]) | temp[2]) | temp[3]);


endmodule
```

MAR

```
module MAR(input clock, input clear, input MARin, input [31:0] BusMuxOut, output
[8:0] Address);

reg [31:0] q;
    // Behavioral section for writing to the register
    always @ ( posedge clock)
        begin
            if(clear) begin
                q <= 32'b0;
            end
            else if(MARin) begin
                q <= BusMuxOut;
            end
        end

    assign Address = q[8:0];

endmodule
```

MDR

```
module MDR(
    input clk, clear, MDRin, read,
    input [31:0] BusMuxOut,
    input [31:0] Mdatain,
    output [31:0] BusMuxInMDR
);
```

```
// Behavioral section for writing to the register
reg [31:0] Din;
reg [31:0] In;
always @ (posedge clk)
    begin
        if (read) begin
            Din <= Mdatain;
        end
        else begin
            Din<= BusMuxOut;
        end

        if(clear) begin
            In <= 32'b0;
        end
        else if(MDRin) begin
            In <= Din;
        end
    end

    assign BusMuxInMDR = In;

endmodule
```

RAM

```
// megafunction wizard: %RAM: 1-PORT%
// GENERATION: STANDARD
// VERSION: WM1.0
// MODULE: altsyncram

// ============================================================
// File Name: ram1.v
// Megafunction Name(s):
//          altsyncram
//
// Simulation Library Files(s):
//          altera_mf
// ============================================================
// ************************************************************
// THIS IS A WIZARD-GENERATED FILE. DO NOT EDIT THIS FILE!
//
// 13.0.1 Build 232 06/12/2013 SP 1 SJ Full Version
// ************************************************************
```

```
//Copyright (C) 1991-2013 Altera Corporation
//Your use of Altera Corporation's design tools, logic functions
//and other software and tools, and its AMPP partner logic
//functions, and any output files from any of the foregoing
//(including device programming or simulation files), and any
//associated documentation or information are expressly subject
//to the terms and conditions of the Altera Program License
//Subscription Agreement, Altera MegaCore Function License
//Agreement, or other applicable license agreement, including,
//without limitation, that your use is for the sole purpose of
//programming logic devices manufactured by Altera and sold by
//Altera or its authorized distributors.  Please refer to the
//applicable agreement for further details.


// synopsys translate_off
`timescale 1 ps / 1 ps
// synopsys translate_on
module ram1 (
    address,
    clock,
    data,
    wren,
    q);

    input   [8:0]  address;
    input     clock;
    input   [31:0]  data;
    input     wren;
    output  [31:0]  q;
`ifndef ALTERA_RESERVED_QIS
// synopsys translate_off
`endif
    tri1      clock;
`ifndef ALTERA_RESERVED_QIS
// synopsys translate_on
`endif

    wire [31:0] sub_wire0;
    wire [31:0] q = sub_wire0[31:0];

    altsyncram  altsyncram_component (
                .address_a (address),
                .clock0 (clock),
                .data_a (data),
```

```
                .wren_a (wren),
                .q_a (sub_wire0),
                .aclr0 (1'b0),
                .aclr1 (1'b0),
                .address_b (1'b1),
                .addressstall_a (1'b0),
                .addressstall_b (1'b0),
                .byteena_a (1'b1),
                .byteena_b (1'b1),
                .clock1 (1'b1),
                .clocken0 (1'b1),
                .clocken1 (1'b1),
                .clocken2 (1'b1),
                .clocken3 (1'b1),
                .data_b (1'b1),
                .eccstatus (),
                .q_b (),
                .rden_a (1'b1),
                .rden_b (1'b1),
                .wren_b (1'b0));
    defparam
        altsyncram_component.clock_enable_input_a = "BYPASS",
        altsyncram_component.clock_enable_output_a = "BYPASS",
`ifdef NO_PLI
        altsyncram_component.init_file = "lab.rif"
`else
        altsyncram_component.init_file = "lab.hex"
`endif
,
        altsyncram_component.intended_device_family = "Cyclone III",
        altsyncram_component.lpm_hint = "ENABLE_RUNTIME_MOD=NO",
        altsyncram_component.lpm_type = "altsyncram",
        altsyncram_component.numwords_a = 512,
        altsyncram_component.operation_mode = "SINGLE_PORT",
        altsyncram_component.outdata_aclr_a = "NONE",
        altsyncram_component.outdata_reg_a = "UNREGISTERED",
        altsyncram_component.power_up_uninitialized = "FALSE",
        altsyncram_component.read_during_write_mode_port_a = "DONT_CARE",
        altsyncram_component.widthad_a = 9,
        altsyncram_component.width_a = 32,
        altsyncram_component.width_byteena_a = 1;


endmodule
```

```
// ============================================================
// CNX file retrieval info
// ============================================================
// Retrieval info: PRIVATE: ADDRESSSTALL_A NUMERIC "0"
// Retrieval info: PRIVATE: AclrAddr NUMERIC "0"
// Retrieval info: PRIVATE: AclrByte NUMERIC "0"
// Retrieval info: PRIVATE: AclrData NUMERIC "0"
// Retrieval info: PRIVATE: AclrOutput NUMERIC "0"
// Retrieval info: PRIVATE: BYTE_ENABLE NUMERIC "0"
// Retrieval info: PRIVATE: BYTE_SIZE NUMERIC "8"
// Retrieval info: PRIVATE: BlankMemory NUMERIC "0"
// Retrieval info: PRIVATE: CLOCK_ENABLE_INPUT_A NUMERIC "0"
// Retrieval info: PRIVATE: CLOCK_ENABLE_OUTPUT_A NUMERIC "0"
// Retrieval info: PRIVATE: Clken NUMERIC "0"
// Retrieval info: PRIVATE: DataBusSeparated NUMERIC "1"
// Retrieval info: PRIVATE: IMPLEMENT_IN_LES NUMERIC "0"
// Retrieval info: PRIVATE: INIT_FILE_LAYOUT STRING "PORT_A"
// Retrieval info: PRIVATE: INIT_TO_SIM_X NUMERIC "0"
// Retrieval info: PRIVATE: INTENDED_DEVICE_FAMILY STRING "Cyclone III"
// Retrieval info: PRIVATE: JTAG_ENABLED NUMERIC "0"
// Retrieval info: PRIVATE: JTAG_ID STRING "NONE"
// Retrieval info: PRIVATE: MAXIMUM_DEPTH NUMERIC "0"
// Retrieval info: PRIVATE: MIFfilename STRING "lab.hex"
// Retrieval info: PRIVATE: NUMWORDS_A NUMERIC "512"
// Retrieval info: PRIVATE: RAM_BLOCK_TYPE NUMERIC "0"
// Retrieval info: PRIVATE: READ_DURING_WRITE_MODE_PORT_A NUMERIC "2"
// Retrieval info: PRIVATE: RegAddr NUMERIC "1"
// Retrieval info: PRIVATE: RegData NUMERIC "1"
// Retrieval info: PRIVATE: RegOutput NUMERIC "0"
// Retrieval info: PRIVATE: SYNTH_WRAPPER_GEN_POSTFIX STRING "0"
// Retrieval info: PRIVATE: SingleClock NUMERIC "1"
// Retrieval info: PRIVATE: UseDQRAM NUMERIC "1"
// Retrieval info: PRIVATE: WRCONTROL_ACLR_A NUMERIC "0"
// Retrieval info: PRIVATE: WidthAddr NUMERIC "9"
// Retrieval info: PRIVATE: WidthData NUMERIC "32"
// Retrieval info: PRIVATE: rden NUMERIC "0"
// Retrieval info: LIBRARY: altera_mf altera_mf.altera_mf_components.all
// Retrieval info: CONSTANT: CLOCK_ENABLE_INPUT_A STRING "BYPASS"
// Retrieval info: CONSTANT: CLOCK_ENABLE_OUTPUT_A STRING "BYPASS"
// Retrieval info: CONSTANT: INIT_FILE STRING "lab.hex"
// Retrieval info: CONSTANT: INTENDED_DEVICE_FAMILY STRING "Cyclone III"
// Retrieval info: CONSTANT: LPM_HINT STRING "ENABLE_RUNTIME_MOD=NO"
// Retrieval info: CONSTANT: LPM_TYPE STRING "altsyncram"
// Retrieval info: CONSTANT: NUMWORDS_A NUMERIC "512"
// Retrieval info: CONSTANT: OPERATION_MODE STRING "SINGLE_PORT"
```

```
// Retrieval info: CONSTANT: OUTDATA_ACLR_A STRING "NONE"
// Retrieval info: CONSTANT: OUTDATA_REG_A STRING "UNREGISTERED"
// Retrieval info: CONSTANT: POWER_UP_UNINITIALIZED STRING "FALSE"
// Retrieval info: CONSTANT: READ_DURING_WRITE_MODE_PORT_A STRING "DONT_CARE"
// Retrieval info: CONSTANT: WIDTHAD_A NUMERIC "9"
// Retrieval info: CONSTANT: WIDTH_A NUMERIC "32"
// Retrieval info: CONSTANT: WIDTH_BYTEENA_A NUMERIC "1"
// Retrieval info: USED_PORT: address 0 0 9 0 INPUT NODEFVAL "address[8..0]"
// Retrieval info: USED_PORT: clock 0 0 0 0 INPUT VCC "clock"
// Retrieval info: USED_PORT: data 0 0 32 0 INPUT NODEFVAL "data[31..0]"
// Retrieval info: USED_PORT: q 0 0 32 0 OUTPUT NODEFVAL "q[31..0]"
// Retrieval info: USED_PORT: wren 0 0 0 0 INPUT NODEFVAL "wren"
// Retrieval info: CONNECT: @address_a 0 0 9 0 address 0 0 9 0
// Retrieval info: CONNECT: @clock0 0 0 0 0 clock 0 0 0 0
// Retrieval info: CONNECT: @data_a 0 0 32 0 data 0 0 32 0
// Retrieval info: CONNECT: @wren_a 0 0 0 0 wren 0 0 0 0
// Retrieval info: CONNECT: q 0 0 32 0 @q_a 0 0 32 0
// Retrieval info: GEN_FILE: TYPE_NORMAL ram1.v TRUE
// Retrieval info: GEN_FILE: TYPE_NORMAL ram1.inc FALSE
// Retrieval info: GEN_FILE: TYPE_NORMAL ram1.cmp FALSE
// Retrieval info: GEN_FILE: TYPE_NORMAL ram1.bsf FALSE
// Retrieval info: GEN_FILE: TYPE_NORMAL ram1_inst.v FALSE
// Retrieval info: GEN_FILE: TYPE_NORMAL ram1_bb.v TRUE
// Retrieval info: LIB_FILE: altera_mf
```

PC

```
module pc #(parameter VAL = 0)(
    input clock,
    input clear,
    input enable,
    input incPC,
    input branch,
    input [31:0] BusMuxOut,
    output [31:0] BusMuxIn
);

reg [31:0] q;
initial q = VAL;

    // Behavioral section for writing to the register
    always @ (posedge clock)
        begin
            if(clear) begin
                q <= 32'b0;
```

```
            end
            else if(enable) begin
                q <= BusMuxOut;
            end
            else if(incPC) begin
                q <= q + 1;
            end
            else if(branch) begin
                q <= q + BusMuxOut;
            end
        end

    assign BusMuxIn = q;

endmodule
```

Bus

```
module bus(input [31:0] BusMuxInR0, input [31:0] BusMuxInR1, input [31:0] BusMuxI
nR2, input [31:0] BusMuxInR3,input [31:0] BusMuxInR4,input [31:0] BusMuxInR5,inpu
t [31:0] BusMuxInR6,input [31:0] BusMuxInR7,input [31:0] BusMuxInR8,input [31:0]
BusMuxInR9,input [31:0] BusMuxInR10,
input [31:0] BusMuxInR11,input [31:0] BusMuxInR12,input [31:0] BusMuxInR13,input
[31:0] BusMuxInR14,input [31:0] BusMuxInR15,input [31:0] BusMuxInHi,input [31:0]
BusMuxInLo,input [31:0] BusMuxInZHi,input [31:0] BusMuxInZLo,input [31:0] BusMuxI
nPC,
input [31:0] BusMuxInMDR,input [31:0] BusMuxInRInP,input [31:0] BusMuxInRCSign, i
nput R0out, input R1out,input R2out,input R3out,input R4out,input R5out,input R6o
ut,input R7out,input R8out,input R9out,input R10out,input R11out,input R12out,
input R13out,input R14out,input R15out,input HIout,input LOout,input Zhighout,inp
ut Zlowout,input PCout, input MDRout,input InPortout,input Cout, output [31:0] Bu
sMuxOut);

reg [31:0] out;

always @ (*) begin
// R0out or R1out or R2out or R4out or R5out or R6out or R7out or R8out or R9out
or R10out or R11out or R12out or R13out or R14out or R15out or MDRout or PCout or
 HIout or LOout or Zhighout or Zlowout or InPortout or Cout
    if(R0out) begin
        out = BusMuxInR0;
    end
    else if(Cout) begin
        out =  BusMuxInRCSign;
    end
```

```
    else if(R1out) begin
        out = BusMuxInR1;
    end
    else if(R2out) begin
        out = BusMuxInR2;
    end
    else if(R3out) begin
        out = BusMuxInR3;
    end
    else if(R4out) begin
        out = BusMuxInR4;
    end
    else if(R5out) begin
        out = BusMuxInR5;
    end
    else if(R6out) begin
        out = BusMuxInR6;
    end
    else if(R7out) begin
        out = BusMuxInR7;
    end
    else if(R8out) begin
        out = BusMuxInR8;
    end
    else if(R9out) begin
        out = BusMuxInR9;
    end
    else if(R10out) begin
        out = BusMuxInR10;
    end
    else if(R11out) begin
        out = BusMuxInR11;
    end
    else if(R12out) begin
        out = BusMuxInR12;
    end
    else if(R13out)begin
        out = BusMuxInR13;
    end
    else if(R14out)begin
        out = BusMuxInR14;
    end
    else if(R15out)begin
        out = BusMuxInR15;
    end
```

```
    else if(HIout)begin
        out = BusMuxInHi;
    end
    else if(LOout)begin
        out = BusMuxInLo;
    end
    else if(Zhighout)begin
        out = BusMuxInZHi;
    end
    else if(Zlowout)begin
        out = BusMuxInZLo;
    end
    else if(PCout)begin
        out = BusMuxInPC;
    end
    else if(MDRout)begin
        out = BusMuxInMDR;
    end
    else if(InPortout)begin
        out = BusMuxInRInP;
    end
//  else if(Cout) begin
//      out =  BusMuxInRCSign;
//  end
    else begin
        out = 32'bx;
    end
end
assign BusMuxOut = out;
endmodule
```

ALU

```
module ALU(
    input [4:0] aluControl,
    input [31:0] BusMuxInY,
     input [31:0] BusMuxOut,
    output [31:0] Zlowout,
     output [31:0] Zhighout
);

//have two outputs instead
//reg [4:0] aluControl;
reg [31:0] COut;
reg [31:0] temp;
```

```
reg [31:0] temp1;
reg [31:0] temp2;
wire [63:0] ZOut;
integer i;

boothmult Mult(ZOut, BusMuxInY, BusMuxOut);

always @ (*) begin

     // aluControl = Operator[31:27];

     temp1 = BusMuxOut;
     temp2 = BusMuxInY;
    if(aluControl == 5'b00011) begin //Add
        COut = BusMuxInY + BusMuxOut;
     end
    else if(aluControl == 5'b00100) begin //Sub
        COut = BusMuxInY - BusMuxOut;
     end
    else if(aluControl == 5'b00101) begin //Shift Right
        for (i = 0 ; i < 31 ; i = i + 1) begin
            COut[i] = BusMuxInY[i+1];
          end
        COut[31] = 0;
     end
    else if(aluControl == 5'b00110) begin //Shift Left
          for (i = 1 ; i < 32 ; i = i + 1) begin
            COut[i] = BusMuxInY[i-1];
          end
        COut[0] = 0;
     end
    else if(aluControl == 5'b00111) begin //Rotate Right
        for (i = 0 ; i < 31 ; i = i + 1) begin
            COut[i] = BusMuxInY[i+1];
          end
        COut[31] = BusMuxInY[0];
     end
    else if(aluControl == 5'b01000) begin //Rotate Left
          for (i = 1 ; i < 32 ; i = i + 1) begin
            COut[i] = BusMuxInY[i-1];
          end
        COut[0] = BusMuxInY[31];
     end
    else if(aluControl == 5'b01001) begin //AND
    //loop with for loop, then use logial and for each bit
```

```
        for (i = 0 ; i < 32 ; i = i + 1) begin
            COut = (BusMuxInY & temp1);
        end
    end
    else if(aluControl == 5'b01010) begin //OR
        for (i = 0 ; i < 32 ; i = i + 1) begin
            COut[i] = BusMuxInY[i] | BusMuxOut[i];
        end
    end
    else if(aluControl == 5'b01110) begin //Multiply
        COut = ZOut[31:0];
        temp = ZOut[63:32];
     end
    else if(aluControl == 5'b01111) begin //Divide
        COut = BusMuxInY / BusMuxOut;
     end
    else if(aluControl == 5'b10000) begin // Negate
        for (i = 0 ; i < 32 ; i = i + 1) begin
                COut[i] = ~temp1[i];
          end
          COut[0] = COut[0] + 1'b1;
     end
    else if(aluControl == 5'b10001) begin // Not
        for (i = 0 ; i < 32 ; i = i + 1) begin
                COut[i] = ~temp1[i];
          end
     end
     if(aluControl != 5'b01111)
        temp = {32{COut[31]}};
    // assign the results to z high and z low

end

assign Zhighout = temp; // ZOut[31:0]
assign Zlowout = COut;  // ZOut[63:32]

endmodule

// Z register holds the results of the operation in ALU
```

Testbenches and Simulation runs

Note that for all testbenches, steps T0-T2 are used to fetch the instruction from memory at address 0 by preloading the value of 0 in the PC (PCout = 1 and MARIn=1 to store address 0 in MAR for memory access of ram). Read and MDRIn are set to 1 in order to store the instruction,

which is then transferred to the IR via the bus (MDRout=1 and IRin=1). The instruction bit pattern is displayed by the contents of IR in all signal printouts.

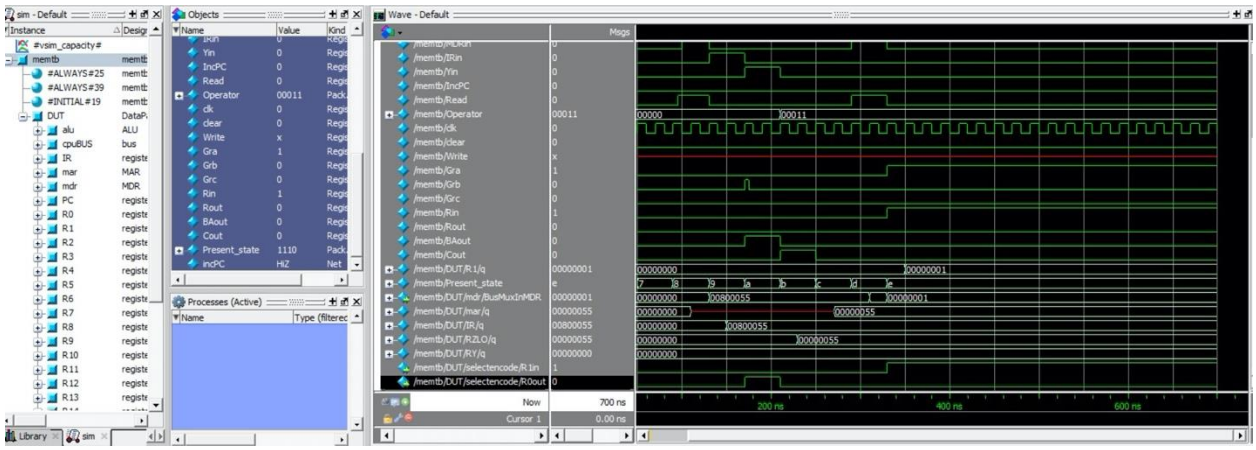
Load Instruction – ld R1, $85

In this case, we are preloading the value of 0 into R0. The select and encode module sets R0out to 1 in order to retrieve the value and store it into the Y register to compute the addition with the sign extended value 85. The result is then sent to the MAR via the bus to perform a memory read at the address 0x55 computed (85 in decimal). To verify, we placed a value of 1 at that address and checked that R1 contained 1.

```
`timescale 1ns/10ps

module memtb;
   reg PCout, Zlowout, MDRout; // add any other signals to see in your simulation
   reg MARin, Zin, PCin, MDRin, IRin, Yin;
   reg incPC, Read;
    reg [4:0] Operator;
   reg clk;
    reg clear, Write;
   reg Gra, Grb, Grc, Rin, Rout, BAout, Cout;

   parameter   Default = 4'b0000, T0= 4'b0111,T1= 4'b1000,T2= 4'b1001, T3= 4'b1010, T4= 4'b1011, T5=
4'b1100,T6= 4'b1101,T7= 4'b1110;
   reg[3:0] Present_state= Default;

DataPath DUT(PCout, Zlowout, MDRout, MARin, Zin, PCin, MDRin, IRin, Yin, incPC, Read, Write, Operator, clk,
clear,
Gra, Grb, Grc, Rin, Rout, BAout,Cout);


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


always @(posedge clk)     //finite state machine; if clk rising-edge
   begin
      case (Present_state)
         Default     :  #40 Present_state = T0;
         T0          :  #40 Present_state = T1;
         T1          :  #40 Present_state = T2;
         T2          :  #40 Present_state = T3;
         T3          :  #40 Present_state = T4;
         T4          :  #40 Present_state = T5;
         T5          :  #40 Present_state = T6;
         T6          :  #40 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 clk cycle
            Default: begin
                PCout <= 0;   Zlowout <= 0;   MDRout<= 0;   //initialize the signals
                MARin <= 0;   Zin <= 0;
                PCin <=0;   MDRin <= 0;   IRin  <= 0;   Yin <= 0;
                incPC <= 0;   Read <= 0;   Operator <= 5'b00000;
                Gra<= 0; Grb<= 0; Grc<= 0; Rin<= 0; Rout<= 0; BAout<= 0; Cout<=0;
                clear <= 1; //initialize registers to 0
                    #10 clear <= 0;
            end

            T0: begin
                //load value in PC register (0) in MAR
                #5 PCout <= 1;
                    #5 MARin <= 1;  //incPC <= 1; Zin <= 1; used to inc PC by 4
            end
            T1: begin
                //Zlowout <= 1; PCin <= 1;
                PCout <= 0;
                //MDR will grab value from ram @ address 0, this addess should contain instruction
                #5 Read <= 1;
                    #5 MDRin <= 1;

            end
            T2: begin
                Read <= 0;
                MARin <= 0;
                MDRin <= 0;
                //load MDR value to bus which contains instruction, instruction is then stored in IR
                MDRout<= 1; IRin <= 1;
            end

            T3: begin
                MDRout<= 0;
                IRin <= 0;
                //select R0out since Rb = 0000
                Grb <= 1;
                //set to 1 to generate R0out = 1 and store value of R0 (0) into Y register
                BAout <= 1;
                Yin <= 1;
                    #5 Grb <= 0;
            end

            T4: begin
```

```
                BAout <= 0;
            Yin <= 0;
            //store C sign extended value on bus (should be 85 decimal)
            Cout <=1;
            //perform add between 85/BuxMuxOut + R0 (0)/BuxMuxInY
                Operator <= 5'b00011;
            //store result in Z register
            Zin <=1;

        end

        //read from ram
        T5: begin
                Cout <= 0;
            Zin <=0;
                // Grb <= 0;

            //store Z register contents (85) in bus
            Zlowout <= 1;

            //load 85/busMuxOut into MAR to get address of 85
            MARin <= 1;

        end

        T6: begin
            Zlowout <=0;
            MARin<=0;
            //read from ram at address 85 and load that value in MDR -> we will need to have a preset value in memory
at address 85
                Read <= 1;
            MDRin <= 1;

        end

        T7: begin
                Read <= 0;
                MDRin <= 0;
                //load value in MDR on Bus
                MDRout <= 1;
                //set R1in signal to 1, value that was in memory at address 85 should now be stored in R1
                Gra <= 1;  //RA in Selectencode should be 0001 in binary
                Rin <= 1;
        end

    endcase
  end
endmodule
```



Load Instruction – ld R0, $35(R1)

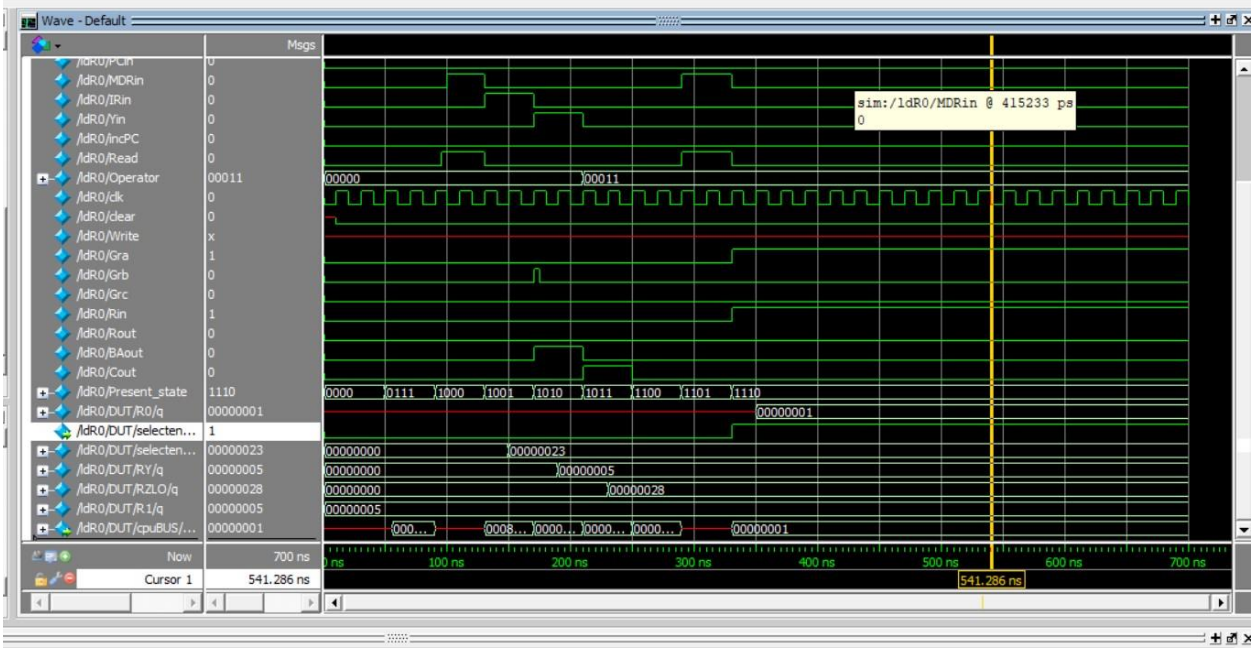
In this case, we are preloading the value of 5 into R1. The select and encode module sets R1out to 1 in order to retrieve the value and store it into the Y register to compute the addition with the sign extended value 35. The result is then sent to the MAR via the bus to perform a memory read at the address 0x28 computed (40 in decimal). To verify, we placed a value of 1 at that address and checked that R0 contained 1.

```
`timescale 1ns/10ps

module memtb;
    reg PCout, Zlowout, MDRout, R2out, R4out;
    reg MARin, Zin, PCin, MDRin, IRin, Yin;
    reg incPC, Read;
    reg [4:0] Operator;
    reg clk;
    reg clear,Write;
    reg Gra, Grb, Grc, Rin, Rout, BAout, Cout;

    parameter   Default = 4'b0000, T0= 4'b0111, T1= 4'b1000, T2= 4'b1001, T3= 4'b1010, T4= 4'b1011, T5=
4'b1100,T6= 4'b1101,T7= 4'b1110;
    reg[3:0] Present_state= Default;

DataPath DUT(PCout, Zlowout, MDRout, MARin, Zin, PCin, MDRin, IRin, Yin, Read, Write, Operator, clk, clear,
Gra, Grb, Grc, Rin, Rout, BAout,Cout);


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


always @(posedge clk)     //finite state machine; if clk rising-edge
    begin
        case (Present_state)
```

```
        Default    :  #40 Present_state = T0;
        T0         :  #40 Present_state = T1;
        T1         :  #40 Present_state = T2;
        T2         :  #40 Present_state = T3;
        T3         :  #40 Present_state = T4;
        T4         :  #40 Present_state = T5;
        T5         :  #40 Present_state = T6;
        T6         :  #40 Present_state = T7;
    endcase
  end

always @(Present_state)     // do the required job ineach state
  begin
    case (Present_state)             //assert the required signals in each clk cycle
      Default: begin
        PCout <= 0;   Zlowout <= 0;   MDRout<= 0;   //initialize the signals
        MARin <= 0;   Zin <= 0;
        PCin <=0;   MDRin <= 0;   IRin  <= 0;   Yin <= 0;
        incPC <= 0;   Read <= 0;   Operator <= 5'b00000;
        Gra<= 0; Grb<= 0; Grc<= 0; Rin<= 0; Rout<= 0; BAout<= 0; Cout<=0;
        clear <= 1; //initialize registers to 0
            #10 clear <= 0;
      end

      T0: begin
        //load value in PC register (0) in MAR
        #5 PCout <= 1;
        #5 MARin <= 1;
      end
      T1: begin
        PCout <= 0;
        //MDR will grab value from ram @ address 0, this addess should contain instruction
        #5 Read <= 1;
        #5 MDRin <= 1;

      end
      T2: begin
        Read <= 0;
        MARin <= 0;
        MDRin <= 0;
        //load MDR value to bus which contains instruction, instruction is then stored in IR
        MDRout<= 1; IRin <= 1;
      end

      T3: begin
        MDRout<= 0;
        IRin <= 0;
        //select R1out since Rb = 0001
```

```
            Grb <= 1;
            //set to 1 to generate R1out = 1 and store value of R1 (5 decimal) into Y register
            BAout <= 1;
            Yin <= 1;
            #5 Grb <= 0;
        end

        T4: begin
            BAout <= 0;
            Yin <= 0;
            //store C sign extended value on bus (should be 35 decimal)
            Cout <=1;
            //perform add between 35/BuxMuxOut + R1 (5)/BuxMuxInY
            Operator <= 5'b00011;
            //store result in Z register
            Zin <=1;

        end

        //read from ram
        T5: begin
            Cout <= 0;
            Zin <=0;

            //store Z register contents (40) in bus
            Zlowout <= 1;

            //load 40/busMuxOut into MAR to get address of 40
            MARin <= 1;

        end

        T6: begin
            Zlowout <=0;
            MARin<=0;
            //read from ram at address 40 and load that value in MDR -> we will need to have a preset value in memory
at address 40
            Read <= 1;
            MDRin <= 1;

        end

        T7: begin
                Read <= 0;
                MDRin <= 0;
                //load value in MDR on Bus
                MDRout <= 1;
                //set R0in signal to 1, value that was in memory at address 40 should now be stored in R1
```

```
            Gra <= 1;  //RA in Selectencode should be 0001 in binary
            Rin <= 1;
        end

    endcase
  end
endmodule
```



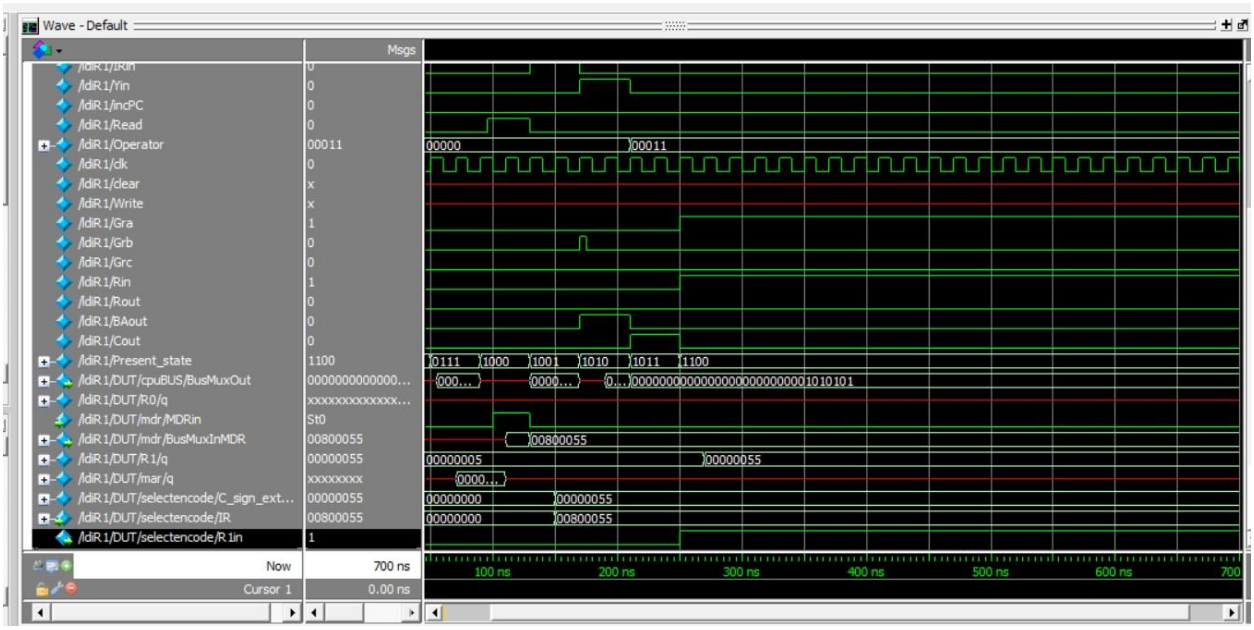
Load Instruction – ldi R1, $85

The testbench written to test this instruction required less steps as the value in the instruction is directly loaded to register R1. That is, the C sign extended value contains 0x55 (85), Gra and Rin =1 to isolate the four bits in the instruction that correspond to 0001 to set R1in to 1 and load the immediate value 0x55 in R1. The results clearly indicate that R1 contains 0x55 (85) after the fifth cycle.

```
`timescale 1ns/10ps

module ldiR1;
   reg PCout, Zlowout, MDRout, R2out, R4out;// add any other signals to see in your simulation
   reg MARin, Zin, PCin, MDRin, IRin, Yin;
   reg incPC, Read;
   reg [4:0] Operator;
   reg clk;
   reg clear,Write;
   reg Gra, Grb, Grc, Rin, Rout, BAout, Cout;

   parameter   Default = 4'b0000, T0= 4'b0111,T1= 4'b1000,T2= 4'b1001, T3= 4'b1010, T4= 4'b1011, T5=
4'b1100,T6= 4'b1101,T7= 4'b1110;
   reg[3:0] Present_state= Default;
```

```
DataPath DUT(PCout, Zlowout, MDRout, MARin, Zin, PCin, MDRin, IRin, Yin, Read, Write, Operator, clk, clear,
Gra, Grb, Grc, Rin, Rout, BAout,Cout);


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


always @(posedge clk)     //finite state machine; if clk rising-edge
   begin
      case (Present_state)
         Default     :   #40 Present_state = T0;
         T0          :   #40 Present_state = T1;
         T1          :   #40 Present_state = T2;
         T2          :   #40 Present_state = T3;
         T3          :   #40 Present_state = T4;
         T4          :   #40 Present_state = T5;
      endcase
   end

always @(Present_state)     // do the required job ineach state
   begin
      case (Present_state)             //assert the required signals in each clk cycle
         Default: begin
            PCout <= 0;   Zlowout <= 0;   MDRout<= 0;   //initialize the signals
            MARin <= 0;   Zin <= 0;
            PCin <=0;   MDRin <= 0;   IRin  <= 0;   Yin <= 0;
            incPC <= 0;   Read <= 0;   Operator <= 5'b00000;
            Gra<= 0; Grb<= 0; Grc<= 0; Rin<= 0; Rout<= 0; BAout<= 0; Cout<=0;
            clear <= 1; //initialize registers to 0
                #10 clear <= 0;
         end

         T0: begin
            //load value in PC register (0) in MAR
            #5 PCout <= 1;
                #5 MARin <= 1;  //IncPC <= 1; Zin <= 1; used to inc PC by 4
         end
         T1: begin
            //Zlowout <= 1; PCin <= 1;
            PCout <= 0;
            //MDR will grab value from ram @ address 0, this addess should contain instruction
            #5 Read <= 1;
            #5 MDRin <= 1;

         end
```

```
        T2: begin
            Read <= 0;
            MARin <= 0;
            MDRin <= 0;
            //load MDR value to bus which contains instruction, instruction is then stored in IR
            MDRout<= 1; IRin <= 1;
        end

        T3: begin
            MDRout<= 0;
            IRin <= 0;
            //select R0out since Rb = 0000
            Grb <= 1;
            //set to 1 to generate R0out = 1
            BAout <= 1;
            Yin <= 1;
            #5 Grb <= 0;
        end

        T4: begin
            BAout <= 0;
            Yin <= 0;
            //store C sign extended value on bus (should be 85 decimal)
            Cout <=1;
            //perform add between 85/BuxMuxOut + R0 (0)/BuxMuxInY
            Operator <= 5'b00011;
            //store result in Z register
            Zin <=1;

        end

        //read from ram
        T5: begin
            Cout <= 0;
            Zin <=0;

            //store Z register contents (85) in bus
            Zlowout <= 1;
            Gra <= 1;  //RA in Selectencode should be 0001 in binary
            Rin <= 1;


        end


    endcase
  end
endmodule
```



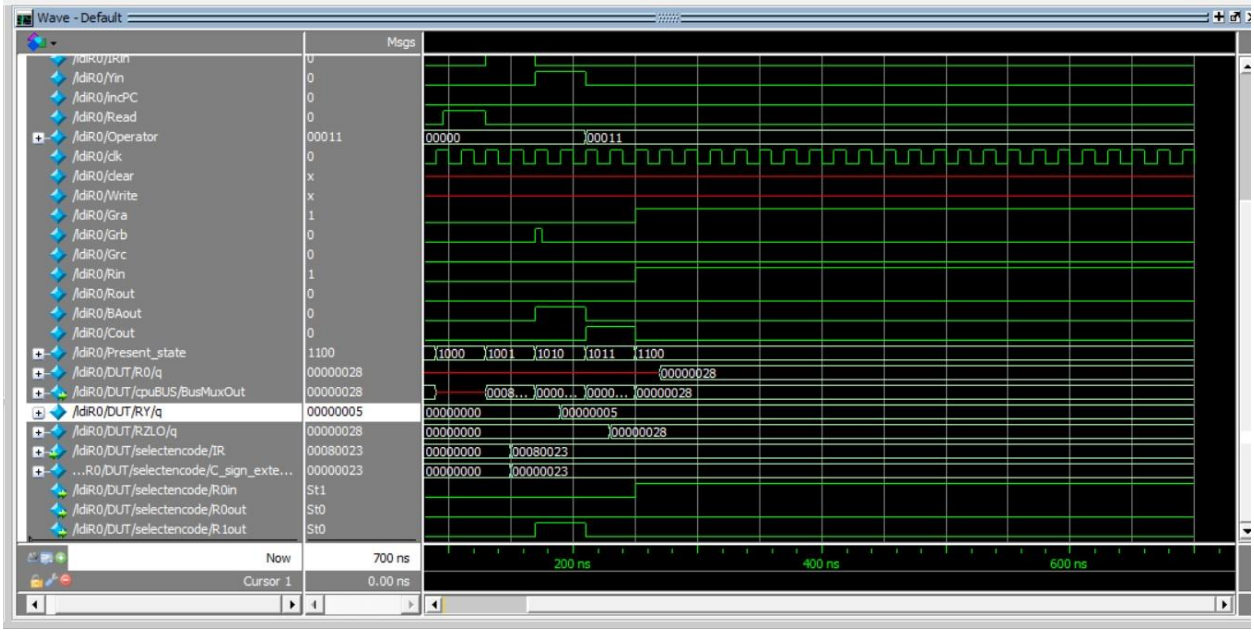
Load Instruction – ldi R0, $35(R1)

This load instruction loads the immediate value that is the sum of the contents of R1 and 35. That is, the C sign extended value contains 0x23 (35) and R1 was preloaded with the value 5. To validate this test, we verified that R0 contained 0x28.

```
`timescale 1ns/10ps

module ldiR0;
    reg PCout, Zlowout, MDRout, R2out, R4out;// add any other signals to see in your simulation
    reg MARin, Zin, PCin, MDRin, IRin, Yin;
    reg incPC, Read;
    reg [4:0] Operator;
    reg clk;
    reg clear,Write;
    reg Gra, Grb, Grc, Rin, Rout, BAout, Cout;

    parameter   Default = 4'b0000, T0= 4'b0111,T1= 4'b1000,T2= 4'b1001, T3= 4'b1010, T4= 4'b1011, T5=
4'b1100,T6= 4'b1101,T7= 4'b1110;
    reg[3:0] Present_state= Default;

DataPath DUT(PCout, Zlowout, MDRout, MARin, Zin, PCin, MDRin, IRin, Yin, Read, Write, Operator, clk, clear,
Gra, Grb, Grc, Rin, Rout, BAout,Cout);


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

```
always @(posedge clk)     //finite state machine; if clk rising-edge
   begin
      case (Present_state)
         Default     :  #40 Present_state = T0;
         T0          :  #40 Present_state = T1;
         T1          :  #40 Present_state = T2;
         T2          :  #40 Present_state = T3;
         T3          :  #40 Present_state = T4;
         T4          :  #40 Present_state = T5;
      endcase
   end

always @(Present_state)     // do the required job ineach state
   begin
      case (Present_state)             //assert the required signals in each clk cycle
         Default: begin
            PCout <= 0;   Zlowout <= 0;   MDRout<= 0;   //initialize the signals
            MARin <= 0;   Zin <= 0;
            PCin <=0;   MDRin <= 0;   IRin  <= 0;   Yin <= 0;
            incPC <= 0;   Read <= 0;   Operator <= 5'b00000;
            Gra<= 0; Grb<= 0; Grc<= 0; Rin<= 0; Rout<= 0; BAout<= 0; Cout<=0;
            clear <= 1; //initialize registers to 0
                #10 clear <= 0;
         end

         T0: begin
            //load value in PC register (0) in MAR
            #5 PCout <= 1;
                #5 MARin <= 1;  //IncPC <= 1; Zin <= 1; used to inc PC by 4
         end
         T1: begin
            //Zlowout <= 1; PCin <= 1;
            PCout <= 0;
            //MDR will grab value from ram @ address 0, this addess should contain instruction
            #5 Read <= 1;
            #5 MDRin <= 1;

         end
         T2: begin
            Read <= 0;
            MARin <= 0;
            MDRin <= 0;
            //load MDR value to bus which contains instruction, instruction is then stored in IR
            MDRout<= 1; IRin <= 1;
         end
```

```
        T3: begin
            MDRout<= 0;
            IRin <= 0;
            Grb <= 1;
            //set to 1 to generate R1out = 1 and store value of R1 (5 hex) into Y register
            BAout <= 1;
            Yin <= 1;
            #5 Grb <= 0;
        end

        T4: begin
            BAout <= 0;
            Yin <= 0;
            //store C sign extended value on bus (should be 35 decimal)
            Cout <=1;
            //perform add between 35/BuxMuxOut + R1 (5)/BuxMuxInY
            Operator <= 5'b00011;
            //store result in Z register
            Zin <=1;

        end

        //read from ram
        T5: begin
            Cout <= 0;
            Zin <=0;

            //store Z register contents (85) in bus
            Zlowout <= 1;
            Gra <= 1;  //RA in Selectencode should be 0000 in binary
            Rin <= 1;


        end
       //roin, r1out, y register, csignextended, z register, r1, r0

    endcase
  end
endmodule
```



Store Instruction – st $90, R1

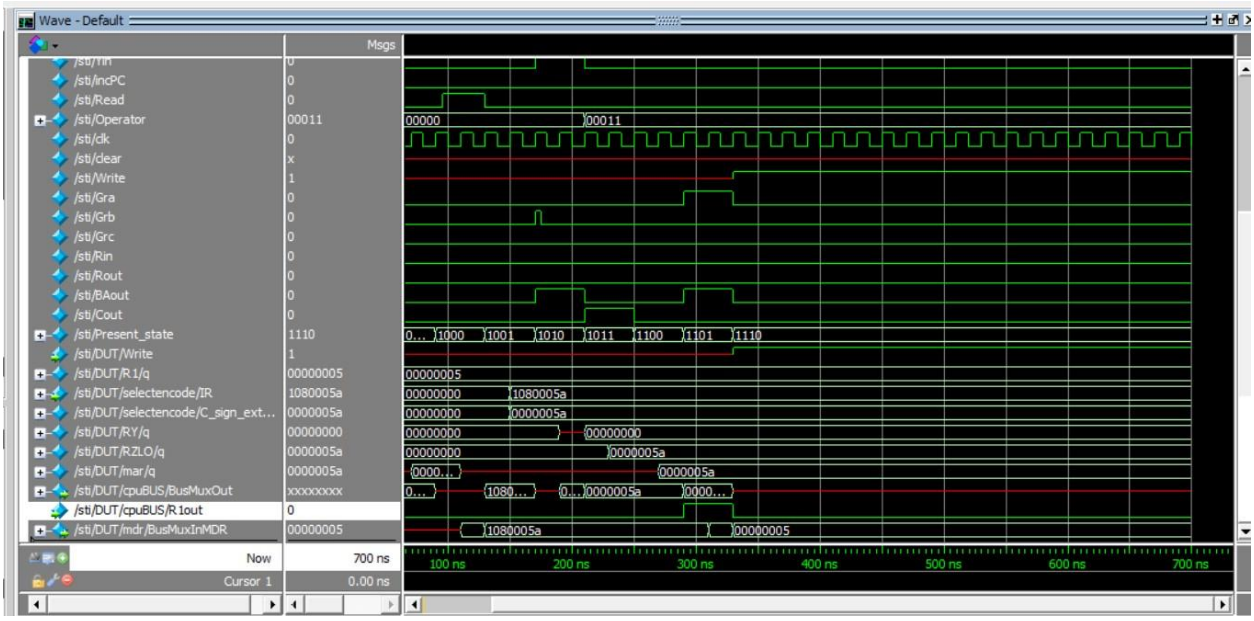
This instruction verifies that the design can write the contents of R1 to the address 90 in ram. After loading the instruction into IR (T0-T2), the address is computed (90 + contents of R0 which are 0). Once the address is stored into the MAR and the value of R1 is transferred to MDR via the bus, the write signal is set to 1 to perform a memory write. Since R1 was preloaded with 5, we verified the memory contents at address 0x5A.

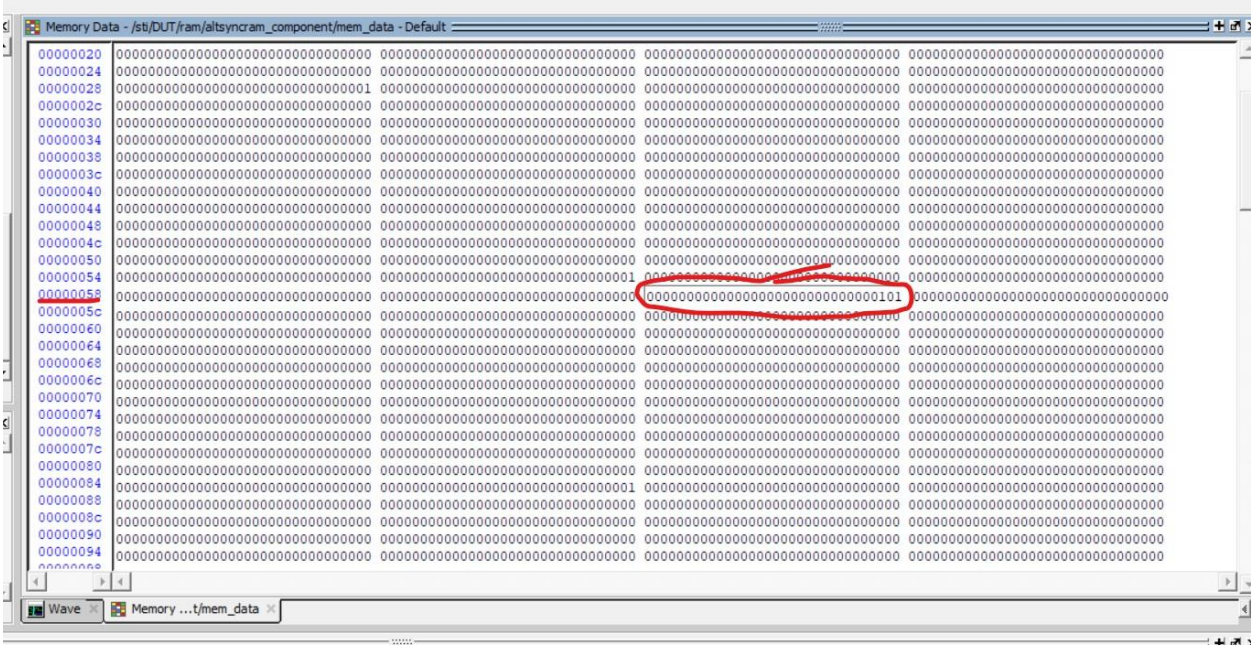
```
`timescale 1ns/10ps

module store;
    reg PCout, Zlowout, MDRout, R2out, R4out;// add any other signals to see in your simulation
    reg MARin, Zin, PCin, MDRin, IRin, Yin;
    reg incPC, Read;
    reg [4:0] Operator;
    reg clk;
    reg clear,Write;
    reg Gra, Grb, Grc, Rin, Rout, BAout, Cout;

    parameter   Default = 4'b0000, T0= 4'b0111,T1= 4'b1000,T2= 4'b1001, T3= 4'b1010, T4= 4'b1011, T5=
4'b1100,T6= 4'b1101,T7= 4'b1110;
    reg[3:0] Present_state= Default;

DataPath DUT(PCout, Zlowout, MDRout, MARin, Zin, PCin, MDRin, IRin, Yin, Read, Write, Operator, clk, clear,
Gra, Grb, Grc, Rin, Rout, BAout,Cout);


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

```
always @(posedge clk)     //finite state machine; if clk rising-edge
   begin
      case (Present_state)
         Default     :   #40 Present_state = T0;
         T0          :   #40 Present_state = T1;
         T1          :   #40 Present_state = T2;
         T2          :   #40 Present_state = T3;
         T3          :   #40 Present_state = T4;
         T4          :   #40 Present_state = T5;
         T5          :   #40 Present_state = T6;
         T6          :   #40 Present_state = T7;
      endcase
   end

always @(Present_state)     // do the required job ineach state
   begin
      case (Present_state)             //assert the required signals in each clk cycle
         Default: begin
            PCout <= 0;   Zlowout <= 0;   MDRout<= 0;   //initialize the signals
            MARin <= 0;   Zin <= 0;
            PCin <=0;   MDRin <= 0;   IRin  <= 0;   Yin <= 0;
            incPC <= 0;   Read <= 0;   Operator <= 5'b00000;
            Gra<= 0; Grb<= 0; Grc<= 0; Rin<= 0; Rout<= 0; BAout<= 0; Cout<=0;
            clear <= 0;
         end

         T0: begin
            //load value in PC register (0) in MAR
            #5 PCout <= 1;
            #5 MARin <= 1;  //IncPC <= 1; Zin <= 1; used to inc PC by 4
         end
         T1: begin
            //Zlowout <= 1; PCin <= 1;
            PCout <= 0;
            //MDR will grab value from ram @ address 0, this addess should contain instruction
            #5 Read <= 1;
            #5 MDRin <= 1;

         end
         T2: begin
            Read <= 0;
            MARin <= 0;
            MDRin <= 0;
            //load MDR value to bus which contains instruction, instruction is then stored in IR
            MDRout<= 1; IRin <= 1;
         end

         T3: begin
```

```
    MDRout<= 0;
    IRin <= 0;
    //select R0out since Rb = 0000
    Grb <= 1;
    //set to 1 to generate R0out = 1 and store value of R0 (0 hex) into Y register
    BAout <= 1;
    Yin <= 1;
    #5 Grb <= 0;
end

T4: begin
    BAout <= 0;
    Yin <= 0;
    //store C sign extended value on bus (should be 90 decimal)
    Cout <=1;
    //perform add between 90/BuxMuxOut + R0 (0)/BuxMuxInY
    Operator <= 5'b00011;
    //store result in Z register
    Zin <=1;

end

T5: begin
    Cout <= 0;
    Zin <=0;

    //store Z register contents (85) in bus
    Zlowout <= 1;

    //load 90/busMuxOut into MAR to get address of 85
    MARin <= 1;

end

 T6: begin
   Zlowout <=0;
   MARin<=0;
   Gra <= 1;  //RA in Selectencode should be 0001 in binary
   BAout <= 1;
   MDRin <= 1; //load value of 85 from R1 to MDR

end

T7: begin
       MDRin <= 0;
       Gra <= 0;
       BAOut <=0;
       Write <= 1;
```

```
        end


    endcase
  end
endmodule
```





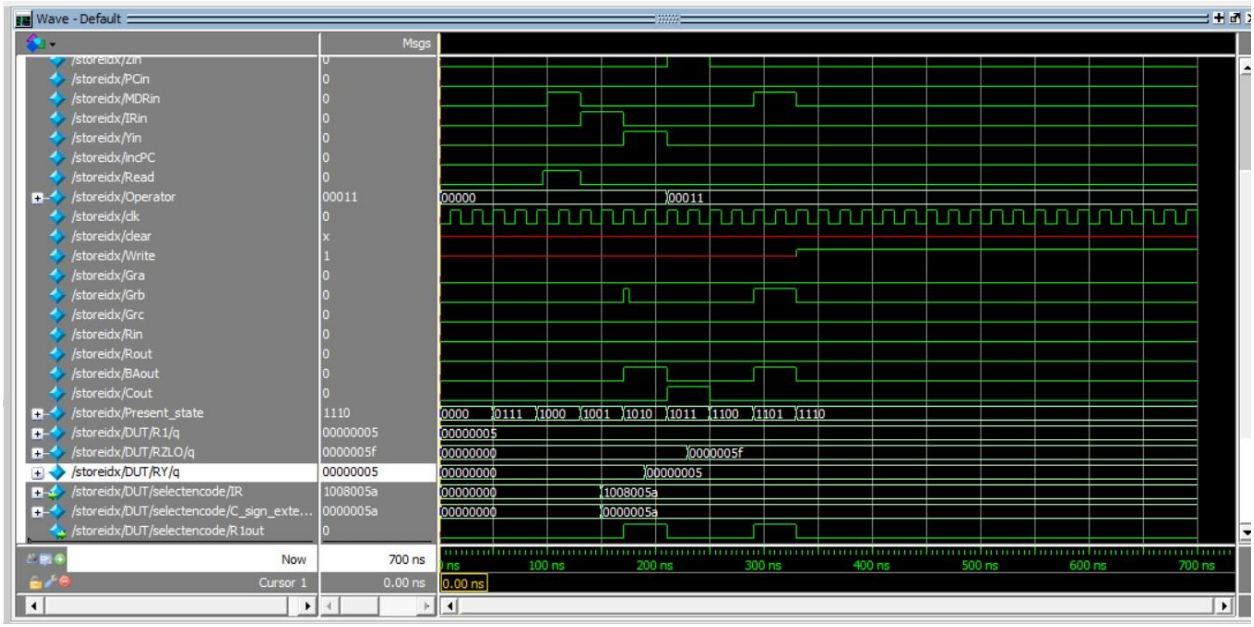
Store Instruction – st $90(R1), R1

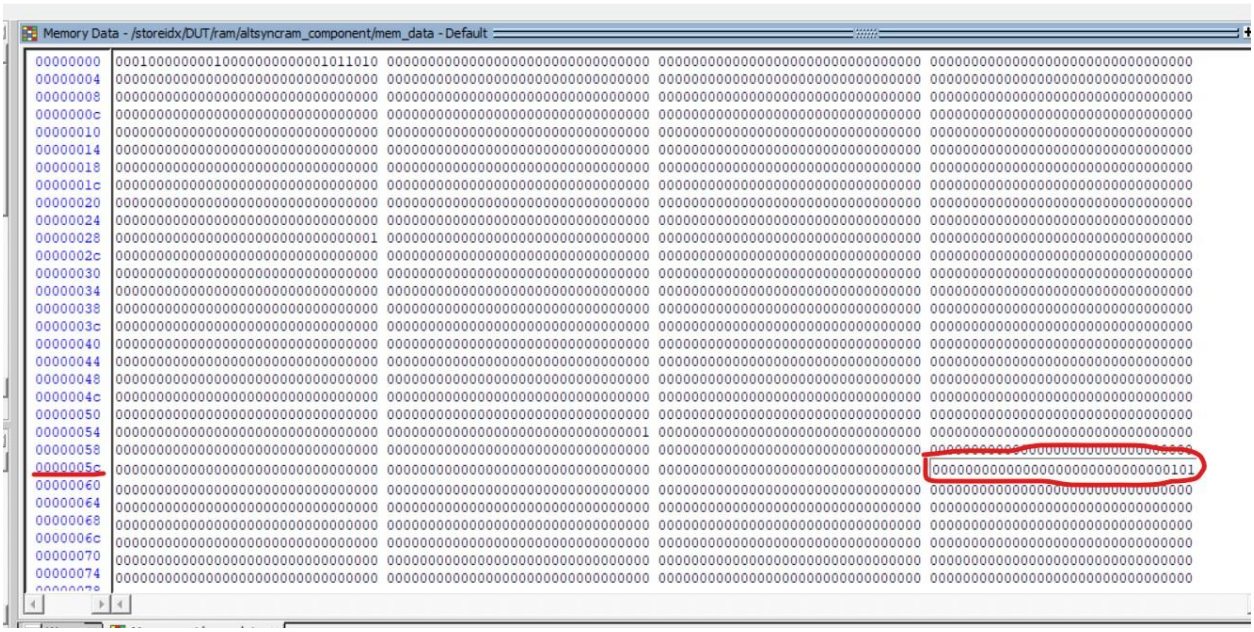
This instruction follows the same steps as the previous test except that the address is calculated by adding 90 with the value stored in R1 (5). Therefore, we verified that 5 was written at address 0x5f.

```
`timescale 1ns/10ps

module storeidx;
    reg PCout, Zlowout, MDRout, R2out, R4out;// add any other signals to see in your simulation
    reg MARin, Zin, PCin, MDRin, IRin, Yin;
    reg incPC, Read;
    reg [4:0] Operator;
    reg clk;
    reg clear,Write;
    reg Gra, Grb, Grc, Rin, Rout, BAout, Cout;

    parameter   Default = 4'b0000, T0= 4'b0111,T1= 4'b1000,T2= 4'b1001, T3= 4'b1010, T4= 4'b1011, T5=
4'b1100,T6= 4'b1101,T7= 4'b1110;
    reg[3:0] Present_state= Default;

DataPath DUT(PCout, Zlowout, MDRout, MARin, Zin, PCin, MDRin, IRin, Yin, Read, Write, Operator, clk, clear,
Gra, Grb, Grc, Rin, Rout, BAout,Cout);


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


always @(posedge clk)     //finite state machine; if clk rising-edge
    begin
        case (Present_state)
            Default     :   #40 Present_state = T0;
            T0          :   #40 Present_state = T1;
            T1          :   #40 Present_state = T2;
            T2          :   #40 Present_state = T3;
            T3          :   #40 Present_state = T4;
            T4          :   #40 Present_state = T5;
            T5          :   #40 Present_state = T6;
            T6          :   #40 Present_state = T7;
        endcase
    end

always @(Present_state)     // do the required job ineach state
    begin
        case (Present_state)              //assert the required signals in each clk cycle
            Default: begin
                PCout <= 0;   Zlowout <= 0;   MDRout<= 0;   //initialize the signals
                MARin <= 0;   Zin <= 0;
                PCin <=0;   MDRin <= 0;   IRin  <= 0;   Yin <= 0;
                incPC <= 0;   Read <= 0;   Operator <= 5'b00000;
                Gra<= 0; Grb<= 0; Grc<= 0; Rin<= 0; Rout<= 0; BAout<= 0; Cout<=0;
```

```
        clear <= 0;
    end

    T0: begin
        //load value in PC register (0) in MAR
        #5 PCout <= 1;
        #5 MARin <= 1;  //IncPC <= 1; Zin <= 1; used to inc PC by 4
    end
    T1: begin
        //Zlowout <= 1; PCin <= 1;
        PCout <= 0;
        //MDR will grab value from ram @ address 0, this addess should contain instruction
        #5 Read <= 1;
        #5 MDRin <= 1;

    end
    T2: begin
        Read <= 0;
        MARin <= 0;
        MDRin <= 0;
        //load MDR value to bus which contains instruction, instruction is then stored in IR
        MDRout<= 1; IRin <= 1;
    end

    T3: begin
        MDRout<= 0;
        IRin <= 0;
        //select R1out since Rb = 0001
        Grb <= 1;
        //set to 1 to generate R1out = 1 and store value of R1 (85 hex) into Y register
        BAout <= 1;
        Yin <= 1;
        #5 Grb <= 0;
    end

    T4: begin
        BAout <= 0;
        Yin <= 0;
        //store C sign extended value on bus (should be 90 decimal)
        Cout <=1;
        //perform add between 90/BuxMuxOut + R1 (85)/BuxMuxInY
        Operator <= 5'b00011;
        //store result in Z register
        Zin <=1;

    end

    T5: begin
```

```
            Cout <= 0;
            Zin <=0;

            //store Z register contents (85) in bus
            Zlowout <= 1;

            //load 90/busMuxOut into MAR to get address of 85
            MARin <= 1;

         end

          T6: begin
            Zlowout <=0;
            MARin<=0;
            Grb <= 1;  //RA in Selectencode should be 0001 in binary
            BAout <= 1;
            MDRin <= 1; //load value of 85 from R1 to MDR

         end

         T7: begin
               MDRin <= 0;
               Grb <= 0;
               BAOut <=0;
               Write <= 1;

         end


      endcase
   end
endmodule
```


Wave - Default
Msgs
/storeidx/PCin
/storeidx/MDRin
/storeidx/IRin
/storeidx/Yin
/storeidx/incPC
/storeidx/Read
/storeidx/Operator
/storeidx/clk
/storeidx/clear
/storeidx/Write
/storeidx/Gra
/storeidx/Grb
/storeidx/Grc
/storeidx/Rin
/storeidx/Rout
/storeidx/BAout
/storeidx/Cout
/storeidx/Present_state
/storeidx/DUT/R1/q
/storeidx/DUT/RZLO/q
/storeidx/DUT/RY/q
/storeidx/DUT/selectencode/IR
/storeidx/DUT/selectencode/C_sign_exte...
/storeidx/DUT/selectencode/R1out
00011
1110
00000005
0000005f
1008005a
Now
700 ns
Cursor 1
0.00 ns
100 ns
200 ns
300 ns
400 ns
500 ns
600 ns


Memory Data - /storeidx/DUT/ram/altsyncram_component/mem_data - Default
00000000
00000004
00000008
0000000c
00000010
00000014
00000018
0000001c
00000020
00000024
00000028
0000002c
00000030
00000034
00000038
0000003c
00000040
00000044
00000048
0000004c
00000050
00000054
00000058
0000005c
00000060
00000064
00000068
0000006c
00000070
00000074

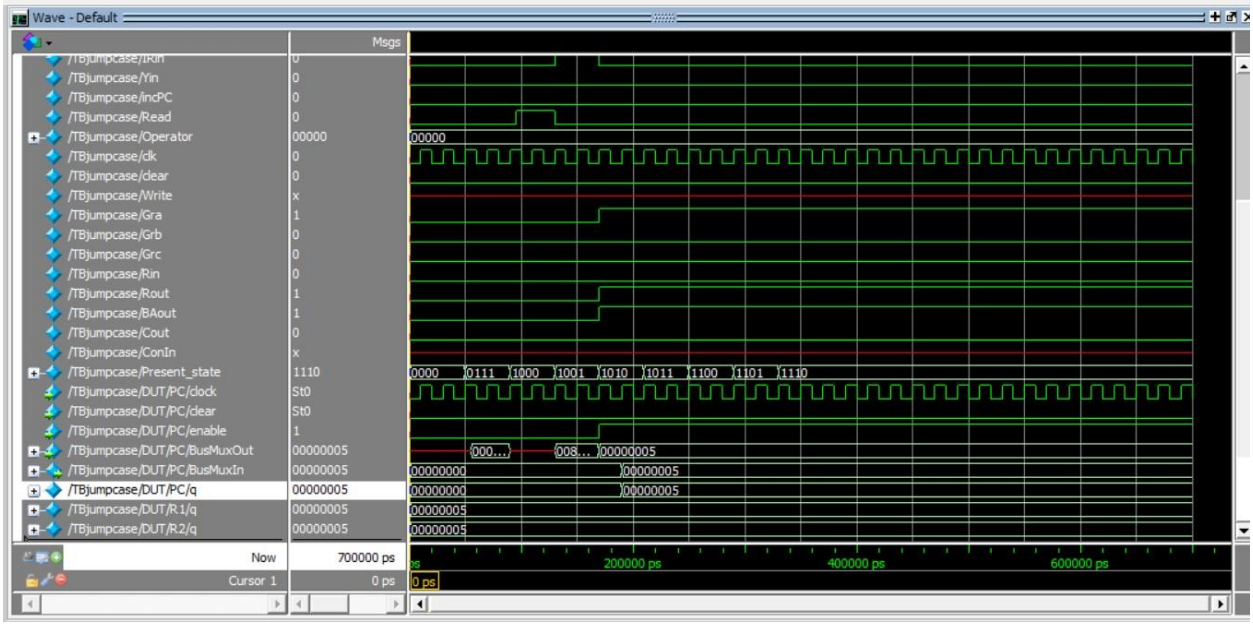
Verify Jump Case

Here we are verifying the functionality of the jr instruction by simulating the control sequence for jr R1. Note that R1 was preloaded with the value 5.

```
`timescale 1ns/10ps

module TBjumpcase;
    reg PCout, Zlowout, MDRout, R2out, R4out;// add any other signals to see in y
our simulation
    reg MARin, Zin, PCin, MDRin, IRin, Yin;
    reg incPC, Read;
     reg [4:0] Operator;
    reg clk;
     reg clear, Write;
    reg Gra, Grb, Grc, Rin, Rout, BAout, Cout, ConIn;

    parameter   Default = 4'b0000, T0= 4'b0111,T1= 4'b1000,T2= 4'b1001, T3= 4'b10
10, T4= 4'b1011, T5= 4'b1100,T6= 4'b1101,T7= 4'b1110;
    reg[3:0] Present_state= Default;

DataPath DUT(PCout, Zlowout, MDRout, MARin, Zin, PCin, MDRin, IRin, Yin, Read, Wr
ite, Operator, clk, clear,
Gra, Grb, Grc, Rin, Rout, BAout, Cout, ConIn);


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


always @(posedge clk)     //finite state machine; if clk rising-edge
    begin
        case (Present_state)
            Default     :   #40 Present_state = T0;
            T0          :   #40 Present_state = T1;
            T1          :   #40 Present_state = T2;
            T2          :   #40 Present_state = T3;
            T3          :   #40 Present_state = T4;
            T4          :   #40 Present_state = T5;
            T5          :   #40 Present_state = T6;
            T6          :   #40 Present_state = T7;
        endcase
    end
```

```
always @(Present_state)     // do the required job ineach state
    begin
        case (Present_state)              //assert the required signals in each c
lk cycle
            Default: begin
                PCout <= 0;   Zlowout <= 0;   MDRout<= 0;   //initialize the sign
als
                MARin <= 0;   Zin <= 0;
                PCin <= 0;   MDRin <= 0;   IRin  <= 0;   Yin <= 0;
                incPC <= 0;   Read <= 0;   Operator <= 5'b00000;
                Gra<= 0; Grb <= 0; Grc <= 0; Rin <= 0; Rout <= 0; BAout <= 0; Cou
t <= 0;
                // clear <= 1; //initialize registers to 0
                     clear <= 0;
            end

            T0: begin
                //load value in PC register (0) in MAR
                #5 PCout <= 1;
                     #5 MARin <= 1;  //IncPC <= 1; Zin <= 1; used to inc PC by 4
            end
            T1: begin
                //Zlowout <= 1; PCin <= 1;
                PCout <= 0;
                     MARin <= 0;
                //MDR will grab value from ram @ address 0, this addess should co
ntain instruction
                #5 Read <= 1;
                     #5 MDRin <= 1;

            end
            T2: begin
                Read <= 0;
                MARin <= 0;
                MDRin <= 0;
                //load MDR value to bus which contains instruction, instruction i
s then stored in IR
                MDRout <= 1; IRin <= 1;
            end

            T3: begin
                     MDRout <= 0;
                     IRin <= 0;

                Gra <= 1;
```

```
                Rout <= 1;
                PCin <= 1;
                     BAout <= 1;
            end

        endcase
    end
endmodule
```



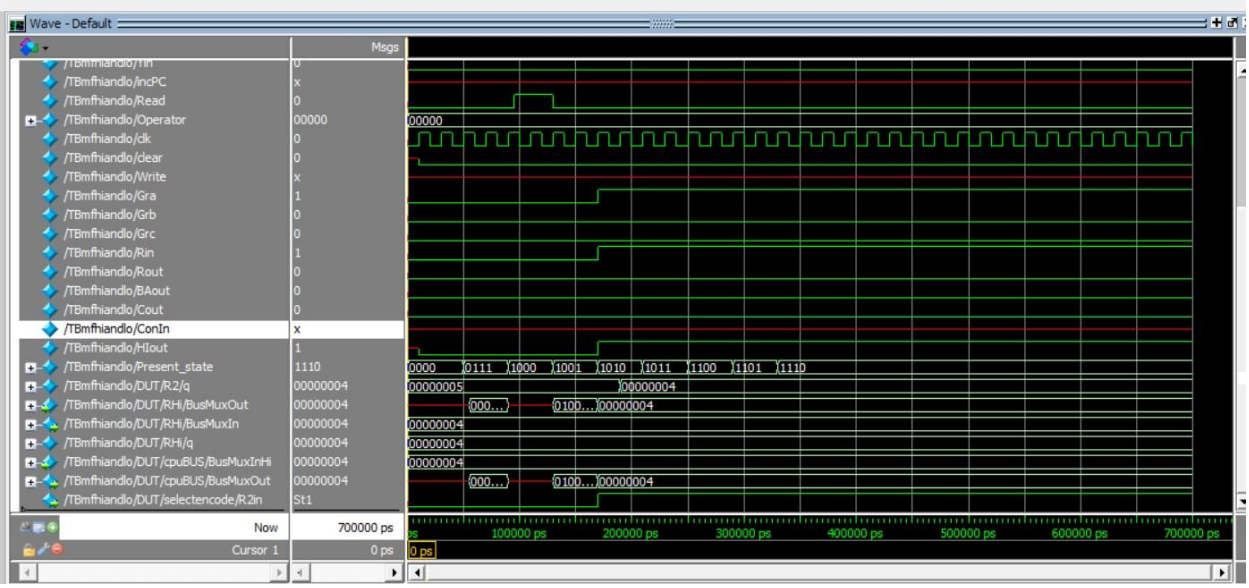
Mfhi & mflo

Here we are verifying the functionality by simulating the control sequences for mfhi R1 and mflo R1. Note that R1 was preloaded with the value 5.

```
`timescale 1ns/10ps

module TBmfhiandlo;
    reg PCout, Zlowout, MDRout, R2out, R4out;// add any other signals to see in y
our simulation
    reg MARin, Zin, PCin, MDRin, IRin, Yin;
    reg incPC, Read;
     reg [4:0] Operator;
    reg clk;
     reg clear,Write;
    reg Gra, Grb, Grc, Rin, Rout, BAout, Cout, ConIn, HIout;

    parameter   Default = 4'b0000, T0= 4'b0111,T1= 4'b1000,T2= 4'b1001, T3= 4'b10
10, T4= 4'b1011, T5= 4'b1100,T6= 4'b1101,T7= 4'b1110;
```

```
    reg[3:0] Present_state= Default;

DataPath DUT(PCout, Zlowout, MDRout, MARin, Zin, PCin, MDRin, IRin, Yin, Read, Wr
ite, Operator, clk, clear,
Gra, Grb, Grc, Rin, Rout, BAout, Cout, ConIn, HIout);


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


always @(posedge clk)     //finite state machine; if clk rising-edge
    begin
        case (Present_state)
            Default     :   #40 Present_state = T0;
            T0          :   #40 Present_state = T1;
            T1          :   #40 Present_state = T2;
            T2          :   #40 Present_state = T3;
            T3          :   #40 Present_state = T4;
            T4          :   #40 Present_state = T5;
            T5          :   #40 Present_state = T6;
            T6          :   #40 Present_state = T7;
        endcase
    end

always @(Present_state)     // do the required job ineach state
    begin
        case (Present_state)              //assert the required signals in each c
lk cycle
            Default: begin
                PCout <= 0;   Zlowout <= 0;   MDRout<= 0;   //initialize the sign
als
                MARin <= 0;   Zin <= 0;
                PCin <=0;   MDRin <= 0;   IRin  <= 0;   Yin <= 0;
                //IncPC <= 0;
                     Read <= 0;   Operator <= 5'b00000;
                Gra<= 0; Grb<= 0; Grc<= 0; Rin<= 0; Rout<= 0; BAout<= 0; Cout<=0;
                     #10 clear <= 0; HIout <= 0;
            end

            T0: begin
                //load value in PC register (0) in MAR
                #5 PCout <= 1;
```

```
                    #5 MARin <= 1;  //IncPC <= 1; Zin <= 1; used to inc PC by 4
            end
            T1: begin
                //Zlowout <= 1; PCin <= 1;
                PCout <= 0;
                     MARin <= 0;
                //MDR will grab value from ram @ address 0, this addess should co
ntain instruction
                #5 Read <= 1;
                     #5 MDRin <= 1;

            end
            T2: begin
                Read <= 0;
                MARin <= 0;
                MDRin <= 0;
                //load MDR value to bus which contains instruction, instruction i
s then stored in IR
                MDRout<= 1; IRin <= 1;
            end

            T3: begin
                     MDRout <= 0;
                     IRin <= 0;
                Read <= 0;
                MDRin <= 0;
                //load value in MDR on Bus
                     HIout <= 1;
                //set R1in signal to 1, value that was in memory at address 85 sh
ould now be stored in R1
                Gra <= 1;  //RA in Selectencode should be 0001 in binary
                Rin <= 1;
            end
        endcase
    end
endmodule
```



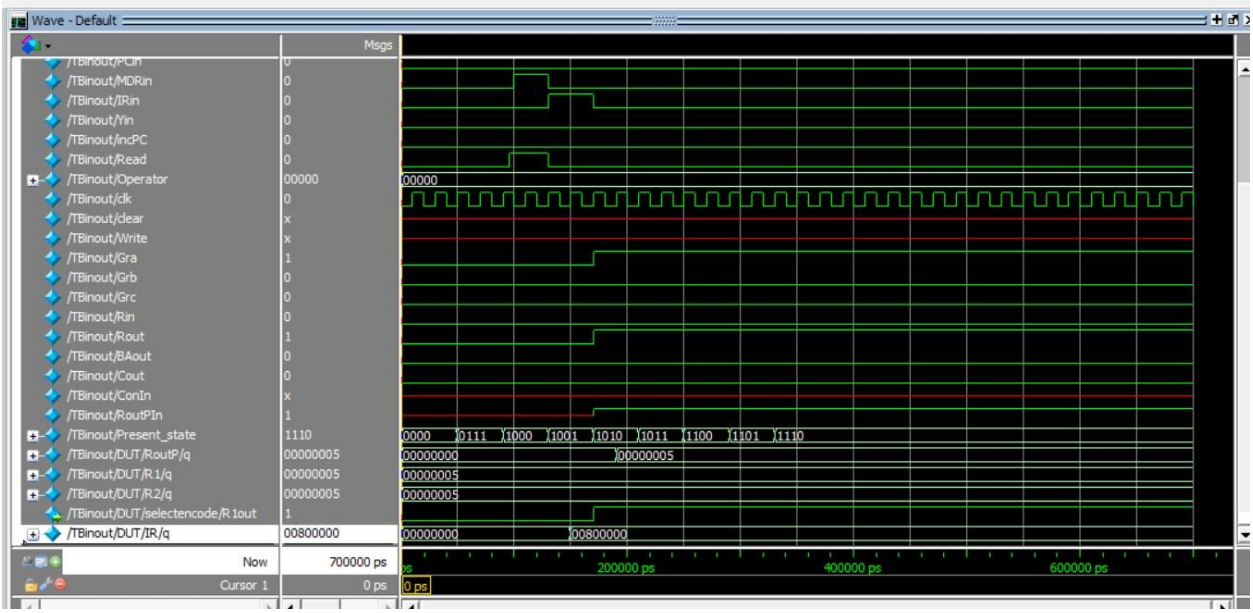
In/Out

Here we are verifying the functionality of our Output Port logic by simulating the control sequence for out R1. Note that R1 was preloaded with the value 5.

```
`timescale 1ns/10ps

module TBinout;
    reg PCout, Zlowout, MDRout, R2out, R4out;// add any other signals to see in y
our simulation
    reg MARin, Zin, PCin, MDRin, IRin, Yin;
    reg incPC, Read;
     reg [4:0] Operator;
    reg clk;
     reg clear,Write;
    reg Gra, Grb, Grc, Rin, Rout, BAout, Cout, ConIn, RoutPIn;

    parameter   Default = 4'b0000, T0= 4'b0111,T1= 4'b1000,T2= 4'b1001, T3= 4'b10
10, T4= 4'b1011, T5= 4'b1100,T6= 4'b1101,T7= 4'b1110;
    reg[3:0] Present_state= Default;

DataPath DUT(PCout, Zlowout, MDRout, MARin, Zin, PCin, MDRin, IRin, Yin, Read, Wr
ite, Operator, clk, clear,
Gra, Grb, Grc, Rin, Rout, BAout, Cout, ConIn, RoutPIn);


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

```
always @(posedge clk)     //finite state machine; if clk rising-edge
    begin
        case (Present_state)
            Default     :   #40 Present_state = T0;
            T0          :   #40 Present_state = T1;
            T1          :   #40 Present_state = T2;
            T2          :   #40 Present_state = T3;
            T3          :   #40 Present_state = T4;
            T4          :   #40 Present_state = T5;
            T5          :   #40 Present_state = T6;
            T6          :   #40 Present_state = T7;
        endcase
    end

always @(Present_state)     // do the required job ineach state
    begin
        case (Present_state)             //assert the required signals in each c
lk cycle
            Default: begin
                PCout <= 0;   Zlowout <= 0;   MDRout<= 0;   //initialize the sign
als
                MARin <= 0;   Zin <= 0;
                PCin <=0;   MDRin <= 0;   IRin  <= 0;   Yin <= 0;
                incPC <= 0;   Read <= 0;   Operator <= 5'b00000;
                Gra<= 0; Grb<= 0; Grc<= 0; Rin<= 0; Rout<= 0; BAout<= 0; Cout<=0;
                // clear <= 1; //initialize registers to 0
                     // #10 clear <= 0;
            end

            T0: begin
                //load value in PC register (0) in MAR
                #5 PCout <= 1;
                     #5 MARin <= 1;  //IncPC <= 1; Zin <= 1; used to inc PC by 4
            end
            T1: begin
                //Zlowout <= 1; PCin <= 1;
                PCout <= 0;
                //MDR will grab value from ram @ address 0, this addess should co
ntain instruction
                #5 Read <= 1;
                     #5 MDRin <= 1;

            end
```

```
            T2: begin
                Read <= 0;
                MARin <= 0;
                MDRin <= 0;
                //load MDR value to bus which contains instruction, instruction i
s then stored in IR
                MDRout<= 1; IRin <= 1;
            end

            T3: begin
                     MDRout <= 0;
                     IRin <= 0;

                Gra <= 1;
                Rout <= 1;
                RoutPIn <= 1;
            end
        endcase
    end
endmodule
```



Addi instruction – R2, R1, -5

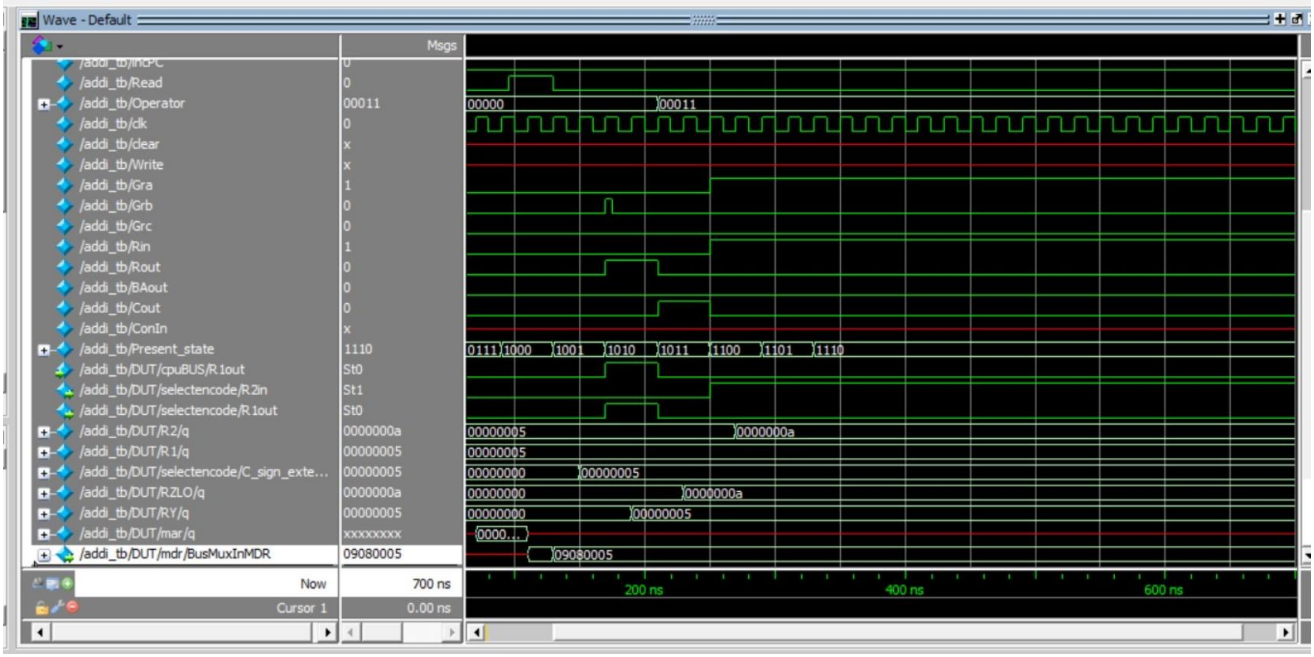
This instruction verifies that the design can subtract 5 from the contents of R1 and store the result in R2. First the instruction into IR (T0-T2), then the add operation is performed. The result is stored in RZLo and then moved into the destination register, R2.

```
// Op-code: 31-27, Ra: 26-23, C2: 22-19, C: 18-0
```

```
`timescale 1ns/10ps

module addi_tb;
    reg PCout, Zlowout, MDRout; // add any other signals to see in your
simulation
    reg MARin, Zin, PCin, MDRin, IRin, Yin;
    reg incPC, Read;
     reg [4:0] Operator;
    reg clk;
     reg clear, Write;
    reg Gra, Grb, Grc, Rin, Rout, BAout, Cout, ConIn;

    parameter   Default = 4'b0000, T0= 4'b0111,T1= 4'b1000,T2= 4'b1001, T3=
4'b1010, T4= 4'b1011, T5= 4'b1100,T6= 4'b1101,T7= 4'b1110;
    reg[3:0] Present_state= Default;

DataPath DUT(PCout, Zlowout, MDRout, MARin, Zin, PCin, MDRin, IRin, Yin, Read,
Write, Operator, clk, clear,
Gra, Grb, Grc, Rin, Rout, BAout,Cout, ConIn);

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

always @(posedge clk)     //finite state machine; if clk rising-edge
    begin
        case (Present_state)
            Default     :   #40 Present_state = T0;
            T0          :   #40 Present_state = T1;
            T1          :   #40 Present_state = T2;
            T2          :   #40 Present_state = T3;
            T3          :   #40 Present_state = T4;
            T4          :   #40 Present_state = T5;
            T5          :   #40 Present_state = T6;
            T6          :   #40 Present_state = T7;
        endcase
```

```
    end

always @(Present_state)     // do the required job ineach state
    begin
        case (Present_state)              //assert the required signals in each
clk cycle
            Default: begin
                PCout <= 0;   Zlowout <= 0;   MDRout<= 0;   //initialize the
signals
                MARin <= 0;   Zin <= 0;
                PCin <=0;   MDRin <= 0;   IRin  <= 0;   Yin <= 0;
                incPC <= 0;   Read <= 0;   Operator <= 5'b00000;
                Gra <= 0; Grb <= 0; Grc <= 0; Rin <= 0; Rout <= 0; BAout <= 0;
Cout <=0;
                // clear <= 1; //initialize registers to 0
                    // #10 clear <= 0;
            end
            T0: begin
                //load value in PC register (0) in MAR
                #5 PCout <= 1;
                    #5 MARin <= 1;  //IncPC <= 1; Zin <= 1; used to inc PC by 4
                // incPC <= 1;
                    // Zin <= 1;
            end
            T1: begin
                // Zlowout <= 1; PCin <= 1;
                PCout <= 0;
                //MDR will grab value from ram @ address 0, this addess should
contain instruction
                #5 Read <= 1; // Mdatain <= 1;
                    #5 MDRin <= 1;

            end
            T2: begin
                Read <= 0;
                MARin <= 0;
                MDRin <= 0;
                //load MDR value to bus which contains instruction, instruction
is then stored in IR
                MDRout<= 1; IRin <= 1;
            end
            T3: begin
                MDRout<= 0;
                IRin <= 0;
                //select R0out since Rb = 0000
```

```
                Grb <= 1;
                //set to 1 to generate R0out = 1 and store value of R0 (0) into Y
register
                Rout <= 1;
                Yin <= 1;
                    #5 Grb <= 0;
            end

            T4: begin
                    Rout <= 0;
                Yin <= 0;
                //store C sign extended value on bus (should be 85 decimal)
                Cout <= 1;
                //perform add between 85/BuxMuxOut + R0 (0)/BuxMuxInY
                    Operator <= 5'b00011; // Add
                //store result in Z register
                Zin <=1;


            end

            //read from ram
            T5: begin
                    Cout <= 0;
                Zin <=0;
                    // Grb <= 0;


                //store Z register contents (85) in bus
                Zlowout <= 1;
                Gra <= 1;
                //load 85/busMuxOut into MAR to get address of 85
                Rin <= 1;

            end
        endcase
    end
endmodule
```



Andi instruction – andi R2, R1, $26

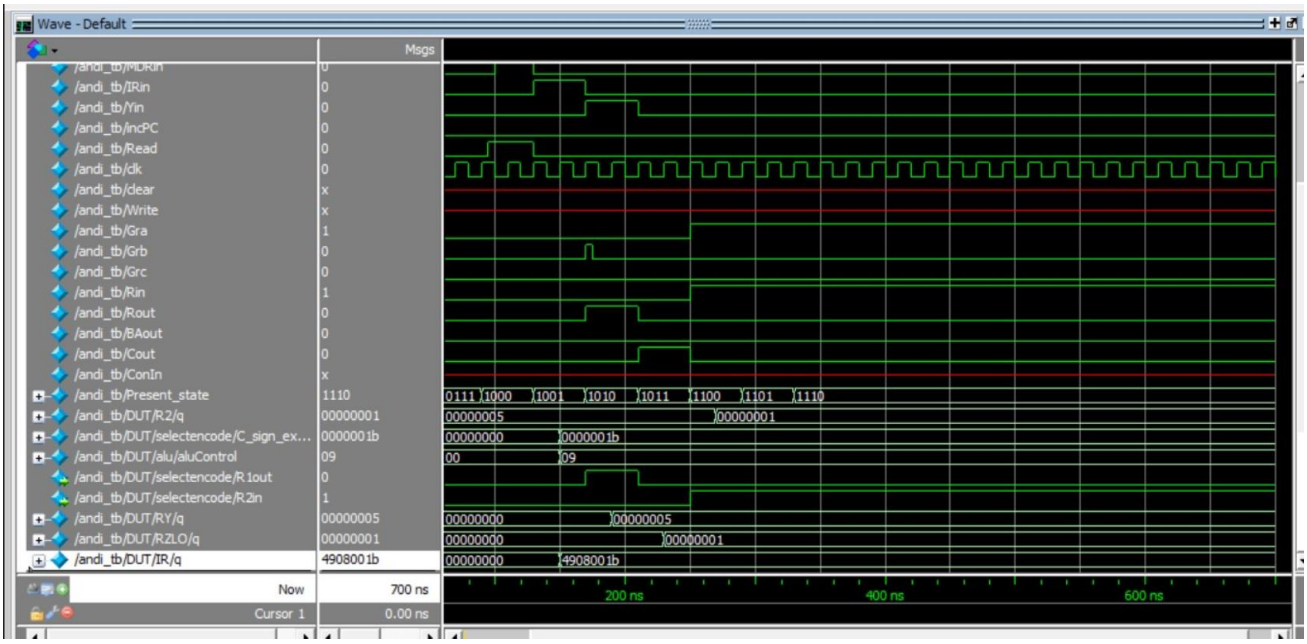
This instruction shows that the design can perform an and immediate operation on $26 and the value of R1, then subsequently store the result in R2. The steps T0-T2 are executed first, performing the fetch instruction. The andi immediate instruction is performed and stored in the RZLo and moved to the appropriate destination register, R2.

```
`timescale 1ns/10ps


module andi_tb;
    reg PCout, Zlowout, MDRout; // add any other signals to see in your
simulation
    reg MARin, Zin, PCin, MDRin, IRin, Yin;
    reg incPC, Read;
     reg [4:0] Operator;
    reg clk;
     reg clear, Write;
    reg Gra, Grb, Grc, Rin, Rout, BAout, Cout, ConIn;


    parameter   Default = 4'b0000, T0= 4'b0111,T1= 4'b1000,T2= 4'b1001, T3=
4'b1010, T4= 4'b1011, T5= 4'b1100,T6= 4'b1101,T7= 4'b1110;
    reg[3:0] Present_state= Default;


DataPath DUT(PCout, Zlowout, MDRout, MARin, Zin, PCin, MDRin, IRin, Yin, Read,
Write, Operator, clk, clear,
```

```
Gra, Grb, Grc, Rin, Rout, BAout,Cout, ConIn);


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


always @(posedge clk)     //finite state machine; if clk rising-edge
    begin
        case (Present_state)
            Default     :   #40 Present_state = T0;
            T0          :   #40 Present_state = T1;
            T1          :   #40 Present_state = T2;
            T2          :   #40 Present_state = T3;
            T3          :   #40 Present_state = T4;
            T4          :   #40 Present_state = T5;
            T5          :   #40 Present_state = T6;
            T6          :   #40 Present_state = T7;
        endcase
    end

always @(Present_state)     // do the required job ineach state
    begin
        case (Present_state)              //assert the required signals in each
clk cycle
            Default: begin
                PCout <= 0;   Zlowout <= 0;   MDRout<= 0;   //initialize the
signals
                MARin <= 0;   Zin <= 0;
                PCin <=0;   MDRin <= 0;   IRin  <= 0;   Yin <= 0;
                incPC <= 0;   Read <= 0;   Operator <= 5'b00000;
                Gra <= 0; Grb <= 0; Grc <= 0; Rin <= 0; Rout <= 0; BAout <= 0;
Cout <=0;
                // clear <= 1; //initialize registers to 0
                    // #10 clear <= 0;
            end
            T0: begin
                //load value in PC register (0) in MAR
                #5 PCout <= 1;
                    #5 MARin <= 1;  //IncPC <= 1; Zin <= 1; used to inc PC by 4
                // incPC <= 1;
                    // Zin <= 1;
```

```
            end
            T1: begin
                // Zlowout <= 1; PCin <= 1;
                PCout <= 0;
                //MDR will grab value from ram @ address 0, this addess should
contain instruction
                #5 Read <= 1; // Mdatain <= 1;
                    #5 MDRin <= 1;

            end
            T2: begin
                Read <= 0;
                MARin <= 0;
                MDRin <= 0;
                //load MDR value to bus which contains instruction, instruction
is then stored in IR
                MDRout<= 1; IRin <= 1;
            end
            T3: begin
                MDRout<= 0;
                IRin <= 0;
                //select R0out since Rb = 0000
                Grb <= 1;
                //set to 1 to generate R0out = 1 and store value of R0 (0) into Y
register
                Rout <= 1;
                Yin <= 1;
                    #5 Grb <= 0;
            end

            T4: begin
                    Rout <= 0;
                Yin <= 0;
                //store C sign extended value on bus (should be 85 decimal)
                Cout <= 1;
                //perform add between 85/BuxMuxOut + R0 (0)/BuxMuxInY
                    Operator <= 5'b00011; // Add
                //store result in Z register
                Zin <=1;


            end


            //read from ram
```

```
            T5: begin
                    Cout <= 0;
                Zin <=0;
                    // Grb <= 0;


                //store Z register contents (85) in bus
                Zlowout <= 1;
                Gra <= 1;
                //load 85/busMuxOut into MAR to get address of 85
                Rin <= 1;

            end
        endcase
    end
endmodule
```



Ori instruction – ori R2, R1, $26

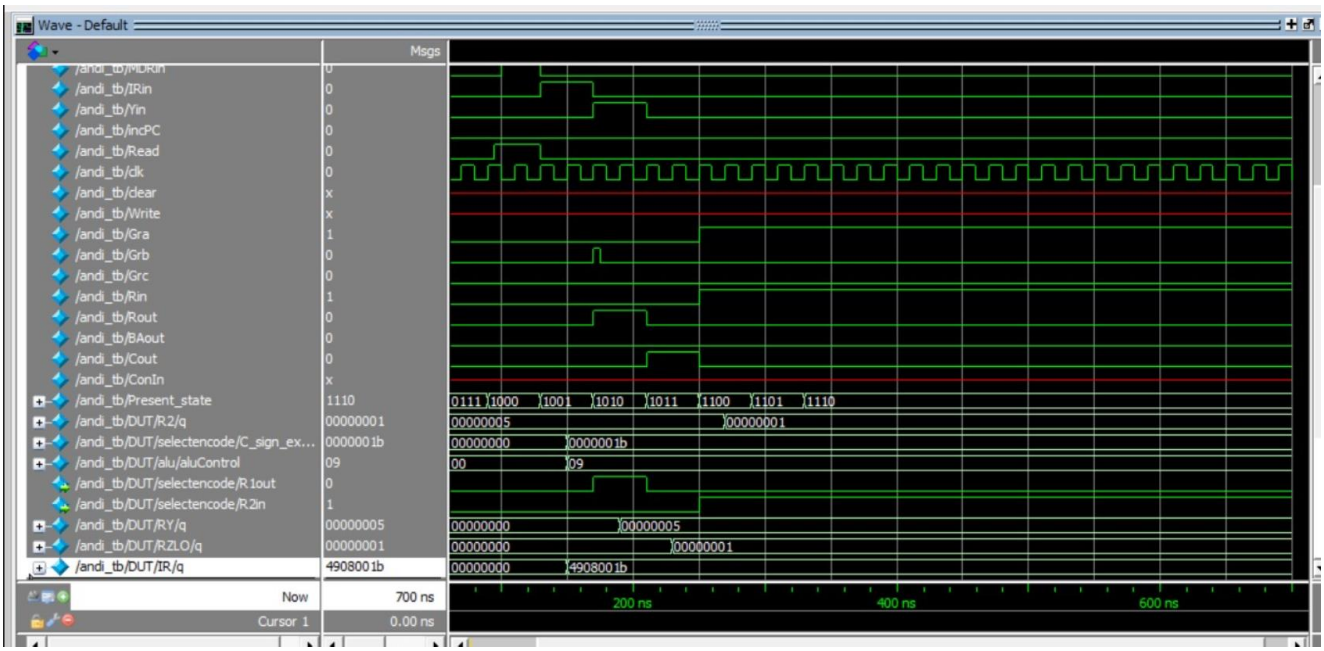
This instruction shows that the design can perform an or immediate operation on $26 and the value of R1, then subsequently store the result in R2. The steps T0-T2 are executed first, performing the fetch instruction. The or immediate instruction is performed and stored in the RZLo and moved to the appropriate destination register, R2.

```
`timescale 1ns/10ps


module ori_tb;
```

```
    reg PCout, Zlowout, MDRout; // add any other signals to see in your
simulation
    reg MARin, Zin, PCin, MDRin, IRin, Yin;
    reg incPC, Read;
     reg [4:0] Operator;
    reg clk;
     reg clear, Write;
    reg Gra, Grb, Grc, Rin, Rout, BAout, Cout, ConIn;


    parameter   Default = 4'b0000, T0= 4'b0111,T1= 4'b1000,T2= 4'b1001, T3=
4'b1010, T4= 4'b1011, T5= 4'b1100,T6= 4'b1101,T7= 4'b1110;
    reg[3:0] Present_state= Default;


DataPath DUT(PCout, Zlowout, MDRout, MARin, Zin, PCin, MDRin, IRin, Yin, Read,
Write, Operator, clk, clear,
Gra, Grb, Grc, Rin, Rout, BAout,Cout, ConIn);


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


always @(posedge clk)     //finite state machine; if clk rising-edge
    begin
        case (Present_state)
            Default     :   #40 Present_state = T0;
            T0          :   #40 Present_state = T1;
            T1          :   #40 Present_state = T2;
            T2          :   #40 Present_state = T3;
            T3          :   #40 Present_state = T4;
            T4          :   #40 Present_state = T5;
            T5          :   #40 Present_state = T6;
            T6          :   #40 Present_state = T7;
        endcase
    end

always @(Present_state)     // do the required job ineach state
    begin
        case (Present_state)              //assert the required signals in each
clk cycle
```

```
            Default: begin
                PCout <= 0;   Zlowout <= 0;   MDRout<= 0;   //initialize the
signals
                MARin <= 0;   Zin <= 0;
                PCin <=0;   MDRin <= 0;   IRin  <= 0;   Yin <= 0;
                incPC <= 0;   Read <= 0;   Operator <= 5'b00000;
                Gra <= 0; Grb <= 0; Grc <= 0; Rin <= 0; Rout <= 0; BAout <= 0;
Cout <=0;
                // clear <= 1; //initialize registers to 0
                    // #10 clear <= 0;
            end
            T0: begin
                //load value in PC register (0) in MAR
                #5 PCout <= 1;
                    #5 MARin <= 1;  //IncPC <= 1; Zin <= 1; used to inc PC by 4
                // incPC <= 1;
                    // Zin <= 1;
            end
            T1: begin
                // Zlowout <= 1; PCin <= 1;
                PCout <= 0;
                //MDR will grab value from ram @ address 0, this addess should
contain instruction
                #5 Read <= 1; // Mdatain <= 1;
                    #5 MDRin <= 1;

            end
            T2: begin
                Read <= 0;
                MARin <= 0;
                MDRin <= 0;
                //load MDR value to bus which contains instruction, instruction
is then stored in IR
                MDRout<= 1; IRin <= 1;
            end
            T3: begin
                MDRout<= 0;
                IRin <= 0;
                //select R0out since Rb = 0000
                Grb <= 1;
                //set to 1 to generate R0out = 1 and store value of R0 (0) into Y
register
                Rout <= 1;
                Yin <= 1;
                    #5 Grb <= 0;
```

```
            end

            T4: begin
                    Rout <= 0;
                Yin <= 0;
                //store C sign extended value on bus (should be 85 decimal)
                Cout <= 1;
                //perform add between 85/BuxMuxOut + R0 (0)/BuxMuxInY
                    Operator <= 5'b00011; // Add
                //store result in Z register
                Zin <=1;


            end

            //read from ram
            T5: begin
                    Cout <= 0;
                Zin <=0;
                    // Grb <= 0;

                //store Z register contents (85) in bus
                Zlowout <= 1;
                Gra <= 1;
                //load 85/busMuxOut into MAR to get address of 85
                Rin <= 1;

            end
        endcase
    end
endmodule
```



Brzr instruction – brzr R2, 35

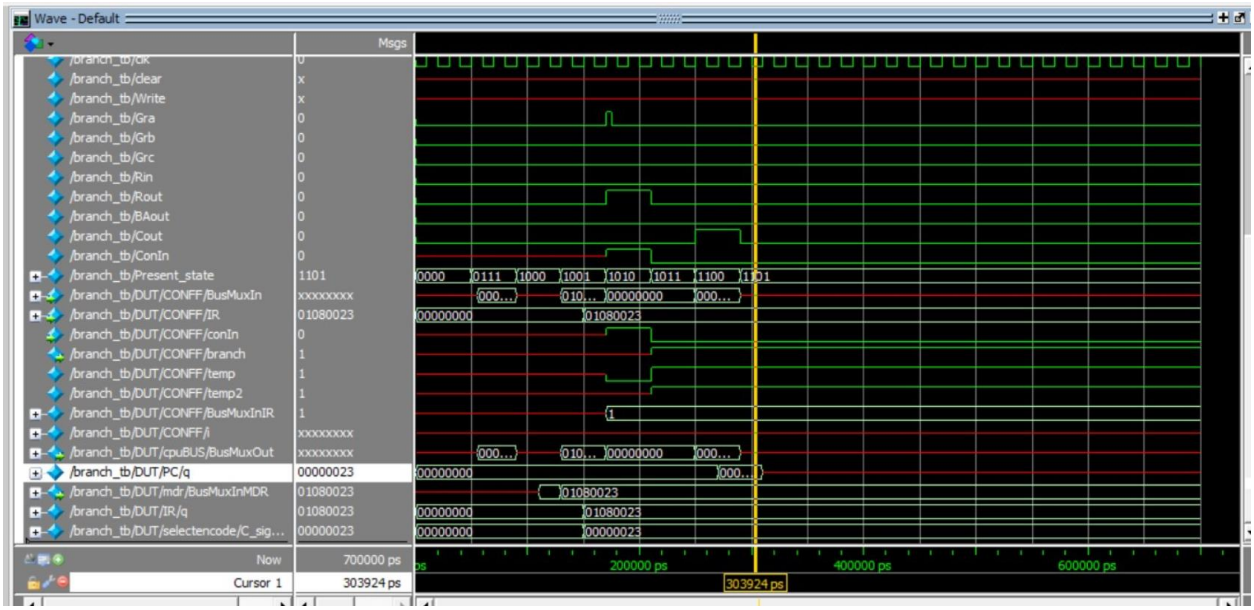
This instruction shows that the design can perform a brzr branch (branch if zero) operation on 35 and the value of R2. The steps T0-T2 are executed first, performing the fetch instruction. Then, an operation is performed to determine whether or not to branch. If required, the branch is then performed.

```
// make tbs for:
// branch instructions: brzr, brnz, brpl, brmi
// Op-code: 31-27, Ra: 26-23, C2: 22-19, C: 18-0
// brzr: 0108 0023, C2: 0001
// 0 0000, 0010, 0001, 000 0000 0000 0010 0011
// brnz: 0110 0023, C2: 0010
// 0 0000, 0010, 0010, 000 0000 0000 0010 0011
// brpl: 0120 0023, C2: 0100
// brmi: 0140 0023, C2: 1000


`timescale 1ns/10ps


module branch_tb;
    reg PCout, Zlowout, MDRout, R2out, R4out;// add any other signals to see in
your simulation
    reg MARin, Zin, PCin, MDRin, IRin, Yin;
    reg Read;
     reg [4:0] Operator;
    reg clk;
```

```
    reg clear, Write;
    reg Gra, Grb, Grc, Rin, Rout, BAout, Cout;
    reg ConIn;

    parameter   Default = 4'b0000, T0= 4'b0111, T1= 4'b1000, T2= 4'b1001, T3=
4'b1010, T4= 4'b1011, T5= 4'b1100, T6= 4'b1101;
    reg[3:0] Present_state= Default;

DataPath DUT(PCout, Zlowout, MDRout, MARin, Zin, PCin, MDRin, IRin, Yin, Read,
Write, Operator, clk, clear,
Gra, Grb, Grc, Rin, Rout, BAout, Cout, ConIn);

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

always @(posedge clk)     //finite state machine; if clk rising-edge
    begin
        case (Present_state)
            Default     :   #40 Present_state = T0;
            T0          :   #40 Present_state = T1;
            T1          :   #40 Present_state = T2;
            T2          :   #40 Present_state = T3;
            T3          :   #40 Present_state = T4;
            T4          :   #40 Present_state = T5;
            T5          :   #40 Present_state = T6;
        endcase
    end

always @(Present_state)     // do the required job ineach state
    begin
        case (Present_state)              //assert the required signals in each
clk cycle
            Default: begin
                PCout <= 0;   Zlowout <= 0;   MDRout<= 0;   //initialize the
signals
                MARin <= 0;   Zin <= 0;
                PCin <=0;   MDRin <= 0;   IRin  <= 0;   Yin <= 0;
                // incPC <= 0;
```

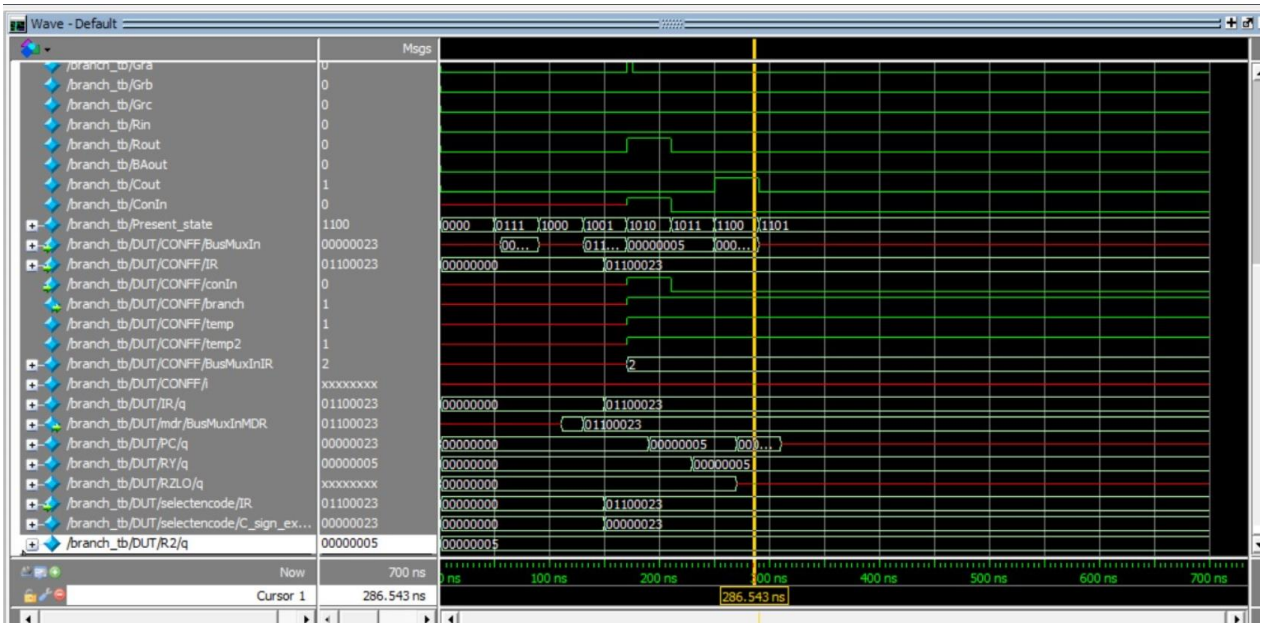
```
                Read <= 0;   Operator <= 5'b00000;
            Gra<= 0; Grb<= 0; Grc<= 0; Rin<= 0; Rout<= 0; BAout<= 0; Cout<=0;
            // clear <= 1; //initialize registers to 0
                //#10 clear <= 0;
        end

        T0: begin
            //load value in PC register (0) in MAR
            #5 PCout <= 1;
                #5 MARin <= 1;  //IncPC <= 1; Zin <= 1; used to inc PC by 4
            // incPC <= 1;
                // Zin <= 1;
        end
        T1: begin
            // Zlowout <= 1; PCin <= 1;
            PCout <= 0;
            //MDR will grab value from ram @ address 0, this addess should
contain instruction
            #5 Read <= 1; // Mdatain <= 1;
                #5 MDRin <= 1;

        end
        T2: begin
            Read <= 0;
            MARin <= 0;
            MDRin <= 0;
            //load MDR value to bus which contains instruction, instruction
is then stored in IR
            MDRout<= 1; IRin <= 1;
        end


        T3: begin
            MDRout<= 0;
            IRin <= 0;

            Gra <= 1;
            Rout <= 1;
            ConIn <= 1;
            //set to 1 to generate R0out = 1 and store value of R0 (0) into Y
register
                #5 Gra <= 0;
        end

        T4: begin
```

```
                Rout <= 0;
                ConIn <= 0;

            PCout <= 1; Yin <= 1;
        end

        T5: begin
            PCout <= 0; Yin <= 0;

                Cout <= 1;
            Operator <= 5'b00011; // ADD
                Zin <= 1;

            // store Z register contents (85) in bus
            // Zlowout <= 1;
            // load 85/busMuxOut into MAR to get address of 85
            // MARin <= 1;
        end

        T6: begin
            Cout <= 0;
            Zin <= 0;

            Zlowout <= 1;
            // PC <= Con;
            // read from ram at address 85 and load that value in MDR -> we
will need to have a preset value in memory at address 85
                // Read <= 1;
            // MDRin <= 1;

        end

    endcase
  end
endmodule
```



Brnz instruction – brnz R2, 35

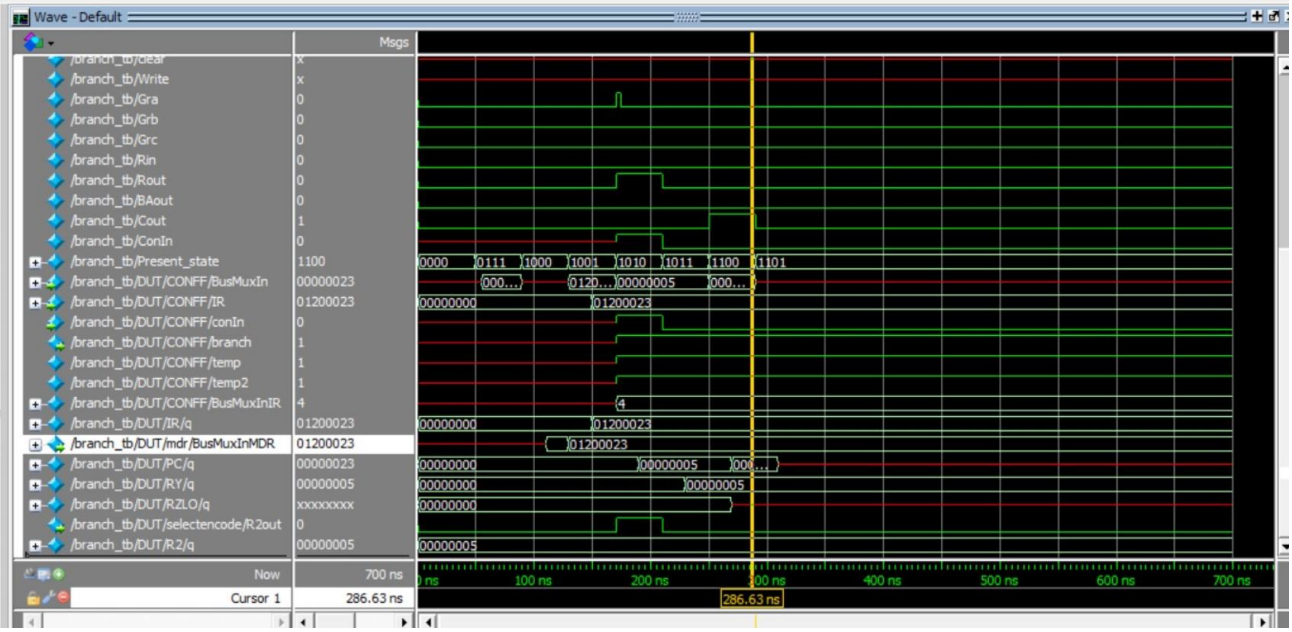
This instruction shows that the design can perform a brnz branch (branch if non-zero) operation on 35 and the value of R2. The steps T0-T2 are executed first, performing the fetch instruction. Then, an operation is performed to determine whether or not to branch. If required, the branch is then performed. Refer to the brzr instruction for the testbench code. The same testbench was used for all branch operations, only the operation code was changed for each.



Brpl instruction – brpl R2, 35

This instruction shows that the design can perform a brpl branch (branch if positive) operation on 35 and the value of R2. The steps T0-T2 are executed first, performing the fetch instruction.

Then, an operation is performed to determine whether or not to branch. If required, the branch is then performed. Refer to the brzr instruction for the testbench code. The same testbench was used for all branch operations, only the operation code was changed for each.



Brmi instruction – brmi R2, 35

This instruction shows that the design can perform a brmi branch (branch if negative) operation on 35 and the value of R2. The steps T0-T2 are executed first, performing the fetch instruction. Then, an operation is performed to determine whether or not to branch. If required, the branch is then performed. Refer to the brzr instruction for the testbench code. The same testbench was used for all branch operations, only the operation code was changed for each.