MCU Report

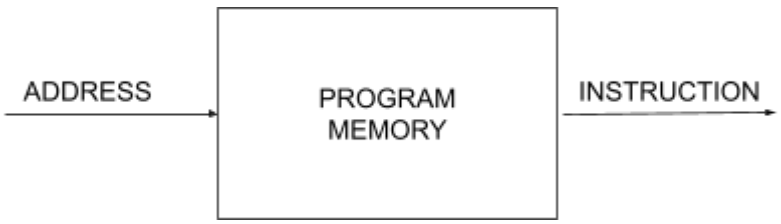
1.1 INTRODUCTION:

The objective of this project is to design an 8-bit microcontroller with components like Program Memory, Data Memory, Register File, ALU, Instruction Decoder, Data Hazard Stall and Branch Detection.

2.1 PROGRAM MEMORY:

The Program Memory is the module which contains all the instructions that the MCU can perform. The value from the program counter goes into the Program Memory. The program counter gives an 8-bit address value which can access 256 memory locations in the Program Memory. Each memory unit consists of a 17-bit value which contains the Instruction to be executed.

2.2 Module diagram for Program Memory:

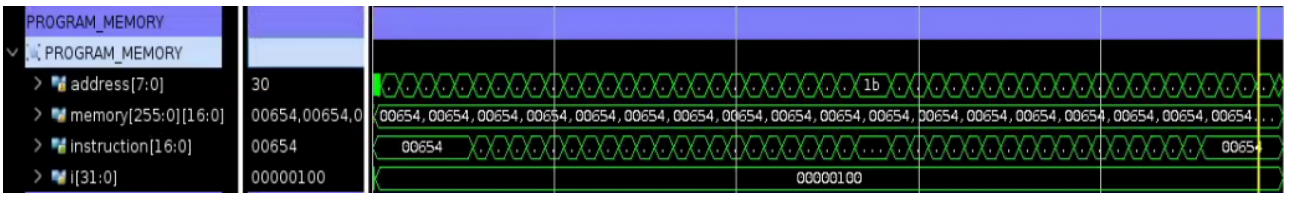


2.3 Verilog Code for Program Memory:

```
module program_memory(address, instruction);
   input [7:0] address;
   output [16:0] instruction;


   reg [16:0] memory [255:0];
   reg [16:0] instruction;
   integer i;
   initial begin

      for(i=0; i<256 ; i=i+1)begin
         case(i)
         0: memory[i] = 17'b00000011001010100;
```

```
        6: memory[i] = 17'b00001011001010100;
        8: memory[i] = 17'b00010011001000011;
        10: memory[i] = 17'b00011011001000011;
        12: memory[i] = 17'b00100011001010100;
        14: memory[i] = 17'b00101011001010100;
        16: memory[i] = 17'b00110011001010100;
        18: memory[i] = 17'b00111011001010100;
        20: memory[i] = 17'b01000011001000011;
        22: memory[i] = 17'b01001011001010100;
        24: memory[i] = 17'b01010011011010100;
        26: memory[i] = 17'b01010011001010100;
        28: memory[i] = 17'b01011011001010100;
        30: memory[i] = 17'b01100011001010100;
        32: memory[i] = 17'b01101011001010100;
        34: memory[i] = 17'b01110011001010100;
        36: memory[i] = 17'b01111011001010100;
        38: memory[i] = 17'b10001011001010100;
        40: memory[i] = 17'b10010011001010100;
        42: memory[i] = 17'b10011011001010100;
        44: memory[i] = 17'b10100011001010100;

        default: memory[i] = 17'b00000011001010100;
        endcase
        #0.01;
    end
end

always@(address)begin
    instruction = memory[address];
end


Endmodule
```

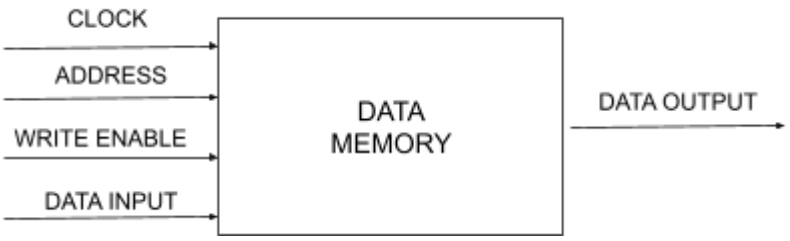
2.4 Waveform of Program Memory:



3.1 DATA MEMORY:

The data memory is the memory unit in which the values can be stored into. The data memory consists of a write enable, clk, data input and address as inputs, and a data output as output. This module works with the clock. Whenever the write enable is set to 1, the module writes the value in data input into the address provided. Else, it just reads the value in the address provided and give it as the output.

3.2 Module Diagram for Data Memory:



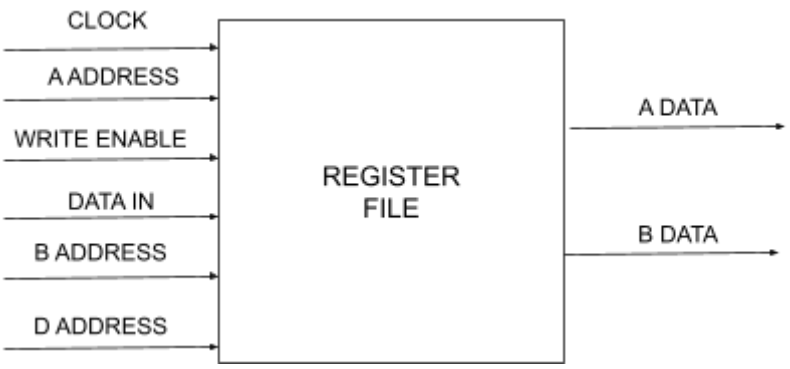
3.3 Verilog code for Data Memory:

```
module data_memory(clk, address, write_enable, data_input, data_output);
   input clk;
   input [7:0] address;
   input write_enable;
   input  [7:0] data_input;
   output [7:0] data_output;

   reg [7:0] memory [255:0];
   reg [7:0] data_output;

   always @(posedge clk)begin

      data_output <= memory[address];

   end

   always @(negedge clk) begin
      if(write_enable)begin
         memory[address] <= data_input ;
      end
   end

endmodule
```

4.1 REGISTER FILE

The register file is a module which has been designed to contain 8 registers, each of which can store an 8 bit value in it. Of the 8 registers, register 0 always has the value 0 stored in it. The operation is similar to that of a data memory. The module stores a value into the register when the read-write enable is set to 1. Else it just reads the value of registers based on the address provided and gives them out as the output.

4.2 Module diagram for Register File:

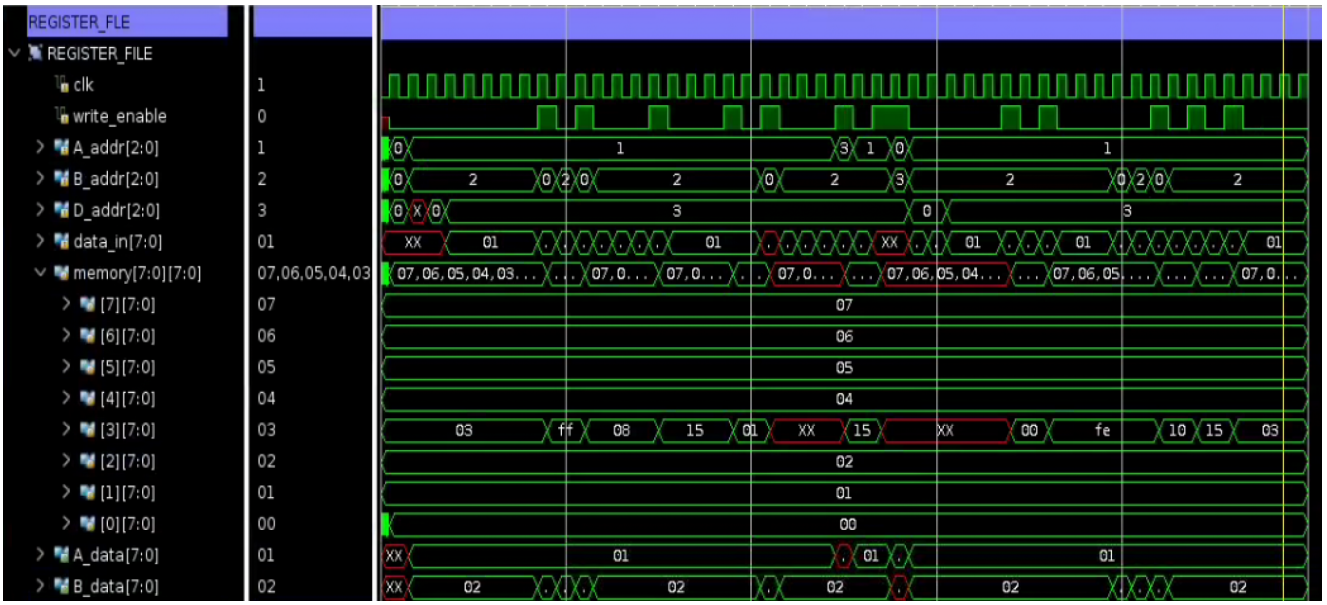


4.3 Verilog code for Register File:

```
module register_file(clk, A_addr, B_addr, D_addr, data_in, write_enable, A_data, B_data);
  input clk, write_enable;
  input [2:0] A_addr, B_addr, D_addr;
  input [7:0] data_in;
  output [7:0] A_data, B_data;

  reg [7:0] memory[7:0];
  reg [7:0] A_data;
  reg [7:0] B_data;
  integer i;

  initial begin
    for(i=1 ; i<8 ; i=i+1)begin
      memory[i] <= i;
      //memory_B[i] <= i;
      #0.01;
    end
  end
  always@(posedge clk)begin
    memory[0] <= 0;
```

```
    end

    always@(negedge clk)begin

        if (write_enable)begin
            memory [D_addr] <= data_in;
        end
    end

    always@(A_addr)begin
        A_data <= memory[A_addr];
    end

    always@(B_addr)begin
        B_data <= memory[B_addr];
    end

endmodule
```

4.4 Waveform of Register File:

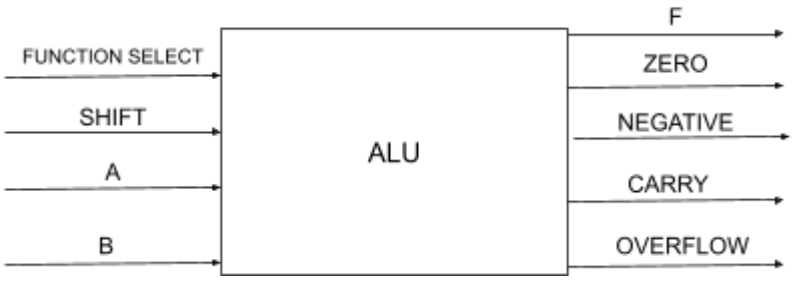


5.1 ALU:

The ALU is the module which handles all the arithmetic and logical operations. It consists of a Function select, shift, A and B as the inputs and F, zero, negative, carry and overflow as the outputs. The outputs contain 4 flags which are used in different operations such as branch at zero or branch at non-zero. The designed ALU

performs operations such as addition, subtraction, Exclusive-OR, OR, NOT, AND, set if less than and logical shift left.

5.2 Module diagram for ALU:



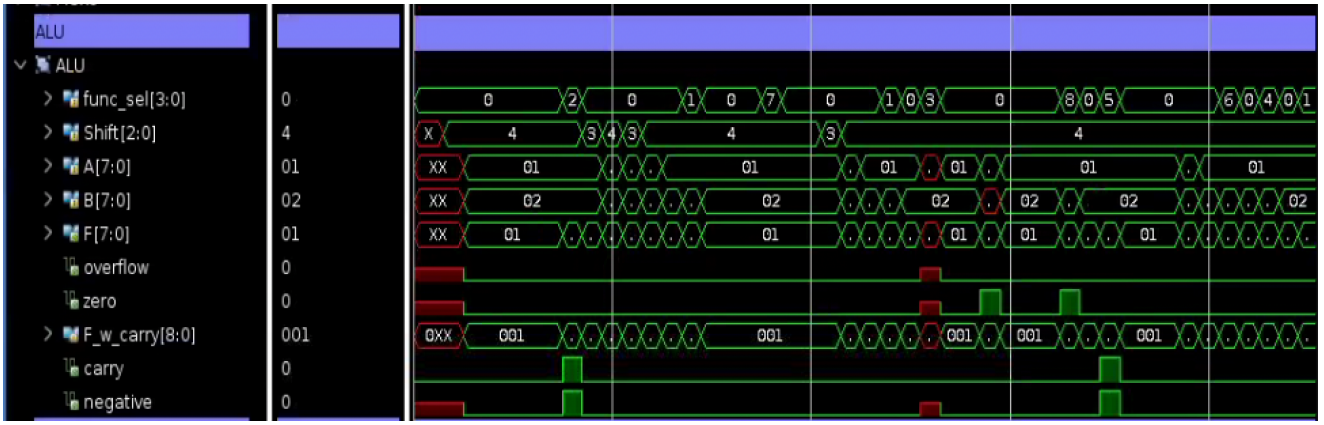
5.3 Verilog code for ALU:

```
module ALU(func_sel, Shift, A, B, F, zero, negative, carry, overflow);
   input [3:0] func_sel;
   input [2:0] Shift;
   input [7:0]  A, B;
   output [7:0] F;
   output zero, negative, carry, overflow;

   wire [7:0] A,B;
   reg [7:0] F;
   reg overflow, zero;
   reg [8:0] F_w_carry;
   reg carry, negative;

   parameter ADD = 4'd1,
   SUB = 4'd2,
   XOR = 4'd3,
   OR = 4'd4,
   NOT = 4'd5,
   LSL = 4'd6,
   SLT = 4'd7,
   AND = 4'd8,
   PASSB = 4'd0,
   PASSA = 4'd9;



   always@(func_sel)begin
      carry <= 0;
      negative <= 0;
```

```
zero <= 0;
overflow <= 0;

case(func_sel)
   ADD: begin
      assign F_w_carry = A+B;
   end

   SUB:begin
      assign F_w_carry = A + (~B) + 1;
      F = F_w_carry[7:0];
   end

   XOR:begin
      assign F_w_carry = A^B;
  end

   OR:begin
      assign F_w_carry = A|B;
   end

   NOT:begin
      assign F_w_carry = ~A[7:0];
      F_w_carry[8] <= 0;
   end

   LSL:begin
      assign F_w_carry = A << Shift;
      F = F_w_carry[7:0];
   end

   SLT:begin
      assign F = (A<B) ? 1 : 0 ;
      assign F_w_carry = (A<B) ? 1 : 0 ;
  end
   AND:begin
      assign F_w_carry= A & B ;
   end

   PASSA: assign F_w_carry = B;
   PASSB: assign F_w_carry = A;
```

```
    endcase
    assign F = F_w_carry[7:0];
    assign zero = F ? 0 : 1;
    assign negative = F[7];
    assign carry = F_w_carry[8];
    assign overflow = F_w_carry[8] ^ F_w_carry[7];
  end

endmodule
```

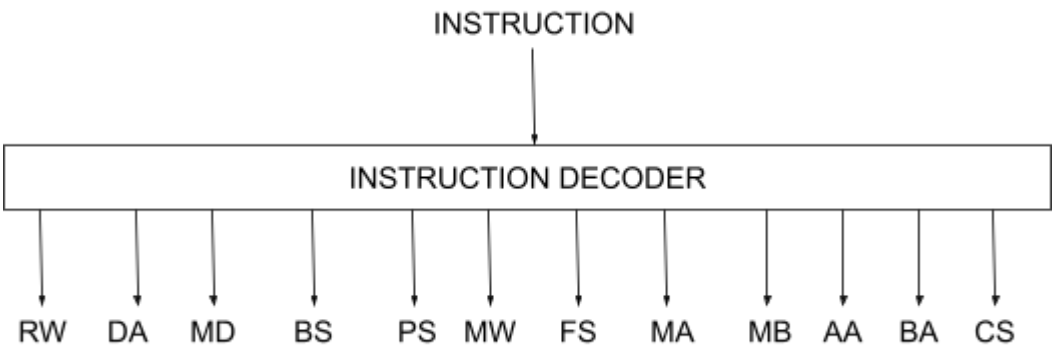
5.4 Waveform of ALU:



6.1 INSTRUCTION DECODER:

The instruction decoder is the main brain of the MCU. The Instruction Decoder plays a vital role in deciding what operations should take place for a particular instruction. It makes sure if the read-write of the register file should be set, operation of various MUXs etc. The input for this module is a 17-bit instruction and has 12 outputs namely read-write of register file, data address, MUX D sel, branch select, zero toggle, memory write, function select, MUX A sel, MUX B sel, Register A address, Register B address and constant select.

6.2 Module diagram for Instruction Decoder:



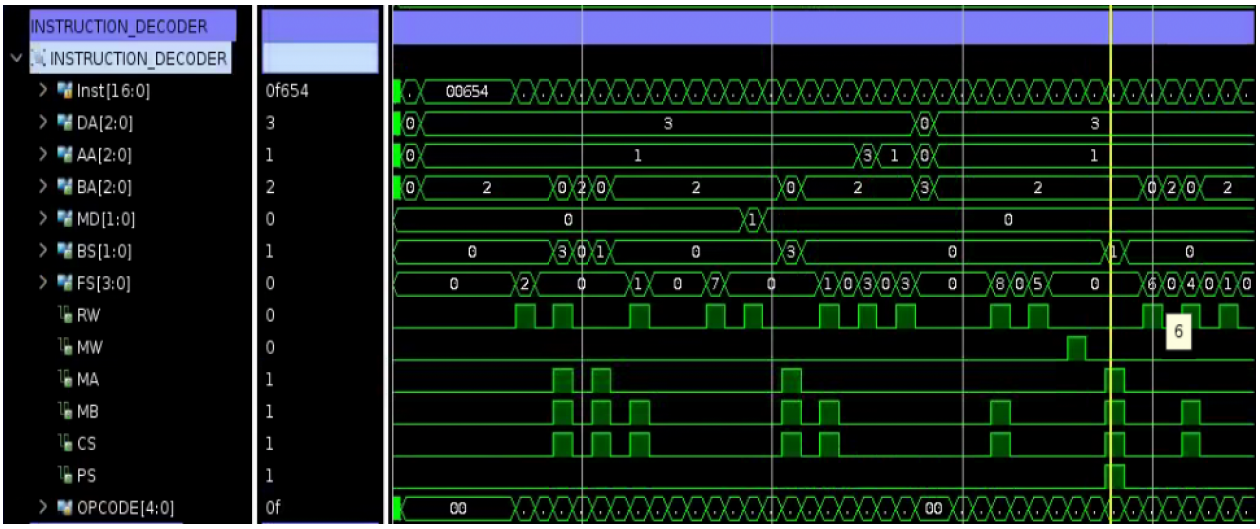
6.3 Verilog code of Instruction Decoder:

```
module Inst_Decode(Inst, RW,DA, MD, BS, PS, MW, FS, MA, MB, MD, AA, BA, CS);
   input Inst;
   output RW,DA, MD, BS, PS, MW, FS, MA, MB, AA, BA, CS;

   wire [16:0] Inst;
   reg [2:0] DA, AA, BA;
   reg [1:0] MD, BS;
   reg [3:0] FS;
   reg RW, MW, MA,MB, CS, PS;
   reg [4:0]OPCODE;

   always@(*)begin
     OPCODE <= Inst[16:12];

     case(OPCODE)
         5'b00000:begin RW=0; DA=Inst[11:9]; MD=2'b00; BS=2'b00; PS=0; MW=0;
FS=4'b0000; MA=0; MB=0; AA=Inst[8:6]; BA=Inst[5:3]; CS = 0; end

         5'b00001:begin RW=1; DA=Inst[11:9]; MD=2'b00; BS=2'b00; PS=0; MW=0;
FS=4'b0010; MA=0; MB=0; AA=Inst[8:6]; BA=Inst[5:3]; CS = 0; end //SUB

         5'b00010:begin RW=1; DA=Inst[11:9]; MD=2'b00; BS=2'b11; PS=0; MW=0;
FS=4'b0000; MA=1; MB=1; AA=Inst[8:6]; BA=Inst[5:3]; CS = 1; end //JML

         5'b00011:begin RW=0; DA=Inst[11:9]; MD=0; BS=2'b01; PS=0; MW=0;
FS=4'b0000; MA=1; MB=1; AA=Inst[8:6]; BA=Inst[5:3]; CS=1 ; end //BZ

         5'b00100:begin RW=1; DA=Inst[11:9]; MD=2'b00; BS=2'b00; PS=0; MW=0;
FS=4'b0001; MA=0; MB=1; AA=Inst[8:6]; BA=Inst[5:3]; CS = 1; end //ADI

         //5'b00101:begin RW=; DA=; MD=; BS=; PS=; MW=; FS=; MA=; MB=; AA=; BA=;
CS = ; end //OUT

         5'b00110:begin RW=1; DA=Inst[11:9]; MD=2'b00; BS=2'b00; PS=0; MW=0;
FS=4'b0111; MA=0; MB=0; AA=Inst[8:6]; BA=Inst[5:3]; CS = 0; end //SLT

         5'b00111:begin RW=1; DA=Inst[11:9]; MD=2'b01; BS=2'b00; PS=0; MW=0;
FS=4'b0000; MA=0; MB=0; AA=Inst[8:6]; BA=Inst[5:3]; CS = 0; end //LD
```

```
        5'b01000:begin RW=1'b0; DA=Inst[11:9]; MD=2'b00; BS=2'b11; PS=1'b0;
MW=1'b0; FS=4'b0000; MA=1; MB=1; AA=Inst[8:6]; BA=Inst[5:3]; CS = 1'b1; end //JMP

        5'b01001:begin RW=1; DA=Inst[11:9]; MD=2'b00; BS=0; PS=0; MW=0;
FS=4'b0001; MA=0; MB=1; AA=Inst[8:6]; BA=Inst[5:3]; CS = 1; end //AIU

        5'b01010:begin RW=1; DA=Inst[11:9]; MD=2'b00; BS=2'b00; PS=0; MW=0;
FS=4'b0011; MA=0; MB=0; AA=Inst[8:6]; BA=Inst[5:3]; CS = 0; end //XOR

        //5'b01011:begin RW=; DA=; MD=; BS=; PS=; MW=; FS=; MA=; MB=; AA=; BA=;
CS = ; end //IN

        5'b01100:begin RW=1; DA=Inst[11:9]; MD=2'b00; BS=2'b00; PS=0; MW=0;
FS=4'b1000; MA=0; MB=1; AA=Inst[8:6]; BA=Inst[5:3]; CS = 1; end //ANI

        5'b01101:begin RW=1; DA=Inst[11:9]; MD=2'b00; BS=2'b00; PS=0; MW=0;
FS=4'b0101; MA=0; MB=0; AA=Inst[8:6]; BA=Inst[5:3]; CS = 0; end //NOT

        5'b01110:begin RW=0; DA=Inst[11:9]; MD=2'b00; BS=2'b00; PS=0; MW=1;
FS=4'b0000; MA=0; MB=0; AA=Inst[8:6]; BA=Inst[5:3]; CS = 0; end //STR

        5'b01111:begin RW=0; DA=Inst[11:9]; MD=2'b00; BS=2'b01; PS=1; MW=0;
FS=4'b0000; MA=1; MB=1; AA=Inst[8:6]; BA=Inst[5:3]; CS = 1; end //BNZ

        5'b10000:begin RW=1; DA=Inst[11:9]; MD=2'b00; BS=2'b00; PS=0; MW=0;
FS=4'b0001; MA=0; MB=0; AA=Inst[8:6]; BA=0; CS = 0; end //MOV

        5'b10001:begin RW=1; DA=Inst[11:9]; MD=2'b00; BS=2'b00; PS=0; MW=0;
FS=4'b0110; MA=0; MB=0; AA=Inst[8:6]; BA=0; CS = 0; end //LSL

        5'b10010:begin RW=1; DA=Inst[11:9]; MD=2'b00; BS=2'b00; PS=0; MW=0;
FS=4'b0100; MA=0; MB=1; AA=Inst[8:6]; BA=0; CS = 1; end //ORI

        5'b10011:begin RW=1; DA=Inst[11:9]; MD=2'b00; BS=2'b00; PS=0; MW=0;
FS=4'b0001; MA=0; MB=0; AA=Inst[8:6]; BA=Inst[5:3]; CS = 0; end //ADD

        //5'b10100:begin RW=0; DA=0; MD=2'b00; BS=2'b10; PS=0; MW=0;
FS=4'b0000; MA=0; MB=0; AA=Inst[8:6]; BA=0; CS = 1; end //JMR

        default: begin RW=0; DA=Inst[11:9]; MD=0; BS=0; PS=0; MW=0; FS=0; MA=0;
MB=0; AA=Inst[8:6]; BA=Inst[5:3]; CS = 0; end
    endcase
```

```
    end

endmodule
```

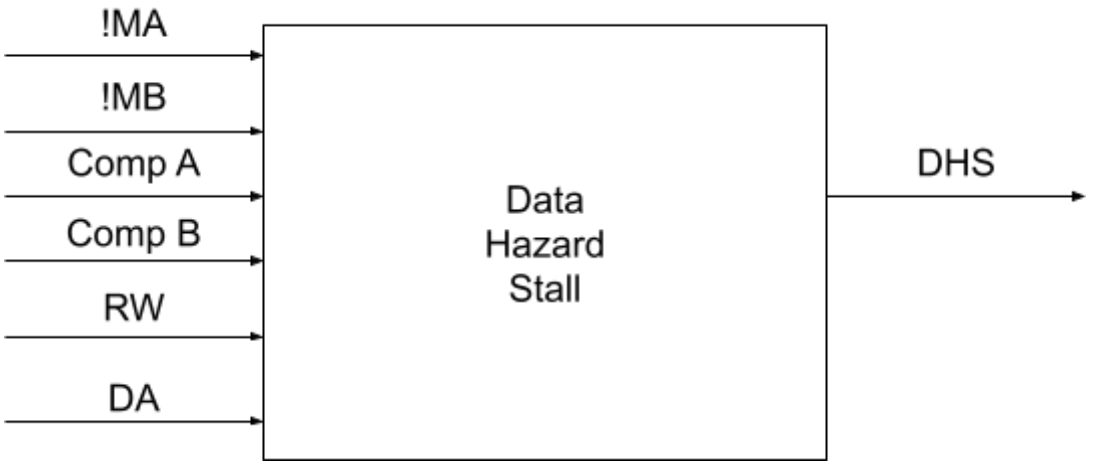
6.4 Waveform of Instruction Decoder:



7.1 DATA HAZARD STALL:

The Data Hazard Stall is a concept which is implemented to prevent an invalid data getting written in the write back stage. When an operation is using a register, it is likely that the next instruction might use the same register as its operand aswell. But as various stages of the pipeline might take different clock cycles to complete its writeback stage to occur, the fetch cycle may take a wrong data as its operands. Hence in situations like this a Data Hazard Stall can be crutial. When the DHS is set, it makes sure that the value doesn't get written into the register file or data memory.

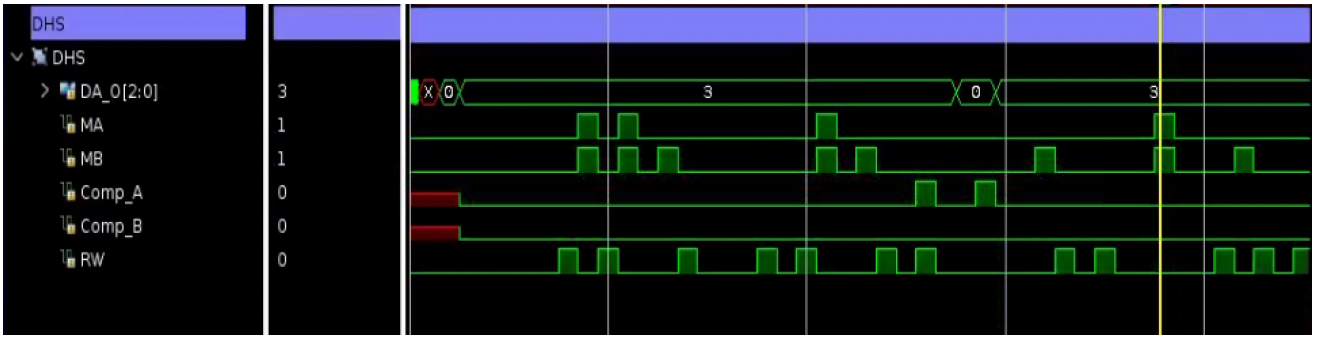
7.2 Module diagram for DHS:



7.3 Verilog code for DHS:

```
module DHS_m(DA_O, MA, RW, Comp_A, Comp_B, MB, HA, HB);

   input DA_O, MA, RW, Comp_A, Comp_B, MB;
   output HA, HB;

   reg HA, HB;
   wire [2:0] DA_O;
   wire MA, MB, Comp_A, Comp_B, RW;

   always@(*)begin

      HA = (DA_O[2]||DA_O[1]||DA_O[0]) && RW && Comp_A && !MA;
      HB = (DA_O[2]||DA_O[1]||DA_O[0]) && RW && Comp_B && !MB;
   end


endmodule
```

7.4 Waveform of DHS:



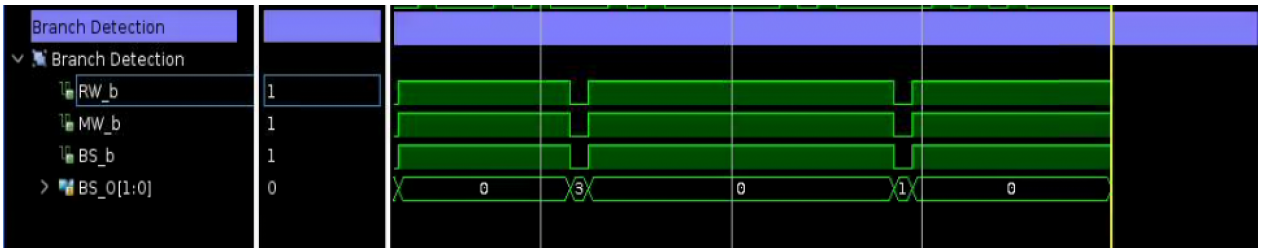
8.1 BRANCH DETECTION:

The Branch Detection is a concept which works similar to a DHS. When branch select is nonzero, then the compliment of the BS value which is zero, is given as the input for RW, MW and BS to avoid any values written into register or data memory.

8.2 Verilog code for Branch Detection:

```
module branch_detect(BS_O, RW_b, BS_b, MW_b);
   input BS_O;
   output RW_b,BS_b,MW_b;
   reg RW_b, MW_b, BS_b;
   wire [1:0] BS_O;


   always@(*)begin
      RW_b = !(BS_O[0] || BS_O[1]);
      BS_b = !(BS_O[0] || BS_O[1]);
      MW_b = !(BS_O[0] || BS_O[1]);
   end

endmodule
```

8.3 Waveform of Branch Detection:



9.1 MAIN MCU:

Once all these modules were developed, these modules have to be stitched together to work synchronously with the clock of the MCU. There are 4 MUXs namely MUX A, MUX B, MUX C and MUX D which must be designed and various registers and wires that must be initialized to connect various components of the MCU together. There are 3 program counter registers that are working 1 clock cycle behind each other. The IR register gets the 17 bit value from the program memory. There are 2 registers that save the value of register A and B. All these are connections are connected to work synchronously together.

9.2 Verilog code for MCU:

```
module MCU_main(clk);

   input clk;
   wire clk;
```

```
reg a;
wire b;
reg [7:0] address;
wire [16:0] instruction;
reg [16:0] IR;
reg [7:0] ALU_out, Mem_out;

reg [7:0] Bus_A, Bus_B;

//INSTRUCTION DECODER
wire RW, PS, MW, CS, MA, MB;
wire [2:0] DA, AA, BA;
wire [1:0] BS, MD;
wire [3:0] FS;

reg RW_O, PS_O, MW_O, CS_O, MA_O, MB_O;
reg [2:0] DA_O, AA_O, BA_O;
reg [1:0] BS_O, MD_O;
reg [3:0] FS_O;

reg RW_O_1;
reg [1:0] MD_O_1;
reg [2:0] DA_O_1;

//wire [16:0] Inst;

//BRANCH SELECT
wire zero;
reg [7:0] BrA, RAA;
wire [7:0] MC_out;


//DECODE
reg [5:0] IM ;
reg [2:0] SH ,SH_O;

// REGISTER FILE
wire [7:0] A_data, B_data;

//MUXs
wire [7:0] MA_out, MB_out, MD_out;
```

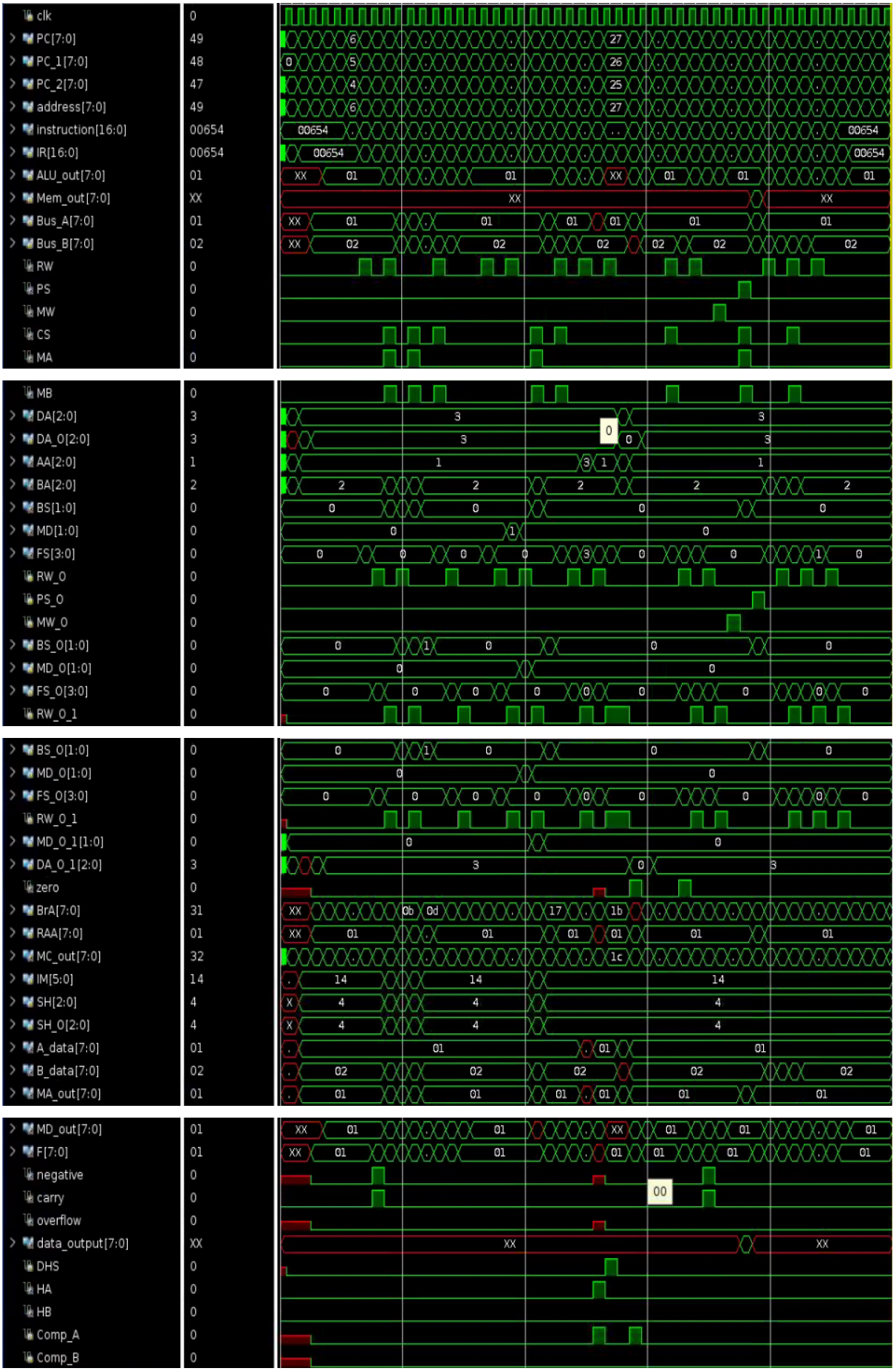
```
    //ALU
    wire [7:0] F;
    wire negative, carry, overflow;

    //DATA MEMORY
    wire [7:0] data_output;

    reg [7:0] PC, PC_1, PC_2, PC_3;
    reg [1:0] MC;

    //DHS
    wire HA, HB;
    reg DHS;

    //Comp
    reg Comp_A, Comp_B;

    //BRANCH DETECT
    wire RW_b, MW_b, BS_b;

    program_memory pm_main(.address(address), .instruction(instruction));
    Inst_Decode Instdecode(.Inst(IR), .RW(RW), .DA(DA), .MD(MD), .BS(BS), .PS(PS),
.MW(MW), .FS(FS), .MA(MA), .MB(MB), .AA(AA), .BA(BA), .CS(CS));
    branch_seq Branch_sel (.BS(BS_O), .zero(zero), .PS(PS_O), .PC(PC), .BrA(BrA),
.RAA(RAA), .MC_out(MC_out));
    register_file regfile(.clk(clk), .A_addr(AA), .B_addr(BA), .D_addr(DA_O_1),
.data_in(MD_out), .write_enable(RW_O_1), .A_data(A_data), .B_data(B_data));
    MUXs MUX_inst (.MA_out(MA_out), .MB_out(MB_out), .A_data(A_data),
.B_data(B_data), .PC_1(PC_1), .CS(CS), .Inst(IR), .MA(MA), .MB(MB),
.MD_out(MD_out), .F_out(ALU_out), .Mem_out(Mem_out), .MD(MD_O_1));
    ALU ALU_test (.func_sel(FS_O), .Shift(SH_O), .A(Bus_A), .B(Bus_B), .F(F),
.zero(zero), .negative(negative), .carry(carry), .overflow(overflow));
    data_memory DM_TEST (.clk(clk), .address(address), .write_enable(MW_O),
.data_input(Bus_B), .data_output(data_output) );
    DHS_m DHS_test(.DA_O(DA_O), .MA(MA), .RW(RW_O), .Comp_A(Comp_A),
.Comp_B(Comp_B), .MB(MB), .HA(HA), .HB(HB));
    branch_detect br_test(.BS_O(BS_O), .RW_b(RW_b), .BS_b(BS_b), .MW_b(MW_b));

    initial begin

        PC <= 8'b00000000;
```

```
    PC_1 <= 8'b00000000;
    PC_2 <= 8'b00000000;
    PC_3 <= 8'b00000000;
    IR = 17'd0;

    #10;
    //repeat(5) @ (posedge clk);


end



always@(posedge clk)begin

    if(DHS)begin
        PC = PC;
        PC_1 = PC_1;
        PC_2 = PC_2;
        IR = 0;
        DHS <= 0;

        RW_O = 0;
        MW_O = 0;
        DA_O <= 0;
        BS_O <= 0;

    end

    else begin
        PC = MC_out;
        PC_1 = (PC - 1'b1);
        PC_2 = (PC_1 - 1'b1);

        DA_O <= DA;
        BS_O <= BS;
        RW_O <= RW;
        MW_O <= MW;
    end

    if (BS_O)begin
        IR = 0;
```

```
      BS_O = 0;
   end

   else begin
      IR = address;
      BS_O = BS;
   end
end

always@(PC)begin

   address = PC;
   //BSM = BS;
end

always@(PC_1)begin
   IR = instruction ;
   SH = IR [2:0];
   IM = IR [5:0];
   //PC = MC_out;

   Comp_A = (DA_O == AA) ? 1 : 0;
   Comp_B = (DA_O == BA) ? 1 : 0;

end

always@(PC_2)begin
   Bus_A = MA_out;
   Bus_B = MB_out;
   RAA = Bus_A;

   MD_O_1 = MD_O;
   RW_O_1 = RW_O;
   DA_O_1 = DA_O;


   BS_O = BS;
   PS_O = PS;
   MW_O = MW && (MW_b);
   FS_O = FS;
   SH_O = SH;
```

```
    RW_O = RW && (RW_b);
    DA_O = DA ;
    MD_O = MD;

    BrA = PC_2 + Bus_B;

    ALU_out = F;
    Mem_out = data_output;

    DHS = HA || HB ;
  end

  always@(PC_3)begin

  end

endmodule
```

9.3 Waveform of MCU:


clk 0
PC[7:0] 49
PC_1[7:0] 48
PC_2[7:0] 47
address[7:0] 49
instruction[16:0] 00654
IR[16:0] 00654
ALU_out[7:0] 01
Mem_out[7:0] XX
Bus_A[7:0] 01
Bus_B[7:0] 02
RW 0
PS 0
MW 0
CS 0
MA 0
MB 0
DA[2:0] 3
DA_0[2:0] 3
AA[2:0] 1
BA[2:0] 2
BS[1:0] 0
MD[1:0] 0
FS[3:0] 0
RW_0 0
PS_0 0
MW_0 0
BS_0[1:0] 0
MD_0[1:0] 0
FS_0[3:0] 0
RW_0_1 0
BS_0[1:0] 0
MD_0[1:0] 0
FS_0[3:0] 0
RW_0_1 0
MD_0_1[1:0] 0
DA_0_1[2:0] 3
zero 0
BrA[7:0] 31
RAA[7:0] 01
MC_out[7:0] 32
IM[5:0] 14
SH[2:0] 4
SH_0[2:0] 4
A_data[7:0] 01
B_data[7:0] 02
MA_out[7:0] 01
MD_out[7:0] 01
F[7:0] 01
negative 0
carry 0
overflow 0
data_output[7:0] XX
DHS 0
HA 0
HB 0
Comp_A 0
Comp_B 0

OVERALL VERILOG CODE:

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 10/30/2021 08:12:12 PM
// Design Name:
// Module Name: MCU_main
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
module program_memory(address, instruction);
    input [7:0] address;
    output [16:0] instruction;


    reg [16:0] memory [255:0];
    reg [16:0] instruction;
    integer i;
    initial begin

        for(i=0; i<256 ; i=i+1)begin
            case(i)
            0: memory[i] = 17'b00000011001010100;
            6: memory[i] = 17'b00001011001010100;
            8: memory[i] = 17'b00010011001000011;
            10: memory[i] = 17'b00011011001000011;
            12: memory[i] = 17'b00100011001010100;
            14: memory[i] = 17'b00101011001010100;
            16: memory[i] = 17'b00110011001010100;
            18: memory[i] = 17'b00111011001010100;
```

```
            20: memory[i] = 17'b01000011001000011;
            22: memory[i] = 17'b01001011001010100;
            24: memory[i] = 17'b01010011011010100;
            26: memory[i] = 17'b01010011001010100;
            28: memory[i] = 17'b01011011001010100;
            30: memory[i] = 17'b01100011001010100;
            32: memory[i] = 17'b01101011001010100;
            34: memory[i] = 17'b01110011001010100;
            36: memory[i] = 17'b01111011001010100;
            38: memory[i] = 17'b10001011001010100;
            40: memory[i] = 17'b10010011001010100;
            42: memory[i] = 17'b10011011001010100;
            44: memory[i] = 17'b10100011001010100;

            default: memory[i] = 17'b00000011001010100;
            endcase
            #0.01;
        end
    end

    always@(address)begin
        instruction = memory[address];
    end


endmodule

module branch_seq(BS, zero, PS, PC, BrA, RAA, MC_out);

    input BS, zero, PS, PC, BrA, RAA;
    output [7:0] MC_out;



    wire [1:0] BS;
    wire zero, PS;
    wire [7:0] RAA, BrA, PC;


    reg [1:0] MC;
    reg [7:0] MC_out;
```

```
    always@(*)begin

        MC[0] <= BS[0] && (BS[1] || (PS ^ BS[0]));
        MC[1] <= BS[1];

        case(MC)
            2'b00: MC_out= PC + 1'b1;
            2'b10: MC_out= RAA;
            2'b01: MC_out= BrA;
            2'b11: MC_out= BrA;
        endcase


    end

endmodule

module register_file(clk, A_addr, B_addr, D_addr, data_in, write_enable, A_data,
B_data);
    input clk, write_enable;
    input [2:0] A_addr, B_addr, D_addr;
    input [7:0] data_in;
    output [7:0] A_data, B_data;

    reg [7:0] memory[7:0];
    reg [7:0] A_data;
    reg [7:0] B_data;
    integer i;

    initial begin
        for(i=1 ; i<8 ; i=i+1)begin
            memory[i] <= i;
            //memory_B[i] <= i;
            #0.01;
        end
    end
    always@(posedge clk)begin
        memory[0] <= 0;
    end

    always@(negedge clk)begin
```

```
    if (write_enable)begin
      memory [D_addr] <= data_in;
    end
  end

  always@(A_addr)begin
    A_data <= memory[A_addr];
  end

  always@(B_addr)begin
    B_data <= memory[B_addr];
  end


endmodule

module Inst_Decode(Inst, RW,DA, MD, BS, PS, MW, FS, MA, MB, MD, AA, BA, CS);
  input Inst;
  output RW,DA, MD, BS, PS, MW, FS, MA, MB, AA, BA, CS;

  wire [16:0] Inst;
  reg [2:0] DA, AA, BA;
  reg [1:0] MD, BS;
  reg [3:0] FS;
  reg RW, MW, MA,MB, CS, PS;
  reg [4:0]OPCODE;



  always@(*)begin
    OPCODE <= Inst[16:12];

    case(OPCODE)
      5'b00000:begin RW=0; DA=Inst[11:9]; MD=2'b00; BS=2'b00; PS=0; MW=0;
FS=4'b0000; MA=0; MB=0; AA=Inst[8:6]; BA=Inst[5:3]; CS = 0; end

      5'b00001:begin RW=1; DA=Inst[11:9]; MD=2'b00; BS=2'b00; PS=0; MW=0;
FS=4'b0010; MA=0; MB=0; AA=Inst[8:6]; BA=Inst[5:3]; CS = 0; end //SUB

      5'b00010:begin RW=1; DA=Inst[11:9]; MD=2'b00; BS=2'b11; PS=0; MW=0;
FS=4'b0000; MA=1; MB=1; AA=Inst[8:6]; BA=Inst[5:3]; CS = 1; end //JML
```

```
5'b00011:begin RW=0; DA=Inst[11:9]; MD=0; BS=2'b01; PS=0; MW=0;
FS=4'b0000; MA=1; MB=1; AA=Inst[8:6]; BA=Inst[5:3]; CS=1 ; end //BZ

5'b00100:begin RW=1; DA=Inst[11:9]; MD=2'b00; BS=2'b00; PS=0; MW=0;
FS=4'b0001; MA=0; MB=1; AA=Inst[8:6]; BA=Inst[5:3]; CS = 1; end //ADI

//5'b00101:begin RW=; DA=; MD=; BS=; PS=; MW=; FS=; MA=; MB=; AA=; BA=;
CS = ; end //OUT

5'b00110:begin RW=1; DA=Inst[11:9]; MD=2'b00; BS=2'b00; PS=0; MW=0;
FS=4'b0111; MA=0; MB=0; AA=Inst[8:6]; BA=Inst[5:3]; CS = 0; end //SLT

5'b00111:begin RW=1; DA=Inst[11:9]; MD=2'b01; BS=2'b00; PS=0; MW=0;
FS=4'b0000; MA=0; MB=0; AA=Inst[8:6]; BA=Inst[5:3]; CS = 0; end //LD

5'b01000:begin RW=1'b0; DA=Inst[11:9]; MD=2'b00; BS=2'b11; PS=1'b0;
MW=1'b0; FS=4'b0000; MA=1; MB=1; AA=Inst[8:6]; BA=Inst[5:3]; CS = 1'b1; end //JMP

5'b01001:begin RW=1; DA=Inst[11:9]; MD=2'b00; BS=0; PS=0; MW=0;
FS=4'b0001; MA=0; MB=1; AA=Inst[8:6]; BA=Inst[5:3]; CS = 1; end //AIU

5'b01010:begin RW=1; DA=Inst[11:9]; MD=2'b00; BS=2'b00; PS=0; MW=0;
FS=4'b0011; MA=0; MB=0; AA=Inst[8:6]; BA=Inst[5:3]; CS = 0; end //XOR

//5'b01011:begin RW=; DA=; MD=; BS=; PS=; MW=; FS=; MA=; MB=; AA=; BA=;
CS = ; end //IN

5'b01100:begin RW=1; DA=Inst[11:9]; MD=2'b00; BS=2'b00; PS=0; MW=0;
FS=4'b1000; MA=0; MB=1; AA=Inst[8:6]; BA=Inst[5:3]; CS = 1; end //ANI

5'b01101:begin RW=1; DA=Inst[11:9]; MD=2'b00; BS=2'b00; PS=0; MW=0;
FS=4'b0101; MA=0; MB=0; AA=Inst[8:6]; BA=Inst[5:3]; CS = 0; end //NOT

5'b01110:begin RW=0; DA=Inst[11:9]; MD=2'b00; BS=2'b00; PS=0; MW=1;
FS=4'b0000; MA=0; MB=0; AA=Inst[8:6]; BA=Inst[5:3]; CS = 0; end //STR

5'b01111:begin RW=0; DA=Inst[11:9]; MD=2'b00; BS=2'b01; PS=1; MW=0;
FS=4'b0000; MA=1; MB=1; AA=Inst[8:6]; BA=Inst[5:3]; CS = 1; end //BNZ

5'b10000:begin RW=1; DA=Inst[11:9]; MD=2'b00; BS=2'b00; PS=0; MW=0;
FS=4'b0001; MA=0; MB=0; AA=Inst[8:6]; BA=0; CS = 0; end //MOV
```

```
        5'b10001:begin RW=1; DA=Inst[11:9]; MD=2'b00; BS=2'b00; PS=0; MW=0;
FS=4'b0110; MA=0; MB=0; AA=Inst[8:6]; BA=0; CS = 0; end //LSL

        5'b10010:begin RW=1; DA=Inst[11:9]; MD=2'b00; BS=2'b00; PS=0; MW=0;
FS=4'b0100; MA=0; MB=1; AA=Inst[8:6]; BA=0; CS = 1; end //ORI

        5'b10011:begin RW=1; DA=Inst[11:9]; MD=2'b00; BS=2'b00; PS=0; MW=0;
FS=4'b0001; MA=0; MB=0; AA=Inst[8:6]; BA=Inst[5:3]; CS = 0; end //ADD

        //5'b10100:begin RW=0; DA=0; MD=2'b00; BS=2'b10; PS=0; MW=0;
FS=4'b0000; MA=0; MB=0; AA=Inst[8:6]; BA=0; CS = 1; end //JMR

        default: begin RW=0; DA=Inst[11:9]; MD=0; BS=0; PS=0; MW=0; FS=0; MA=0;
MB=0; AA=Inst[8:6]; BA=Inst[5:3]; CS = 0; end
    endcase

  end

endmodule


module MUXs(PC_1, Inst, A_data, B_data, CS, MA, MB, MD, MA_out, MB_out, MD_out,
F_out, Mem_out);

  input PC_1, Inst, A_data, B_data, CS, MD, MA, MB, F_out, Mem_out;
  output MA_out, MB_out, MD_out;

  reg [7:0] MA_out, MB_out, MD_out;
  wire [7:0] A_data, B_data, PC_1;
  wire CS;
  wire [16:0] Inst;
  wire MA, MB;
  wire [1:0] MD;
  wire [7:0] F_out, Mem_out;

  always@(*)begin
    case(MA)
      1'b0: MA_out = A_data;
      1'b1: MA_out = PC_1;
    endcase

    case(MB)
```

```
            1'b0: MB_out = B_data;
            //1'b1: MB_out = {CS,Inst[5:0]};

            1'b1:begin
               if (CS == 1)begin
                  MB_out = {Inst[5], Inst[5], Inst[5:0]};
                  //MB_out =  Inst[5:0];
               end

               else
                  MB_out = {1'b0, 1'b0, Inst[5:0]};
            end
         endcase

         case(MD)
            2'b00: MD_out = F_out;
            2'b01: MD_out = Mem_out;

         endcase
      end
endmodule

module ALU(func_sel, Shift, A, B, F, zero, negative, carry, overflow);
   input [3:0] func_sel;
   input [2:0] Shift;
   input [7:0]  A, B;
   output [7:0] F;
   output zero, negative, carry, overflow;

   wire [7:0] A,B;
   reg [7:0] F;
   reg overflow, zero;
   reg [8:0] F_w_carry;
   reg carry, negative;

   parameter ADD = 4'd1,
   SUB = 4'd2,
   XOR = 4'd3,
   OR = 4'd4,
   NOT = 4'd5,
   LSL = 4'd6,
   SLT = 4'd7,
```

```
AND = 4'd8,
PASSB = 4'd0,
PASSA = 4'd9;



always@(func_sel)begin
   carry <= 0;
   negative <= 0;
   zero <= 0;
   overflow <= 0;

   case(func_sel)
      ADD: begin
         assign F_w_carry = A+B;
      end

      SUB:begin
         assign F_w_carry = A + (~B) + 1;
         F = F_w_carry[7:0];
      end

      XOR:begin
         assign F_w_carry = A^B;
      end

      OR:begin
         assign F_w_carry = A|B;
      end

      NOT:begin
         assign F_w_carry = ~A[7:0];
         F_w_carry[8] <= 0;
      end

      LSL:begin
         assign F_w_carry = A << Shift;
         F = F_w_carry[7:0];
      end

      SLT:begin
         assign F = (A<B) ? 1 : 0 ;
```

```
            assign F_w_carry = (A<B) ? 1 : 0 ;
          end
           AND:begin
              assign F_w_carry= A & B ;
           end

           PASSA: assign F_w_carry = B;
           PASSB: assign F_w_carry = A;

       endcase
       assign F = F_w_carry[7:0];
       assign zero = F ? 0 : 1;
       assign negative = F[7];
       assign carry = F_w_carry[8];
       assign overflow = F_w_carry[8] ^ F_w_carry[7];
    end

endmodule

module data_memory(clk, address, write_enable, data_input, data_output);
    input clk;
    input [7:0] address;
    input write_enable;
    input  [7:0] data_input;
    output [7:0] data_output;

    //wire write_enable;
    reg [7:0] memory [255:0];
    reg [7:0] data_output;

    always @(posedge clk)begin

       data_output <= memory[address];

    end

    always @(negedge clk) begin
       if(write_enable)begin
          memory[address] <= data_input ;
       end
    end
```

```
endmodule

module DHS_m(DA_O, MA, RW, Comp_A, Comp_B, MB, HA, HB);

    input DA_O, MA, RW, Comp_A, Comp_B, MB;
    output HA, HB;

    reg HA, HB;
    wire [2:0] DA_O;
    wire MA, MB, Comp_A, Comp_B, RW;

    always@(*)begin
        //DHS = () || ();
        HA = (DA_O[2]||DA_O[1]||DA_O[0]) && RW && Comp_A && !MA;
        HB = (DA_O[2]||DA_O[1]||DA_O[0]) && RW && Comp_B && !MB;
    end


endmodule

module branch_detect(BS_O, RW_b, BS_b, MW_b);
    input BS_O;
    output RW_b,BS_b,MW_b;
    reg RW_b, MW_b, BS_b;
    wire [1:0] BS_O;


    always@(*)begin
        RW_b = !(BS_O[0] || BS_O[1]);
        BS_b = !(BS_O[0] || BS_O[1]);
        MW_b = !(BS_O[0] || BS_O[1]);
    end

endmodule


module MCU_main(clk);

    input clk;
    wire clk;
```

```
reg a;
wire b;
reg [7:0] address;
wire [16:0] instruction;
reg [16:0] IR;
reg [7:0] ALU_out, Mem_out;

reg [7:0] Bus_A, Bus_B;

//INSTRUCTION DECODER
wire RW, PS, MW, CS, MA, MB;
wire [2:0] DA, AA, BA;
wire [1:0] BS, MD;
wire [3:0] FS;

reg RW_O, PS_O, MW_O, CS_O, MA_O, MB_O;
reg [2:0] DA_O, AA_O, BA_O;
reg [1:0] BS_O, MD_O;
reg [3:0] FS_O;

reg RW_O_1;
reg [1:0] MD_O_1;
reg [2:0] DA_O_1;

//wire [16:0] Inst;

//BRANCH SELECT
wire zero;
reg [7:0] BrA, RAA;
wire [7:0] MC_out;


//DECODE
reg [5:0] IM ;
reg [2:0] SH ,SH_O;

// REGISTER FILE
wire [7:0] A_data, B_data;

//MUXs
wire [7:0] MA_out, MB_out, MD_out;
```

```
    //ALU
    wire [7:0] F;
    wire negative, carry, overflow;

    //DATA MEMORY
    wire [7:0] data_output;

    reg [7:0] PC, PC_1, PC_2, PC_3;
    reg [1:0] MC;

    //DHS
    wire HA, HB;
    reg DHS;

    //Comp
    reg Comp_A, Comp_B;

    //BRANCH DETECT
    wire RW_b, MW_b, BS_b;

    program_memory pm_main(.address(address), .instruction(instruction));
    Inst_Decode Instdecode(.Inst(IR), .RW(RW), .DA(DA), .MD(MD), .BS(BS), .PS(PS),
.MW(MW), .FS(FS), .MA(MA), .MB(MB), .AA(AA), .BA(BA), .CS(CS));
    branch_seq Branch_sel (.BS(BS_O), .zero(zero), .PS(PS_O), .PC(PC), .BrA(BrA),
.RAA(RAA), .MC_out(MC_out));
    register_file regfile(.clk(clk), .A_addr(AA), .B_addr(BA), .D_addr(DA_O_1),
.data_in(MD_out), .write_enable(RW_O_1), .A_data(A_data), .B_data(B_data));
    MUXs MUX_inst (.MA_out(MA_out), .MB_out(MB_out), .A_data(A_data),
.B_data(B_data), .PC_1(PC_1), .CS(CS), .Inst(IR), .MA(MA), .MB(MB),
.MD_out(MD_out), .F_out(ALU_out), .Mem_out(Mem_out), .MD(MD_O_1));
    ALU ALU_test (.func_sel(FS_O), .Shift(SH_O), .A(Bus_A), .B(Bus_B), .F(F),
.zero(zero), .negative(negative), .carry(carry), .overflow(overflow));
    data_memory DM_TEST (.clk(clk), .address(address), .write_enable(MW_O),
.data_input(Bus_B), .data_output(data_output) );
    DHS_m DHS_test(.DA_O(DA_O), .MA(MA), .RW(RW_O), .Comp_A(Comp_A),
.Comp_B(Comp_B), .MB(MB), .HA(HA), .HB(HB));
    branch_detect br_test(.BS_O(BS_O), .RW_b(RW_b), .BS_b(BS_b), .MW_b(MW_b));

    initial begin

        PC <= 8'b00000000;
```

```
    PC_1 <= 8'b00000000;
    PC_2 <= 8'b00000000;
    PC_3 <= 8'b00000000;
    IR = 17'd0;

    #10;
    //repeat(5) @ (posedge clk);


end



always@(posedge clk)begin

    if(DHS)begin
        PC = PC;
        PC_1 = PC_1;
        PC_2 = PC_2;
        IR = 0;
        DHS <= 0;

        RW_O = 0;
        MW_O = 0;
        DA_O <= 0;
        BS_O <= 0;

    end

    else begin
        PC = MC_out;
        PC_1 = (PC - 1'b1);
        PC_2 = (PC_1 - 1'b1);

        DA_O <= DA;
        BS_O <= BS;
        RW_O <= RW;
        MW_O <= MW;
    end

    if (BS_O)begin
        IR = 0;
```

```
      BS_O = 0;
   end

   else begin
      IR = address;
      BS_O = BS;
   end
end

always@(PC)begin

   address = PC;
   //BSM = BS;
end

always@(PC_1)begin
   IR = instruction ;
   SH = IR [2:0];
   IM = IR [5:0];
   //PC = MC_out;

   Comp_A = (DA_O == AA) ? 1 : 0;
   Comp_B = (DA_O == BA) ? 1 : 0;

end

always@(PC_2)begin
   Bus_A = MA_out;
   Bus_B = MB_out;
   RAA = Bus_A;

   MD_O_1 = MD_O;
   RW_O_1 = RW_O;
   DA_O_1 = DA_O;


   BS_O = BS;
   PS_O = PS;
   MW_O = MW && (MW_b);
   FS_O = FS;
   SH_O = SH;
```

```
        RW_O = RW && (RW_b);
        DA_O = DA ;
        MD_O = MD;

        BrA = PC_2 + Bus_B;

        ALU_out = F;
        Mem_out = data_output;

        DHS = HA || HB ;
    end

    always@(PC_3)begin

    end

endmodule
```

TEST BENCH:

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 11/11/2021 09:46:10 PM
// Design Name:
// Module Name: main_tb
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
```

```
////////////////////////////////////////////////////////////////////////////////


module MCU_tb;
    reg clk;

    reg [7:0] PC;
    MCU_main MCU_test(.clk(clk));

    always #10 clk = ~ clk;
    initial begin
            clk = 0;
            #10;


    end
endmodule
```