Pipelined Design:

By - Satla Shashank (2021122003)

Shubham Priyadarshan(2020102023)

We have implemented a 5 stage-pipelined architecture to design processor for Y-86 instruction set.

The stages are named as Fetch, Decode, Execute, Memory and Writeback.

These stages are separated by pipeline registers.

These Registers get updated at each the positive edge of clock

Instructions are fed into the pipeline starting from the fetch stage and whenever its execution is completed and in the positive edge of next clock cycle next instruction in sequence is fed into the pipeline i.e the first instruction which is in fetch stage will go to decode stage and the 2nd instruction will go into fetch stage. And so on till the entire instruction's executions are completed.

We have used the sign convention for different variables in different stages as follows:

1)Values in Registers are prefixed with upper case stage name.

2)Values in the modules are prefixed with smaller case stage name.

Pipeline Registers:

Fetch register: It Stores the predicted value of PC i.e F_pred_PC.

It takes in the the predicted PC value and stores it as F_predPC which is fed as an input to the fetch block.


```
module F(clk , pred_PC , F_predPC);
input clk;
input [63:0] pred_PC;
output reg[63:0] F_predPC;

always@(posedge clk)
begin
   F_predPC <= pred_PC;
end
endmodule
```

Decode register:

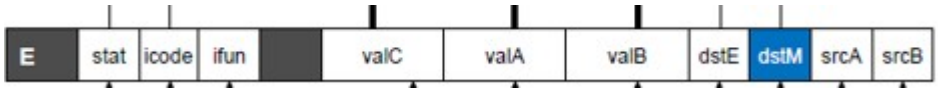
It Store the values of D_iCode, D_iFun , D_stat , D_rA , D_rB , D_valC , D_valP.These are the values of icode,ifun,stat.rA.rB,valC,valP signals of the instruction present in decode stage, which in turn have been used by the decode stage.

```
module D(clk ,f_iCode, f_iFun, f_stat, f_rA, f_rB , f_valC, f_valP, D_iCode , D_iFun , D_stat , D_rA , D_rB , D_valC , D_valP);

input clk;

input[3:0] f_iCode,f_iFun,f_rA,f_rB;

input[2:0] f_stat;

input[63:0] f_valC,f_valP;


output reg[3:0] D_iCode,D_iFun,D_rA,D_rB;

output reg[2:0] D_stat;

output reg[63:0] D_valC,D_valP;


always @(posedge clk ) begin

   D_iCode <= f_iCode;

   D_iFun <= f_iFun;

   D_stat <= f_stat;

   D_rA <= f_rA;

   D_rA <= f_rB;

   D_valC <= f_valC;

   D_valP <= f_valP;


end


endmodule
```

Execute register:

It stores the values of E_stat,E_icode,E_ifun,E_valC,E_valA,E_valB,E_rA,E_rB,E_valP.These are the values of stat,icode,ifun,valC,valA,valB,rA,rB,valP signals of the instruction present in execute stage.

The code implementation was simple as we were to pass all the outputs including the previous stage register passed on value. The values are passed on to the



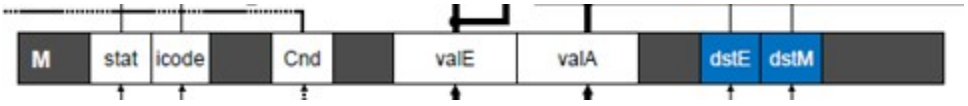

```
module execute_register(
                clk,D_stat,D_icode,D_ifun,D_valP,D_valC,d_valA,d_valB,d_rA,d_rB,
                E_stat,E_icode,E_ifun,E_valC,E_valA,E_valB,E_rA,E_rB,E_valP
                );
    input clk;
    input [2:0] D_stat;
    input [3:0]  D_icode;
    input [3:0] D_ifun;
    input [63:0] D_valC,D_valP;
    input [63:0] d_valA;
    input [63:0] d_valB;
    input [3:0] d_rA;
    input [3:0] d_rB;
```

```
    output reg[2:0] E_stat;
    output reg[3:0]  E_icode;
    output reg[3:0] E_ifun;
    output reg[63:0] E_valC,E_valP;
    output reg[63:0] E_valA;
    output reg[63:0] E_valB;
    output reg[3:0] E_rA;
    output reg[3:0] E_rB;

  always @(posedge clk)
      begin
        E_stat<=D_stat;
        E_icode<=D_icode;
        E_ifun<=D_ifun;
        E_valC<=D_valC;
        E_valA<=d_valA;
        E_valB<=d_valB;
        E_rB<=d_rB;
        E_rA<=d_rA;
        E_valP<=D_valP;
      end
endmodule
```

Memory register:

It stores the values of M_stat,M_icode,M_Cnd,M_valE,M_valA,M_rB,M_rA,M_valP,

M_valC,M_valB .These are the values of sta,icode,Cnd,valE,valA,rB,Ra,valP,valC,valB signals of instruction present in memory stage.



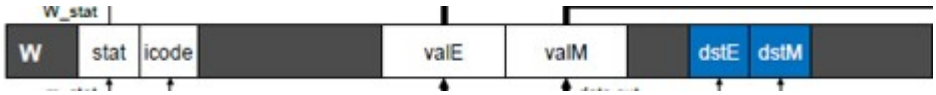

```
module memory_register(
   clk,E_stat,E_icode,e_Cnd,e_valE,E_valP,E_valA,E_valB,E_rA,E_rB,E_valC,
   M_stat,M_icode,M_Cnd,M_valE,M_valA,M_rB,M_rA,M_valP,M_valC,M_valB);


   input clk;
   input [2:0] E_stat;
   input [3:0] E_icode;
   input  e_Cnd;
   input [63:0] e_valE,E_valP,E_valB,E_valC;
   input [63:0] E_valA;
   input [3:0] E_rA;
   input [3:0] E_rB;


   output reg [2:0] M_stat;
   output reg [3:0] M_icode;
   output reg  M_Cnd;
   output reg [63:0] M_valE;
   output reg [63:0] M_valA,M_valP,M_valC,M_valB;
   output reg [3:0] M_rA;
   output reg [3:0] M_rB;
```

```
  always @(posedge clk)
    begin
      M_stat<=E_stat;
      M_icode<=E_icode;
      M_Cnd<=e_Cnd;
      M_valE<=e_valE;
      M_valA<=E_valA;
      M_rB<=E_rB;
      M_rA<=E_rA;
      M_valB<=E_valB;
      M_valC<=E_valC;
      M_valP<=E_valP;
    end
endmodule
```

Writeback register:

It stores the values of W_Cnd,M_valE,m_valM,W_stat,W_iCode,W_valE,W_valM,W_rB,

W_rA,W_valC,W_valA,W_valB,W_valP.These are the values of signals of instruction present in writeback stage.



The signals in these registers are used by combinational logic (fetch,decode,execute,memory,writeback) of corresponding registers (fetch_register,decode_register,execute_register,memory_register,writeback_register) to execute the instruction so that there is no dependency on older stages for a instruction which is moved ahead in the pipeline.

```
module W(clk,m_stat,M_iCode,M_rA,M_rB,M_valC,M_valP,M_valA,M_valB,M_Cnd,W_Cnd,M_valE,m_valM,W_stat,W_iCode,W_valE,W_valM,W_rB,W_rA,W_valC,W_valA,W_valB,W_valP);

input clk,M_Cnd;

input[2:0] m_stat;

input [3:0] M_iCode,M_rA,M_rB;

input [63:0] M_valE,m_valM,M_valC,M_valP,M_valA,M_valB;


output reg[2:0] W_stat;

output reg[3:0] W_iCode,W_rA,W_rB;

output reg[63:0] W_valE,W_valM,W_valA,W_valB,W_valC,W_valP;

output reg W_Cnd;


always @(posedge clk ) begin
   W_stat <= m_stat;
   W_iCode <= M_iCode;
   W_valE<=M_valE;
   W_valM <= m_valM;
   W_valC<=M_valC;
   W_valP<=M_valP;
   W_Cnd<=M_Cnd;
```

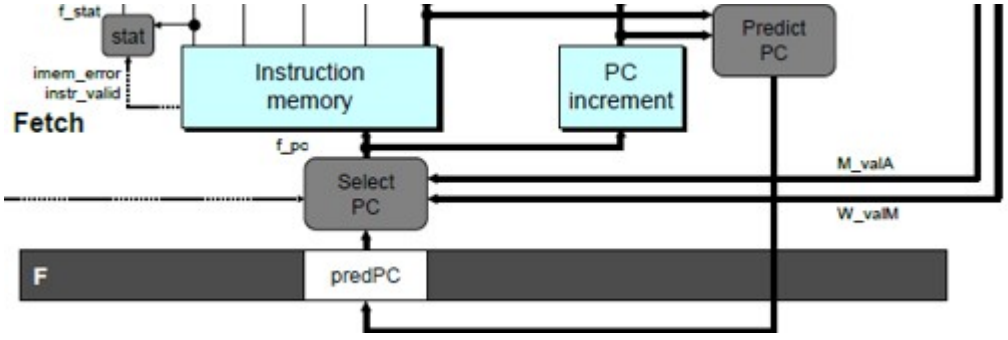
```
   W_valA<=M_valA;
   W_valB<=M_valB;
   W_rB <= M_rB;
   W_rA <= M_rA;

end



endmodule
```

Combinational Logics:

Fetch

It selects the current program counter (PC) and also reads the instruction and computes f_stat,f_icode,f_ifun,rA,rB,f_valC,f_valP, f_instr_Validity,f_imem_error,f_HF signals.

The code is same as the sequential one:



```
module fetch(clk, PC, rA , rB , iCode , iFun ,valC,valP,instr_Validity, imem_error,HF);


// PORT DECLARATION

parameter N = 511;

input [63:0] PC;

input clk;

output reg[3:0] iCode,iFun,rA,rB; // instruction

output reg[63:0] valC,valP;  // values to be assigned for op and PC update

output reg instr_Validity,imem_error,HF; // Validity and memory overflow flag and halt flag 'HF' to distinguish between

                                        //                                  HALT and NOop.


reg [7:0] InstructionMemory [0:N]; // Can take 511 single byte instructions at a time.


reg [0:79]Instruction;


initial begin

   // //   //mrmovq $0x10, %rdx

  InstructionMemory[10]=8'b01000000; //4 0   PC = 10

  InstructionMemory[11]=8'b00000010; //rA=0 rB=2

  InstructionMemory[12]=8'b00000000;

  InstructionMemory[13]=8'b00000000;

  InstructionMemory[14]=8'b00000000;

  InstructionMemory[15]=8'b00000000;

  InstructionMemory[16]=8'b00000000;

  InstructionMemory[17]=8'b00000000;

  InstructionMemory[18]=8'b00000000;

  InstructionMemory[19]=8'b00010000; //V=16


  // OPq addq

  InstructionMemory[20]=8'b01100000; // 6 0  (add)   // PC = 20

  InstructionMemory[21]=8'b00010010; // 1 2  %rcx %rdx
```

```
//jxx
  //InstructionMemory[20]=8'b01110000; //7 0   (jmp)  // PC  = 20;
  //InstructionMemory[21]=8'b00000000;
  //InstructionMemory[22]=8'b00000000;
  //InstructionMemory[23]=8'b00000000;
  //InstructionMemory[24]=8'b00000000;
  //InstructionMemory[25]=8'b00000000;
  //InstructionMemory[26]=8'b00000000;
  //InstructionMemory[27]=8'b00000000;
  //InstructionMemory[28]=8'b00010000; //dest = 16

  //call
  InstructionMemory[31]=8'b10000000; // 80 (call) // PC =31
  InstructionMemory[32]=8'b00000000;
  InstructionMemory[33]=8'b00000000;
  InstructionMemory[34]=8'b00000000;
  InstructionMemory[35]=8'b00000000;
  InstructionMemory[36]=8'b00000000;
  InstructionMemory[37]=8'b00000000;
  InstructionMemory[38]=8'b00000000;
  InstructionMemory[39]=8'b01000000;  // dest

  //rrmov
  //InstructionMemory[20]=8'b00100000; // 2 0  (rrmov)   // PC = 40
  //InstructionMemory[21]=8'b00010011; // 1 3  %rcx %rdx

  //irmovq
  InstructionMemory[50]=8'b00110000; //3 0   PC = 50
  InstructionMemory[51]=8'b11110011; //rA=15 rB=3
  InstructionMemory[52]=8'b00000000;
  InstructionMemory[53]=8'b00000000;
  InstructionMemory[54]=8'b00000000;
  InstructionMemory[55]=8'b00000000;
  InstructionMemory[56]=8'b00000000;
  InstructionMemory[57]=8'b00000000;
  InstructionMemory[58]=8'b00000000;
  InstructionMemory[59]=8'b00000010; // V = 2
```

```
//irmovq
  InstructionMemory[60]=8'b00110000; //3 0   PC = 60
  InstructionMemory[61]=8'b11110100; //rA=15 rB=4
  InstructionMemory[62]=8'b00000000;
  InstructionMemory[63]=8'b00000000;
  InstructionMemory[64]=8'b00000000;
  InstructionMemory[65]=8'b00000000;
  InstructionMemory[66]=8'b00000000;
  InstructionMemory[67]=8'b00000000;
  InstructionMemory[68]=8'b00000000;
  InstructionMemory[69]=8'b00000010; // V = 2

  //rmmov
   //InstructionMemory[60]=8'b01000000; //4 0   PC = 60
  //InstructionMemory[61]=8'b00110101; //rA=3 rB=5
  //InstructionMemory[62]=8'b00000000;
  //InstructionMemory[63]=8'b00000000;
  //InstructionMemory[64]=8'b00000000;
  //InstructionMemory[65]=8'b00000000;
  //InstructionMemory[66]=8'b00000000;
  //InstructionMemory[67]=8'b00000000;
  //InstructionMemory[68]=8'b00000000;
  //InstructionMemory[69]=8'b00000000; // D = 0

  //rmmov
  InstructionMemory[70]=8'b01000000; //4 0   PC = 70
  InstructionMemory[71]=8'b01000011; //rA=4 rB=3
  InstructionMemory[72]=8'b00000000;
  InstructionMemory[73]=8'b00000000;
  InstructionMemory[74]=8'b00000000;
  InstructionMemory[75]=8'b00000000;
  InstructionMemory[76]=8'b00000000;
  InstructionMemory[77]=8'b00000000;
  InstructionMemory[78]=8'b00000000;
  InstructionMemory[79]=8'b00000000; // D = 0
end
```

```
always@(*)
 begin
            Instruction = {InstructionMemory[PC],InstructionMemory[PC+1],
                        InstructionMemory[PC+2],InstructionMemory[PC+3],
                        InstructionMemory[PC+4],InstructionMemory[PC+5],
                        InstructionMemory[PC+6],InstructionMemory[PC+7],
                        InstructionMemory[PC+8],InstructionMemory[PC+9]
                        }; // Concatenating the Instruction memory to make a single 10 byte instruction.
 instr_Validity = 1'b1;
    if(PC>N)
    begin
       imem_error =1;  // Checking for Memory Overflow; assign 1 if TRUE;
    end
    else
    begin
         imem_error =0;  // Assign 0 if FALSE
    end
    iCode = Instruction[0:3];
    iFun  = Instruction[4:7];
    HF = 1'b0;
    if((iCode==4'b0000) & (iFun==4'b0000) ) // Halt ;
    begin
       valP = PC+64'd1;
       HF = 1'b1;
    end
    else if ((iCode==4'b0001)&(iFun==4'b0000)) begin
       valP = PC+64'd1;
    end


    else if((iCode==4'b0010)&(iFun<4'b0111)) // ccmovxx
    begin
       rA = Instruction[8:11];
       rB = Instruction[12:15];
       valP = PC + 64'd2;
    end
    else if ((iCode==4'b0011)&(iFun==4'b0000))//irmovq
    begin
```

```
rA = Instruction[8:11];
        rB = Instruction[12:15];
        valC = Instruction[16:79];          // Stores immediate value to be moved
        valP = PC+64'd10;

    end

    else if((iCode==4'b0100)&(iFun==4'b0000)) //rmmovq
      begin
        rA = Instruction[8:11];
        rB = Instruction[12:15];
        valC = Instruction[16:79];          // Stores Displacement value
        valP = PC+64'd10;
      end

    else if ((iCode==4'b0101)&(iFun==4'b0000)) //mrmovq
    begin
        rA = Instruction[8:11];
        rB = Instruction[12:15];
        valC = Instruction[16:79];          // Stores Displacement value
        valP = PC+ 64'd10;
    end

    else if ((iCode==4'b0110) & (iFun<4'b0100))        //OPq
    begin
        if ((iFun==4'b0000))
        begin                                   //addq
            rA = Instruction[8:11];
            rB = Instruction[12:15];
            valP = PC + 64'd2;
        end

        else if (iFun==4'b0001)
        begin                                   //subq
            rA = Instruction[8:11];
            rB = Instruction[12:15];
            valP = PC + 64'd2;
```

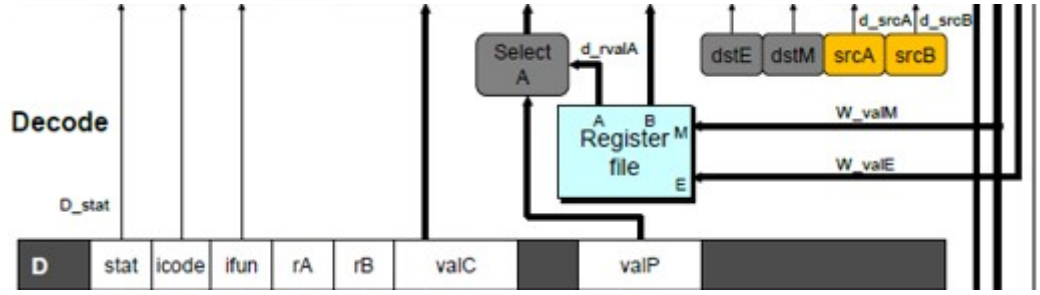
```
end


    else if (iFun==4'b0010)
    begin                           //andq
       rA = Instruction[8:11];
       rB = Instruction[12:15];
       valP = PC + 64'd2;
    end


    else if (iFun==4'b0011)
    begin                           //xorq
       rA = Instruction[8:11];
       rB = Instruction[12:15];
       valP = PC + 64'd2;
    end
  end

 else if ((iCode==4'b0111) & (iFun<4'b0111))        // jxx  Instruction size is of 9 bytes for jump intsruction.
  begin


       valC = Instruction[8:71];  // stores destination
       valP = PC + 64'd9;
  end


  else if ((iCode==4'b1000)&(iFun==4'b0000))       // call
  begin


       valC = Instruction[8:71];
       valP = PC + 64'd9;
  end


  else if((iCode==4'b1001)&(iFun==4'b0000))       //return (ret)
  begin
       valP = PC + 64'd1;
  end
```

```
else if((iCode==4'b1010)&(iFun==4'b0000))        //pushq
    begin
        rA = Instruction[8:11];
        rB = Instruction[12:15];
        valP = PC + 64'd2;
    end

    else if((iCode==4'b1011)&(iFun==4'b0000))        //popq
    begin
        rA = Instruction[8:11];
        rB = Instruction[12:15];
        valP = PC + 64'd2;
    end

    else
    begin
        instr_Validity = 1'b0;                  // If any other combination is entered its invalid;
    end
 end

endmodule
```

Decode Writeback:

It extracts the valA and valB after taking rA, rB as input. Also the decode stage plays an important role in data forwarding which is integral for tackling data dependencies. Depending upon the data dependencies it decides whether to take data from subsequent pipelined stages or simply read from the register file.

Similarly write back updates or writes back the result into the register whether be it valE or valM.



```
module
dec_wb(clk,W_iCode,D_iCode,D_rA,D_rB,W_rA,W_rB,D_valA,D_valB,D_Cnd,W_Cnd,W_valE,W_valM,memreg0,
memreg1,memreg2,memreg3,memreg4,memreg5,memreg6,

        memreg7,memreg8,
memreg9,memreg10,memreg11,memreg12,memreg13,memreg14,e_dstE,M_rA,M_rB,e_valE,m_valM,M_valE,
D_valP);


input clk,W_Cnd,D_Cnd;

input[3:0] D_iCode,D_rA,D_rB,W_iCode,W_rA,W_rB,e_dstE,M_rA,M_rB;

input[63:0] W_valE,W_valM,e_valE,m_valM,M_valE,D_valP;

output reg[63:0] D_valA,D_valB;


output reg[63:0] memreg0,memreg1,memreg2,memreg3,memreg4,memreg5,memreg6,memreg7,memreg8,

          memreg9,memreg10,memreg11,memreg12,memreg13,memreg14;


reg[63:0] memreg[0:14];


integer i;


initial begin

memreg[0] = 64'd0;


for(i=1;i<15;i=i+1)

begin

memreg[i] = memreg[i-1]+64'd1;

end

end



// Decode stage

always@(*)

begin

// for valA

if(D_iCode == 4'b1000 || D_iCode == 4'b0111)  // Data Forwarding

begin

D_valA = D_valP;

D_valB = D_valP;

end
```

```
else if(D_rA==e_dstE)
begin
D_valA = e_valE;
end

else if(D_rA == M_rA)
begin
D_valA = m_valM;
end

else if(D_rA == M_rB)
begin
D_valA = M_valE;
end

else if(D_rA == W_rA)
begin
D_valA = W_valM;
end

else if(D_rA==W_rA)
begin
D_valA = W_valE;
end

// for valB

else if(D_rB==e_dstE)
begin
D_valB = e_valE;
end
```

```
else if(D_rB == M_rA)
begin
D_valB = m_valM;
end

else if(D_rB == M_rB)
begin
D_valB = M_valE;
end

else if(D_rB == W_rA)
begin
D_valB = W_valM;
end

else if(D_rB==W_rA)
begin
D_valB = W_valE;
end



else begin
if(D_iCode==4'b0010)
begin
D_valA = memreg[D_rA];
end

else if(D_iCode==4'b0100) //rmmovq
begin
D_valA<=memreg[D_rA];
D_valB<=memreg[D_rB];
end
```

```
else if(D_iCode==4'b0101) //mrmovq
begin
D_valB<=memreg[D_rB];
end

else if(D_iCode==4'b0110) //OPq
begin
D_valA<=memreg[D_rA];
D_valB<=memreg[D_rB];
end

else if(D_iCode==4'b1000) //call
begin
D_valB<=memreg[4]; //rsp
end

else if((D_iCode==4'b1001) || (D_iCode==4'b1011) ) //ret or popq
begin
D_valA<=memreg[4]; //rsp
D_valB<=memreg[4]; //rsp
end

else if(D_iCode==4'b1010) //pushq
begin
D_valA<=memreg[D_rA];
D_valB<=memreg[4]; //rsp
end
end
```

```
else if((W_iCode>4'b0111) & (W_iCode < 4'b1011))  // for call ret and pushq
begin
memreg[4] <= W_valE;
end

else if(W_iCode == 4'b1011)  // for popq
begin
memreg[4] <= W_valE;
memreg[W_rA] <=W_valM;
end

memreg0 <= memreg[0];
memreg1 <= memreg[1];
memreg2 <= memreg[2];
memreg3 <= memreg[3];
memreg4 <= memreg[4];
memreg5 <= memreg[5];
memreg6 <= memreg[6];
memreg7 <= memreg[7];
memreg8 <= memreg[8];
memreg9 <= memreg[9];
memreg10 <= memreg[10];
memreg11 <= memreg[11];
memreg12 <= memreg[12];
memreg13 <= memreg[13];
memreg14 <= memreg[14];

end
endmodule
```

Execute:

This stage computes e_valE and condition signal e_Cnd using arithmetic logic unit (ALU). This block is almost the same as the sequential one.

```
module execute(clk,icode,ifun,valA,valB,valC,cnd,e_dstE,valE,cc,rA,rB);
//inputs Declaration
input  clk;
input [3:0] icode;
input [3:0] ifun,rA,rB;
input  signed [63:0] valA;
input  signed [63:0] valB;
input signed [63:0] valC;

//Outputs Declaration
output reg signed  [63:0] valE;
output reg cnd;
output reg [2:0] cc;
output reg [3:0] e_dstE;

reg [3:0] icd;
reg signed [63:0] x;
reg signed [63:0] y;
reg [1:0]  control;
wire signed [63:0] z;
wire zf;
wire sf;
wire of;
```

```
reg not_input;
reg or_input1;
reg or_input2;
reg xor_input1;
reg xor_input2;

wire not_output;
wire or_output;
wire xor_output;



ALU_final alu(icd,x,y,control,z,zf,sf,of);
always @(z)
   valE=z;
always @(zf,sf,of)
   begin
      cc[0]=zf;
      cc[1]=sf;
      cc[2]=of;
   end

not notgate(not_output,not_input);
or  orgate(or_output,or_input1,or_input2);
xor xor_gate(xor_output,xor_input1,xor_input2);
```

```
always @(*)
   begin
      if(clk==1)
         begin
         cnd=0;
         //cmovxx
         if(icode==4'b0010)
            begin
               x=valA;
               y=valB;
               icd=icode;
               control=2'b00;
               //rrmovq
               if(ifun==4'b0000)
               begin
                  cnd=1;
               end
               //cmovle
               else if(ifun==4'b0001)
                  begin
                     // (sf^of)||zf
                     xor_input1=cc[1];
                     xor_input2=cc[2];
                     or_input1=xor_output;
                     or_input2=cc[0];
                     if(or_output)
                        begin
                           cnd=1;
                        end
                  end
```

```
//cmovl
                else if(ifun==4'b0010)
                   begin
                      // sf^of
                      xor_input1=cc[1];
                      xor_input2=cc[2];
                      if(xor_output)
                         begin
                            cnd=1;
                         end
                   end
                //cmove
                else if(ifun==4'b0011)
                   begin
                      // zf
                      if(cc[0])
                         begin
                            cnd=1;
                         end
                   end
                //cmovne
                else if(ifun==4'b0100)
                   begin
                      // !zf
                      not_input=cc[0];
                      if(not_output)
                         begin
                            cnd=1;
                         end
                   end
```

```
//cmovge
                else if(ifun==4'b0101)
                   begin
                      // !(sf^of)
                      xor_input1=cc[1];
                      xor_input2=cc[2];
                      not_input=xor_output;
                      if(not_output)
                         begin
                            cnd=1;
                         end
                   end
                //cmovg
                else if(ifun==4'b0110)
                   begin
                      //!(sf^of)) && (!zf)
                      xor_input1=cc[1];
                      xor_input2=cc[2];
                      not_input=xor_output;
                      if(not_output)
                         begin
                            not_input=cc[0];
                            if(not_output)
                               begin
                                  cnd=1;
                               end
                         end
                   end
             end
          //irmovq
          else if(icode==4'b0011)
             begin
                x=valC;
                y=64'b0;
                icd=icode;
                control=2'b00;
             end
```

```
//rmmovq
        else if(icode==4'b0100)
           begin
           icd=icode;
              x=valC;
              y=valB;
              control=2'b00;


           end
        //mrmovq
        else if(icode==4'b0101)
           begin
              icd=icode;
              x=valC;
              y=valB;


           end
        //Arithmatic and logic Operations
        else if(icode==4'b0110)
           begin
              x=valA;
              y=valB;
              icd=icode;
              control=2'b00;
              //add
              if(ifun==4'b0000)
                 begin
                    control=2'b00;
                 end
              //sub
              else if(ifun==4'b0001)
                 begin
                    control=2'b01;
                 end
```

```
//and
              else if(ifun==4'b0010)
                 begin
                    control=2'b10;
                 end
              //xor
              else if(ifun==4'b0011)
                 begin
                    control=2'b11;
                 end


           end
        //call
        else if(icode==4'b1000)
           begin
              x=-64'b1;
              y=valB;
              icd=icode;
              control=2'b00;


           end
        //ret
        else if(icode==4'b1001)
           begin
              x=64'b1;
              y=valB;
              icd=icode;
              control=2'b00;


           end
```

```
//pushq
        else if(icode==4'b1010)
          begin
            x=-64'b1;
            y=valB;
            icd=icode;
            control=2'b00;
     end
        //popq
        else if(icode==4'b1011)
          begin
            x=64'b1;
            y=valB;
            icd=icode;
            control=2'b00;
         end
    //jxx
        else if(icode==4'b0111)
          begin
            //jmp
            if(ifun==4'b0000)
            begin
               cnd=1;
            end
            //jle
            else if(ifun==4'b0001)
               begin
                  // (sf^of)||zf
                  xor_input1=cc[1];
                  xor_input2=cc[2];
                  or_input1=xor_output;
                  or_input2=cc[0];
                  if(or_output)
                     begin
                        cnd=1;
                     end
               end
```

```
//jge
               else if(ifun==4'b0101)
                  begin
                     // !(sf^of)
                     xor_input1=cc[0];
                     xor_input2=cc[1];
                     not_input=xor_output;
                     if(not_output)
                        begin
                           cnd=1;
                        end
                  end
               //jg
               else if(ifun==4'b0110)
                  begin
                     //!(sf^of)) && (!zf)
                     xor_input1=cc[1];
                     xor_input2=cc[2];
                     not_input=xor_output;
                     if(not_output)
                        begin
                           not_input=cc[0];
                           if(not_output)
                              begin
                                 cnd=1;
                              end
                        end
                  end
            end
         end

         e_dstE = rB;

   end
endmodule
```

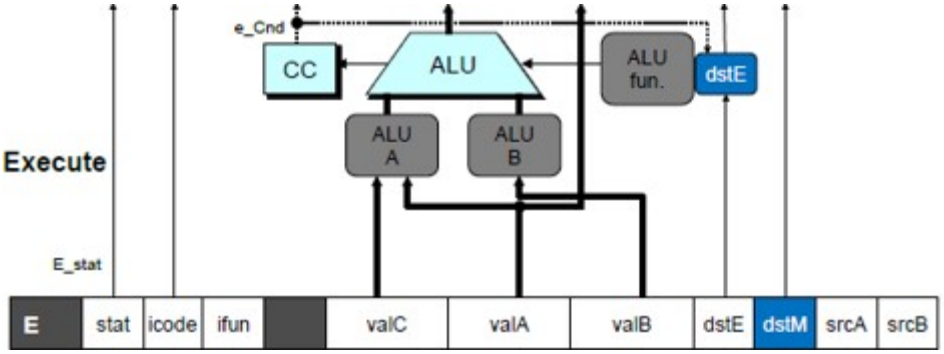
```
//jl
 else if(ifun==4'b0010)
                    begin
// sf^of
                        xor_input1=cc[1];
                        xor_input2=cc[2];
                        if(xor_output)
                           begin
                              cnd=1;
                           end
                     end
                  //je
                  else if(ifun==4'b0011)
                     begin
                        // zf
                        if(cc[0])
                           begin
                              cnd=1;
                           end
                     end
                  //jne
                  else if(ifun==4'b0100)
                     begin
                        // !zf
                        not_input=cc[0];
                        if(not_output)
                           begin
                              cnd=1;
                           end
                     end
```

```
 2'b10:begin
          ansfinal=ans3;
          overflowfinal=1'b0;
        end
      2'b11:begin
          ansfinal=ans4;
          overflowfinal=1'b0;
        end
    endcase
  end
always @(*)
  result= ansfinal;

always @(*)
    begin
       if(icode==4'b0110)
          begin
             if(ansfinal==64'b0)
                begin
                   zf=1;
                end
             else
                begin
                   zf=0;
                end
            if(ansfinal[63]==1'b1)
                begin
                   sf=1;
                end
             else
                begin
                   sf=0;
                end
             OF= overflowfinal;
          end
    end
endmodule
```

```
module XOR(x,y,z);
input [63:0] x,y;
output [63:0] z;

genvar i;

generate
for(i=0;i<64;i = i+1) begin

xor(z[i] , x[i] , y[i]);

end

endgenerate

endmodule

module AND(x,y,z);
input[0:63] x,y;
output[0:63]z;

genvar i;

generate

for(i=0;i<64;i = i+1) begin

and(z[i] , x[i] , y[i]);

end

endgenerate

endmodule
```

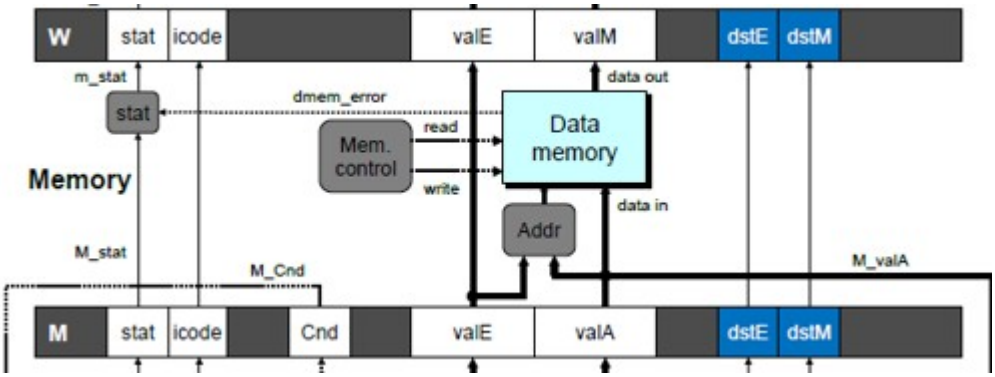
```
module ADD(x,y,sum,OF);
input signed[63:0] x,y;
output signed[63:0] sum;
output signed OF;
wire signed[63:0] ci;

wire carry_in;
assign carry_in = 1'b0;



full_adder FA1(x[0],y[0],carry_in,sum[0],ci[0]);
genvar i;
generate
for(i=1;i<64;i=i+1)
begin
full_adder FA0(x[i] , y[i] , ci[i-1] , sum[i] , ci[i]);
end
endgenerate

xor(OF,ci[62],ci[63]);

endmodule

module full_adder(x,y,carry_in,sum,carry);
  input signed x,y,carry_in;
  output signed sum,carry;
  wire signed w1,w2,w3,w4,w5;
  xor(w1,x,y);
  xor(sum,w1,carry_in);
  and(w2,x,y);
  and(w3,x,carry_in);
  and(w4,y,carry_in);
  or(w5,w2,w3);
  or(carry,w5,w4);
endmodule
```

```
module  SUB(x,y,z,OF);
input signed[63:0] x,y;
output signed[63:0] z;
output OF;

wire signed[63:0] u,v,w;
wire signed[63:0] k;



twocomp comp(y , u);




ADD sub_add(x , u , w , OF);

assign z=w;

endmodule




module twocomp(x ,y);
input signed[63:0] x;
output signed[63:0] y;
wire signed[63:0] w;
wire c_out;
genvar i;
generate
for(i=0 ; i<64 ; i = i+1)
begin
not (w[i] , x[i]);
end
endgenerate
ADD twocomp1(w,64'b1,y,c_out);
endmodule
```



Memory:

Reading or writing to data memory happens in this stage. It computes valM, data_memory_error signals and it gets its input from memory register.



```
module memory(clk,icode,valA,valE,valP,valM,data_memory_error);
input clk;
input [3:0] icode;
input [63:0] valA;
input [63:0] valE;
input [63:0] valP;
output reg [63:0] valM;
output reg data_memory_error;

reg [63:0] data_memory[1023:0];

initial valM=64'b0;

always @(*)
  begin
    data_memory_error=0;
    //rmmovq
    if(icode==4'b0100)
      begin
        if((valE>1023)||(valE<0))
          data_memory_error=1;
        else
          data_memory[valE]=valA;
      end
    //mrmovq
    else if(icode==4'b0101)
      begin
        if((valE>1023)||(valE<0))
          data_memory_error=1;
        else
          valM=data_memory[valE];
      end
```

```
//call
      else if(icode==4'b1000)
         begin
            if((valE>1023)||(valE<0))
               data_memory_error=1;
            else
               data_memory[valE]=valP;
         end
      //ret
      else if(icode==4'b1001)
         begin
            if((valA>1023)||(valA<0))
               data_memory_error=1;
            else
            valM=data_memory[valA];
         end
      //pushq
      else if(icode==4'b1010)
         begin
            if((valE>1023)||(valE<0))
               data_memory_error=1;
            else
            data_memory[valE]=valA;
         end
      //popq
      else if(icode==4'b1011)
         begin
            if((valA>1023)||(valA<0))
               data_memory_error=1;
            else
               valM=data_memory[valA];
         end
   end
endmodule
```

.

Data Forwarding:

We have implemented data forwarding in the processor design.

As In the naive architecture register file is not written until the completion of write back stage and source operands read from register file in the decode stage.

So to avoid stalling for majority of instructions we have passed value directly from generating instruction in execute and memory stage to decode stage.

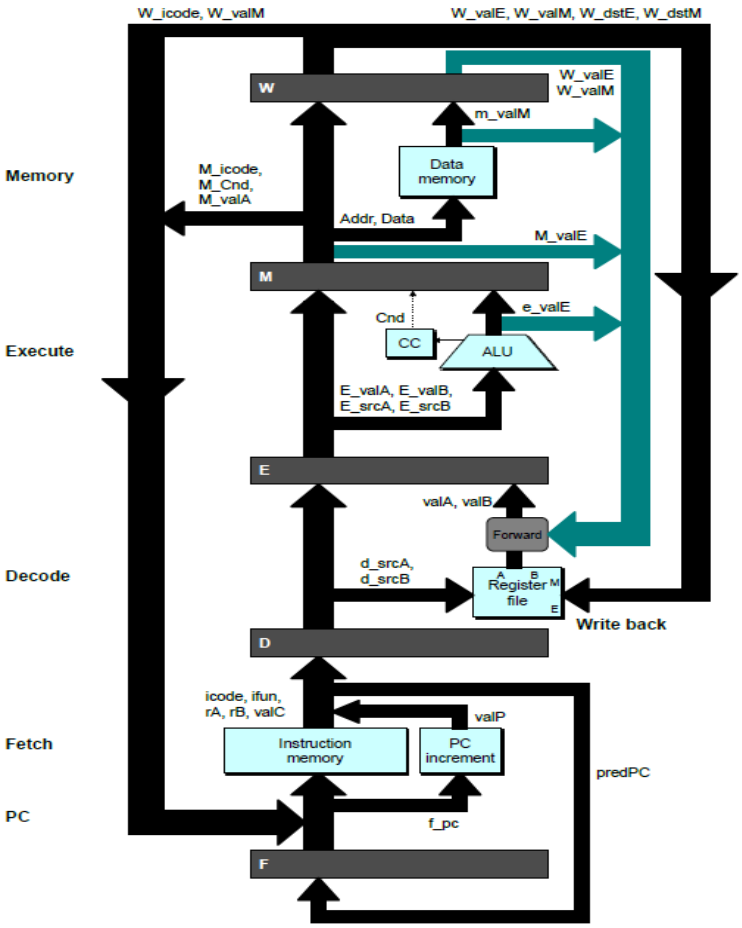
The priority order is followed as we have multiple forwarding sources and created logic blocks to select from multiple sources for valA and valB in decode stage.

Forwarding Sources:

1)Execute: valE

2)Memory: valE, valM

3)Write back: valE, valM

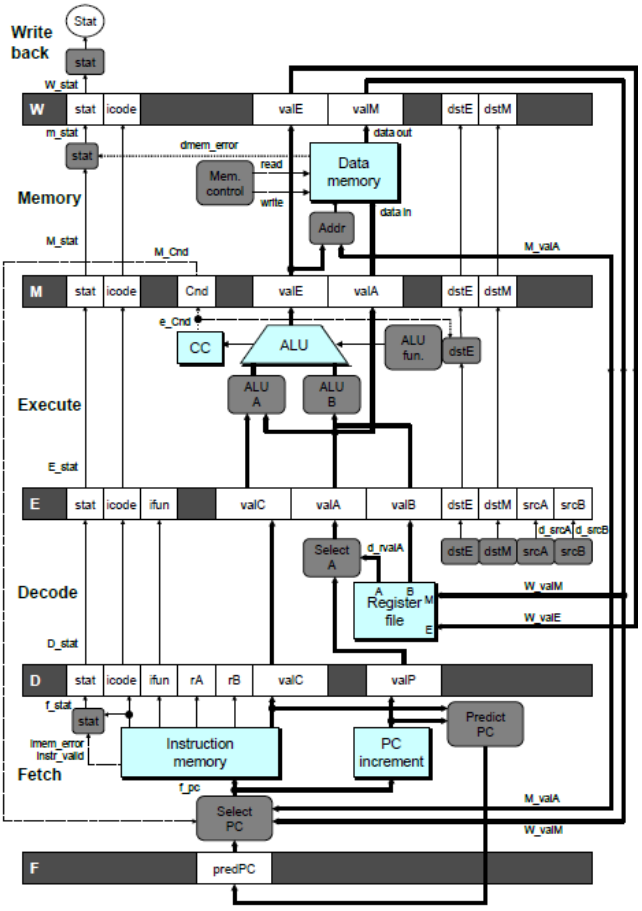



```
// for valA
if(D_iCode == 4'b1000 || D_iCode == 4'b0111)
begin
D_valA = D_valP;
D_valB = D_valP;
end

else if(D_rA==e_dstE)
begin
D_valA = e_valE;
end

else if(D_rA == M_rA)
begin
D_valA = m_valM;
end
```

```
else if(D_rA == M_rB)
begin
D_valA = M_valE;
end

else if(D_rA == W_rA)
begin
D_valA = W_valM;
end

else if(D_rA==W_rA)
begin
D_valA = W_valE;
end
```

Processor:

The final pipelined version of the implemented processor can be mapped as:



The implentation is a Naive five stage pipelined Y-86 processor. Combined with the Data forwarding it can take care of the Data dependency but not the load use hazard.

The final code implementation can be seen in the next document.

```
`timescale 1ns / 1ps

`include "D.v"
`include "execute_register.v"
`include "F.v"
`include "memory_register.v"
`include "W.v"
`include "fetch.v"
`include "execute_new.v"
`include "dec_wb.v"
`include "memory.v"
`include "pc_update.v"

module proc;
  reg clk;

  reg [63:0] PC;

  reg stat[0:2]; //  |AOK|INS|HLT|

  wire [63:0] updated_pc;
  wire [63:0] f_pred_pc;

  wire [2:0]  f_stat,cc;
  wire [3:0]  f_icode;
  wire [3:0]  f_ifun;
```

```
wire [3:0]  f_rA;
wire [3:0]  f_rB,e_dstE;
wire [63:0] f_valC;
wire [63:0] f_valP;
wire        imem_error;
wire        hltins;
wire        instr_valid;

wire [2:0]  d_stat;
wire [3:0]  d_icode;
wire [3:0]  d_ifun;
wire [3:0]  d_rA;
wire [3:0]  d_rB;
wire [63:0] d_valC;
wire [63:0] d_valP;
wire [63:0] d_valA;
wire [63:0] d_valB;

wire [2:0]  e_stat;
wire [3:0]  e_icode;
wire [3:0]  e_ifun;
wire        e_cnd;
wire [3:0]  e_rA;
wire [3:0]  e_rB;
wire [63:0] e_valC;
wire [63:0] e_valP;
wire [63:0] e_valA;
wire [63:0] e_valB;
wire [63:0] e_valE;
```

```
  wire [2:0]  m_stat;
  wire [3:0]  m_icode;
  wire        m_cnd;
  wire [3:0]  m_rA;
  wire [3:0]  m_rB;
  wire [63:0] m_valC;
  wire [63:0] m_valP;
  wire [63:0] m_valA;
  wire [63:0] m_valB;
  wire [63:0] m_valE;
  wire [63:0] m_valM;

  wire [2:0]  w_stat ;
  wire [3:0]  w_icode;
  wire        w_cnd;
  wire [3:0]  w_rA;
  wire [3:0]  w_rB;
  wire [63:0] w_valC;
  wire [63:0] w_valP;
  wire [63:0] w_valA;
  wire [63:0] w_valB;
  wire [63:0] w_valE;
  wire [63:0] w_valM;

  wire [63:0]
memreg0,memreg1,memreg2,memreg3,memreg4,memreg5,memreg6,memreg7,memreg8,
            memreg9,memreg10,memreg11,memreg12,memreg13,memreg14;
```

```
F freg(
  .clk(clk),
  .pred_PC(f_valP),
  .F_predPC(f_pred_pc)
);

D dreg(

  .clk(clk),

  .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),

  .D_stat   (d_stat),
  .D_iCode  (d_icode),
  .D_iFun   (d_ifun),
  .D_rA     (d_rA),
  .D_rB     (d_rB),
  .D_valC   (d_valC),
  .D_valP   (d_valP)

);

execute_register ereg(
```

```
  .clk(clk),

  .D_stat  (d_stat),
  .D_icode (d_icode),
  .D_ifun  (d_ifun),
  .d_rA    (d_rA),
  .d_rB    (d_rB),
  .D_valC  (d_valC),
  .D_valP  (d_valP),
  .d_valA  (d_valA),
  .d_valB  (d_valB),

  .E_stat  (e_stat),
  .E_icode (e_icode),
  .E_ifun  (e_ifun),
  .E_rA    (e_rA),
  .E_rB    (e_rB),
  .E_valC  (e_valC),
  .E_valP  (e_valP),
  .E_valA  (e_valA),
  .E_valB  (e_valB)

);

memory_register mreg(

  .clk(clk),

  .E_stat  (e_stat),
  .E_icode (e_icode),
```

```
  .E_rA    (e_rA),
  .E_rB    (e_rB),
  .E_valC  (e_valC),
  .E_valP  (e_valP),
  .E_valA  (e_valA),
  .E_valB  (e_valB),
  .e_Cnd   (e_cnd),
  .e_valE  (e_valE),

  .M_stat  (m_stat),
  .M_icode (m_icode),
  .M_rA    (m_rA),
  .M_rB    (m_rB),
  .M_valC  (m_valC),
  .M_valP  (m_valP),
  .M_valA  (m_valA),
  .M_valB  (m_valB),
  .M_Cnd   (m_cnd),
  .M_valE  (m_valE)

);

W wreg(

  .clk(clk),

  .m_stat  (m_stat),
  .M_iCode (m_icode),
  .M_rA    (m_rA),
```

```
  .M_rB    (m_rB),
  .M_valC  (m_valC),
  .M_valP  (m_valP),
  .M_valA  (m_valA),
  .M_valB  (m_valB),
  .M_Cnd   (m_cnd),
  .M_valE  (m_valE),
  .m_valM  (m_valM),

  .W_stat  (w_stat),
  .W_iCode (w_icode),
  .W_rA    (w_rA),
  .W_rB    (w_rB),
  .W_valC  (w_valC),
  .W_valP  (w_valP),
  .W_valA  (w_valA),
  .W_valB  (w_valB),
  .W_Cnd   (w_cnd),
  .W_valE  (w_valE),
  .W_valM  (w_valM)
);

pc_update pcup(
  .clk(clk),

  .icode(w_icode),
  .cnd(w_cnd),
  .valC(w_valC),
  .valM(w_valM),
```

```
  .valP(f_pred_pc),
  .PC_updated(updated_pc)
);

fetch fetch(          // fetch
  .clk(clk),
  .PC(PC),
  .rA(f_rA),
  .rB(f_rB),
  .iCode(f_icode),
  .iFun(f_ifun),

  .valC(f_valC),
  .valP(f_valP),
  .instr_Validity(instr_Validity),
  .imem_error(imem_error),
  .HF(hltins)
);

execute execute(
  .clk(clk),
  .icode(e_icode),
  .ifun(e_ifun),
  .valA(e_valA),
  .valB(e_valB),
  .valC(e_valC),
  .valE(e_valE),
  .cc(cc),
  .cnd(e_cnd),
```

```
  .e_dstE(e_dstE)
);

dec_wb decode_wb(

  .clk(clk),

  .D_iCode(d_icode),
  .D_rA(d_rA),
  .D_rB(d_rB),
  .D_Cnd(d_cnd),
  .D_valA(d_valA),
  .D_valB(d_valB),

  .W_iCode(w_icode),
  .W_rA(w_rA),
  .W_rB(w_rB),
  .W_Cnd(w_cnd),
  .W_valE(w_valE),
  .W_valM(w_valM),

  .memreg0(memreg0),
  .memreg1(memreg1),
  .memreg2(memreg2),
  .memreg3(memreg3),
  .memreg4(memreg4),
  .memreg5(memreg5),
  .memreg6(memreg6),
  .memreg7(memreg7),
```

```
  .memreg8(memreg8),
  .memreg9(memreg9),
  .memreg10(memreg10),
  .memreg11(memreg11),
  .memreg12(memreg12),
  .memreg13(memreg13),
  .memreg14(memreg14)


);

memory mem(          // memory
  .clk(clk),
  .icode(m_icode),
  .valA(m_valA),

  .valE(m_valE),
  .valP(m_valP),
  .valM(m_valM),
  .data_memory_error(datamem)
);



initial begin
  $dumpfile("proc.vcd");
  $dumpvars(0,proc);
  stat[0]=1;
  stat[1]=0;
  stat[2]=0;
```

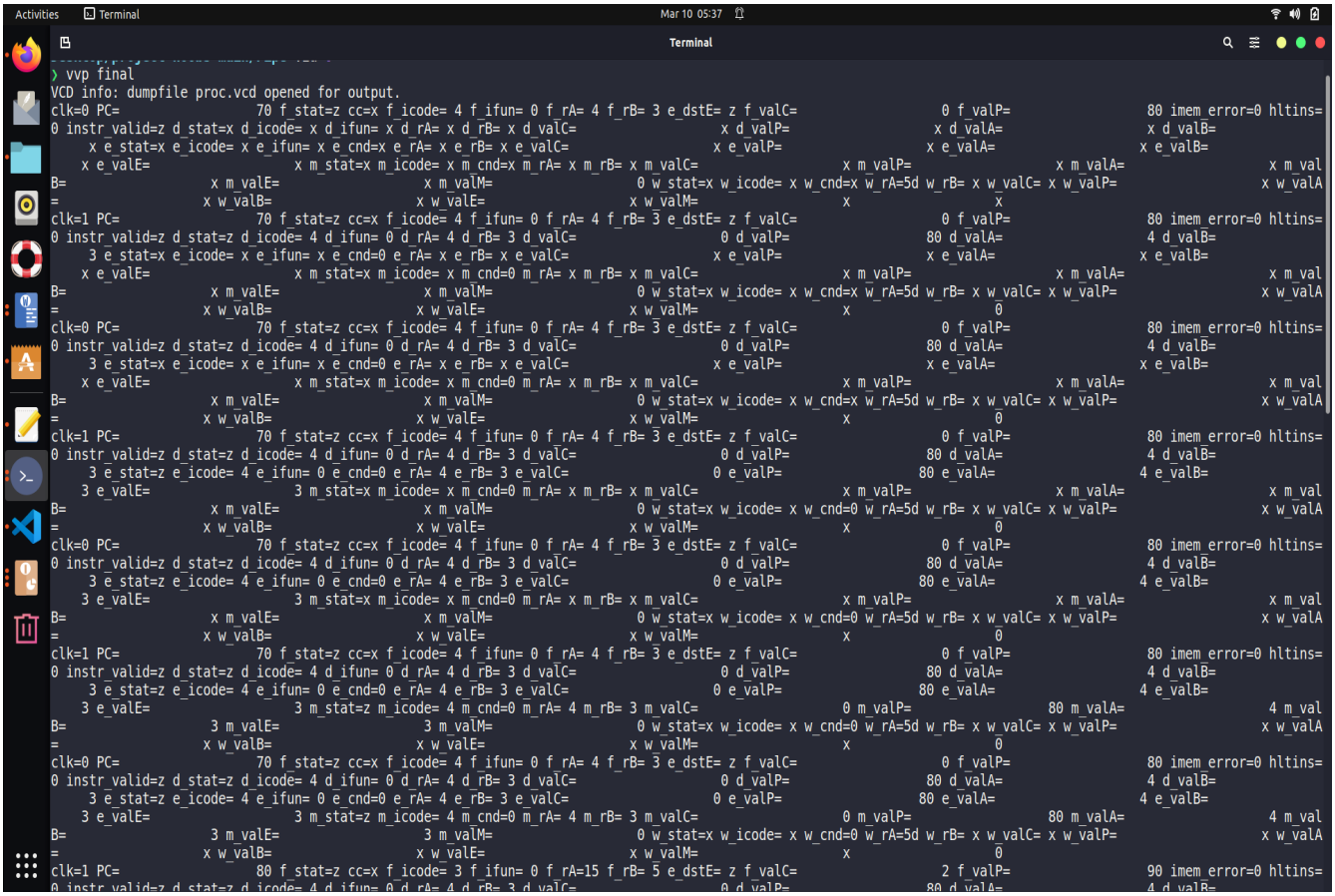
```
  clk=0;
  PC=64'd20;
end
initial begin
 #5 clk=~clk;
  #5 clk=~clk;
   #5 clk=~clk;
    #5 clk=~clk;
     #5 clk=~clk;
      #5 clk=~clk;
       #5 clk=~clk;
        #5 clk=~clk;
         #5 clk=~clk;
          #5 clk=~clk;
           #5 clk=~clk;
            #5 clk=~clk;
             #5 clk=~clk;
              #5 clk=~clk;
               #5 clk=~clk;
                #5 clk=~clk;
                 #5 clk=~clk;
                  #5 clk=~clk;
                   #5 clk=~clk;
                    #5 clk=~clk;
                     #5 clk=~clk;
                      #5 clk=~clk;
                       #5 clk=~clk;
                        #5 clk=~clk;
                         #5 clk=~clk;
                          #5 clk=~clk;
```

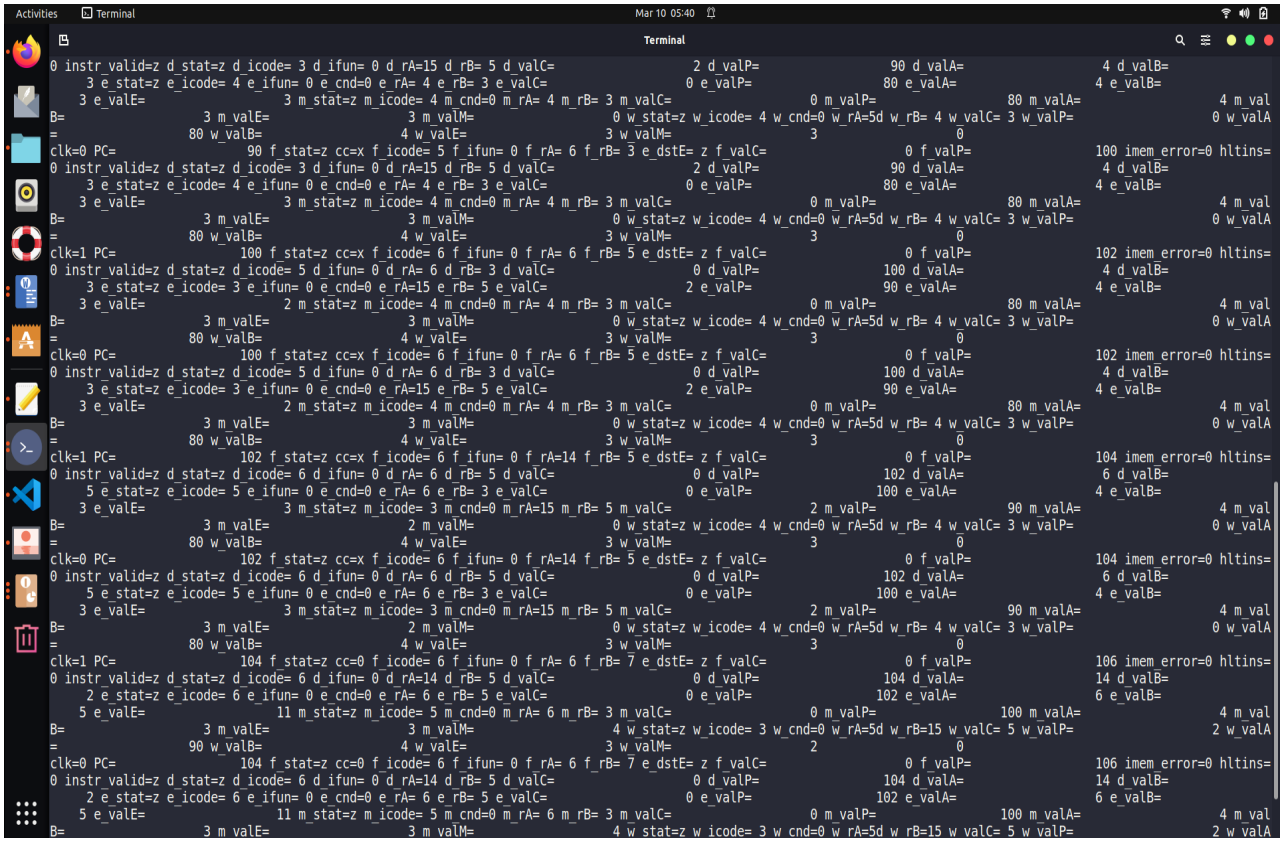
```
                    end


always@(*)
begin
  PC=updated_pc;
end

always@(*)
begin
  if(hltins)
  begin
    stat[2]=hltins;
    stat[1]=1'b0;
    stat[0]=1'b0;
  end
  else if(instr_valid==1'b0)
  begin
    stat[1]=instr_valid;
    stat[2]=1'b0;
    stat[0]=1'b0;
  end
  else
  begin
    stat[0]=1'b1;
    stat[1]=1'b0;
    stat[2]=1'b0;
  end
end
```

```
 always@(*)
 begin
  if(stat[2]==1'b1)
  begin
    $finish;
  end
 end


 initial
   $monitor("clk=%d PC=%d f_stat=%d cc=%d f_icode=%d f_ifun=%d f_rA=%d f_rB=%d
e_dstE=%d f_valC=%d f_valP=%d imem_error=%d hltins=%d instr_valid=%d d_stat=%d
d_icode=%d d_ifun=%d d_rA=%d d_rB=%d d_valC=%d d_valP=%d d_valA=%d d_valB=%d
e_stat=%d e_icode=%d e_ifun=%d e_cnd=%d e_rA=%d e_rB=%d e_valC=%d e_valP=%d
e_valA=%d e_valB=%d e_valE=%d m_stat=%d m_icode=%d m_cnd=%d m_rA=%d m_rB=%d
m_valC=%d m_valP=%d m_valA=%d m_valB=%d m_valE=%d m_valM=%d w_stat=%d
w_icode=%d w_cnd=%d w_rA=5d w_rB=%d w_valC=%d w_valP=%d w_valA=%D w_valB=%d
w_valE=%d w_valM=
%d",clk,PC,f_stat,cc,f_icode,f_ifun,f_rA,f_rB,e_dstE,f_valC,f_valP,imem_error,hltins,instr_va
lid,d_stat,d_icode,d_ifun,d_rA,d_rB,d_valC,d_valP,d_valA,d_valB,e_stat,e_icode,e_ifun,e_c
nd,e_rA,e_rB,e_valC,e_valP,e_valA,e_valB,e_valE ,m_stat,m_icode,m_cnd,m_rA,m_rB,m_va
lC,m_valP,m_valA,m_valB,m_valE,m_valM,
w_stat,w_icode,w_cnd,w_rA,w_rB,w_valC,w_valP,w_valA,w_valB,w_valE,w_valM);

        //$monitor("clk=%d f=%d d=%d e=%d m=%d wb=
%d",clk,f_icode,d_icode,e_icode,m_icode,w_icode);


endmodule
```

The above code was tested for variety of instructions:

for rmmov,irmov,mrmov,add consecutively:


Activities
Terminal
Mar 10 05:37
Terminal
vvp final
VCD info: dumpfile proc.vcd opened for output.



The iCode, iFun, valA, valB, valE, valC values indicate that the processor is working in a fine condition.

This concludes our report.