374 Phase 3 Report

March 31, 2023

Group 4:
Liam Salass (20229595)
Abdellah Ghassel (20230384)

"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

Table of Figures

Introduction & Motivation

In this phase, we focus on the testing of the Control Unit in the Mini SRC, which is a critical component for managing the overall operation of the processor. By designing the Control Unit in Verilog, we aim to consolidate the work accomplished in the previous phases and bring the Mini SRC to a fully functional state.

The motivation behind Phase 3 of the CPU Design Project is to gain a thorough understanding of the Control Unit's role in orchestrating the Mini SRC's operation. By analyzing the interaction between the Control Unit and the datapath components, we aim to ensure the seamless execution of instructions and to further enhance the overall performance of the Mini SRC. Furthermore, this phase provides an opportunity to delve into the intricacies of control signal generation, branching and instruction flow management, and external inputs handling.

The Control Unit

A 32 bit control unit is a component of a CPU that controls the execution of instructions. It functions as a finite state machine, which means that it has a set of states and transitions between them based on some inputs and outputs. The inputs to the control unit are the instruction register (IR) opcode data, which specify the type and format of the instruction to be executed, and the clock signal, which synchronizes the operation of the control unit with other components. The outputs are the control signals that are sent to the arithmetic logic unit (ALU), the registers (Ra, Rb, Rc, Rin, Rout), and other components to perform the required operations.

Depending on the instruction, different states are sequentially cycled through by the control unit. For example, a typical instruction cycle may consist of 3 states: fetch, decode, and execute. In each state, different control signals are sent out to perform specific tasks. For instance, in the fetch state, the control unit sends a signal to the memory to read the instruction from the program counter (PC) address and store it in the IR. It also increments the PC by 1 to point to the next instruction. In the decode state, the control unit decodes the opcode data in the IR and determines the type and format of the instruction. It also sends signals to read the operands from the registers or memory as specified by the instruction. In the execute state, the control unit sends signals to the ALU to perform the required arithmetic or logical operation on the operands and produce a result. It also sets or clears some flags based on the outcome of the operation. The execute state can be broken down into specific instruction states. For example, a load instructions execution state can be broken down into 5 separate substates, where different signals are sent out sequentially in order to properly execute the instruction.

The control unit can also handle different types of instructions such as R-type, I-type, J-type, etc. Each type of instruction has a different format and requires different control signals. For example, an R-type instruction has three register operands (Ra, Rb, Rc) and an ALU operation code. An I-type instruction has two register operands (Ra, Rb) and an immediate value. A J-type instruction has a jump address. The control unit can decode these formats and send appropriate signals to execute them.

One of the instructions that the control unit can handle is Xor, which performs a bitwise exclusive OR operation on two operands. The Xor instruction was not part of the original CPU specs document, but it was added later. The Xor instruction was implemented by adding a new IR opcodes (11110 in the first 5 bits), a new XOR alu function, and new XOR execution states to the control unit.

```verilog
//Control unit for CPU
//Finite state machine logic for control
`timescale 1ns/1ps
module ctrl_unit(
    //Test Bench inputs/outputs but goes through datapath
    input clk, reset, stop,
    output reg run, clear,
    //Datapath inputs/outputs
    input [31:0] IRdata,
    output reg read, write,
    output reg BAout, Rin, Rout,
    output reg Gra, Grb, Grc,
    output reg CONN_in,
    output reg MARin, MDRin,
    output reg HIin, LOin,
    output reg Yin, Zin,
    output reg PCin, IRin, incPC,
    output reg InPortIn, OutPortIn,
    output reg HIout, LOout, ZLowOut, ZHighOut,
    output reg MDRout, Cout, InPortOut, PCout,
    output reg [4:0] alu_opcode,
    output reg jal_flag
);
    wire [4:0] ir_op;

    assign ir_op = IRdata[31:27];

    //Control unit states
    parameter   reset_state = 8'b00000000,
                //Fetch states
                fetch0 = 8'b00000001, fetch1 = 8'b00000010, fetch2= 8'b00000011,
                //Add states
                add3 = 8'b00000100, add4= 8'b00000101, add5= 8'b00000110,
                //Sub states
                sub3 = 8'b00000111, sub4 = 8'b00001000, sub5 = 8'b00001001,
                //Multiplication states
                mul3 = 8'b00001010, mul4 = 8'b00001011, mul5 = 8'b00001100, mul6 =
8'b00001101,
                //Division states
                div3 = 8'b00001110, div4 = 8'b00001111,div5 = 8'b00010000, div6 = 8'b00010001,
                //Or states
                or3 = 8'b00010010, or4 = 8'b00010011, or5 = 8'b00010100,
                //Xor states
                xor3 = 8'b10010101, xor4 = 8'b10010110, xor5 = 8'b10011111,
                //And states
                and3 = 8'b00010101, and4 = 8'b00010110, and5 = 8'b00010111,
                //Shift left states
                shl3 = 8'b00011000, shl4 = 8'b00011001, shl5 = 8'b00011010,
                //Shift right states
                shr3 = 8'b00011011, shr4 = 8'b00011100,  shr5 = 8'b00011101,
                //Shift right arithmatic states
                shra3 = 8'b10011011, shra4 = 8'b10011100,  shra5 = 8'b10011101,
                //Rotate left states
                rol3 = 8'b00011110, rol4 = 8'b00011111, rol5 = 8'b00100000,
                //Rotate right states
                ror3 = 8'b00100001, ror4 = 8'b00100010,  ror5 = 8'b00100011,
                //Negate states
                neg3 = 8'b00100100, neg4 = 8'b00100101, neg5 = 8'b00100110,
                //Not states
                not3 = 8'b00100111, not4 = 8'b00101000,  not5 = 8'b00101001,
                //Load states
                ld3 = 8'b00101010, ld4 = 8'b00101011, ld5 = 8'b00101100, ld6 = 8'b00101101,
ld7 = 8'b00101110,
                //Load immediate states
                ldi3 = 8'b00101111, ldi4 = 8'b00110000, ldi5 = 8'b00110001,
                //Store States
                st3 = 8'b00110010, st4 = 8'b00110011, st5 = 8'b00110100, st6 = 8'b00110101,
st7 = 8'b00110110,
                //Add immediate states
                addi3 = 8'b00110111, addi4 = 8'b00111000, addi5 = 8'b00111001,
                //And immediate states
                andi3 = 8'b00111010, andi4 = 8'b00111011, andi5 = 8'b00111100,
                //Or immediate states
                ori3 = 8'b00111101, ori4 = 8'b00111110, ori5 = 8'b00111111,
                //Branch if States
                br3 = 8'b01000000, br4 = 8'b01000001, br5 = 8'b01000010, br6 = 8'b01000011,
                //Move reg to PC state
                jr3 = 8'b01000100,
                //Move PC to reg state
                jal3 = 8'b01000101, jal4 = 8'b10000001,
                //Move HI to reg state
                mfhi3 = 8'b01000111,
                //Move LO to reg state
                mflo3 = 8'b01001000,
                //Move Inport to reg state
                in3 = 8'b01001001,
                //Move reg to Outport state
                out3 = 8'b01001010,
                //No operation state (go to fetch0)
                nop3 = 8'b01001011,
                //Halt state
                halt = 8'b01001100;

    //ALU opcodes
    parameter   alu_nop = 5'b00000,
                alu_addop = 5'b00001,
                alu_subop = 5'b00010,
                alu_mulop = 5'b00011,
                alu_divop = 5'b00100,
                alu_shrop = 5'b00101,
                alu_shlop = 5'b00110,
                alu_shraop = 5'b00111,
                alu_rorop = 5'b01000,
                alu_rolop = 5'b01001,
                alu_andop = 5'b01010,
                alu_orop = 5'b01011,
                alu_negop = 5'b01100,
                alu_xorop = 5'b01101,
                alu_norop = 5'b01110,
                alu_notop = 5'b01111;


    //IR opcodes
    parameter   ir_add  = 5'b00011,
                ir_sub  = 5'b00100,
                ir_mul  = 5'b01111,
                ir_div  = 5'b10000,
                ir_shl  = 5'b01001,
                ir_shr  = 5'b00111,
                ir_shra = 5'b01000,
                ir_rol  = 5'b01011,
                ir_ror  = 5'b01010,
                ir_and  = 5'b00101,
                ir_or   = 5'b00110,
                ir_xor  = 5'b11110, //Xor opcode is 11110
                ir_neg  = 5'b10001,
                ir_not  = 5'b10010,
                ir_ld   = 5'b00000,
                ir_ldi  = 5'b00001,
                ir_st   = 5'b00010,
                ir_addi = 5'b01100,
                ir_andi = 5'b01101,
                ir_ori  = 5'b01110,
                ir_br   = 5'b10011,
                ir_jr   = 5'b10100,
                ir_jal  = 5'b10101,
                ir_mfhi = 5'b11000,
                ir_mflo = 5'b11001,
                ir_in   = 5'b10110,
                ir_out  = 5'b10111,
                ir_nop  = 5'b11010,
                ir_halt = 5'b11011;


    reg [7:0] present_state = halt;
```

```
always @(posedge clk, posedge reset, posedge stop) begin

    if (reset) begin    //reset the processor
        #5 present_state = reset_state;
        #10 present_state = fetch0;
    end

    if (stop) begin
        present_state = halt;
    end

    case (present_state)
        fetch0 : #40 present_state = fetch1;
        fetch1 : #40 present_state = fetch2;
        fetch2 : begin
                #40
                case (ir_op)
                ir_add  :  present_state = add3;
                ir_sub  :  present_state = sub3;
                ir_mul  :  present_state = mul3;
                ir_div  :  present_state = div3;
                ir_shl  :  present_state = shl3;
                ir_shr  :  present_state = shr3;
                ir_shra :  present_state = shra3;
                ir_rol  :  present_state = rol3;
                ir_ror  :  present_state = ror3;
                ir_and  :  present_state = and3;
                ir_or   :  present_state = or3;
                ir_xor  :  present_state = xor3;
                ir_neg  :  present_state = neg3;
                ir_not  :  present_state = not3;
                ir_ld   :  present_state = ld3;
                ir_ldi  :  present_state = ldi3;
                ir_st   :  present_state = st3;
                ir_addi :  present_state = addi3;
                ir_andi :  present_state = andi3;
                ir_ori  :  present_state = ori3;
                ir_br   :  present_state = br3;
                ir_jr   :  present_state = jr3;
                ir_jal  :  present_state = jal3;
                ir_mfhi :  present_state = mfhi3;
                ir_mflo :  present_state = mflo3;
                ir_in   :  present_state = in3;
                ir_out  :  present_state = out3;
                ir_nop  :  present_state = fetch0;
                ir_halt :  present_state = halt;
            endcase
        end
        //Add instruction
        add3    : #40 present_state = add4;
        add4    : #40 present_state = add5;
        add5    : #40 present_state = fetch0;

        //Sub instruction
        sub3    : #40 present_state = sub4;
        sub4    : #40 present_state = sub5;
        sub5    : #40 present_state = fetch0;

        //Mul instruction
        mul3    : #40 present_state = mul4;
        mul4    : #40 present_state = mul5;
        mul5    : #40 present_state = mul6;
        mul6    : #40 present_state = fetch0;

        //Div instruction
        div3    : #40 present_state = div4;
        div4    : #40 present_state = div5;
        div5    : #40 present_state = div6;
        div6    : #40 present_state = fetch0;

        //Or instruction
        or3     : #40 present_state = or4;
        or4     : #40 present_state = or5;
        or5     : #40 present_state = fetch0;

        //Xor instruction
        xor3    : #40 present_state = xor4;
        xor4    : #40 present_state = xor5;
        xor5    : #40 present_state = fetch0;

        //And instruction
        and3    : #40 present_state = and4;
        and4    : #40 present_state = and5;
        and5    : #40 present_state = fetch0;

        //Shift left instrcutions
        shl3    : #40 present_state = shl4;
        shl4    : #40 present_state = shl5;
        shl5    : #40 present_state = fetch0;
```

```
            //Shift right instructions
            shr3    : #40 present_state = shr4;
            shr4    : #40 present_state = shr5;
            shr5    : #40 present_state = fetch0;

            //Shift right arithmatic instructions
            shra3   : #40 present_state = shra4;
            shra4   : #40 present_state = shra5;
            shra5   : #40 present_state = fetch0;

            //Rotate left instructions
            rol3    : #40 present_state = rol4;
            rol4    : #40 present_state = rol5;
            rol5    : #40 present_state = fetch0;

            //Rotate right instructions
            ror3    : #40 present_state = ror4;
            ror4    : #40 present_state = ror5;
            ror5    : #40 present_state = fetch0;

            //Negate instructions
            neg3    : #40 present_state = neg4;
            neg4    : #40 present_state = neg5;
            neg5    : #40 present_state = fetch0;

            //Not instructions
            not3    : #40 present_state = not4;
            not4    : #40 present_state = not5;
            not5    : #40 present_state = fetch0;

            //Load instructions
            ld3     : #40 present_state = ld4;
            ld4     : #40 present_state = ld5;
            ld5     : #40 present_state = ld6;
            ld6     : #40 present_state = ld7;
            ld7     : #40 present_state = fetch0;

            //Load immediate instructions
            ldi3    : #40 present_state = ldi4;
            ldi4    : #40 present_state = ldi5;
            ldi5    : #40 present_state = fetch0;

            //Store instructions
            st3     : #40 present_state = st4;
            st4     : #40 present_state = st5;
            st5     : #40 present_state = st6;
            st6     : #40 present_state = st7;
            st7     : #40 present_state = fetch0;

            //Add immediate instructions
            addi3   : #40 present_state = addi4;
            addi4   : #40 present_state = addi5;
            addi5   : #40 present_state = fetch0;

            //And immediate instructions
            andi3   : #40 present_state = andi4;
            andi4   : #40 present_state = andi5;
            andi5   : #40 present_state = fetch0;

            //Or immediate instructions
            ori3    : #40 present_state = ori4;
            ori4    : #40 present_state = ori5;
            ori5    : #40 present_state = fetch0;

            //Branch instructions
            br3     : #40 present_state = br4;
            br4     : #40 present_state = br5;
            br5     : #40 present_state = br6;
            br6     : #40 present_state = fetch0;

            //Jump register instructions
            jr3     : #40 present_state = fetch0;

            //Jump and link instructions
            jal3    : #40 present_state = jal4;
            jal4    : #40 present_state = fetch0;

            //Move from HI instructions
            mfhi3   : #40 present_state = fetch0;

            //Move from LO instructions
            mflo3   : #40 present_state = fetch0;

            //Input instructions
            in3     : #40 present_state = fetch0;

            //Output instructions
            out3    : #40 present_state = fetch0;

            //If restart, go to halt
```

```
        reset_state : begin
            #40 present_state = fetch0;
        end

        //halt state
        halt : begin
            present_state = halt;
        end
    endcase

end

always @(present_state) begin
    case (present_state)
        reset_state : begin
            run = 1; clear = 1;
            read = 0; write = 0;
            Gra = 0; Grb = 0; Grc = 0; Rin = 0;
            BAout = 0; Rin = 0; Rout = 0;
            CONN_in = 0;
            MARin = 0; MDRin = 0;
            HIin = 0; LOin = 0;
            Yin = 0; Zin = 0;
            PCin = 0; IRin = 0; incPC = 0;
            InPortIn = 0; OutPortIn = 0;
            jal_flag = 0;
            HIout = 0; LOout = 0; ZLowOut = 0; ZHighOut = 0;
            MDRout = 0; Cout = 0; InPortOut = 0; PCout = 0;
            #3 clear = 0; run = 0;
        end

        fetch0: begin
            #10 PCout = 1; MARin = 1; Zin = 1; incPC = 1;
            #15 PCout = 0; MARin = 0; Zin = 0; incPC = 0;
        end
        fetch1: begin
            #10 ZLowOut = 1; PCin = 1; read = 1; MDRin = 1;
            #15 ZLowOut = 0; PCin = 0; read = 0; MDRin = 0;
        end
        fetch2: begin
            #10 MDRout = 1; IRin = 1;
            #10 MDRout = 0; IRin = 0;
        end

        //Alu instruction states
```

```
            add3, sub3, and3, or3, shl3, shr3, shra3, ror3, rol3, addi3, andi3, ori3, neg3,
not3, xor3: begin
                #10 Grb = 1; Rout = 1; Yin = 1;
                #15 Grb = 0; Rout = 0; Yin = 0;
            end
            mul3, div3: begin
                #10 Gra = 1; Rout = 1; Yin = 1;
                #15 Gra = 0; Rout = 0; Yin = 0;
            end
            add4 : begin
                #10 Zin = 1; Grc = 1; Rout = 1; alu_opcode = alu_addop;
                #15 Zin = 0; Grc = 0; Rout = 0; alu_opcode = alu_nop;
            end
            sub4 : begin
                #10 Zin = 1; Grc = 1; Rout = 1; alu_opcode = alu_subop;
                #15 Zin = 0; Grc = 0; Rout = 0; alu_opcode = alu_nop;
            end
            and4 : begin
                #10 Zin = 1; Grc = 1; Rout = 1; alu_opcode = alu_andop;
                #15 Zin = 0; Grc = 0; Rout = 0; alu_opcode = alu_nop;
            end
            or4 : begin
                #10 Zin = 1; Grc = 1; Rout = 1; alu_opcode = alu_orop;
                #15 Zin = 0; Grc = 0; Rout = 0; alu_opcode = alu_nop;
            end
            xor4 : begin
                #10 Zin = 1; Grc = 1; Rout = 1; alu_opcode = alu_xorop;
                #15 Zin = 0; Grc = 0; Rout = 0; alu_opcode = alu_nop;
            end
            shl4 : begin
                #10 Zin = 1; Grc = 1; Rout = 1; alu_opcode = alu_shlop;
                #15 Zin = 0; Grc = 0; Rout = 0; alu_opcode = alu_nop;
            end
            shr4 : begin
                #10 Zin = 1; Grc = 1; Rout = 1; alu_opcode = alu_shrop;
                #15 Zin = 0; Grc = 0; Rout = 0; alu_opcode = alu_nop;
            end
            shra4 : begin
                #10 Zin = 1; Grc = 1; Rout = 1; alu_opcode = alu_shraop;
                #15 Zin = 0; Grc = 0; Rout = 0; alu_opcode = alu_nop;
            end
            ror4  :begin
                #10 Zin = 1; Grc = 1; Rout = 1; alu_opcode = alu_rorop;
                #15 Zin = 0; Grc = 0; Rout = 0; alu_opcode = alu_nop;
            end
            rol4 : begin
                #10 Zin = 1; Grc = 1; Rout = 1; alu_opcode = alu_rolop;
                #15 Zin = 0; Grc = 0; Rout = 0; alu_opcode = alu_nop;
            end
            addi4  :begin
                #10 Zin = 1; Cout = 1; alu_opcode = alu_addop;
                #15 Zin = 0; Cout = 0; alu_opcode = alu_nop;
            end
            andi4 : begin
                #10 Zin = 1; Cout = 1; alu_opcode = alu_andop;
                #15 Zin = 0; Cout = 0; alu_opcode = alu_nop;
            end
            ori4 : begin
                #10 Zin = 1; Cout = 1; alu_opcode = alu_orop;
                #15 Zin = 0; Cout = 0; alu_opcode = alu_nop;
            end
            mul4 : begin
                #10 Zin = 1; Grb = 1; Rout = 1; alu_opcode = alu_mulop;
                #15 Zin = 0; Grb = 0; Rout = 0; alu_opcode = alu_nop;
            end
            div4 : begin
                #10 Zin = 1; Grb = 1; Rout = 1; alu_opcode = alu_divop;
                #15 Zin = 0; Grb = 0; Rout = 0; alu_opcode = alu_nop;
            end
            neg4 : begin
                #10 Zin = 1; alu_opcode = alu_negop;
                #15 Zin = 0; alu_opcode = alu_nop;
            end
            not4 : begin
                #10 Zin = 1; alu_opcode = alu_notop;
                #15 Zin = 0; alu_opcode = alu_nop;
            end
            add5, sub5, and5, or5, shl5, shr5, shra5, ror5, rol5, andi5, addi5, ori5, not5,
neg5, xor5: begin
                #10 ZLowOut = 1; Gra = 1; Rin = 1;
                #15 ZLowOut = 0; Gra = 0; Rin = 0;
            end
            mul5, div5 : begin
                #10 ZLowOut = 1; LOin = 1; Rin = 1;
                #15 ZLowOut = 0; LOin = 0; Rin = 0;
            end
            mul6, div6 : begin
                #10 ZHighOut = 1; HIin = 1; Rin = 1;
                #15 ZHighOut = 0; HIin = 0; Rin = 0;
            end
```

```
//Load instruction states
ld3, ldi3: begin
    #10 Grb = 1; BAout = 1; Yin = 1;
    #10 Grb = 0; BAout = 0; Yin = 0;
end
ld4, ldi4 : begin
    #10 Zin = 1; Cout = 1; alu_opcode = alu_addop;
    #15 Zin = 0; Cout = 0; alu_opcode = alu_nop;
end
ld5 : begin
    #10 ZLowOut = 1; MARin = 1;
    #15 ZLowOut = 0; MARin = 0;
end
ldi5 : begin
    #10 ZLowOut = 1; Rin = 1; Gra = 1;
    #15 ZLowOut = 0; Rin = 0; Gra = 0;
end
ld6 : begin
    #10 read = 1; MDRin = 1;
    #15 read = 0; MDRin = 0;
end
ld7 : begin
    #10 MDRout = 1; Rin = 1; Gra = 1;
    #15 MDRout = 0; Rin = 0; Gra = 0;
end

//Store instruction states
st3 : begin
    #10 Grb = 1; BAout = 1; Yin = 1;
    #10 Grb = 0; BAout = 0; Yin = 0;
end
st4: begin
    #10 Zin = 1; Cout = 1; alu_opcode = alu_addop;
    #15 Zin = 0; Cout = 0; alu_opcode = alu_nop;
end
st5 : begin
    #10 ZLowOut = 1; MARin = 1;
    #15 ZLowOut = 0; MARin = 0;
end
st6 : begin
    #10 write = 1; MDRin = 1; Rout = 1; Gra = 1;
    #15 write = 0; MDRin = 0; Rout = 0; Gra = 0;
end
```

```verilog
            //Branch instruction states
            br3: begin
                #10 Gra = 1; Rout = 1;
                #15 Gra = 0; Rout = 0;
            end
            br4: begin
                #10 PCout = 1; Yin = 1;
                #15 PCout = 0; Yin = 0;
            end
            br5: begin
                #10 Cout = 1; Zin = 1; CONN_in = 1; alu_opcode = alu_nop;
                #15 Cout = 0; Zin = 0; CONN_in = 0;
            end
            br6: begin
                #10 ZLowOut = 1; PCin = 1;
                #15 ZLowOut = 0; PCin = 0;
            end

            //Jump register and Jump and Link Register instructions
            jr3 : begin
                #10 Gra = 1; Rout = 1; PCin = 1;
                #10 Gra = 0; Rout = 0; PCin = 0;
            end
            jal3 : begin
                #10 PCout = 1; jal_flag = 1;
                #10 PCout = 0; jal_flag = 0;
            end
            jal4: begin
                #10 PCin = 1; Gra = 1; Rout = 1;
                #10 PCin = 0; Gra = 0; Rout = 0;
            end

            //Input Output instruction states
            in3: begin
                #10 InPortIn = 1; Rin = 1; Gra = 1;
                #15 InPortIn = 0; Rin = 0; Gra = 0;
            end
            out3: begin
                #10 OutPortIn = 1; Rout = 1; Gra = 1;
                #15 OutPortIn = 0; Rout = 0; Gra = 0;
            end

            //Mfhi and Mflo instruction states
            mfhi3: begin
                #10 HIout = 1; Rin = 1; Gra = 1;
                #15 HIout = 0; Rin = 0; Gra = 0;
            end
            mflo3: begin
                #10 LOout = 1; Rin = 1; Gra = 1;
                #15 LOout = 0; Rin = 0; Gra = 0;
            end

            //No operation instruction states
            nop3: begin
                #10 alu_opcode = alu_nop;
            end

            //If halted, show that not running
            halt: begin
                run = 0;
                clear = 0;
            end
    endcase
end

endmodule
```

The Datapath

The Datapath module is the part of the processor that connects all components and is the highest-level entity. It consists of registers, ALU, bus, and multiplexers.

To integrate the control unit into the datapath module, we made some changes to the inputs and outputs of the datapath. The datapath now only takes inputs for clock, clear, and stop. These are signals that control the timing, resetting, and halting of the processor. It also takes in the InPortData, which is the data from the input port. The datapath outputs run, and OutPortData. The run signal indicates whether the processor is executing an instruction or not. The OutPortData is the data to be sent to the output port.

The control unit uses the Jal_flag and or's it with R15in so that when doing a jump and link instruction the R15 value stores the PC value as a return register. The Jal_flag is a signal that indicates whether the current instruction is a jump and link instruction or not. The R15 in is the input to register 15 from the multiplexer. The PC value is the value of the program counter register, which holds the address of the next instruction to be executed. By or'ing the Jal_flag with R15 in, we ensure that register 15 will store the PC value only when executing a jump and link instruction. This way, register 15 can be used as a return address for subroutine calls.

```verilog
module datapath (
    //Test Bench inputs/outputs but goes through datapath to control unit
    output run,
    input wire clk, reset, stop,
    //Inport data from external device
    input [31:0] InPortData
);

wire clr;

wire R0out, R1out, R2out, R3out, R4out, R5out, R6out, R7out, R8out, R9out, R10out, R11out,
R12out, R13out, R14out, R15out;
wire R0in, R1in, R2in, R3in, R4in, R5in, R6in, R7in, R8in, R9in, R10in, R11in, R12in, R13in,
R14in, R15in;
wire branch_flag;
//control unit wires
wire PCout, read, write, BAout, Rin, Rout, Gra, Grb, Grc, CONN_in, MARin, MDRin, HIin, LOin,
Yin, jal_flag, R15jal,
        Zin, PCin, IRin, incPC, InPortIn, OutPortIn, HIout, LOout, ZLowOut, ZHighOut, MDRout,
Cout, InPortOut;
wire [4:0] opcode;

wire [31:0] BusMuxIn_R0,
            BusMuxIn_R1,
            BusMuxIn_R2,
            BusMuxIn_R3,
            BusMuxIn_R4,
            BusMuxIn_R5,
            BusMuxIn_R6,
            BusMuxIn_R7,
            BusMuxIn_R8,
            BusMuxIn_R9,
            BusMuxIn_R10,
            BusMuxIn_R11,
            BusMuxIn_R12,
            BusMuxIn_R13,
            BusMuxIn_R14,
            BusMuxIn_R15;

wire [31:0] IRdata,
            BusMuxIn_HI,
            BusMuxIn_LO,
            BusMuxIn_Zhigh,
            BusMuxIn_Zlow,
            BusMuxIn_PC,
            BusMuxIn_MDR,
            BusMuxIn_InPort,
            RamDataOut,
            Yout,
            d_pc,
            MARout,
            BusMuxOut,
            C_sign_extended,
            OutPortOut;

wire [63:0] CRegOut;


//Control unit initialization
ctrl_unit cu (
    //Test Bench inputs/outputs but goes through datapath
    .run(run),
    .clear(clr),
    .clk(clk),
    .reset(reset),
    .stop(stop),
    //Datapath inputs/outputs
    .IRdata(IRdata),
    .read(read),
    .write(write),
    .BAout(BAout),
    .Rin(Rin),
    .Rout(Rout),
    .Gra(Gra),
    .Grb(Grb),
    .Grc(Grc),
    .CONN_in(CONN_in),
    .MARin(MARin),
    .MDRin(MDRin),
    .HIin(HIin),
    .LOin(LOin),
    .Yin(Yin),
    .Zin(Zin),
    .PCin(PCin),
    .IRin(IRin),
    .incPC(incPC),
    .InPortIn(InPortIn),
    .OutPortIn(OutPortIn),
    .HIout(HIout),
    .LOout(LOout),
    .ZLowOut(ZLowOut),
    .ZHighOut(ZHighOut),
    .MDRout(MDRout),
    .Cout(Cout),
    .InPortOut(InPortOut),
    .PCout(PCout),
    .alu_opcode(opcode),
    .jal_flag(jal_flag)
);

assign R15jal = (R15in | jal_flag);

regR0 R0 (BAout, clr, clk, R0in, BusMuxOut, BusMuxIn_R0); //input signal is always 0 for R0
(special reg)
reg32bit R2 (clr, clk, R2in, BusMuxOut, BusMuxIn_R2);
reg32bit R3 (clr, clk, R3in, BusMuxOut, BusMuxIn_R3);
reg32bit R1 (clr, clk, R1in, BusMuxOut, BusMuxIn_R1);
reg32bit R4 (clr, clk, R4in, BusMuxOut, BusMuxIn_R4);
reg32bit R5 (clr, clk, R5in, BusMuxOut, BusMuxIn_R5);
reg32bit R6 (clr, clk, R6in, BusMuxOut, BusMuxIn_R6);
reg32bit R7 (clr, clk, R7in, BusMuxOut, BusMuxIn_R7);
reg32bit R8 (clr, clk, R8in, BusMuxOut, BusMuxIn_R8);
reg32bit R9 (clr, clk, R9in, BusMuxOut, BusMuxIn_R9);
reg32bit R10 (clr, clk, R10in, BusMuxOut, BusMuxIn_R10);
reg32bit R11 (clr, clk, R11in, BusMuxOut, BusMuxIn_R11);
reg32bit R12 (clr, clk, R12in, BusMuxOut, BusMuxIn_R12);
reg32bit R13 (clr, clk, R13in, BusMuxOut, BusMuxIn_R13);
```

```
reg32bit R14 (clr, clk, R14in, BusMuxOut,  BusMuxIn_R14);
reg32bit R15 (clr, clk, R15jal, BusMuxOut,  BusMuxIn_R15);

reg32bit HI (clr, clk, HIin, BusMuxOut, BusMuxIn_HI);
reg32bit LO (clr, clk, LOin, BusMuxOut, BusMuxIn_LO);
reg32bit ZHigh (clr, clk, Zin, CRegOut[63:32], BusMuxIn_Zhigh);
reg32bit ZLow (clr, clk, Zin, CRegOut[31:0], BusMuxIn_Zlow);


//PC reg initialization, using specific incPC input to increment PC by 1 each time set to high
regPC PC (clr, clk, PCin, BusMuxOut, BusMuxIn_PC);

//Input and output ports added to datapath (p2)
reg32bit InPort (clr, clk, InPortIn, InPortData, BusMuxIn_InPort);
reg32bit OutPort (clr, clk, OutPortIn, BusMuxOut, OutPortOut);

//MDR reg initialization

MD_reg32 MDR (.clr(clr), .clk(clk), .read(read), .MDRin(MDRin), .BusMuxOut(BusMuxOut),
.Mdatain(RamDataOut), .Q(BusMuxIn_MDR)); //special MDR reg
reg32bit MAR (clr, clk, MARin, BusMuxOut, MARout);

// Goes into ALU A input
reg32bit Y (clr, clk, Yin, BusMuxOut, Yout);

//Instruction register. IRdata doesn't go on the bus, but leads to CON
reg32bit IR (clr, clk, IRin, BusMuxOut, IRdata);

//Memory initialization
//RamDataOut used as MDR input since, using BusMuxIn_MDR would have two registers writing to
the same input.
ram myRam (.clk(clk), .read(read), .write(write), .MARout(MARout[8:0]), .D(BusMuxIn_MDR),
.Q(RamDataOut));

//Control Branch logic
CONN_FF myConn_ff (
    .IRdata(IRdata),
    .BusMuxOut(BusMuxOut),
    .CONN_in(CONN_in),
    .CONN_out(branch_flag)
);


//Select and Encode logic for selecting register functions based on opcode
select_and_encode mySAE (
    .irOut(IRdata),
    .Gra(Gra),
    .Grb(Grb),
    .Grc(Grc),
    .Rin(Rin),
    .Rout(Rout),
    .BAout(BAout),
    .R0in(R0in),
    .R1in(R1in),
    .R2in(R2in),
    .R3in(R3in),
    .R4in(R4in),
    .R5in(R5in),
    .R6in(R6in),
    .R7in(R7in),
    .R8in(R8in),
    .R9in(R9in),
    .R10in(R10in),
    .R11in(R11in),
    .R12in(R12in),
    .R13in(R13in),
    .R14in(R14in),
    .R15in(R15in),
    .R0out(R0out),
    .R1out(R1out),
    .R2out(R2out),
    .R3out(R3out),
    .R4out(R4out),
    .R5out(R5out),
    .R6out(R6out),
    .R7out(R7out),
    .R8out(R8out),
    .R9out(R9out),
    .R10out(R10out),
    .R11out(R11out),
    .R12out(R12out),
    .R13out(R13out),
    .R14out(R14out),
    .R15out(R15out),
    .C_sign_extended(C_sign_extended)
);

//bus
bus myBus (
    //encoder
    .R0out(R0out),
    .R1out(R1out),
    .R2out(R2out),
    .R3out(R3out),
    .R4out(R4out),
    .R5out(R5out),
    .R6out(R6out),
    .R7out(R7out),
    .R8out(R8out),
    .R9out(R9out),
    .R10out(R10out),
    .R11out(R11out),
    .R12out(R12out),
    .R13out(R13out),
    .R14out(R14out),
    .R15out(R15out),
    .HIout(HIout),
    .LOout(LOout),
    .ZHighOut(ZHighOut),
    .ZLowOut(ZLowOut),
    .PCout(PCout),
    .MDRout(MDRout),
    .InPortOut(InPortOut),
    .Cout(Cout),
    //multiplexer
    .BusMuxIn_R0(BusMuxIn_R0),
    .BusMuxIn_R1(BusMuxIn_R1),
    .BusMuxIn_R2(BusMuxIn_R2),
    .BusMuxIn_R3(BusMuxIn_R3),
    .BusMuxIn_R4(BusMuxIn_R4),
    .BusMuxIn_R5(BusMuxIn_R5),
    .BusMuxIn_R6(BusMuxIn_R6),
    .BusMuxIn_R7(BusMuxIn_R7),
    .BusMuxIn_R8(BusMuxIn_R8),
    .BusMuxIn_R9(BusMuxIn_R9),
    .BusMuxIn_R10(BusMuxIn_R10),
    .BusMuxIn_R11(BusMuxIn_R11),
    .BusMuxIn_R12(BusMuxIn_R12),
    .BusMuxIn_R13(BusMuxIn_R13),
    .BusMuxIn_R14(BusMuxIn_R14),
    .BusMuxIn_R15(BusMuxIn_R15),
    .BusMuxIn_HI(BusMuxIn_HI),
    .BusMuxIn_LO(BusMuxIn_LO),
    .BusMuxIn_Zhigh(BusMuxIn_Zhigh),
    .BusMuxIn_Zlow(BusMuxIn_Zlow),
    .BusMuxIn_PC(BusMuxIn_PC),
    .BusMuxIn_MDR(BusMuxIn_MDR),
    .BusMuxIn_InPort(BusMuxIn_InPort),
    .C_sign_extended(C_sign_extended),
    .BusMuxOut(BusMuxOut)
);

//alu
alu_test myAlu(
    .clk(clk),
    .clr(clr),
    .incPC(incPC),
    .CONN_out(branch_flag),
    .B(BusMuxOut),
    .A(Yout),
    .opcode(opcode),
    .C(CRegOut)
    );
endmodule
```

Xor Module (Bonus)

A new module for Xor'ing two 32 bit values was wired into the ALU and given its own opcode. The module code is as follows:

```
module xor32 (a,b,c);
    input wire [31:0]a,b;
    output reg [31:0]c;
    integer i;

    always @(*)begin
        for (i = 0; i < 31; i = i + 1) begin
            c[i] = ((a[i])^(b[i]));
        end
    end

endmodule
```

Updated Ram

The ram was updated to read in the ram.hex file. The `ifdef and `endif code was the new added lines that read the file in when initializing RAM.

```verilog
//512 x 32 RAM
module ram(
    input clk, read, write,
    input [8:0] MARout,     //address
    input [31:0] D,
    output [31:0] Q
);
    reg [31:0] mem [511:0];

    `ifdef MODEL_TECH
    initial $readmemh("../../ram.hex", mem);
    `else
    initial $readmemh("ram.hex", mem);
    `endif

    assign Q = (write || !read) ?  32'bZZZZZZZZ : mem[MARout];

   always @(posedge clk) begin
        if(write) mem[MARout] = D;
    end
endmodule
```

Testbench

This is a test bench for a Verilog module called tb_minisrc. It simulates the behavior of the datapath module by providing inputs and observing outputs. The test bench sets up an initial state for the simulation by initializing the inputs clock, reset, stop, and InPortData. The clock signal is toggled every 2 time units to drive the simulation. The reset signal is asserted for 2 time units at the beginning of the simulation to reset the datapath module.

The code for the testbench is as follows:

```
// Running the simulation outlined in CPU_Phase3

`timescale 1ns/1ps

module tb_minisrc;
    //Inputs
    reg clock;
    reg [31:0] InPortData;
    //Outputs
    reg reset, stop;
    //wire clear;
    wire run;

    datapath dp (
    .run(run),
    .clk(clock),
    .reset(reset),
    .stop(stop),
    //.clr(clear),
    .InPortData(InPortData)
);

    initial begin
        #2 clock = 0; reset = 1; stop = 0; InPortData = 0;
        #2 reset = 0;
        #2 stop = 0;

    end
    always
        #2 clock = ~clock;
endmodule
```

The instruction that were pre-loaded into the memory are as follows:

```
0: ldi R1, 2
1: ldi R0, 0(R1)
2: ld R2, 104
3: ldi R2, -4(R2)
4: ld R1, 1(R2)
5: ldi R3, 105
6: brmi R3, 4
7: ldi R3, 2(R3)
8: ld R7, -3(R3)
9: nop
10: brpl R7, 2
11: ldi R2, 5
12: ldi R3, 2(R1)
13: add R3, R2, R3
14: addi R7, R7, 2
15: neg R7, R7
16: not R7, R7
17: andi R7, R7, 15
18: ror R1, R1, R0
19: ori R7, R1, 28
20: shra R7, R7, R0
21: shr R2, R3, R0
22: st 82, R2
23: rol R2, R2, R0
24: or R2, R3, R0
25: and R1, R2, R1
26: st 96(R1), R3
27: sub R3, R2, R3
28: shl R1, R2, R0
29: ldi R4, 6
30: ldi R5, 50
31: mul R5, R4
32: mfhi R7
33: mflo R6
34: div R5, R4
35: ldi R8, -1(R4)
36: ldi R9, -19(R5)
37: ldi R10, 0(R6)
38: ldi R11, 0(R7)
39: jal R10
40: xor R1, R2, R3
41: halt
82: 00000026 (immediate value in hex)
```

104: 00000055 (immediate value in hex)
240: 0000FFFF (immediate value in hex)
300: add R13, R8, R10
301: sub R12, R9, R11
302: sub R13, R13, R12
303: jr R15

Note that the value on the left hand side is the decimal representation of the address in memory where the value or instruction was loaded into. Some of the values were immediate values that were to be loaded into register. These were located at 82, 104, and 240 in memory.

The testbench had the following output in the wave form and memory viewer.

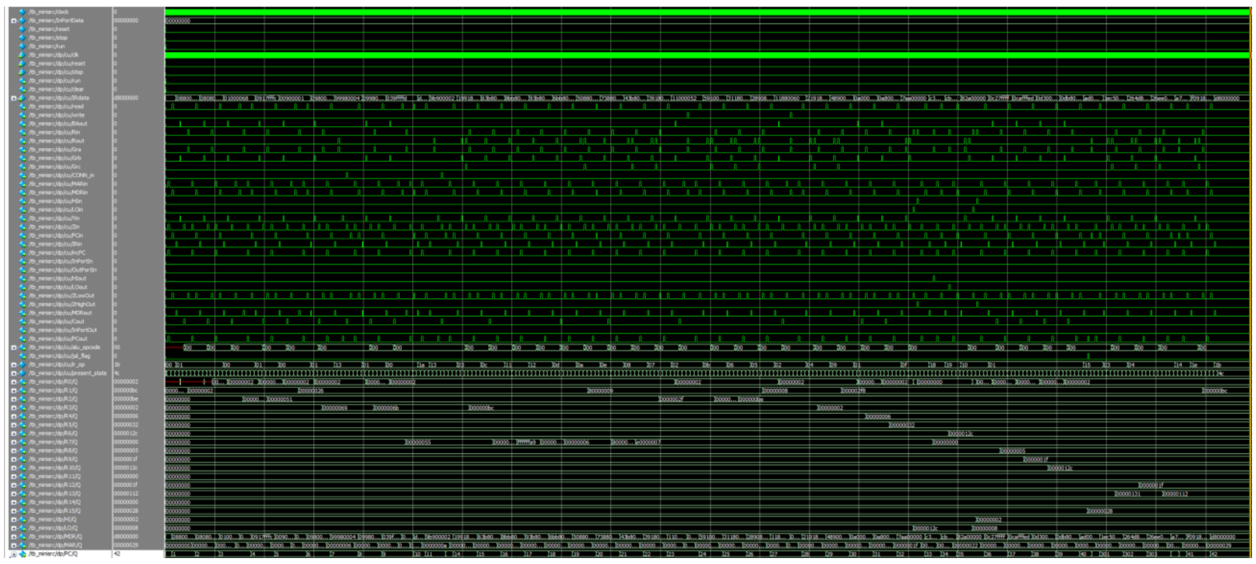


Figure 1: Entire waveform of all instructions.

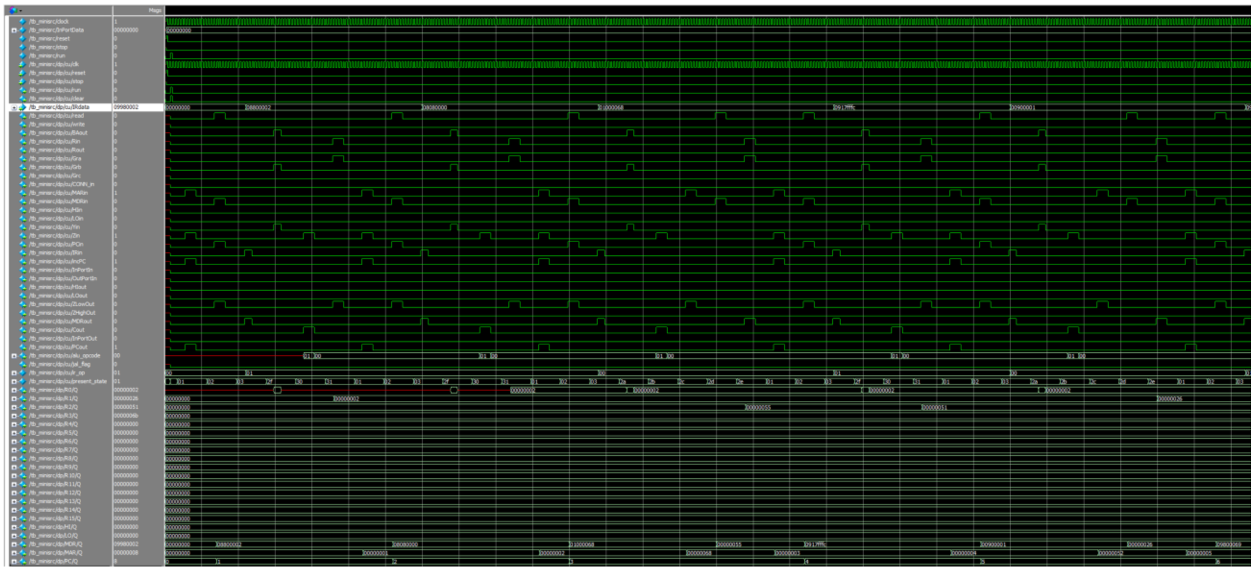


Figure 2: Instructions 1 to 5

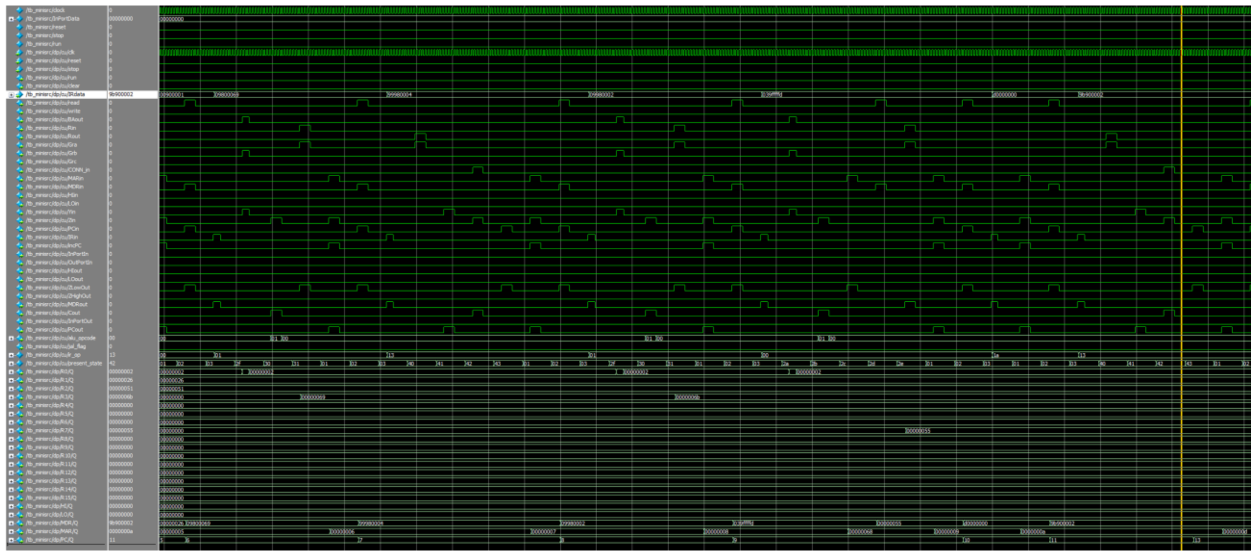


Figure 3: Instructions 6 to 11

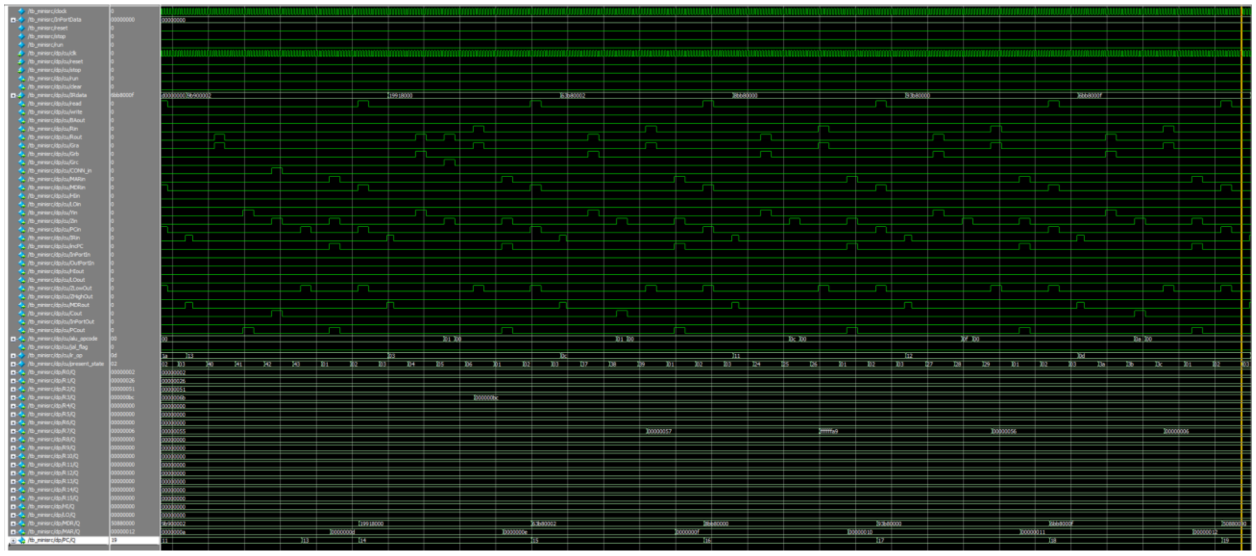


Figure 4: Instructions 13 -18 (note 12 is skipped due to branch)

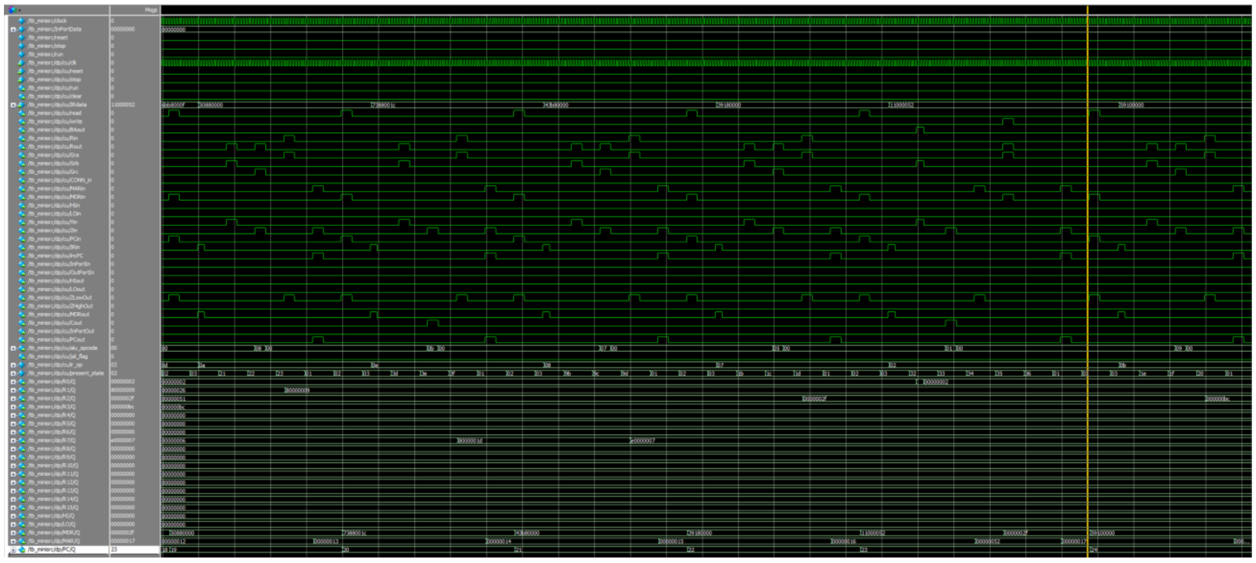


Figure 5: Instructions 19 to 23

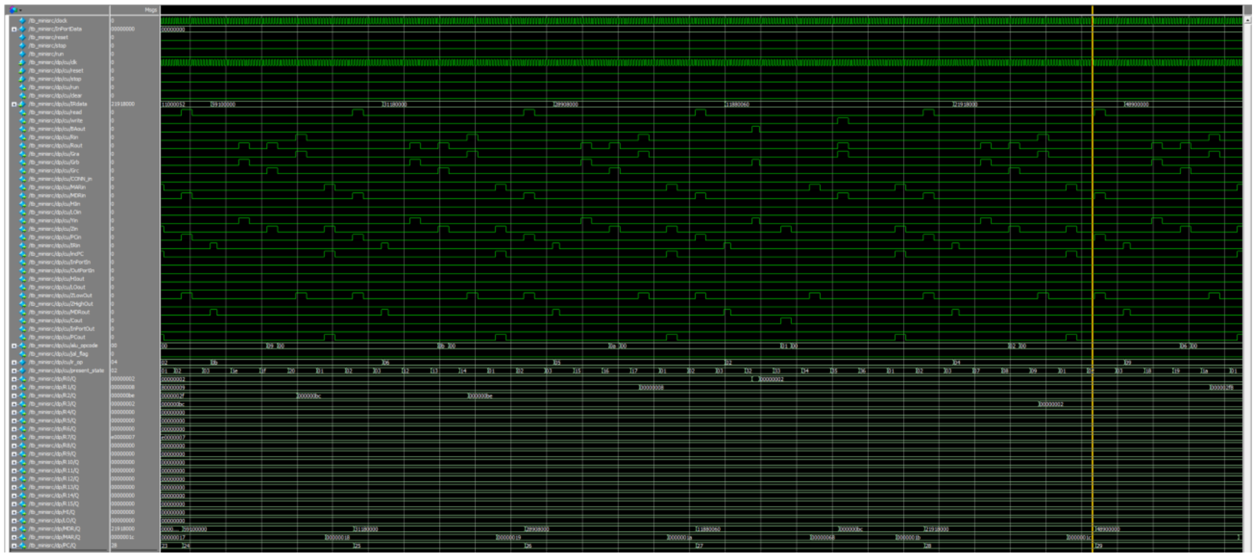


Figure 6: Instruction 24 to 28

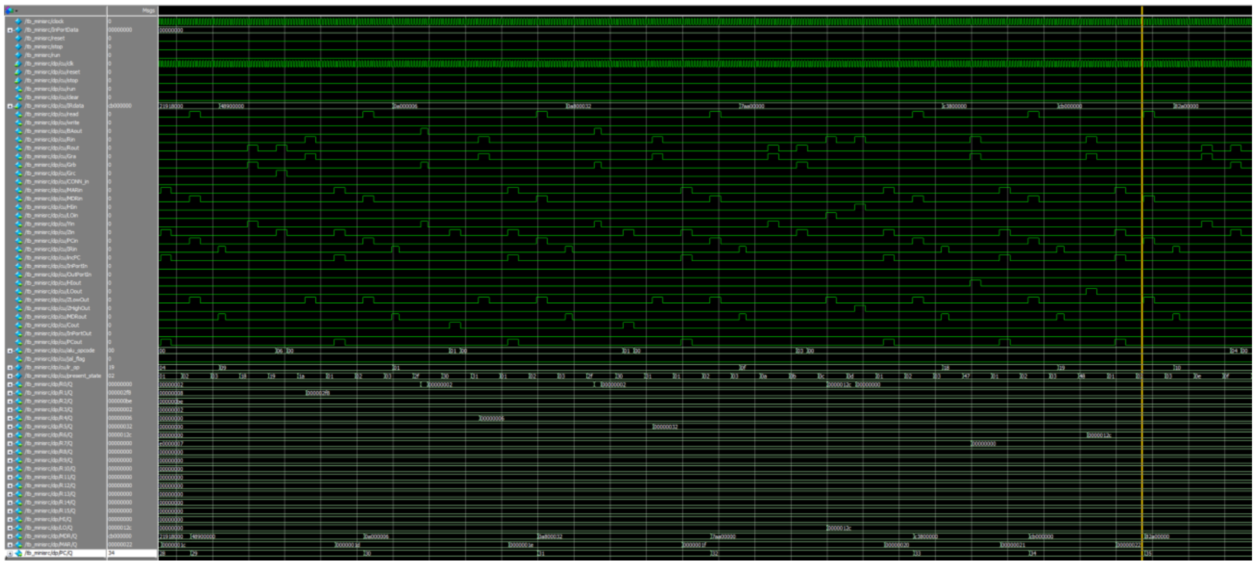


Figure 7: Instruction 29 to 34

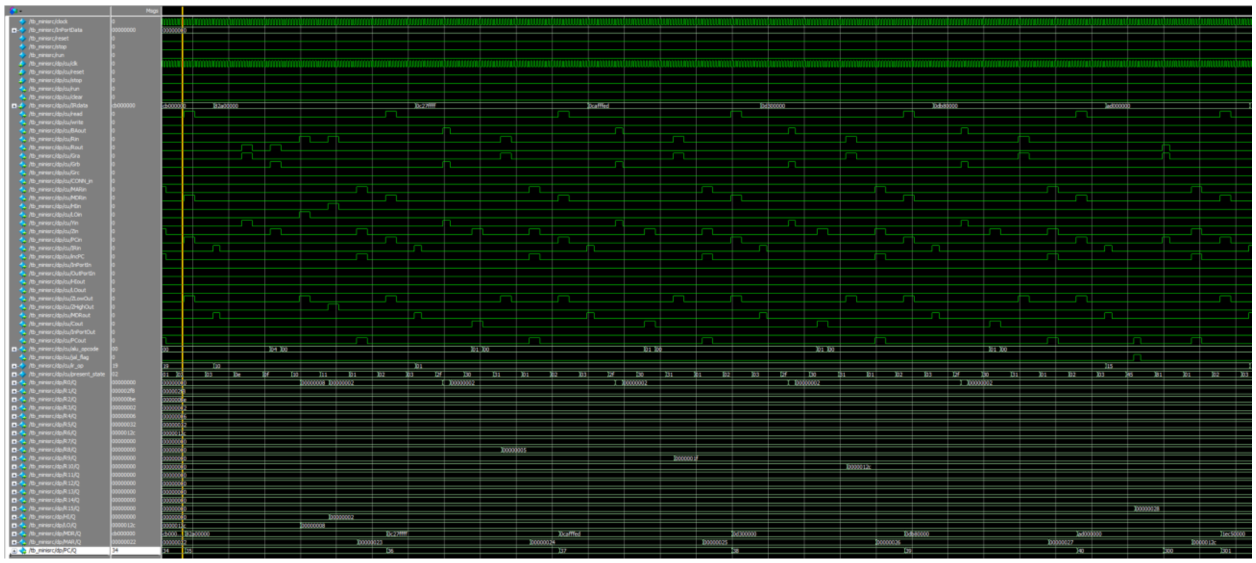


Figure 8: Instruction 35 to 40

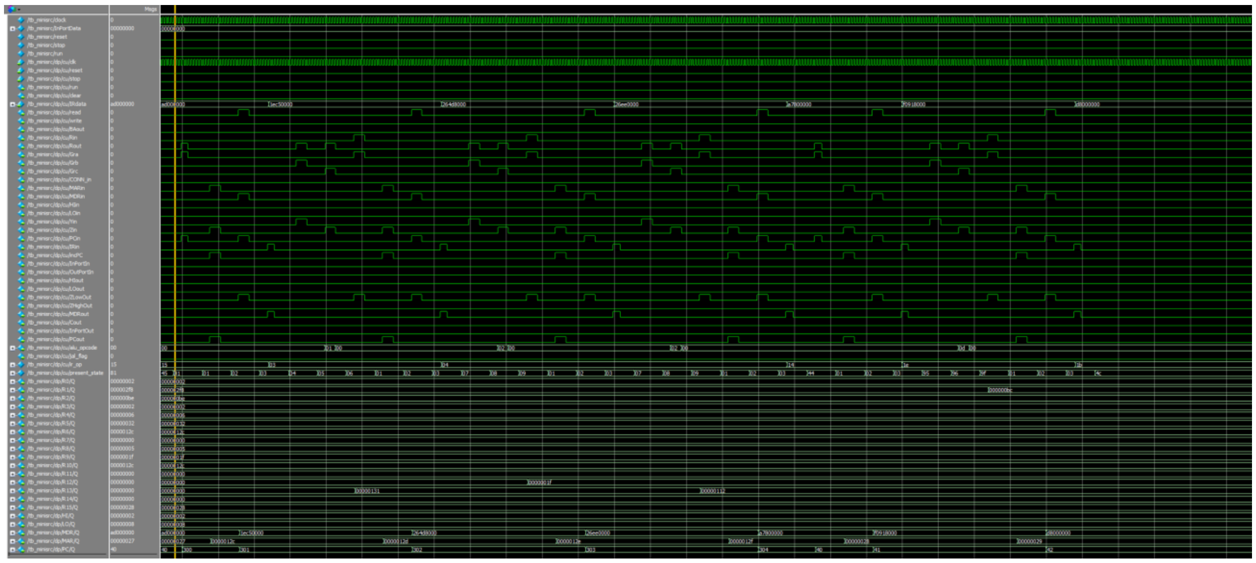
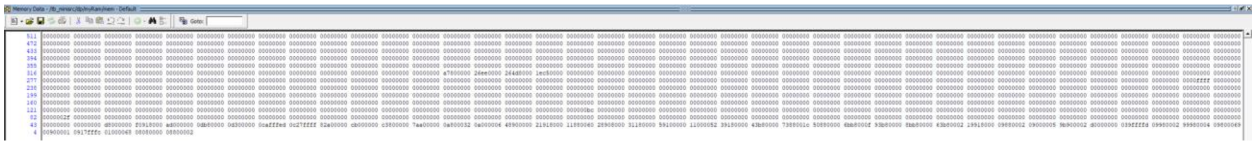


Figure 9: Instruction 300 to 304 then back to 40 - 42 (halt)



Memory Data - /tb_minisrc/dp/myRam/mem - Default

```
511 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
472 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
433 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
394 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
355 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
316 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 a7800000 26ee0000 264d8000 1ec50000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
277 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 0000ffff 00000000
238 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
199 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
160 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
121 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 000000bc 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
 82 0000002f 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
 43 00000000 00000000 d8000000 f0918000 ad000000 0db80000 0d300000 0cafffed 0c27ffff 82a00000 cb000000 c3800000 7aa00000 0a800032 0a000006 48900000 21918000 11880060 28908000 31180000 59100000 11000052 39180000 43b80000 7388001c 50880000 6bb8000f 93b80000 8bb80000 63b80002 19918000 09880002 09000005 9b900002 d0000000 039ffffd 09980002 99980004 09800069
  4 00900001 0917fffc 01000068 08080000 08800002
```

Figure 10: Contents of Memory after the program ran.