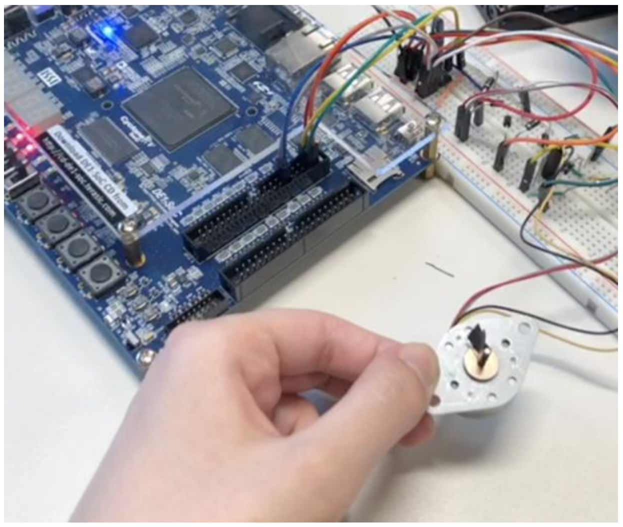
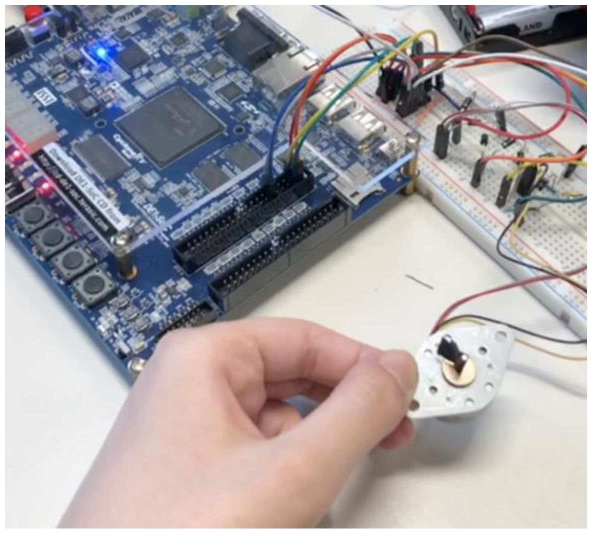
# MECHTRON 3TB4 Lab5 Report

Noor Al-Rajab | 400291137

Kavya Sundaresan | 400307169

In this report, the process of describing a specialized processor for moving a motor using Verilog and an FPGA microcontroller will be discussed. That process consisted of the following steps: writing smaller modules to construct the datapath, writing an FSM module to communicate with the datapath and writing assembly code which the program can interpret and run on the FPGA.

First, each lab partner wrote several modules to represent circuit components, such as ALU, Register File, Program Counter etc. Before integrating them, each component was simulated and given specific inputs to see if the obtained output is as expected. Next, components were integrated in a higher-level module "datapath." After writing some instructions in the instruction ROM, datapath was also simulated to ensure all modules are interacting correctly to produce the expected output. The FSM module was then written and simulated before integration with datapath. Finally, the datapath module and FSM module are instantiated in the module "Lab5." This module was then fully capable of running an instruction ROM file. The lowest computer programming language is Assembly, but only machine code (binary) can be used in the instruction ROM. We thought we had to convert it to machine code manually, so we created a Python script that does it for us, which can be found in the appendix. We later on realized that a compiler was already given, but creating the Python code was good practice. We then wrote special assembly code that controls the motor, and we used our Python script to compile it, and we put it into the ROM. Since the LEDRs were programmed to give the same outputs as the pins, the LEDRs can be used to check what the pins are outputting before having to wire up the motor. Once the LEDRs showed the expected behavior corresponding to motor windings, the circuit was built. The board pins don't supply enough electricity to drive the stepper motor, hence a motor driver interface was built. This circuit works in the following way: if a high input is supplied from the board, it uses a high input from a second power source, in this case 4 batteries in series supplying 6 Volts total. This is used for all 4 motor inputs using a 4-circuit chip. When we connected the motor to the circuit and the circuit to the board and ran the Verilog program, the motor started moving in steps.

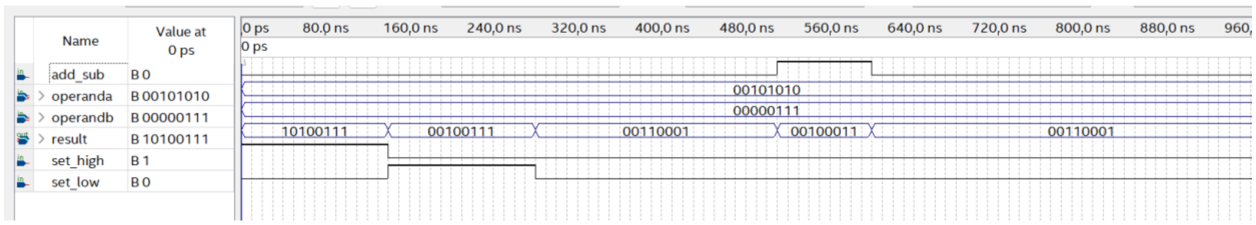



# Appendix

## Modules

ALU

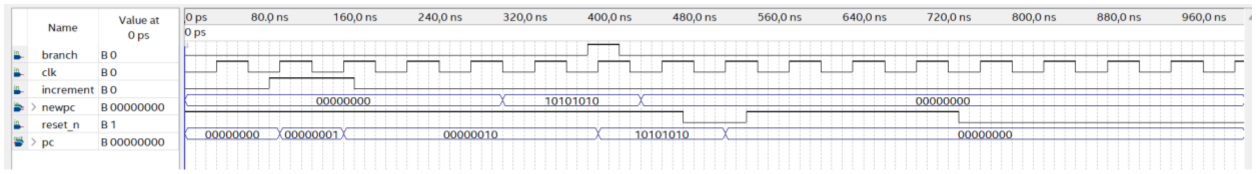
```
// this module is similar to alu but does not use Quartus library modules
module alu (input add_sub, set_low, set_high, input [7:0] operanda , operandb, output reg [7:0] result);


always @(*) begin
   if (set_low == 1) begin
      result[7:4] = operanda[7:4];
      result[3:0] = operandb[3:0];
   end else if (set_high == 1) begin
      result[3:0] = operanda[3:0];
      result[7:4] = operandb[3:0];
   end else if (add_sub ==0) begin
      result = operanda[7:0]+operandb[7:0]; //add
   end else if(add_sub == 1) begin
      result = operanda[7:0]-operandb[7:0]; //subtract
   end

end

endmodule
```



PC

```
module pc (input clk, reset_n, branch, increment, input [7:0] newpc,
         output reg [7:0] pc);
parameter RESET_LOCATION = 8'h00;
```

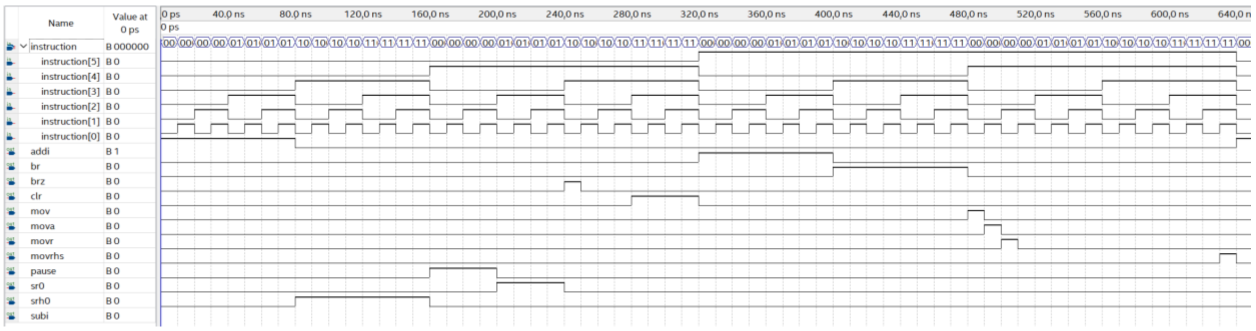
```
initial pc=8'b0; //initialization

always@(posedge clk) begin
   // the order of if statements makes sure it doesn't increment when it's loading a new pc
   if(~reset_n) pc<=RESET_LOCATION;
   else if(branch) pc[7:0]<=newpc[7:0];
   else if(increment) pc[7:0]<=pc[7:0]+8'b1;
end

endmodule

//WORKS
```



## Decoder

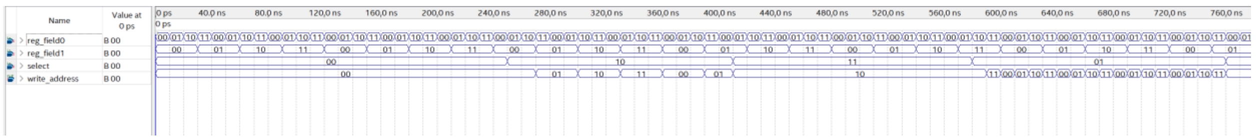
```
module decoder (input [5:0] instruction, //changed from 7:2 to 5:0 cuz first one is just no
                output wire br, brz, addi, subi, sr0, srh0, clr, mov, mova, movr, movrhs, pause
);

assign br=instruction[5]&!instruction[4]&!instruction[3];
assign clr=!instruction[5]&instruction[4]&instruction[3]&!instruction[2]&!instruction[1]&!instruction[0];
assign brz=instruction[5]&!instruction[4]&instruction[3];
assign addi=!instruction[5]&!instruction[4]&!instruction[3];
assign subi=!instruction[5]&!instruction[4]&instruction[3];
assign sr0=!instruction[5]&instruction[4]&!instruction[3]&!instruction[2];
assign srh0=!instruction[5]&instruction[4]&!instruction[3]&instruction[2];
assign mov=!instruction[5]&instruction[4]&instruction[3]&instruction[2];
assign mova=instruction[5]&instruction[4]&!instruction[3]&!instruction[2]&!instruction[1]&!instruction[0];
assign movr=instruction[5]&instruction[4]&!instruction[3]&!instruction[2]&!instruction[1]&instruction[0];
assign movrhs=instruction[5]&instruction[4]&!instruction[3]&!instruction[2]&instruction[1]&!instruction[0];
assign pause=instruction[5]&instruction[4]&instruction[3]&instruction[2]&instruction[1]&instruction[0];
```

```
endmodule

//WORKS
```



<u>Write address select</u>

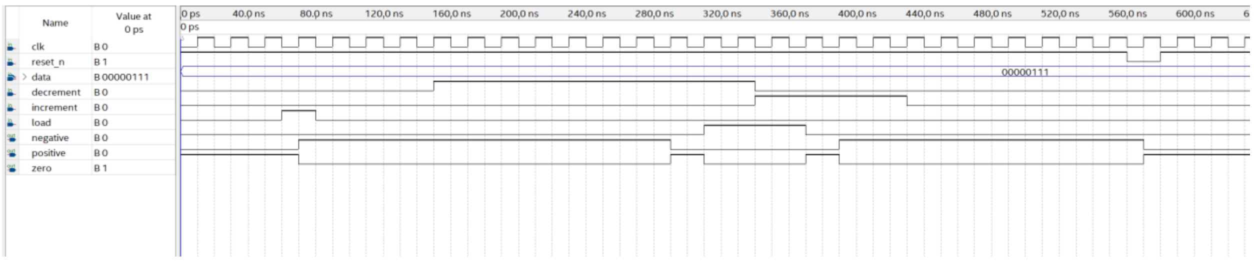
```
module write_address_select (input [1:0] select, input [1:0] reg_field0, reg_field1, output reg [1:0] write_address);

always@(*)
begin
  case(select)
    2'b00: write_address = 2'b00;
    2'b01: write_address = reg_field0;
    2'b10: write_address = reg_field1;
    2'b11: write_address = 2'b10;
  endcase;
end

endmodule

//WORKS
```



<u>Temp register</u>

```
module temp_register (input clk, reset_n, load, increment, decrement, input [7:0] data,
          output negative, positive, zero);
reg signed[7:0]half_steps;
```

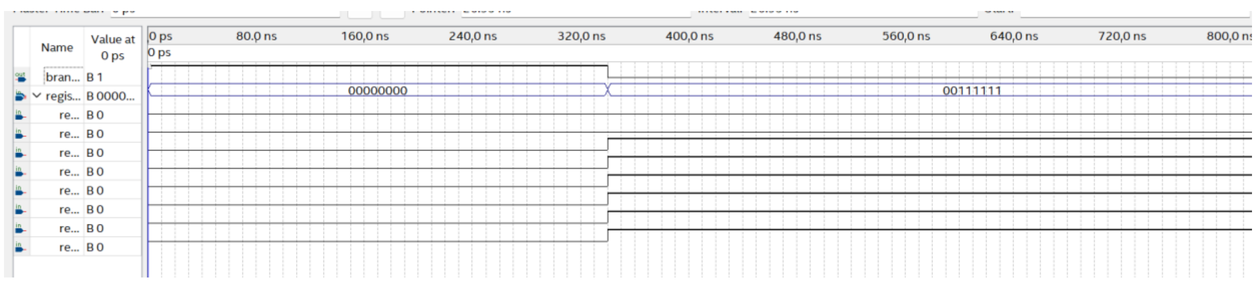
```
assign zero =
~(half_steps[7]|half_steps[6]|half_steps[5]|half_steps[4]|half_steps[3]|half_steps[2]|half_steps[1]|half_steps[0]); //it's
zero if it's zero
assign positive= half_steps>0?1'b1:1'b0;
assign negative= half_steps<0?1'b1:1'b0;
always@(posedge clk) begin
  if(~reset_n) begin
    half_steps<=8'b0;
  end else if(load) begin
    half_steps[7:0]<=data[7:0];
  end else if(increment) begin
    half_steps[7:0]<=half_steps[7:0]+8'b1;
  end else if(decrement) begin
    half_steps[7:0]<=half_steps[7:0]-8'b1;
  end

end
endmodule
```



Branch Logic

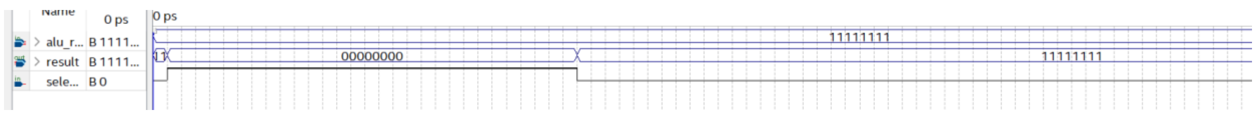
```
module branch_logic (input [7:0] register0, output reg branch);

always @(*) begin
  case(register0)
  8'b00000000: branch = 1'b1;
  default: branch = 1'b0;
  endcase
end
```

```
endmodule

//WORKS
```



Result Mux

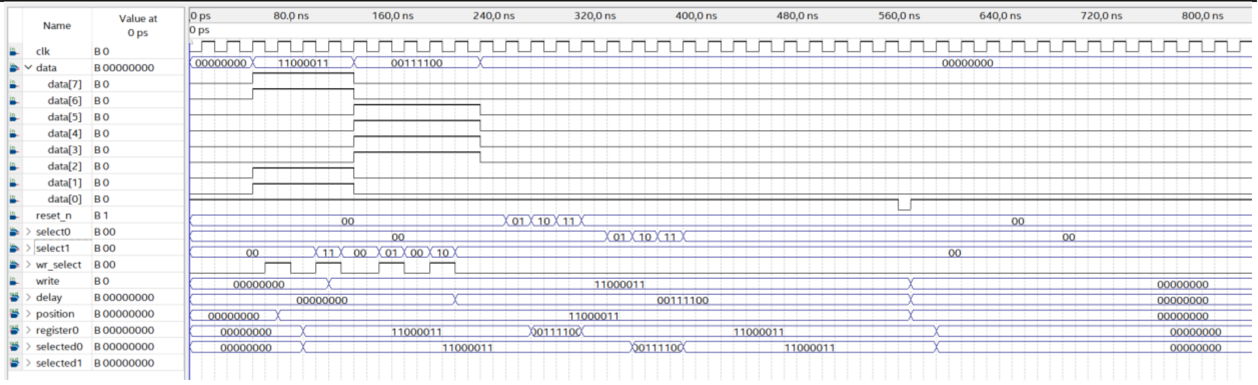
```
module result_mux (
    input select_result,
    input [7:0] alu_result,
    output reg [7:0] result
);

always@(*)
begin
    case(select_result)
        1'b0: result[7:0] = alu_result[7:0];
        1'b1: result[7:0] =  8'b00000000;
    endcase;
end

endmodule

//WORKS
```



Reg File

```
// This module implements the register file
```

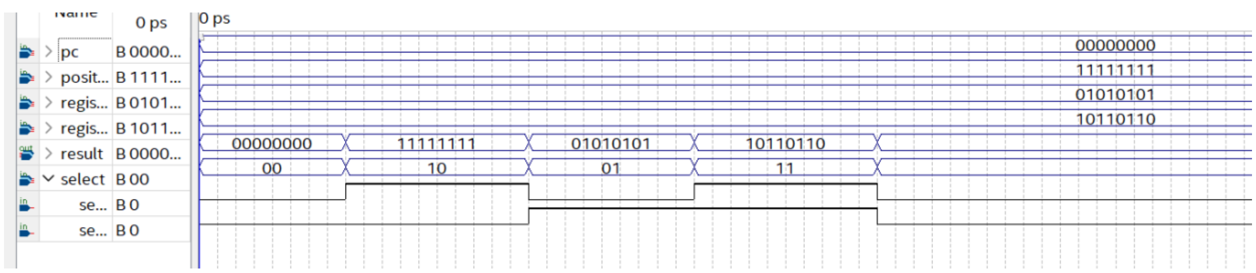
```verilog
module regfile (input clk, reset_n, write, input [7:0] data, input [1:0] select0, select1, wr_select,
                output reg [7:0] selected0, selected1, output [7:0] delay, position, register0);

// The comment /* synthesis preserve */ after the declaration of a register
// prevents Quartus from optimizing it, so that it can be observed in simulation
// It is important that the comment appear before the semicolon
reg [7:0] reg0 /* synthesis preserve */;
reg [7:0] reg1 /* synthesis preserve */;
reg [7:0] reg2 /* synthesis preserve */;
reg [7:0] reg3 /* synthesis preserve */;

assign register0 = reg0;
assign position = reg2;
assign delay = reg3;

always@(posedge clk) begin
   if (~reset_n) begin
      reg0 <= 8'b00000000;
      reg1 <= 8'b00000000;
      reg2 <= 8'b00000000;
      reg3 <= 8'b00000000;
   end else if (write) begin
      case(wr_select)
         2'b00: reg0 <= data;
         2'b01: reg1 <= data;
         2'b10: reg2 <= data;
         2'b11: reg3 <= data;
      endcase
   end

   case(select0)
      2'b00: selected0 <= reg0;
      2'b01: selected0 <= reg1;
      2'b10: selected0 <= reg2;
      2'b11: selected0 <= reg3;
   endcase
   case(select1)
      2'b00: selected1 <= reg0;
      2'b01: selected1 <= reg1;
      2'b10: selected1 <= reg2;
      2'b11: selected1 <= reg3;
   endcase
end

endmodule
```



| Name | Value at 0 ps |
|---|---|
| clk | B 0 |
| data | B 00000000 |
| data[7] | B 0 |
| data[6] | B 0 |
| data[5] | B 0 |
| data[4] | B 0 |
| data[3] | B 0 |
| data[2] | B 0 |
| data[1] | B 0 |
| data[0] | B 0 |
| reset_n | B 1 |
| select0 | B 00 |
| select1 | B 00 |
| wr_select | B 00 |
| write | B 0 |
| delay | B 00000000 |
| position | B 00000000 |
| register0 | B 00000000 |
| selected0 | B 00000000 |
| selected1 | B 00000000 |

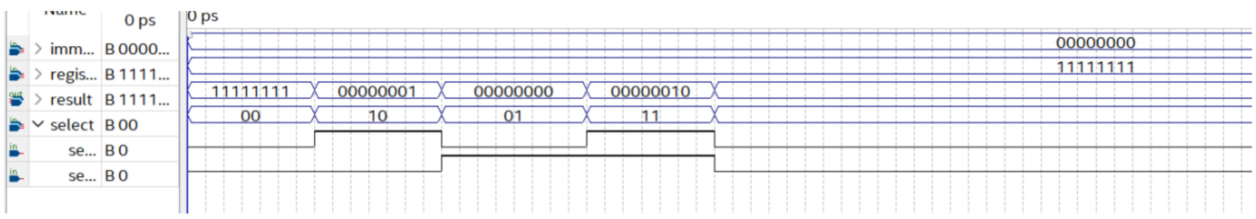
## Op1 Mux

```
module op1_mux (input [1:0] select, input [7:0] pc, register, register0, position,
            output reg [7:0] result);

always@(*)
begin
   case(select)
      2'b00: result = pc;
      2'b01: result = register;
      2'b10: result = position;
      2'b11: result = register0;
   endcase;
end
endmodule

//WORKS
```



## Op2 Mux

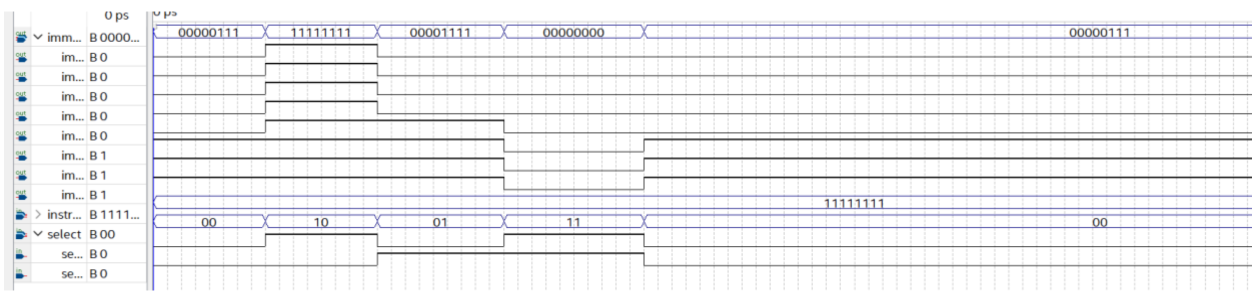
```
module op2_mux (input [1:0] select, input [7:0] register, immediate,
                output reg [7:0] result);

always@(*)
begin
   case(select)
      2'b00: result = register;
      2'b01: result = immediate;
      2'b10: result = 8'b00000001;
      2'b11: result = 8'b00000010;
   endcase;
end

endmodule

//WORKS
```



## Immediate Extractor

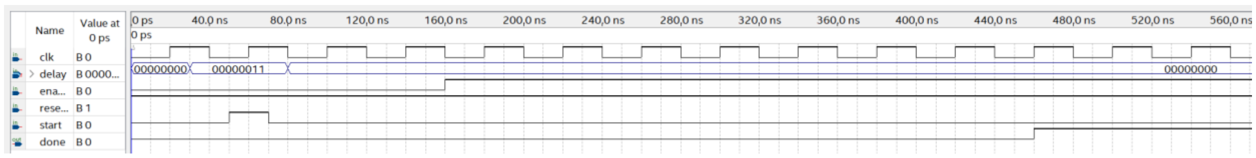
```
module immediate_extractor (input [7:0] instruction, input [1:0] select, output reg [7:0] immediate);

always@(*)
begin
   if (select == 2'b00) begin
      immediate[2:0] = instruction[4:2];
```

```
        immediate[7:3] = 4'b00000;
    end else if (select == 2'b01) begin
        immediate[3:0] = instruction[3:0];
        immediate[7:4] = 4'b0000;
    end else if (select == 2'b10) begin
        immediate[4:0] = instruction[4:0];
        immediate[5] = instruction[4];
        immediate[6] = instruction[4];
        immediate[7] = instruction[4];
    end else if (select == 2'b11) begin
        immediate = 8'b00000000; //MOV
    end
end

endmodule

//WORKS
```



## Delay Counter

```
module delay_counter (input clk, reset_n, start, enable, input [7:0] delay, output reg done);
//parameter BASIC_PERIOD=20'd500000;   // can change this value to make delay longer
//parameter BASIC_PERIOD=20'd1;
parameter BASIC_PERIOD=20'd100000;

reg [7:0] decrement;
reg [19:0] count;

always@(posedge clk) begin
    if(~reset_n) begin
        done <= 0;
        count <= 20'b0;
        decrement <= 8'b00000000;
    end else if (start) begin
        done <= 0;
        count <= 20'b0;
        decrement <= delay;

    end else if (enable) begin
        if (count < BASIC_PERIOD) begin
            count <= count + 20'b1;
        end else begin
            decrement <= decrement - 8'b00000001;
            if (decrement == 8'b00000000) begin
                done <= 1;
            end
            count <= 20'b0;
        end
    end
end

endmodule
```



<u>Assembly Code to add 5 + 69</u>

This assembly code was used to add 5 to 69 and store the result in register 0.

```
CLR R0
SR0 5
SRH0 4
MOV R1, R0
ADDI R1, 5
MOV R0, R1
```

Binary

01100000

01000101

01010100

01110100

00010101

01110001

## FSM Code

```
module control_fsm (
    input clk, reset_n,
    // Status inputs
    input br, brz, addi, subi, sr0, srh0, clr, mov, mova, movr, movrhs, pause,
    input delay_done,
    input temp_is_positive, temp_is_negative, temp_is_zero,
    input register0_is_zero,
    // Control signal outputs
    output reg write_reg_file,
    output reg result_mux_select,
    output reg [1:0] op1_mux_select, op2_mux_select,
    output reg start_delay_counter, enable_delay_counter,
    output reg commit_branch, increment_pc,
    output reg alu_add_sub, alu_set_low, alu_set_high,
    output reg load_temp_register, increment_temp_register, decrement_temp_register,
    output reg [1:0] select_immediate,
    output reg [1:0] select_write_address,
    output wire [4:0] currentState //for debugging

);
parameter RESET=5'b00000, FETCH=5'b00001, DECODE=5'b00010,
            BR=5'b00011, BRZ=5'b00100, ADDI=5'b00101, SUBI=5'b00110, SR0=5'b00111,
            SRH0=5'b01000, CLR=5'b01001, MOV=5'b01010, MOVA=5'b01011,
            MOVR=5'b01100, MOVRHS=5'b01101, PAUSE=5'b01110 , MOVR_STAGE2=5'b01111, // fsm only works
when pause is 01111 instead of 01110
            MOVR_DELAY=5'b10000, MOVRHS_STAGE2=5'b10001, MOVRHS_DELAY=5'b10010,
         PAUSE_DELAY=5'b10011;

reg [4:0] state=RESET;
reg [4:0] next_state_logic=FETCH; // NOT REALLY A REGISTER!!!
reg [4:0] pc_tracker = 5'b0;

// for debugging. uncomment only ONE of the next two lines
assign currentState[4:0]=state[4:0]; // for debugging
//assign currentState[4:0]=pc_tracker[4:0]; // for debugging

// Next state logic
always@(posedge clk or negedge reset_n) begin
   if(~reset_n) state<=RESET;
   else state<=next_state_logic;
end

// State register
always@(state) begin
   // to avoid inferred latches
   write_reg_file = 1'b0;
   result_mux_select = 1'b0;
   op1_mux_select[1:0] = 2'b0;
   op2_mux_select[1:0] = 2'b0;
   start_delay_counter = 1'b0;
   enable_delay_counter = 1'b0;
   commit_branch = 1'b0;
   alu_add_sub = 1'b0;
   alu_set_low = 1'b0;
   alu_set_high = 1'b0;
   load_temp_register = 1'b0;
   increment_temp_register = 1'b0;
   decrement_temp_register = 1'b0;
   select_immediate[1:0] = 2'b0;
   select_write_address[1:0] = 2'b0;
   increment_pc=1'b0;

   case(state)
```

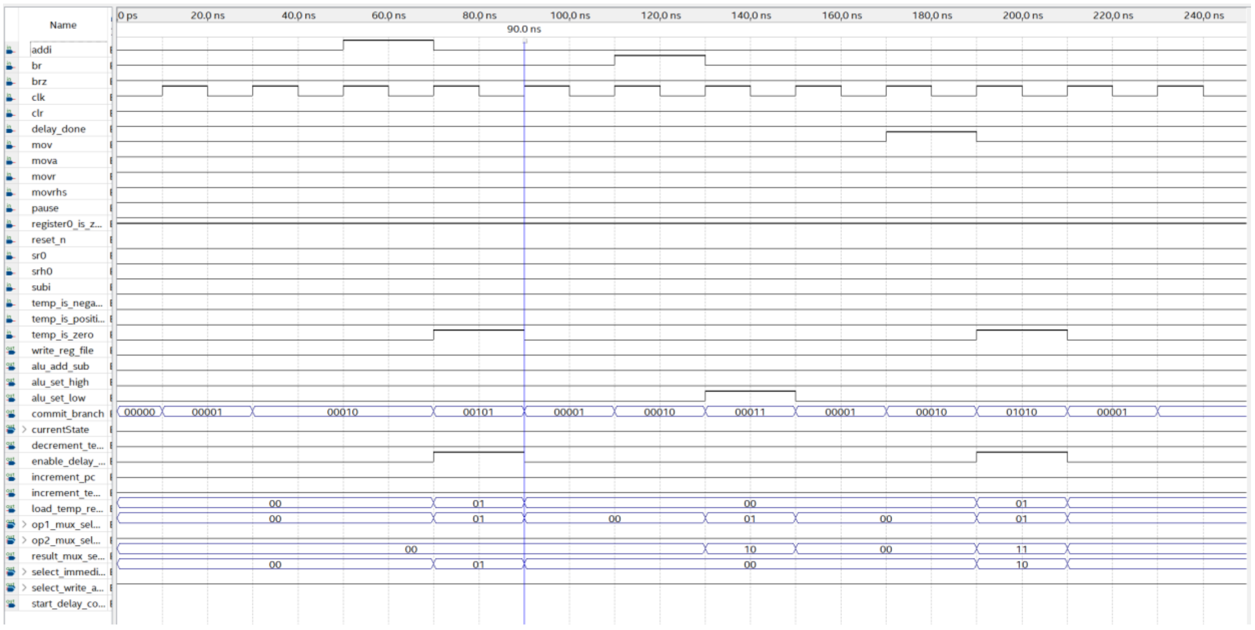
```
RESET: begin
  // Noor: although diagram says we should reset pc and registers at RESET
  // it's impossible to do it through code but it'll be done automatically
  // when user presses key0 to reset because of lab5 module.
    next_state_logic = FETCH;
  end
FETCH: begin
    //load memory[pc] to instruction (alr done in datapath)
    alu_set_low=1'b0;
    alu_set_high=1'b0;
    next_state_logic=DECODE;
  end
DECODE: begin
    if(addi) next_state_logic=ADDI;
    else if(subi) next_state_logic=SUBI;
    else if(mov) next_state_logic=MOV;
    else if(sr0) next_state_logic=SR0;
    else if(srh0) next_state_logic=SRH0;
    else if(clr) next_state_logic=CLR;
    else if(br) next_state_logic=BR;
    else if(brz) next_state_logic=BRZ;
    else if(movr) next_state_logic=MOVR;
    else if(movrhs) next_state_logic=MOVRHS;
    else if(pause) next_state_logic=PAUSE;
  end
BR: begin
    // idk what "sext" is so i ignored it cuz they ignored it on the slides
    select_immediate=2'b10;
    op1_mux_select=2'b00;
    op2_mux_select=2'b01;
    alu_add_sub=1'b0;
    commit_branch=1'b1;
    next_state_logic = FETCH;
  end
BRZ: begin
    if(register0_is_zero) begin
      op1_mux_select = 2'b00;
```

```
            op2_mux_select = 2'b01;
            select_immediate = 2'b10;
            alu_add_sub = 1'b0;
            commit_branch = 1'b1;
            next_state_logic = FETCH;
        end else begin
            increment_pc=1'b1;
            next_state_logic = FETCH;
        end
    end
ADDI: begin
        select_immediate=2'b00; // select imm3
        //operand 1 mux
        op1_mux_select=2'b01; // select "register"
        //operand 2 mux
        op2_mux_select=2'b01; // select "immediate"
        alu_add_sub=1'b0; // add
        result_mux_select=1'b0; // select alu output
        select_write_address=2'b01; // specify write address (not sure)
        write_reg_file = 1'b1; // write to regfile register
        increment_pc=1'b1;
        pc_tracker=pc_tracker+5'b1;
        next_state_logic=FETCH;
    end
SUBI: begin
        select_immediate=2'b00; // select imm3
        //operand 1 mux
        op1_mux_select=2'b01; // select "register"
        //operand 2 mux
        op2_mux_select=2'b01; // select "immediate"
        alu_add_sub=1'b1; // subtract
        result_mux_select=1'b0; // select alu output
        select_write_address=2'b01; // specify write address (not sure)
        write_reg_file = 1'b1; // write to regfile register
        increment_pc=1'b1;
        pc_tracker=pc_tracker+5'b1;
        next_state_logic=FETCH;
    end
  SR0: begin
        select_immediate=2'b01; // select imm4
        //operand 1 mux
        op1_mux_select=2'b11; // select reg0
        //operand 2 mux
        op2_mux_select=2'b01; // select "immediate"
        alu_set_low=1'b1; // concat. op1[7:4] with op2[3:0]
        result_mux_select=1'b0; // select alu output
        select_write_address=2'b00; // specify write address 0
        write_reg_file = 1'b1; // write to regfile register
        increment_pc=1'b1;
        pc_tracker=pc_tracker+5'b1;
        next_state_logic=FETCH;
    end
  SRH0: begin
        select_immediate=2'b01; // select imm4
        //operand 1 mux
        op1_mux_select=2'b11; // select reg0
        //operand 2 mux
        op2_mux_select=2'b01; // select "immediate"
        alu_set_high=1'b1; // concat. op2[3:0] with op1[3:0]
        result_mux_select=1'b0; // select alu output
        select_write_address=2'b00; // specify write address 0
        write_reg_file = 1'b1; // write to regfile register
        increment_pc=1'b1;
        pc_tracker=pc_tracker+5'b1;
        next_state_logic=FETCH;
    end
  CLR: begin
        result_mux_select=1'b1; // select const 0, not a result
        select_write_address = 2'b01; // select reg specified in instruction
        write_reg_file = 1'b1; // enable writing
        increment_pc=1'b1;
        pc_tracker=pc_tracker+5'b1;
        next_state_logic = FETCH;
    end
    MOV: begin
        select_write_address=2'b10; // choose reg field 1 (R_d)
        // only way to get a value to write it to the regfile has to be through alu
        select_immediate=2'b11; // outputs zero
        op1_mux_select=2'b01; // selected1
        op2_mux_select=2'b01; // immediate
        alu_add_sub=1'b0; // add
        result_mux_select=1'b0;
        write_reg_file=1'b1; // write to regfile
        increment_pc=1'b1;
        pc_tracker=pc_tracker+5'b1;
        next_state_logic=FETCH;
      end
    MOVR: begin
        load_temp_register=1'b1;
        next_state_logic=MOVR_STAGE2;
      end
    MOVRHS: begin
        load_temp_register=1'b1;
        next_state_logic=MOVRHS_STAGE2;
      end
    PAUSE: begin
        start_delay_counter = 1'b1;
        next_state_logic=PAUSE_DELAY;
      end
    MOVR_STAGE2: begin
        load_temp_register=1'b0; // if you were loading, stop
        if(temp_is_zero) begin
          increment_pc=1'b1;
          pc_tracker=pc_tracker+5'b1;
          next_state_logic=FETCH;
        end
        else if(temp_is_positive) begin
          start_delay_counter = 1'b1;
          decrement_temp_register=1'b1;
          op1_mux_select=2'b10;
          op2_mux_select=2'b11;
```

```verilog
          alu_add_sub=1'b0;
          result_mux_select=1'b1;
          select_write_address=2'b11;
          write_reg_file=1'b1;
          next_state_logic=MOVR_DELAY;
        end
        else if(temp_is_negative) begin
          start_delay_counter = 1'b1;
          increment_temp_register=1'b1;
          op1_mux_select=2'b10;
          op2_mux_select=2'b11;
          alu_add_sub=1'b1;
          result_mux_select=1'b1;
          select_write_address=2'b11;
          write_reg_file=1'b1;
          next_state_logic=MOVR_DELAY;
        end
      end
    MOVR_DELAY: begin
        enable_delay_counter=1'b1;
        //the next two lines aren't on the slides but are in appendix A
        if(delay_done) next_state_logic=MOVR_STAGE2;
        else next_state_logic=MOVR_DELAY;
      end
    MOVRHS_STAGE2: begin
        if(temp_is_zero) begin
          increment_pc=1'b1;
          pc_tracker=pc_tracker+5'b1;
          next_state_logic=FETCH;
        end
        else if(temp_is_positive) begin
          start_delay_counter = 1'b1;
          decrement_temp_register=1'b1;
          op1_mux_select=2'b10;
          op2_mux_select=2'b10;
          alu_add_sub=1'b0;
          result_mux_select=1'b0;
```

```
            select_write_address=2'b11;
            write_reg_file=1'b1;
            next_state_logic=MOVRHS_DELAY;
        end
        else if(temp_is_negative) begin
            start_delay_counter = 1'b1;
            increment_temp_register=1'b1;
            op1_mux_select=2'b10;
            op2_mux_select=2'b10;
            alu_add_sub=1'b1;
            result_mux_select=1'b0;
            select_write_address=2'b11;
            write_reg_file=1'b1;
            next_state_logic=MOVRHS_DELAY;
        end
    end
MOVRHS_DELAY: begin
        enable_delay_counter=1'b1;
        if(delay_done) next_state_logic=MOVRHS_STAGE2;
        else next_state_logic=MOVRHS_DELAY;
    end
PAUSE_DELAY: begin
        enable_delay_counter = 1'b1;
        if(delay_done) begin
            increment_pc=1'b1;
            pc_tracker=pc_tracker+5'b1;
            next_state_logic=FETCH;
        end else next_state_logic=PAUSE_DELAY;
    end
default: begin
    next_state_logic=FETCH;
    write_reg_file = 1'b0;
    result_mux_select = 1'b0;
    op1_mux_select[1:0] = 2'b0;
    op2_mux_select[1:0] = 2'b0;
    start_delay_counter = 1'b0;
    enable_delay_counter = 1'b0;
```

```
            commit_branch = 1'b0;
            alu_add_sub = 1'b0;
            alu_set_low = 1'b0;
            alu_set_high = 1'b0;
            load_temp_register = 1'b0;
            increment_temp_register = 1'b0;
            decrement_temp_register = 1'b0;
            select_immediate[1:0] = 2'b0;
            select_write_address[1:0] = 2'b0;
            increment_pc=1'b0;
        end

    endcase
end

// Output logic
endmodule
```



## Datapath

```
module datapath (input clk, reset_n,
        // Control signals
        input write_reg_file, result_mux_select,
        input [1:0] op1_mux_select, op2_mux_select,
```

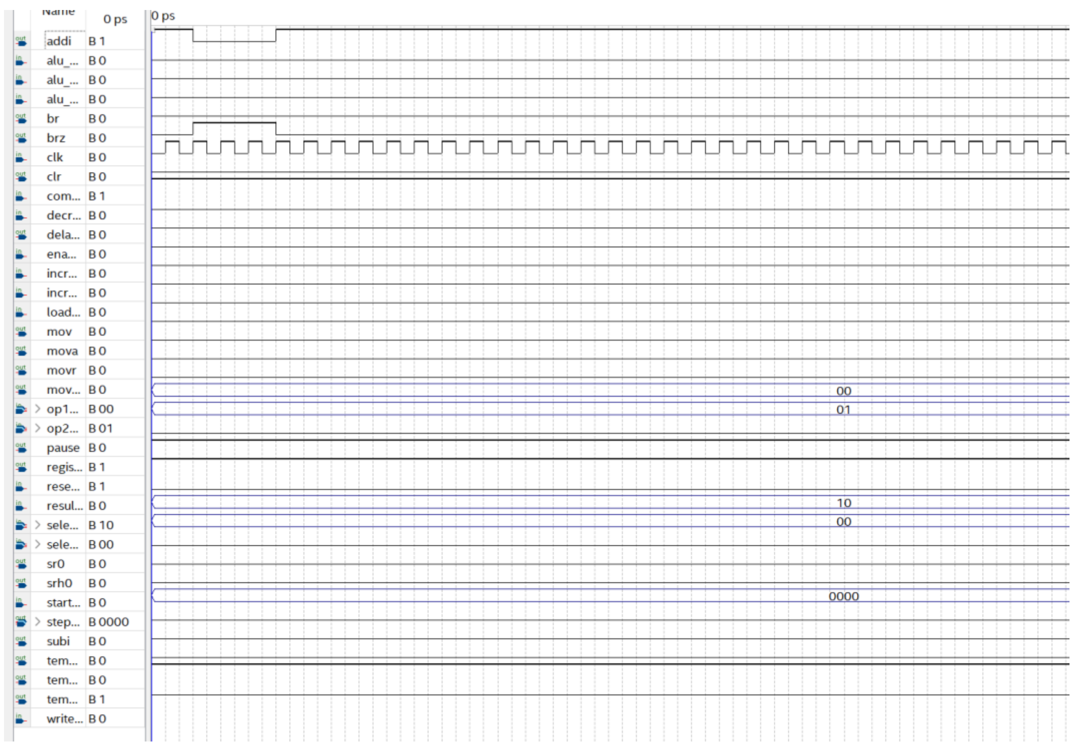
```verilog
        input start_delay_counter, enable_delay_counter,
        input commit_branch, increment_pc,
        input alu_add_sub, alu_set_low, alu_set_high,
        input load_temp, increment_temp, decrement_temp,
        input [1:0] select_immediate,
        input [1:0] select_write_address,
        // Status outputs
        output br, brz, addi, subi, sr0, srh0, clr, mov, mova, movr, movrhs, pause,
        output delay_done,
        output temp_is_positive, temp_is_negative, temp_is_zero,
        output register0_is_zero,
        // Motor control outputs
        output [3:0] stepper_signals,
        // debugging
        output [7:0] debug_reg1, debug_reg2

);
// The comment /*synthesis keep*/ after the declaration of a wire
// prevents Quartus from optimizing it, so that it can be observed in simulation
// It is important that the comment appear before the semicolon
wire [7:0] position /*synthesis keep*/;
wire [7:0] delay /*synthesis keep*/;
wire [7:0] register0 /*synthesis keep*/;

wire [7:0] alu_result; //alu output
wire [7:0] pc_result; //pc output
wire [7:0] instruction_output; //instruction memory output
wire [2:0] wr_addr_sel; //wire address selector output
wire [7:0] res_mux; //result mux output
wire [7:0] selected0;
wire [7:0] selected1;
wire [7:0] mux1out;
wire [7:0] mux2out;
wire [7:0] imm; //immediate extractor output

decoder the_decoder (
  // Inputs
```

```
    .instruction (instruction_output[7:2]),
    // Outputs
    .br (br),
    .brz (brz),
    .addi (addi),
    .subi (subi),
    .sr0 (sr0),
    .srh0 (srh0),
    .clr (clr),
    .mov (mov),
    .mova (mova),
    .movr (movr),
    .movrhs (movrhs),
    .pause (pause)
);
regfile the_regfile(
    // Inputs
    .clk (clk),
    .reset_n (reset_n),
    .write (write_reg_file),
    .data (res_mux[7:0]),
    .select0 (instruction_output[1:0]),
    .select1 (instruction_output[3:2]),
    .wr_select (wr_addr_sel),
    // Outputs
    .selected0 (selected0),
    .selected1 (selected1),
    .delay (delay),
    .position (position),
    .register0 (register0)
);

op1_mux the_op1_mux(
    // Inputs
    .select (op1_mux_select),
    .pc (pc_result),
    .register (selected0),
    .register0 (register0),
    .position (position),
    // Outputs
    .result(mux1out)
);

op2_mux the_op2_mux(
    // Inputs
    .select (op2_mux_select),
    .register (selected1),
    .immediate (imm),
    // Outputs
    .result (mux2out)
);

delay_counter the_delay_counter(
    // Inputs
    .clk(clk),
    .reset_n (reset_n),
    .start (start_delay_counter),
    .enable (enable_delay_counter),
    .delay (delay),
    // Outputs
    .done (delay_done)
);

stepper_rom the_stepper_rom(
    // Inputs
    .address (position[2:0]),
    .clock (clk),
    // Outputs
    .q (stepper_signals) //going to stepper motor
);

assign debug_reg1[7:0] = register0[7:0]; //for debugging. remove later. use this to check value of registers 0, 2, 3 or
other wires
assign debug_reg2[7:0] = delay[7:0];
```

```
pc the_pc(
   // Inputs
   .clk (clk),
   .reset_n (reset_n),
   .branch (commit_branch),
   .increment (increment_pc),
   .newpc (alu_result),
   // Outputs
   .pc (pc_result)
);

instruction_rom the_instruction_rom(
   // Inputs
   .address (pc_result),
   .clock (clk),
   // Outputs
   .q (instruction_output)
);

alu the_alu(
   // Inputs
   .add_sub (alu_add_sub),
   .set_low (alu_set_low),
   .set_high (alu_set_high),
   .operanda (mux1out),
   .operandb (mux2out),
   // Outputs
   .result (alu_result)
);

temp_register the_temp_register(
   // Inputs
   .clk (clk),
   .reset_n (reset_n),
   .load (load_temp),
   .increment (increment_temp),
   .decrement (decrement_temp),
   .data (selected0),
   // Outputs
   .negative (temp_is_negative),
   .positive (temp_is_positive),
   .zero (temp_is_zero)
);

immediate_extractor the_immediate_extractor(
   // Inputs
   .instruction (instruction_output), //unsure about the bit range
   .select (select_immediate),
   // Outputs
   .immediate (imm)
);

write_address_select the_write_address_select(
   // Inputs
   .select (select_write_address),
   .reg_field0 (instruction_output[1:0]),
   .reg_field1 (instruction_output[3:2]),
   // Outputs
   .write_address(wr_addr_sel)
);

result_mux the_result_mux (
   .select_result (result_mux_select),
   .alu_result (alu_result),
   .result (res_mux[7:0])
);

branch_logic the_branch_logic(
   // Inputs
   .register0 (register0),
   // Outputs
   .branch (register0_is_zero)
);
```

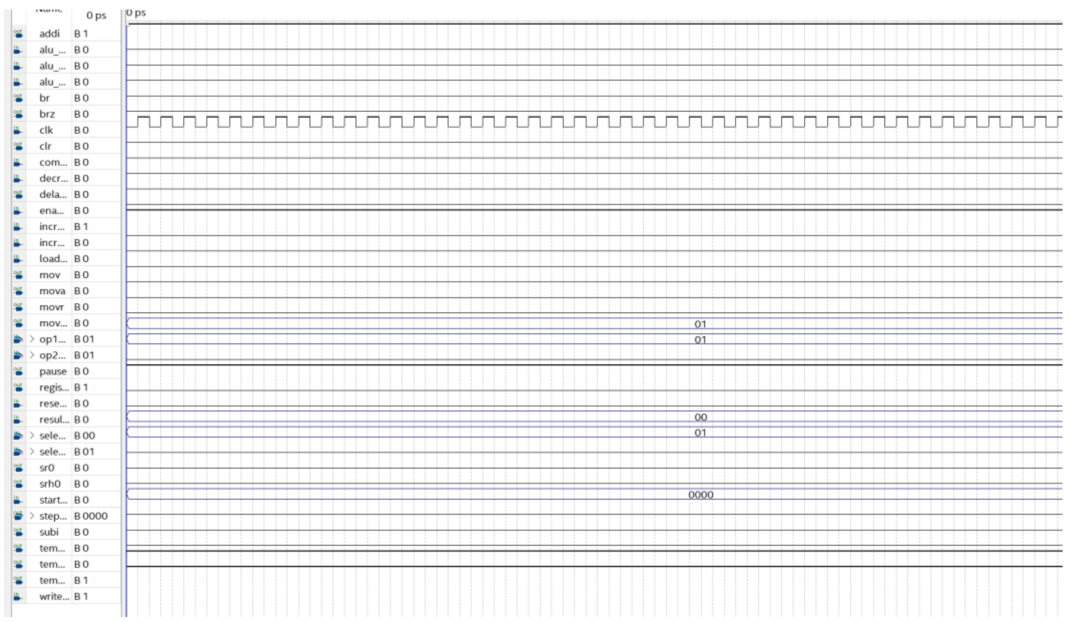
```
Endmodule
```

## BRZ Command



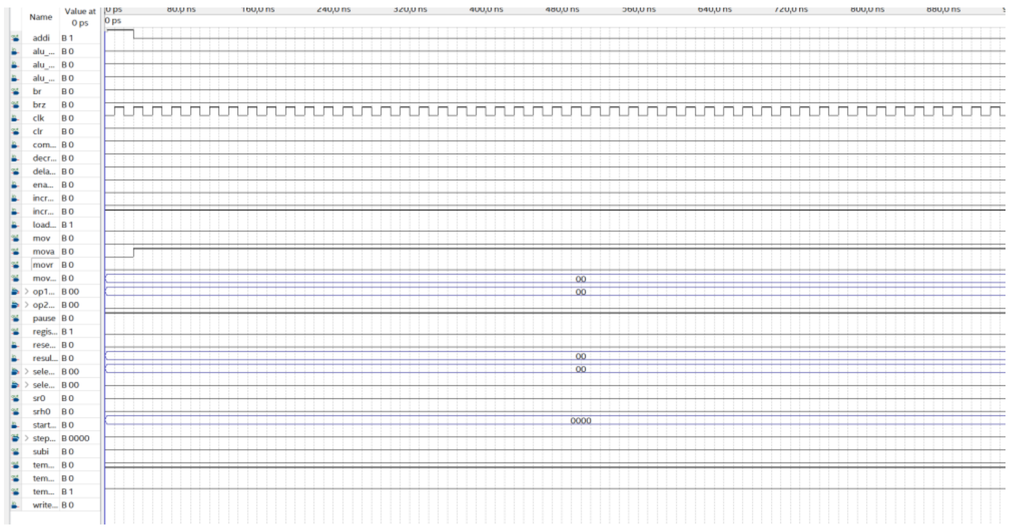
| Name | 0 ps |
|---|---|
| addi | B 1 |
| alu_... | B 0 |
| alu_... | B 0 |
| alu_... | B 0 |
| br | B 0 |
| brz | B 0 |
| clk | B 0 |
| clr | B 0 |
| com... | B 1 |
| decr... | B 0 |
| dela... | B 0 |
| ena... | B 0 |
| incr... | B 0 |
| incr... | B 0 |
| load... | B 0 |
| mov | B 0 |
| mova | B 0 |
| movr | B 0 |
| mov... | B 0 |
| op1... | B 00 |
| op2... | B 01 |
| pause | B 0 |
| regis... | B 1 |
| rese... | B 1 |
| resul... | B 0 |
| sele... | B 10 |
| sele... | B 00 |
| sr0 | B 0 |
| srh0 | B 0 |
| start... | B 0 |
| step... | B 0000 |
| subi | B 0 |
| tem... | B 0 |
| tem... | B 0 |
| tem... | B 1 |
| write... | B 0 |

## ADDI Command



| Name | 0 ps |
|---|---|
| addi | B 1 |
| alu_... | B 0 |
| alu_... | B 0 |
| alu_... | B 0 |
| br | B 0 |
| brz | B 0 |
| clk | B 0 |
| clr | B 0 |
| com... | B 0 |
| decr... | B 0 |
| dela... | B 0 |
| ena... | B 0 |
| incr... | B 1 |
| incr... | B 0 |
| load... | B 0 |
| mov | B 0 |
| mova | B 0 |
| movr | B 0 |
| mov... | B 0 |
| op1... | B 01 |
| op2... | B 01 |
| pause | B 0 |
| regis... | B 1 |
| rese... | B 0 |
| resul... | B 0 |
| sele... | B 00 |
| sele... | B 01 |
| sr0 | B 0 |
| srh0 | B 0 |
| start... | B 0 |
| step... | B 0000 |
| subi | B 0 |
| tem... | B 0 |
| tem... | B 0 |
| tem... | B 1 |
| write... | B 1 |

MOVR Command



## Assembly

The assembly code below causes the motor to move three steps clockwise and then move four half-steps counter-clockwise. The delay between each step is 1/2 of a second.

```
CLR R0 # R0 <- 0
SR0 2 # R0 <- 2
SRH0 3 # R0 <- 50

MOV R3, R0 # R3 <- 1 (define delay to be 0.5 of a second)

CLR R0 # R0 <- 0
SR0 2 # R0 <- 1
MOV R1, R0 # R1 <- 1 step

CLR R0 # R0 <- 0
SR0 3 # R0 <- 3 (setting the iterator to loop 3 times for 3 CW steps)

MOVR R1 # Move the motor 1 step clockwise
SUBI R0, 1 # decrement loop iterator
PAUSE
BRZ 2 # When finished loop of 3 steps, go to half-stepping
BR -4 # Branch to MOVR for next iteration of loop

CLR R0 # R0 <- 0
SR0 15 # R0 <- 7
SRH0 15 # R0 <- 15 or -1
MOV R1, R0 # R1 <- -1 step

CLR R0 # R0 <- 0
```

```
SR0 4 # R0 <- 4 (setting the iterator to loop 4 times for 4 CCW steps)

MOVRHS R1 # Move the motor 1 half step counter clockwise
SUBI R0, 1 # decrement loop iterator
PAUSE
BRZ -16 # When finished loop of 4 steps, go back to full stepping
BR -4 # Branch to MOVR for next iteration of loop
```

Assembly to Machine Code Converter

```
script="ADDI 1 69;ADDI 1 5;MOV 0 1";
lines = script.split(";");
binaryline="";
for line in lines:
    args = line.split(" ");
    if(args[0]=="BR"):
        binaryline+="100"+format(int(args[1]), '05b')+"\n";
    elif(args[0]=="BRZ"):
        binaryline+="101"+format(int(args[1]), '05b')+"\n";
    elif(args[0]=="ADDI"):
        binaryline+="000"+format(int(args[1]), '03b')+format(int(args[2]),
'02b')+"\n";
    elif(args[0]=="SUBI"):
        binaryline+="001"+format(int(args[1]), '03b')+format(int(args[2]),
'02b')+"\n";
    elif(args[0]=="SR0"):
        binaryline+="0100"+format(int(args[1]), '04b')+"\n";
    elif(args[0]=="SRH0"):
        binaryline+="0101"+format(int(args[1]), '04b')+"\n";
    elif(args[0]=="CLR"):
        binaryline+="011000"+format(int(args[1]), '02b')+"\n";
    elif(args[0]=="MOV"):
        binaryline+="0111"+format(int(args[1]), '02b')+format(int(args[2]),
'02b')+"\n";
    elif(args[0]=="MOVR"):
        binaryline+="110001"+format(int(args[1]), '02b')+"\n";
    elif(args[0]=="MOVRHS"):
        binaryline+="110010"+format(int(args[1]), '02b')+"\n";
    elif(args[0]=="PAUSE"):
        binaryline+="11111111"+"\n";
print(binaryline);
```

Machine code

```
01100000
01000001
01111100
01000000
01010011
01110100
01100000
01001010
11000101
00100100
10100010
10011101
11111111
01000000
01011101
01111100
01100000
01001010
11000101
00100100
10100010
10111101
11111111
11001001
10000000
```