ELEC 374: MiniSRC CPU Project
Phase 3 Report

Group 4

Presented by:

Jacob Chisholm (21jc138)

Hendrix Gryspeerdt (21hgg3)

Luke Strickland (21laps1)

Date: 2025-02-20

“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

List of Figures

Test Bench

Figure 1 shows the definitions of the opcode signatures. This is done to reduce the work when coding the control unit and the test bench and make it more readable in general, which help with debugging.

```
// Opcode Signatures
// R Format Instructions
`define ISA_ADD  5'b00011
`define ISA_SUB  5'b00100
`define ISA_AND  5'b00101
`define ISA_OR   5'b00110
`define ISA_ROR  5'b00111
`define ISA_ROL  5'b01000
`define ISA_SRL  5'b01001
`define ISA_SRA  5'b01010
`define ISA_SLL  5'b01011
// I Format Instructions
`define ISA_LD   5'b00000
`define ISA_LI   5'b00001
`define ISA_ST   5'b00010
`define ISA_ADDI 5'b01100
`define ISA_ANDI 5'b01101
`define ISA_ORI  5'b01110
`define ISA_DIV  5'b01111
`define ISA_MUL  5'b10000
`define ISA_NEG  5'b10001
`define ISA_NOT  5'b10010
// B Format Instructions
`define ISA_BRx  5'b10011
// J Format Instructions
`define ISA_JAL  5'b10100
`define ISA_JFR  5'b10101
`define ISA_IN   5'b10110
`define ISA_OUT  5'b10111
`define ISA_MFL  5'b11000
`define ISA_MFH  5'b11001
// M Format Instructions
`define ISA_NOP  5'b11010
`define ISA_HLT  5'b11011 // internally implemented by preventing the step counter from incrementing in the control unit.

// Branch Codes
// Branch if Zero
`define ISA_BR_ZERO 2'b00
// Branch if NonZero
`define ISA_BR_NZRO 2'b01
// Branch if Positive
`define ISA_BR_POSI 2'b10
// Branch if negative
`define ISA_BR_NEGA 2'b11
```

Figure 1: Opcode signatures defined

Figure 2 and Figure 3 show the entirety of the test bench code that runs the specified program for phase 3 of the project. The test bench places all necessary instructions into memory and then simulates the processor as normal.

```
`timescale 1ns/1ps
`include "../../Control/ISA.vh"
`include "../../constants.vh"
`include "../sim_ISA.vh"

module sim_LAB3_prog();

parameter SA = `START_PC_ADDRESS;
// largest valid memory address, doing +1 because in logical simulation, the pc is incremented before the oMemRead signal is de-asserted for a
`define MEM_MAX ((8'hBC) + 1)

wire Clk;
reg nRst = 1'b0;

ClockGenerator cg(
    .nRst(nRst),
    .oClk(Clk)
);

wire mem_read, mem_write;
wire [31:0] proc_mem_out, proc_mem_addr;
reg [31:0] proc_mem_in;
reg [31:0] mem[0:`MEM_MAX];
wire [31:0] oPort;

Processor proc(
    .iClk(Clk),
    .nRst(nRst),
    .oMemAddr(proc_mem_addr),
    .oMemData(proc_mem_out),
    .iMemData(proc_mem_in),
    .iMemRdy(1'b1),
    .oMemRead(mem_read),
    .oMemWrite(mem_write)
    ,.iPORT(0)
    ,.oPORT(oPort)
);
```

Figure 2: Test bench code part 1

```
initial begin
    // Initialize Data Memory
    // Initialize memory locations 0x54 and 0x92 with the 32-bit hexadecimal values 0x97 and 0x46, respectively.
    mem[8'h54]  = 32'h97;
    mem[8'h92] = 32'h46;

    // we can't use R0 the way they wanted to use it here, because we have R0 as a constant R0 = 0; so we will use register R9 instead.
    // Encode your program in memory with the starting address zero
    mem[8'h00] = `INS_I(`ISA_ADDI, 4'd3, 4'd0, 19'h65); // ldi R3, 0x65 ; R3 = 0x65 // addi r3, r0, 0x65 ; R3 = 0x65
    mem[8'h01] = `INS_I(`ISA_ADDI, 4'd3, 4'd3, 19'h3);  // ldi R3, 3(R3) ; R3 = 0x68 // addi R3, R3, 3 ; R3 = 0x68
    mem[8'h02] = `INS_I(`ISA_LD, 4'd2, 4'd0, 19'h54);   // ld R2, 0x54 ; R2 = (0x54) = 0x97
    mem[8'h03] = `INS_I(`ISA_ADDI, 4'd2, 4'd2, 19'd1); // ldi R2, 1(R2) ; R2 = 0x98
    mem[8'h04] = `INS_I(`ISA_LD, 4'd9, 4'd2, -19'd6); // ld R0, -6(R2) ; R0 = (0x92) = 0x46 // ld R9, -6(R2) ; R9 = (0x92) = 0x46
    mem[8'h05] = `INS_I(`ISA_ADDI, 4'd1, 4'd0, 19'd3); // ldi R1, 3 ; R1 = 3
    mem[8'h06] = `INS_I(`ISA_ADDI, 4'd3, 4'd0, 19'h57); // ldi R3, 0x57 ; R3 = 0x57
    mem[8'h07] = `INS_B(`ISA_BRx, 4'd3, `ISA_BR_NEGA, 19'd3); // brmi R3, 3 ; continue with the next instruction (will not branch)
    mem[8'h08] = `INS_I(`ISA_ADDI, 4'd3, 4'd3, 19'd3); // ldi R3, 3(R3) ; R3 = 0x5A
    mem[8'h09] = `INS_I(`ISA_LD, 4'd4, 4'd3, -19'd6); // ld R4, -6(R3) ; R4 = (0x5A - 6) = 0x97
    mem[8'h0A] = `INS_M(`ISA_NOP); // nop
    mem[8'h0B] = `INS_B(`ISA_BRx, 4'd4, `ISA_BR_POSI, 19'd1); // brpl R4, 2 ; continue with the instruction at "target" (will branch) // brpl
    mem[8'h0C] = `INS_I(`ISA_ADDI, 4'd6, 4'd3, 19'd7); // ldi R6, 7(R3) ; this instruction will not execute
    mem[8'h0D] = `INS_I(`ISA_ADDI, 4'd5, 4'd6, -19'd4); // ldi R5, -4(R6) ; this instruction will not execute
    mem[8'h0E] = `INS_M(`ISA_NOP); // filled this in because we are incrementing PC by 1 and branches are multiplied by 4.
    mem[8'h0F] = `INS_M(`ISA_NOP);
    mem[8'h10] = `INS_R(`ISA_ADD, 4'd3, 4'd3, 4'd1); // add R3, R3, R1 ; R3 = 0x5D
    mem[8'h11] = `INS_I(`ISA_ADDI, 4'd4, 4'd4, 19'd2); // addi R4, R4, 2 ; R4 = 0x99
    mem[8'h12] = `INS_I(`ISA_NEG, 4'd4, 4'd4, 19'd0); // neg R4, R4 ; R4 = 0xFFFFFF67
    mem[8'h13] = `INS_I(`ISA_NOT, 4'd4, 4'd4, 19'd0); // not R4, R4 ; R4 = 0x98
    mem[8'h14] = `INS_I(`ISA_ANDI, 4'd4, 4'd4, 19'hF); // andi R4, R4, 0xF ; R4 = 8
    mem[8'h15] = `INS_R(`ISA_ROR, 4'd2, 4'd9, 4'd1); // ror R2, R0, R1 ; R2 = 0xC0000008 // ror R2, R9, R1 ; R2 = 0xC0000008
    mem[8'h16] = `INS_I(`ISA_ORI, 4'd4, 4'd2, 19'd7); // ori R4, R2, 7 ; R4 = 0xC000000F
    mem[8'h17] = `INS_R(`ISA_SRA, 4'd2, 4'd4, 4'd1); // shra R2, R4, R1 ; R2 = 0xF8000001
    mem[8'h18] = `INS_R(`ISA_SRA, 4'd3, 4'd3, 4'd1); // shr R3, R3, R1 ; R3 = 0xB
    mem[8'h19] = `INS_I(`ISA_ST, 4'd3, 4'd0, 19'h92); // st 0x92, R3 ; (0x92) = 0xB new value in memory with address 0x92
    mem[8'h1A] = `INS_R(`ISA_ROL, 4'd3, 4'd9, 4'd1); // rol R3, R0, R1 ; R3 = 0x230 // rol R3, R9, R1 ; R3 = 0x230
    mem[8'h1B] = `INS_R(`ISA_OR, 4'd5, 4'd1, 4'd9); // or R5, R1, R0 ; R5 = 0x47  // or R5, R1, R9 ; R5 = 0x47
    mem[8'h1C] = `INS_R(`ISA_AND, 4'd2, 4'd3, 4'd9); // and R2, R3, R0 ; R2 = 0 // and R2, R3, R9 ; R2 = 0
    mem[8'h1D] = `INS_I(`ISA_ST, 4'd5, 4'd2, 19'h54); // st 0x54(R2), R5 ; (0x54) = 0x47 new value in memory with address 0x54
    mem[8'h1E] = `INS_R(`ISA_SUB, 4'd9, 4'd3, 4'd5); // sub R0, R3, R5 ; R0 = 0x1E9 // sub R9, R3, R5 ; R9 = 0x1E9
    mem[8'h1F] = `INS_R(`ISA_SLL, 4'd2, 4'd3, 4'd1); // shl R2, R3, R1 ; R2 = 0x1180
    mem[8'h20] = `INS_I(`ISA_ADDI, 4'd5, 4'd0, 19'd8); // ldi R5, 8 ; R5 = 8
    mem[8'h21] = `INS_I(`ISA_ADDI, 4'd6, 4'd0, 19'h17); // ldi R6, 0x17 ; R6 = 0x17
    mem[8'h22] = `INS_I(`ISA_MUL, 4'd5, 4'd6, 19'd0); // mul R6, R5 ; HI = 0; LO = 0xB8  // note that the registers are swapped in the instruc
    mem[8'h23] = `INS_J(`ISA_MFH, 4'd4); // mfhi R4 ; R4 = 0
    mem[8'h24] = `INS_J(`ISA_MFL, 4'd7); // mflo R7 ; R7 = 0xB8
    mem[8'h25] = `INS_I(`ISA_DIV, 4'd5, 4'd6, 19'd0); // div R6, R5 ; HI = 7 , LO = 2 // note that the registers are swapped in the instructio
    mem[8'h26] = `INS_I(`ISA_ADDI, 4'd10, 4'd5, 19'd1); // ldi R10, 1(R5) ; R10 = 9 setting up argument registers
    mem[8'h27] = `INS_I(`ISA_ADDI, 4'd11, 4'd6, -19'd3); // ldi R11, -3(R6) ; R11 = 0x14 R10, R11, R12, and R13
    mem[8'h28] = `INS_I(`ISA_ADDI, 4'd12, 4'd7, 19'd1); // ldi R12, 1(R7) ; R12 = 0xB9
    mem[8'h29] = `INS_I(`ISA_ADDI, 4'd13, 4'd4, 19'd4); // ldi R13, 4(R4) ; R13 = 4
    mem[8'h2A] = `INS_J(`ISA_JAL, 4'd12); // jal R12 ; address of subroutine subA in R12 - return address in R8
    mem[8'h2B] = `INS_M(`ISA_HLT); // halt ; upon return, the program halts

    // ORG 0xB9
    mem[8'hB9] = `INS_R(`ISA_ADD, 4'd15, 4'd10, 4'd12); // add R15, R10, R12 ; R14 and R15 are return value registers R15 = 0xC2
    mem[8'hBA] = `INS_R(`ISA_SUB, 4'd14, 4'd11, 4'd13); // sub R14, R11, R13 ; R15 = 0xC2, R14 = 0x10
    mem[8'hBB] = `INS_R(`ISA_SUB, 4'd15, 4'd15, 4'd14); // sub R15, R15, R14 ; R15 = 0xB2
    mem[8'hBC] = `INS_J(`ISA_JFR, 4'd8); // jr R8

    #1
    // Initialize registers R0 - R15 and the PC to 0 with the Reset input signal.
    nRst = 1'b1;
end

always @(mem_read, mem_write) begin
    if (proc_mem_addr > `MEM_MAX) begin
        $display("Memory address out of bounds: 0x%0h", proc_mem_addr);
        proc_mem_in = `INS_M(`ISA_NOP);
    end
    else if (mem_read) begin
        proc_mem_in = mem[proc_mem_addr];
    end
    else if (mem_write) begin
        mem[proc_mem_addr] = proc_mem_out;
        $display("Write addr: 0x%0h data: 0x%0h", proc_mem_addr, proc_mem_out);
    end
end

endmodule
```

Figure 3:Test bench code part 2. Note that the extra No-Op instructions are placed there because the branch instruction is designed to increment the PC based on byte-addressable memory (which is planned to be implemented in phase 4) where each word is 4 bytes. For the sake of simplicity, in this phase of the project, the memory is only word addressable, hence the branch instruction always branches by a multiple of 4.

Functional Simulation with Memory

Figure 4, Figure 5, Figure 6, Figure 7, Figure 8, Figure 9, Figure 10 shows all the waveforms generated by the test bench while the processor was in execution before it halts. The figures are in order of program execution. Figure 11 and Figure 12 show the contents of memory before and after the execution respectively. As seen, everything worked as intended without any errors.

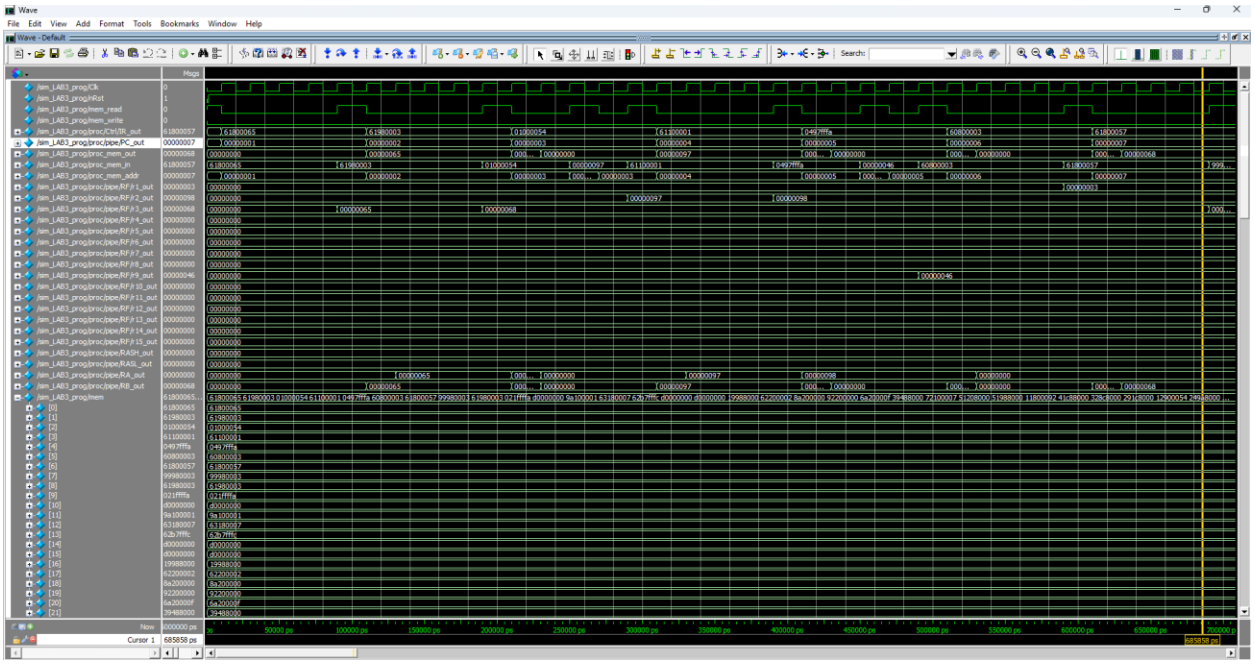

Figure 4: Instructions with PC 00-06

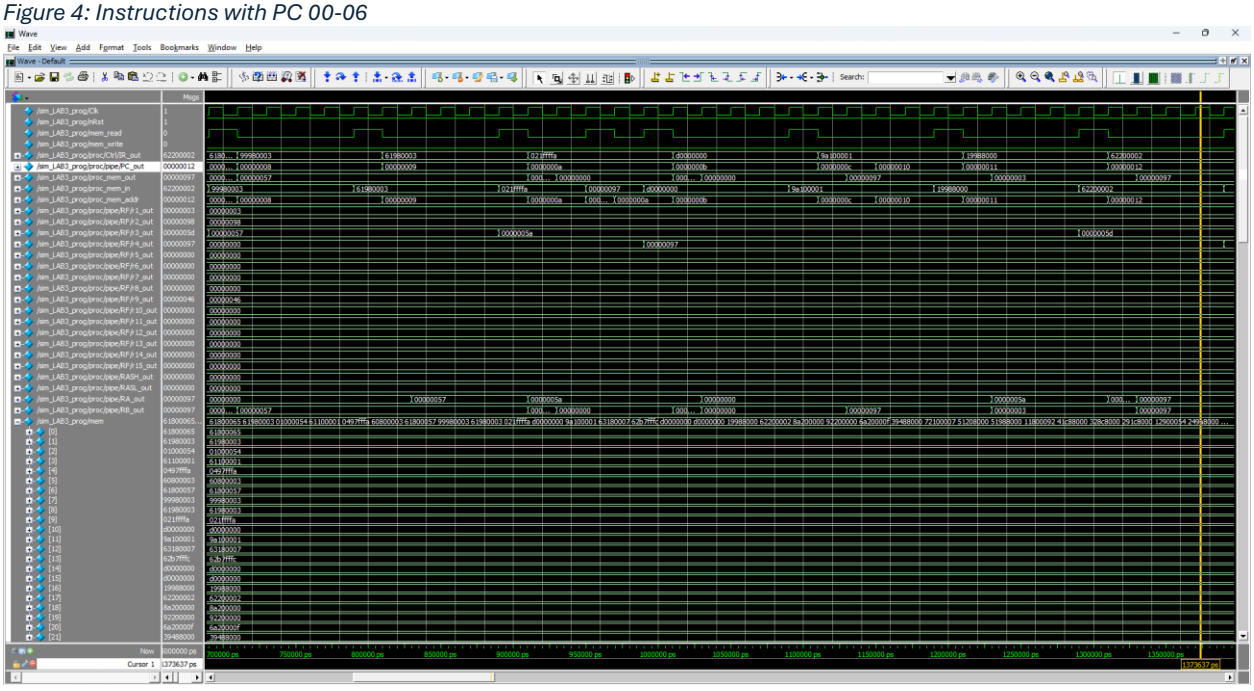

Figure 5: Instructions with PC 07-11

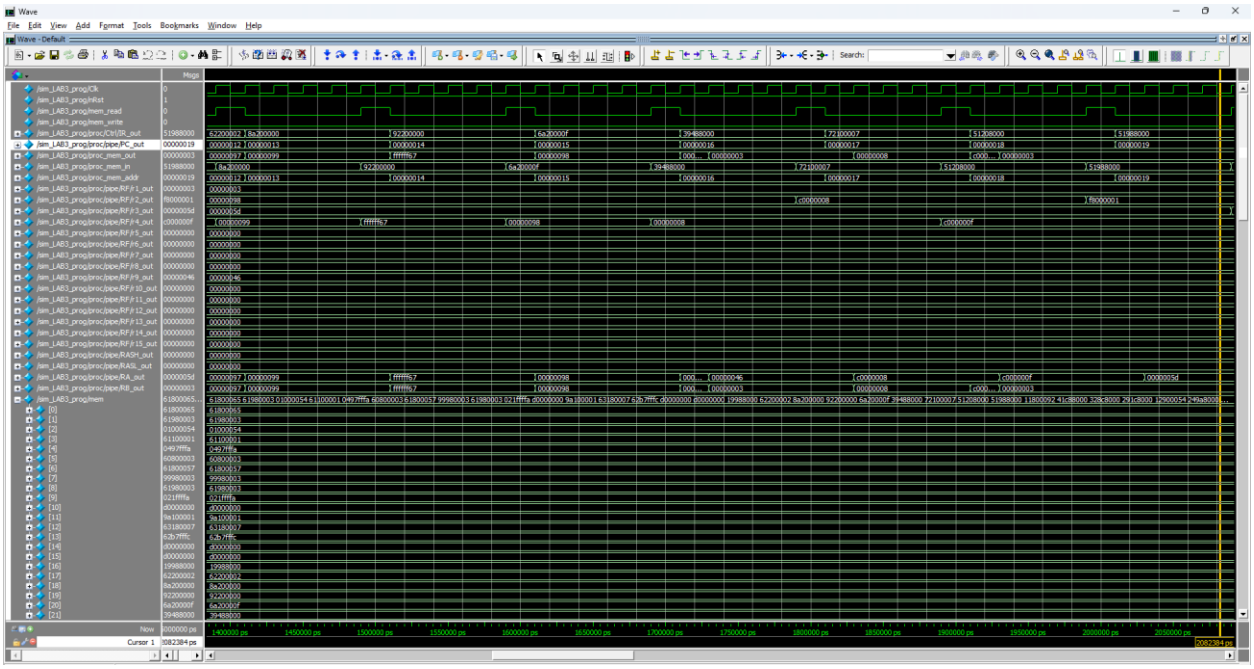

Figure 6: Instructions with PC 12-18

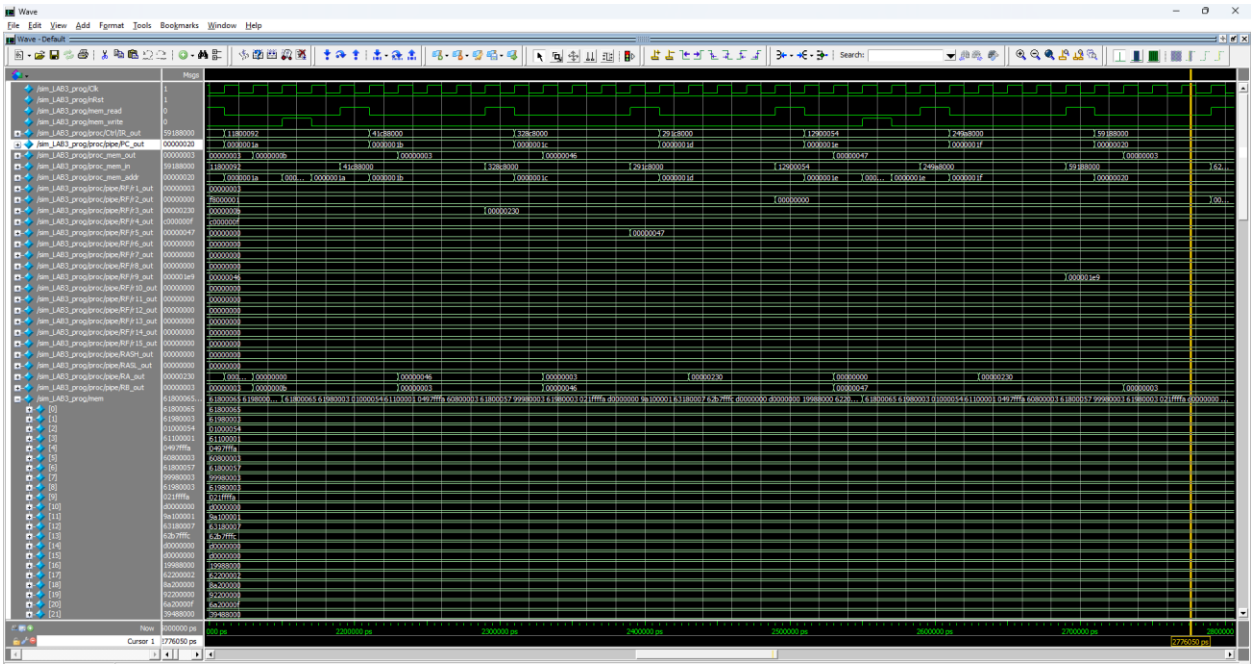

Figure 7: Instructions with PC 19-1f

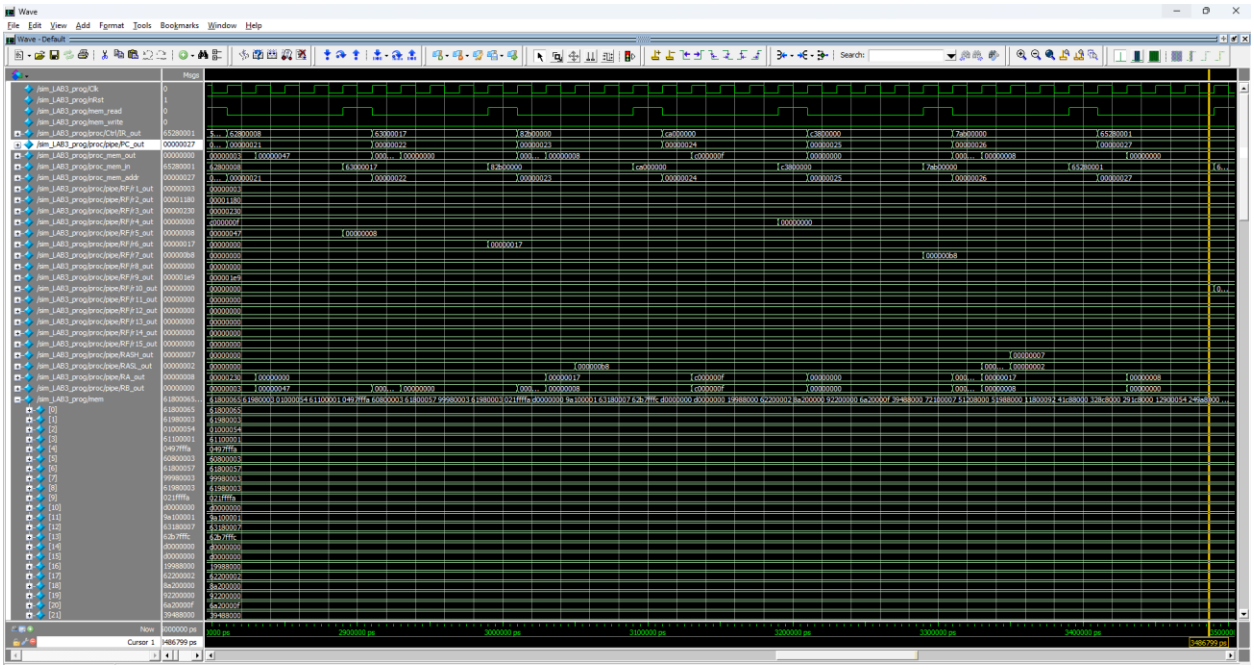

Figure 8: Instructions with PC 20-26

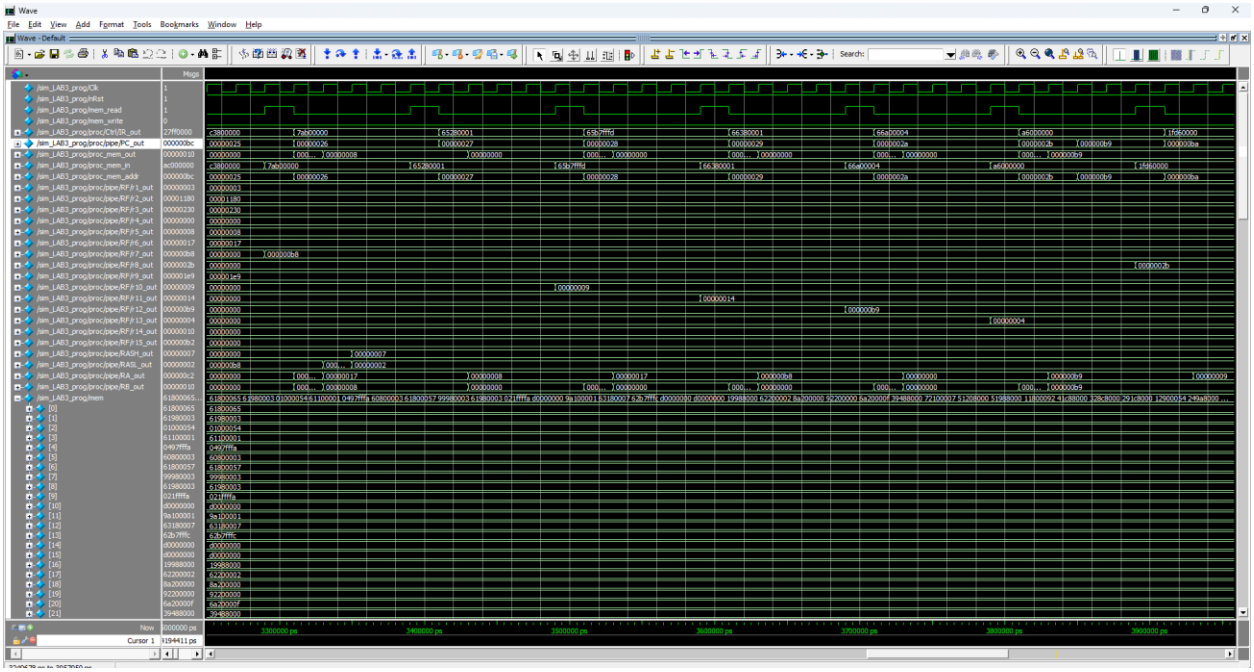

Figure 9: Instructions with PC 25-b9 including jump

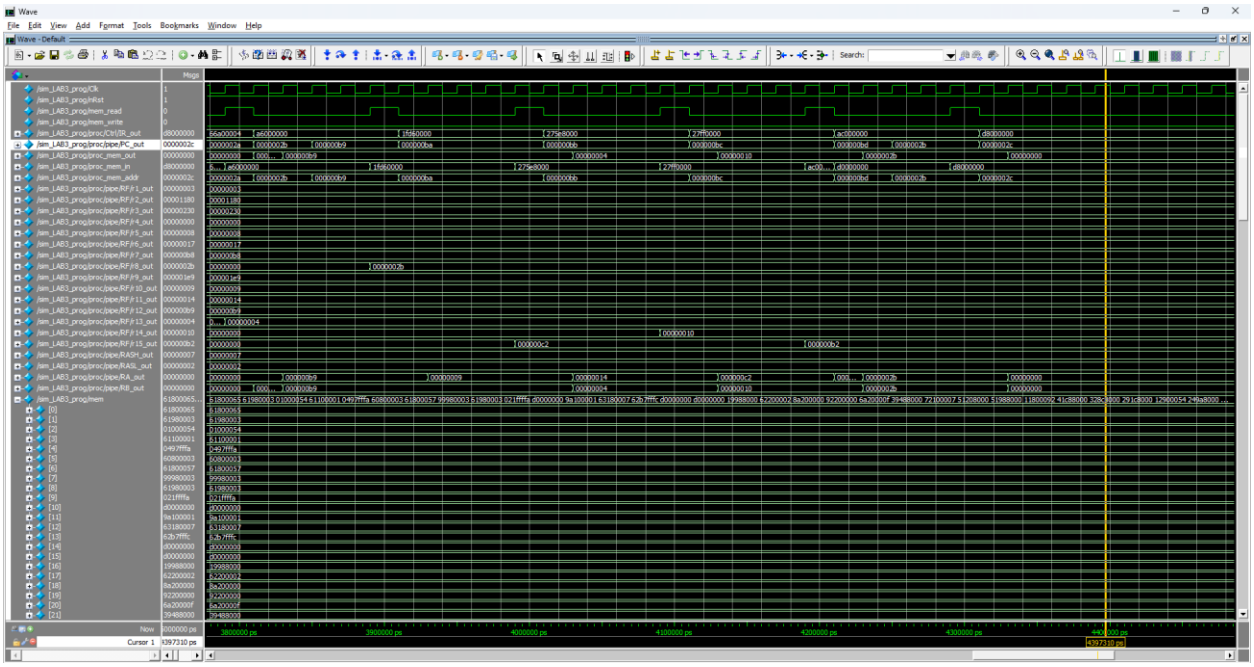

Figure 10: Instructions with PC b9-2b including jump

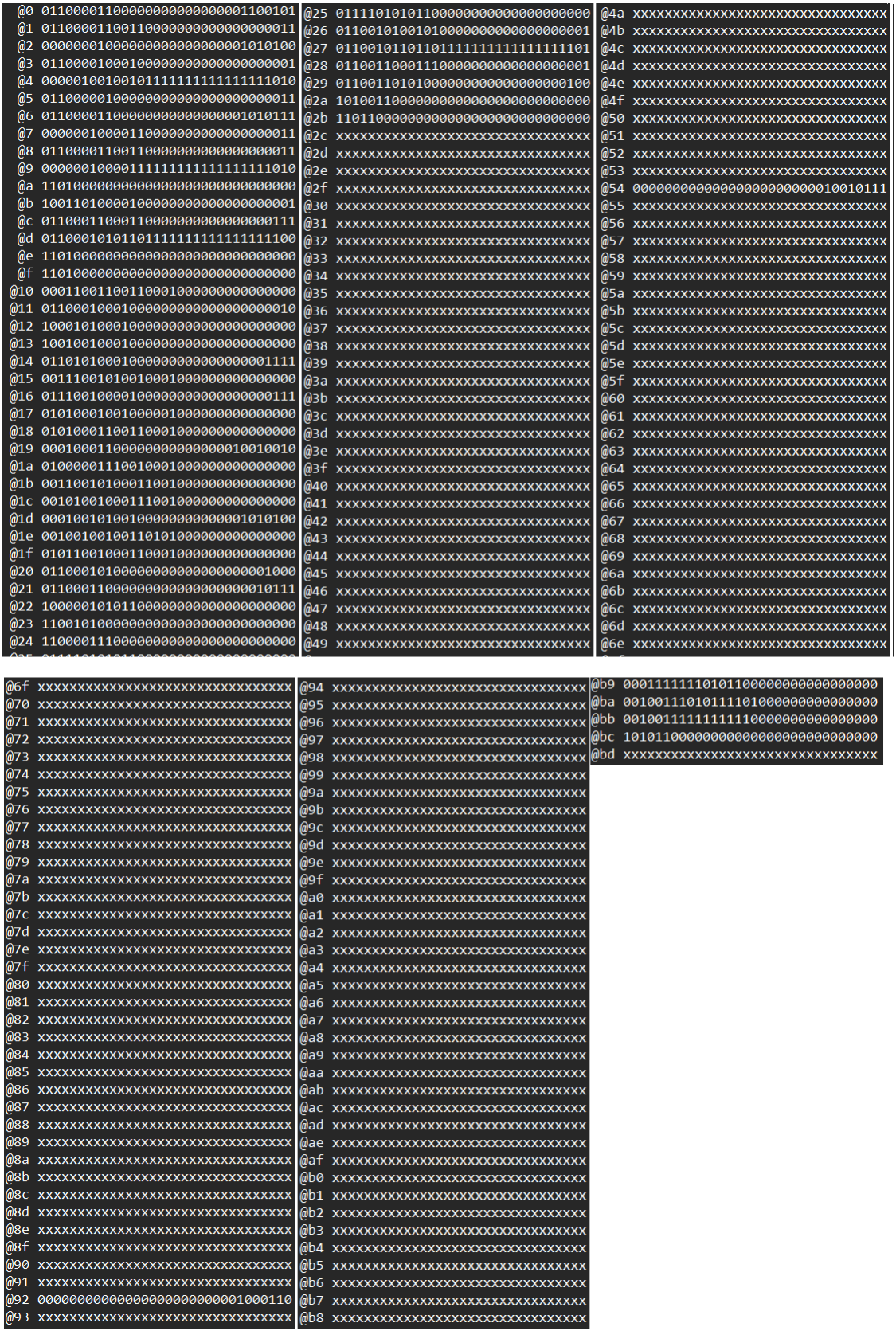

Figure 11: Memory contents before execution

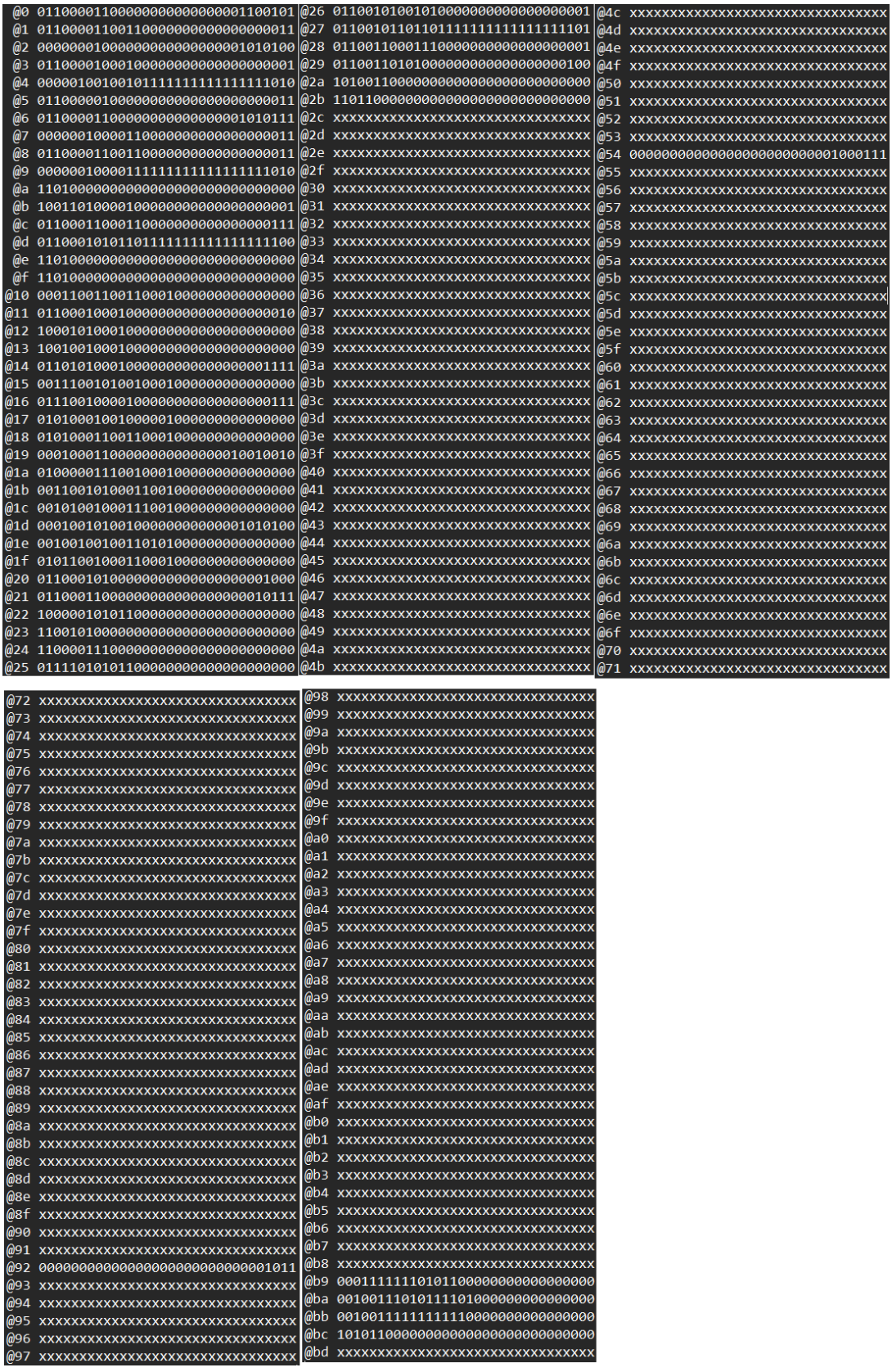

Figure 12: Memory contents after execution

Control Unit Code

Figure 13, Figure 14, Figure 15, Figure 16, Figure 17, Figure 18 and Figure 19 show the code for the control unit. Notice that the control unit uses a decode module, the code for that is in Figure 20. Figure 21 shows the implementation of the control unit in the processor.

```
1  `include "ISA.vh"
2  `include "ALU.vh"
3
4
5  module Control (
6      // Clock, reset and ready signals
7      // Ready is an active high that allows the next step to continue
8      iClk, nRst, iRdy,
9      // Memory Signals/Control
10     iMemData, oMemRead, oMemWrite,
11     // Pipe Control
12     oPipe_nRst,
13     // Program Counter Control
14     oPC_nRst, oPC_en, oPC_tmpEn, oPC_load, oPC_offset,
15     // Register File Control
16     oRF_Write,
17     oRF_AddrA, oRF_AddrB, oRF_AddrC,
18     // Write Back Register Control
19     oRWB_en,
20     // ALU Control
21     oALU_Ctrl, oRA_en, oRB_en,
22     oRZH_en, oRZL_en, oRAS_en,
23     // Jump Feedback
24     iJ_zero, iJ_nZero, iJ_pos, iJ_neg,
25     // Port Register enable
26     oREP_en,
27     // Multiplexers
28     oMUX_BIS, oMUX_RZHS, oMUX_WBM, oMUX_MAP, oMUX_ASS, oMUX_WBP, oMUX_WBE,
29     // Imm32 Output
30     oImm32
31 );
32
33 // Clock, reset and ready signals
34 // Ready is an active high that allows the next step to continue
35 input wire iClk, nRst, iRdy;
36 // Memory Signals/Control
37 input wire [31:0] iMemData;
38 output wire oMemRead, oMemWrite;
39 // Pipe Control
40 output wire oPipe_nRst;
41 // Program Counter Control
42 output wire oPC_nRst, oPC_en, oPC_tmpEn, oPC_load, oPC_offset;
```

Figure 13: Control unit code part 1

```
43   // Register File Control
44   output wire oRF_Write;
45   output wire [3:0] oRF_AddrA, oRF_AddrB, oRF_AddrC;
46   // Write Back Register Control
47   output wire oRWB_en;
48   // ALU Control
49   output wire [3:0] oALU_Ctrl;
50   output wire oRA_en, oRB_en;
51   output wire oRZH_en, oRZL_en, oRAS_en;
52   // Jump Feedback
53   input wire iJ_zero, iJ_nZero, iJ_pos, iJ_neg;
54   // External Port Enable
55   output wire oREP_en;
56   // Multiplexers
57   output wire oMUX_BIS, oMUX_RZHS, oMUX_WBM, oMUX_MAP, oMUX_ASS, oMUX_WBP, oMUX_WBE;
58   // Imm32 Output
59   output wire [31:0] oImm32;
60
61   // Step Counter
62   reg [5:1] Cycle;
63
64   // IR
65   wire IR_en;
66   wire [31:0] IR_out;
67
68   // Decoder IO
69   wire [3:0] ID_RA, ID_RB, ID_RC;
70   wire [4:0] ID_OpCode;
71   wire [31:0] ID_imm32, ID_BRD;
72   wire [1:0] ID_BRC;
73
74   // OpCode R-Format Wires
75   wire OP_LD, OP_LI,  OP_ST,  OP_ADD, OP_SUB, OP_AND,
76        OP_OR, OP_ROR, OP_ROL, OP_SRL, OP_SRA, OP_SLL;
77   // OpCode I-Format Wires
78   wire OP_ADDI, OP_ANDI, OP_ORI, OP_DIV, OP_MUL, OP_NEG, OP_NOT;
79   // OpCode B-Format Wires
80   wire OP_BRx;
81   // OpCode J-Format Wires
82   wire OP_JAL, OP_JFR, OP_IN, OP_OUT, OP_MFL, OP_MFH;
83   // OpCode M-Format Wires
84   wire OP_NOP, OP_HLT;
```

Figure 14: Control unit code part 2

```
85   // OpCode Format Wires
86   // (Useful for data path MUX Assignments)
87   wire OPF_R, OPF_I, OPF_B, OPF_J, OPF_M;
88   // Branch Conditional Wires
89   wire BR_ZERO, BR_NZRO, BR_POS, BR_NEG;
90   wire BR_TRUE;
91
92   // Assign Cycle
93   always @(posedge iClk or negedge nRst)
94   begin
95       if(!nRst)
96           Cycle = 5'b00001;
97       else begin
98           if(iRdy && !OP_HLT) Cycle = {Cycle[4:1], Cycle[5]};
99       end
100  end
101
102  // Instruction Register
103  assign IR_en = Cycle[1];
104  REG32 IR(.iClk(iClk), .nRst(nRst), .iEn(IR_en), .iD(iMemData), .oQ(IR_out));
105
106  // Decoder
107  Decode decoder(
108      .iINS(IR_out),
109      .oImm32(ID_imm32),
110      .oRa(ID_RA),
111      .oRb(ID_RB),
112      .oRc(ID_RC),
113      .oCode(ID_OpCode),
114      // Branch Distance
115      .oBRD(ID_BRD),
116      // Branch Code
117      .oBRC(ID_BRC)
118  );
119
120  // Assign OP-Code Types
121
122  // Assign R-Format Wires
123  assign OP_ADD = (ID_OpCode == `ISA_ADD);
124  assign OP_SUB = (ID_OpCode == `ISA_SUB);
125  assign OP_AND = (ID_OpCode == `ISA_AND);
126  assign OP_OR  = (ID_OpCode == `ISA_OR);
127  assign OP_ROR = (ID_OpCode == `ISA_ROR);
128  assign OP_ROL = (ID_OpCode == `ISA_ROL);
129  assign OP_SRL = (ID_OpCode == `ISA_SRL);
```

Figure 15: Control unit code part 4

```
130   assign OP_SRA = (ID_OpCode == `ISA_SRA);
131   assign OP_SLL = (ID_OpCode == `ISA_SLL);
132   // Opcode Format Wire (Useful for data path MUX Assignments)
133   assign OPF_R  = (OP_ADD || OP_SUB || OP_AND || OP_OR || OP_ROR || OP_ROL || OP_SRL || OP_SRA || OP_SLL);
134   // Assign I-Format Wires
135   assign OP_LD  = (ID_OpCode == `ISA_LD);
136   assign OP_LI  = (ID_OpCode == `ISA_LI);
137   assign OP_ST  = (ID_OpCode == `ISA_ST);
138   assign OP_ADDI = (ID_OpCode == `ISA_ADDI);
139   assign OP_ANDI = (ID_OpCode == `ISA_ANDI);
140   assign OP_ORI  = (ID_OpCode == `ISA_ORI);
141   assign OP_DIV  = (ID_OpCode == `ISA_DIV);
142   assign OP_MUL  = (ID_OpCode == `ISA_MUL);
143   assign OP_NEG  = (ID_OpCode == `ISA_NEG);
144   assign OP_NOT  = (ID_OpCode == `ISA_NOT);
145   // Opcode Format Wire (Useful for data path MUX Assignments)
146   assign OPF_I   = (OP_LD || OP_LI || OP_ST || OP_ADDI || OP_ANDI || OP_ORI || OP_DIV || OP_MUL || OP_NEG || OP_NOT);
147   // Assign B-Format Wires
148   assign OP_BRx = (ID_OpCode == `ISA_BRx);
149   // Opcode Format Wire (Useful for data path MUX Assignments)
150   assign OPF_B = OP_BRx;
151   // Assign J-Format Wires
152   assign OP_JAL = (ID_OpCode == `ISA_JAL);
153   assign OP_JFR = (ID_OpCode == `ISA_JFR);
154   assign OP_IN  = (ID_OpCode == `ISA_IN);
155   assign OP_OUT = (ID_OpCode == `ISA_OUT);
156   assign OP_MFL = (ID_OpCode == `ISA_MFL);
157   assign OP_MFH = (ID_OpCode == `ISA_MFH);
158   // Opcode Format Wire (Useful for data path MUX Assignments)
159   assign OPF_J  = (OP_JAL || OP_JFR || OP_MFL || OP_MFH || OP_IN || OP_OUT);
160   // Assign M-Format Wires
161   assign OP_NOP = (ID_OpCode == `ISA_NOP);
162   assign OP_HLT = (ID_OpCode == `ISA_HLT);
163   // Opcode Format Wire (Useful for data path MUX Assignments)
164   assign OPF_M  =  (OP_NOP || OP_HLT);
165
166   // Assign Branch Wires
167   // iJ_xxx based on RF_RB in data path
168   assign BR_ZERO = (ID_BRC == `ISA_BR_ZERO) && iJ_zero;
169   assign BR_NZRO = (ID_BRC == `ISA_BR_NZRO) && iJ_nZero;
170   assign BR_POS = (ID_BRC == `ISA_BR_POSI) &&  iJ_pos;
171   assign BR_NEG = (ID_BRC == `ISA_BR_NEGA) &&  iJ_neg;
172   assign BR_TRUE = (BR_ZERO || BR_NZRO || BR_POS || BR_NEG) && OP_BRx;
```

Figure 16: Control unit code part 5

```
174  // Assign Control outputs based on Codes and Cycle
175
176  // Pipe Reset Signal
177  assign oPipe_nRst = nRst;
178
179  // Program Counter Control Signals
180  // PC Reset (Should only be reset on CPU reset)
181  assign oPC_nRst = nRst;
182  // PC Load Enable
183  assign oPC_en = Cycle[1] || (Cycle[3] && (BR_TRUE || OP_JAL || OP_JFR));
184  assign oPC_tmpEn = Cycle[1];
185  // PC Jump Enable
186  assign oPC_offset = Cycle[3] && BR_TRUE;
187  assign oPC_load = Cycle[3] && (OP_JFR || OP_JAL);
188
189  // Register File Control Signals
190  assign oRF_Write = Cycle[5] && (OPF_R || (OPF_I && ~OP_DIV && ~OP_MUL && ~OP_ST) || OP_MFH || OP_MFL || OP_JAL || OP_IN);
191  // Note: Most ISA's use RC as the write back address, MiniSRC uses RA
192  // RA is dependent on ISA type, use R0 if RA is not specified
193  // RA is used to load PC on JMP/JAL
194  assign oRF_AddrA =  (OPF_R | OPF_I) ? ID_RB :
195                      (OPF_J) ? ID_RA : 4'h0;
196  // RB is dependent on ISA type, use R0 if RB is not specified
197  assign oRF_AddrB =  (OPF_I | OPF_B | OPF_J) ? ID_RA :
198                      (OPF_R) ? ID_RC : 4'h0;
199  // Store is always RA
200  // ISA Specification states to store PC in r15 on JAL (Jump and Link)
201  assign oRF_AddrC = (OP_JAL) ? 4'h8 : ID_RA;
202
203  // Register File Write Back Register Load Enable
204  assign oRWB_en = 1'b1;
```

Figure 17: Control unit code part 6

```
206   // ALU Control Signals Also, this should be renamed to "Ctrl" like the key on the keyboard.
207  assign oALU_Ctrl =  (OP_ADD || OP_ADDI) ? `CTRL_ALU_ADD :
208                      (OP_SUB)            ? `CTRL_ALU_SUB :
209                      (OP_OR  || OP_ORI)  ? `CTRL_ALU_OR  :
210                      (OP_AND || OP_ANDI) ? `CTRL_ALU_AND :
211                      (OP_MUL)            ? `CTRL_ALU_MUL :
212                      (OP_DIV)            ? `CTRL_ALU_DIV :
213                      (OP_SLL)            ? `CTRL_ALU_SLL :
214                      (OP_SRL)            ? `CTRL_ALU_SRL :
215                      (OP_SRA)            ? `CTRL_ALU_SRA :
216                      (OP_ROR)            ? `CTRL_ALU_ROR :
217                      (OP_ROL)            ? `CTRL_ALU_ROL :
218                      (OP_NOT)            ? `CTRL_ALU_NOT :
219                      (OP_NEG)            ? `CTRL_ALU_NEG :
220                      // ALU Add is default for most instructions - so why not remove the (OP_ADD || OP_ADDI) ?
221                      `CTRL_ALU_ADD;
```

Figure 18: Control unit code part 7

```
222   // ALU Input A Register Load Enable
223   assign oRA_en = 1'b1;
224   // ALU Input B Register Load Enable
225   assign oRB_en = 1'b1;
226
227   // ALU Result High Load EN
228   assign oRZH_en = 1'b1;
229   // ALU Result Low Load EN
230   assign oRZL_en = 1'b1;
231   // ALU Result Save EN
232   assign oRAS_en = (OP_DIV || OP_MUL);
233
234   // External Port Register Enable
235   assign oREP_en = OP_OUT && Cycle[4];
236
237   // ALU B Input Select (Selects Imm)
238   assign oMUX_BIS = OPF_I && ~(OP_DIV || OP_MUL);
239   // ALU Result High Select
240   assign oMUX_RZHS = (OP_MFH);
241   // RF Write Back Select
242   assign oMUX_WBM = (OP_LD || OP_LI);
243   // Memory Address Output Select
244   // assign oMUX_MA = Cycle[1];
245   assign oMUX_MAP = ~((OP_LD || OP_ST || OP_LI) && Cycle[4]);
246   // ALU Storage Select
247   assign oMUX_ASS = (OP_MFL || OP_MFH);
248   // Write Back Program Counter Select
249   assign oMUX_WBP = OP_JAL;
250   // Write Back External Port Select
251   assign oMUX_WBE = OP_IN;
252
253   // Immediate value output
254   // Assign Imm32 branch distance if the branch is true
255   assign oImm32 = OP_BRx ? ID_BRD : ID_imm32;
256
257   // Memory Read/Write Signals
258   assign oMemRead = Cycle[1] || (Cycle[4] && (OP_LD || OP_LI));
259   assign oMemWrite = Cycle[4] && OP_ST;
260
261   endmodule
```

Figure 19: Control unit code part 8

```
module Decode (
    // Input Instruction
    iINS,
    // Immediate Value (Sign Extended)
    oImm32,
    // Reg A, Reg B, Reg C addresses
    oRa, oRb, oRc,
    // OP Code
    oCode,
    // Branch Distance
    oBRD,
    // Branch Code
    oBRC
);

// Taken from instruction formats: section 2.1 of the Processor specifications

input wire [31:0] iINS;
output wire [31:0] oImm32;
output wire [3:0] oRa, oRb, oRc;
output wire [4:0] oCode;
output wire [31:0] oBRD;
output wire [1:0] oBRC;

assign oCode  = iINS[31:27];
assign oRa    = iINS[26:23];
assign oRb    = iINS[22:19];
assign oRc    = iINS[18:15];
assign oImm32 = {{13{iINS[18]}}, iINS[18:0]};
assign oBRD   = {{11{iINS[18]}}, iINS[18:0], 2'b00};
assign oBRC   = iINS[20:19];

endmodule
```

Figure 20: Decode module code

```
// Control Unit
Control Ctrl(
    // Clock, reset and ready signals
    // Ready is an active high that allows the next step to continue
    .iClk(iClk),
    .nRst(nRst),
    .iRdy(iMemRdy),
    // Memory Signals/Control
    .iMemData(iMemData),
    .oMemRead(oMemRead),
    .oMemWrite(oMemWrite),
    // Pipe Control
    .oPipe_nRst(pipe_rst),
    // Program Counter Control
    .oPC_nRst(PC_nRst),
    .oPC_en(PC_en),
    .oPC_tmpEn(PC_tmpEn),
    .oPC_load(PC_load),
    .oPC_offset(PC_offset),
    // Register File Control
    .oRF_Write(RF_iWrite),
    .oRF_AddrA(RF_iAddrA),
    .oRF_AddrB(RF_iAddrB),
    .oRF_AddrC(RF_iAddrC),
    .oRWB_en(RWB_en),
    // ALU Control
    .oALU_Ctrl(ALU_iCtrl),
    .oRA_en(RA_en),
    .oRB_en(RB_en),
    .oRZH_en(RZH_en),
    .oRZL_en(RZL_en),
    .oRAS_en(RAS_en),
    // Jump Feedback
    .iJ_zero(J_zero),
    .iJ_nZero(J_nZero),
    .iJ_pos(J_pos),
    .iJ_neg(J_neg),
    // External Port Register Enable
    .oREP_en(REP_en),
    // Multiplexers
    .oMUX_BIS(MUX_BIS),
    .oMUX_RZHS(MUX_RZHS),
    .oMUX_WBM(MUX_WBM),
    .oMUX_MAP(MUX_MAP),
    .oMUX_ASS(MUX_ASS),
    .oMUX_WBP(MUX_WBP),
    .oMUX_WBE(MUX_WBE),
    // Imm32 Output
    .oImm32(CT_imm32)
);
```

Figure 21: Implementation of control unit in the processor