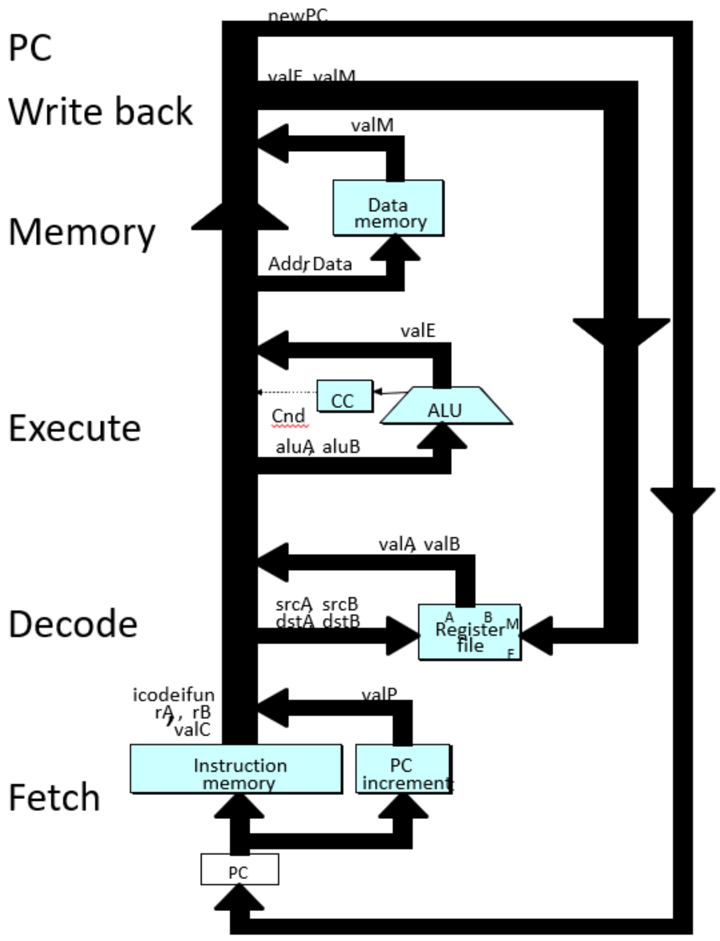
Intro to processor Architecture

Project: Pipeline implementation for the Y86-64 ISA

Implementing a pipeline for the Y86-64 ISA (Instruction Set Architecture) involves breaking down the execution of instructions into multiple stages .

Each instruction sequentially goes through following common stages:

- Fetch
- Decode
- Execute
- Memory
- Write – back
- PC update
  The processor loops indefinitely , performing the functions in each stage unless any exception condition occurs.



The Instruction set we are using here are:

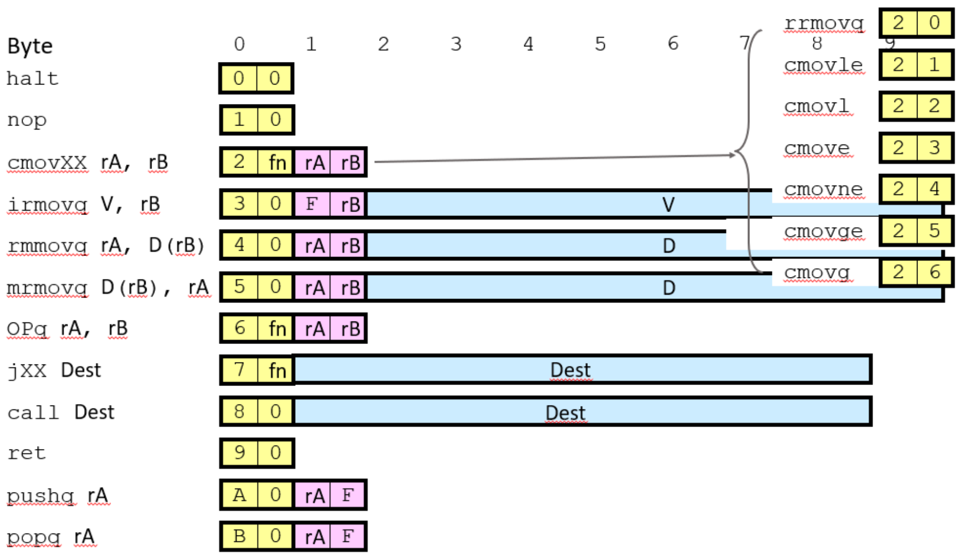
***

Byte	0	1	2	3	4	5	6	7	8	9
halt	0 0									
nop	1 0									
cmovXX rA, rB	2 fn	rA rB								
irmovq V, rB	3 0	F rB	V							
rmmovq rA, D(rB)	4 0	rA rB	D							
mrmovq D(rB), rA	5 0	rA rB	D							
OPq rA, rB	6 fn	rA rB								
jXX Dest	7 fn	Dest								
call Dest	8 0	Dest								
ret	9 0									
pushq rA	A 0	rA F								
popq rA	B 0	rA F								

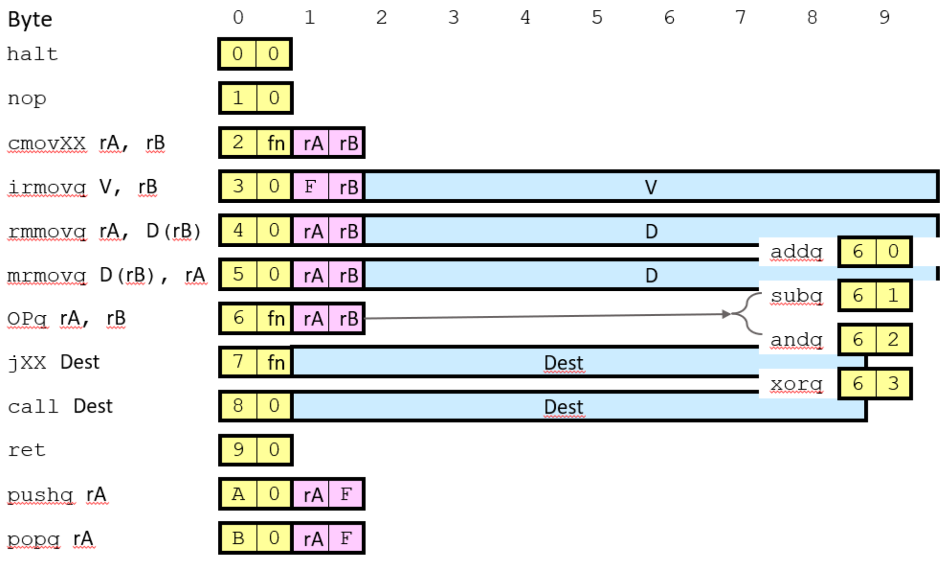
The register order in encoding here is correct - Verified

The register order in encoding here is correct - Verified

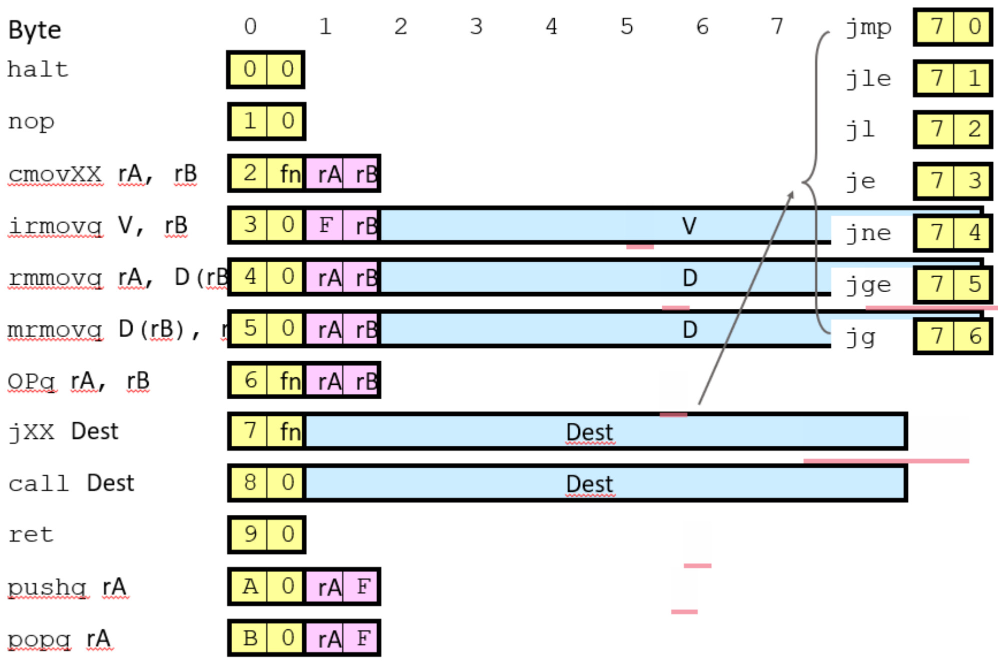
***


Byte
0 1 2 3 4 5 6 7 8 9
halt 0 0
nop 1 0
cmovXX rA, rB 2 fn rA rB
irmovq V, rB 3 0 F rB V
rmmovq rA, D(rB) 4 0 rA rB D
mrmovq D(rB), rA 5 0 rA rB D
OPq rA, rB 6 fn rA rB
jXX Dest 7 fn Dest
call Dest 8 0 Dest
ret 9 0
pushq rA A 0 rA F
popq rA B 0 rA F
rrmovq 2 0
cmovle 2 1
cmovl 2 2
cmove 2 3
cmovne 2 4
cmovge 2 5
cmovg 2 6

***


Byte
0 1 2 3 4 5 6 7 8 9
halt 0 0
nop 1 0
cmovXX rA, rB 2 fn rA rB
irmovq V, rB 3 0 F rB V
rmmovq rA, D(rB) 4 0 rA rB D
mrmovq D(rB), rA 5 0 rA rB D
OPq rA, rB 6 fn rA rB
jXX Dest 7 fn Dest
call Dest 8 0 Dest
ret 9 0
pushq rA A 0 rA F
popq rA B 0 rA F
addq 6 0
subq 6 1
andq 6 2
xorq 6 3

***



Fetch (F) : Fetch the instruction from Memory.

Decode (D) : Decode the instruction , determining its type and operand addresses.

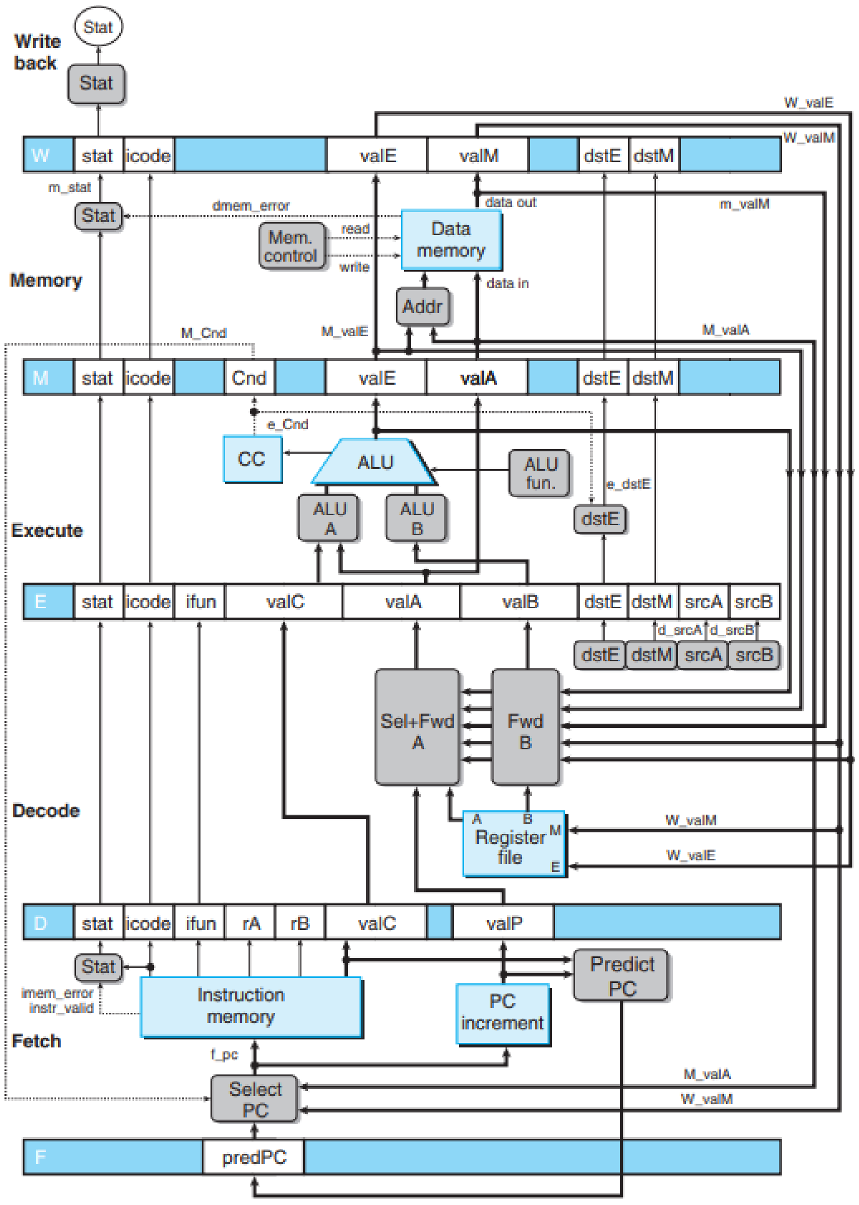
Execute (E) : Execute the instruction. This may involve arithmetic or logical operations , branch calculations , or memory accesses.

Memory (M) : If the instruction involves memory access (load/store) , perform the memory operation.

Write-back (W) : write the results of the instruction back to the appropriate registers.

- Each stage of the pipeline operates concurrently , with each instruction moving from one stage to the next on each clock cycle. However , due to dependencies between instructions , hazards may occur , which need to be handled appropriately to ensure correct execution.

=>OVERALL FINAL IMPLEMENTATION OF PIPELINING :



This is the Hardware structure of PIPE .

*Brief overview about some common hazards are handled in a pipeline :

1) Data Hazard: This occurs when the result of an instruction being executed in one stage is needed by another instruction in a subsequent stage . This is typically resolved by forwarding the data directly from the execution stage to the stages that need it(Data forwarding) or by stalling the pipeline until the data is available.

2)Control Hazard: This occurs when the pipeline needs to make a decision (eg: branch prediction) based on the results of an instruction that hasn’t completed execution yet. This is often resolved be predicting the outcome of the branch and speculatively continuing execution , then flushing the pipeline if the prediction was incorrect.

Here the PIPE- uses nearly the same set of hardware units as our sequential design SEQ, but with the pipeline registers separating the stages.

The pipeline registers are labeled as follows :

F: It holds a predicted value of the program counter, as will be discussed shortly.

D : It sits between the fetch and decode stages. It holds information about the most recently fetched instruction for processing for processing by the decode stage.

E : It sits between the decode and execute stages. It holds information about the most recently decoded instruction and the values read from the register file for processing by the execute stage.

M : It sits between the execute and memory stages. It holds the results of the most recently executed instruction for processing by the memory stage. It also holds information about branch conditions and branch targets for processing conditional jumps.

W : It sits between the memory stage and the feedback paths that supply the computed results to the register file for writing and the return address to the PC selection logic when completing a ret instruction.

Here the Aim is to achieve a 5 stage pipelined implementation of the processor.

1)sequential: The sequential implementation of the Y86-64 processor works with fetch , decode , execute , memory , writeback and PC update. In this implementation, only one instruction will be there in whole architecture in one clock cycle.

2) 5-stage pipeline: The pipelined implementation of the Y86-64 works in the same way as that of the sequential implementation with the modules being same but with inclusion of the pipelined registers , slightly change in the fetch and decode blocks , addition of support for data forwarding and PC prediction for improving the performance and the addition of the pipeline control logic for eliminating pipeline hazards . This type implementation increases throughput but increases latency is a trade-off.

SEQUENTIAL (SEQ) :

Fetch:

*It reads bytes of an instruction from memory using the PC value as address -> Extracts the two 4- bit portions of instruction specifiers byte referred to as icode and ifun.

Possibly fetches the register specifier byte giving one or both of the register operand specifiers rA and rB.

Also possibly fetched an 8-byte constant word valC-> computes valP as the address of the next instruction in the sequence , i.e

⇨ ValP=PC + length of fetched instruction.

CODE:

```
`timescale 1ns / 1ps

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

    input clk;
    input [63:0] PC;
    output reg [3:0] icode;
    output reg [3:0] ifun;
    output reg [3:0] rA;
    output reg [3:0] rB;
    output reg [63:0] valC;
    output reg [63:0] valP;
    output reg instr_valid;
    output reg imem_error;
    output reg hlt;
    reg xflag;
    integer clockbreak;

    //reg [63:0] PC;
    reg [7:0] inst_stack[0:2048];

    reg [0:79]instruction;
    reg [0:79]revbyteinstr;
    reg [63:0]temppc;

    initial begin
        xflag=0;
        temppc=64'b0;
        clockbreak=0;
        /*
        inst_stack[ 0 ] = 8'b00110000;
        inst_stack[ 1 ] = 8'b11110101;
        inst_stack[ 2 ] = 8'b00000000;
        inst_stack[ 3 ] = 8'b00000001;
        inst_stack[ 4 ] = 8'b00000000;
        inst_stack[ 5 ] = 8'b00000000;
        inst_stack[ 6 ] = 8'b00000000;
        inst_stack[ 7 ] = 8'b00000000;
        inst_stack[ 8 ] = 8'b00000000;
        inst_stack[ 9 ] = 8'b00000000;
        inst_stack[ 10 ] = 8'b00110000;
        inst_stack[ 11 ] = 8'b11110100;
        inst_stack[ 12 ] = 8'b00000000;
        inst_stack[ 13 ] = 8'b00000001;
        inst_stack[ 14 ] = 8'b00000000;
        inst_stack[ 15 ] = 8'b00000000;
        inst_stack[ 16 ] = 8'b00000000;
        inst_stack[ 17 ] = 8'b00000000;
        inst_stack[ 18 ] = 8'b00000000;
        inst_stack[ 19 ] = 8'b00000000;
        inst_stack[ 20 ] = 8'b00000000;
        */
        /*
        inst_stack[ 0 ] = 8'b00110000;
        inst_stack[ 1 ] = 8'b11110101;
        inst_stack[ 2 ] = 8'b00000000;
        inst_stack[ 3 ] = 8'b00000010;
```

```
inst_stack[ 4 ] = 8'b00000000;
inst_stack[ 5 ] = 8'b00000000;
inst_stack[ 6 ] = 8'b00000000;
inst_stack[ 7 ] = 8'b00000000;
inst_stack[ 8 ] = 8'b00000000;
inst_stack[ 9 ] = 8'b00000000;
inst_stack[ 10 ] = 8'b10000000;
inst_stack[ 11 ] = 8'b00111000;
inst_stack[ 12 ] = 8'b00000000;
inst_stack[ 13 ] = 8'b00000000;
inst_stack[ 14 ] = 8'b00000000;
inst_stack[ 15 ] = 8'b00000000;
inst_stack[ 16 ] = 8'b00000000;
inst_stack[ 17 ] = 8'b00000000;
inst_stack[ 18 ] = 8'b00000000;
inst_stack[ 19 ] = 8'b00000000;
inst_stack[ 20 ] = 8'b10000000;
inst_stack[ 24 ] = 8'b00001101;
inst_stack[ 25 ] = 8'b00000000;
inst_stack[ 26 ] = 8'b00001101;
inst_stack[ 27 ] = 8'b00000000;
inst_stack[ 28 ] = 8'b00001101;
inst_stack[ 32 ] = 8'b11000000;
inst_stack[ 33 ] = 8'b00000000;
inst_stack[ 34 ] = 8'b11000000;
inst_stack[ 35 ] = 8'b00000000;
inst_stack[ 36 ] = 8'b11000000;
inst_stack[ 40 ] = 8'b00000000;
inst_stack[ 41 ] = 8'b00001011;
inst_stack[ 42 ] = 8'b00000000;
inst_stack[ 43 ] = 8'b00001011;
inst_stack[ 44 ] = 8'b00000000;
inst_stack[ 45 ] = 8'b00001011;
inst_stack[ 48 ] = 8'b00000000;
inst_stack[ 49 ] = 8'b10100000;
inst_stack[ 50 ] = 8'b00000000;
inst_stack[ 51 ] = 8'b10100000;
inst_stack[ 52 ] = 8'b00000000;
inst_stack[ 53 ] = 8'b10100000;
inst_stack[ 56 ] = 8'b00110000;
inst_stack[ 57 ] = 8'b11110111;
inst_stack[ 58 ] = 8'b00011000;
inst_stack[ 59 ] = 8'b00000000;
inst_stack[ 60 ] = 8'b00000000;
inst_stack[ 61 ] = 8'b00000000;
inst_stack[ 62 ] = 8'b00000000;
inst_stack[ 63 ] = 8'b00000000;
inst_stack[ 64 ] = 8'b00000000;
inst_stack[ 65 ] = 8'b00000000;
inst_stack[ 66 ] = 8'b00110000;
inst_stack[ 67 ] = 8'b11110110;
inst_stack[ 68 ] = 8'b00000100;
inst_stack[ 69 ] = 8'b00000000;
inst_stack[ 70 ] = 8'b00000000;
inst_stack[ 71 ] = 8'b00000000;
inst_stack[ 72 ] = 8'b00000000;
inst_stack[ 73 ] = 8'b00000000;
inst_stack[ 74 ] = 8'b00000000;
inst_stack[ 75 ] = 8'b00000000;
inst_stack[ 76 ] = 8'b10000000;
inst_stack[ 77 ] = 8'b01010110;
```

```
inst_stack[ 78 ] = 8'b00000000;
inst_stack[ 79 ] = 8'b00000000;
inst_stack[ 80 ] = 8'b00000000;
inst_stack[ 81 ] = 8'b00000000;
inst_stack[ 82 ] = 8'b00000000;
inst_stack[ 83 ] = 8'b00000000;
inst_stack[ 84 ] = 8'b00000000;
inst_stack[ 85 ] = 8'b10010000;
inst_stack[ 86 ] = 8'b00110000;
inst_stack[ 87 ] = 8'b11111000;
inst_stack[ 88 ] = 8'b00001000;
inst_stack[ 89 ] = 8'b00000000;
inst_stack[ 90 ] = 8'b00000000;
inst_stack[ 91 ] = 8'b00000000;
inst_stack[ 92 ] = 8'b00000000;
inst_stack[ 93 ] = 8'b00000000;
inst_stack[ 94 ] = 8'b00000000;
inst_stack[ 95 ] = 8'b00000000;
inst_stack[ 96 ] = 8'b00110000;
inst_stack[ 97 ] = 8'b11111001;
inst_stack[ 98 ] = 8'b00000001;
inst_stack[ 99 ] = 8'b00000000;
inst_stack[ 100 ] = 8'b00000000;
inst_stack[ 101 ] = 8'b00000000;
inst_stack[ 102 ] = 8'b00000000;
inst_stack[ 103 ] = 8'b00000000;
inst_stack[ 104 ] = 8'b00000000;
inst_stack[ 105 ] = 8'b00000000;
inst_stack[ 106 ] = 8'b01100011;
inst_stack[ 107 ] = 8'b00000000;
inst_stack[ 108 ] = 8'b01100010;
inst_stack[ 109 ] = 8'b01100110;
inst_stack[ 110 ] = 8'b01110000;
inst_stack[ 111 ] = 8'b10000111;
inst_stack[ 112 ] = 8'b00000000;
inst_stack[ 113 ] = 8'b00000000;
inst_stack[ 114 ] = 8'b00000000;
inst_stack[ 115 ] = 8'b00000000;
inst_stack[ 116 ] = 8'b00000000;
inst_stack[ 117 ] = 8'b00000000;
inst_stack[ 118 ] = 8'b00000000;
inst_stack[ 119 ] = 8'b01010000;
inst_stack[ 120 ] = 8'b10100111;
inst_stack[ 121 ] = 8'b00000000;
inst_stack[ 122 ] = 8'b00000000;
inst_stack[ 123 ] = 8'b00000000;
inst_stack[ 124 ] = 8'b00000000;
inst_stack[ 125 ] = 8'b00000000;
inst_stack[ 126 ] = 8'b00000000;
inst_stack[ 127 ] = 8'b00000000;
inst_stack[ 128 ] = 8'b00000000;
inst_stack[ 129 ] = 8'b01100000;
inst_stack[ 130 ] = 8'b10100000;
inst_stack[ 131 ] = 8'b01100000;
inst_stack[ 132 ] = 8'b10000111;
inst_stack[ 133 ] = 8'b01100001;
inst_stack[ 134 ] = 8'b10010110;
inst_stack[ 135 ] = 8'b01110100;
inst_stack[ 136 ] = 8'b01110111;
inst_stack[ 137 ] = 8'b00000000;
inst_stack[ 138 ] = 8'b00000000;
```

```
    inst_stack[ 139 ] = 8'b00000000;
    inst_stack[ 140 ] = 8'b00000000;
    inst_stack[ 141 ] = 8'b00000000;
    inst_stack[ 142 ] = 8'b00000000;
    inst_stack[ 143 ] = 8'b00000000;
    inst_stack[ 144 ] = 8'b10010000;
    inst_stack[ 145 ] = 8'b00000000;
    */
    //nop
    inst_stack[0] = 8'b00010000;
    //irmovq
    inst_stack[1] = 8'b00110000;
    inst_stack[2] = 8'b11000001;
    inst_stack[3] = 8'b00100110;//valc
    inst_stack[4] = 8'b00000110;
    inst_stack[5] = 8'b00000000;
    inst_stack[6] = 8'b00000000;
    inst_stack[7] = 8'b00000000;
    inst_stack[8] = 8'b00000000;
    inst_stack[9] = 8'b00000000;
    inst_stack[10] = 8'b00000000;
    //cmov-movq
    inst_stack[11] = 8'b00100000;
    inst_stack[12] = 8'b00010010;
    //irmovq
    inst_stack[13] = 8'b00110000;
    inst_stack[14] = 8'b01100000;
    inst_stack[15] = 8'b01011010;//valc
    inst_stack[16] = 8'b00010110;
    inst_stack[17] = 8'b00000000;
    inst_stack[18] = 8'b00000000;
    inst_stack[19] = 8'b00000000;
    inst_stack[20] = 8'b00000000;
    inst_stack[21] = 8'b00000000;
    inst_stack[22] = 8'b00000000;
    //opq
    inst_stack[23] = 8'b01100001;
    inst_stack[24] = 8'b00000010;
    //jXX
    inst_stack[25] = 8'b01110000;
    inst_stack[26] = 8'b00100001;//valC
    inst_stack[27] = 8'b00000000;
    inst_stack[28] = 8'b00000000;
    inst_stack[29] = 8'b00000000;
    inst_stack[30] = 8'b00000000;
    inst_stack[31] = 8'b00000000;
    inst_stack[32] = 8'b00000000;
    inst_stack[33] = 8'b00000000;
    //nop
    inst_stack[34] = 8'b00000001;
    //halt
    inst_stack[35] = 8'b00000000;

end


always@(*) begin

    //clockbreak=clockbreak+1;
    //if(clockbreak>200) begin
    //    hlt=0;
```

```
        //    valP=64'bX;
        //end
        //
        //temppc = PC;
        //if((^temppc === 1'bX)) begin
        //    xflag=1;
        //    $display("PC is undefined");
        //end

        instruction = {
            inst_stack[PC],
            inst_stack[PC+64'd1],
            inst_stack[PC+64'd2],
            inst_stack[PC+64'd3],
            inst_stack[PC+64'd4],
            inst_stack[PC+64'd5],
            inst_stack[PC+64'd6],
            inst_stack[PC+64'd7],
            inst_stack[PC+64'd8],
            inst_stack[PC+64'd9]
        };

        revbyteinstr = {
            inst_stack[PC+64'd9], // 0
            inst_stack[PC+64'd8], // 8
            inst_stack[PC+64'd7], // 16
            inst_stack[PC+64'd6],
            inst_stack[PC+64'd5],
            inst_stack[PC+64'd4],
            inst_stack[PC+64'd3],
            inst_stack[PC+64'd2],  // 56
            inst_stack[PC+64'd1],  // 64
            inst_stack[PC]         // 72
        };

        icode = instruction[0:3];
        ifun = instruction[4:7];

        if (icode > 4'b1011)
            instr_valid = 1'b0;
        else
            instr_valid = 1'b1;


        case (icode)
            4'b0000: begin // halt
                hlt = 1;
                valP = PC;
            end
            4'b0001, 4'b1001: begin // nop
                valP = PC + 64'd1;
            end
            4'b0010, 4'b0110, 4'b1010, 4'b1011: begin // cmovXX, opq,
pushq, popq
                rA = instruction[8:11];
                rB = instruction[12:15];
                valP = PC + 64'd2;
            end
            4'b0011, 4'b0100, 4'b0101: begin // irmovq, rmmovq, mrmovq
                rA = instruction[8:11];
                rB = instruction[12:15];
                // FLIP BYTES 3-10
                //valC = instruction[16:79];
                valC = revbyteinstr[0:63];
                valP = PC + 64'd10;
            end
            4'b0111, 4'b1000: begin // jXX, call
                // FLIP BYTES 2-9
                //valC = instruction[8:71];
                valC = revbyteinstr[8:71];
                valP = PC + 64'd9;
            end
        endcase

        //if(xflag==1) begin
        //    hlt=1;
        //    //valP=2048;
        //end
        //PC = valP;
        //$monitor("clk=%d icode=%d ifun=%d rA=%d rB=%d PC=%d, valP=%d \n",
clk, icode, ifun, rA, rB, PC, valP);
    end

endmodule
```

DECODE:

It Reads up to two operands from the register file giving values valA and/or valB.

For some instructions , it reads register %rsp.

CODE:

```
`timescale 1ns / 1ps

module decode(clk, icode, rA, rB, valA, valB, reg_mem0, reg_mem1, reg_mem2,
reg_mem3, reg_mem4, reg_mem5, reg_mem6, reg_mem7, reg_mem8, reg_mem9,
reg_mem10, reg_mem11, reg_mem12, reg_mem13, reg_mem14);
    input clk;
    input [3:0] icode;
    input [3:0] rA;
    input [3:0] rB;
    output reg [63:0] valA;
    output reg [63:0] valB;
    input [63:0] reg_mem0;
    input [63:0] reg_mem1;
    input [63:0] reg_mem2;
    input [63:0] reg_mem3;
    input [63:0] reg_mem4;
    input [63:0] reg_mem5;
    input [63:0] reg_mem6;
    input [63:0] reg_mem7;
    input [63:0] reg_mem8;
    input [63:0] reg_mem9;
    input [63:0] reg_mem10;
    input [63:0] reg_mem11;
    input [63:0] reg_mem12;
    input [63:0] reg_mem13;
    input [63:0] reg_mem14;

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

    //always @(*) begin
    //    reg_mem[0] <= reg_mem0;
    //    reg_mem[1] <= reg_mem1;
    //    reg_mem[2] <= reg_mem2;
    //    reg_mem[3] <= reg_mem3;
    //    reg_mem[4] <= reg_mem4;
    //    reg_mem[5] <= reg_mem5;
    //    reg_mem[6] <= reg_mem6;
    //    reg_mem[7] <= reg_mem7;
    //    reg_mem[8] <= reg_mem8;
    //    reg_mem[9] <= reg_mem9;
    //    reg_mem[10] <=reg_mem10;
    //    reg_mem[11] <=reg_mem11;
    //    reg_mem[12] <=reg_mem12;
    //    reg_mem[13] <=reg_mem13;
    //    reg_mem[14] <=reg_mem14;
    //end

    always @(*) begin
        reg_mem[0] <= reg_mem0;
        reg_mem[1] <= reg_mem1;
        reg_mem[2] <= reg_mem2;
        reg_mem[3] <= reg_mem3;
        reg_mem[4] <= reg_mem4;
        reg_mem[5] <= reg_mem5;
        reg_mem[6] <= reg_mem6;
        reg_mem[7] <= reg_mem7;
        reg_mem[8] <= reg_mem8;
        reg_mem[9] <= reg_mem9;
        reg_mem[10] <=reg_mem10;
        reg_mem[11] <=reg_mem11;
        reg_mem[12] <=reg_mem12;
        reg_mem[13] <=reg_mem13;
        reg_mem[14] <=reg_mem14;
        #5
        case(icode)
            4'b0010: begin // cmovxx
                #2
                valA = reg_mem[rA];
            end

            4'b0101, 4'b0110: begin // rmmov, opq
                #2
                valA = reg_mem[rA];
                #2
                valB = reg_mem[rB];
            end


            4'b0100: begin//mrmov
                #2
                valB = reg_mem[rB];
            end
```

```
            4'b1000: begin//call
                #2
                valB = reg_mem[4];
            end

            4'b1001, 4'b1011: begin//ret, popq
                #2
                valA = reg_mem[4];
                #2
                valB = reg_mem[4];

            end

            4'b1010: begin//pushq
                #2
                valA = reg_mem[rA];
                #2
                valB = reg_mem[4];
            end

        endcase

        //$display("valA=%d, valB=%d", valA, valB);

        //$display("reg%d=%x, reg%d=%x", rA, reg_mem[rA], rB, reg_mem[rB]);
    end

endmodule
```

EXECUTE:

*ALU either performs operation given be ifun , computes effective address of a memory reference , or increments or decrements the stack pointer .

Resulting value ->ValE.

Condition codes are possibly set.

For a jump instruction , tests condition code and branch condition (referred to by ifun) to determine if branch should be taken or not.

CODE:

```
`timescale 1ns / 1ps
```

```verilog
`include "./alu/alu.v"

module execute(clk, valA, valB, valC, ifun, icode, sf, of, zf, valE, cond);

    input clk;
    input [63:0] valA;
    input [63:0] valB;
    input [63:0] valC;
    input [3:0] ifun;
    input [3:0] icode;
    output reg sf;
    output reg zf;
    output reg of;
    output reg [63:0] valE;
    output reg [0:0] cond;

    reg a_sf;
    reg b_sf;
    reg ans;


    always @(*) begin

        cond = 0;

        case(icode)

            4'b0010: begin // cmovxx
                #2
                valE = valA;

                case(ifun)
                    4'b0000:begin //rrmovq
                        #1
                        cond = 1;
                    end
                    4'b0001:begin //cmovle
                        if(sf==1 || zf==1)
                            #1
                            cond = 1;
                    end
                    4'b0010:begin // cmovl
                        if(sf==1)
                            #1
                            cond = 1;
                    end
                    4'b0011:begin // cmove
                        if(zf==1)
                            #1
                            cond = 1;
                    end
                    4'b0100:begin //cmovne
                        if(zf==0)
                            #1
                            cond = 1;
                    end
                    4'b0101:begin //comvge
                        if(sf==0 || zf ==1)
                            #1
                            cond = 1;
                    end
                    4'b0110:begin //cmovg
                        if(sf==0)
                            #1
                            cond = 1;
                    end
                endcase
            end

            4'b0011: begin //irmovq
                #2
                valE =  valC;
            end

            4'b0100, 4'b0101:begin //rmmovq, mrmovq
                #2;
                valE = valB+valC;
            end

            4'b0110:begin //opq

                case(ifun)
                    4'b0000:begin
                        ans = valA + valB;
                    end
                    4'b0001:begin
                        ans = valA - valB;
                    end
                    4'b0010:begin
                        ans = valA & valB;
                    end
                    4'b0011:begin
                        ans = valA ^ valB;
                    end
                endcase

                zf = (ans == 1'b0);
                sf = (ans < 1'b0);
                a_sf = (valA < 1'b0);
                b_sf = (valB < 1'b0);
                of = (a_sf == b_sf) && (sf != a_sf);

                #2
                valE = ans;
            end

            4'b0111:begin//jXX

                case(ifun)
                    4'b0000:begin //jmp
                        #1
                        cond = 1;
                    end
                    4'b0001:begin //jle
                        if(sf==1 || zf==1)
                            #1
                            cond = 1;
                    end
                    4'b0010:begin // jl
                        if(sf==1)
                            #1
                            cond = 1;
```

```
                    end
                    4'b0011:begin // je
                        if(zf==1)
                            #1
                            cond = 1;
                    end
                    4'b0100:begin //jne
                        if(zf==0)
                            #1
                            cond = 1;
                    end
                    4'b0101:begin //jge
                        if(sf==0 || zf ==1)
                            #1
                            cond = 1;
                    end
                    4'b0110:begin //jg
                        if(sf==0)
                            #1
                            cond = 1;
                    end

                endcase
            end

            4'b1000, 4'b1010:begin // call, pushq
                #2
                valE = valB - 8;
            end

            4'b1001, 4'b1011:begin // ret, popq
                #2
                valE = valB + 8;
            end
        endcase

        //$display("Execute valE: %d\n", valE);
    end
endmodule
```

MEMORY

It may read or write data from /to memory respectively .

Value read referred to as valM.

CODE:

```
`timescale 1ns / 1ps

module memory(clk, icode, valA, valM, valP, valE, dmem_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 dmem_error;


    reg [63:0] memory[0:2048];

    reg [0:0] read;
    reg [63:0] address;
    reg [63:0] value;

initial begin
    valM = 64'b0;
    dmem_error = 0;
    //address = 64'b0;
    /*
    genvar i; //memiter
    generate for(i=0; i<2047; i=i+1)
    begin
        memory[i]=64'b0;
    end
    endgenerate
    */
end

//check for memory error
    always @(*) begin
        dmem_error=0;
        case(icode)

            4'b0100,4'b1010: begin // rmmovq, pushq
                read = 0;
                #2
                address = valE;
                value = valA;
            end
            4'b0101: begin //mrmovq
                read = 1;
                #2
                address = valE;
            end
            4'b1000 : begin //call
                read = 0;
                #2
                address = valE;
                value = valP;
            end
            4'b1001, 4'b1011 : begin // ret, popq
                read = 1;
                #2
                address = valA;
            end
        endcase

        //case(icode)
        //
        //    4'b0100,4'b1010: begin // rmmovq, pushq
        //        #2
        //        memory[valE] = valA;
        //    end
        //    4'b0101: begin //mrmovq
        //        #2
        //        valM = memory[valE];
        //    end
        //    4'b1000 : begin //call
        //        #2
        //        memory[valE] = valP;
        //    end
        //    4'b1001, 4'b1011 : begin // ret, popq
        //        #2
        //        valM = memory[valA];
        //    end
        //endcase


        //if(address>2047) begin
        //    dmem_error = 1;
        //    $display("RAM wrong location");
        //    //$finish;
        //end

        // read and write operations
        if (read == 0) begin
            #2
            memory[address] = value;
        end
        if(read == 1) begin
            #2
            valM = memory[address];
        end

        //if((^valM === 1'bX)) begin
        //    valM = 64'b0;
        //end

        //$display("Memblock valM=%h", valM);
    end
endmodule
```

WRITEBACK:

It writes up to two results to the register file.

CODE:

```
`timescale 1ns / 1ps

 module writeback(
    input clk,
    input cond,
    input [3:0] icode,
    input [3:0] rA,
    input [3:0] rB,
    input [63:0] valE,
    input [63:0] valM,
    output reg [63:0] reg_mem0,
    output reg [63:0] reg_mem1,
    output reg [63:0] reg_mem2,
    output reg [63:0] reg_mem3,
    output reg [63:0] reg_mem4,
    output reg [63:0] reg_mem5,
    output reg [63:0] reg_mem6,
    output reg [63:0] reg_mem7,
    output reg [63:0] reg_mem8,
    output reg [63:0] reg_mem9,
    output reg [63:0] reg_mem10,
    output reg [63:0] reg_mem11,
    output reg [63:0] reg_mem12,
    output reg [63:0] reg_mem13,
    output reg [63:0] reg_mem14
    );

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

    always @(*) begin
        reg_mem[0] <= reg_mem0;
        reg_mem[1] <= reg_mem1;
        reg_mem[2] <= reg_mem2;
        reg_mem[3] <= reg_mem3;
        reg_mem[4] <= reg_mem4;
        reg_mem[5] <= reg_mem5;
        reg_mem[6] <= reg_mem6;
        reg_mem[7] <= reg_mem7;
        reg_mem[8] <= reg_mem8;
        reg_mem[9] <= reg_mem9;
        reg_mem[10] <=reg_mem10;
        reg_mem[11] <=reg_mem11;
        reg_mem[12] <=reg_mem12;
        reg_mem[13] <=reg_mem13;
        reg_mem[14] <=reg_mem14;
    end

    always @(*) begin
        case(icode)

            4'b0010: begin // cmovxx
                if(cond == 1'b1) begin
                    #5
                    reg_mem[rB]=valE;
                    $display("test1");
                end
            end

            4'b0011: begin//irmovq
                #5
                reg_mem[rB]=valE;
                $display("test2");
            end

            4'b0110: begin//opq
                #5
                reg_mem[rB]=valE;
                $display("test2");
            end

            4'b0101: begin//mrmovq
                #5
                reg_mem[rA]=valM;
                $display("test3");
            end
```

```
            4'b1000, 4'b1001, 4'b1010: begin//call, ret, pushq
                #5
                reg_mem[4]=valE;
                $display("test4");
            end

            4'b1011: begin//popq
                #5
                reg_mem[4]=valE;
                #2
                reg_mem[rA]=valM;
                $display("test5");
            end

        endcase
    end



    always @(*) begin

        reg_mem0 <= reg_mem[0];
        reg_mem1 <= reg_mem[1];
        reg_mem2 <= reg_mem[2];
        reg_mem3 <= reg_mem[3];
        reg_mem4 <= reg_mem[4];
        reg_mem5 <= reg_mem[5];
        reg_mem6 <= reg_mem[6];
        reg_mem7 <= reg_mem[7];
        reg_mem8 <= reg_mem[8];
        reg_mem9 <= reg_mem[9];
        reg_mem10 <= reg_mem[10];
        reg_mem11 <= reg_mem[11];
        reg_mem12 <= reg_mem[12];
        reg_mem13 <= reg_mem[13];
        reg_mem14 <= reg_mem[14];

        $display(" rax=%d, rcx=%d, rdx=%d, rbx=%d, rsp=%d, rbp=%d, rsi=%d,
rdi=%d, r8=%d, r9=%d, r10=%d, r11=%d, r12=%d, r13=%d, r14=%d\n",
reg_mem[0],reg_mem[1],reg_mem[2],reg_mem[3],reg_mem[4],reg_mem[5],reg_mem[6
],reg_mem[7],reg_mem[8],reg_mem[9],reg_mem[10],reg_mem[11],reg_mem[12],reg_
mem[13],reg_mem[14]);

    end

 endmodule
```

PC UPDATE:

The PC is set to address of next instruction or valP.

CODE:

```
`timescale 1ns / 1ps

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

```
    input clk;
    input [3:0] icode;
    input [63:0] valP;
    input [63:0] valC;
    input [63:0] valM;
    input wire cond;
    output reg [63:0] PC;

    always @(*) begin
        #500
        case(icode)

            4'b0000, 4'b0001, 4'b0010, 4'b0011, 4'b0100, 4'b0101, 4'b0110,
4'b1010, 4'b1011: begin //halt, nop, cmovxx, irmov, rmmov, mrmov, opq,
pushq, popq
                #5
                PC = valP;
            end

            4'b0111:begin //jXX
                if(cond == 1)
                    #5
                    PC = valC;
                else
                    #5
                    PC = valP;
            end

            4'b1000:begin //call
                #5
                PC = valC;
            end

            4'b1001:begin //ret
                #5
                PC = valM;
            end
        endcase

        //if((^PC === 1'bX) && clk==1 ) begin
        //        $display("PC reached undefined");
        //        //$finish;
        //end
    end
endmodule
```

CODE for wrapper:

```
`timescale 1ns / 1ps

`include "fetch.v"
`include "decode.v"
`include "execute.v"
`include "memory.v"
`include "writeback.v"
`include "pc_update.v"

module proc_wrapper;
```

```
reg clk;
reg [63:0] PC;
reg stat[0:2];

wire [3:0] icode;
wire [3:0] ifun;
wire [3:0] rA;
wire [3:0] rB;
wire [63:0] valC;
wire [63:0] valP;
wire instr_valid;
wire dmem_error;
wire [63:0] valA;
wire [63:0] valB;
wire [63:0] valE;
wire [63:0] valM;
wire cond;
wire hlt;
wire [63:0] updated_pc;

wire [63:0] reg_mem0;
wire [63:0] reg_mem1;
wire [63:0] reg_mem2;
wire [63:0] reg_mem3;
wire [63:0] reg_mem4;
wire [63:0] reg_mem5;
wire [63:0] reg_mem6;
wire [63:0] reg_mem7;
wire [63:0] reg_mem8;
wire [63:0] reg_mem9;
wire [63:0] reg_mem10;
wire [63:0] reg_mem11;
wire [63:0] reg_mem12;
wire [63:0] reg_mem13;
wire [63:0] reg_mem14;

wire sf;
wire zf;
wire of;


fetch fetch(
.clk(clk),
.PC(PC),
.icode(icode),
.ifun(ifun),
.rA(rA),
.rB(rB),
.valC(valC),
.valP(valP),
.instr_valid(instr_valid),
.imem_error(imem_error),
.hlt(hlt)
);

decode decode(
.clk(clk),
.icode(icode),
.rA(rA),
.rB(rB),
.valA(valA),
.valB(valB),
.reg_mem0(reg_mem0),
.reg_mem1(reg_mem1),
.reg_mem2(reg_mem2),
.reg_mem3(reg_mem3),
.reg_mem4(reg_mem4),
.reg_mem5(reg_mem5),
.reg_mem6(reg_mem6),
.reg_mem7(reg_mem7),
.reg_mem8(reg_mem8),
.reg_mem9(reg_mem9),
.reg_mem10(reg_mem10),
.reg_mem11(reg_mem11),
.reg_mem12(reg_mem12),
.reg_mem13(reg_mem13),
.reg_mem14(reg_mem14)
);

execute execute(
.clk(clk),
.valA(valA),
.valB(valB),
.valC(valC),
.ifun(ifun),
.icode(icode),
.sf(sf),
.zf(zf),
.of(of),
.valE(valE),
.cond(cond)
);

memory memory(
.clk(clk),
.icode(icode),
.valA(valA),
.valE(valE),
.valP(valP),
.valM(valM),
.dmem_error(dmem_error)
);

writeback wb(
.clk(clk),
.cond(cond),
.icode(icode),
.rA(rA),
.rB(rB),
.valE(valE),
.valM(valM),
.reg_mem0(reg_mem0),
.reg_mem1(reg_mem1),
.reg_mem2(reg_mem2),
.reg_mem3(reg_mem3),
.reg_mem4(reg_mem4),
.reg_mem5(reg_mem5),
.reg_mem6(reg_mem6),
.reg_mem7(reg_mem7),
.reg_mem8(reg_mem8),
.reg_mem9(reg_mem9),
.reg_mem10(reg_mem10),
    .reg_mem11(reg_mem11),
    .reg_mem12(reg_mem12),
    .reg_mem13(reg_mem13),
    .reg_mem14(reg_mem14)
    );

    pc_update pcup(
    .clk(clk),
    .icode(icode),
    .valP(valP),
    .valC(valC),
    .valM(valM),
    .cond(cond),
    .PC(updated_pc)
    );

    always #2000 clk=~clk;

    initial begin
        $dumpfile("wrapper.vcd");
        stat[0] = 1'b0; //aok
        stat[1] = 1'b0; //inst_valid
        stat[2] = 1'b0; //halt

        clk = 0;
        PC = 64'b0;
    end

    always@(posedge clk)begin
        #30
        PC = updated_pc;
        if(hlt ==1)
        begin
            stat[0] = 1'b0; //aok
            stat[1] = 1'b0; //inst_valid
            stat[2] = 1'b1; //halt
        end

        else if(instr_valid==0)
        begin
            stat[0] = 1'b0; //aok
            stat[1] = 1'b1; //inst_valid
            stat[2] = 1'b0; //halt
        end

        else
        begin
            stat[0] = 1'b1; //aok
            stat[1] = 1'b0; //inst_valid
            stat[2] = 1'b0; //halt
        end

        if(stat[2]==1'b1)
        begin
            $finish;
        end
        $display("clk=%d icode=%d ifun=%d rA=%d rB=%d rsp=%d valA=%d
valB=%d valE=%d valP=%d valC=%d halt=%d PC=%d dmem_error=%d \n", clk,
icode, ifun, rA, rB, reg_mem4, valA, valB, valE, valP, valC, stat[2], PC,
dmem_error);
```

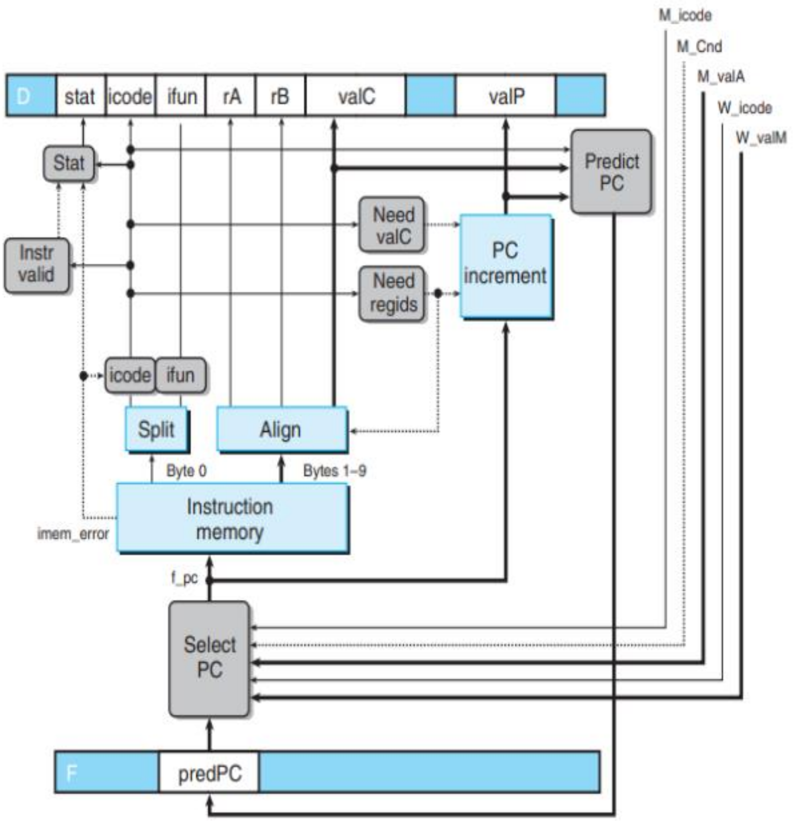
```
    end

endmodule
```

5-Stage Pipeline:

Fetch and PC prediction:

In this stage we fetch the instruction from main memory and then also predict PC. Within the one cycle time limit , the processor can only predict the address of the next instruction.

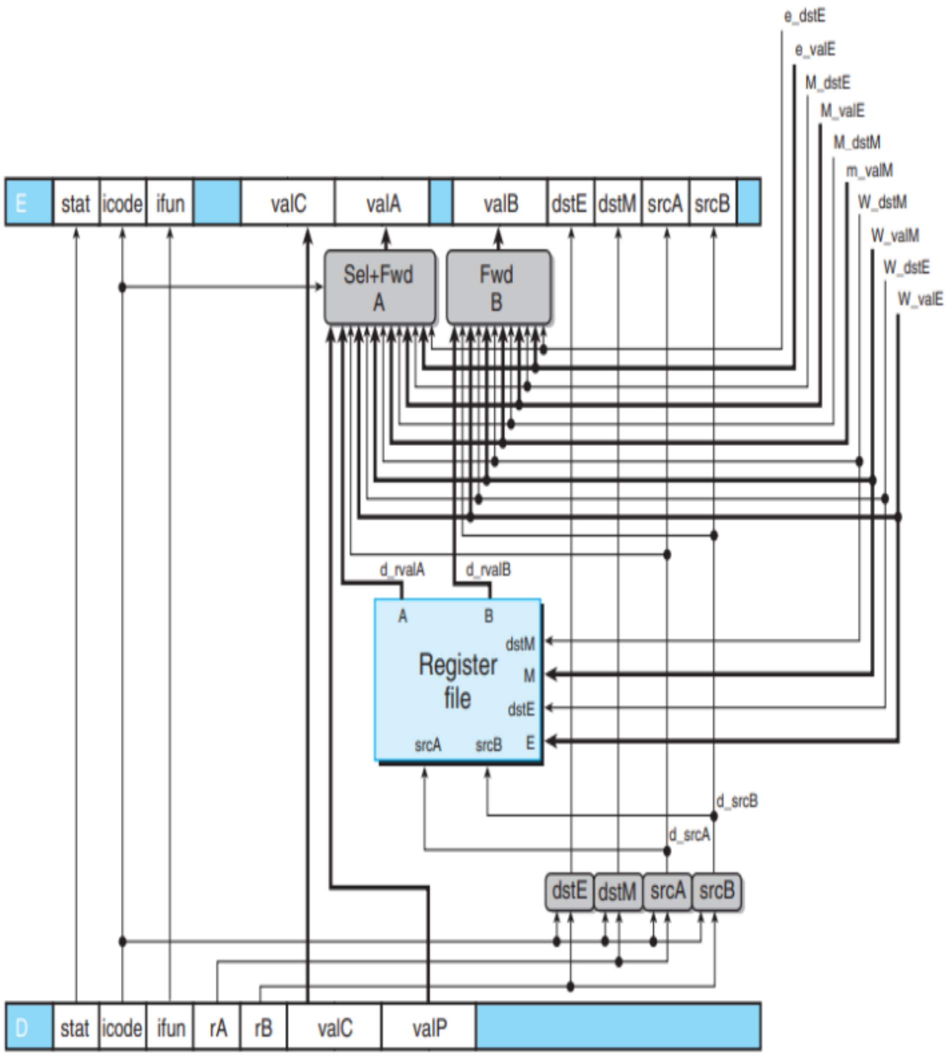


The fetch stage in a Y86-64 bit processor is responsible for retrieving the next instruction from memory and preparing it for execution . The fetch stage works as

- The program counter (PC) holds the address of the next instruction to be fetched.

- The fetch stage sends a read request to the memory subsystem at the address held by the PC.
- The instruction is retrieved from memory and stored in a buffer called the instruction register (IR).
- The PC is incremented to point to the next instruction.
- The fetched instruction is then passed on to the next stage of the pipeline, which is typically the decode stage.
- The PC selection logic chooses between three program counter sources. As a mispredicted branch enters the memory stage, the value of `valP` for this instruction (indicating the address of the following instruction) is read from pipeline register M (signal `M_valA`). When a ret instruction enters the write-back stage, the return address is read from pipeline register W (signal `W_valM`). All other cases use the predicted value of the PC, stored in pipeline register F (signal `F_predPC`).
- Overall, the fetch stage is responsible for reading instructions from memory and making them available for execution by the processor. Overall one of the major difference between `fetch` stage execution in a sequential model and this pipeline model is that in this model we move the PC update stage so that its logic is active at the beginning of the clock cycle by making it compute the PC value for the current instruction.

DECODE:



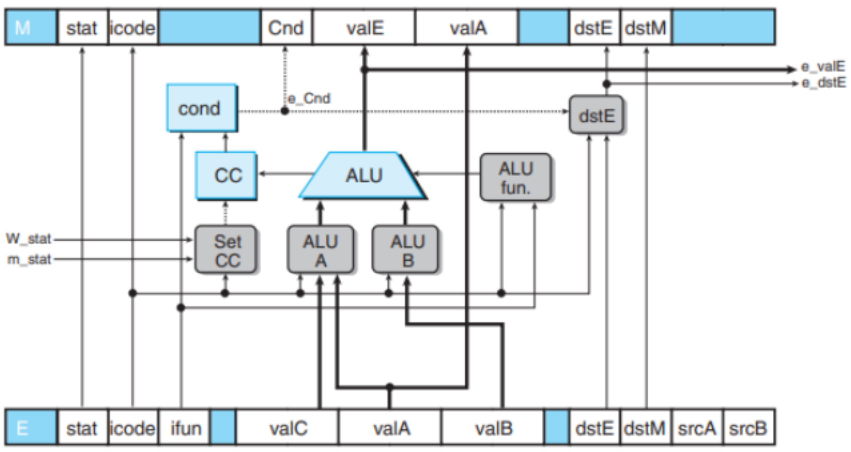
The decode stage is responsible for decoding the instruction fetched in the previous cycle and preparing the operands for the execution stage. The decode stage works as follows:

- The instruction that was fetched in the previous cycle is loaded from the instruction register (IR) into the decode register (DR).
- The opcode of the instruction is extracted from the instruction and decoded to determine the type of instruction and the operands that are needed.
- The registers that are specified as operands in the instruction are read from the register file and their values are passed on to the execution stage.

- If the instruction involves a memory access, the address of the memory location is computed based on the operands and passed on to the execution stage.
- Control signals are generated based on the instruction type and passed on to the execution stage to enable the appropriate functional units.
- The decoded instruction and its operands are then passed on to the execution stage to be executed.

Pipelining's decode stage is crucial because it gets the operands ready for the execution stage so that it may start processing the instruction as soon as it becomes available. The fetch, decode, and execute phases of the CPU can be combined to boost throughput and overall performance.

EXECUTE:



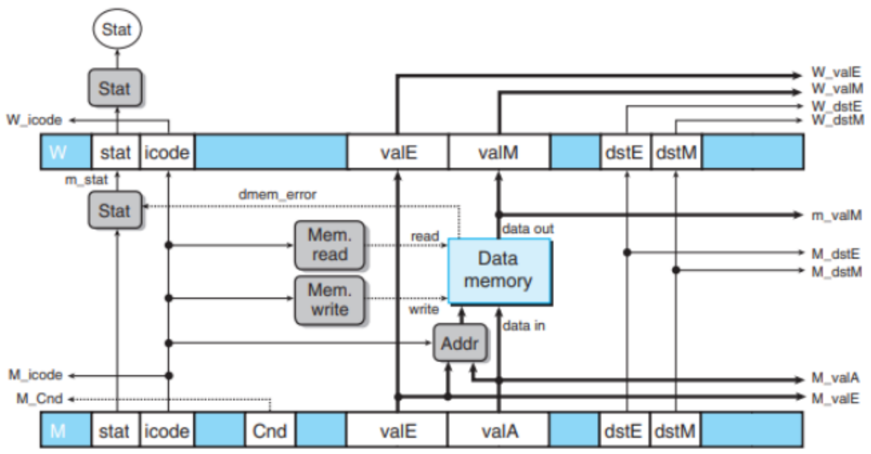
The execute stage in a Y86 64-bit processor with pipelining is responsible for carrying out the operation specified by the instruction, using the operands that were fetched and prepared in the previous stages. The execute stage works as follows:

- The instruction and its operands are received from the previous stage, typically the decode stage. The operation specified by the instruction is performed on the operands.

- If the instruction involves a memory access, the memory subsystem is accessed to read or write the data.
- If the instruction is a branch instruction, the branch condition is evaluated, and the program counter (PC) is updated accordingly.
- The result of the operation is then passed on to the next stage of the pipeline, typically the memory stage or the write-back stage.

Hazard detection and handling techniques, such as forwarding, stalling, and branch prediction, may also be used during the execution stage to ensure correct program execution in the presence of hazards

MEMORY:



The memory stage in a Y86 64-bit processor with pipelining is responsible for accessing memory to read or write data, and also for handling any memory-related hazards that may occur in the pipeline. The memory stage works as follows:

- If the instruction involves a memory access, the memory address is computed based on the operands received from the previous stage, typically the execute stage.
- A read or write request is sent to the memory subsystem to access the data at the memory location specified by the address.
- If the instruction is a load instruction, the data that is read from memory is passed on to the next stage of the pipeline, typically the write-back stage.

- If the instruction is a store instruction, the data to be written to memory is passed on to the memory subsystem.
- If a memory-related hazard occurs, such as a load-use hazard, where a later instruction depends on the data loaded by an earlier instruction, the pipeline may need to be stalled or data forwarding techniques may need to be used to resolve the hazard.

WRITE BACK:

Registers are updated in this stage. Often, instructions require registered values from previous instructions but this cannot be done until the previous instructions have gone through writeback stage. Thus it is not uncommon to stall instructions to avoid such control hazards.

=>We already explained about the registers which are introduced in between the stages.

- Codes for the each registers are:

Fetch register:

CODE:

```
module fetch_reg(clk,predit_pc,predit_pc_f);
input clk;
input [63:0]predit_pc;
output reg [63:0]predit_pc_f;

 always @(posedge clk) begin
    predit_pc_f<=predit_pc;
 end
endmodule
```

Decode register:

CODE:

```
module
decode_reg(clk,stat_f,icode_f,ifun_f,rA_f,rB_f,valc_f,valp_f,stat_d,icode_d
,ifun_d,rA_d,rB_d,valc_d,valp_d);
input clk;
input [2:0]stat_f;
input [3:0]icode_f;
input [3:0]ifun_f;
```

```
input[3:0]rA_f;
input [3:0]rB_f;
input [63:0]valc_f;
input [63:0]valp_f;
output reg [2:0]stat_d;
output reg [3:0]icode_d;
output reg [3:0]ifun_d;
output reg [3:0]rA_d;
output reg [3:0]rB_d;
output reg [63:0]valc_d;
output reg [63:0]valp_d;

 always @(posedge clk) begin

    stat_d <= stat_f;
    icode_d <= icode_f;
    ifun_d <= ifun_f;
    rA_d <= rA_f;
    rB_d <= rB_f;
    valc_d <= valc_f;
    valp_d <= valp_f;

 end
endmodule
```

EXECUTE register:

CODE:

```
module
execute_reg(clk,stat_d,icode_d,ifun_d,valc_d,valp_d,valA_d,valB_d,rA_d,rB_d
,stat_e,icode_e,ifun_e,valA_e,valB_e,rA_e,rB_e,valp_e,valc_e);
input clk;
input [2:0]stat_d;
input [3:0]icode_d;
input [3:0]ifun_d;
input [3:0]rA_d;
input [3:0]rB_d;
input [63:0]valA_d;
input [63:0]valB_d;
input [63:0]valc_d;
input [63:0]valp_d;
output reg[2:0]stat_e;
output reg[3:0]icode_e;
output reg[3:0]ifun_e;
output reg[3:0]rA_e;
output reg [3:0]rB_e;
output reg[63:0]valc_e;
output reg[63:0]valp_e;
output reg[63:0]valA_e;
output reg [63:0]valB_e;

always @(posedge clk) begin

    stat_e <= stat_d;
    icode_e <= icode_d;
    ifun_e <= ifun_d;
    rA_e <= rA_d;
    rB_e <= rB_d;
    valc_e <= valc_d;
    valp_e <= valp_d;
    valA_e <= valA_d;
    valB_e <= valB_d;
end
endmodule
```

MEMORY Register:

CODE:

```
module
memory_reg(clk,stat_e,icode_e,ifun_e,cnd_e,valE_e,valA_e,valB_e,valc_e,valp
_e,rA_e,rB_e,stat_m,icode_m,ifun_m,valp_m,valc_m,valA_m,valB_m,valE_m,cnd_m
,rA_m,rB_m);
input clk;
input [2:0]stat_e;
input [3:0]icode_e;
input [3:0]ifun_e;
input [3:0]rA_e;
input [3:0]rB_e;
input cnd_e;
input [63:0]valA_e;
input [63:0]valB_e;
input [63:0]valc_e;
input [63:0]valp_e;
input [63:0]valE_e;
output reg [2:0]stat_m;
output reg [3:0]icode_m;
output reg [3:0]ifun_m;
output reg [3:0]rA_m;
output reg [3:0]rB_m;
output reg cnd_m;
output reg [63:0]valA_m;
output reg [63:0]valB_m;
output reg [63:0]valc_m;
output reg [63:0]valp_m;
output reg [63:0]valE_m;

always @(posedge clk) begin
    stat_m <= stat_e;
    icode_m <= icode_e;
    ifun_m <= ifun_e;
    rA_m <= rA_e;
    rB_m <= rB_e;
    cnd_m <= cnd_e;
    valA_m <= valA_e;
    valB_m <= valB_e;
    valp_m <= valp_e;
    valc_m <= valc_e;
    valE_m <= valE_e;
end
endmodule
```

Write back register:

Code:

```
module
writeback_reg(clk,stat_m,icode_m,valA_m,valB_m,rA_m,rB_m,valc_m,valp_m,valE
_m,valM_m,stat_w,icode_w,ifun_w,valA_w,valB_w,valc_w,valp_w,valE_w,rA_w,rB_
w,valM_w);
input clk;
input [2:0]stat_m;
input [3:0]icode_m;
input [3:0]ifun_m;
input cnd_m;
input [63:0]valA_m;
input [63:0]valB_m;
input [63:0]valc_m;
input [63:0]valE_m;
input [63:0]valM_m;
input [3:0]rA_m;
input [3:0]rB_m;
input [63:0]valp_m;
output reg[2:0]stat_w;
output reg[3:0]icode_w;
output reg [3:0]ifun_w;
output reg cnd_w;
output reg [3:0]rA_w;
output reg [3:0]rB_w;
output reg [63:0]valA_w;
output reg [63:0]valB_w;
output reg [63:0]valc_w;
output reg [63:0]valE_w;
output reg [63:0]valM_w;
output reg [63:0]valp_w;

always @(posedge clk) begin

    stat_w <= stat_m;
    icode_w <= icode_m;
    ifun_w <= ifun_m;
    rA_w <= rA_m;
    rB_w <= rB_m;
    valA_w <= valA_m;
    valB_w <= valB_m;
    valc_w <= valc_m;
    valE_w <= valE_m;
    valM_w <= valM_m;
    valp_w <= valp_m;
end
endmodule
```

WRAPPER:

Code:

```
`timescale 1ns / 1ps

`include "fetch_reg.v"
`include "decode_reg.v"
`include "execute_reg.v"
`include "memory_reg.v"
`include "writeback_reg.v"
`include "fetch.v"
`include "decode.v"
`include "execute.v"
`include "memory.v"
```

```
`include "writeback.v"
`include "pc_update.v"

module proc_wrapper;

    reg clk;
    reg [63:0] PC;
    reg stat[0:2];

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

    wire [2:0]stat_f;
    wire [3:0]icode_f;
    wire [3:0]ifun_f;
    wire[3:0]rA_f;
    wire [3:0]rB_f;
    wire [63:0]valc_f;
    wire [63:0]valp_f;
    wire instr_valid;
    wire imem_error;
    wire hlt;

    wire [2:0]stat_d;
    wire [3:0]icode_d;
    wire [3:0]ifun_d;
    wire [3:0]rA_d;
    wire [3:0]rB_d;
    wire [63:0]valA_d;
    wire [63:0]valB_d;
    wire [63:0]valc_d;
    wire [63:0]valp_d;

    wire [2:0]stat_e;
    wire [3:0]icode_e;
    wire [3:0]ifun_e;
    wire [3:0]rA_e;
    wire [3:0]rB_e;
    wire [63:0]valc_e;
    wire [63:0]valp_e;
    wire [63:0]valA_e;
    wire [63:0]valB_e;
    wire [63:0]valE_e;
    wire cnd_e;

    wire [2:0]stat_m;
    wire [3:0]icode_m;
    wire [3:0]ifun_m;
    wire [3:0]rA_m;
    wire [3:0]rB_m;
    wire cnd_m;
    wire [63:0]valA_m;
    wire [63:0]valB_m;
    wire [63:0]valc_m;
    wire [63:0]valp_m;
    wire [63:0]valE_m;
    wire [63:0]valM_m;

    wire [2:0]stat_w;
    wire [3:0]icode_w;
    wire cnd_w;
    wire [3:0]rA_w;
    wire [3:0]rB_w;
    wire [63:0]valA_w;
    wire [63:0]valB_w;
    wire [63:0]valc_w;
    wire [63:0]valE_w;
    wire [63:0]valM_w;
    wire [63:0]valp_w;


    wire [63:0] reg_mem0;
    wire [63:0] reg_mem1;
    wire [63:0] reg_mem2;
    wire [63:0] reg_mem3;
    wire [63:0] reg_mem4;
    wire [63:0] reg_mem5;
    wire [63:0] reg_mem6;
    wire [63:0] reg_mem7;
    wire [63:0] reg_mem8;
    wire [63:0] reg_mem9;
    wire [63:0] reg_mem10;
    wire [63:0] reg_mem11;
    wire [63:0] reg_mem12;
    wire [63:0] reg_mem13;
    wire [63:0] reg_mem14;

    fetch_reg freg(
        .clk(clk),
        .predit_pc(valp_f),
        .predit_pc_f(predit_pc_f)
    );

    decode_reg dreg(
        .clk(clk),

        .stat_f(stat_f),
        .icode_f(icode_f),
        .ifun_f(ifun_f),
        .rA_f(rA_f),
        .rB_f(rB_f),
        .valc_f(valc_f),
        .valp_f(valp_f),

        .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)
    );

    execute_reg ereg(
        .clk(clk),

        .stat_d(stat_d),
        .icode_d(icode_d),
        .ifun_d(ifun_d),
        .rA_d(rA_d),
        .rB_d(rB_d),
        .valA_d(valA_d),
            .valB_d(valB_d),
            .valc_d(valc_d),
            .valp_d(valp_d),

            .stat_e(stat_e),
            .icode_e(icode_e),
            .ifun_e(ifun_e),
            .rA_e(rA_e),
            .rB_e(rB_e),
            .valA_e(valA_e),
            .valB_e(valB_e),
            .valc_e(valc_e),
            .valp_e(valp_e)
        );

        memory_reg mreg(
            .clk(clk),

            .stat_e(stat_e),
            .icode_e(icode_e),
            .ifun_e(ifun_e),
            .rA_e(rA_e),
            .rB_e(rB_e),
            .cnd_e(cnd_e),
            .valA_e(valA_e),
            .valB_e(valB_e),
            .valc_e(valc_e),
            .valp_e(valp_e),
            .valE_e(valE_e),

            .stat_m(stat_m),
            .icode_m(icode_m),
            .ifun_m(ifun_m),
            .rA_m(rA_m),
            .rB_m(rB_m),
            .cnd_m(cnd_m),
            .valA_m(valA_m),
            .valB_m(valB_m),
            .valc_m(valc_m),
            .valp_m(valp_m),
            .valE_m(valE_m)
        );

        writeback_reg wreg(
            .clk(clk),

            .stat_m(stat_m),
            .icode_m(icode_m),
            .valA_m(valA_m),
            .valB_m(valB_m),
            .valc_m(valc_m),
            .valE_m(valE_m),
            .valM_m(valM_m),
            .rA_m(rA_m),
            .rB_m(rB_m),
            .valp_m(valp_m),

            .stat_w(stat_w),
            .icode_w(icode_w),
            .rA_w(rA_w),
            .rB_w(rB_w),
            .valA_w(valA_w),
            .valB_w(valB_w),
            .valc_w(valc_w),
            .valE_w(valE_w),
            .valM_w(valM_w),
            .valp_w(valp_w)
        );

        pc_update pcup(
        .clk(clk),

        .icode(icode_w),
        .valP(predit_pc_f),
        .valC(valc_w),
        .valM(valM_w),
        .cond(cnd_w),
        .PC(updated_pc)
        );


        fetch fetch(
        .clk(clk),

        .PC(PC),
        .icode(icode_f),
        .ifun(ifun_f),
        .rA(rA_f),
        .rB(rB_f),
        .valC(valc_f),
        .valP(valp_f),
        .instr_valid(instr_valid),
        .imem_error(imem_error),
        .hlt(hlt)
        );

        decode decode(
        .clk(clk),
        .icode(icode_d),
        .rA(rA_d),
        .rB(rB_d),
        .valA(valA_d),
        .valB(valB_d),
        .reg_mem0(reg_mem0),
        .reg_mem1(reg_mem1),
        .reg_mem2(reg_mem2),
        .reg_mem3(reg_mem3),
        .reg_mem4(reg_mem4),
        .reg_mem5(reg_mem5),
        .reg_mem6(reg_mem6),
        .reg_mem7(reg_mem7),
        .reg_mem8(reg_mem8),
        .reg_mem9(reg_mem9),
        .reg_mem10(reg_mem10),
        .reg_mem11(reg_mem11),
        .reg_mem12(reg_mem12),
        .reg_mem13(reg_mem13),
        .reg_mem14(reg_mem14)
        );

        execute execute(
        .clk(clk),
        .valA(valA_e),
        .valB(valB_e),
        .valC(valc_e),
        .ifun(ifun_e),
        .icode(icode_e),
        .sf(sf),
        .zf(zf),
        .of(of),
        .valE(valE_e),
        .cond(cond_e)
        );

        memory memory(
        .clk(clk),
        .icode(icode_m),
        .valA(valA_m),
        .valE(valE_m),
        .valP(valp_m),
        .valM(valM_m),
        .dmem_error(dmem_error)
        );

        writeback wb(
        .clk(clk),
        .cond(cond_w),
        .icode(icode_w),
        .rA(rA_w),
        .rB(rB_w),
        .valE(valE_w),
        .valM(valM_w),
        .reg_mem0(reg_mem0),
        .reg_mem1(reg_mem1),
        .reg_mem2(reg_mem2),
        .reg_mem3(reg_mem3),
        .reg_mem4(reg_mem4),
        .reg_mem5(reg_mem5),
        .reg_mem6(reg_mem6),
        .reg_mem7(reg_mem7),
        .reg_mem8(reg_mem8),
        .reg_mem9(reg_mem9),
        .reg_mem10(reg_mem10),
        .reg_mem11(reg_mem11),
        .reg_mem12(reg_mem12),
        .reg_mem13(reg_mem13),
        .reg_mem14(reg_mem14)
        );

        always #2000 clk=~clk;

        initial begin
            $dumpfile("wrapper.vcd");
            stat[0] = 1'b0; //aok
            stat[1] = 1'b0; //inst_valid
            stat[2] = 1'b0; //halt

            clk = 0;
            PC = 64'b0;
        end

        always@(posedge clk)begin
            #30
```

```
        PC = updated_pc;
        if(hlt ==1)
        begin
            stat[0] = 1'b0; //aok
            stat[1] = 1'b0; //inst_valid
            stat[2] = 1'b1; //halt
        end

        else if(instr_valid==0)
        begin
            stat[0] = 1'b0; //aok
            stat[1] = 1'b1; //inst_valid
            stat[2] = 1'b0; //halt
        end

        else
        begin
            stat[0] = 1'b1; //aok
            stat[1] = 1'b0; //inst_valid
            stat[2] = 1'b0; //halt
        end

        if(stat[2]==1'b1)
        begin
            $finish;
        end
        //$display("clk=%d icode=%d ifun=%d rA=%d rB=%d rsp=%d valA=%d
valB=%d valE=%d valP=%d valC=%d halt=%d PC=%d dmem_error=%d \n", clk,
icode, ifun, rA, rB, reg_mem4, valA, valB, valE, valP, valC, stat[2], PC,
dmem_error);
        //$display("clk=%d f=%d d=%d e=%d m=%d wb=%d",clk,icode_f,icode_d,
icode_e, icode_m, icode_w);
        //$display("0=%d 1=%d 2=%d 3=%d 4=%d zf=%d sf=%d
of=%d\2n",reg_mem0,reg_mem1,reg_mem2,reg_mem3,reg_mem4,zf,sf,of);
    end


endmodule
```