Intro to Processor Architecture

Project Report

Arya Yogesh Topale 2022102052

Rohan Sridhar 2022102075

Contents

1. Overview/Objective:

The fundamental aim of this project is to create a processor architecture design using Verilog, which follows the Y86-64 ISA. The processor's capability should encompass the execution of all instructions within the Y86-64 ISA. Our final objective is to develop a 5-stage pipelined implementation of the Y86-64 architecture.

2. Sequential Design:

- **Fetch Stage:-**

In the fetch stage, we are supposed to read instructions from an instruction memory(which is an array of registers). By reading these instructions we can find icode, ifun, rA, rB and valC.

If the PC value (that we received at the positive edge of the clock cycle), is out of bounds, we get a memory error(imem_error).

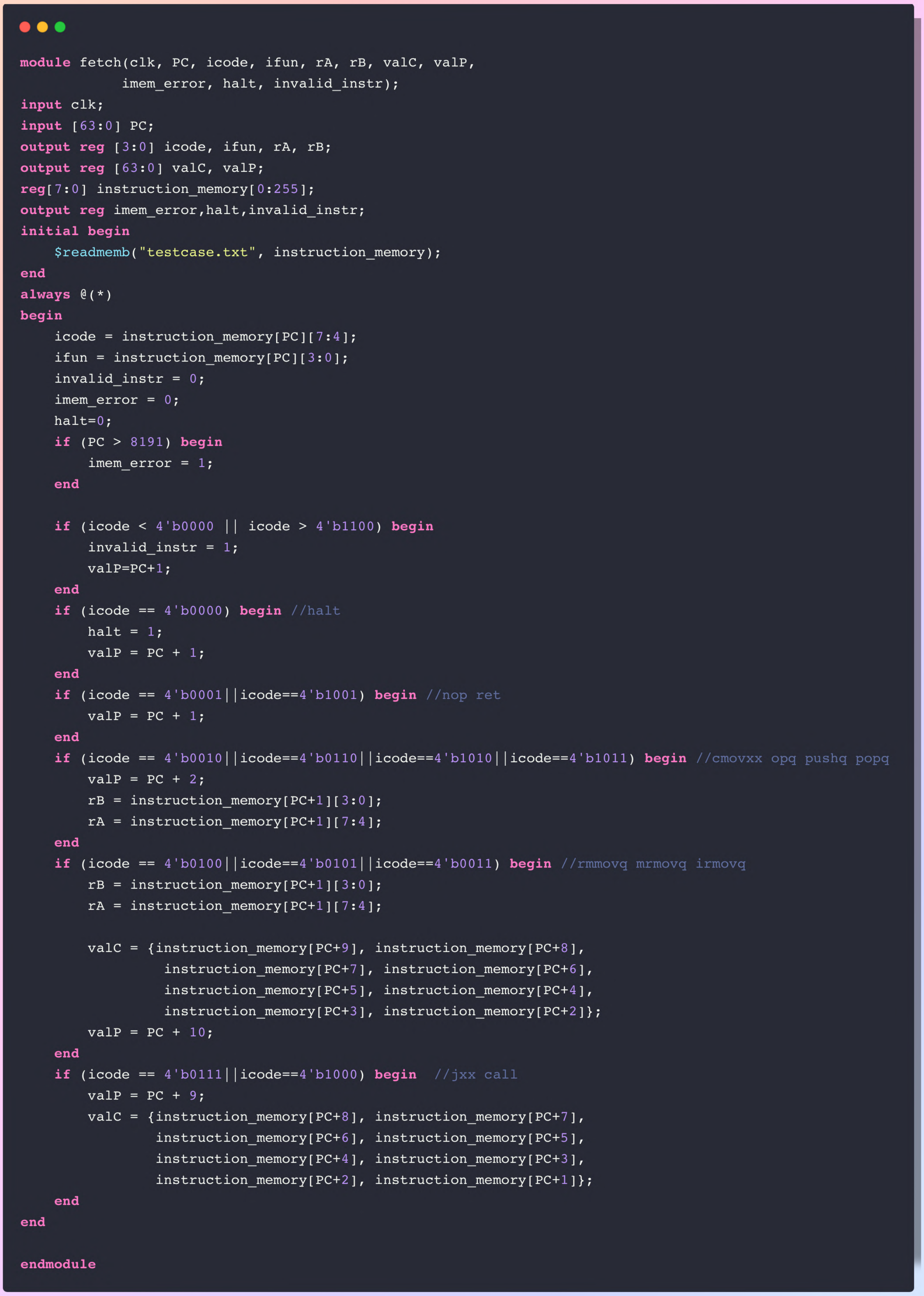
Using icode, that we have obtained from memory, we can tell which instruction to execute and can find the value of valP.

For halt, nop and ret instruction: valP=PC+1

For cmovxx, opq, pushq, popq: valP=PC+2

For rmmovq, mrmovq and irmovq: valP=PC+10 and valC is the last 8 bytes of the instruction memory.

For conditional jumps and call: valP=PC+9 and valC is again the last 8 bytes of the instruction memory.



```verilog
module fetch(clk, PC, icode, ifun, rA, rB, valC, valP,
            imem_error, halt, invalid_instr);
input clk;
input [63:0] PC;
output reg [3:0] icode, ifun, rA, rB;
output reg [63:0] valC, valP;
reg[7:0] instruction_memory[0:255];
output reg imem_error,halt,invalid_instr;
initial begin
    $readmemb("testcase.txt", instruction_memory);
end
always @(*)
begin
    icode = instruction_memory[PC][7:4];
    ifun = instruction_memory[PC][3:0];
    invalid_instr = 0;
    imem_error = 0;
    halt=0;
    if (PC > 8191) begin
        imem_error = 1;
    end

    if (icode < 4'b0000 || icode > 4'b1100) begin
        invalid_instr = 1;
        valP=PC+1;
    end
    if (icode == 4'b0000) begin //halt
        halt = 1;
        valP = PC + 1;
    end
    if (icode == 4'b0001||icode==4'b1001) begin //nop ret
        valP = PC + 1;
    end
    if (icode == 4'b0010||icode==4'b0110||icode==4'b1010||icode==4'b1011) begin //cmovxx opq pushq popq
        valP = PC + 2;
        rB = instruction_memory[PC+1][3:0];
        rA = instruction_memory[PC+1][7:4];
    end
    if (icode == 4'b0100||icode==4'b0101||icode==4'b0011) begin //rmmovq mrmovq irmovq
        rB = instruction_memory[PC+1][3:0];
        rA = instruction_memory[PC+1][7:4];

        valC = {instruction_memory[PC+9], instruction_memory[PC+8],
                 instruction_memory[PC+7], instruction_memory[PC+6],
                 instruction_memory[PC+5], instruction_memory[PC+4],
                 instruction_memory[PC+3], instruction_memory[PC+2]};
        valP = PC + 10;
    end
    if (icode == 4'b0111||icode==4'b1000) begin  //jxx call
        valP = PC + 9;
        valC = {instruction_memory[PC+8], instruction_memory[PC+7],
                instruction_memory[PC+6], instruction_memory[PC+5],
                instruction_memory[PC+4], instruction_memory[PC+3],
                instruction_memory[PC+2], instruction_memory[PC+1]};
    end
end

endmodule
```

• <u>**Decode Stage:-**</u>

In the decode stage, we are required to read from the register files and assign those values to valA and valB.

The register file(reg_file) is an array of length 15, containing 64-bit registers.

Depending on the values of icode(instruction type), we can decide the values of valA and valB:

<u>in cmovxx</u>: valA = reg_file[rA]

<u>in rmmovq and opq</u>: valA = reg_file[rA] and valB = reg_file[rB]

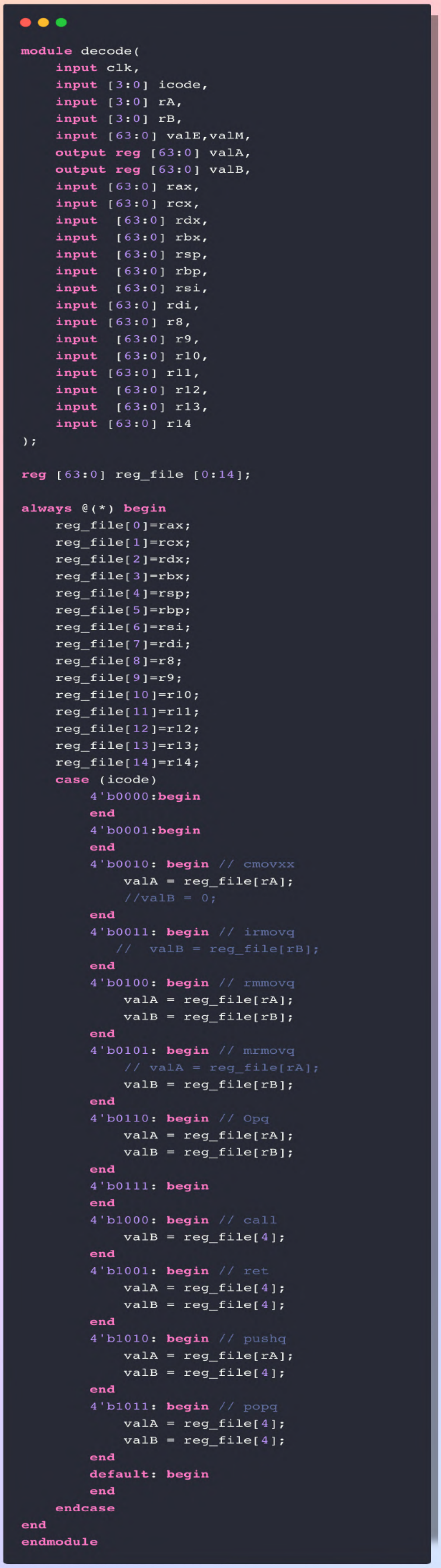
<u>in mrmovq</u>: valB = reg_file[rB]

<u>in call</u>: valB = reg_file[4]

<u>in ret and popq</u>: valA = reg_file[4] and valB = reg_file[4]

<u>in pushq</u>: valA = reg_file[rA] and valB = reg_file[4]

where reg_file[4] represents the stack pointer register %rsp.



```verilog
module decode(
    input clk,
    input [3:0] icode,
    input [3:0] rA,
    input [3:0] rB,
    input [63:0] valE,valM,
    output reg [63:0] valA,
    output reg [63:0] valB,
    input [63:0] rax,
    input [63:0] rcx,
    input  [63:0] rdx,
    input  [63:0] rbx,
    input  [63:0] rsp,
    input  [63:0] rbp,
    input  [63:0] rsi,
    input [63:0] rdi,
    input [63:0] r8,
    input  [63:0] r9,
    input  [63:0] r10,
    input [63:0] r11,
    input  [63:0] r12,
    input  [63:0] r13,
    input [63:0] r14
);

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

always @(*) begin
    reg_file[0]=rax;
    reg_file[1]=rcx;
    reg_file[2]=rdx;
    reg_file[3]=rbx;
    reg_file[4]=rsp;
    reg_file[5]=rbp;
    reg_file[6]=rsi;
    reg_file[7]=rdi;
    reg_file[8]=r8;
    reg_file[9]=r9;
    reg_file[10]=r10;
    reg_file[11]=r11;
    reg_file[12]=r12;
    reg_file[13]=r13;
    reg_file[14]=r14;
    case (icode)
        4'b0000:begin
        end
        4'b0001:begin
        end
        4'b0010: begin // cmovxx
            valA = reg_file[rA];
            //valB = 0;
        end
        4'b0011: begin // irmovq
           //  valB = reg_file[rB];
        end
        4'b0100: begin // rmmovq
            valA = reg_file[rA];
            valB = reg_file[rB];
        end
        4'b0101: begin // mrmovq
            // valA = reg_file[rA];
            valB = reg_file[rB];
        end
        4'b0110: begin // Opq
            valA = reg_file[rA];
            valB = reg_file[rB];
        end
        4'b0111: begin
        end
        4'b1000: begin // call
            valB = reg_file[4];
        end
        4'b1001: begin // ret
            valA = reg_file[4];
            valB = reg_file[4];
        end
        4'b1010: begin // pushq
            valA = reg_file[rA];
            valB = reg_file[4];
        end
        4'b1011: begin // popq
            valA = reg_file[4];
            valB = reg_file[4];
        end
        default: begin
        end
    endcase
end
endmodule
```

- **Execute Stage:-**

In the execute stage, we require the use of the ALU. Here along with icode, ifun is also needed.

valE is given different values according to the instruction type(depending on values of icode and ifun)

for irmovq: valE = valC

for rmmovq and mrmovq: valE=valB+valC

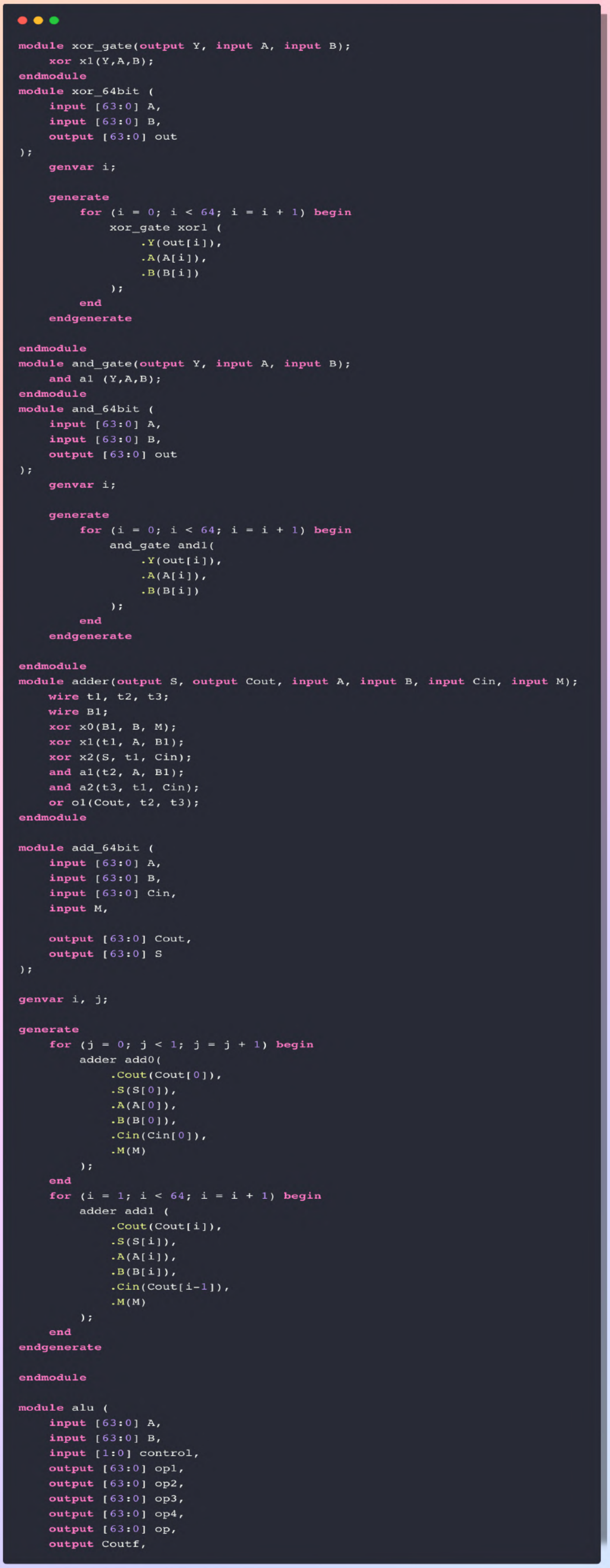
for cmovxx: we check the move conditions and set cnd to 1 if true, otherwise set it to 0

for opq: valE= valB op valA, and op depends on the ifun values.

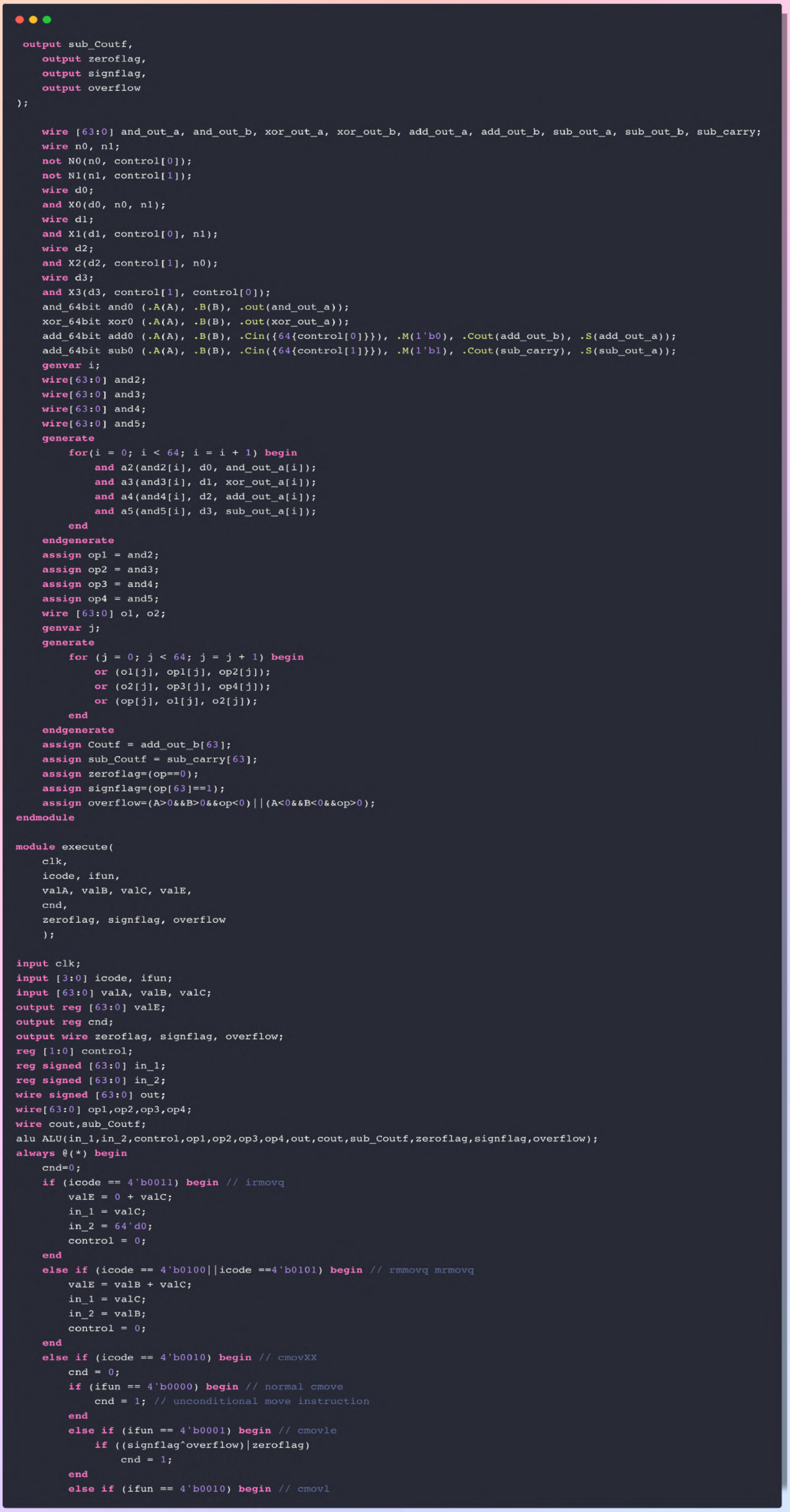
for jxx: similar to cmovxx, we check the jump conditions and set cnd to 1 if true, otherwise set it to 0

for call and pushq: valE=valB-8

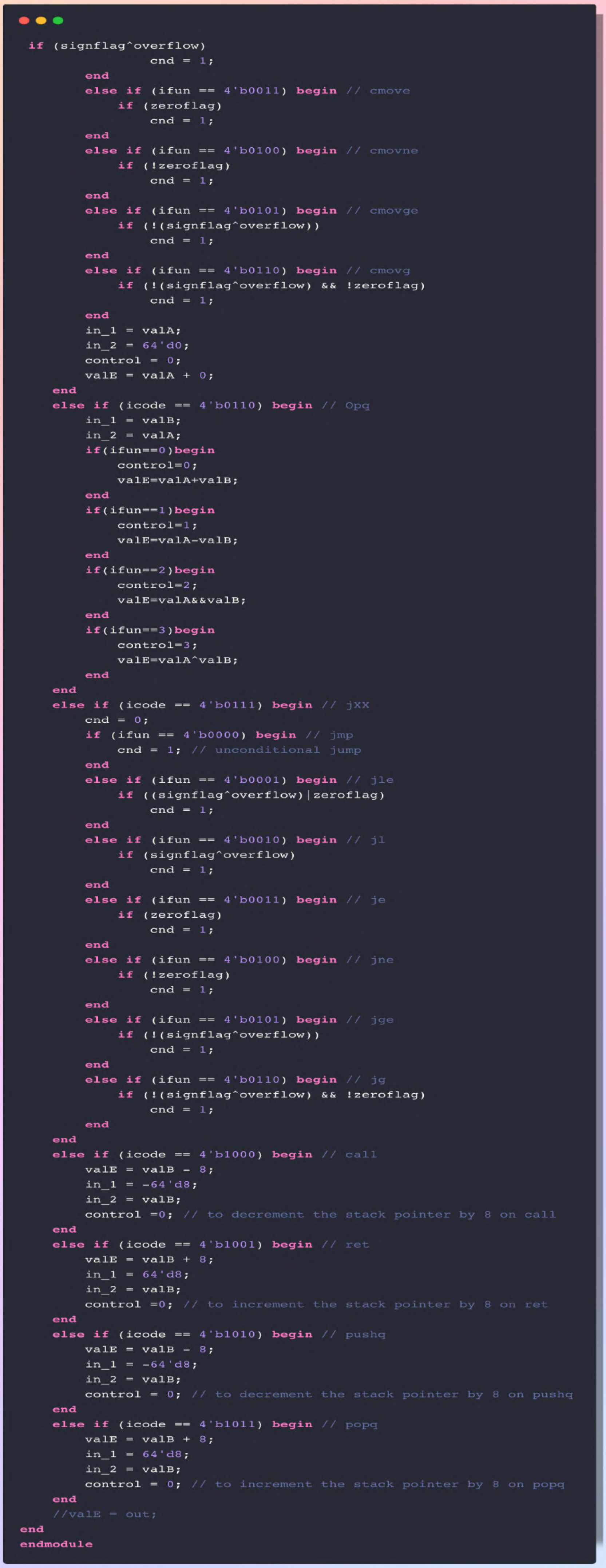
for ret and popq: valE=valB+8



```
module xor_gate(output Y, input A, input B);
    xor x1(Y,A,B);
endmodule
module xor_64bit (
    input [63:0] A,
    input [63:0] B,
    output [63:0] out
);
    genvar i;

    generate
        for (i = 0; i < 64; i = i + 1) begin
            xor_gate xor1 (
                .Y(out[i]),
                .A(A[i]),
                .B(B[i])
            );
        end
    endgenerate

endmodule
module and_gate(output Y, input A, input B);
    and a1 (Y,A,B);
endmodule
module and_64bit (
    input [63:0] A,
    input [63:0] B,
    output [63:0] out
);
    genvar i;

    generate
        for (i = 0; i < 64; i = i + 1) begin
            and_gate and1(
                .Y(out[i]),
                .A(A[i]),
                .B(B[i])
            );
        end
    endgenerate

endmodule
module adder(output S, output Cout, input A, input B, input Cin, input M);
    wire t1, t2, t3;
    wire B1;
    xor x0(B1, B, M);
    xor x1(t1, A, B1);
    xor x2(S, t1, Cin);
    and a1(t2, A, B1);
    and a2(t3, t1, Cin);
    or o1(Cout, t2, t3);
endmodule

module add_64bit (
    input [63:0] A,
    input [63:0] B,
    input [63:0] Cin,
    input M,

    output [63:0] Cout,
    output [63:0] S
);

genvar i, j;

generate
    for (j = 0; j < 1; j = j + 1) begin
        adder add0(
            .Cout(Cout[0]),
            .S(S[0]),
            .A(A[0]),
            .B(B[0]),
            .Cin(Cin[0]),
            .M(M)
        );
    end
    for (i = 1; i < 64; i = i + 1) begin
        adder add1 (
            .Cout(Cout[i]),
            .S(S[i]),
            .A(A[i]),
            .B(B[i]),
            .Cin(Cout[i-1]),
            .M(M)
        );
    end
endgenerate

endmodule

module alu (
    input [63:0] A,
    input [63:0] B,
    input [1:0] control,
    output [63:0] op1,
    output [63:0] op2,
    output [63:0] op3,
    output [63:0] op4,
    output [63:0] op,
    output Coutf,
```



```
 output sub_Coutf,
    output zeroflag,
    output signflag,
    output overflow
);

    wire [63:0] and_out_a, and_out_b, xor_out_a, xor_out_b, add_out_a, add_out_b, sub_out_a, sub_out_b, sub_carry;
    wire n0, n1;
    not N0(n0, control[0]);
    not N1(n1, control[1]);
    wire d0;
    and X0(d0, n0, n1);
    wire d1;
    and X1(d1, control[0], n1);
    wire d2;
    and X2(d2, control[1], n0);
    wire d3;
    and X3(d3, control[1], control[0]);
    and_64bit and0 (.A(A), .B(B), .out(and_out_a));
    xor_64bit xor0 (.A(A), .B(B), .out(xor_out_a));
    add_64bit add0 (.A(A), .B(B), .Cin({64{control[0]}}), .M(1'b0), .Cout(add_out_b), .S(add_out_a));
    add_64bit sub0 (.A(A), .B(B), .Cin({64{control[1]}}), .M(1'b1), .Cout(sub_carry), .S(sub_out_a));
    genvar i;
    wire[63:0] and2;
    wire[63:0] and3;
    wire[63:0] and4;
    wire[63:0] and5;
    generate
        for(i = 0; i < 64; i = i + 1) begin
            and a2(and2[i], d0, and_out_a[i]);
            and a3(and3[i], d1, xor_out_a[i]);
            and a4(and4[i], d2, add_out_a[i]);
            and a5(and5[i], d3, sub_out_a[i]);
        end
    endgenerate
    assign op1 = and2;
    assign op2 = and3;
    assign op3 = and4;
    assign op4 = and5;
    wire [63:0] o1, o2;
    genvar j;
    generate
        for (j = 0; j < 64; j = j + 1) begin
            or (o1[j], op1[j], op2[j]);
            or (o2[j], op3[j], op4[j]);
            or (op[j], o1[j], o2[j]);
        end
    endgenerate
    assign Coutf = add_out_b[63];
    assign sub_Coutf = sub_carry[63];
    assign zeroflag=(op==0);
    assign signflag=(op[63]==1);
    assign overflow=(A>0&&B>0&&op<0)||(A<0&&B<0&&op>0);
endmodule

module execute(
    clk,
    icode, ifun,
    valA, valB, valC, valE,
    cnd,
    zeroflag, signflag, overflow
    );

input clk;
input [3:0] icode, ifun;
input [63:0] valA, valB, valC;
output reg [63:0] valE;
output reg cnd;
output wire zeroflag, signflag, overflow;
reg [1:0] control;
reg signed [63:0] in_1;
reg signed [63:0] in_2;
wire signed [63:0] out;
wire[63:0] op1,op2,op3,op4;
wire cout,sub_Coutf;
alu ALU(in_1,in_2,control,op1,op2,op3,op4,out,cout,sub_Coutf,zeroflag,signflag,overflow);
always @(*) begin
    cnd=0;
    if (icode == 4'b0011) begin // irmovq
        valE = 0 + valC;
        in_1 = valC;
        in_2 = 64'd0;
        control = 0;
    end
    else if (icode == 4'b0100||icode ==4'b0101) begin // rmmovq mrmovq
        valE = valB + valC;
        in_1 = valC;
        in_2 = valB;
        control = 0;
    end
    else if (icode == 4'b0010) begin // cmovXX
        cnd = 0;
        if (ifun == 4'b0000) begin // normal cmove
            cnd = 1; // unconditional move instruction
        end
        else if (ifun == 4'b0001) begin // cmovle
            if ((signflag^overflow)|zeroflag)
                cnd = 1;
        end
        else if (ifun == 4'b0010) begin // cmovl
```



```
 if (signflag^overflow)
                cnd = 1;
        end
        else if (ifun == 4'b0011) begin // cmove
            if (zeroflag)
                cnd = 1;
        end
        else if (ifun == 4'b0100) begin // cmovne
            if (!zeroflag)
                cnd = 1;
        end
        else if (ifun == 4'b0101) begin // cmovge
            if (!(signflag^overflow))
                cnd = 1;
        end
        else if (ifun == 4'b0110) begin // cmovg
            if (!(signflag^overflow) && !zeroflag)
                cnd = 1;
        end
        in_1 = valA;
        in_2 = 64'd0;
        control = 0;
        valE = valA + 0;
    end
    else if (icode == 4'b0110) begin // Opq
        in_1 = valB;
        in_2 = valA;
        if(ifun==0)begin
            control=0;
            valE=valA+valB;
        end
        if(ifun==1)begin
            control=1;
            valE=valA-valB;
        end
        if(ifun==2)begin
            control=2;
            valE=valA&&valB;
        end
        if(ifun==3)begin
            control=3;
            valE=valA^valB;
        end
    end
    else if (icode == 4'b0111) begin // jXX
        cnd = 0;
        if (ifun == 4'b0000) begin // jmp
            cnd = 1; // unconditional jump
        end
        else if (ifun == 4'b0001) begin // jle
            if ((signflag^overflow)|zeroflag)
                cnd = 1;
        end
        else if (ifun == 4'b0010) begin // jl
            if (signflag^overflow)
                cnd = 1;
        end
        else if (ifun == 4'b0011) begin // je
            if (zeroflag)
                cnd = 1;
        end
        else if (ifun == 4'b0100) begin // jne
            if (!zeroflag)
                cnd = 1;
        end
        else if (ifun == 4'b0101) begin // jge
            if (!(signflag^overflow))
                cnd = 1;
        end
        else if (ifun == 4'b0110) begin // jg
            if (!(signflag^overflow) && !zeroflag)
                cnd = 1;
        end
    end
    else if (icode == 4'b1000) begin // call
        valE = valB - 8;
        in_1 = -64'd8;
        in_2 = valB;
        control =0; // to decrement the stack pointer by 8 on call
    end
    else if (icode == 4'b1001) begin // ret
        valE = valB + 8;
        in_1 = 64'd8;
        in_2 = valB;
        control =0; // to increment the stack pointer by 8 on ret
    end
    else if (icode == 4'b1010) begin // pushq
        valE = valB - 8;
        in_1 = -64'd8;
        in_2 = valB;
        control = 0; // to decrement the stack pointer by 8 on pushq
    end
    else if (icode == 4'b1011) begin // popq
        valE = valB + 8;
        in_1 = 64'd8;
        in_2 = valB;
        control = 0; // to increment the stack pointer by 8 on popq
    end
    //valE = out;
end
endmodule
```

• **<u>Memory Stage:-</u>**

Data is either read from memory or written into memory in this stage, whether to read or write can be decided by icode values.

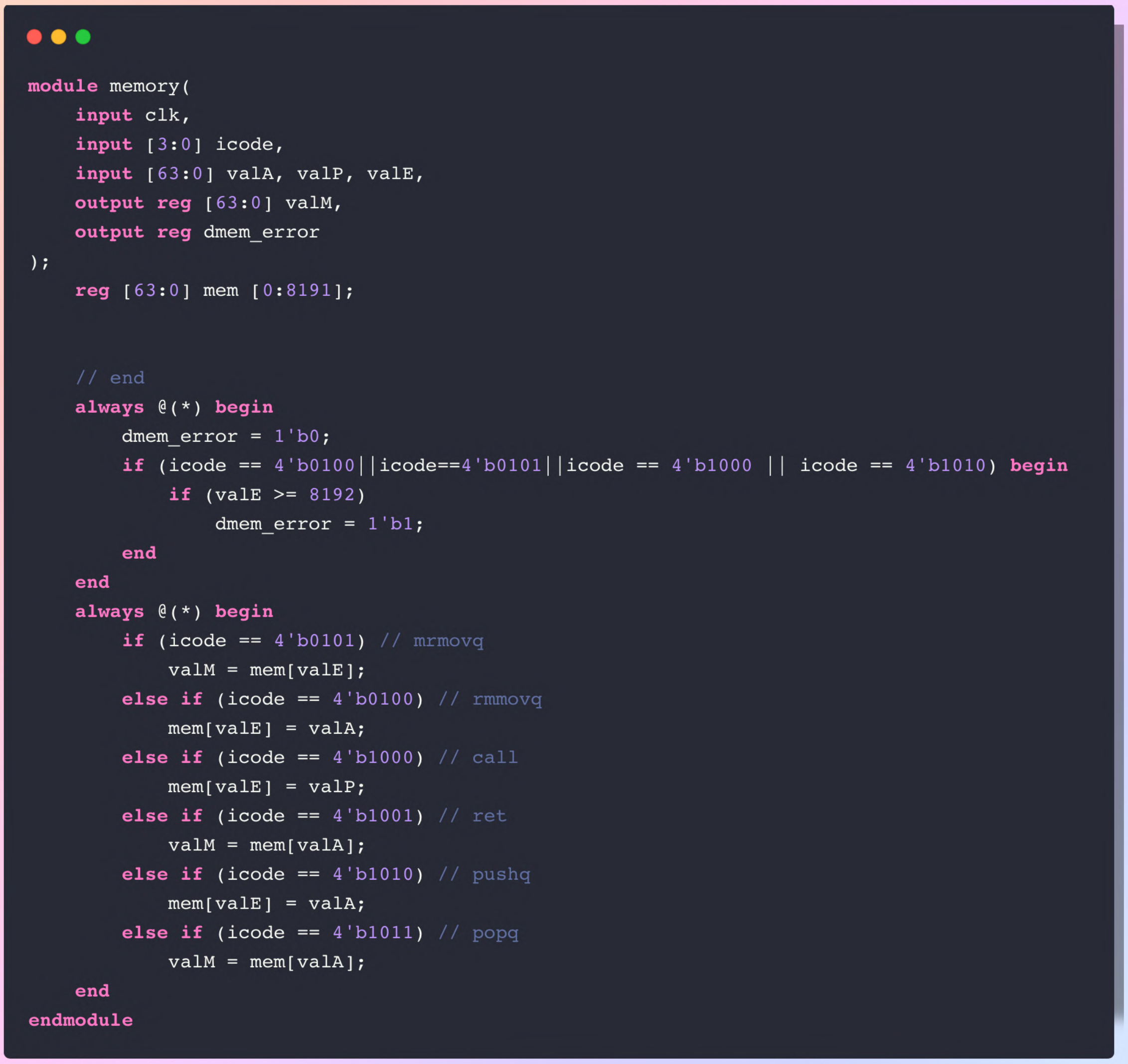
We have declared our memory as a register array, i.e.
reg [63:0] mem [0:8191]
Detailed working of this stage is given below:

<u>for rmmovq and pushq</u>: mem[valE] = valA

<u>for call</u>: mem[valE] = valP

<u>for mrmovq</u>: valM = mem[valE]

<u>for ret and popq</u>: valM = mem[valA]



```verilog
module memory(
    input clk,
    input [3:0] icode,
    input [63:0] valA, valP, valE,
    output reg [63:0] valM,
    output reg dmem_error
);
    reg [63:0] mem [0:8191];


    // end
    always @(*) begin
        dmem_error = 1'b0;
        if (icode == 4'b0100||icode==4'b0101||icode == 4'b1000 || icode == 4'b1010) begin
            if (valE >= 8192)
                dmem_error = 1'b1;
        end
    end
    always @(*) begin
        if (icode == 4'b0101) // mrmovq
            valM = mem[valE];
        else if (icode == 4'b0100) // rmmovq
            mem[valE] = valA;
        else if (icode == 4'b1000) // call
            mem[valE] = valP;
        else if (icode == 4'b1001) // ret
            valM = mem[valA];
        else if (icode == 4'b1010) // pushq
            mem[valE] = valA;
        else if (icode == 4'b1011) // popq
            valM = mem[valA];
    end
endmodule
```

- **Write-Back Stage:-**

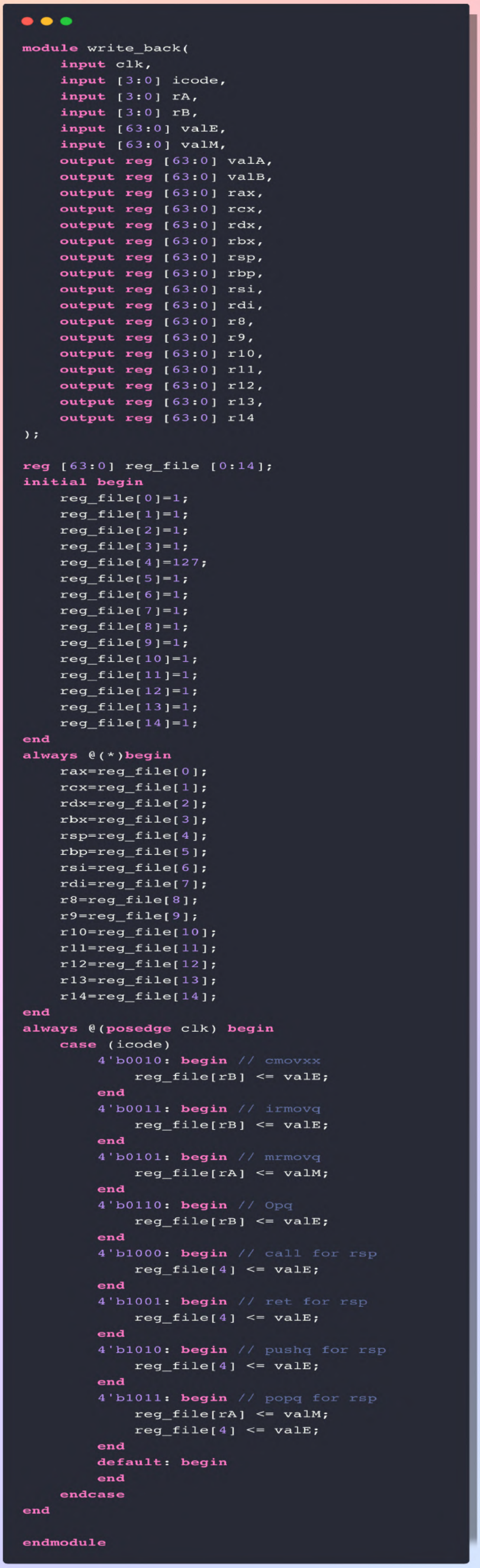
In this stage, the values valE or valM(found from execute and memory stages) are written into registers rA, rB or %rsp, depending on the values of icode.

for irmovq,cmovxx and opq: reg_file[rB] <= valE (for cmovxx is cnd is 1)

for mrmovq: reg_file[rA] <= valM

for call, ret, pushq: reg_file[4] <= valE

for popq: reg_file[rA] <= valM and reg_file[4] <= valE



```verilog
module write_back(
    input clk,
    input [3:0] icode,
    input [3:0] rA,
    input [3:0] rB,
    input [63:0] valE,
    input [63:0] valM,
    output reg [63:0] valA,
    output reg [63:0] valB,
    output reg [63:0] rax,
    output reg [63:0] rcx,
    output reg [63:0] rdx,
    output reg [63:0] rbx,
    output reg [63:0] rsp,
    output reg [63:0] rbp,
    output reg [63:0] rsi,
    output reg [63:0] rdi,
    output reg [63:0] r8,
    output reg [63:0] r9,
    output reg [63:0] r10,
    output reg [63:0] r11,
    output reg [63:0] r12,
    output reg [63:0] r13,
    output reg [63:0] r14
);

reg [63:0] reg_file [0:14];
initial begin
    reg_file[0]=1;
    reg_file[1]=1;
    reg_file[2]=1;
    reg_file[3]=1;
    reg_file[4]=127;
    reg_file[5]=1;
    reg_file[6]=1;
    reg_file[7]=1;
    reg_file[8]=1;
    reg_file[9]=1;
    reg_file[10]=1;
    reg_file[11]=1;
    reg_file[12]=1;
    reg_file[13]=1;
    reg_file[14]=1;
end
always @(*)begin
    rax=reg_file[0];
    rcx=reg_file[1];
    rdx=reg_file[2];
    rbx=reg_file[3];
    rsp=reg_file[4];
    rbp=reg_file[5];
    rsi=reg_file[6];
    rdi=reg_file[7];
    r8=reg_file[8];
    r9=reg_file[9];
    r10=reg_file[10];
    r11=reg_file[11];
    r12=reg_file[12];
    r13=reg_file[13];
    r14=reg_file[14];
end
always @(posedge clk) begin
    case (icode)
        4'b0010: begin // cmovxx
            reg_file[rB] <= valE;
        end
        4'b0011: begin // irmovq
            reg_file[rB] <= valE;
        end
        4'b0101: begin // mrmovq
            reg_file[rA] <= valM;
        end
        4'b0110: begin // Opq
            reg_file[rB] <= valE;
        end
        4'b1000: begin // call for rsp
            reg_file[4] <= valE;
        end
        4'b1001: begin // ret for rsp
            reg_file[4] <= valE;
        end
        4'b1010: begin // pushq for rsp
            reg_file[4] <= valE;
        end
        4'b1011: begin // popq for rsp
            reg_file[rA] <= valM;
            reg_file[4] <= valE;
        end
        default: begin
        end
    endcase
end

endmodule
```

• **PC update stage:-**

In this stage, the PC is updated to the address of the next instruction.

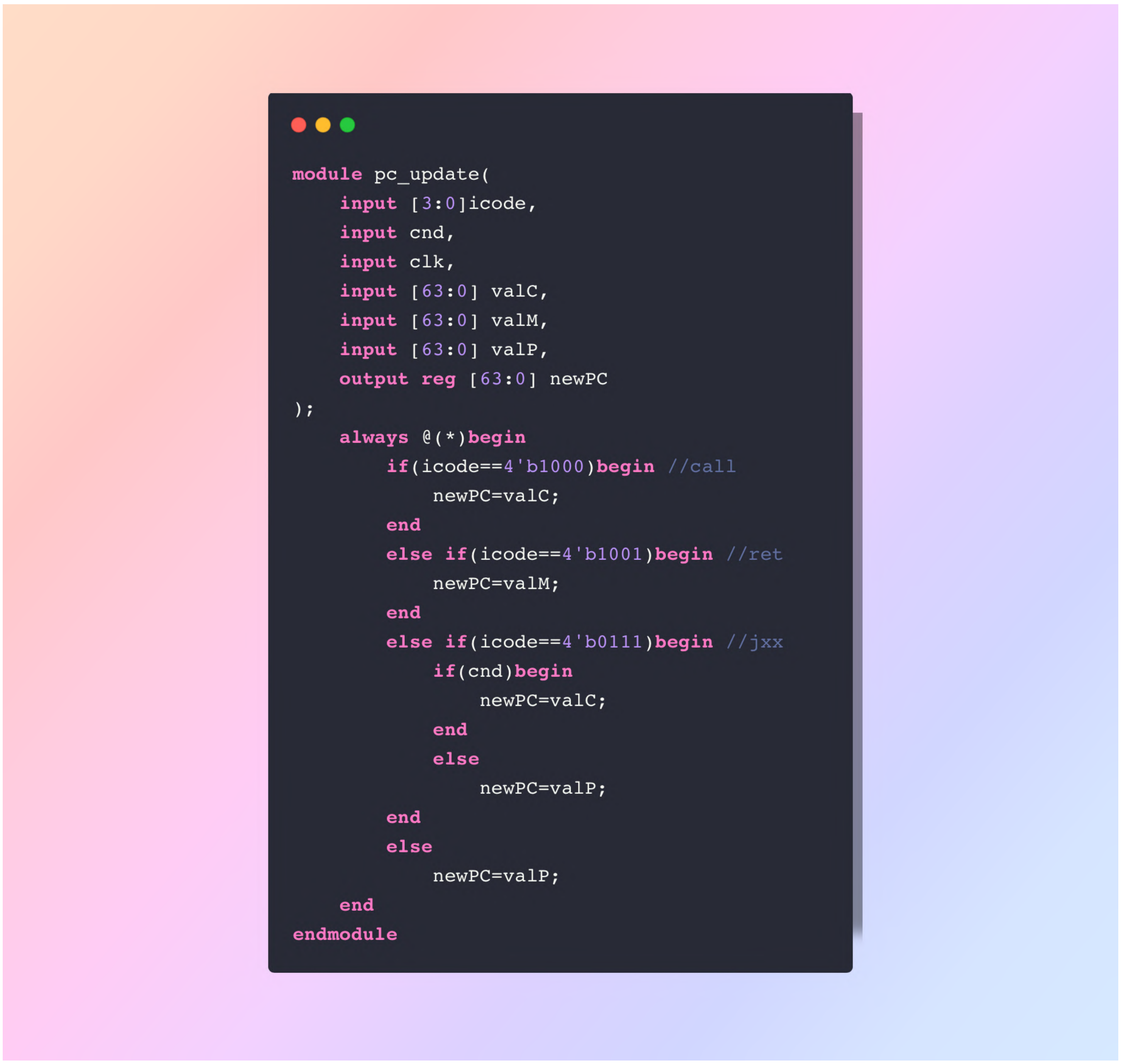
It can either take values valP, valC or valM depending on the instruction type(value of icode)

for call: newPC=valC

for ret: newPC=valM

for jxx: newPC=valC, if condition is true(cnd=1); otherwise it is valP

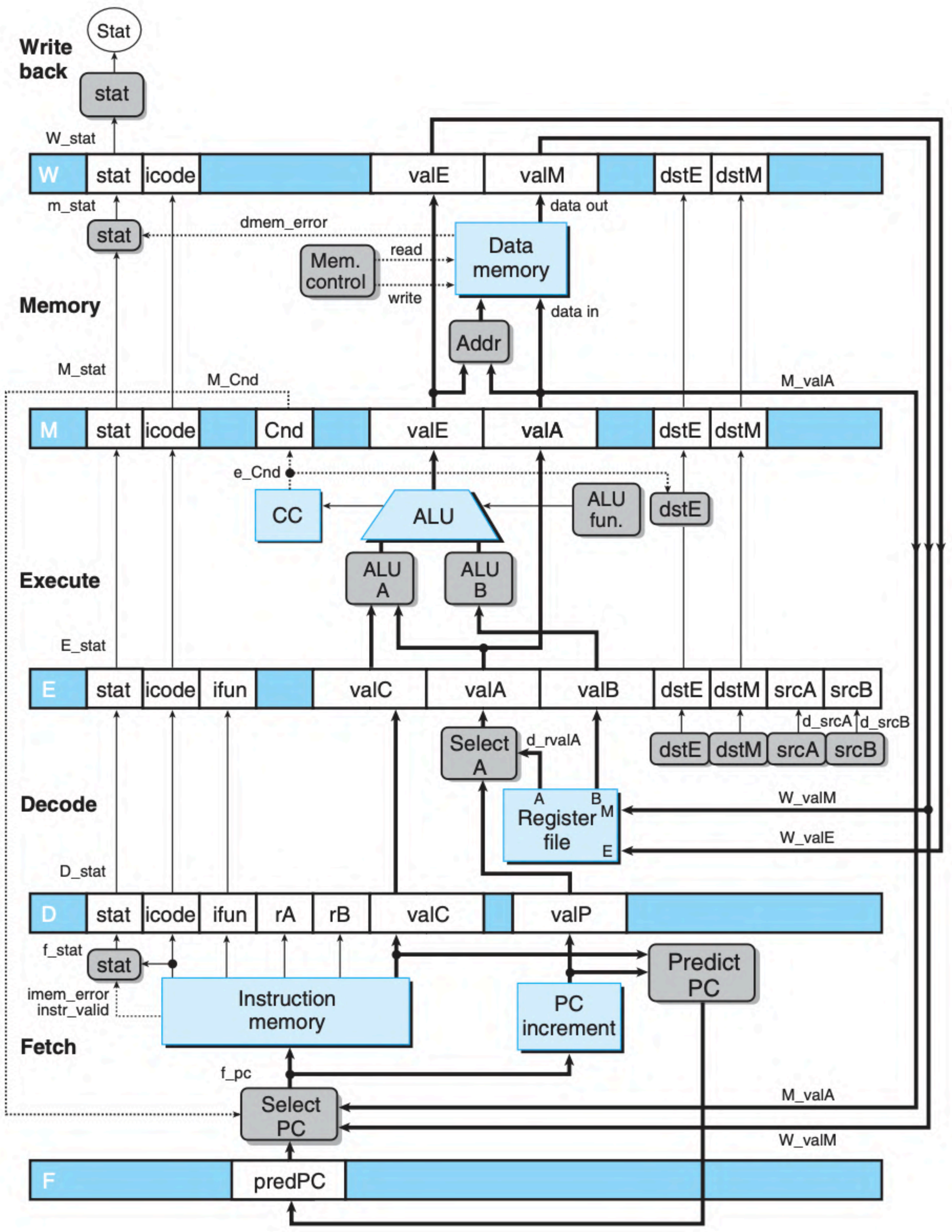
for all other cases: newPC=valP



```
module pc_update(
    input [3:0]icode,
    input cnd,
    input clk,
    input [63:0] valC,
    input [63:0] valM,
    input [63:0] valP,
    output reg [63:0] newPC
);
    always @(*)begin
        if(icode==4'b1000)begin //call
            newPC=valC;
        end
        else if(icode==4'b1001)begin //ret
            newPC=valM;
        end
        else if(icode==4'b0111)begin //jxx
            if(cnd)begin
                newPC=valC;
            end
            else
                newPC=valP;
        end
        else
            newPC=valP;
    end
endmodule
```

3. Pipelining

This section of the report contains details about the pipelined Y86-64 processor architecture.



The main advantage of implementing a pipelined architecture is it's increased throughput, i.e. we can process more number of instructions in a given interval of time. In SEQ, an instruction can only start executing, when the previous one has completed it's execution. But in our pipelined implementation(PIPE) we can process multiple instructions together, with every instruction going through some stage(fetch, decode,etc.)

Pipeline Registers:- There are 5 pipeline registers namely:

- F holds a predicted value of PC, and is inserted before the fetch stage.
- D is inserted between the fetch stage and the decode stage. It holds information about the most recently fetched instruction for processing by the decode stage.
- E is inserted between the decode and the execute stage. It holds information about the most recently decoded instruction and the values read from the register file for processing by the execute stage.
- M is inserted between execute and memory stages. It holds the results of the most recently executed instruction for processing by the memory stage.
- W is inserted after the memory stage. It has feedback paths that provide values to the register files for writing and also provide the return addresses to the PC selection logic(for a ret instruction)

Other changes from SEQ to Pipeline implementation include updating the PC at the start of the clock cycle and using status codes.

Now for predicting the next value of PC (predPC) we predict this value to be:

A. valP, for instructions that don't transfer control
B. valC, for call and unconditional jumps

C. valC, for conditional jumps(typically right 60% of the times)

The Select PC block then selects one of the three values for computation in the fetch stage:

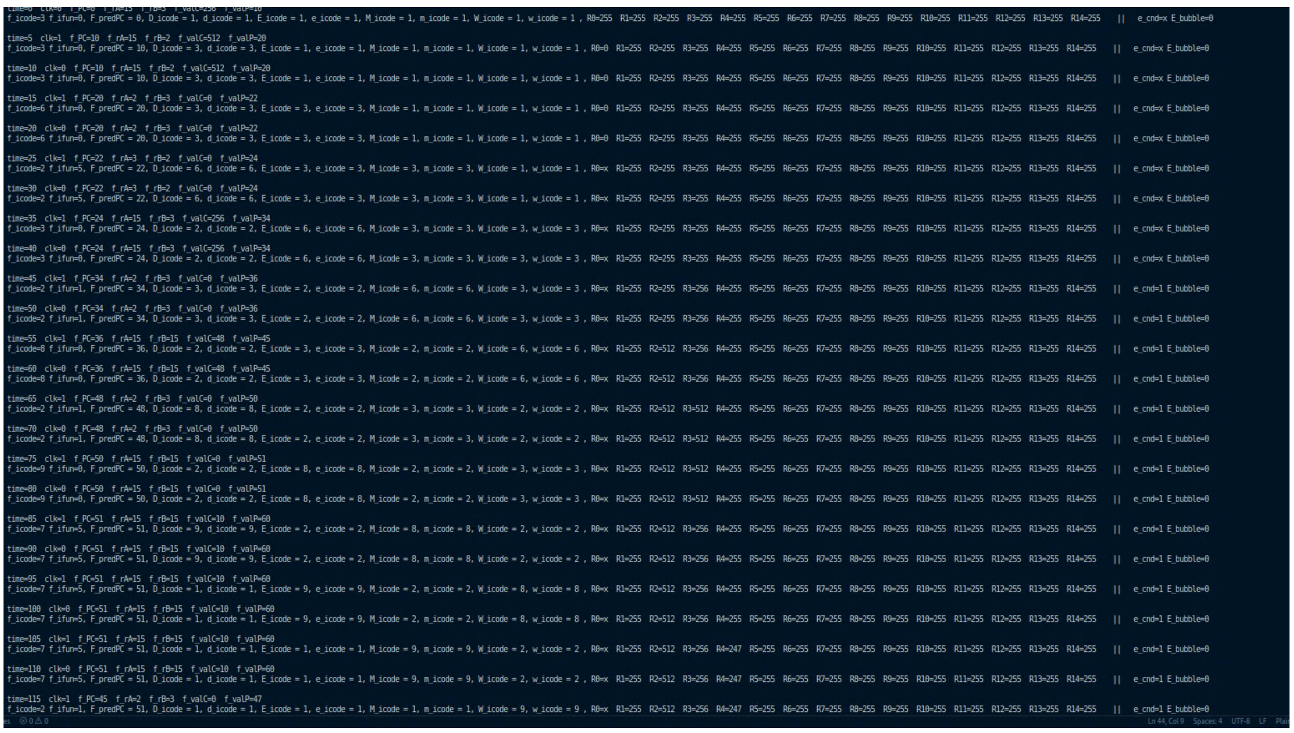
1. Predicted PC value
2. The value of valP for a not taken branch(stored in register M, M_valA)
3. The value of the return address when a ret instruction reaches register W(W_valM).

Data Hazards:

Data Hazards occur when there is a data dependency, i.e. results computed by one instruction are immediately required by another one.

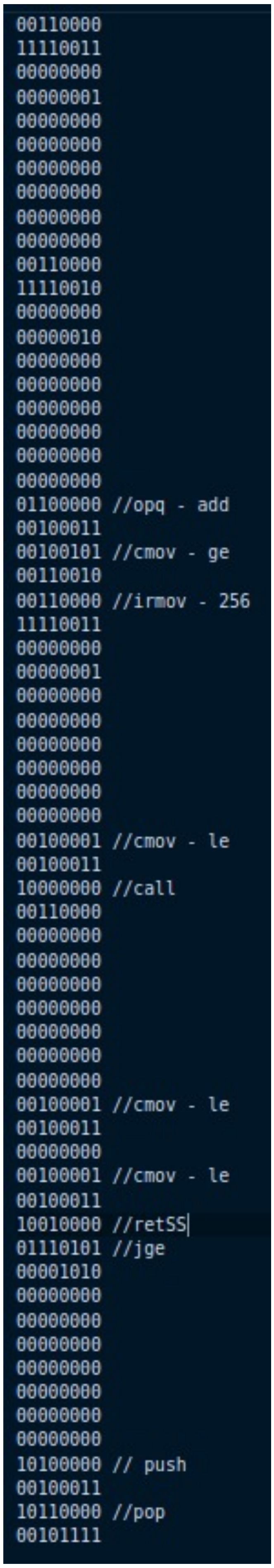
To avoid data hazards we could use techniques like Data Forwarding and/or Stalling. We prefer to use Data forwarding since Stalling requires the use of too many nops(bubbles).

Using Data Forwarding we can directly pass values from earlier registers(like in execute or memory stage) to the decode stage, rather than waiting for the instruction to pass through the write back stage.



```
time=0  clk=0  f_PC=0  f_rA=15  f_rB=3  f_valC=256  f_valP=10
f_icode=3 f_ifun=0, F_predPC = 0, D_icode = 1, d_icode = 1, E_icode = 1, e_icode = 1, M_icode = 1, m_icode = 1, W_icode = 1, w_icode = 1 , R0=255  R1=255  R2=255  R3=255  R4=255  R5=255  R6=255  R7=255  R8=255  R9=255  R10=255  R11=255  R12=255  R13=255  R14=255    ||   e_cnd=x E_bubble=0

time=5  clk=1  f_PC=10  f_rA=15  f_rB=2  f_valC=512  f_valP=20
f_icode=3 f_ifun=0, F_predPC = 10, D_icode = 3, d_icode = 3, E_icode = 1, e_icode = 1, M_icode = 1, m_icode = 1, W_icode = 1, w_icode = 1 , R0=0  R1=255  R2=255  R3=255  R4=255  R5=255  R6=255  R7=255  R8=255  R9=255  R10=255  R11=255  R12=255  R13=255  R14=255    ||   e_cnd=x E_bubble=0

time=10  clk=0  f_PC=10  f_rA=15  f_rB=2  f_valC=512  f_valP=20
f_icode=3 f_ifun=0, F_predPC = 10, D_icode = 3, d_icode = 3, E_icode = 1, e_icode = 1, M_icode = 1, m_icode = 1, W_icode = 1, w_icode = 1 , R0=0  R1=255  R2=255  R3=255  R4=255  R5=255  R6=255  R7=255  R8=255  R9=255  R10=255  R11=255  R12=255  R13=255  R14=255    ||   e_cnd=x E_bubble=0

time=15  clk=1  f_PC=20  f_rA=2  f_rB=3  f_valC=0  f_valP=22
f_icode=6 f_ifun=0, F_predPC = 20, D_icode = 3, d_icode = 3, E_icode = 3, e_icode = 3, M_icode = 1, m_icode = 1, W_icode = 1, w_icode = 1 , R0=0  R1=255  R2=255  R3=255  R4=255  R5=255  R6=255  R7=255  R8=255  R9=255  R10=255  R11=255  R12=255  R13=255  R14=255    ||   e_cnd=x E_bubble=0

time=20  clk=0  f_PC=20  f_rA=2  f_rB=3  f_valC=0  f_valP=22
f_icode=6 f_ifun=0, F_predPC = 20, D_icode = 3, d_icode = 3, E_icode = 3, e_icode = 3, M_icode = 1, m_icode = 1, W_icode = 1, w_icode = 1 , R0=0  R1=255  R2=255  R3=255  R4=255  R5=255  R6=255  R7=255  R8=255  R9=255  R10=255  R11=255  R12=255  R13=255  R14=255    ||   e_cnd=x E_bubble=0

time=25  clk=1  f_PC=22  f_rA=3  f_rB=2  f_valC=0  f_valP=24
f_icode=2 f_ifun=5, F_predPC = 22, D_icode = 6, d_icode = 6, E_icode = 3, e_icode = 3, M_icode = 3, m_icode = 3, W_icode = 1, w_icode = 1 , R0=x  R1=255  R2=255  R3=255  R4=255  R5=255  R6=255  R7=255  R8=255  R9=255  R10=255  R11=255  R12=255  R13=255  R14=255    ||   e_cnd=x E_bubble=0

time=30  clk=0  f_PC=22  f_rA=3  f_rB=2  f_valC=0  f_valP=24
f_icode=2 f_ifun=5, F_predPC = 22, D_icode = 6, d_icode = 6, E_icode = 3, e_icode = 3, M_icode = 3, m_icode = 3, W_icode = 1, w_icode = 1 , R0=x  R1=255  R2=255  R3=255  R4=255  R5=255  R6=255  R7=255  R8=255  R9=255  R10=255  R11=255  R12=255  R13=255  R14=255    ||   e_cnd=x E_bubble=0

time=35  clk=1  f_PC=24  f_rA=15  f_rB=3  f_valC=256  f_valP=34
f_icode=3 f_ifun=0, F_predPC = 24, D_icode = 2, d_icode = 2, E_icode = 6, e_icode = 6, M_icode = 3, m_icode = 3, W_icode = 3, w_icode = 3 , R0=x  R1=255  R2=255  R3=255  R4=255  R5=255  R6=255  R7=255  R8=255  R9=255  R10=255  R11=255  R12=255  R13=255  R14=255    ||   e_cnd=x E_bubble=0

time=40  clk=0  f_PC=24  f_rA=15  f_rB=3  f_valC=256  f_valP=34
f_icode=3 f_ifun=0, F_predPC = 24, D_icode = 2, d_icode = 2, E_icode = 6, e_icode = 6, M_icode = 3, m_icode = 3, W_icode = 3, w_icode = 3 , R0=x  R1=255  R2=255  R3=255  R4=255  R5=255  R6=255  R7=255  R8=255  R9=255  R10=255  R11=255  R12=255  R13=255  R14=255    ||   e_cnd=x E_bubble=0

time=45  clk=1  f_PC=34  f_rA=2  f_rB=3  f_valC=0  f_valP=36
f_icode=2 f_ifun=1, F_predPC = 34, D_icode = 3, d_icode = 3, E_icode = 2, e_icode = 2, M_icode = 6, m_icode = 6, W_icode = 3, w_icode = 3 , R0=x  R1=255  R2=255  R3=256  R4=255  R5=255  R6=255  R7=255  R8=255  R9=255  R10=255  R11=255  R12=255  R13=255  R14=255    ||   e_cnd=1 E_bubble=0

time=50  clk=0  f_PC=34  f_rA=2  f_rB=3  f_valC=0  f_valP=36
f_icode=2 f_ifun=1, F_predPC = 34, D_icode = 3, d_icode = 3, E_icode = 2, e_icode = 2, M_icode = 6, m_icode = 6, W_icode = 3, w_icode = 3 , R0=x  R1=255  R2=255  R3=256  R4=255  R5=255  R6=255  R7=255  R8=255  R9=255  R10=255  R11=255  R12=255  R13=255  R14=255    ||   e_cnd=1 E_bubble=0

time=55  clk=1  f_PC=36  f_rA=15  f_rB=15  f_valC=48  f_valP=45
f_icode=8 f_ifun=0, F_predPC = 36, D_icode = 2, d_icode = 2, E_icode = 3, e_icode = 3, M_icode = 2, m_icode = 2, W_icode = 6, w_icode = 6 , R0=x  R1=255  R2=512  R3=256  R4=255  R5=255  R6=255  R7=255  R8=255  R9=255  R10=255  R11=255  R12=255  R13=255  R14=255    ||   e_cnd=1 E_bubble=0

time=60  clk=0  f_PC=36  f_rA=15  f_rB=15  f_valC=48  f_valP=45
f_icode=8 f_ifun=0, F_predPC = 36, D_icode = 2, d_icode = 2, E_icode = 3, e_icode = 3, M_icode = 2, m_icode = 2, W_icode = 6, w_icode = 6 , R0=x  R1=255  R2=512  R3=256  R4=255  R5=255  R6=255  R7=255  R8=255  R9=255  R10=255  R11=255  R12=255  R13=255  R14=255    ||   e_cnd=1 E_bubble=0

time=65  clk=1  f_PC=48  f_rA=2  f_rB=3  f_valC=0  f_valP=50
f_icode=2 f_ifun=1, F_predPC = 48, D_icode = 8, d_icode = 8, E_icode = 2, e_icode = 2, M_icode = 3, m_icode = 3, W_icode = 2, w_icode = 2 , R0=x  R1=255  R2=512  R3=512  R4=255  R5=255  R6=255  R7=255  R8=255  R9=255  R10=255  R11=255  R12=255  R13=255  R14=255    ||   e_cnd=1 E_bubble=0

time=70  clk=0  f_PC=48  f_rA=2  f_rB=3  f_valC=0  f_valP=50
f_icode=2 f_ifun=1, F_predPC = 48, D_icode = 8, d_icode = 8, E_icode = 2, e_icode = 2, M_icode = 3, m_icode = 3, W_icode = 2, w_icode = 2 , R0=x  R1=255  R2=512  R3=512  R4=255  R5=255  R6=255  R7=255  R8=255  R9=255  R10=255  R11=255  R12=255  R13=255  R14=255    ||   e_cnd=1 E_bubble=0

time=75  clk=1  f_PC=50  f_rA=15  f_rB=15  f_valC=0  f_valP=51
f_icode=9 f_ifun=0, F_predPC = 50, D_icode = 2, d_icode = 2, E_icode = 8, e_icode = 8, M_icode = 2, m_icode = 2, W_icode = 3, w_icode = 3 , R0=x  R1=255  R2=512  R3=512  R4=255  R5=255  R6=255  R7=255  R8=255  R9=255  R10=255  R11=255  R12=255  R13=255  R14=255    ||   e_cnd=1 E_bubble=0

time=80  clk=0  f_PC=50  f_rA=15  f_rB=15  f_valC=0  f_valP=51
f_icode=9 f_ifun=0, F_predPC = 50, D_icode = 2, d_icode = 2, E_icode = 8, e_icode = 8, M_icode = 2, m_icode = 2, W_icode = 3, w_icode = 3 , R0=x  R1=255  R2=512  R3=512  R4=255  R5=255  R6=255  R7=255  R8=255  R9=255  R10=255  R11=255  R12=255  R13=255  R14=255    ||   e_cnd=1 E_bubble=0

time=85  clk=1  f_PC=51  f_rA=15  f_rB=15  f_valC=10  f_valP=60
f_icode=7 f_ifun=5, F_predPC = 51, D_icode = 9, d_icode = 9, E_icode = 2, e_icode = 2, M_icode = 8, m_icode = 8, W_icode = 2, w_icode = 2 , R0=x  R1=255  R2=512  R3=256  R4=255  R5=255  R6=255  R7=255  R8=255  R9=255  R10=255  R11=255  R12=255  R13=255  R14=255    ||   e_cnd=1 E_bubble=0

time=90  clk=0  f_PC=51  f_rA=15  f_rB=15  f_valC=10  f_valP=60
f_icode=7 f_ifun=5, F_predPC = 51, D_icode = 9, d_icode = 9, E_icode = 2, e_icode = 2, M_icode = 8, m_icode = 8, W_icode = 2, w_icode = 2 , R0=x  R1=255  R2=512  R3=256  R4=255  R5=255  R6=255  R7=255  R8=255  R9=255  R10=255  R11=255  R12=255  R13=255  R14=255    ||   e_cnd=1 E_bubble=0

time=95  clk=1  f_PC=51  f_rA=15  f_rB=15  f_valC=10  f_valP=60
f_icode=7 f_ifun=5, F_predPC = 51, D_icode = 1, d_icode = 1, E_icode = 9, e_icode = 9, M_icode = 2, m_icode = 2, W_icode = 8, w_icode = 8 , R0=x  R1=255  R2=512  R3=256  R4=255  R5=255  R6=255  R7=255  R8=255  R9=255  R10=255  R11=255  R12=255  R13=255  R14=255    ||   e_cnd=1 E_bubble=0

time=100  clk=0  f_PC=51  f_rA=15  f_rB=15  f_valC=10  f_valP=60
f_icode=7 f_ifun=5, F_predPC = 51, D_icode = 1, d_icode = 1, E_icode = 9, e_icode = 9, M_icode = 2, m_icode = 2, W_icode = 8, w_icode = 8 , R0=x  R1=255  R2=512  R3=256  R4=255  R5=255  R6=255  R7=255  R8=255  R9=255  R10=255  R11=255  R12=255  R13=255  R14=255    ||   e_cnd=1 E_bubble=0

time=105  clk=1  f_PC=51  f_rA=15  f_rB=15  f_valC=10  f_valP=60
f_icode=7 f_ifun=5, F_predPC = 51, D_icode = 1, d_icode = 1, E_icode = 1, e_icode = 1, M_icode = 9, m_icode = 9, W_icode = 2, w_icode = 2 , R0=x  R1=255  R2=512  R3=256  R4=247  R5=255  R6=255  R7=255  R8=255  R9=255  R10=255  R11=255  R12=255  R13=255  R14=255    ||   e_cnd=1 E_bubble=0

time=110  clk=0  f_PC=51  f_rA=15  f_rB=15  f_valC=10  f_valP=60
f_icode=7 f_ifun=5, F_predPC = 51, D_icode = 1, d_icode = 1, E_icode = 1, e_icode = 1, M_icode = 9, m_icode = 9, W_icode = 2, w_icode = 2 , R0=x  R1=255  R2=512  R3=256  R4=247  R5=255  R6=255  R7=255  R8=255  R9=255  R10=255  R11=255  R12=255  R13=255  R14=255    ||   e_cnd=1 E_bubble=0

time=115  clk=1  f_PC=45  f_rA=2  f_rB=3  f_valC=0  f_valP=47
f_icode=2 f_ifun=1, F_predPC = 51, D_icode = 1, d_icode = 1, E_icode = 1, e_icode = 1, M_icode = 1, m_icode = 1, W_icode = 9, w_icode = 9 , R0=x  R1=255  R2=512  R3=256  R4=247  R5=255  R6=255  R7=255  R8=255  R9=255  R10=255  R11=255  R12=255  R13=255  R14=255    ||   e_cnd=1 E_bubble=0
```

Testcases:



```
00110000
11110011
00000000
00000001
00000000
00000000
00000000
00000000
00000000
00000000
00110000
11110010
00000000
00000010
00000000
00000000
00000000
00000000
00000000
00000000
01100000 //opq - add
00100011
00100101 //cmov - ge
00110010
00110000 //irmov - 256
11110011
00000000
00000001
00000000
00000000
00000000
00000000
00000000
00000000
00100001 //cmov - le
00100011
10000000 //call
00110000
00000000
00000000
00000000
00000000
00000000
00000000
00000000
00100001 //cmov - le
00100011
00000000
00100001 //cmov - le
00100011
10010000 //retSS
01110101 //jge
00001010
00000000
00000000
00000000
00000000
00000000
00000000
00000000
10100000 // push
00100011
10110000 //pop
00101111
```

Challenges Faced:-

1. Sequential: It was difficult understanding how the instruction memory and stack memory was getting read and how the stack pointer was getting updated. Also while implementing call and ret function, we were not able to figure out some of the errors and so had to make changes in our implementation.
2. Pipelining: It was really difficult to understand the terminologies used in the reference books and also the flow of control was difficult to understand. Some concepts like data forwarding were tough to implement. Debugging the code was tough because there were a lot of instructions to be considered for debugging a single line of code.

Acknowledgements:-

This project has been an excellent learning opportunity. In this course, we've gained a better grasp of processor architecture, instruction sets, memory, and various other concepts. We're grateful to Professor Deepak and the Teaching Assistants for their guidance throughout the project.

Regards,

Arya and Rohan.