ELEC 374: MiniSRC CPU Project

Phase 2 Report

Group 4

Jacob Chisholm (20335775)

Hendrix Gryspeerdt (21hgg3)

Luke Strickland (21laps1)

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

Table of Contents

Test Benches

All test benches were simulated using an instruction decode unit. This allows the simulations of several instructions to be sequenced and easily simulated using a series of verilog define macros. The test bench code is shown below:

```
`include "ISA.vh"
`include "ALU.vh"


module Control (
    // Clock, reset and ready signals
    // Ready is an active high that allows the next step to continue
    iClk, nRst, iRdy,
    // Memory Signals/Control
    iMemData, oMemRead, oMemWrite,
    // Pipe Control
    oPipe_nRst,
    // Program Counter Control
    oPC_nRst, oPC_en, oPC_tmpEn, oPC_load, oPC_offset,
    // Register File Control
    oRF_Write,
    oRF_AddrA, oRF_AddrB, oRF_AddrC,
    // Write Back Register Control
    oRWB_en,
    // ALU Control
    oALU_Ctrl, oRA_en, oRB_en,
    oRZH_en, oRZL_en, oRAS_en,
    // Jump Feedback
    iJ_zero, iJ_nZero, iJ_pos, iJ_neg,
    // Port Register enable
    oREP_en,
    // Multiplexers
    oMUX_BIS, oMUX_RZHS, oMUX_WBM, oMUX_MAP, oMUX_ASS, oMUX_WBP, oMUX_WBE,
    // Imm32 Output
    oImm32
);

// Clock, reset and ready signals
// Ready is an active high that allows the next step to continue
input wire iClk, nRst, iRdy;
// Memory Signals/Control
```

```
input wire [31:0] iMemData;
output wire oMemRead, oMemWrite;
// Pipe Control
output wire oPipe_nRst;
// Program Counter Control
output wire oPC_nRst, oPC_en, oPC_tmpEn, oPC_load, oPC_offset;
// Register File Control
output wire oRF_Write;
output wire [3:0] oRF_AddrA, oRF_AddrB, oRF_AddrC;
// Write Back Register Control
output wire oRWB_en;
// ALU Control
output wire [3:0] oALU_Ctrl;
output wire oRA_en, oRB_en;
output wire oRZH_en, oRZL_en, oRAS_en;
// Jump Feedback
input wire iJ_zero, iJ_nZero, iJ_pos, iJ_neg;
// External Port Enable
output wire oREP_en;
// Multiplexers
output wire oMUX_BIS, oMUX_RZHS, oMUX_WBM, oMUX_MAP, oMUX_ASS, oMUX_WBP,
oMUX_WBE;
// Imm32 Output
output wire [31:0] oImm32;

// Step Counter
reg [5:1] Cycle;

// IR
wire IR_en;
wire [31:0] IR_out;

// Decoder IO
wire [3:0] ID_RA, ID_RB, ID_RC;
wire [4:0] ID_OpCode;
wire [31:0] ID_imm32, ID_BRD;
wire [1:0] ID_BRC;

// OpCode R-Format Wires
wire OP_LD, OP_LI,  OP_ST,  OP_ADD, OP_SUB, OP_AND,
```

```
    OP_OR, OP_ROR, OP_ROL, OP_SRL, OP_SRA, OP_SLL;
// OpCode I-Format Wires
wire OP_ADDI, OP_ANDI, OP_ORI, OP_DIV, OP_MUL, OP_NEG, OP_NOT;
// OpCode B-Format Wires
wire OP_BRx;
// OpCode J-Format Wires
wire OP_JAL, OP_JFR, OP_IN, OP_OUT, OP_MFL, OP_MFH;
// OpCode M-Format Wires
wire OP_NOP, OP_HLT;
// OpCode Format Wires
// (Useful for data path MUX Assignments)
wire OPF_R, OPF_I, OPF_B, OPF_J, OPF_M;
// Branch Conditional Wires
wire BR_ZERO, BR_NZRO, BR_POS, BR_NEG;
wire BR_TRUE;

// Assign Cycle
always @(posedge iClk or negedge nRst)
begin
    if(!nRst)
        Cycle = 5'b00001;
    else begin
        if(iRdy) Cycle = {Cycle[4:1], Cycle[5]};
    end
end

// Instruction Register
assign IR_en = Cycle[1];
REG32 IR(.iClk(iClk), .nRst(nRst), .iEn(IR_en), .iD(iMemData),
.oQ(IR_out));

// Decoder
Decode decoder(
    .iINS(IR_out),
    .oImm32(ID_imm32),
    .oRa(ID_RA),
    .oRb(ID_RB),
    .oRc(ID_RC),
    .oCode(ID_OpCode),
    // Branch Distance
```

```
    .oBRD(ID_BRD),
    // Branch Code
    .oBRC(ID_BRC)
);

// Assign OP-Code Types

// Assign R-Format Wires
assign OP_ADD = (ID_OpCode == `ISA_ADD);
assign OP_SUB = (ID_OpCode == `ISA_SUB);
assign OP_AND = (ID_OpCode == `ISA_AND);
assign OP_OR  = (ID_OpCode == `ISA_OR);
assign OP_ROR = (ID_OpCode == `ISA_ROR);
assign OP_ROL = (ID_OpCode == `ISA_ROL);
assign OP_SRL = (ID_OpCode == `ISA_SRL);
assign OP_SRA = (ID_OpCode == `ISA_SRA);
assign OP_SLL = (ID_OpCode == `ISA_SLL);
// Opcode Format Wire (Useful for data path MUX Assignments)
assign OPF_R  = (OP_ADD || OP_SUB || OP_AND || OP_OR || OP_ROR || OP_ROL
|| OP_SRL || OP_SRA || OP_SLL);
// Assign I-Format Wires
assign OP_LD  = (ID_OpCode == `ISA_LD);
assign OP_LI  = (ID_OpCode == `ISA_LI);
assign OP_ST  = (ID_OpCode == `ISA_ST);
assign OP_ADDI = (ID_OpCode == `ISA_ADDI);
assign OP_ANDI = (ID_OpCode == `ISA_ANDI);
assign OP_ORI  = (ID_OpCode == `ISA_ORI);
assign OP_DIV  = (ID_OpCode == `ISA_DIV);
assign OP_MUL  = (ID_OpCode == `ISA_MUL);
assign OP_NEG  = (ID_OpCode == `ISA_NEG);
assign OP_NOT  = (ID_OpCode == `ISA_NOT);
// Opcode Format Wire (Useful for data path MUX Assignments)
assign OPF_I   = (OP_LD || OP_LI || OP_ST || OP_ADDI || OP_ANDI || OP_ORI
|| OP_DIV || OP_MUL || OP_NEG || OP_NOT);
// Assign B-Format Wires
assign OP_BRx = (ID_OpCode == `ISA_BRx);
// Opcode Format Wire (Useful for data path MUX Assignments)
assign OPF_B = OP_BRx;
// Assign J-Format Wires
assign OP_JAL = (ID_OpCode == `ISA_JAL);
```

```
assign OP_JFR = (ID_OpCode == `ISA_JFR);
assign OP_IN  = (ID_OpCode == `ISA_IN);
assign OP_OUT = (ID_OpCode == `ISA_OUT);
assign OP_MFL = (ID_OpCode == `ISA_MFL);
assign OP_MFH = (ID_OpCode == `ISA_MFH);
// Opcode Format Wire (Useful for data path MUX Assignments)
assign OPF_J  = (OP_JAL || OP_JFR || OP_MFL || OP_MFH || OP_IN || OP_OUT);
// Assign M-Format Wires
assign OP_NOP = (ID_OpCode == `ISA_NOP);
assign OP_HLT = (ID_OpCode == `ISA_HLT);
// Opcode Format Wire (Useful for data path MUX Assignments)
assign OPF_M  =  (OP_NOP || OP_HLT);

// Assign Branch Wires
// iJ_xxx based on RF_RB in data path
assign BR_ZERO = (ID_BRC == `ISA_BR_ZERO) && iJ_zero;
assign BR_NZRO = (ID_BRC == `ISA_BR_NZRO) && iJ_nZero;
assign BR_POS = (ID_BRC == `ISA_BR_POSI) &&  iJ_pos;
assign BR_NEG = (ID_BRC == `ISA_BR_NEGA) &&  iJ_neg;
assign BR_TRUE = (BR_ZERO || BR_NZRO || BR_POS || BR_NEG) && OP_BRx;

// Assign Control outputs based on Codes and Cycle

// Pipe Reset Signal
assign oPipe_nRst = nRst;

// Program Counter Control Signals
// PC Reset (Should only be reset on CPU reset)
assign oPC_nRst = nRst;
// PC Load Enable
assign oPC_en = Cycle[1] || (Cycle[3] && (BR_TRUE || OP_JAL || OP_JFR));
assign oPC_tmpEn = Cycle[1];
// PC Jump Enable
assign oPC_offset = Cycle[3] && BR_TRUE;
assign oPC_load = Cycle[3] && (OP_JFR || OP_JAL);

// Register File Control Signals
assign oRF_Write = Cycle[5] && ((OPF_R && ~OP_ST) || (OPF_I && ~OP_DIV &&
~OP_MUL) || OP_MFH || OP_MFL || OP_JAL || OP_IN);
// Note: Most ISA's use RC as the write back address, MiniSRC uses RA
```

```
// RA is dependent on ISA type, use R0 if RA is not specified
// RA is used to load PC on JMP/JAL
assign oRF_AddrA =  (OPF_R | OPF_I) ? ID_RB :
                    (OPF_J) ? ID_RA : 4'h0;
// RB is dependent on ISA type, use R0 if RB is not specified
assign oRF_AddrB =  (OPF_I | OPF_B | OPF_J) ? ID_RA :
                    (OPF_R) ? ID_RC : 4'h0;
// Store is always RA
// ISA Specification states to store PC in r15 on JAL (Jump and Link)
assign oRF_AddrC = (OP_JAL) ? 4'hF : ID_RA;

// Register File Write Back Register Load Enable
assign oRWB_en = 1'b1;

// ALU Control Signals Also, this should be renamed to "Ctrl" like the key
on the keyboard.
assign oALU_Ctrl =  (OP_ADD || OP_ADDI) ? `CTRL_ALU_ADD :
                    (OP_SUB)            ? `CTRL_ALU_SUB :
                    (OP_OR  || OP_ORI)  ? `CTRL_ALU_OR  :
                    (OP_AND || OP_ANDI) ? `CTRL_ALU_AND :
                    (OP_MUL)            ? `CTRL_ALU_MUL :
                    (OP_DIV)            ? `CTRL_ALU_DIV :
                    (OP_SLL)            ? `CTRL_ALU_SLL :
                    (OP_SRL)            ? `CTRL_ALU_SRL :
                    (OP_SRA)            ? `CTRL_ALU_SRA :
                    (OP_ROR)            ? `CTRL_ALU_ROR :
                    (OP_ROL)            ? `CTRL_ALU_ROL :
                    (OP_NOT)            ? `CTRL_ALU_NOT :
                    (OP_NEG)            ? `CTRL_ALU_NEG :
                    // ALU Add is default for most instructions - so why
not remove the (OP_ADD || OP_ADDI) ?
                    `CTRL_ALU_ADD;
// ALU Input A Register Load Enable
assign oRA_en = 1'b1;
// ALU Input B Register Load Enable
assign oRB_en = 1'b1;

// ALU Result High Load EN
assign oRZH_en = 1'b1;
// ALU Result Low Load EN
```

```
assign oRZL_en = 1'b1;
// ALU Result Save EN
assign oRAS_en = (OP_DIV || OP_MUL);

// External Port Register Enable
assign oREP_en = OP_OUT && Cycle[4];

// ALU B Input Select (Selects Imm)
assign oMUX_BIS = OPF_I && ~(OP_DIV || OP_MUL);
// ALU Result High Select
assign oMUX_RZHS = (OP_MFH);
// RF Write Back Select
assign oMUX_WBM = (OP_LD || OP_LI);
// Memory Address Output Select
// assign oMUX_MA = Cycle[1];
assign oMUX_MAP = ~((OP_LD || OP_ST || OP_LI) && Cycle[4]);
// ALU Storage Select
assign oMUX_ASS = (OP_MFL || OP_MFH);
// Write Back Program Counter Select
assign oMUX_WBP = OP_JAL;
// Write Back External Port Select
assign oMUX_WBE = OP_IN;

// Immediate value output
// Assign Imm32 branch distance if the branch is true
assign oImm32 = OP_BRx ? ID_BRD : ID_imm32;

// Memory Read/Write Signals
assign oMemRead = Cycle[1] || (Cycle[4] && (OP_LD || OP_LI));
assign oMemWrite = Cycle[4] && OP_ST;

endmodule
```

Memory Instructions

The CPU specification document details two load instructions. As an extension to the MiniSRC processor, register zero was tied into logic level zero. Thus, the ldi instruction is redundant and only the ld instruction has been implemented.

Memory Instruction Code

Rather than using a bus design, the extended MiniSRC utilizes a 5 stage pipeline. Thus, the only required addition to support load and store instruction is an expansion on the write back multiplexer.

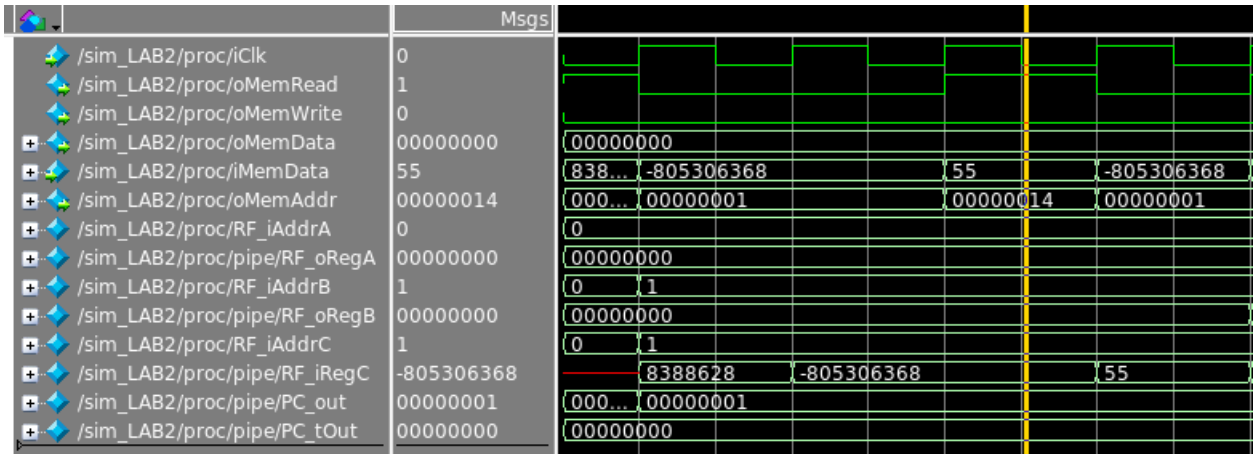
```
// Memory
assign oMemAddr = iMUX_MAP ? PC_out : RZX_out ;
assign oMemData = RB_out;
assign oPORT = RB_out;

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

Load Instruction Demo

To demonstrate the load instruction operation, the following instruction was simulated.

```
// Initialize Data Memory
d_mem[0]  = 32'd2;
d_mem[1] = 32'd1;

// ld r1, 0(r0)
i_mem[0] = `INS_I(`ISA_LD, 4'd1, 4'd0, 19'd20);
```

Based on the memory multiplexer, address 20 corresponds to data memory address 0.

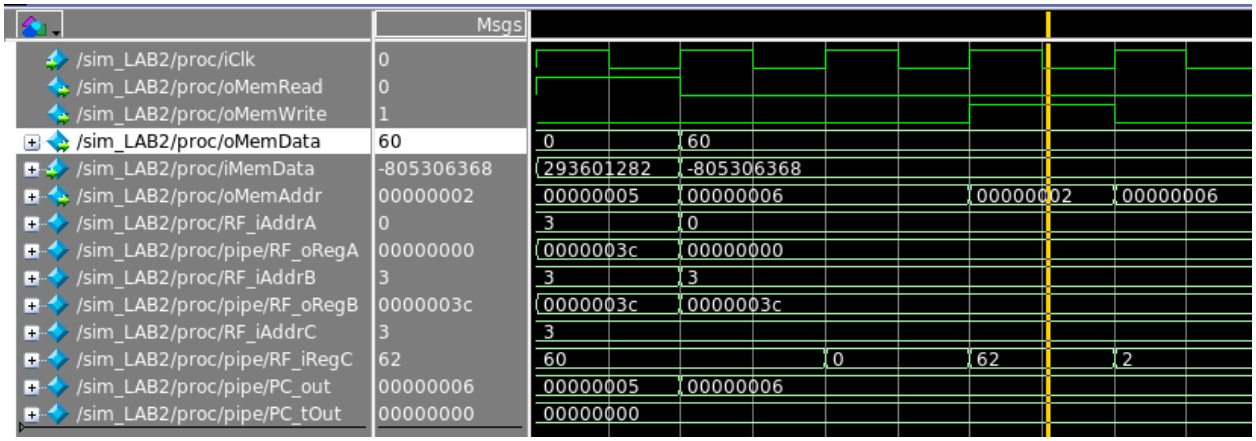


As seen in the above simulation, 55 was loaded into register 1.

Store Instruction Demo

To test the store instruction, a series of instructions were simulated before storing the result back to memory. In this series, 55 is loaded out of memory, 5 is added to it, then it is divided by 1. Afterwards, both the high and low registers of the division are written back to memory.

```
// mfh r3
i_mem[4] = `INS_J(`ISA_MFH, 4'd3);
// st r3, 2(r0)
i_mem[5] = `INS_I(`ISA_ST, 4'd3, 4'd0, 19'd2);
// mfl r3
i_mem[6] = `INS_J(`ISA_MFL, 4'd3);
// st r3, 2(r0)
i_mem[7] = `INS_I(`ISA_ST, 4'd3, 4'd0, 19'd3);
```



As expected, the store instruction returned a result of 60 followed by a result of 0. Additionally, the values were written to the correct addresses.

```
# Write addr: 2 data: 60
# Write addr: 3 data: 0
```

ALU Immediate Instructions

ALU Immediate instructions are handled in the same way as typical ALU instructions. However, the second ALU input is switched to the immediate input using the following multiplexer.

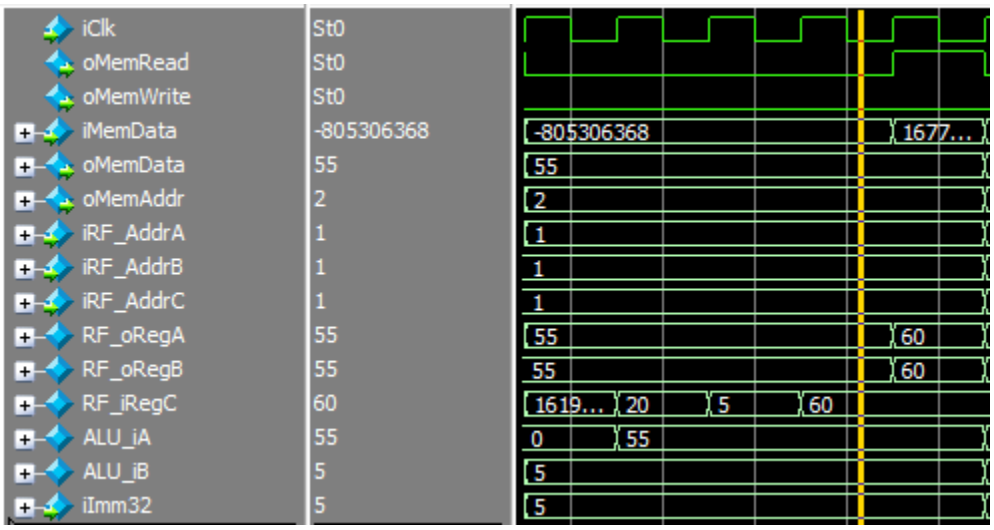
```
// ALU Input Multiplexers

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

Add Immediate Demo

The following code sample was simulated on the processor to confirm the correctness of the ADDI instruction:

```
// ld r1, 0(r0)
i_mem[0] = `INS_I(`ISA_LD, 4'd1, 4'd0, 19'd20);
// addi r1, r1, 5
i_mem[1] = `INS_I(`ISA_ADDI, 4'd1, 4'd1, 19'd5);
```

The following image shows the addi waveform, immediately following the load.

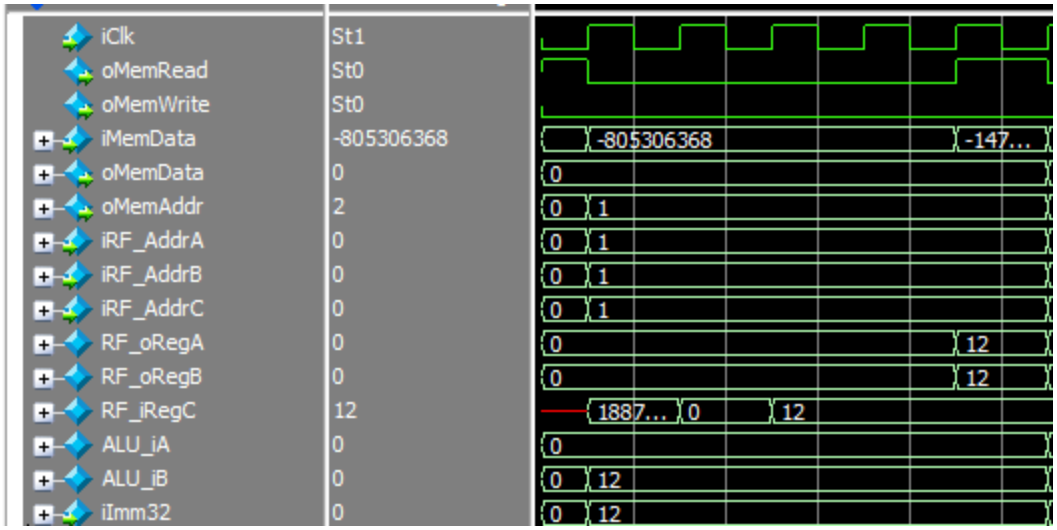


As expected, the value stored in R1 by the load instruction is loaded as the first ALU input, the immediate value is loaded as the second value, and the result is written back to R1.

OR Immediate Instruction

In addition to the add immediate instruction, the or immediate instruction was also tested using the following code:

```
// ori r1, r1, 12
i_mem[0] = `INS_I(`ISA_ORI, 4'd1, 4'd1, 19'd12);
```

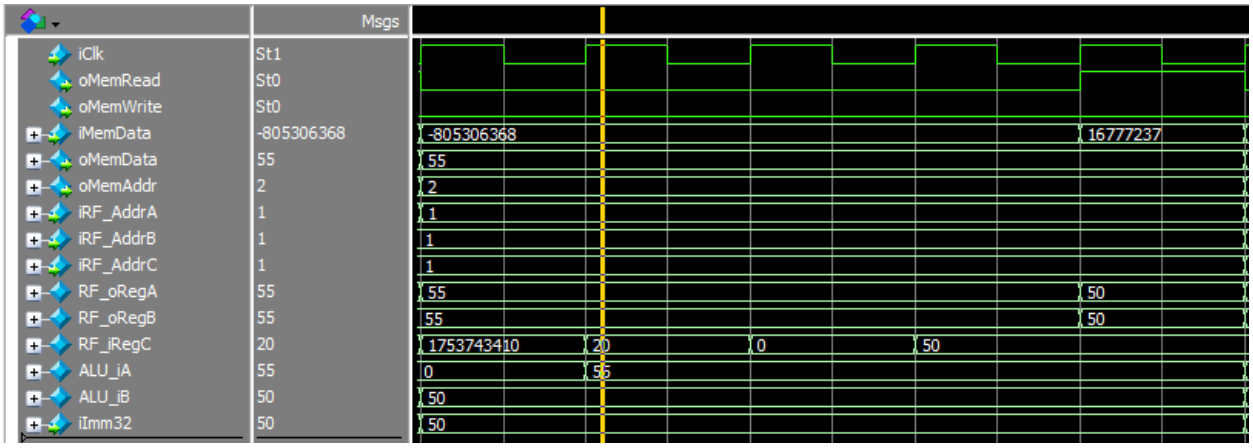


AND Immediate Instruction

Finally, the and immediate instruction was tested:

```
// ld r1, 0(r0)
i_mem[0] = `INS_I(`ISA_LD, 4'd1, 4'd0, 19'd20);
// andi r1, r1, 50
i_mem[1] = `INS_I(`ISA_ANDI, 4'd1, 4'd1, 19'd50);
```

The simulation is shown after the completion of the load instruction:



Branch Instructions

Branch decisions are done in the ID (Instruction Decoding) stage of the pipeline using the following branch logic.

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

```
// Assign Branch Wires
// iJ_xxx based on RF_RB in data path
assign BR_ZERO = (ID_BRC == `ISA_BR_ZERO) && iJ_zero;
assign BR_NZRO = (ID_BRC == `ISA_BR_NZRO) && iJ_nZero;
assign BR_POS = (ID_BRC == `ISA_BR_POSI) &&  iJ_pos;
assign BR_NEG = (ID_BRC == `ISA_BR_NEGA) &&  iJ_neg;
assign BR_TRUE = (BR_ZERO || BR_NZRO || BR_POS || BR_NEG) && OP_BRx;
```

If the branch outcome is true, the following signals are asserted within the program counter, causing it to calculate the counter position for the next instruction by adding the offset immediate value to itself to value instead of incrementing.

```
// Program Counter Control Signals
// PC Reset (Should only be reset on CPU reset)
assign oPC_nRst = nRst;
// PC Load Enable
assign oPC_en = Cycle[1] || (Cycle[3] && (BR_TRUE || OP_JAL || OP_JFR));
// PC Jump Enable
assign oPC_offset = Cycle[3] && BR_TRUE;
assign oPC_load = Cycle[3] && (OP_JFR || OP_JAL);
```

The test bench code for testing these instructions is shown below.

```
1     `timescale 1ns/1ps
  1   `include "../../Control/ISA.vh"
  2   `include "../../constants.vh"
  3   `include "../sim_ISA.vh"
  4
  5   module sim_LAB2_BRx();
  6
  7   parameter SA = `START_PC_ADDRESS;
  8   `define BRANCH_OFFSET 27
  9   `define N_instructions (13+`BRANCH_OFFSET*4*4 + 1)
 10
 11   wire Clk;
 12   reg nRst = 1'b0;
 13
 14   ClockGenerator cg(
 15       .nRst(nRst),
 16       .oClk(Clk)
 17   );
 18
 19   wire mem_read, mem_write;
 20   wire [31:0] proc_mem_out, proc_mem_addr;
 21   reg [31:0] proc_mem_in;
 22   reg [31:0] i_mem[0:`N_instructions];
 23   reg [31:0] d_mem[0:255];
 24   wire [31:0] oPort;
 25
 26   Processor proc(
 27       .iClk(Clk),
 28       .nRst(nRst),
 29       .oMemAddr(proc_mem_addr),
 30       .oMemData(proc_mem_out),
 31       .iMemData(proc_mem_in),
 32       .iMemRdy(1'b1),
 33       .oMemRead(mem_read),
 34       .oMemWrite(mem_write)
 35       // ,.iPort(0)
 36       // ,.oPort(oPort)
 37   );
```

```
39  initial begin
40      // Initialize Data Memory
41      d_mem[0]  = 32'd2;
42      d_mem[1] = 32'd1;
43
44      // brzr r1, 27
45      // addi r1, r0, 1
46      i_mem[0] = `INS_I(`ISA_ADDI, 4'd1, 4'd0, 19'd1);
47      // branch not taken
48      i_mem[1] = `INS_B(`ISA_BRx, 4'd1, `ISA_BR_ZERO, 19'd`BRANCH_OFFSET);
49      // addi r1, r0, 0
50      i_mem[2] = `INS_I(`ISA_ADDI, 4'd1, 4'd0, 19'd0);
51      // branch taken
52      i_mem[3] = `INS_B(`ISA_BRx, 4'd1, `ISA_BR_ZERO, 19'd`BRANCH_OFFSET);
53      // instruction should get ignored
54      // addi r1, r0, 0xBAD
55      i_mem[4] = `INS_I(`ISA_ADDI, 4'd1, 4'd0, 19'hBAD);
56
57      // brnz r1, 27
58      // branch not taken
59      i_mem[4+`BRANCH_OFFSET*4] = `INS_B(`ISA_BRx, 4'd1, `ISA_BR_NZRO, 19'd`BRANCH_OFFSET);
60      // addi r1, r0, -1
61      i_mem[5+`BRANCH_OFFSET*4] = `INS_I(`ISA_ADDI, 4'd1, 4'd0, -19'd1);
62      // branch taken
63      i_mem[6+`BRANCH_OFFSET*4] = `INS_B(`ISA_BRx, 4'd1, `ISA_BR_NZRO, 19'd27);
64      // instruction should get ignored
65      // addi r1, r0, 0xBAD
66      i_mem[7+`BRANCH_OFFSET*4] = `INS_I(`ISA_ADDI, 4'd1, 4'd0, 19'hBAD);
```

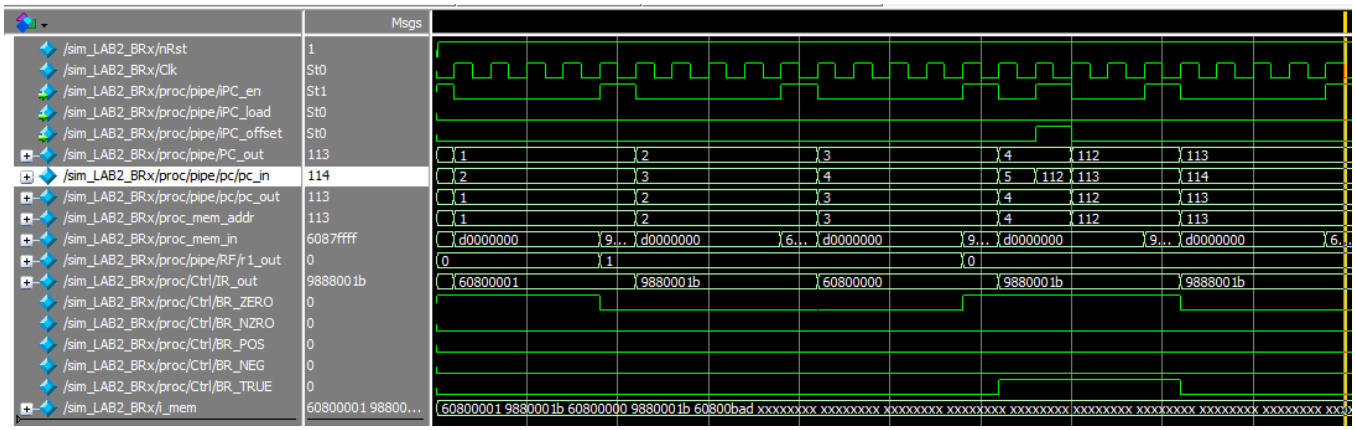
```
    // brpl r1, 27
    // branch not taken
    i_mem[7+`BRANCH_OFFSET*4*2] = `INS_B(`ISA_BRx, 4'd1, `ISA_BR_POSI, 19'd`BRANCH_OFFSET);
    // addi r1, r0, 1
    i_mem[8+`BRANCH_OFFSET*4*2] = `INS_I(`ISA_ADDI, 4'd1, 4'd0, 19'd1);
    // branch taken
    i_mem[9+`BRANCH_OFFSET*4*2] = `INS_B(`ISA_BRx, 4'd1, `ISA_BR_POSI, 19'd`BRANCH_OFFSET);
    // instruction should get ignored
    // addi r1, r0, 0xBAD
    i_mem[10+`BRANCH_OFFSET*4*2] = `INS_I(`ISA_ADDI, 4'd1, 4'd0, 19'hBAD);

    // brmi r2, 27
    // branch not taken
    i_mem[10+`BRANCH_OFFSET*4*3] = `INS_B(`ISA_BRx, 4'd1, `ISA_BR_NEGA, 19'd`BRANCH_OFFSET);
    // addi r1, r0, -1
    i_mem[11+`BRANCH_OFFSET*4*3] = `INS_I(`ISA_ADDI, 4'd1, 4'd0, -19'd1);
    // branch taken
    i_mem[12+`BRANCH_OFFSET*4*3] = `INS_B(`ISA_BRx, 4'd1, `ISA_BR_NEGA, 19'd`BRANCH_OFFSET);
    // instruction should get ignored
    // addi r1, r0, 0xBAD
    i_mem[13+`BRANCH_OFFSET*4*3] = `INS_I(`ISA_ADDI, 4'd1, 4'd0, 19'hBAD);

    // loop back to beginning using jump
    i_mem[13+`BRANCH_OFFSET*4*4] = `INS_J(`ISA_JFR, 4'd0);
    #1
    nRst = 1'b1;
end
```

```
always @(mem_read, mem_write) begin
    if(mem_read) begin
        if(proc_mem_addr < `N_instructions) begin
            proc_mem_in = i_mem[((proc_mem_addr-SA))];
        end
        else begin
            proc_mem_in = d_mem[proc_mem_addr-`N_instructions];
        end
    end
    else begin
        proc_mem_in = `INS_M(`ISA_NOP);
    end
    if(mem_write) begin
        d_mem[proc_mem_addr] = proc_mem_out;
        $display("Write addr: %0d data: %0d", proc_mem_addr, proc_mem_out);
    end
end

endmodule
```

In this scenario, the program counter is set to increment by 1 word per instruction, and memory is only word addressable in this test bench. However, the implementation of the branch instruction calculates the program counter offset based on 4 times the immediate value to account for when the memory is byte addressable and the program counter increments by 4

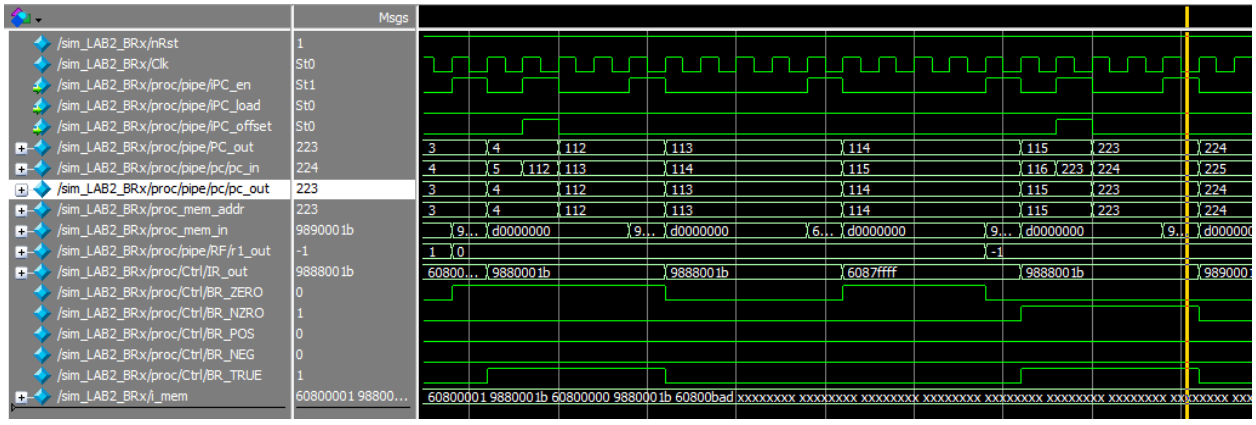
bytes per instruction. To address this, minor adjustments were made to the memory layout of the instructions for this test bench.

Branch on Zero



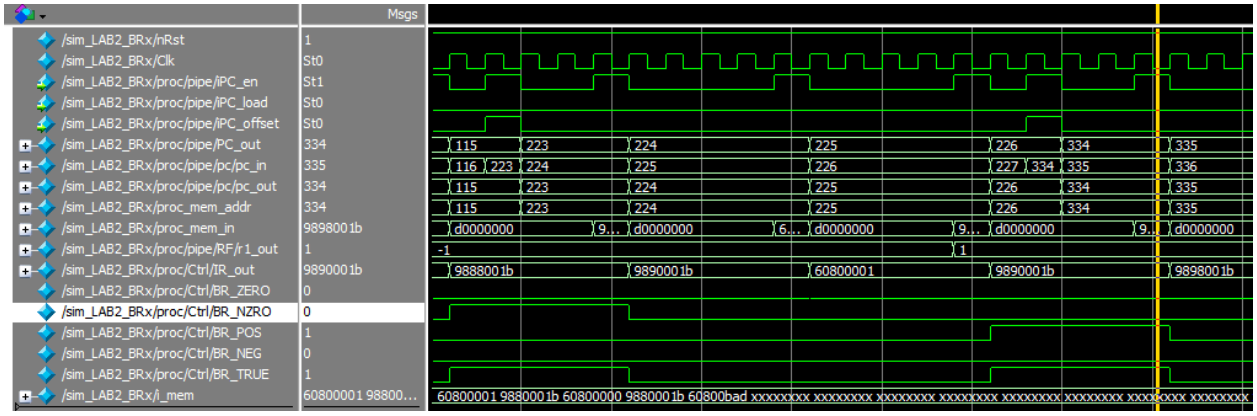
As you can see, the successful branch of the second brzr instruction causes the program counter to arrive at memory address 3+1+27*4 = 112 at the location of the first branch on non-zero instruction.

Branch on Non-Zero



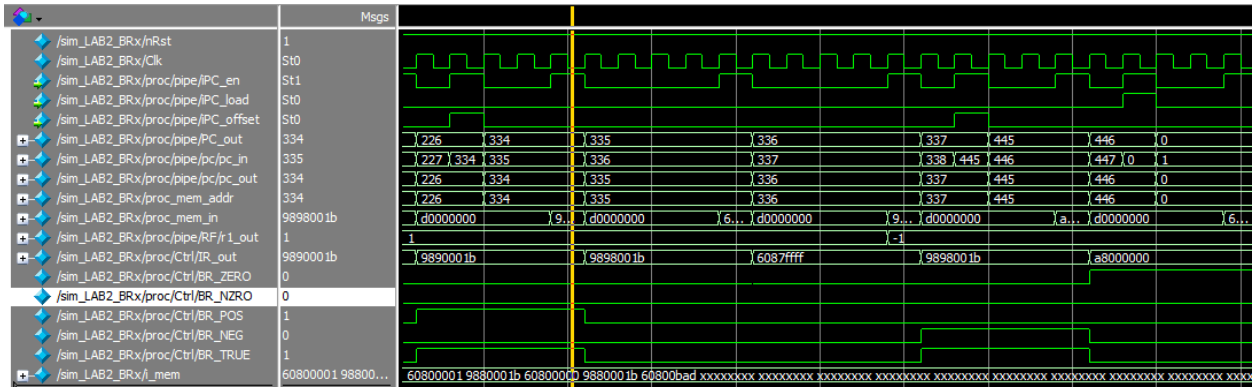
As you can see in the waveform, the branch on Non-Zero does not offset the program counter when R1 contains the value 0. However, after R1 is set to the value -1, then the branch on does occur, offsetting the program counter to memory address 7+27*4*2=223 which is the address of the first branch on positive instruction.

Branch on Positive



As you can see, when the value of register R1 contained positive 1, the branch on positive offset the program counter to memory address 10+27*4*3=334 to the first branch on negative instruction.

Branch on Negative



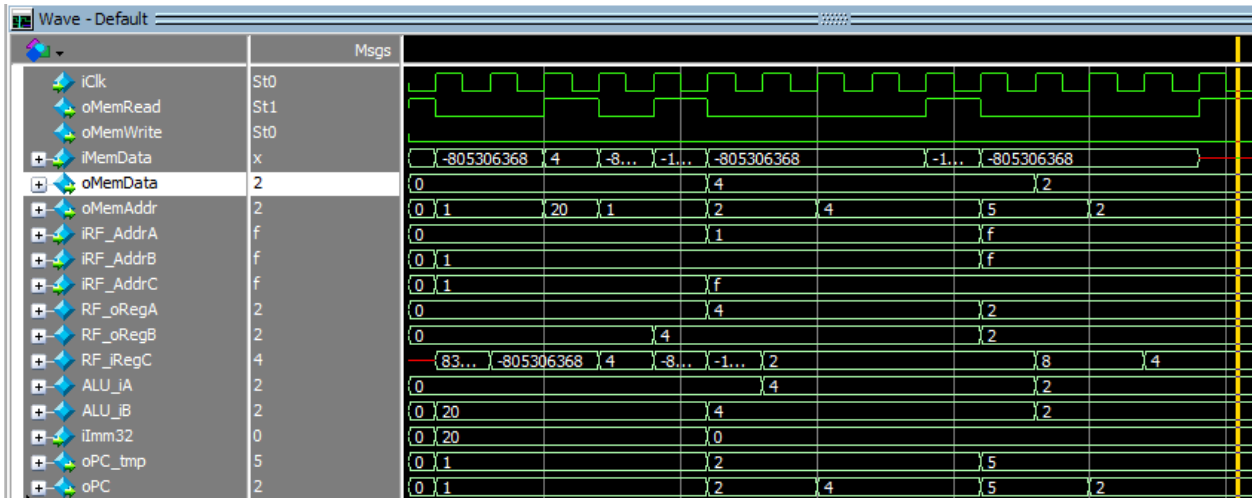
Finally, the brmi instruction successfully offsets the program counter to memory address 13+27*4*4=445 where the jump from register instruction then returns the flow of execution to the beginning again.

Jump Instructions

Jump and Link (JAL)

Fundamentally, the jump and link instruction is identical to the branch instruction. However, when a jump and link instruction is detected, the program counter must be written back to R15. The functional simulation of the jump and link is shown below.

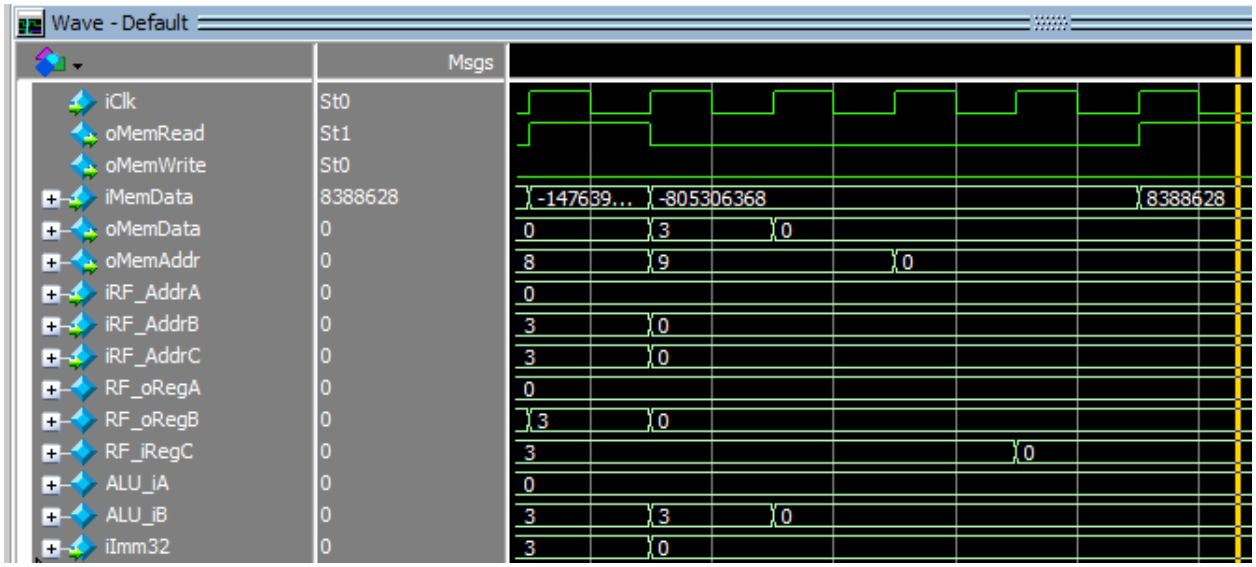
```
// ld r1, 0(r0)
i_mem[0] = `INS_I(`ISA_LD, 4'd1, 4'd0, 19'd20);
// jal r1
i_mem[1] = `INS_J(`ISA_JAL, 4'd1);
```



Jump from Register

In the following functional simulation, the Jump from Register instruction was used to jump back to the first instruction.

```
// jmp to beginning
i_mem[8] = `INS_J(`ISA_JFR, 4'd0);
```



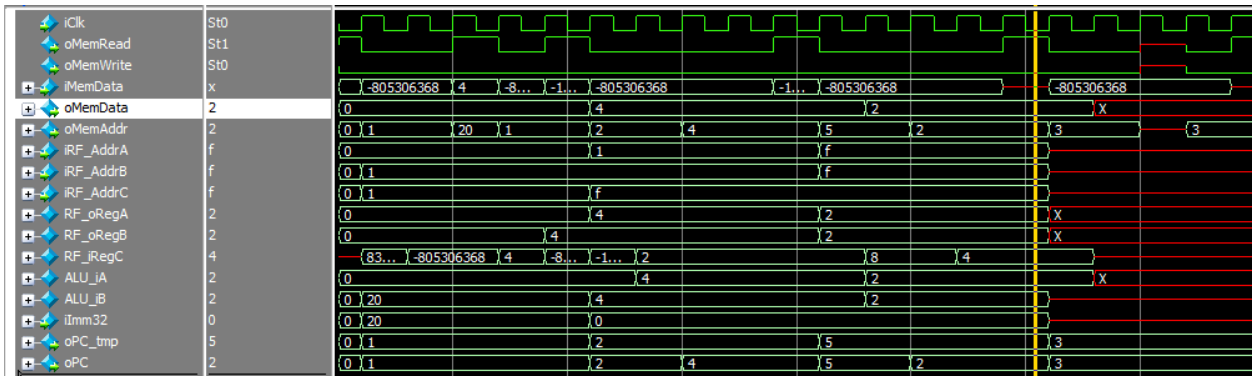
As seen in the above simulation, after the jump from register instruction completes, the next instruction is read from address 0.

Procedure Call Simulation

The following simulation loads the address of the return instruction into R1. Afterwards, it uses the JAL instruction to move to the procedure call containing only the return instruction.

```
// ld r1, 0(r0)
i_mem[0] = `INS_I(`ISA_LD, 4'd1, 4'd0, 19'd20);
// jal r1
i_mem[1] = `INS_J(`ISA_JAL, 4'd1);
// jfr ra
i_mem[4] = `INS_J(`ISA_JFR, 4'hF);
```

In the functional simulation shown below, the JFR returns to the address immediately after the JAL instruction.



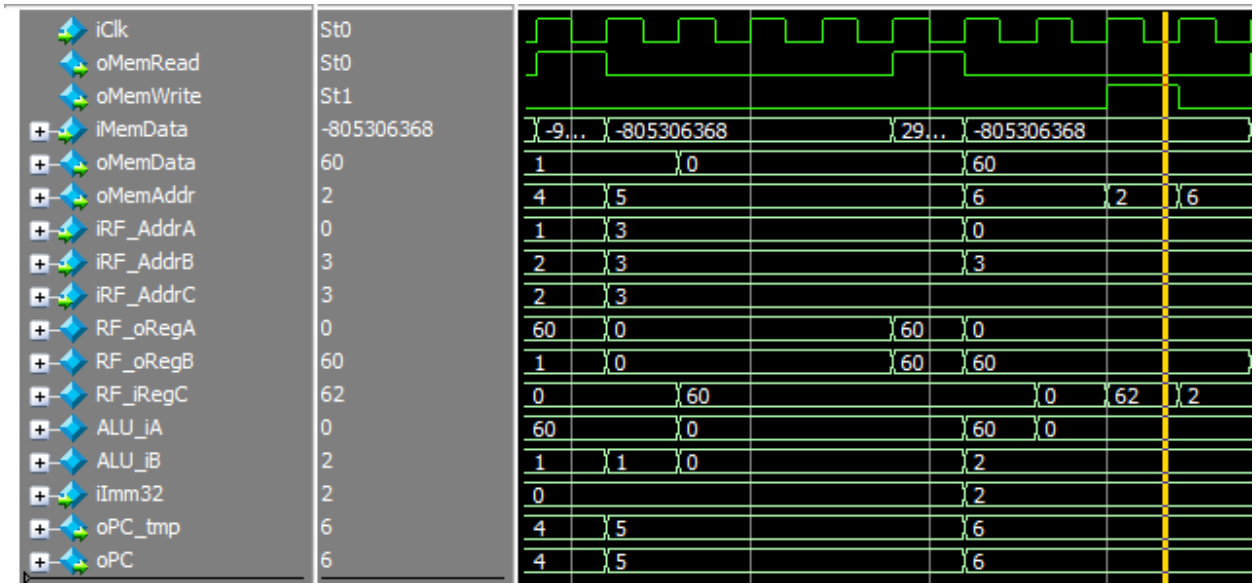
Note that the address after the jump instruction contains no meaningful instruction encoding. Thus the undefined behavior occurring at the end of the simulation is expected.

Special Instructions

Special instructions include the move from low and move from high instructions used to pull from the ALU low and high registers. To test these instructions, the following sample code was used:

```
// Initialize Data Memory
d_mem[0]  = 32'd55;
d_mem[1] = 32'd1;

// ld r1, 0(r0)
i_mem[0] = `INS_I(`ISA_LD, 4'd1, 4'd0, 19'd20);
// neg r1, r1
i_mem[1] = `INS_I(`ISA_ADDI, 4'd1, 4'd1, 19'd5);
// ld r2, 1(r0)
i_mem[2] = `INS_I(`ISA_LD, 4'd2, 4'd0, 19'd21);
// div r3, r1, r2
i_mem[3] = `INS_I(`ISA_DIV, 4'd2, 4'd1, 19'd0);
// mfh r3
i_mem[4] = `INS_J(`ISA_MFH, 4'd3);
// st r3, 2(r0)
i_mem[5] = `INS_I(`ISA_ST, 4'd3, 4'd0, 19'd2);
// mfl r3
i_mem[6] = `INS_J(`ISA_MFL, 4'd3);
// st r3, 2(r0)
i_mem[7] = `INS_I(`ISA_ST, 4'd3, 4'd0, 19'd3);
// jmp to beginning
i_mem[8] = `INS_J(`ISA_JFR, 4'd0);
#1
```

The following simulation shows the waveforms for the move from high instruction.



Finally, the results written to memory are displayed in the console. A screenshot is shown below:

```
VSIM 40> run 800ns
# Write addr: 2 data: 60
# Write addr: 3 data: 0
```

Port Instructions

In addition to memory, the processor also contains a single port mapped IO interface. The following simulation reads and writes to the port.

```
// Initialize Data Memory
d_mem[0]  = 32'd55;
d_mem[1] = 32'd1;

// ld r1, 0(r0)
i_mem[0] = `INS_I(`ISA_LD, 4'd1, 4'd0, 19'd20);
// addi r1, r1, 5
i_mem[1] = `INS_I(`ISA_ADDI, 4'd1, 4'd1, 19'd5);
// in r5
i_mem[2] = `INS_J(`ISA_IN, 4'd5);
// out r1
i_mem[3] = `INS_J(`ISA_OUT, 4'd1);
```

As seen in the following simulation, the result of the addi instruction appears on the output link of the port. Additionally, the value fixed to the input link of the port is loaded into register 5.