INTERNATIONAL INSTITUTE OF INFORMATION TECHNOLOGY

HYDERABAD

INTRODUCTION TO PROCESSOR ARCHITECTURE FINAL PROJECT REPORT

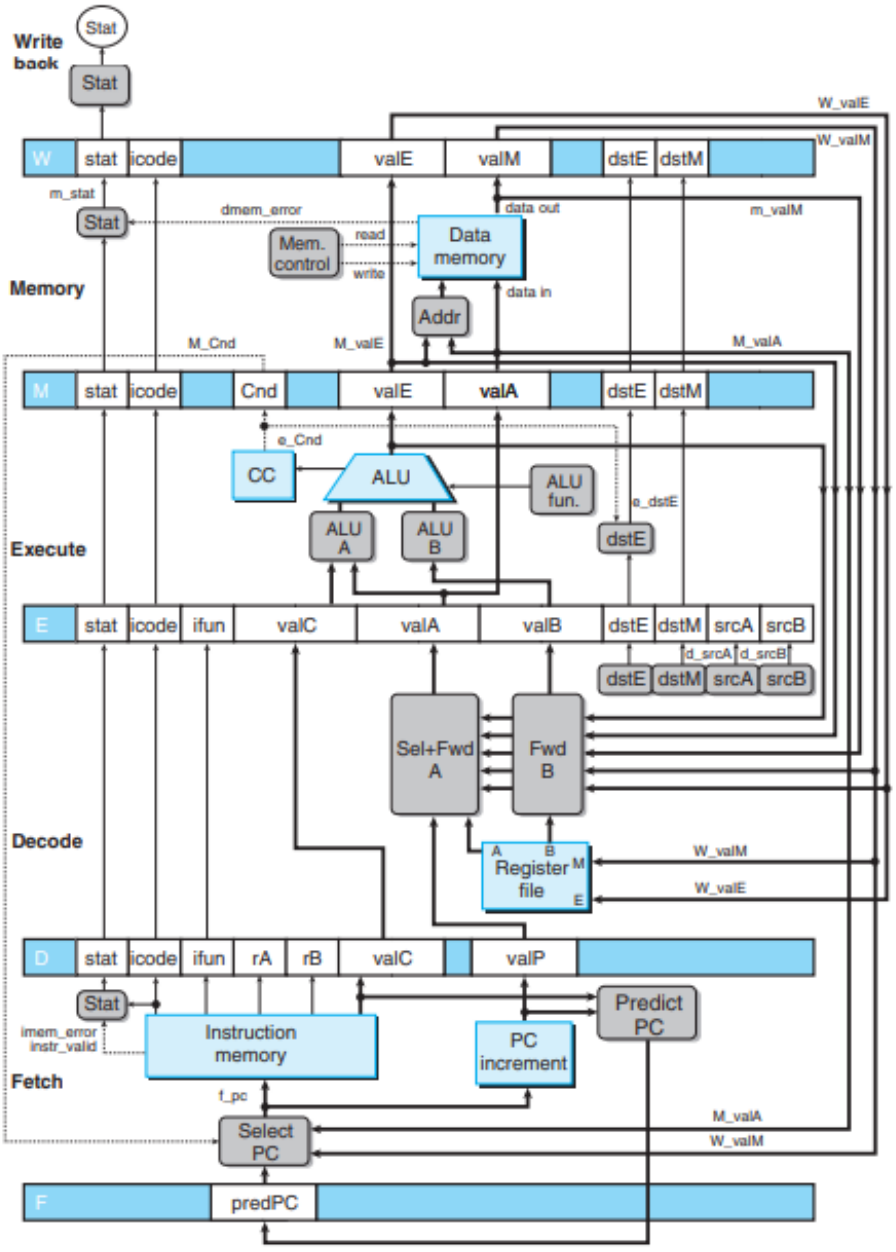
IMPLEMENTATION OF Y86-64 PIPELINED PROCESSOR.

BY TEAM: ***COOLZ***

Souvik Karfa (2020102051)
Yellapragada Sri Krishna Sarat Chandra (2021122002)

Table Of Contents

PIPELINED IMPLEMENTATION:



DEFINE THE MODULE PIPE (FOR PIPELINED IMPLEMENTATION) :

The processor takes clock and instruction memory as the inputs and keeps giving out the status as the output. Here, the instruction memory is given as a text file.

The pipeline registers corresponding to different modules are initiated here. Given below is the implementation of E pipeline register:

```
// E pipeline register
   reg [2:0] E_stat = 1;
   reg [3:0] E_icode = 1;
   reg [3:0] E_ifun = 0;
   reg [63:0] E_valC = 0;
   reg [63:0] E_valA = 0;
   reg [63:0] E_valB = 0;
   reg [3:0] E_dstE = 0;
   reg [3:0] E_dstM = 0;
   reg [3:0] E_srcA = 0;
   reg [3:0] E_srcB = 0;
```

Wires for the outputs of different stages are also initiated here. Given below is the implementation of Decode stage output:

```
// Decode stage output
   wire [2:0] d_stat;
   wire [3:0] d_icode;
   wire [3:0] d_ifun;
   wire [63:0] d_valC;
   wire [63:0] d_valA;
   wire [63:0] d_valB;
   wire [3:0] d_dstE;
   wire [3:0] d_dstM;
   wire [3:0] d_srcA;
   wire [3:0] d_srcB;
```

INTEGRATION OF MODULES:

All the sub-modules are integrated in this block and initiated here. Given is an example of the initiation of the fetch module:

```
fetch f(
    // Inputs from F register
    .F_predPC(F_predPC),

    // Inputs forwarded from M register
    .M_icode(M_icode),
    .M_Cnd(M_Cnd),
    .M_valA(M_valA),

    // Inputs forwarded from W register
    .W_icode(W_icode),
    .W_valM(W_valM),

    // Outputs
    .f_stat(f_stat),
    .f_icode(f_icode),
    .f_ifun(f_ifun),
    .f_rA(f_rA),
    .f_rB(f_rB),
    .f_valC(f_valC),
    .f_valP(f_valP),
    .f_predPC(f_predPC)
);
```

INTEGRATE THE PIPELINE CONTROL LOGIC MODULE:

```
PIPE_CON pip_con(
        // Inputs
        .D_icode(D_icode),
        .d_srcA(d_srcA),
        .d_srcB(d_srcB),
        .E_icode(E_icode),
        .E_dstM(E_dstM),
        .e_Cnd(e_Cnd),
        .M_icode(M_icode),
        .m_stat(m_stat),
        .W_stat(W_stat),

        // Outputs
        .W_stall(W_stall),
        .M_bubble(M_bubble),
        .E_bubble(E_bubble),
        .D_bubble(D_bubble),
        .D_stall(D_stall),
        .F_stall(F_stall)
    );
```

UPDATE THE PIPELINE REGISTERS:

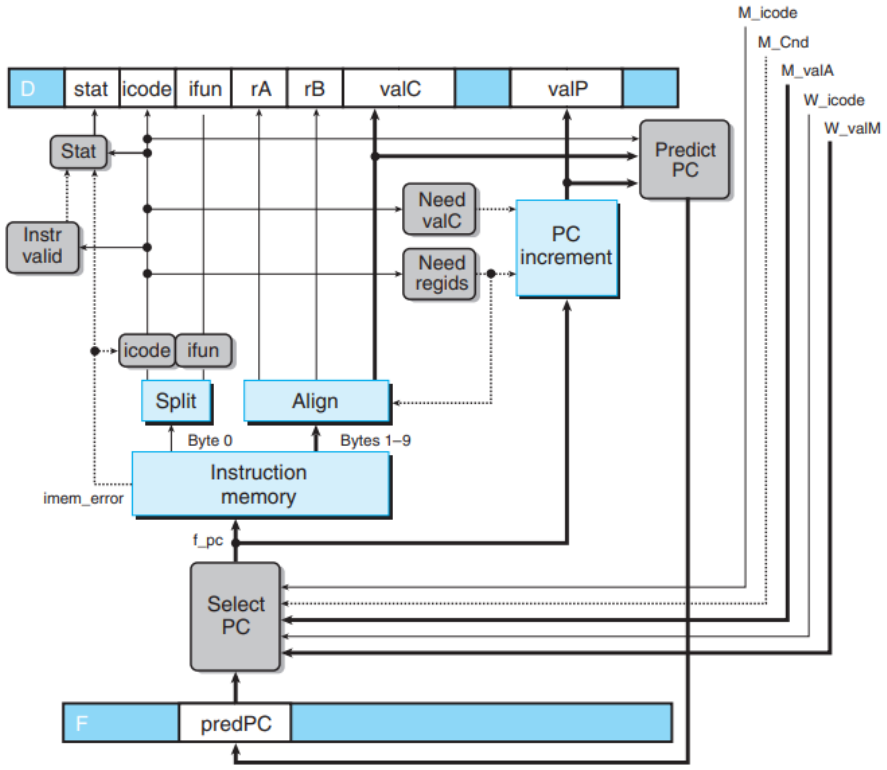
The registers updated at the positive edge of each clock cycle depending on the values of pipeline control signals of stall and bubble. Given below is an example of the implementation of D register update:

```
always @(posedge clk) begin
    if (D_stall == 0) begin
       if (D_bubble == 0) begin
           D_stat <= f_stat;
           D_icode <= f_icode;
           D_ifun <= f_ifun;
           D_rA <= f_rA;
           D_rB <= f_rB;
           D_valC <= f_valC;
           D_valP <= f_valP;
        end
        else begin
           D_stat <= 1;
           D_icode <= 1;
           D_ifun <= 0;
           D_rA <= 0;
           D_rB <= 0;
           D_valC <= 0;
           D_valP <= 0;
        end
    end
end
```

TESTBENCH:

```
module tb();
    reg clk;
    wire [2:0] Stat;
    PIPE dut(.clk(clk), .Stat(Stat));

    initial begin
        clk <= 0;
            forever #50 clk <= ~clk;
    end

    initial begin
        $dumpvars(0, tb);
        $monitor ("clk = %b Stat = %b", clk, Stat);
    end

    always @(*) begin
        if(Stat == 2) begin
            $finish;
        end
    end
endmodule
```

FETCH MODULE:



DEFINE THE FETCH MODULE:

```
module fetch(
    // Inputs
    input [63:0] F_predPC,
    input [3:0] M_icode,
    input M_Cnd,
    input [63:0] M_valA,
    input [3:0] W_icode,
    input [63:0] W_valM,

    // Outputs
    output [2:0] f_stat,
    output reg[3:0] f_icode,
    output reg[3:0] f_ifun,
    output reg[3:0] f_rA,
    output reg[3:0] f_rB,
    output reg[63:0] f_valC,
    output reg[63:0] f_valP,
    output [63:0] f_predPC
);
```

CHECK THE INSTRUCTION VALIDITY:

```
    reg instr_valid;

    always @(*) begin
        if (f_icode >= 0 && f_icode <= 11) begin
            instr_valid <= 1;
        end
        else begin
            instr_valid <= 0;
        end
    end
```

SELECT PC MODULE:

Given below is the implementation logic of selecting the PC.

```
always @(*) begin
    if (M_icode == 7 && M_Cnd == 0) begin
       f_pc <= M_valA;
   end
   else if (W_icode == 9) begin
       f_pc <= W_valM;
   end
   else begin
       f_pc <= F_predPC;
    end
end
```

PREDICT PC MODULE:

Given below is the implementation logic of selecting the PC.

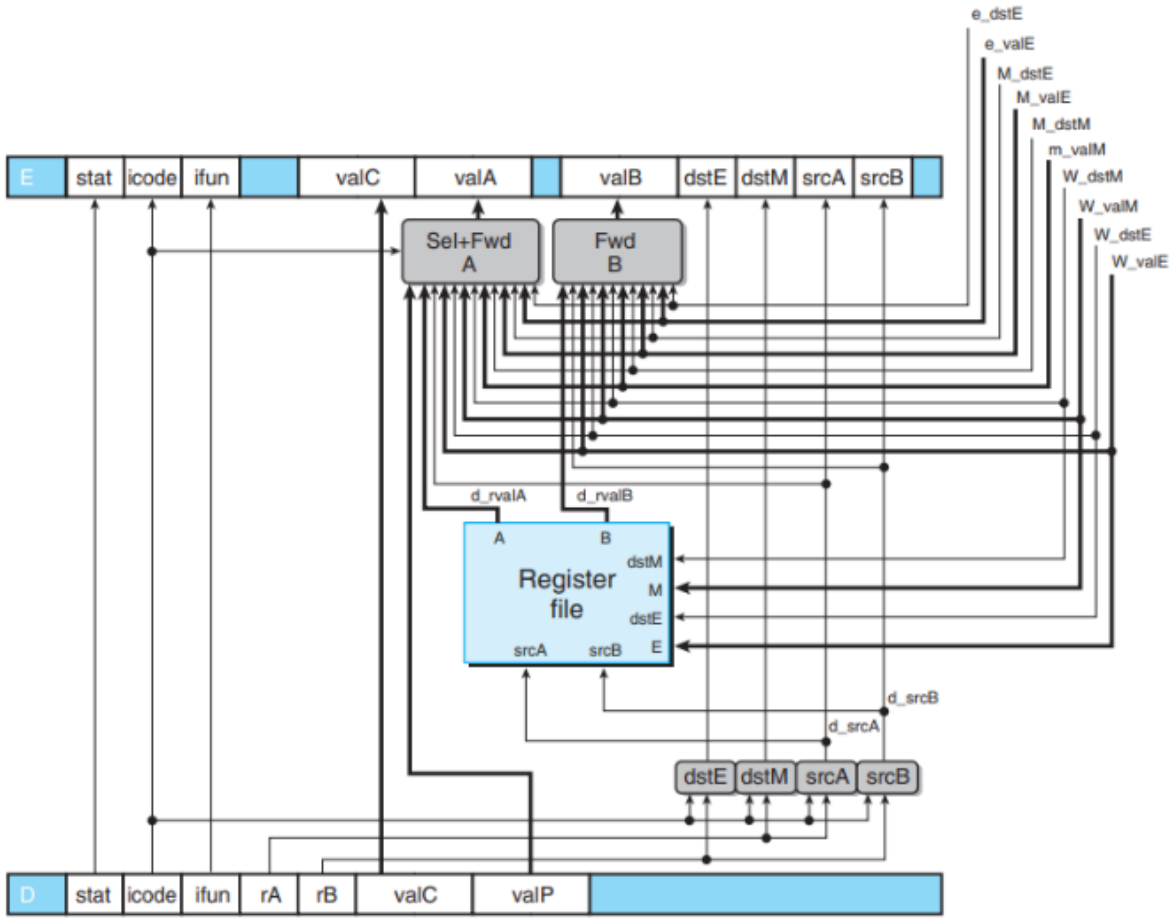
```
always @(*) begin
    if (f_icode == 7 || f_icode == 8) begin
       f_predPC <= f_valC;
    end
    else begin
        f_predPC <= f_valP;
    end
end
```

STATUS MODULE:

Status code of the newly fetched instruction is implemented in the following way:

```
always @(*) begin
    if (imem_error == 1) begin
         f_stat <= 3;
    end
    else if (instr_valid == 0) begin
        f_stat <= 4;
    end
    else if (f_icode == 0) begin
        f_stat <= 2;
    end
    else begin
        f_stat <= 1;
    end
end
```

DECODE MODULE:



SETUP THE REGISTER ARRAY:

```
// Initiating register arrays
reg [63:0] reg_array[0:15];
integer i;
initial begin
    for (i = 0; i < 16; i = i+1) begin
     reg_array[i] <= 0;
    end
end
```

UPDATING THE REGISTER FILE:

```
// Updating register file at positive edge of clock
always @(posedge clk) begin
   reg_array[W_dstM] <= W_valM;
   reg_array[W_dstE] <= W_valE;
end
```

ASSIGN d_dstE:

```
// Selecting d_dstE
always @(*) begin
    if (d_icode == 2 || d_icode == 3 || d_icode == 6) begin

        d_dstE <= D_rB;
    end
    else if (d_icode == 8 || d_icode == 9 || d_icode == 10 || d_icode == 11) begin

       d_dstE <= 4;
    end
    else begin

        d_dstE <= 15;
    end
end
```

ASSIGN d_srcA:

```
// Selecting d_srcA
always @(*) begin
        if (d_icode == 2 || d_icode == 4 || d_icode == 6) begin
            d_srcA <= D_rA;
        end
        else if (d_icode == 9 || d_icode == 10 || d_icode == 11) begin
            d_srcA <= 4;
        end
        else begin
            d_srcA <= 15;
        end
    end
```

READING FROM THE REGISTERS:

```
    // Reading d_rvalA and d_rvalB from register
    wire [63:0] d_rvalA;
    wire [63:0] d_rvalB;


    assign d_rvalA = reg_array[d_srcA];
    assign d_rvalB = reg_array[d_srcB];
```

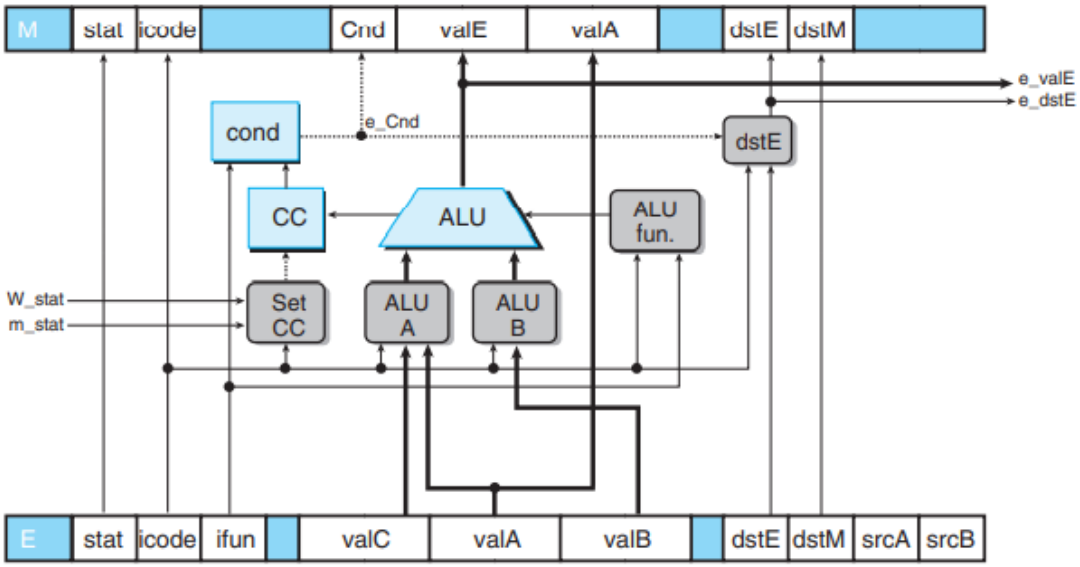
SELECT+FWD A:

```
// Sel+Fwd A Block
always @(*) begin
    if (d_icode == 7 || d_icode == 8) begin
        d_valA <= D_valP;
    end
    else if (d_srcA == e_dstE) begin
        d_valA <= e_valE;
    end
    else if (d_srcA == M_dstM) begin
        d_valA <= m_valM;
    end
    else if (d_srcA == M_dstE) begin
        d_valA <= M_valE;
    end
    else if (d_srcA == W_dstM) begin
        d_valA <= W_valM;
    end
    else if (d_srcA == W_dstE) begin
        d_valA <= W_valE;
    end
    else begin
        d_valA <= d_rvalA;
    end
end
```

SELECT B:

```
// Fwd B Block
always @(*) begin
   if (d_srcB == e_dstE) begin
        d_valB <= e_valE;
   end
   else if (d_srcB == M_dstM) begin
        d_valB <= m_valM;
   end
   else if (d_srcB == M_dstE) begin
        d_valB <= M_valE;
   end
   else if (d_srcB == W_dstM) begin
        d_valB <= W_valM;
   end
   else if (d_srcB == W_dstE) begin
        d_valB <= W_valE;
   end
   else begin
        d_valB <= d_rvalB;
   end
end
```

EXECUTE MODULE:



ASSIGN ALU VALUES:

```
reg [63:0] aluA, aluB;
always @(*) begin
    if (e_icode == 2) begin
        aluA <= E_valA;
        aluB <= 0;
    end
    else if (e_icode == 3) begin
        aluA <= E_valC;
        aluB <= 0;
    end
    else if (e_icode == 4) begin
        aluA <= E_valC;
        aluB <= E_valB;
    end
    else if (e_icode == 5) begin
        aluA <= E_valC;
        aluB <= E_valB;
    end
    else if (e_icode == 6) begin
        aluA <= E_valB;
        aluB <= E_valA;
    end
    else if (e_icode == 8) begin
        aluA <= -8;
        aluB <= E_valB;
    end
```

```
    else if (e_icode == 10) begin
        aluA <= -8;
        aluB <= E_valB;
    end
    else if (e_icode == 11) begin
        aluA <= 8;
        aluB <= E_valB;
    end
    else begin
        aluA <= 0;
        aluB <= 0;
    end
end
```

GET THE ALU'S FUNCTION:

```
// Setting up ALU operation
wire [3:0] alu_fun;
assign alu_fun = (e_icode == 6) ? E_ifun : 0;
```

DECIDE SET_CC:

```
wire set_cc;
    assign set_cc = ((e_icode == 6) && (m_stat != 2 && m_stat != 3 && m_stat != 4)
                     && (W_stat != 2 && W_stat != 3 && W_stat != 4)) ? 1 : 0;
```

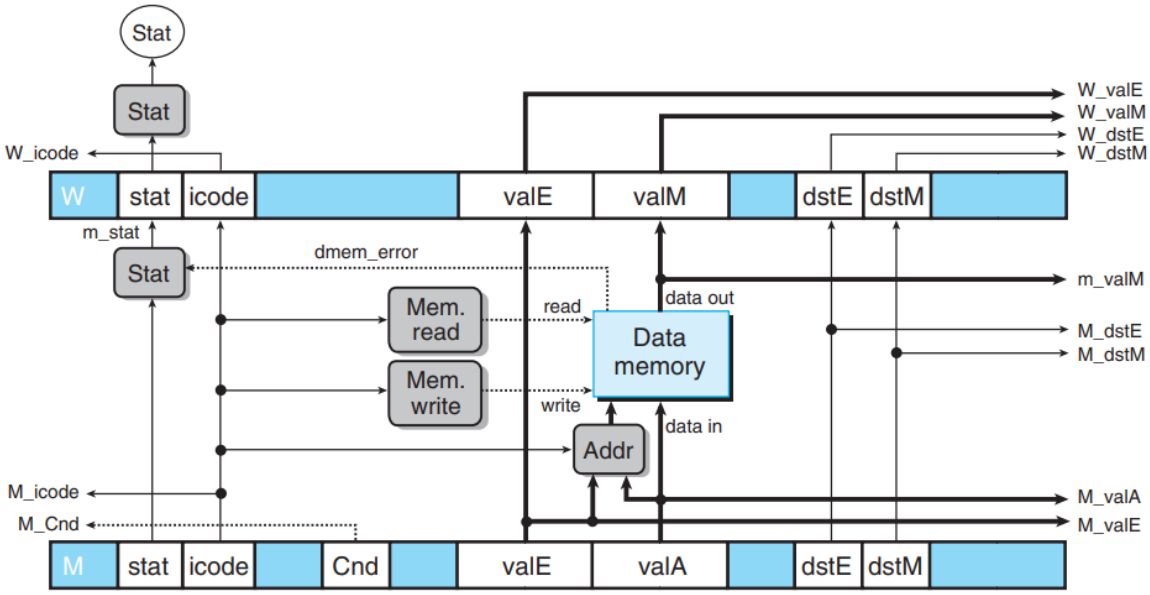
SETUP CONDITION CODE REGISTERS:

```
// Setting up condition code registers
 reg Z = 0, S = 0, Ov = 0;


 always @(posedge clk) begin
     if (set_cc == 1) begin
         Z <= (e_valE == 0) ? 1 : 0;
         S <= (e_valE[63] == 1) ? 1 : 0;


         if (alu_fun == 0) begin
             Ov <= ((aluA[63] == 1 && aluB[63] == 1 && e_valE[63] == 0)
                     || (aluA[63] == 0 && aluB[63] == 0 && e_valE[63] == 1))
                     ? 1 : 0;
         end
         else if (alu_fun == 1) begin
             Ov <= ((aluA[63] == 1 && aluB[63] == 0 && e_valE[63] == 0)
                     || (aluA[63] == 0 && aluB[63] == 1 && e_valE[63] == 1))
                     ? 1 : 0;
         end
         else begin
             Ov <= 0;
         end
     end
 end
```

GET THE DESTINATION VALUE:

```
    // Select e_dstE
    always @(*) begin

        if (e_icode == 2 && e_Cnd == 0) begin

            e_dstE <= 15;
        end
        else begin

            e_dstE <= E_dstE;
        end
    end


endmodule
```

MEMORY MODULE:



INITIATING THE DATA MEMORY:

```
// Initiating data memory
reg [63:0] Data_Mem[0:4095];
integer i;

initial begin
    for (i = 0; i < 4096; i = i+1) begin
        Data_Mem[i] <= 0;
    end
end
```

WRITE BLOCK:

```
// Mem_write block
reg Mem_write;

always @(*) begin

    if (m_icode == 4 || m_icode == 8 || m_icode == 10) begin

        Mem_write <= 1;
    end
    else begin

        Mem_write <= 0;
    end
end
```

READ BLOCK:

```
    // Mem_read block
    reg Mem_read;
    always @(*) begin
        if (m_icode == 5 || m_icode == 9 || m_icode == 11) begin
            Mem_read <= 1;
        end
        else begin
            Mem_read <= 0;
        end
    end
```

SELECT ADDRESS:

```
// Selecting Address
reg [63:0] m_addr;
always @(*) begin
    if (m_icode == 4 || m_icode == 5 || m_icode == 8 || m_icode == 10) begin
        m_addr <= m_valE;
    end
    else if (m_icode == 9 || m_icode == 11) begin
         m_addr <= M_valA;
    end
    else begin
         m_addr <= 4095;
    end
end
```

GENERATE dmem_error:

```
    // Checking memory error
    reg dmem_error;
    always @(*) begin
        if (m_addr < 4096 && m_addr >= 0) begin
            dmem_error <= 0;
        end
        else begin
            dmem_error <= 1;
        end
    end
```

WRITE BACK TO MEMORY:

```
// Assigning data_in
wire [63:0] m_data_in;
assign m_data_in = M_valA;

// Writing back to data memory at positive clock edge
always @(posedge clk) begin
    if (dmem_error == 0 && Mem_write == 1) begin
        Data_Mem[m_addr] <= m_data_in;
    end
end
```

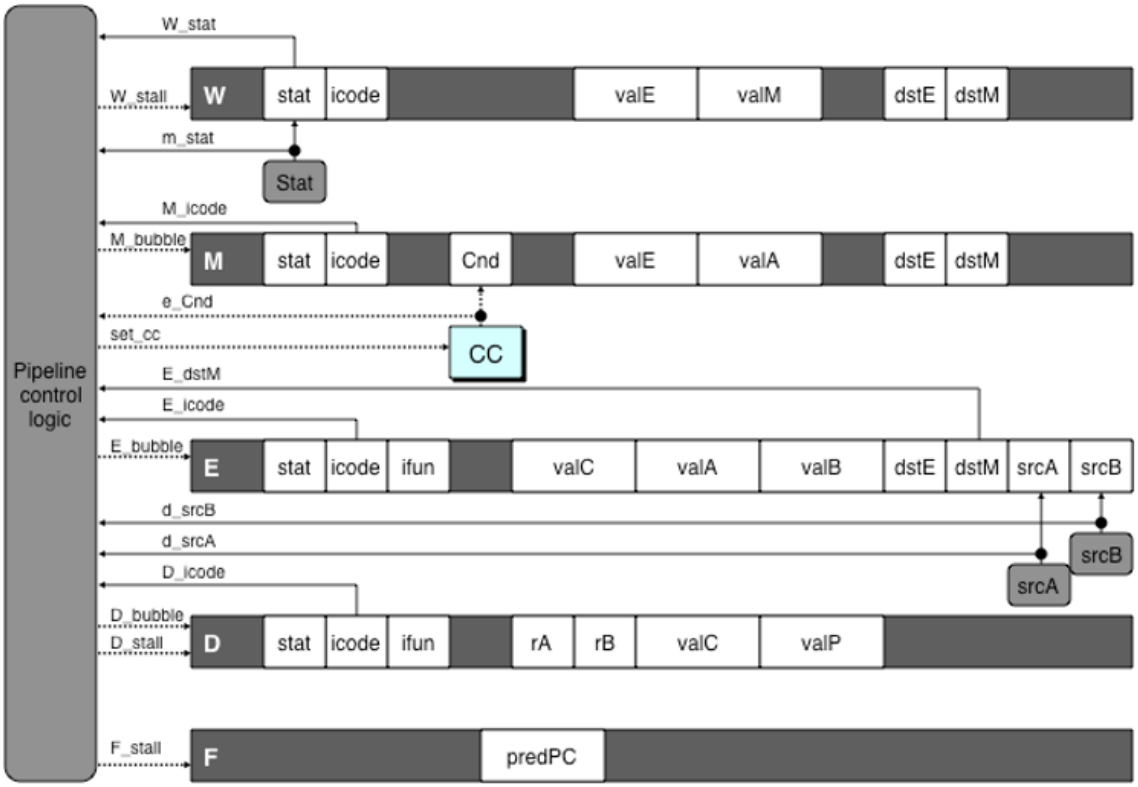
SET THE STATUS:

```
    // Setting up m_stat
    always @(*) begin
        if (dmem_error == 1) begin
            m_stat <= 3;
        end
        else begin
            m_stat <= M_stat;
        end
    end
endmodule
```

GET valM:

```
    // Reading from data memory
    always @(*) begin
        if (dmem_error == 0 && Mem_read == 1) begin
            m_valM <= Data_Mem[m_addr];
        end
        else begin
            m_valM <= 0;
        end
    end
```

PIPELINE CONTROL LOGIC:



PREDICT HAZARD:

```
// Predicting the Hazards

wire Ret;

wire Miss_Pred;

wire LU_Haz;

wire Exp;


assign Ret = (D_icode == 9 || E_icode == 9 || M_icode == 9) ? 1 : 0;

assign LU_Haz = ((E_icode == 5 || E_icode == 11)
                  && (E_dstM == d_srcA || E_dstM == d_srcB))
                  ? 1 : 0;

assign Miss_Pred = (E_icode == 7 && e_Cnd == 0) ? 1 : 0;
```

GENERATE STALLS AND BUBBLES:

```
// Assigning F_stall according to the Hazards
always @(*) begin
    if (Ret == 1 && LU_Haz == 1) begin
        F_stall <= 1;
    end
    else if (Ret == 1 && Miss_Pred == 1) begin
        F_stall <= 1;
    end
    else if (Ret == 1) begin
        F_stall <= 1;
    end
    else if (LU_Haz == 1) begin
        F_stall <= 1;
    end
    else begin
        F_stall <= 0;
    end
end
```

```
// Assigning D_stall according to the Hazards
always @(*) begin
    if (LU_Haz == 1 && Ret == 1) begin
        D_stall <= 1;
    end
    else if (LU_Haz == 1) begin
        D_stall <= 1;
    end
    else begin
        D_stall <= 0;
    end
end
```

```
    // Assigning D_bubble according to the Hazards
    always @(*) begin
        if (D_stall == 0) begin
            if (Ret == 1 && Miss_Pred == 1) begin
                D_bubble <= 1;
            end
            else if (Ret == 1) begin
                D_bubble <= 1;
            end
            else if (Miss_Pred == 1) begin
                D_bubble <= 1;
            end
            else begin
                D_bubble <= 0;
            end
        end
        else begin
            D_bubble <= 1;
        end
    end
```

```
    // Assigning E_bubble according to the Hazards
    always @(*) begin
        if (LU_Haz == 1 && Ret == 1) begin
            E_bubble <= 1;
        end
        else if (Ret == 1 && Miss_Pred == 1) begin
            E_bubble <= 1;
        end
        else if (LU_Haz == 1) begin
            E_bubble <= 1;
        end
        else if (Miss_Pred == 1) begin
            E_bubble <= 1;
        end
        else begin
            E_bubble <= 0;
        end
    end
```

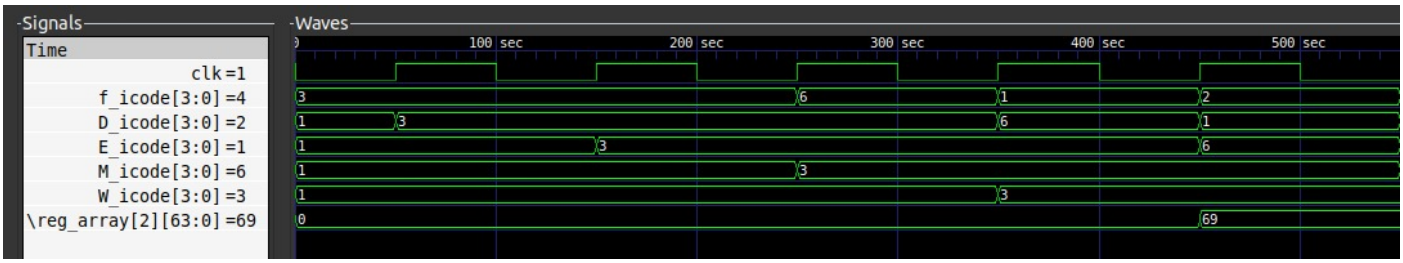
```
    // Assigning M_bubble according to the Hazards
    always @(*) begin

        if (m_stat == 2 || m_stat == 3 || m_stat == 4) begin

            M_bubble <= 1;
        end
        else if (W_stat == 2 || W_stat == 3 || W_stat == 4) begin

            M_bubble <= 1;
        end
        else begin

            M_bubble <= 0;
        end
    end
```

```
    // Assigning W_stall according to the Hazards
    always @(*) begin

        if (W_stat == 2 || W_stat == 3 || W_stat == 4) begin

            W_stall <= 1;
        end
        else begin

            W_stall <= 0;
        end
    end

endmodule
```

TEST CODE AND OUTPUTS:

LINE 1:

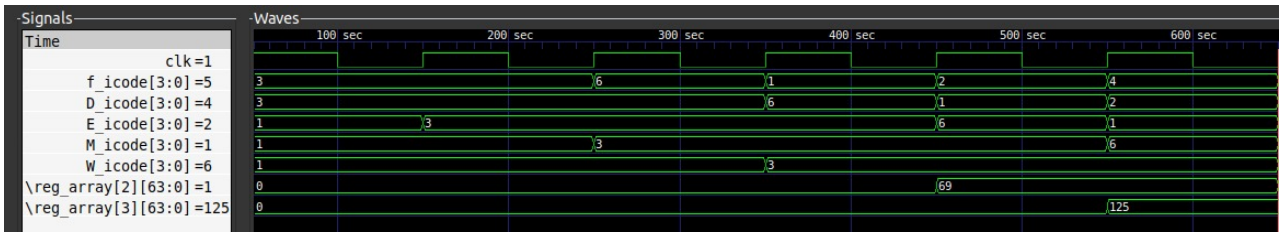
```
00110000 // 0 irmovq
11110010
01000101
00000000
00000000
00000000
00000000
00000000
00000000
00000000
```



1. This irmovq instruction is successfully executed as we can see the value of register 2 updated to the mentioned immediate value after 5 clock cycles.
2. This shows that our PIPE processor is working correctly for instructions that does not use data forwarding and produces hazard.

LINE 2:

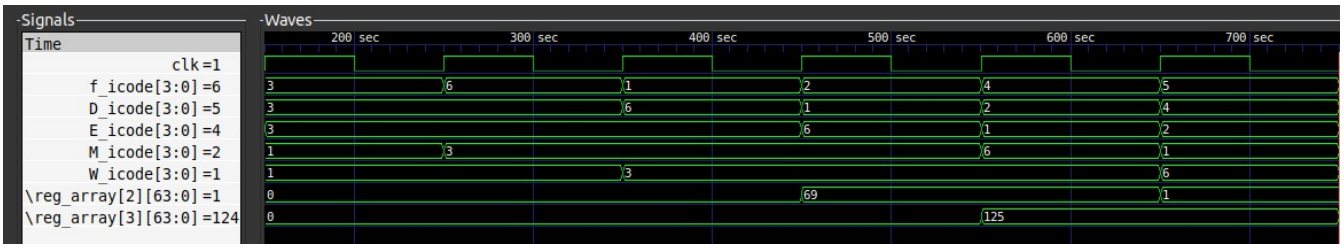
```
00110000 // 10 irmovq

11110011

01111101

00000000

00000000

00000000

00000000

00000000

00000000

00000000
```



1. Here also, we can see successful execution as value of register 3 is getting updated after 5 clock cycles to 125.

LINE 3:

```
00110000 // 20 irmovq

11110010

00000001

00000000

00000000

00000000

00000000

00000000

00000000

00000000
```



1. Here again, we have a successful execution as we can see the register 2 updated from its previous value to 1 after 5 clock cycles.

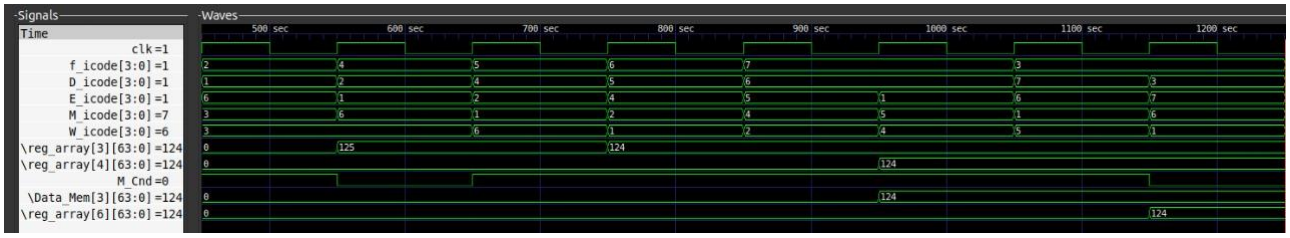
LINE 4 & 5:

```
01100001 // 30 Opq

00100011

00010000 // 32 nop
```

1. Here, we can see successful execution of Opq instruction as the value of register 3 is updated by the result reg_3 – reg_2.
2. As the registers used in this instruction are to be updated by the immediate previous instruction, we have to use data forwarding in this instruction.
3. Successful execution shows that data forwarding is working as expected.

LINE 6, 7, 8:

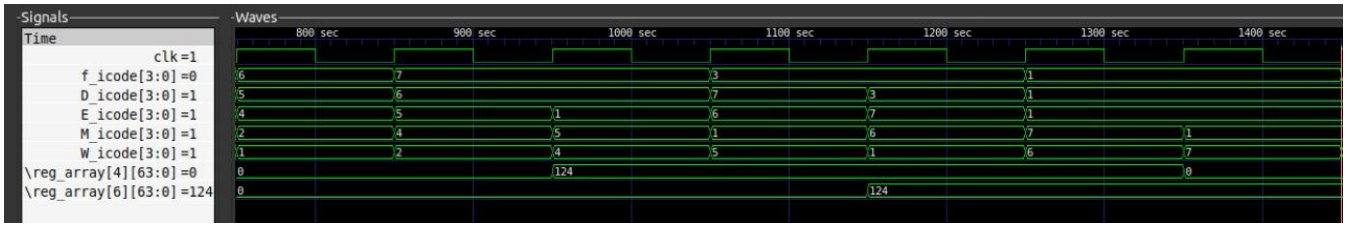
```
00100100 // 33 cmov uneq #taken
00110100
01000000 // 35 rmmovq
01000010
00000010
00000000
00000000
00000000
00000000
00000000
00000000
00000000
01010000 // 45 mrmovq
01100010
00000010
00000000
00000000
00000000
00000000
00000000
00000000
00000000
```



1. These instructions are successfully updated as we can see, register 4 been updated after the cmov instruction, data memory 3 being updated and register 6 being updated to correct values.
2. This also shows correct functioning of data forwarding.

LINE 9:

```
01100001 // 55 Opq

01100100
```



1. This Opq instruction uses register values that are to be updated by the immediate previous instruction by a memory value (mrmovq). This gives rise to a Load-Use Hazard.
2. This hazard is successfully bypassed by using stall in F and D registers in the 3rd clock cycle.
3. This proves that our pipeline control logic is working correctly.

LINE 10 & so on …

```
01110100 // 57 cJmp uneq #not taken
00000000
00000000
00000000
00000000
00000000
00000000
00000000
00000000
00010000 // 66
00010000 // 67
00000000 // 68
00000000 // 69
```

1. This is an mis-predicted jump instruction.
2. We can see that the processor fetches two extra instructions by assuming that the jump will be taken but realizes in later stage and corrects itself.
3. This is a mis-predicted jump hazard, which is correctly bypassed. This shows our pipeline control logic is working correctly.