Intro to Processor Architecture

FINAL PROJECT REPORT :

Notion Link : Intro to Processor Architecture

Jampana Koundinya Varma - 2021112022

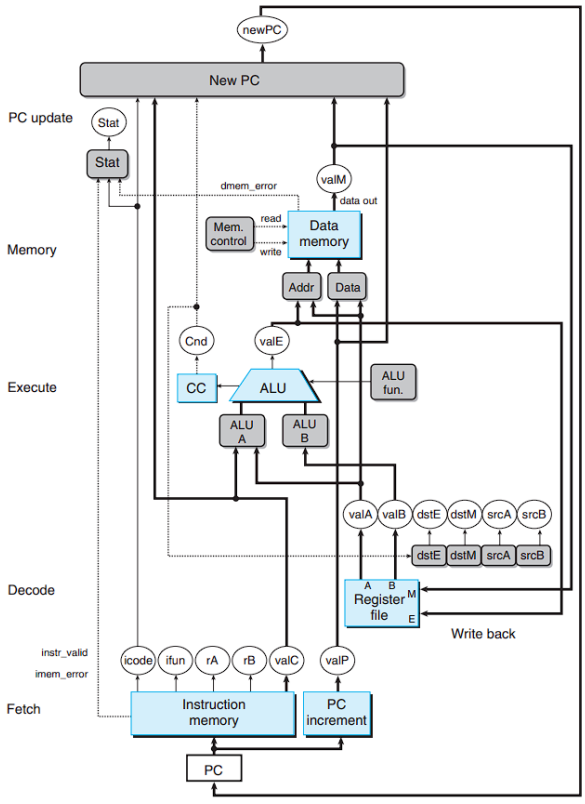
T Ravindra - 2021102025

Sequential implementation :

The sequential implementation of the y86 processor is a basic implementation where instructions are executed one at a time in sequence. Each instruction is fetched from memory, decoded, and executed before moving on to the next instruction.

The sequential implementation of the y86 processor is a simple and efficient approach that can be used to teach the basics of computer architecture and assembly language programming. Its advantages include simplicity, predictability, and efficiency for certain types of applications.

Block Diagram :



Stages :

The Y86 processor is a simple 8-stage pipeline processor with six major blocks: Fetch, Decode, Execute, Memory, Writeback, and PC Update.

In the **Fetch stage**, the processor fetches instructions from memory using the program counter (PC). The instruction is then sent to the **Decode stage** where it is decoded to determine the operation to be performed. In the **Execute stage**, the operation is performed using the operands obtained from the register file or memory.

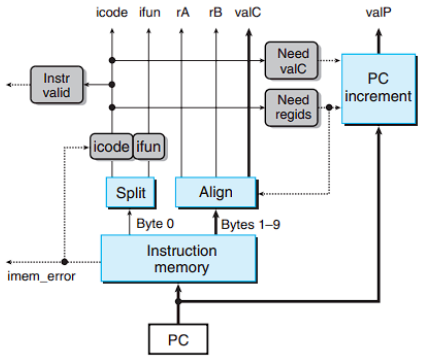
In the **Memory stage**, memory operations such as load and store are performed, and in the **Writeback stage**, the result of the operation is written back to the register file.

In the **PC Update stage**, the PC gets updated so that the next instruction can be performed.

The control signals are generated based on the instruction being executed and the current state of the processor. Overall, the Y86 processor is a simple but effective example of a pipeline processor, and understanding its implementation can be useful for understanding more complex processors.

Now, let us go through the each stage in detail.

Fetch Stage :



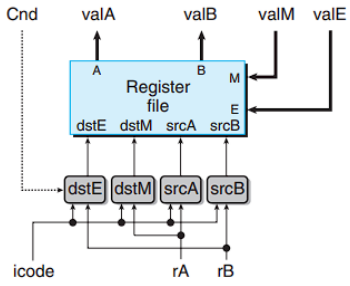
The fetch stage includes the instruction memory hardware unit. This unit reads 10 bytes from memory at a time, using the PC as the address of the first byte (byte 0). This byte is interpreted as the instruction byte and is split (by the unit labeled “Split”) into two 4-bit quantities. The control logic blocks labeled ***“icode”*** and ***“ifun”*** then compute the instruction and function codes as equaling either the values read from memory or, in the event that the instruction address is not valid (as indicated by the signal imem_error), the values corresponding to a nop instruction. Based on the value of *icode*, we can compute three 1-bit signals (shown as dashed lines)

Given below is the code for implementing the fetch stage of the sequential version.

```
module fetch(icode, ifun, rA, rB, valC, valP, imem_error, instr_valid, clk, PC, instr, halt_flag);

    input clk;                              // clock
    input [63:0] PC;                        // Program counter
    input [0:79] instr;                     // Current instruction
    output reg [3:0] icode, ifun;           // icode is the type of instruction out of the 12 types and ifun gives the exact instructi
    output reg [3:0] rA, rB;                // register addresses
    output reg signed [63:0] valC;          // 8 byte values i.e F
    output reg [63:0] valP;                 // incremented PC
    output reg imem_error = 0;              // unaccessible memory error flag
    output reg instr_valid = 1;             // invalid instruction flag
    output reg halt_flag = 0;               // halt flag

    always@(*)
    begin

        if(PC > 255)
        begin
            imem_error = 1;
            $finish;
        end

        {icode, ifun} = instr[0:7];

        case (icode)
        4'h0:begin                          //halt
            valP = PC + 1;
            halt_flag = 1;
        end

        4'h1:valP = PC + 1;                 // nop

        4'h2:begin                          // cmovq
                {rA, rB}=instr[8:15];
                valP = PC + 2;
        end

        4'h3:begin                          // irmovq
```

```
                {rA, rB, valC} = instr[8:79];
                valP = PC + 10;
        end

        4'h4:begin                      // rmmovq
            {rA, rB, valC} = instr[8:79];
            valP = PC + 10;
        end

        4'h5:begin                      // mrmovq
            {rA, rB, valC} = instr[8:79];
            valP = PC + 10;
        end

        4'h6:begin                      // OPq
            {rA, rB} = instr[8:15];
            valP = PC + 2;
        end

        4'h7:begin                      //jxx
            valC = instr[8:71];
            valP = PC + 9;
        end

        4'h8:begin                      //call
            valC = instr[8:71];
            valP = PC + 9;
        end

        4'h9:                           //ret
            valP = PC + 1;

        4'hA:begin                      //pushq
            {rA, rB} = instr[8:15];
            valP = PC + 2;
        end

        4'hB:begin                      //popq
            {rA, rB} = instr[8:15];
            valP = PC + 2;
        end

        default:                        // invalid instruction
            instr_valid = 1'b0;
    endcase
    end

endmodule
```

Decode & Write-Back Stage :



The decode stage is where the instruction is decoded and the required registers and memory locations are identified. The control logic reads the instruction from memory and decodes the opcode, which identifies the operation to be performed. The decode stage also identifies the operands of the instruction, which are the values that are used in the operation.

After the instruction is decoded, the values of the operands are retrieved from the registers or memory locations. This is done in the execute stage, which follows the decode stage.

The writeback stage is where the result of the operation is written back to the register or memory location. This stage occurs after the execute stage, where the operation is performed. The result of the operation is stored in a temporary register called the "destination register". In the writeback stage, the result is written back to the specified register or memory location.

For example, in the add instruction, the two operands are added together in the execute stage, and the result is stored in the destination register. In the writeback stage, the result is written back to the destination register specified in the instruction.

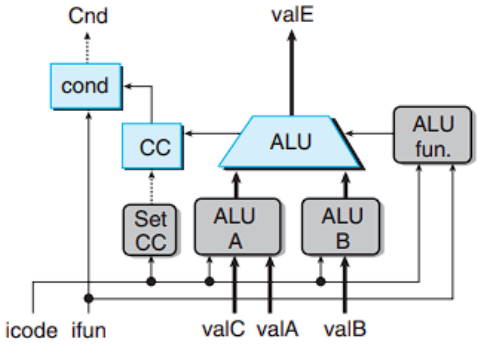
Given below is the code for implementing the decode + write back stage of the sequential processor

```
module decode(clk, icode, ifun, rA, rB, valA, valB, cnd, valE, valM, reg_0, reg_1, reg_2, reg_3, reg_4, reg_5, reg_6, reg_7, reg_8, re

    input clk, cnd;
    input [3:0] icode, ifun;
    input [3:0] rA, rB;
    input signed[63:0] valE, valM;

    output reg signed [63:0] valA, valB;

    output reg signed [63:0] reg_0;
    output reg signed [63:0] reg_1;
    output reg signed [63:0] reg_2;
    output reg signed [63:0] reg_3;
    output reg signed [63:0] reg_4;
    output reg signed [63:0] reg_5;
    output reg signed [63:0] reg_6;
    output reg signed [63:0] reg_7;
    output reg signed [63:0] reg_8;
    output reg signed [63:0] reg_9;
    output reg signed [63:0] reg_10;
    output reg signed [63:0] reg_11;
    output reg signed [63:0] reg_12;
    output reg signed [63:0] reg_13;
    output reg signed [63:0] reg_14;

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

    always@(*)

    begin

        reg_mem[0]  = 64'd1;
        reg_mem[1]  = 64'd2;
        reg_mem[2]  = 64'd3;
        reg_mem[3]  = 64'd3;
        reg_mem[4]  = 64'd5;
        reg_mem[5]  = 64'd1;
        reg_mem[6]  = 64'd2;
        reg_mem[7]  = 64'd3;
        reg_mem[8]  = 64'd4;
        reg_mem[9]  = 64'd62;
        reg_mem[10] = 64'd1;
        reg_mem[11] = 64'd2;
        reg_mem[12] = 64'd3;
        reg_mem[13] = 64'd4;
        reg_mem[14] = 64'd5;

        case(icode)

        4'b0010 : begin //cmovxx
        valB = 0;
        valA = reg_mem[rA];
        end

        4'b0100 : begin //rmmovq
            valA = reg_mem[rA];
            valB = reg_mem[rB];
        end

        4'b0101 : begin //mrmovq
            valA = 0;
            valB = reg_mem[rB];
        end

        4'b0110 : begin //OPq
            valA = reg_mem[rA];
            valB = reg_mem[rB];
        end

        4'b1000 : begin //call
            valB = reg_mem[4];
        end

        4'b1001 : begin //ret
            valA = reg_mem[4];
            valB = reg_mem[4];
        end

        4'b1010 : begin //pushq
            valA = reg_mem[rA];
```

```
            valB = reg_mem[4];
        end

        4'b1011 : begin //popq
            valA = reg_mem[4];
            valB = reg_mem[4];
        end

        endcase

        reg_0 = reg_mem[0];
        reg_1 = reg_mem[1];
        reg_2 = reg_mem[2];
        reg_3 = reg_mem[3];
        reg_4 = reg_mem[4];
        reg_5 = reg_mem[5];
        reg_6 = reg_mem[6];
        reg_7 = reg_mem[7];
        reg_8 = reg_mem[8];
        reg_9 = reg_mem[9];
        reg_10 = reg_mem[10];
        reg_11 = reg_mem[11];
        reg_12 = reg_mem[12];
        reg_13 = reg_mem[13];
        reg_14 = reg_mem[14];

    end

    always@(posedge clk)
    begin
        case(icode)

        4'b0010 : begin //cmovxx
            if(cnd == 1'b1)
            begin
                reg_mem[rB] = valE;
            end
        end

        4'b0011 : begin //irmovq
            reg_mem[rB] = valE;
        end

        4'b0101 : begin //mrmovq
            reg_mem[rA] = valM;
        end

        4'b0110 : begin //OPq
            reg_mem[rB] = valE;
        end

        4'b1000 : begin //call
            reg_mem[4] = valE;
        end

        4'b1001 : begin //ret
            reg_mem[4] = valE;
        end

        4'b1010 : begin //pushq
            reg_mem[4] = valE;
        end

        4'b1011 : begin //popq
            reg_mem[4] = valE;
            reg_mem[rA] = valM;
        end

        endcase

        reg_0 <= reg_mem[0];
        reg_1 <= reg_mem[1];
        reg_2 <= reg_mem[2];
        reg_3 <= reg_mem[3];
        reg_4 <= reg_mem[4];
        reg_5 <= reg_mem[5];
        reg_6 <= reg_mem[6];
        reg_7 <= reg_mem[7];
        reg_8 <= reg_mem[8];
        reg_9 <= reg_mem[9];
        reg_10 <= reg_mem[10];
        reg_11 <= reg_mem[11];
        reg_12 <= reg_mem[12];
        reg_13 <= reg_mem[13];
        reg_14 <= reg_mem[14];
    end
```

```
endmodule
```

Execute Stage :



During the execute stage, the operands identified in the decode stage are used to perform the operation specified by the instruction. The execute stage can involve a variety of operations, such as arithmetic and logic operations, memory operations, and control flow operations.

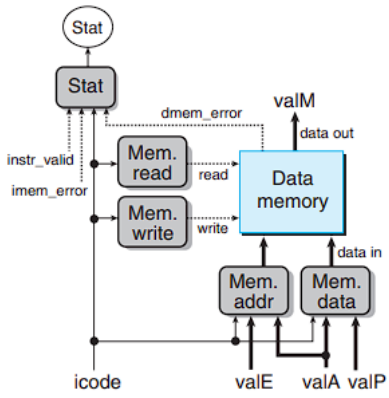
For arithmetic and logic operations, the ALU (Arithmetic Logic Unit) performs the necessary calculations based on the operands provided. For example, in an add instruction, the ALU would add the two operands and generate a result.

For memory operations, the execute stage involves reading or writing data to memory. This could involve loading data from memory into a register, storing data from a register into memory, or performing a combination of both.

```
module execute(clk, valE, cnd, CC_out, icode, ifun, valC, valA, valB, CC_in);

    output reg [63:0] valE;
    output reg cnd;
    output reg [2:0] CC_out;

    input [3:0] icode, ifun;
    input clk;
    input signed [63:0] valC, valA, valB;
    input [2:0] CC_in;              // ZF, SF, OF

    wire opq_of;
    reg prev_PC;

    wire zeroflag, signflag, overflow;
    assign zeroflag = CC_in[0];
    assign signflag = CC_in[1];
    assign overflow = CC_in[2];

    wire [63:0] valE_AB, valE_CB, valE_OP, valE_INC, valE_DEC;
    wire dummmy;

    always @(posedge clk) begin
        if(icode==2 | icode==7)begin
            case (ifun)
                4'h0: cnd = 1;                          // unconditional
                4'h1: cnd = (overflow^signflag)|zeroflag;   // le
                4'h2: cnd = overflow^signflag;              // l
                4'h3: cnd = zeroflag;                       // e
                4'h4: cnd = ~zeroflag;                      // ne
                4'h5: cnd = ~(signflag^overflow);           // ge
                4'h6: cnd = ~(signflag^overflow)&~zeroflag; // g
            endcase
        end
    end

    alu aluAB(.out(valE_AB), .overflow(dummy), .control(2'b0), .a(valA), .b(valB));
    alu aluOP(.out(valE_OP), .overflow(opq_of), .control(ifun[1:0]), .a(valA), .b(valB));
    alu aluCB(.out(valE_CB), .overflow(dummy), .control(2'b0), .a(valC), .b(valB));
    alu aluIN(.out(valE_INC), .overflow(dummy), .control(2'b0), .b(64'd1), .a(valB));
    alu aluDE(.out(valE_DEC), .overflow(dummy), .control(2'b1), .b(64'd1), .a(valB));
```

```
    always@*
    begin

        case (icode)

        4'h2:   valE = valE_AB;     // cmovx
        4'h3:   valE = valE_CB;     // irmovq
        4'h4:   valE = valE_CB;     // rmmovq
        4'h5:   valE = valE_CB;     // mrmovq
        4'h6:begin                  // opq
                valE = valE_OP;
                CC_out[2] <= opq_of;
                CC_out[1] <= valE[63];
                if(valE == 0) begin
                    CC_out[0] <= 1;
                end
                else begin
                    CC_out[0] <= 0;
                end
        end
        4'h8: valE = valE_DEC;        // call
        4'h9: valE = valE_INC;        // ret
        4'hA: valE = valE_DEC;        // pushq
        4'hB: valE = valE_INC;        // popq
        default: valE = 0;

        endcase

    end
endmodule
```

Memory Stage :



During the memory stage, the processor reads or writes data to memory based on the operation specified by the instruction. For example, if the instruction is a load instruction, the memory stage reads data from memory and stores it in a register. If the instruction is a store instruction, the memory stage writes data from a register to memory.

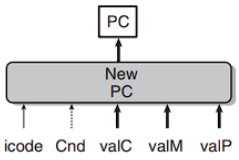
The memory stage can also include other memory-related operations, such as moving data from one memory location to another, copying data, or allocating memory. In addition, the memory stage can involve operations that access peripherals, such as reading from or writing to input/output devices.

Given below is the code for memory stage implementation of sequential

```
module memory(valM, mem_error, clk, icode, valE, valA, valP);

    input clk;
    input [3:0] icode;
    input signed [63:0] valE, valA;
    input [63:0] valP;

    output reg [63:0] valM;
    output reg mem_error;

    reg [63:0] memory [16383:0];
```

```
    always@*
    begin
        mem_error = 0;
        if(valE > 16383) begin
            mem_error = 1;
        end
        case(icode)
            4'h5:  valM = memory[valE];   // mrmovq
            4'h9:  valM = memory[valA];   // ret
            4'hB:  valM = memory[valA];   // popq
        endcase
    end

    always@(posedge clk)
    begin
        mem_error = 0;
        if(valE > 16383) begin
            mem_error = 1;
        end
        case(icode)
            4'h4:  memory[valE] <= valA;  // rmmovq
            4'h8:  memory[valE] <= valP;  // call
            4'hA:  memory[valE] <= valA;  // pushq
        endcase
    end

endmodule
```

PC Update :



During the PC update stage, the PC is updated based on the result of the previous instruction. The result can be a new memory address, a computed value, or a branch target address. The PC is updated to point to the address of the next instruction to be executed.

For example, if the previous instruction was a branch instruction, the PC update stage would update the PC to point to the target address of the branch. If the previous instruction was a computation or a memory operation, the PC update stage would simply increment the PC to point to the next instruction.

The PC update stage can also handle other control flow operations, such as jumps or calls, which change the order of instructions to be executed. For example, a jump instruction would set the PC to the specified address, effectively skipping over the next few instructions.

```
module pc_update(PC, clk, icode, cnd, valC, valM, valP);

    input [3:0] icode;
    input cnd, clk;
    input signed [63:0] valC, valM;
    input [63:0] valP;

    output reg [63:0] PC;

    always@(posedge clk)begin
        case(icode)
            4'h0: PC <= 0;
            4'h1: PC <= valP;
            4'h2: PC <= valP;
            4'h3: PC <= valP;
            4'h4: PC <= valP;
            4'h5: PC <= valP;
            4'h6: PC <= valP;
            4'h7: PC <= cnd ? valC:valP;
            4'h8: PC <= valC;
            4'h9: PC <= valM;
            4'hA: PC <= valP;
            4'hB: PC <= valP;
```

```
        endcase
    end

endmodule
```

Testbench :

The testbench includes status codes and are implemented using the below code

```
always@(instr_valid, imem_error, mem_error, icode) begin

        if(instr_valid==0)
        stat = 4'b0001;
        else if(imem_error==1)
        stat = 4'b0010;
        else if(mem_error==1)
        stat = 4'b0010;
        else if(icode==4'b0000)
        stat = 4'b0100;
        else
        stat = 4'b1000;

    end

    always@(stat) begin
        if(stat==4'b0100) begin
        $display("Halting");
        $finish;
        end
        else if(stat==4'b0010) begin
        $display("Address Error");
        $finish;
        end
        else if(stat==4'b0001) begin
        $display("Instruction Error");
        $finish;
        end
    end
```

The given instructions for the processor are as follows

```
//cmovxx
    instr_mem[0]=8'b00100000; //2 fn
    instr_mem[1]=8'b00010011; //rA rB

    //irmovq
      instr_mem[2]=8'b00110000; //3 0
      instr_mem[3]=8'b00000010; //F rB
      instr_mem[4]=8'b00000000; //V
      instr_mem[5]=8'b00000000; //V
      instr_mem[6]=8'b00000000; //V
      instr_mem[7]=8'b00000000; //V
      instr_mem[8]=8'b00000000; //V
      instr_mem[9]=8'b00000000; //V
      instr_mem[10]=8'b00000000; //V
      instr_mem[11]=8'b00010001; //V=17

    //rmmovq
      instr_mem[12]=8'b01000000; //4 0
      instr_mem[13]=8'b01010010; //rA rB
      instr_mem[14]=8'b00000000; //D
      instr_mem[15]=8'b00000000; //D
      instr_mem[16]=8'b00000000; //D
      instr_mem[17]=8'b00000000; //D
      instr_mem[18]=8'b00000000; //D
      instr_mem[19]=8'b00000000; //D
      instr_mem[20]=8'b00000000; //D
      instr_mem[21]=8'b00000001; //D

    //mrmovq
      instr_mem[22]=8'b01010000; //5 0
      instr_mem[23]=8'b00000111; //rA rB
      instr_mem[24]=8'b00000000; //D
      instr_mem[25]=8'b00000000; //D
      instr_mem[26]=8'b00000000; //D
      instr_mem[27]=8'b00000000; //D
      instr_mem[28]=8'b00000000; //D
      instr_mem[29]=8'b00000000; //D
```

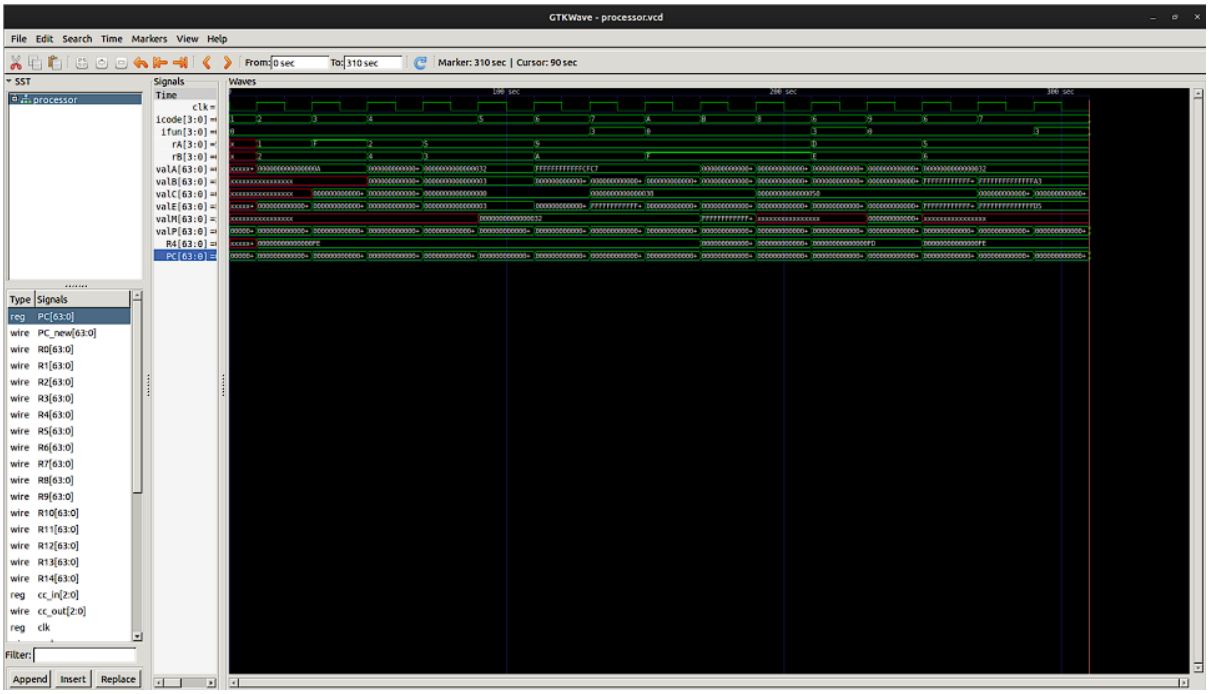
```
      instr_mem[30]=8'b00000000; //D
      instr_mem[31]=8'b00000001; //D

    //OPq
        instr_mem[32]=8'b01100001; //6 fn
        instr_mem[33]=8'b00100011; //rA rB


    //cmovxx
        instr_mem[34]=8'b00100000; //2 fn
        instr_mem[35]=8'b00110100; //rA rB

        instr_mem[36]=8'b00100101; // 2 ge
        instr_mem[37]=8'b01010011; // rA rB

    //nop
        instr_mem[38]=8'b00010000; // 1 0

    //je
        instr_mem[39]=8'b01110011; // 7 je
        instr_mem[40]=8'b00000000; //D
        instr_mem[41]=8'b00000000; //D
        instr_mem[42]=8'b00000000; //D
        instr_mem[43]=8'b00000000; //D
        instr_mem[44]=8'b00000000; //D
        instr_mem[45]=8'b00000000; //D
        instr_mem[46]=8'b00000000; //D
        instr_mem[47]=8'd49; //D

    //halt
        instr_mem[48]=8'b00000000; // 0 0

    //pushq
        instr_mem[49]=8'b10100000; // 10 0
        instr_mem[50]=8'b10011111; // 9 F

    //popq
        instr_mem[51]=8'b10110000; // 10 0
        instr_mem[52]=8'b10011111; // 9 F

    //call
        instr_mem[53]=8'h80;
        instr_mem[54]=8'b00000000; //D
        instr_mem[55]=8'b00000000; //D
        instr_mem[56]=8'b00000000; //D
        instr_mem[57]=8'b00000000; //D
        instr_mem[58]=8'b00000000; //D
        instr_mem[59]=8'b00000000; //D
        instr_mem[60]=8'b00000000; //D
        instr_mem[61]=8'd62; //D

    //halt
        instr_mem[62]=8'b00000000; // 0 0
```

And for the above set of instructions the following output is obtained

```
clk=1 PC=   0 icode=0010 ifun=0000 cnd=x CC_out=xxx rA=0001 rB=0011 ,valA=                2,valB=                0, valP=

clk=0 PC=   0 icode=0010 ifun=0000 cnd=x CC_out=xxx rA=0001 rB=0011 ,valA=                2,valB=                0, valP=

clk=1 PC=   2 icode=0011 ifun=0000 cnd=1 CC_out=xxx rA=0000 rB=0010 ,valA=                2,valB=                0, valP=

clk=0 PC=   2 icode=0011 ifun=0000 cnd=1 CC_out=xxx rA=0000 rB=0010 ,valA=                2,valB=                0, valP=

clk=1 PC=  12 icode=0100 ifun=0000 cnd=1 CC_out=xxx rA=0101 rB=0010 ,valA=                1,valB=                3, valP=

clk=0 PC=  12 icode=0100 ifun=0000 cnd=1 CC_out=xxx rA=0101 rB=0010 ,valA=                1,valB=                3, valP=

clk=1 PC=  22 icode=0101 ifun=0000 cnd=1 CC_out=xxx rA=0000 rB=0111 ,valA=                0,valB=                3, valP=

clk=0 PC=  22 icode=0101 ifun=0000 cnd=1 CC_out=xxx rA=0000 rB=0111 ,valA=                0,valB=                3, valP=

clk=1 PC=  32 icode=0110 ifun=0001 cnd=1 CC_out=001 rA=0010 rB=0011 ,valA=                3,valB=                3, valP=

clk=0 PC=  32 icode=0110 ifun=0001 cnd=1 CC_out=001 rA=0010 rB=0011 ,valA=                3,valB=                3, valP=

clk=1 PC=  34 icode=0010 ifun=0000 cnd=1 CC_out=001 rA=0011 rB=0100 ,valA=                3,valB=                0, valP=

clk=0 PC=  34 icode=0010 ifun=0000 cnd=1 CC_out=001 rA=0011 rB=0100 ,valA=                3,valB=                0, valP=

clk=1 PC=  36 icode=0010 ifun=0101 cnd=1 CC_out=001 rA=0101 rB=0011 ,valA=                1,valB=                0, valP=

clk=0 PC=  36 icode=0010 ifun=0101 cnd=1 CC_out=001 rA=0101 rB=0011 ,valA=                1,valB=                0, valP=
```

```
clk=1 PC=  38 icode=0001 ifun=0000 cnd=1 CC_out=001 rA=0101 rB=0011 ,valA=                  1,valB=                  0, valP=

clk=0 PC=  38 icode=0001 ifun=0000 cnd=1 CC_out=001 rA=0101 rB=0011 ,valA=                  1,valB=                  0, valP=

clk=1 PC=  39 icode=0111 ifun=0011 cnd=1 CC_out=001 rA=0101 rB=0011 ,valA=                  1,valB=                  0, valP=

clk=0 PC=  39 icode=0111 ifun=0011 cnd=1 CC_out=001 rA=0101 rB=0011 ,valA=                  1,valB=                  0, valP=

clk=1 PC=  49 icode=1010 ifun=0000 cnd=1 CC_out=001 rA=1001 rB=1111 ,valA=                 62,valB=                  5, valP=

clk=0 PC=  49 icode=1010 ifun=0000 cnd=1 CC_out=001 rA=1001 rB=1111 ,valA=                 62,valB=                  5, valP=

clk=1 PC=  51 icode=1011 ifun=0000 cnd=1 CC_out=001 rA=1001 rB=1111 ,valA=                  5,valB=                  5, valP=

clk=0 PC=  51 icode=1011 ifun=0000 cnd=1 CC_out=001 rA=1001 rB=1111 ,valA=                  5,valB=                  5, valP=

clk=1 PC=  53 icode=1000 ifun=0000 cnd=1 CC_out=001 rA=1001 rB=1111 ,valA=                  5,valB=                  5, valP=

clk=0 PC=  53 icode=1000 ifun=0000 cnd=1 CC_out=001 rA=1001 rB=1111 ,valA=                  5,valB=                  5, valP=

Halting
.\processor.v:51: $finish called at 240 (1s)
clk=1 PC=  62 icode=0000 ifun=0000 cnd=1 CC_out=001 rA=1001 rB=1111 ,valA=                  5,valB=                  5, valP=
```

GTKWave :



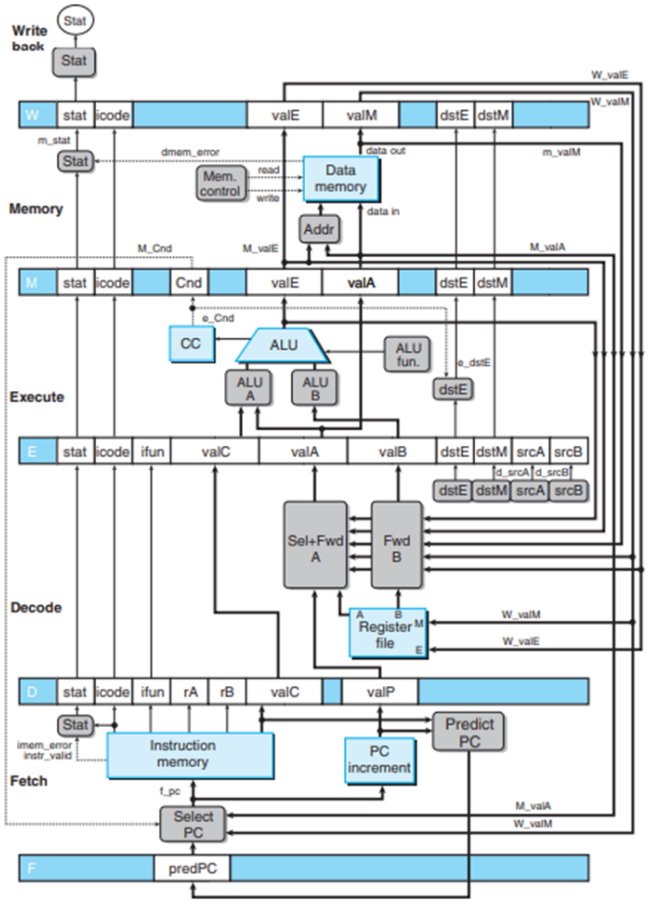
Pipelined implementation :

In pipeline implementation of the y86 processor, the instruction execution process is divided into five stages: fetch, decode, execute, memory, and writeback. Each instruction is broken down into smaller tasks, and each task is executed by a different stage of the pipeline. This allows multiple instructions to be in different stages of execution at the same time, leading to improved performance and throughput.

One of the main advantages of pipeline implementation is increased speed. Because each instruction is broken down into smaller tasks, multiple instructions can be executing simultaneously, resulting in a higher rate of instruction throughput. This leads to faster execution of programs and increased overall performance.

Another advantage is improved efficiency. Pipeline implementation allows for better resource utilization by minimizing idle time and reducing the amount of time that resources are unused. Additionally, pipeline implementation allows for more efficient use of resources such as memory and registers.

Block Diagram :



Stages :

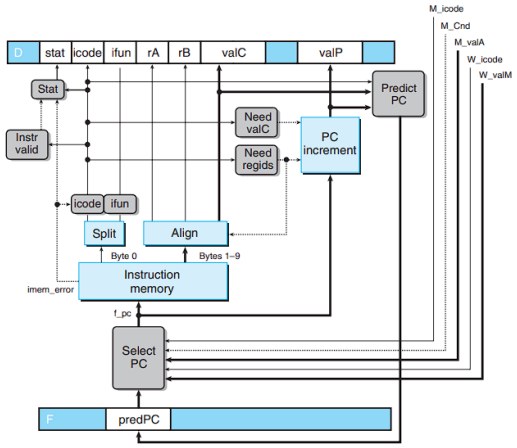
The pipeline implementation of the y86 processor consists of five stages, each of which performs a specific task in the execution of an instruction. These stages are:

1. **Fetch**: In this stage, the processor fetches the instruction from memory, reads the instruction opcode and increments the program counter to point to the next instruction.
2. **Decode**: In this stage, the instruction opcode is decoded, and the instruction operands are read from registers or memory. The instruction operands are then prepared for the execution stage.
3. **Execute**: In this stage, the instruction is executed by performing arithmetic, logical, or control operations on the operands. The result of the execution is then prepared for the memory stage.
4. **Memory**: In this stage, the processor reads from or writes to memory, or performs input/output operations. The result of the operation is then prepared for the writeback stage.
5. **Writeback**: In this stage, the result of the previous instruction is written back to a register. This stage completes the execution of the instruction.

In a pipelined processor, these stages are executed simultaneously for different instructions. For example, while the execute stage of one instruction is being performed, the decode stage of the next instruction can be executed. This overlap allows the processor to execute multiple instructions simultaneously, leading to improved performance and throughput.

Now let us go through each stage in detail :

Fetch Stage :



The fetch stage is the first stage in the pipeline. Its primary function is to fetch the instruction from memory and load it into an instruction register for the subsequent stages to execute. The process starts with incrementing the program counter (PC) to point to the next instruction in memory.

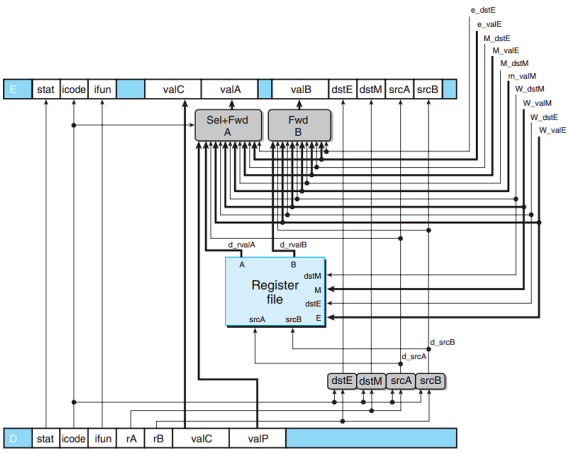
Then, the instruction is fetched from memory using the address in the PC, and once the instruction has been fetched, it is loaded into an instruction register in the processor. This stage is critical to the pipeline implementation of the y86 processor, as it sets the stage for the subsequent stages in the pipeline.

By fetching the instruction and loading it into the instruction register, the processor can begin decoding and executing the instruction in the subsequent stages, allowing for more efficient use of resources and improved performance.

In the pipelined implementation we additionally have the predictPC part where we predict the PC as follow

```
always @(*) begin
        case (f_icode)
            4'h7:   f_predPC <= f_valC;
            4'h8:   f_predPC <= f_valC;
            default:f_predPC <= f_valP;
        endcase
    end
```

Decode & Writeback Stage :



The decode and writeback stage is responsible for decoding the instruction and fetching the operands from registers or memory. The instruction is decoded to determine the operation to be performed and the operands that are needed for the

instruction. Once the instruction is decoded, the operands are fetched from either the register file or memory, depending on the addressing mode used by the instruction.

After the operands have been fetched, the operation specified by the instruction is performed on the operands fetched in the previous step. This may involve arithmetic or logical operations, data movement, or branching. Once the operation has been performed, the result of the operation is written back to the register file. This allows the result of the instruction to be stored in a register where it can be accessed by subsequent instructions.

In decode stage we firstly set the register values and then look for the data forwarding conditions as follows

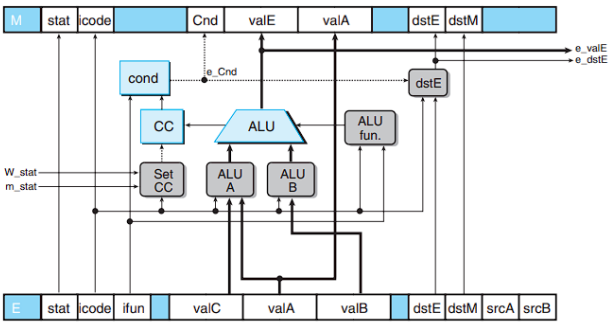
```
// Data forwarding of A
    always @(*) begin
        if (d_icode == 7 || d_icode == 8)
            begin
                d_valA <= D_valP;
            end
        else if (d_srcA == e_dstE)
            begin
                d_valA <= e_valE;
            end
        else if (d_srcA == M_dstM)
            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 <= reg_array[d_srcA];
            end
    end

    // Data forwarding of B
    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 <= reg_array[d_srcB];
            end
    end
```

For setting the data forwarding we need to first set the source registers of decode stage, the code for it is as follows

```
// d_dstE
    always @(*)
    begin
        case (d_icode)
            4'h2:   d_dstE <= D_rB;
            4'h3:   d_dstE <= D_rB;
            4'h6:   d_dstE <= D_rB;
```

```
            4'h8:   d_dstE <= 4;
            4'h9:   d_dstE <= 4;
            4'hA:   d_dstE <= 4;
            4'hB:   d_dstE <= 4;
            default:d_dstE <= 15;
        endcase
    end

    // d_dstM
    always @(*)
    begin
        case (d_icode)
            4'h5:   d_dstM <= D_rA;
            4'hB:   d_dstM <= D_rA;
            default:d_dstM <= 15;
        endcase
    end

    // d_srcA
    always @(*)
    begin
        case (d_icode)
            4'h2:   d_srcA <= D_rA;
            4'h4:   d_srcA <= D_rA;
            4'h6:   d_srcA <= D_rA;
            4'h9:   d_srcA <= 4;
            4'hA:   d_srcA <= 4;
            4'hB:   d_srcA <= 4;
            default:d_srcA <= 15;
        endcase
    end

    // d_srcB
    always @(*)
    begin
        case (d_icode)
            4'h4:   d_srcB <= D_rB;
            4'h5:   d_srcB <= D_rB;
            4'h6:   d_srcB <= D_rB;
            4'h8:   d_srcB <= 4;
            4'h9:   d_srcB <= 4;
            4'hA:   d_srcB <= 4;
            4'hB:   d_srcB <= 4;
            default:d_srcB <= 15;
        endcase
    end
```

Execute Stage :



The execute stage performs the actual operation specified by the instruction, such as arithmetic or logical operations, data movement, or branching. The operands for the operation are typically fetched from the register file or memory in the previous stage, and are available for use by the execute stage.

During the execute stage, the ALU (Arithmetic Logic Unit) performs the specified operation on the operands. The result of the operation may be written to a temporary register, which is used to store intermediate results during the execution of multi-stage instructions. The result may also be used to update the condition codes, which are used to determine whether subsequent instructions should be executed. Once the operation is complete, the result is passed on to the next stage in the pipeline.

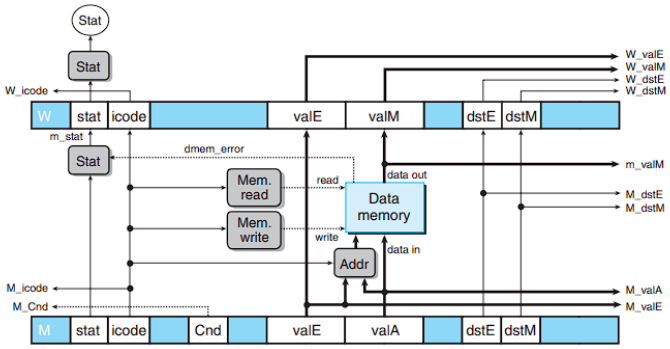
The outputs obtained from this block include M_stat, M_icode, Cnd, M_valE, M_valA, M_dstE, M_dstM, e_valE and e_dstE. Here E_stat, E_ifun, E_icode, E_valA, E_valB, E_valC, E_dstE and E_dstM which are the inputs when passed through this

block compute M_stat, M_icode, Cnd, M_valE, M_valA, M_dstE, M_dstM and e_valE as outputs in the similar way to that of sequential. The value e_dstE is computed based on e_cnd which will make it either E_dstE or an empty register.

We update the conditional codes at the positive edge of the clock as follows

```
always @(posedge clk )
    begin
        if(set_cc == 1)
            begin
                ZF = (e_valE == 64'b0); // zero flag
                SF = (e_valE[63] == 1); // signed flag
                OF = (aluA[63] == aluB[63]) && (e_valE[63] != aluA[63]); // overflow flag
            end
    end
```

Memory Stage :



During the memory stage, the processor may access the data cache or main memory to read or write data. For memory read operations, the processor fetches data from the memory location specified by the instruction and stores it in a temporary register. For memory write operations, the processor stores the data from a register into the specified memory location.

If the instruction being executed does not involve any memory operations, the memory stage is simply used to pass the result of the execute stage to the next stage of the pipeline.

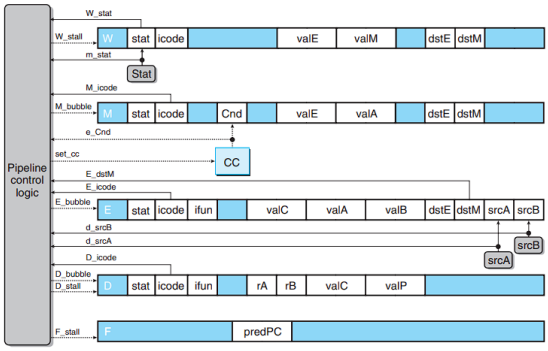
Memory stage allows the processor to access memory simultaneously with other pipeline stages. By overlapping the memory access with other stages, the processor can minimize the number of cycles needed to complete an instruction, improving overall performance.

This takes the inputs as the outputs from the memory pipelined register which include M_stat, M_icode, M_Cnd, M_valE, M_valA, M_dstE and M_dstM. The outputs obtained from this block include W_stat, W_icode, W_valE, W_valM, W_dstE, W_dstM and m_valM.

The address selection for either reading or writing the memory is handled by the code below

```
always @(*)
    begin
        case (m_icode)
            4'h4:   m_addr <= m_valE;
            4'h5:   m_addr <= m_valE;
            4'h8:   m_addr <= m_valE;
            4'h9:   m_addr <= M_valA;
            4'hA:   m_addr <= m_valE;
            4'hB:   m_addr <= M_valA;
            default:m_addr <= 4095;
        endcase
    end
```

Pipeline Control logic :



The pipeline control logic is responsible for managing the pipeline stages and ensuring that instructions are executed in the correct order. The pipeline control logic coordinates the flow of instructions through the pipeline, ensuring that each stage completes its operation before passing the instruction on to the next stage.

The pipeline control logic solves the following issues :

- Load/use hazards: The pipeline must stall for one cycle between an instruction that reads a value from memory and an instruction that uses this value.
- Processing return: The pipeline must stall until the ret instruction reaches the write-back stage.
- Mis-predicted branches: By the time the branch logic detects that a jump should not have been taken, several instructions at the branch target will have started down the pipeline. These instructions must be canceled, and fetching should begin at the instruction following the jump instruction
- Exceptions: When an instruction causes an exception, we want to disable the updating of the programmer-visible state by later instructions and halt execution once the excepting instruction reaches the write-back stage.

The following code checks for the control logic

```
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;
```

Testbench :

The set of instructions for checking the implemented pipelined processor are as follows:

```
instr_mem[0] = 8'h10; //nop instruction
instr_mem[1] = 8'h10; //nop instruction
instr_mem[2] = 8'h20; //rrmovq instruction
instr_mem[3] = 8'h12;
instr_mem[4] = 8'h30;//irmovq instruction
instr_mem[5] = 8'hF2;
instr_mem[6] = 8'h00;
instr_mem[7] = 8'h00;
instr_mem[8] = 8'h00;
instr_mem[9] = 8'h00;
instr_mem[10] = 8'h00;
instr_mem[11] = 8'h00;
instr_mem[12] = 8'h00;
instr_mem[13] = 8'h02;
instr_mem[14] = 8'h40;//rmmovq instruction
instr_mem[15] = 8'h24;
{instr_mem[16],instr_mem[17],instr_mem[18],instr_mem[19],instr_mem[20],instr_mem[21],instr_mem[22],instr_mem[23]} = 64'd1;
instr_mem[24] = 8'h40;//rmmovq instruction
instr_mem[25] = 8'h53;
{instr_mem[26],instr_mem[27],instr_mem[28],instr_mem[29],instr_mem[30],instr_mem[31],instr_mem[32],instr_mem[33]} = 64'd0;
instr_mem[34] = 8'h50;//mrmovq instruction
instr_mem[35] = 8'h53;
{instr_mem[36],instr_mem[37],instr_mem[38],instr_mem[39],instr_mem[40],instr_mem[41],instr_mem[42],instr_mem[43]} = 64'd0;
```

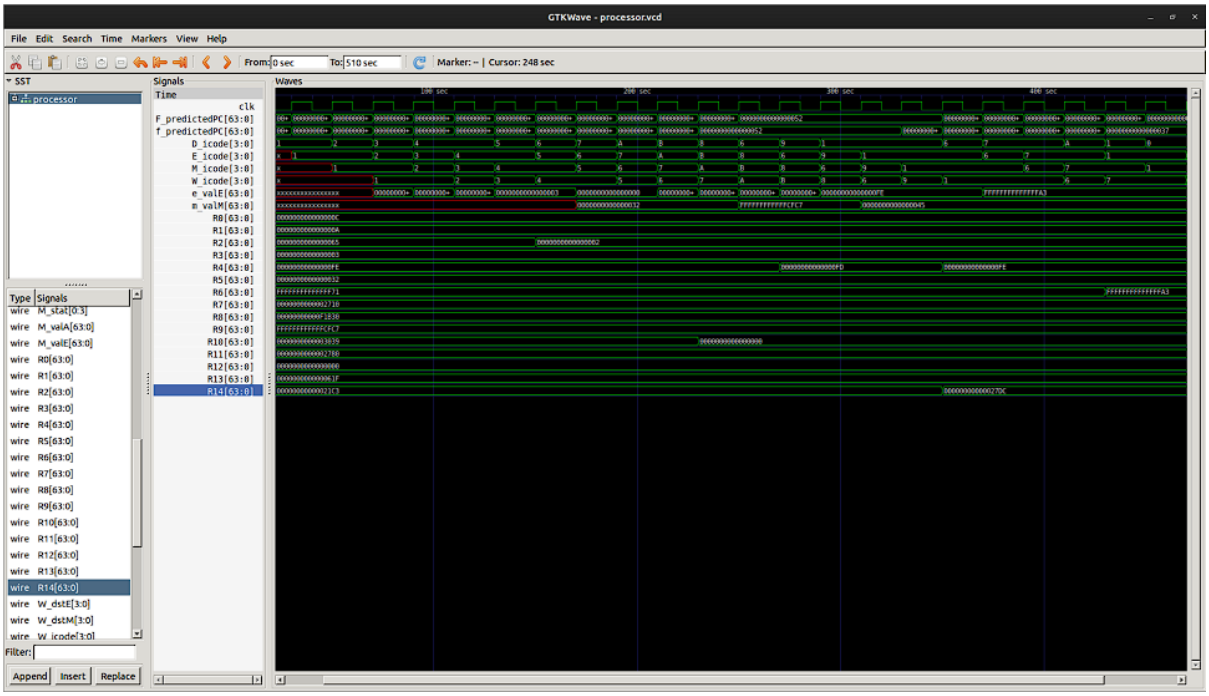
```
instr_mem[44] = 8'h60;
instr_mem[45] = 8'h9A;
instr_mem[46] = 8'h73;
{instr_mem[47],instr_mem[48],instr_mem[49],instr_mem[50],instr_mem[51],instr_mem[52],instr_mem[53],instr_mem[54]} = 64'd56;
instr_mem[55] = 8'h00;
instr_mem[56] = 8'hA0;
instr_mem[57] = 8'h9F;
instr_mem[58] = 8'hB0;
instr_mem[59] = 8'h9F;
instr_mem[60] = 8'h80;
{instr_mem[61],instr_mem[62],instr_mem[63],instr_mem[64],instr_mem[65],instr_mem[66],instr_mem[67],instr_mem[68]} = 64'd80;
instr_mem[69] = 8'h60;
instr_mem[70] = 8'h56;
instr_mem[71] = 8'h70;
{instr_mem[72],instr_mem[73],instr_mem[74],instr_mem[75],instr_mem[76],instr_mem[77],instr_mem[78],instr_mem[79]} = 64'd46;
instr_mem[80] = 8'h63;
instr_mem[81] = 8'hDE;
instr_mem[82] = 8'h90;
```

The output for the above set of instructions are as follows

```
time=0, clk=0, f_pc=1, f_icode=1, f_ifun=0, f_rA=15, f_rB=15, f_valP=1, f_valC=x, D_icode=1, E_icode=1, M_icode=1, W_icode=1 Stat= 8 r

time=10, clk=1, f_pc=2, f_icode=1, f_ifun=0, f_rA=15, f_rB=15, f_valP=2, f_valC=x, D_icode=1, E_icode=1, M_icode=1, W_icode=1 Stat= 8

time=20, clk=0, f_pc=2, f_icode=1, f_ifun=0, f_rA=15, f_rB=15, f_valP=2, f_valC=x, D_icode=1, E_icode=1, M_icode=1, W_icode=1 Stat= 8

time=30, clk=1, f_pc=4, f_icode=2, f_ifun=0, f_rA=1, f_rB=2, f_valP=4, f_valC=0, D_icode=1, E_icode=1, M_icode=1, W_icode=1 Stat= 8 re

time=40, clk=0, f_pc=4, f_icode=2, f_ifun=0, f_rA=1, f_rB=2, f_valP=4, f_valC=0, D_icode=1, E_icode=1, M_icode=1, W_icode=1 Stat= 8 re

time=50, clk=1, f_pc=14, f_icode=3, f_ifun=0, f_rA=15, f_rB=2, f_valP=14, f_valC=2, D_icode=2, E_icode=1, M_icode=1, W_icode=1 Stat= 8

time=60, clk=0, f_pc=14, f_icode=3, f_ifun=0, f_rA=15, f_rB=2, f_valP=14, f_valC=2, D_icode=2, E_icode=1, M_icode=1, W_icode=1 Stat= 8

time=70, clk=1, f_pc=24, f_icode=4, f_ifun=0, f_rA=2, f_rB=4, f_valP=24, f_valC=1, D_icode=3, E_icode=2, M_icode=1, W_icode=1 Stat= 8

time=80, clk=0, f_pc=24, f_icode=4, f_ifun=0, f_rA=2, f_rB=4, f_valP=24, f_valC=1, D_icode=3, E_icode=2, M_icode=1, W_icode=1 Stat= 8

time=90, clk=1, f_pc=34, f_icode=4, f_ifun=0, f_rA=5, f_rB=3, f_valP=34, f_valC=0, D_icode=4, E_icode=3, M_icode=2, W_icode=1 Stat= 8

time=100, clk=0, f_pc=34, f_icode=4, f_ifun=0, f_rA=5, f_rB=3, f_valP=34, f_valC=0, D_icode=4, E_icode=3, M_icode=2, W_icode=1 Stat= 8

time=110, clk=1, f_pc=44, f_icode=5, f_ifun=0, f_rA=5, f_rB=3, f_valP=44, f_valC=0, D_icode=4, E_icode=4, M_icode=3, W_icode=2 Stat= 8

time=120, clk=0, f_pc=44, f_icode=5, f_ifun=0, f_rA=5, f_rB=3, f_valP=44, f_valC=0, D_icode=4, E_icode=4, M_icode=3, W_icode=2 Stat= 8

time=130, clk=1, f_pc=46, f_icode=6, f_ifun=0, f_rA=9, f_rB=10, f_valP=46, f_valC=0, D_icode=5, E_icode=4, M_icode=4, W_icode=3 Stat=

time=140, clk=0, f_pc=46, f_icode=6, f_ifun=0, f_rA=9, f_rB=10, f_valP=46, f_valC=0, D_icode=5, E_icode=4, M_icode=4, W_icode=3 Stat=

time=150, clk=1, f_pc=56, f_icode=7, f_ifun=3, f_rA=0, f_rB=0, f_valP=55, f_valC=56, D_icode=6, E_icode=5, M_icode=4, W_icode=4 Stat=

time=160, clk=0, f_pc=56, f_icode=7, f_ifun=3, f_rA=0, f_rB=0, f_valP=55, f_valC=56, D_icode=6, E_icode=5, M_icode=4, W_icode=4 Stat=

time=170, clk=1, f_pc=58, f_icode=10, f_ifun=0, f_rA=9, f_rB=15, f_valP=58, f_valC=0, D_icode=7, E_icode=6, M_icode=5, W_icode=4 Stat=

time=180, clk=0, f_pc=58, f_icode=10, f_ifun=0, f_rA=9, f_rB=15, f_valP=58, f_valC=0, D_icode=7, E_icode=6, M_icode=5, W_icode=4 Stat=

time=190, clk=1, f_pc=60, f_icode=11, f_ifun=0, f_rA=9, f_rB=15, f_valP=60, f_valC=0, D_icode=10, E_icode=7, M_icode=6, W_icode=5 Stat

time=200, clk=0, f_pc=60, f_icode=11, f_ifun=0, f_rA=9, f_rB=15, f_valP=60, f_valC=0, D_icode=10, E_icode=7, M_icode=6, W_icode=5 Stat

time=210, clk=1, f_pc=80, f_icode=8, f_ifun=0, f_rA=0, f_rB=0, f_valP=69, f_valC=80, D_icode=11, E_icode=10, M_icode=7, W_icode=6 Stat

time=220, clk=0, f_pc=80, f_icode=8, f_ifun=0, f_rA=0, f_rB=0, f_valP=69, f_valC=80, D_icode=11, E_icode=10, M_icode=7, W_icode=6 Stat

time=230, clk=1, f_pc=82, f_icode=6, f_ifun=3, f_rA=13, f_rB=14, f_valP=82, f_valC=0, D_icode=8, E_icode=11, M_icode=10, W_icode=7 Sta

time=240, clk=0, f_pc=82, f_icode=6, f_ifun=3, f_rA=13, f_rB=14, f_valP=82, f_valC=0, D_icode=8, E_icode=11, M_icode=10, W_icode=7 Sta

time=250, clk=1, f_pc=83, f_icode=9, f_ifun=0, f_rA=x, f_rB=x, f_valP=83, f_valC=0, D_icode=6, E_icode=8, M_icode=11, W_icode=10 Stat=

time=260, clk=0, f_pc=83, f_icode=9, f_ifun=0, f_rA=x, f_rB=x, f_valP=83, f_valC=0, D_icode=6, E_icode=8, M_icode=11, W_icode=10 Stat=

time=270, clk=1, f_pc=84, f_icode=x, f_ifun=x, f_rA=15, f_rB=15, f_valP=84, f_valC=0, D_icode=9, E_icode=6, M_icode=8, W_icode=11 Stat

time=280, clk=0, f_pc=84, f_icode=x, f_ifun=x, f_rA=15, f_rB=15, f_valP=84, f_valC=0, D_icode=9, E_icode=6, M_icode=8, W_icode=11 Stat

time=290, clk=1, f_pc=84, f_icode=x, f_ifun=x, f_rA=15, f_rB=15, f_valP=84, f_valC=0, D_icode=1, E_icode=9, M_icode=6, W_icode=8 Stat=

time=300, clk=0, f_pc=84, f_icode=x, f_ifun=x, f_rA=15, f_rB=15, f_valP=84, f_valC=0, D_icode=1, E_icode=9, M_icode=6, W_icode=8 Stat=

time=310, clk=1, f_pc=84, f_icode=x, f_ifun=x, f_rA=15, f_rB=15, f_valP=84, f_valC=0, D_icode=1, E_icode=1, M_icode=9, W_icode=6 Stat=
```

```
time=320, clk=0, f_pc=84, f_icode=x, f_ifun=x, f_rA=15, f_rB=15, f_valP=84, f_valC=0, D_icode=1, E_icode=1, M_icode=9, W_icode=6 Stat=

time=330, clk=1, f_pc=71, f_icode=6, f_ifun=0, f_rA=5, f_rB=6, f_valP=71, f_valC=0, D_icode=1, E_icode=1, M_icode=1, W_icode=9 Stat= 8

time=340, clk=0, f_pc=71, f_icode=6, f_ifun=0, f_rA=5, f_rB=6, f_valP=71, f_valC=0, D_icode=1, E_icode=1, M_icode=1, W_icode=9 Stat= 8

time=350, clk=1, f_pc=46, f_icode=7, f_ifun=2, f_rA=0, f_rB=0, f_valP=80, f_valC=46, D_icode=6, E_icode=1, M_icode=1, W_icode=1 Stat=

time=360, clk=0, f_pc=46, f_icode=7, f_ifun=2, f_rA=0, f_rB=0, f_valP=80, f_valC=46, D_icode=6, E_icode=1, M_icode=1, W_icode=1 Stat=

time=370, clk=1, f_pc=56, f_icode=7, f_ifun=3, f_rA=0, f_rB=0, f_valP=55, f_valC=56, D_icode=7, E_icode=6, M_icode=1, W_icode=1 Stat=

time=380, clk=0, f_pc=56, f_icode=7, f_ifun=3, f_rA=0, f_rB=0, f_valP=55, f_valC=56, D_icode=7, E_icode=6, M_icode=1, W_icode=1 Stat=

time=390, clk=1, f_pc=58, f_icode=10, f_ifun=0, f_rA=9, f_rB=15, f_valP=58, f_valC=0, D_icode=7, E_icode=7, M_icode=6, W_icode=1 Stat=

time=400, clk=0, f_pc=58, f_icode=10, f_ifun=0, f_rA=9, f_rB=15, f_valP=58, f_valC=0, D_icode=7, E_icode=7, M_icode=6, W_icode=1 Stat=

time=410, clk=1, f_pc=60, f_icode=11, f_ifun=0, f_rA=9, f_rB=15, f_valP=60, f_valC=0, D_icode=10, E_icode=7, M_icode=7, W_icode=6 Stat

time=420, clk=0, f_pc=60, f_icode=11, f_ifun=0, f_rA=9, f_rB=15, f_valP=60, f_valC=0, D_icode=10, E_icode=7, M_icode=7, W_icode=6 Stat

time=430, clk=1, f_pc=56, f_icode=0, f_ifun=0, f_rA=15, f_rB=15, f_valP=56, f_valC=0, D_icode=1, E_icode=1, M_icode=7, W_icode=7 Stat=

time=440, clk=0, f_pc=56, f_icode=0, f_ifun=0, f_rA=15, f_rB=15, f_valP=56, f_valC=0, D_icode=1, E_icode=1, M_icode=7, W_icode=7 Stat=

time=450, clk=1, f_pc=58, f_icode=10, f_ifun=0, f_rA=9, f_rB=15, f_valP=58, f_valC=0, D_icode=0, E_icode=1, M_icode=1, W_icode=7 Stat=

time=460, clk=0, f_pc=58, f_icode=10, f_ifun=0, f_rA=9, f_rB=15, f_valP=58, f_valC=0, D_icode=0, E_icode=1, M_icode=1, W_icode=7 Stat=

time=470, clk=1, f_pc=60, f_icode=11, f_ifun=0, f_rA=9, f_rB=15, f_valP=60, f_valC=0, D_icode=10, E_icode=0, M_icode=1, W_icode=1 Stat

time=480, clk=0, f_pc=60, f_icode=11, f_ifun=0, f_rA=9, f_rB=15, f_valP=60, f_valC=0, D_icode=10, E_icode=0, M_icode=1, W_icode=1 Stat

time=490, clk=1, f_pc=80, f_icode=8, f_ifun=0, f_rA=0, f_rB=0, f_valP=69, f_valC=80, D_icode=11, E_icode=10, M_icode=0, W_icode=1 Stat

time=500, clk=0, f_pc=80, f_icode=8, f_ifun=0, f_rA=0, f_rB=0, f_valP=69, f_valC=80, D_icode=11, E_icode=10, M_icode=0, W_icode=1 Stat

pipe.v:394: $finish called at 510 (1s)
time=510, clk=1, f_pc=82, f_icode=6, f_ifun=3, f_rA=13, f_rB=14, f_valP=82, f_valC=0, D_icode=8, E_icode=11, M_icode=1, W_icode=0 Stat
```

GTKWave :



Challenges Faced during Implementation :

Implementing a pipeline architecture for the y86 processor presents several challenges that need to be addressed to ensure the proper functioning of the processor. Here are some of the key challenges:

1. Hazards: Hazards occur when one instruction depends on the result of a previous instruction, which may not yet be available due to the pipeline's stages. Handling these dependencies is essential to prevent incorrect results, and it requires additional logic to ensure that instructions are executed in the correct order. This is resolved by introducing bubble in appropriate stage.
2. Branches and Jumps: Branches and jumps can also create challenges in a pipelined architecture, as they can disrupt the normal flow of instructions in the pipeline. To handle these cases, the pipeline must detect these instructions and introduce bubble in appropriate stage so that the mispredicted branch is taken care of.
3. Data dependencies: When multiple instructions need to access the same data, such as a register or memory location, they may contend for that resource, leading to delays or incorrect results. We resolved this issue using data forwarding.
4. Return Instruction : In case of a ret instruction, we can never predict the target location as we did in jump. Therefore, we have to stall/bubble the instruction succeeding ret instructions and wait till the ret instruction reaches the memory stage, to find out the exact target location.
5. Instruction set design: The y86 instruction set is relatively simple, but some instructions require multiple pipeline stages to complete, which can introduce additional complexity into the pipeline design. The pipeline must also support all of the y86 instructions, including complex instructions like the divide instruction.