Report

Name: Santrupta Singh, Roll No: 2021102035

Name: Vedant Liladhar Nipane, Roll No: 2021102040

Table of Contents:

Overview

The aim of the project is to implement a pipelined Y86-64 processor architecture with data forwarding and control logic in Verilog. The processor should be able to execute all instructions in the Y86-64 ISA. The first part of the project involved making a sequential implementation of the Y86-64
architecture.

As compared to its sequential counterpart (SEQ), a pipelined implementation (PIPE) of a processor allows for an increased throughput. With SEQ, an instruction is executed only after that which is preceding it has finished executing. In other words, the instructions follow a sequence, hence the name. On the other hand, with PIPE, an instruction is divided into a fixed number of stages.

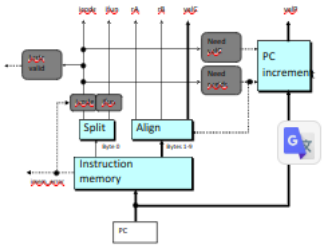
Sequential

Fetch

- In the Fetch stage of SEQ implementation, we are supposed to read instruction by instruction and find the values of `icode`, `ifun`, `rA`, `rB`, `valC` and `valP`.
- We declared the instruction memory as `inst_memo` in the fetch stage.

```
reg [7:0] inst_memo[0:127];
```

- Control Logic:
  - Instr. Valid: Is this instruction valid?
  - icode, ifun: Generate no-op if invalid address
  - Need regids: Does this instruction have a register byte?
  - Need valC: Does this instruction have a constant word?



- Working of the fetch stage:
  - For the current value of PC, `icode` and `ifun` will be calculated from the instruction memory as follows:

```
icode = inst_memo[PC][7:4];
ifun = inst_memo[PC][3:0];
```

  - We need `icode` and `ifun` to decide which instruction and sub-instruction to execute.
  - Depending on the value of `icode`, the rest of the output values are obtained as follows:

```
temp = {inst_memo[PC+1]};
rA = temp[0:3];
rB = temp[4:7];
valP = PC + x; // x depends on the type of instruction
```

  - If `icode` is invalid then the `invalid_instr` is set to 1.
- Code:

```
module fetch(clk, PC, icode, ifun, rA, rB, valC, valP,
             memory_error, halt, invalid_instr);

            input clk;
            input [63:0]PC ;
            reg [0:7] temp;

            output reg [3:0] icode, ifun, rA,rB;
```

```
            output reg [63:0] valC,valP;
            output reg halt,invalid_instr,memory_error;

            reg [7:0] inst_memo[0:127];


initial begin
    halt = 0;
    inst_memo[0]  = 8'b00010000; // nop instruction PC = PC +1 = 1
    inst_memo[1]  = 8'b01100000; // Opq add
    inst_memo[2]  = 8'b00100001; // rA = 0, rB = 1; PC = PC + 2 = 3

    inst_memo[3]  = 8'b00110000; // irmovq instruction PC = PC + 10 = 13
    inst_memo[4]  = 8'b11110010; // F, rB = 2;
    inst_memo[5]  = 8'b00000101; // 1st byte of V = 5, rest all bytes will be zero
    inst_memo[6]  = 8'b00000000; // 2nd byte
    inst_memo[7]  = 8'b00000000; // 3rd byte
    inst_memo[8]  = 8'b00000000; // 4th byte
    inst_memo[9]  = 8'b00000000; // 5th byte
    inst_memo[10] = 8'b00000000; // 6th byte
    inst_memo[11] = 8'b00000000; // 7th byte
    inst_memo[12] = 8'b00000000; // 8th byte (This completes irmovq)

    inst_memo[13] = 8'b00110000; // irmovq instruction PC = PC + 10 = 23
    inst_memo[14] = 8'b11110011; // F, rB = 3;
    inst_memo[15] = 8'b00000101; // 1st byte of V = 5, rest all bytes will be zero
    inst_memo[16] = 8'b00000000; // 2nd byte
    inst_memo[17] = 8'b00000000; // 3rd byte
    inst_memo[18] = 8'b00000000; // 4th byte
    inst_memo[19] = 8'b00000000; // 5th byte
    inst_memo[20] = 8'b00000000; // 6th byte
    inst_memo[21] = 8'b00000000; // 7th byte
    inst_memo[22] = 8'b00000000; // 8th byte (This completes irmovq)

    inst_memo[23] = 8'b00110000; // irmovq instruction PC = PC + 10 = 33
    inst_memo[24] = 8'b11111100; // F, rB = 12;
    inst_memo[25] = 8'b00000101; // 1st byte of V = 5, rest all bytes will be zero
    inst_memo[26] = 8'b00000000; // 2nd byte
    inst_memo[27] = 8'b00000000; // 3rd byte
    inst_memo[28] = 8'b00000000; // 4th byte
    inst_memo[29] = 8'b00000000; // 5th byte
    inst_memo[30] = 8'b00000000; // 6th byte
    inst_memo[31] = 8'b00000000; // 7th byte
    inst_memo[32] = 8'b00000000; // 8th byte (This completes irmovq)

    inst_memo[33] = 8'b01100000; // Opq add // PC = PC + 2 = 37
    inst_memo[34] = 8'b00111100; // rA = 3 and rB = 12, final value in rB(4) = 10;



    inst_memo[37] = 8'b00100000; // rrmovq // PC = PC + 2 = 35
    inst_memo[38] = 8'b11000110; // rA = 12; rB = 5;

    inst_memo[35] = 8'b00100101; // cmovge // PC = PC + 2 = 39
    inst_memo[36] = 8'b01100011; // rA = 5; rB = 6;

  inst_memo[39] = 8'b01100001; // Opq subq // PC = PC + 2 = 41
    inst_memo[40] = 8'b11000110; // rA = 3, rB = 5; both are equal


    inst_memo[41] = 8'b01110011; //je // PC = PC + 9 = 50
    inst_memo[42] = 8'b00110101; // Dest = 53; 1st byte
    inst_memo[43] = 8'b00000000; // 2nd byte
    inst_memo[44] = 8'b00000000; // 3rd byte
    inst_memo[45] = 8'b00000000; // 4th byte
    inst_memo[46] = 8'b00000000; // 5th byte
    inst_memo[47] = 8'b00000000; // 6th byte
    inst_memo[48] = 8'b00000000; // 7th byte
    inst_memo[49] = 8'b00000000; // 8th byte

    inst_memo[50] = 8'b00010000; // nop

    // inst_memo[51] = 8'b01100001; // Opq add
    // inst_memo[52] = 8'b00110101; // rA = 3; rB = 5;
    inst_memo[53] = 8'b00010000; // nop
    inst_memo[54] = 8'b10000000; //call // PC = PC + 9 = 50
    inst_memo[55] = 8'b01000101; // Dest = 53; 1st byte
    inst_memo[56] = 8'b00000000; // 2nd byte
    inst_memo[57] = 8'b00000000; // 3rd byte
    inst_memo[58] = 8'b00000000; // 4th byte
    inst_memo[59] = 8'b00000000; // 5th byte
    inst_memo[60] = 8'b00000000; // 6th byte
    inst_memo[61] = 8'b00000000; // 7th byte
    inst_memo[62] = 8'b00000000;

    inst_memo[69] = 8'b00010000;
```

```
     inst_memo[70] = 8'b00110000; // irmovq instruction PC = PC + 10 = 33
    inst_memo[71] = 8'b11111010; // F, rB = 10;
    inst_memo[72] = 8'b01100101; // 1st byte of V = 101, rest all bytes will be zero
    inst_memo[73] = 8'b00000000; // 2nd byte
    inst_memo[74] = 8'b00000000; // 3rd byte
    inst_memo[75] = 8'b00000000; // 4th byte
    inst_memo[76] = 8'b00000000; // 5th byte
    inst_memo[77] = 8'b00000000; // 6th byte
    inst_memo[78] = 8'b00000000; // 7th byte
    inst_memo[79] = 8'b00000000;

    inst_memo[80] = 8'b01000000; // rmmovq instruction PC = PC + 10 = 33
    inst_memo[81] = 8'b10100111; // 10, rB = 7;
    inst_memo[82] = 8'b00000000; // 1st byte of V = 5, rest all bytes will be zero
    inst_memo[83] = 8'b00000000; // 2nd byte
    inst_memo[84] = 8'b00000000; // 3rd byte
    inst_memo[85] = 8'b00000000; // 4th byte
    inst_memo[86] = 8'b00000000; // 5th byte
    inst_memo[87] = 8'b00000000; // 6th byte
    inst_memo[88] = 8'b00000000; // 7th byte
    inst_memo[89] = 8'b00000000;

 inst_memo[90] = 8'b00110000; // irmovq instruction PC = PC + 10 = 33
    inst_memo[91] = 8'b11110100; // F, rB = 10;
    inst_memo[92] = 8'b00000000; // 1st byte of V = 100, rest all bytes will be zero
    inst_memo[93] = 8'b00000000; // 2nd byte
    inst_memo[94] = 8'b00000000; // 3rd byte
    inst_memo[95] = 8'b00000000; // 4th byte
    inst_memo[96] = 8'b00000000; // 5th byte
    inst_memo[97] = 8'b00000000; // 6th byte
    inst_memo[98] = 8'b00000000; // 7th byte
    inst_memo[99] = 8'b00000000;


    inst_memo[100] = 8'b10010000; //ret




    inst_memo[101] = 8'b00010000; // nop
    inst_memo[102] = 8'b00000000; // halt
end




            always @(*) begin
                memory_error = 0;
                if(PC >102)
                begin
                    memory_error = 1;
                end

                icode = inst_memo[PC][7:4];
                ifun = inst_memo[PC][3:0];

                if(icode >= 4'b0000 && icode <4'b1100)
                begin

                    invalid_instr = 0;

                    case (icode)

                    4'b0000: //halt
                    begin
                        halt = 1;
                    end

                    4'b0001://nop
                    begin
                        valP = PC + 64'd1;
                    end

                    4'b0011://irmove
                    begin
                        temp = {inst_memo[PC+1]};
                        rA = temp[0:3];
                        rB = temp[4:7];
                        valC = { inst_memo[PC+9],
                        inst_memo[PC+8],inst_memo[PC+7],
                        inst_memo[PC+6],inst_memo[PC+5],
                        inst_memo[PC+4],inst_memo[PC+3],
                        inst_memo[PC+2]
```

```
    };

     valP = PC + 64'd10;
end

4'b0101://mrmove
begin
       temp = {inst_memo[PC+1]};
    rA = temp[0:3];
    rB = temp[4:7];
    valC = { inst_memo[PC+9],
    inst_memo[PC+8],inst_memo[PC+7],
    inst_memo[PC+6],inst_memo[PC+5],
    inst_memo[PC+4],inst_memo[PC+3],
    inst_memo[PC+2]
    };

     valP = PC + 64'd10;
end

4'b0100://rmmove
begin
      temp = {inst_memo[PC+1]};
    rA = temp[0:3];
    rB = temp[4:7];
    valC = { inst_memo[PC+9],
    inst_memo[PC+8],inst_memo[PC+7],
    inst_memo[PC+6],inst_memo[PC+5],
    inst_memo[PC+4],inst_memo[PC+3],
    inst_memo[PC+2]
    };

     valP = PC + 64'd10;
end

4'b0010://cmove
begin
      temp = {inst_memo[PC+1]};
    rA = temp[0:3];
    rB = temp[4:7];

     valP = PC + 64'd2;
end

 4'b0110://opq
begin
      temp = {inst_memo[PC+1]};
    rA = temp[0:3];
    rB = temp[4:7];

     valP = PC + 64'd2;
end

 4'b1010://push
begin
      temp = {inst_memo[PC+1]};
    rA = temp[0:3];
    rB = temp[4:7];

     valP = PC + 64'd2;
end 4'b1011://pop
begin
       temp = {inst_memo[PC+1]};
    rA = temp[0:3];
    rB = temp[4:7];

     valP = PC + 64'd2;
end

 4'b0111://jxx
begin
    valC = { inst_memo[PC+8],
    inst_memo[PC+7],inst_memo[PC+6],
    inst_memo[PC+5],inst_memo[PC+4],
    inst_memo[PC+3],inst_memo[PC+2],
    inst_memo[PC+1]
    };

     valP = PC + 64'd9;
end
 4'b1000://call
begin
    valC = { inst_memo[PC+8],
    inst_memo[PC+7],inst_memo[PC+6],
    inst_memo[PC+5],inst_memo[PC+4],
    inst_memo[PC+3],inst_memo[PC+2],
    inst_memo[PC+1]
```

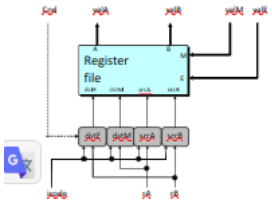
```
                    };

                     valP = PC + 64'd9;
                end
                 4'b1001://ret
                begin
                    valC = PC + 64'd1;
                end

                endcase
            end
            else
            begin
                invalid_instr = 1;
            end
        end
endmodule
```

Decode

- In the decode stage we are required to output `valA` and `valB` based on `icode`, `rA` and `rB`.
- We created 15 register arrays named `reg_file0` to `reg_file14` as inputs.
- We also created a register `temp_memo` which represents the register memory of the processor.

```
reg [63:0] temp_memo[0:14];
```

- Control Logic:
  - srcA, srcB: read port addresses
  - dstE, dstM: write port addresses



- Working of the Decode Stage:
  - The values of `valA` and `valB` are present in the registers given by address `rA` and `rB` . We use an `rA` and `rB` and index and reference the `temp_memo` register array for `valA` and `valB` .
  - Depending on the `icode` of the instructions we decide the `values` of `valA` and `valB` as follows:
    - `valA` = `temp_memo[rA]` for `cmovxx`
    - `valA` = `temp_memo[rA]` for `rmmovq`
    - `valB` = `temp_memo[rB]` for `OPq`
    - `valB` = `temp_memo[rB]` for `mrmovq`
    - `valB = temp_memo[4'b0100]` for `call`
    - `valA` = `temp_memo[4'b0100]` and `valB` = `temp_memo[4'b0100]` for `ret and popq`
    - `valA = temp_memo[rA], valB = temp_memo[4'b0100]` for push
- Code:

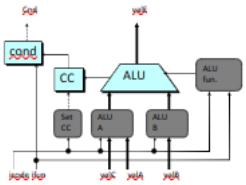
```
module decode_seq(input clk,
                input [3:0] icode,
                input [3:0] rA,
                input [3:0] rB,
                output reg [63:0] valA,
                output reg [63:0] valB,
                input [63:0] reg_file0,
                input [63:0] reg_file1,
```

```
                input [63:0] reg_file2,
                input [63:0] reg_file3,
                input [63:0] reg_file4,
                input [63:0] reg_file5,
                input [63:0] reg_file6,
                input [63:0] reg_file7,
                input [63:0] reg_file8,
                input [63:0] reg_file9,
                input [63:0] reg_file10,
                input [63:0] reg_file11,
                input [63:0] reg_file12,
                input [63:0] reg_file13,
                input [63:0] reg_file14);

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

    always @(*) begin

        temp_memo[0] = reg_file0;
        temp_memo[1] = reg_file1;
        temp_memo[2] = reg_file2;
        temp_memo[3] = reg_file3;
        temp_memo[4] = reg_file4;
        temp_memo[5] = reg_file5;
        temp_memo[6] = reg_file6;
        temp_memo[7] = reg_file7;
        temp_memo[8] = reg_file8;
        temp_memo[9] = reg_file9;
        temp_memo[10] = reg_file10;
        temp_memo[11] = reg_file11;
        temp_memo[12] = reg_file12;
        temp_memo[13] = reg_file13;
        temp_memo[14] = reg_file14;

    if (icode == 4'd2) begin //cmove
        valA = temp_memo[rA];
    end
    else if (icode == 4'd3) begin //irmove
        //Nothing
    end
    else if (icode == 4'd4) begin //rmmove
        valA = temp_memo[rA];
        valB = temp_memo[rB];
    end
    else if (icode == 4'd5) begin //mrmove
        valB = temp_memo[rB];
    end
    else if (icode == 4'd6) begin //operation
        valA = temp_memo[rA];
        valB = temp_memo[rB];
    end
    else if (icode == 4'd7) begin //jxx
        //Nothing
    end
    else if (icode == 4'd8) begin //call Dest
        valB = temp_memo[4'b0100];
    end
    else if (icode == 4'd9) begin //ret
        valA = temp_memo[4'b0100];
        valB = temp_memo[4'b0100];
    end
    else if (icode == 4'd10) begin //push
        valA = temp_memo[rA];
        valB = temp_memo[4'b0100];
    end
    else if (icode == 4'd11) begin //popq rA
        valA = temp_memo[4'b0100];
        valB = temp_memo[4'b0100];
    end
end

endmodule
```

Execute

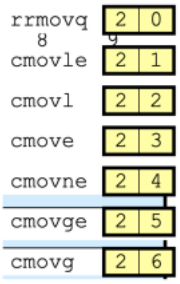
- The execute stage includes the ALU.
- According to the value of `icode` and `ifun`, we can decide the value to be assigned to `valE` accordingly.
- Units of Execute Stage:
  - `ALU`: Implements 4 required functions.Generates condition code values
  - `CC`: Register with 3 condition code bits

- `cond`: Computes conditional jump/move flag



- Control Logic:
  - Set CC: Should condition code register be loaded?
  - ALU A: Input A to ALU
  - ALU B: Input B to ALU
  - ALU fun: What function should ALU compute?
- Implementation:

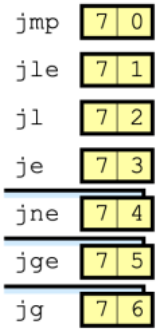
1. For `cmov` we need to check the following move conditions according to `ifun` and set `cnd` to 1 or leave it as zero accordingly:



Instruction		Synonym	Move condition	Description
cmove	S, R	cmovz	ZF	Equal / zero
cmovne	S, R	cmovnz	~ZF	Not equal / not zero
cmovs	S, R		SF	Negative
cmovns	S, R		~SF	Nonnegative
cmovg	S, R	cmovnle	~(SF ^ OF) & ~ZF	Greater (signed >)
cmovge	S, R	cmovnl	~(SF ^ OF)	Greater or equal (signed >=)
cmovl	S, R	cmovnge	SF ^ OF	Less (signed <)
cmovle	S, R	cmovng	(SF ^ OF) | ZF	Less or equal (signed <=)
cmova	S, R	cmovnbe	~CF & ~ZF	Above (unsigned >)
cmovae	S, R	cmovnb	~CF	Above or equal (Unsigned >=)
cmovb	S, R	cmovnae	CF	Below (unsigned <)
cmovbe	S, R	cmovna	CF | ZF	below or equal (unsigned <=)

2. For `irmovq`: `valE = valC`
3. For `mrmovq` and `rmmovq`: `valE=valB+valC`
4. For Operation or `OPq`, we should check the `ifun` values and use `ALU` to perform the desired operation and do `valE = ans` where ans is the output of `ALU`

5. For `jxx`, we should check the `ifun` value and check if the jump conditions are satisfies:



6. For `call` and `pushq` : valE=-64'd8+valB
7. For `ret` and `popq`: `valE=64'd8+valB`

- Code:

```
`include "./ALU/alu.v"

module exec_seq(input clk,
                input [3:0] icode,
                input [3:0] ifun,
                input [63:0] valA,
                input [63:0] valB,
                input [63:0] valC,
                output reg signed [63:0] valE,
                output reg cnd,
                output reg ZF,
                output reg SF,
                output reg OF);

wire signed [63:0] result;
wire overflow;
reg signed [63:0] inpA;
reg signed [63:0] inpB;
reg[1:0] select;
reg signed [63:0] ans;


alu alu_mod1(
    .control(select),
    .a(inpA),
    .b(inpB),
    .out(result),
    .overflow(overflow)
    );

reg cnd1,cnd2,xoro,ando,oro,noto,temp1;

always @(cnd1 or cnd2) begin
    begin
        ando = cnd1 & cnd2;
        oro = cnd1 | cnd2;
        xoro = cnd1 ^ cnd2 ;
        noto = ~cnd1;
    end
end

always @(*)
begin
    ZF = (result == 1'b0); // Output of ALU is zero
    //SF = (result < 1'd0); // Output of the ALU is negative
    if (result[63] == 1'b1) begin
        SF = 1;
    end
    else
    begin
        SF = 0;
    end
    OF = (inpA < 1'b0 == inpB < 1'b0) && (result < 64'b0 != inpA < 1'b0); // signed overflow flag
end

always @(*) begin

    cnd = 0;
    if (icode == 4'd2) begin //cmove

            case (ifun)
                4'd0://rrmove
                begin
                    cnd = 1;
                end
                4'd1://cmovle
                begin
                    cnd1 = SF; cnd2= OF;
                    temp1 = xoro;
                    cnd1 = ZF;
                    cnd1 = noto;
                    cnd2 = temp1;
                    if(oro)begin
                        cnd = 1;
                    end
            valE = result;
```

```
                end
                4'd2://cmovl
                begin
                    cnd1 = SF; cnd2= OF;
                    if(xoro)begin
                        cnd = 1;
                    end
            valE = result;

                end
                4'd3://cmove
                begin
                    if(ZF)begin
                        cnd = 1;
                    end
            valE = result;

                end
                4'd4://cmovne
                begin
                    cnd1 = ZF;
                    if(ZF)begin
                        cnd = 1;
                    end
            valE = result;

                end
                4'd5://cmovge
                begin
                    cnd1= SF;
                    cnd2=OF;
                    temp1 = xoro;
                    cnd = temp1;
                    if(noto)begin
                        cnd = 1;
                    end
            valE = result;

                end
                4'd6://cmovg
                begin
                    cnd1 = SF; cnd2= OF;
                    temp1 = xoro;
                    cnd1 = temp1;
                    if(noto)begin
                        cnd = 1;
                    end
            valE = result;

                end
            endcase
            inpA = valA;
            inpB = 64'd0;
            select = 2'd0;
            valE = result;
    end
    else if (icode == 4'd3) begin //irmove
        inpA = 64'b0;
        inpB = valC;
        select = 2'd0;
        valE = result;
    end
    else if (icode == 4'd4) begin //rmmove
        inpA = valB;
        inpB = valC;
        select = 2'd0;
        valE = result;
    end
    else if (icode == 4'd5) begin //mrmove
        inpA = valB;
        inpB = valC;
        select = 2'd0;
        valE = result;
    end
    else if (icode == 4'd6) begin //operation
        inpA = valA;
            inpB = valB;
            case (ifun)
                4'd0://addq
                begin
                    select = 2'd0;
                end
                4'd1://sub
                begin
                    select = 2'd1;
                end
                4'd2://and
```

```
                begin
                    select = 2'd2;
                end
                4'd3:
                begin
                    select = 2'd3;
                end
            endcase
            valE = result;
    end
    else if (icode == 4'd7) begin //jxx
        case (ifun)
            4'd0://jmp
            begin
                cnd = 1;
            end
            4'd1://jle
            begin
                cnd1 = SF;
                cnd2 = OF;
                temp1 = xoro;
                cnd1 = temp1;
                cnd2 = ZF;
                if(oro)begin
                    cnd = 1;
                end
            end
            4'd2://jl
            begin
                cnd1= SF;
                cnd2 =OF;
                if(xoro)
                begin
                    cnd = 1;
                end
            end
            4'd3://je
            begin
                if(ZF)
                begin
                    cnd = 1;
                end
            end
            4'd4://jne
            begin
                cnd1 = ZF;
                if(noto)begin
                    cnd = 1;
                end
            end
            4'd5://jge
            begin
                cnd1 = SF;
                cnd2 = OF;
                temp1 = xoro;
                cnd1 = temp1;
                if(noto)begin
                    cnd = 1;
                end
            end
            4'd6://jg
            begin
                cnd1 = SF;
                cnd2 = OF;
                temp1 = xoro;
                cnd1 = temp1;
                if(noto)begin
                    cnd = 1;
                end
            end
    endcase
    end
    else if (icode == 4'd8) begin //call Dest
        // valE = valB - 8
            inpA = -64'd8;
            inpB = valB;
            select = 2'b00; // to decrement the stack pointer by 8 on call
            valE = result;
    end
    else if (icode == 4'd9) begin //ret
        // valE = valB + 8
            inpA = 64'd8;
            inpB = valB;
            select = 2'b00; // to increment the stack pointer by 8 on ret
            valE = result;
    end
    else if (icode == 4'd10) begin //pushq rA
```

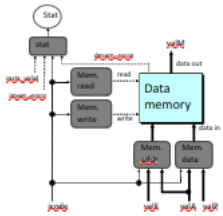
```
        // valE = valB - 8
            inpA = -64'd8;
            inpB = valB;
            select = 2'b00; // to decrement the stack pointer by 8 on pushq
            valE = result;
    end
    else if (icode == 4'd11) begin //popq rA
        // valE = valB + 8
            inpA = 64'd8;
            inpB = valB;
            select = 2'b00; // to increment the stack pointer by 8 on popq
            valE = result;
    end
end

endmodule
```

Memory

- Responsible for reading and writing to memory.
- We declared the data memory as a register array:

```
reg [63:0] memory [0:127];
```

- Based on the `icode` we can decide whether we are required to read or write from or to memory respectively.
- Based on the `icode` we can decide whether we are required to read or write from or to memory respectively.
- The data address is calculated using `icode`, `valE` and `valA` and the data input is calculated using `icode`, `valA` and `valP`.
- Control Logic:
  - stat: What is instruction status?
  - Mem. read: should word be read?
  - Mem. write: should word be written?
  - Mem. addr.: Select address
  - Mem. data.: Select data



- Code:

```
`timescale 1ns/1ps

module memory_seq (
    input clk,
    input wire [3:0] icode,
    input wire [63:0] valA,
    input wire [63:0] valB,
    input wire [63:0] valP,
    input wire [63:0] valE,
    output reg [63:0] valM
);
    reg [63:0] memory [0:127];

    initial begin
        memory[1] = 64'b1;
    end

    always@(*)
    begin
        //memory[0] = 64'b2;
        if(icode == 4'b0101) // mrmovq
```

```
        begin
            valM = memory[valE];
        end
        else if(icode == 4'b1001) //ret
        begin
            valM = memory[valA];
        end
        else if(icode == 4'b1011) //popq
        begin
            valM = memory[valA];
        end
    end

    always@(posedge clk)
    begin
        if(icode == 4'b0100) // rmmovq
        begin
            memory[valE] = valA;
        end
        else if(icode == 4'b1000) //call
        begin
            memory[valE] = valP;
        end
        else if(icode == 4'b1010) //pushq
        begin
            memory[valE] = valA;
        end
    end
endmodule
```

Write Back

- Writes up to two results to the register file.
- We created 15 register arrays named `reg_file0` to `reg_file14` as outputs.
- We also created a register `temp_memo` which represents the register memory of the processor.

```
reg [63:0] temp_memo[0:14];
```

- Implementation:
    1. `irmove`: `temp_memo[rB] = valE`
    2. `mrmove`: `temp_memo[rA] = valE`
    3. `cmove`: `temp_memo[rB] = valE`
    4. `opq`: `temp_memo[rB] = valE`
    5. `push`: `temp_memo[4'b0100] = valE`
    6. `pop`:

        ```
        temp_memo[4'b0100] = valE;
        temp_memo[rA] = valM;
        ```

    7. `call`: `temp_memo[4'b0100] = valE`
    8. `ret`: `temp_memo[4'b0100] = valE`

- Code:

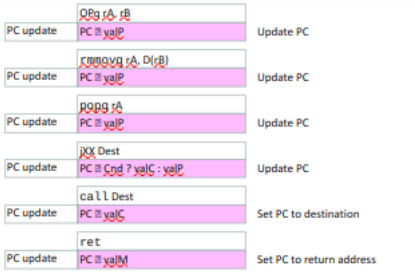
```
module wb_seq(clk, icode, rA, rB, valE, valM, reg_file0, reg_file1, reg_file2, reg_file3, reg_file4, reg_file5, reg_file6, reg_file7,
    input clk;
    input [3:0] icode;
    input [3:0] rA;
    input [3:0] rB;
    input [63:0] valE,valM ;

    output reg signed [63:0] reg_file0;
    output reg signed [63:0] reg_file1;
    output reg signed [63:0] reg_file2;
    output reg signed [63:0] reg_file3;
```

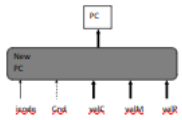
```
    output reg signed [63:0] reg_file4;
    output reg signed [63:0] reg_file5;
    output reg signed [63:0] reg_file6;
    output reg signed [63:0] reg_file7;
    output reg signed [63:0] reg_file8;
    output reg signed [63:0] reg_file9;
    output reg signed [63:0] reg_file10;
    output reg signed [63:0] reg_file11;
    output reg signed [63:0] reg_file12;
    output reg signed [63:0] reg_file13;
    output reg signed [63:0] reg_file14;

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

    initial begin
        temp_memo[0] = 0;
        temp_memo[1] = 0;
        temp_memo[2] = 0;
        temp_memo[3] = 0;
        temp_memo[4] = 0;
        temp_memo[5] = 0;
        temp_memo[6] = 0;
        temp_memo[7] = 0;
        temp_memo[8] = 0;
        temp_memo[9] = 0;
        temp_memo[10] =0;
        temp_memo[11] =0;
        temp_memo[12] =0;
        temp_memo[13] =0;
        temp_memo[14] =0;

    end

    always @(*) begin
        reg_file0 = temp_memo[0];
        reg_file1 = temp_memo[1];
        reg_file2 = temp_memo[2];
        reg_file3 = temp_memo[3];
        reg_file4 = temp_memo[4];
        reg_file5 = temp_memo[5];
        reg_file6 = temp_memo[6];
        reg_file7 = temp_memo[7];
        reg_file8 = temp_memo[8];
        reg_file9 = temp_memo[9];
        reg_file10 = temp_memo[10];
        reg_file11 = temp_memo[11];
        reg_file12 = temp_memo[12];
        reg_file13 = temp_memo[13];
        reg_file14 = temp_memo[14];

        end

    always @(posedge clk) begin


                    case (icode)

                    4'b0000: //halt
                    begin

                    end

                    4'b0001://nop
                    begin
                    end

                    4'b0011://irmove
                    begin
                      temp_memo[rB] = valE;
                    end

                    4'b0101://mrmove
                    begin
                          temp_memo[rA] = valE;
                    end

                    4'b0100://rmmove
                    begin
                    end

                    4'b0010://cmove
                    begin
                        temp_memo[rB] = valE;
                    end

                     4'b0110://opq
                    begin
```

```
                    temp_memo[rB] = valE;
                end

                 4'b1010://push
                begin
                    temp_memo[4'b0100] = valE;
                end
                4'b1011://pop
                begin
                    temp_memo[4'b0100] = valE;
                    temp_memo[rA] = valM;
                end

                 4'b0111://jxx
                begin

                end
                 4'b1000://call
                begin
                   temp_memo[4'b0100] = valE;
                end
                 4'b1001://ret
                begin
                    temp_memo[4'b0100] = valE;
                end

                endcase
    end
endmodule
```

PC Update

- The PC update module is responsible for finding the next value of PC after an instruction has finished executing.
- The PC is updated by the instruction length in bytes for all except `jXX`, call and `ret`.





- Implementation:

1. `jxx`: If `cnd = 1`: `PC_updated` = `valC` or `PC_updated = valP`.
2. `call`: `PC_updated = valC`
3. `ret` : `PC_updated = valM`
4. default case: `PC_updated = valP`

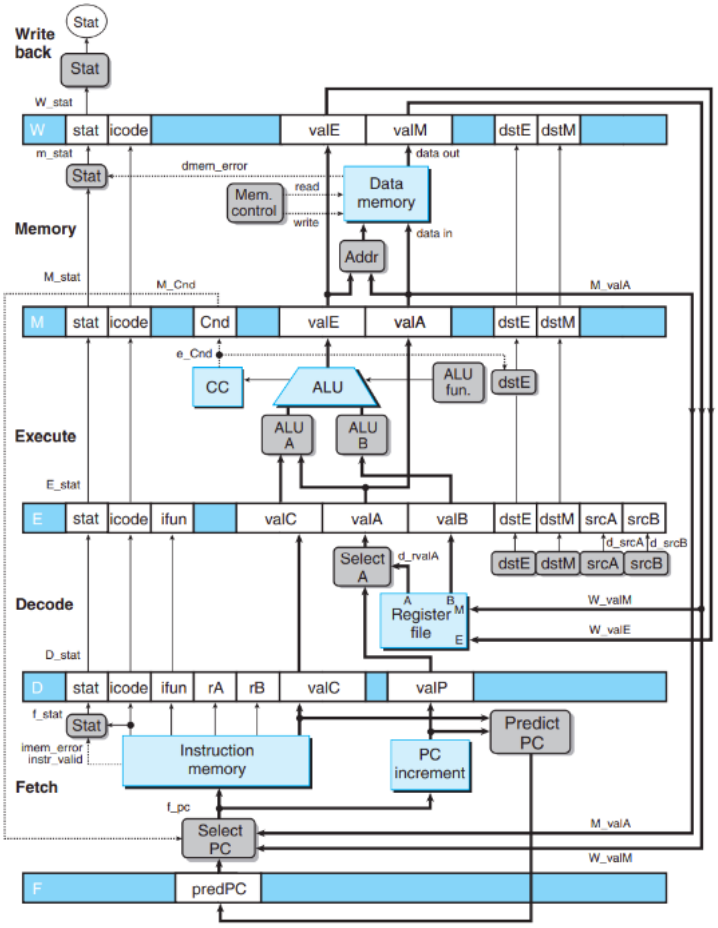
- Code:

```
`timescale 1ns/1ps

module pc_update (
    input clk,
    input cnd,
    input [3:0] icode,
    input [63:0] valC,
    input [63:0] valM,
    input [63:0] valP,
```

```
    output reg [63:0] PC_updated
);
    always@(*)
    begin
        if(icode == 4'b0111) // jxx or jumps
        begin
            if(cnd == 1'b1)
            begin
                PC_updated = valC;
            end
            else
            begin
                PC_updated = valP;
            end
        end

        else if(icode == 4'b1000) // call
        begin
            PC_updated = valC;
        end

        else if(icode == 4'b1001) // ret
        begin
            PC_updated = valM;
        end

        else // default case
        begin
            PC_updated = valP;
        end
    end
endmodule
```

Pipeline

Hardware structure of PIPE:



Changes for Pipeline

Rearranging Stages:

- The PC update stage in the SEQ implementation is the last stage in the cycle of an instruction.
- For the pipelined implementation we should bring the PC update stage to the beginning of the cycle as we want to be able to continuously fetch the next instruction without having to wait for the PC update stage of the previous instruction to end had it been at the end of the cycle. This is known as circuit retiming. This changes the general sentation of the circuit without affecting its local behavior. This also allows us to balance the delays between stages in the pipelined system.
- Now the PC update stage at the beginning of the cycle can keep providing updated PC values to the fetch stage using the required values from different stages from instructions that have passed that stage

Inserting Pipeline Registers

- The next step to pipelining is inserting the pipeline registers.
- We know that in a pipelined implementation we rearrange some of the hardware and signals in the SEQ implmentation and insert pipeline register between each stage.
- These registers stop the signals from one stage from flowing into the next stage and affecting the processing happening there.
- F the register inserted before the fetch stage holds a predicted value of the program counter.
- D sits between the fetch and decode stages. It holds information about the most recently fetched instruction for processing by the decode stage.
- E 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 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 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.

Rearranging and Relabelling signals

- In the pipelined implementation we will have all the signals of an instruction pass through every stage one by one and these will have to be names with respect to the stage it is currently in as it is not possible to have one signal icode and have ti account for all the 5 instructions running at the same time.
- So we maintain the signal at each stage and label them with respect to the stage as f_icode,d_icode,w_icode,etc.

Processor

Defining and initialising inputs and outputs:

```
//_________________FETCH_____________________
  // F stage registers
  reg [63:0] F_predPC = 0;

  // Fetch Stage output
  wire [63:0] f_valC;
  wire [63:0] f_valP;
  wire [63:0] f_predPC;
  wire [2:0] f_stat;
  wire [3:0] f_icode;
  wire [3:0] f_ifun;
  wire [3:0] f_rA;
  wire [3:0] f_rB;
  //_________________DECODE_____________________
  // D stage registers
  reg [3:0] D_icode = 1;
  reg [3:0] D_ifun = 0;
  reg [2:0] D_stat = 1;
  reg [63:0] D_valC = 0;
```

```
reg [63:0] D_valP = 0;
reg [3:0] D_rA = 0;
reg [3:0] D_rB = 0;

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


  wire signed[63:0] reg_file0;
  wire signed[63:0] reg_file1;
  wire signed[63:0] reg_file2;
  wire signed[63:0] reg_file3;
  wire signed[63:0] reg_file4;
  wire signed[63:0] reg_file5;
  wire signed[63:0] reg_file6;
  wire signed[63:0] reg_file7;
  wire signed[63:0] reg_file8;
  wire signed[63:0] reg_file9;
  wire signed[63:0] reg_file10;
  wire signed[63:0] reg_file11;
  wire signed[63:0] reg_file12;
  wire signed[63:0] reg_file13;
  wire signed[63:0] reg_file14;


//__________________EXECUTE__________________________
//E stage registers
reg [2:0] E_stat = 1;
reg [3:0] E_icode = 1;
reg [3:0] E_ifun = 0;
reg [3:0] E_dstE = 0;
reg [3:0] E_dstM = 0;
reg [3:0] E_srcA = 0;
reg [3:0] E_srcB = 0;
reg [63:0] E_valC = 0;
reg [63:0] E_valA = 0;
reg [63:0] E_valB = 0;
reg [63:0] E_valP = 0;


// Execute stage output
wire e_Cnd;
wire [2:0] e_stat;
wire [3:0] e_icode;
wire [3:0] e_dstE;
wire [3:0] e_dstM;
wire [63:0] e_valE;
wire [63:0] e_valA;
wire [63:0] e_valC;

wire ZF;
wire SF;
wire OF;
//__________________MEMORY___________________________
//M stage register
reg M_Cnd = 0;
reg [2:0] M_stat = 1;
reg [3:0] M_icode = 1;
reg [3:0] M_dstE = 0;
reg [3:0] M_dstM = 0;
reg [63:0] M_valE = 0;
reg [63:0] M_valA = 0;
reg [63:0] M_valC = 0;


// Memory stage output
wire [2:0] m_stat;
wire [3:0] m_icode;
wire [63:0] m_valE;
wire [63:0] m_valM;
wire [3:0] m_dstE;
wire [3:0] m_dstM;


//__________________WRITEBACK___________________________
// W stage register
reg [2:0] W_stat = 1;
reg [3:0] W_icode = 1;
```

```
  reg [3:0] W_dstE = 0;
  reg [3:0] W_dstM = 0;
  reg [63:0] W_valE = 0;
  reg [63:0] W_valM = 0;
  reg [63:0] W_valP = 0;
```

Integration of Modules:

```
fetch fetch_stage(
    .F_predPC(F_predPC),

    .M_valA(M_valA),
    .M_icode(M_icode),
    .M_Cnd(M_Cnd),

    .W_icode(W_icode),
    .W_valM(W_valM),

    .f_stat(f_stat),
    .f_icode(f_icode),
    .f_ifun(f_ifun),
    .f_rA(f_rA),
    .f_rB(f_rB),
    .f_valC(f_valC),
    .f_valP(f_valP),
    .f_predPC(f_predPC)
);
  decode decode_stage(
    .clk(clk),
    .D_stat(D_stat),
    .D_icode(D_icode),
    .D_ifun(D_ifun),
    .D_rA(D_rA),
    .D_rB(D_rB),
    .D_valC(D_valC),
    .D_valP(D_valP),

    .e_dstE(e_dstE),
    .e_valE(e_valE),

    .M_dstE(M_dstE),
    .M_valE(M_valE),
    .M_dstM(M_dstM),
    .m_valM(m_valM),

    .W_dstM(W_dstM),
    .W_valM(W_valM),
    .W_dstE(W_dstE),
    .W_valE(W_valE),

    .d_valA(d_valA),
    .d_valB(d_valB),
    .d_dstE(d_dstE),
    .d_stat(d_stat),
    .d_icode(d_icode),
    .d_ifun(d_ifun),
    .d_valC(d_valC),
    .d_dstM(d_dstM),
    .d_srcA(d_srcA),
    .d_srcB(d_srcB),

    .reg_file0(reg_file0),
    .reg_file1(reg_file1),
    .reg_file2(reg_file2),
    .reg_file3(reg_file3),
    .reg_file4(reg_file4),
    .reg_file5(reg_file5),
    .reg_file6(reg_file6),
    .reg_file7(reg_file7),
    .reg_file8(reg_file8),
    .reg_file9(reg_file9),
    .reg_file10(reg_file10),
    .reg_file11(reg_file11),
    .reg_file12(reg_file12),
    .reg_file13(reg_file13),
    .reg_file14(reg_file14)
  );
  execute execute_stage(
    .clk(clk),

    .E_stat(E_stat),
    .E_icode(E_icode),
```

```
    .E_ifun(E_ifun),
    .E_valC(E_valC),
    .E_valA(E_valA),
    .E_valB(E_valB),
    .E_dstE(E_dstE),
    .E_dstM(E_dstM),
    .W_stat(W_stat),
    .m_stat(m_stat),

    .e_stat(e_stat),
    .e_icode(e_icode),
    .e_Cnd(e_Cnd),
    .e_valE(e_valE),
    .e_valA(e_valA),
    .e_dstE(e_dstE),
    .e_dstM(e_dstM),
    .ZF(ZF),
    .SF(SF),
    .OF(OF)
    );


  memory m(
    .clk(clk),

    .M_stat(M_stat),
    .M_icode(M_icode),
    .M_valE(M_valE),
    .M_valA(M_valA),
    .M_dstE(M_dstE),
    .M_dstM(M_dstM),

    .m_stat(m_stat),
    .m_icode(m_icode),
    .m_valE(m_valE),
    .m_valM(m_valM),
    .m_dstE(m_dstE),
    .m_dstM(m_dstM)
    );

wire W_stall,F_stall, M_bubble, E_bubble, D_bubble, D_stall;

  control control_stage(
    // Inputs
    .D_icode(D_icode),
    .d_srcA(d_srcA),
    .d_srcB(d_srcB),
    .E_icode(E_icode),
    .E_dstM(E_dstM),
    .e_Cnd(e_Cnd),
    .M_icode(M_icode),//Wire or reg
    .m_stat(m_stat),
    .W_stat(W_stat),//Wire or reg

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

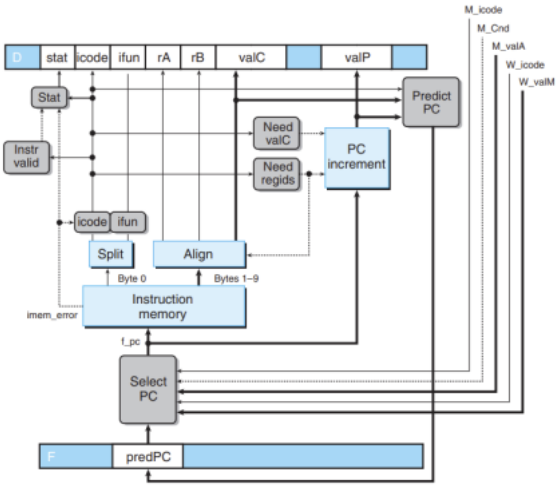
Updating the registers at positive edge of clock depending on the value of pipeline control signals of stall and bubble:

```
always @(posedge clk) begin
  if (F_stall == 0) begin
    F_predPC <= f_predPC;
  end
end

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

```
      D_icode <= 1;
      D_ifun <= 0;
      D_rA <= 0;
      D_rB <= 0;
      D_valC <= 0;
      D_valP <= 0;
    end
  end
end


always @(posedge clk) begin
    if (E_bubble == 0) begin
      E_stat <= d_stat;
      E_icode <= d_icode;
      E_ifun <= d_ifun;
      E_srcA <= d_srcA;
      E_srcB <= d_srcB;
      E_valC <= d_valC;
      E_valA <= d_valA;
      E_valB <= d_valB;
      E_dstE <= d_dstE;
      E_dstM <= d_dstM;
    end
    else begin
      E_stat <= 1;
      E_icode <= 1;
      E_ifun <= 0;
      E_srcA <= 0;
      E_srcB <= 0;
      E_valC <= 0;
      E_valA <= 0;
      E_valB <= 0;
      E_dstE <= 0;
      E_dstM <= 0;
    end

end

always @(posedge clk) begin
    if (M_bubble == 0) begin
      M_stat <= e_stat;
      M_icode <= e_icode;
      M_Cnd <= e_Cnd;
      M_valC <= e_valC;
      M_valA <= e_valA;
      M_dstE <= e_dstE;
      M_dstM <= e_dstM;
      M_valE <= e_valE;
    end
    else begin
      M_stat <= 1;
      M_icode <= 1;
      M_Cnd <= 0;
      M_valE <= 0;
      M_valA <= 0;
      M_dstE <= 0;
      M_dstM <= 0;
    end

end

always @(posedge clk) begin
 status_code = W_stat;

    if (W_stall == 0) begin
        W_stat <= m_stat;
        W_icode <= m_icode;
        W_valE <= m_valE;
        W_valM <= m_valM;
        W_dstE <= m_dstE;
        W_dstM <= m_dstM;
    end
end
```

Fetch Module

In the fetch stage, the predicted PC from F is sent to the select PC block on the positive edge of the clock, which also has several inputs from the later stages (of an earlier instruction) to correctly calculate the correct PC value for an instruction to be fetched. If the predicted PC value is correct, it is passed to the fetch stage, or it is calculated from these later inputs. The required instruction is fetched and the appropriate values needed by the decode stage are sent to be stored in D.



Instruction Set:

```
reg [7:0] inst_memo[0:127];
reg [0:7] temp;
reg invalid_instr, memory_error, halt;

initial begin
    halt = 0;
inst_memo[0] = 8'h10;
        inst_memo[1]  = 8'h10; //nop
// inst_memo[2] = 8'h10; //halt

inst_memo[2] = 8'h20; //rrmovq
inst_memo[3] = 8'h12;
// inst_memo[4]= 8'h00;

inst_memo[4] = 8'h30; //irmovq
inst_memo[5] = 8'hF2;
inst_memo[6] = 8'h00;
inst_memo[7] = 8'h00;
inst_memo[8] = 8'h00;
inst_memo[9] = 8'h00;
inst_memo[10] = 8'h00;
inst_memo[11] = 8'h00;
inst_memo[12] = 8'h00;
inst_memo[13] = 8'b00000010;
// inst_memo[14]=8'h00;

inst_memo[14] = 8'h40; //rmmovq
inst_memo[15] = 8'h24;
{inst_memo[16],inst_memo[17],inst_memo[18],inst_memo[19],inst_memo[20],inst_memo[21],inst_memo[22],inst_memo[23]} = 64'd1;
// inst_memo[24]=8'h00;

inst_memo[24] = 8'h40; //rmmovq
inst_memo[25] = 8'h53;
{inst_memo[26],inst_memo[27],inst_memo[28],inst_memo[29],inst_memo[30],inst_memo[31],inst_memo[32],inst_memo[33]} = 64'd0;
// inst_memo[34] = 8'h00;
inst_memo[34] = 8'h50; //mrmovq
inst_memo[35] = 8'h53;
{inst_memo[36],inst_memo[37],inst_memo[38],inst_memo[39],inst_memo[40],inst_memo[41],inst_memo[42],inst_memo[43]} = 64'd0;
// inst_memo[44]=8'h00;

inst_memo[44] = 8'h60; //opq
inst_memo[45] = 8'h9A;
        // inst_memo[46]  = 8'h10; //nop
// inst_memo[46] = 8'h00; //halt



inst_memo[46] = 8'h73; //jmp
// {inst_memo[47],inst_memo[48],inst_memo[49],inst_memo[50],inst_memo[51],inst_memo[52],inst_memo[53],inst_memo[54]} = 64'd56;
inst_memo[54]=8'd56;
{inst_memo[47],inst_memo[48],inst_memo[49],inst_memo[50],inst_memo[51],inst_memo[52],inst_memo[53]}=56'd0;

inst_memo[55] = 8'h00; //halt

inst_memo[56] = 8'hA0; // pushq
inst_memo[57] = 8'h9F;

inst_memo[58] = 8'hB0; //popq
```

```
inst_memo[59] = 8'h9F;

inst_memo[60] = 8'h80; //call
{inst_memo[61],inst_memo[62],inst_memo[63],inst_memo[64],inst_memo[65],inst_memo[66],inst_memo[67],inst_memo[68]} = 64'd80;

inst_memo[69] = 8'h60; //opq
inst_memo[70] = 8'h56;

// inst_memo[71] = 8'h00;
inst_memo[71] = 8'h72; //jmp
{inst_memo[72],inst_memo[73],inst_memo[74],inst_memo[75],inst_memo[76],inst_memo[77],inst_memo[78],inst_memo[79]} = 64'd46;

inst_memo[80]=8'h10;
inst_memo[81] = 8'h63; //opq
inst_memo[82] = 8'hDE;
// inst_memo[80] = 8'h10;

inst_memo[83]=8'h90;
end
```

Source Code:

```
            always @(*) begin
                //memory_error = 0;
                if(f_PC >102)
                begin
                    //memory_error = 1;
                end
                f_icode = inst_memo[f_PC][7:4];
                f_ifun = inst_memo[f_PC][3:0];

                if(f_icode >= 4'b0000 && f_icode <4'b1100)
                begin

                   // invalid_instr = 0;

                    case (f_icode)

                    4'b0000: //halt
                    begin
                        f_rA = 15;
                        f_rB = 15;
                        halt = 1;
                    end

                    4'b0001://nop
                    begin
                        f_rA = 15;
                        f_rB = 15;
                        f_valC = 0;
                        f_valP = f_PC + 64'd1;
                    end

                    4'b0011://irmove
                    begin
                        temp = {inst_memo[f_PC+1]};
                        f_rA = temp[0:3];
                        f_rB = temp[4:7];
                        f_valC = { inst_memo[f_PC+2],
                        inst_memo[f_PC+3],inst_memo[f_PC+4],
                        inst_memo[f_PC+5],inst_memo[f_PC+6],
                        inst_memo[f_PC+7],inst_memo[f_PC+8],
                        inst_memo[f_PC+9]
                        };

                         f_valP = f_PC + 64'd10;
                    end

                    4'b0101://mrmove
                    begin
                           temp = {inst_memo[f_PC+1]};
                        f_rA = temp[0:3];
                        f_rB = temp[4:7];
                        f_valC = { inst_memo[f_PC+2],
                        inst_memo[f_PC+3],inst_memo[f_PC+4],
                        inst_memo[f_PC+5],inst_memo[f_PC+6],
                        inst_memo[f_PC+7],inst_memo[f_PC+8],
                        inst_memo[f_PC+9]
                        };

                         f_valP = f_PC + 64'd10;
                    end
```

```
                4'b0100://rmmove
                begin
                      temp = {inst_memo[f_PC+1]};
                    f_rA = temp[0:3];
                    f_rB = temp[4:7];
                     f_valC = { inst_memo[f_PC+2],
                    inst_memo[f_PC+3],inst_memo[f_PC+4],
                    inst_memo[f_PC+5],inst_memo[f_PC+6],
                    inst_memo[f_PC+7],inst_memo[f_PC+8],
                    inst_memo[f_PC+9]
                    };

                     f_valP = f_PC + 64'd10;
                end

                4'b0010://cmove
                begin
                      temp = {inst_memo[f_PC+1]};
                    f_rA = temp[0:3];
                    f_rB = temp[4:7];
                    f_valC = 0;
                     f_valP = f_PC + 64'd2;
                end

                 4'b0110://opq
                begin
                      temp = {inst_memo[f_PC+1]};
                    f_rA = temp[0:3];
                    f_rB = temp[4:7];
                    f_valC = 0;
                     f_valP = f_PC + 64'd2;
                end

                 4'b1010://push
                begin
                      temp = {inst_memo[f_PC+1]};
                    f_rA = temp[0:3];
                    f_rB = temp[4:7];

                     f_valP = f_PC + 64'd2;
                end 4'b1011://pop
                begin
                       temp = {inst_memo[f_PC+1]};
                    f_rA = temp[0:3];
                    f_rB = temp[4:7];
                    f_valC = 0;

                     f_valP = f_PC + 64'd2;
                end

                 4'b0111://jxx
                begin
                    f_rA = 15;
                    f_rB = 15;
                    f_valC = { inst_memo[f_PC+1],
                    inst_memo[f_PC+2],inst_memo[f_PC+3],
                    inst_memo[f_PC+4],inst_memo[f_PC+5],
                    inst_memo[f_PC+6],inst_memo[f_PC+7],
                    inst_memo[f_PC+8]
                    };

                     f_valP = f_PC + 64'd9;
                end
                 4'b1000://call
                begin
                    f_rA = 15;
                    f_rB = 15;
                    f_valC = { inst_memo[f_PC+1],
                    inst_memo[f_PC+2],inst_memo[f_PC+3],
                    inst_memo[f_PC+4],inst_memo[f_PC+5],
                    inst_memo[f_PC+6],inst_memo[f_PC+7],
                    inst_memo[f_PC+8]
                    };
                     f_valP = f_PC + 64'd9;
                end
                 4'b1001://ret
                begin
                    f_rA = 15;
                    f_rB = 15;
                    f_valC = 0;
                    f_valP= f_PC + 64'd1;
                end

                endcase
            end
            else
```

```
                begin
                    //invalid_instr = 1;
                end
            end
```

Predict PC Module

```
//To predict next PC value
module pred (
  input [3:0] f_icode,
    input [63:0] f_valC,
    input [63:0] f_valP,
    // Outputs
    output reg[63:0] f_predPC
);

     always @(*) begin

        if (f_icode == 4'b0111 || f_icode == 4'b1000 ) begin

            f_predPC <= f_valC;
        end
        else begin

            f_predPC <= f_valP;
        end
    end

    // always @(*) begin
    //     if
    // end
endmodule
```

Instruction Validity and Memory Error

```
always @(*) begin
    if(f_icode >= 4'b0000 && f_icode <4'b1100)
    begin
        invalid_instr = 0;
    end
    else begin
        invalid_instr = 1;
    end
    memory_error = 0;
    if(f_PC >102)
    begin
        memory_error = 1;
    end
end
```

Select Module

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

    // Outputs
    output reg[63:0] f_PC
);

 always @(*) begin

        if (M_icode == 4'b0111 && M_Cnd == 0) begin
            f_PC <= M_valA;
        end
        else if (W_icode == 9) begin

            f_PC <= W_valM;
        end
        else begin

            f_PC <= F_predPC;
```

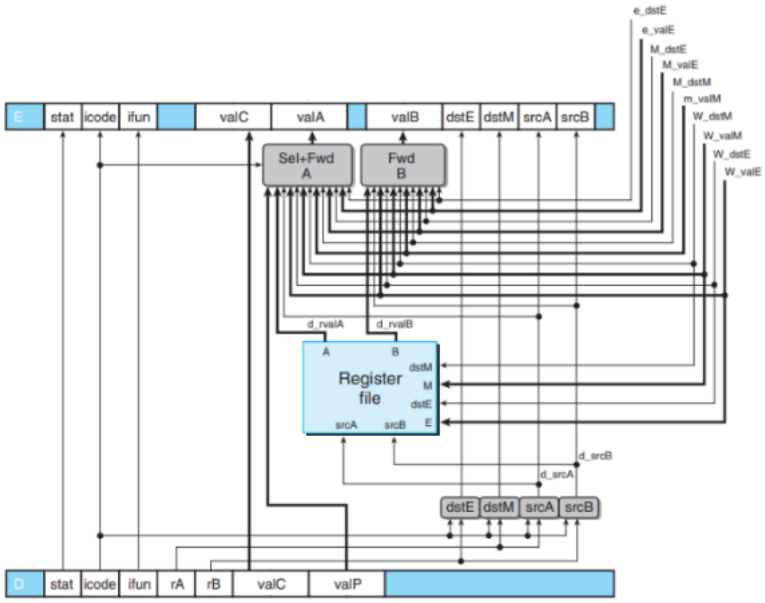
```
        end
    end
endmodule
```

Status Module

```
module Stat(
    // Inputs
    input invalid_instr,
    input memory_error,
    input [3:0] f_icode,

    // Outputs
    output reg[2:0] f_stat
);

    always @(*) begin

        if (memory_error == 1) begin

            f_stat <= 3;
        end
        else if (invalid_instr == 1) begin
            f_stat <= 4;
        end
        else if (f_icode == 0) begin

            f_stat <= 2;
        end
        else begin

            f_stat <= 1;
        end

    end
endmodule
```

Decode

In the decode stage, the instruction is decoded and the required information is sent from D to E. This stage is the one where data forwarding is implemented, which often helps with prevention of loss of data. Since Verilog doesn't allow us to pass 2-dimensional arrays through function calls, all registers are sent separately.



Registers:

```
initial begin
    temp_memo[0] = 64'd12;        //rax
    temp_memo[1] = 64'd10;        //rcx
    temp_memo[2] = 64'd101;       //rdx
    temp_memo[3] = 64'd3;         //rbx
    temp_memo[4] = 64'd254;       //rsp
    temp_memo[5] = 64'd50;        //rbp
    temp_memo[6] = -64'd143;      //rsi
    temp_memo[7] = 64'd10000;     //rdi
    temp_memo[8] = 64'd990000;    //r8
    temp_memo[9] = -64'd12345;    //r9
    temp_memo[10] = 64'd12345;    //r10
    temp_memo[11] = 64'd10112;    //r11
    temp_memo[12] = 64'd0;        //r12
    temp_memo[13] = 64'd1567;     //r13
    temp_memo[14] = 64'd8643;     //r14
    temp_memo[15] = 64'd0;        // random register
end

     always @(posedge clk) begin

        temp_memo[W_dstM] <= W_valM;
        temp_memo[W_dstE] <= W_valE;

        reg_file0 = temp_memo[0];
        reg_file1 = temp_memo[1];
        reg_file2 = temp_memo[2];
        reg_file3 = temp_memo[3];
        reg_file4 = temp_memo[4];
        reg_file5 = temp_memo[5];
        reg_file6 = temp_memo[6];
        reg_file7 = temp_memo[7];
        reg_file8 = temp_memo[8];
        reg_file9 = temp_memo[9];
        reg_file10 = temp_memo[10];
        reg_file11 = temp_memo[11];
        reg_file12 = temp_memo[12];
        reg_file13 = temp_memo[13];
        reg_file14 = temp_memo[14];
    end
```

D Block:

```
always @(*) begin

        d_stat <= D_stat;
        d_icode <= D_icode;
        d_ifun <= D_ifun;
        d_valC <= D_valC;
end
```

Source Code:

```
    always @(*) begin

    if (D_icode == 4'd1) begin //cmove
     d_srcA <= 15;
      d_srcB <= 15;
      d_dstE <= 15;
      d_dstM <= 15;

        d_valA_temp = temp_memo[d_srcA];
        d_valB_temp = temp_memo[d_srcB];
    end
    else if (D_icode == 4'd2) begin //cmove
         d_srcA <= D_rA;
      d_srcB <= 15;
       d_dstE <= D_rB;
       d_dstM <= 15;

        d_valA_temp = temp_memo[d_srcA];
        d_valB_temp = temp_memo[d_srcB];
    end
    else if (D_icode == 4'd3) begin //irmove
    d_srcA <= 15;
      d_srcB <= 15;
       d_dstE <= D_rB;
       d_dstM <= 15;
```

```
        d_valA_temp = temp_memo[d_srcA];
        d_valB_temp = temp_memo[d_srcB];
        //Nothing
    end
    else if (D_icode == 4'd4) begin //rmmove
       d_srcA <= D_rA;
       d_srcB <= D_rB;
d_dstE <= 15;
d_dstM <= 15;
        d_valA_temp = temp_memo[d_srcA];
        d_valB_temp = temp_memo[d_srcB];
    end
    else if (D_icode == 4'd5) begin //mrmove
        d_srcA <= 15;
      d_srcB <= D_rB;
      d_dstE <= 15;
      d_dstM <= D_rA;
        d_valA_temp = temp_memo[d_srcA];
        d_valB_temp = temp_memo[d_srcB];
    end
    else if (D_icode == 4'd6) begin //opeD_rAtion
     d_srcA <= D_rA;
       d_srcB <= D_rB;
        d_dstE <= D_rB;
        d_dstM <= 15;

        d_valA_temp = temp_memo[d_srcA];
        d_valB_temp = temp_memo[d_srcB];
    end
    else if (D_icode == 4'd7) begin //jxx
        d_srcA <= 15;
        d_srcB <= 15;
        d_dstE <= 15;
        d_dstM <= 15;
        d_valA_temp = temp_memo[d_srcA];
        d_valB_temp = temp_memo[d_srcB];
        //Nothing
    end
    else if (D_icode == 4'd8) begin //call Dest
       d_srcA <= 15;
        d_srcB <= 4;
        d_dstE <= 4;
        d_dstM <= 15;
        d_valA_temp = temp_memo[d_srcA];
        d_valB_temp = temp_memo[d_srcB];

    end
    else if (D_icode == 4'd9) begin //ret
       d_srcA <= 4;
      d_srcB <= 4;
  d_dstE <= 4;
  d_dstM <= 15;
        d_valA_temp = temp_memo[d_srcA];
        d_valB_temp = temp_memo[d_srcB];

    end
    else if (D_icode == 4'd10) begin //push
        d_srcA <= 4;
     d_srcB <= 4;
  d_dstE <= 4;
  d_dstM <= 15;
        d_valA_temp = temp_memo[d_srcA];
        d_valB_temp = temp_memo[d_srcB];

    end
    else if (D_icode == 4'd11) begin //popq D_rA
     d_srcA <= 4;
      d_srcB <= 4;
  d_dstE <= 4;
  d_dstM <= D_rA;
        d_valA_temp = temp_memo[d_srcA];
        d_valB_temp = temp_memo[d_srcB];
    end

    end
```

Sel + Fwd A block

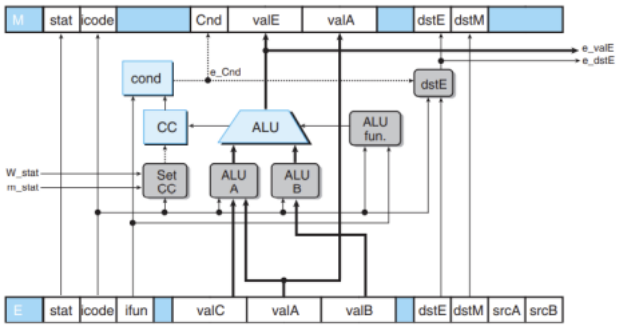
```
     // Sel+Fwd A Block
    always @(*) begin

        if (D_icode == 4'd7 || D_icode == 4'd8) begin

            d_valA = D_valP;
        end
        else if (d_srcA == e_dstE) begin

            d_valA = e_valE;
        end
        else if (d_srcA == M_dstM) begin

            d_valA = m_valM;
        end
        else if (d_srcA == M_dstE) begin

            d_valA = M_valE;
        end
        else if (d_srcA == W_dstM) begin

            d_valA = W_valM;
        end
        else if (d_srcA == W_dstE) begin

            d_valA = W_valE;
        end
        else begin

            d_valA = d_valA_temp;
        end
    end
```

Fwd B Block

```
 // Fwd B Block
    always @(*) begin

        if (d_srcB == e_dstE) begin

            d_valB <= e_valE;
        end
        else if (d_srcB == M_dstM) begin

            d_valB <= m_valM;
        end
        else if (d_srcB == M_dstE) begin

            d_valB <= M_valE;
        end
        else if (d_srcB == W_dstM) begin

            d_valB <= W_valM;
        end
        else if (d_srcB == W_dstE) begin

            d_valB <= W_valE;
        end
        else begin

            d_valB <= d_valB_temp;
        end
    end
endmodule
```

Execute

The execute stage contains the ALU. The instruction that was decoded is
executed as required. The appropriate information is sent from E to M. The condition codes are set, which lets us know whether to update them or not and condition is forwarded to M accordingly. The implementation, for the larger part, is very similar to that in SEQ.



E block (reg)

```
    always @(*) begin

        e_stat <= E_stat;
        e_icode <= E_icode;
        e_valA <= E_valA;
        e_dstM <= E_dstM;
    end
```

ALU

```
wire signed [63:0] result;
wire overflow;
reg signed [63:0] inpA;
reg signed [63:0] inpB;
reg[1:0] select;
reg signed [63:0] ans;

alu alu_mod1(
    .control(select),
    .a(inpA),
    .b(inpB),
    .out(result),
    .overflow(overflow)
 );
```

Condition Codes

```
reg cnd1,cnd2,xoro,ando,oro,noto,temp1;

always @(cnd1 or cnd2) begin
    begin
        ando = cnd1 & cnd2;
        oro = cnd1 | cnd2;
        xoro = cnd1 ^ cnd2 ;
        noto = ~cnd1;
    end
end
```

Flags

```
always @(*)
begin
    ZF = (result == 1'b0); // Output of ALU is zero
    //SF = (result < 1'd0); // Output of the ALU is negative
    if (result[63] == 1'b1) begin
        SF = 1;
    end
    else
    begin
        SF = 0;
    end
    OF = (inpA < 1'b0 == inpB < 1'b0) && (result < 64'b0 != inpA < 1'b0); // signed overflow flag
end
```

Assign ALU values

```
always @(*) begin

    e_Cnd = 1'b0;
    if (E_icode == 4'd2) begin //cmove

            case (E_ifun)
                4'd0://rrmove
                begin
                    e_Cnd = 1'b1;
                end
                4'd1://cmovle
                begin
                    cnd1 = SF; cnd2= OF;
                    temp1 = xoro;
                    cnd1 = ZF;
                    cnd1 = noto;
                    cnd2 = temp1;
                    if(oro)begin
                        e_Cnd = 1'b1;
                    end
            e_valE= result;

                end
                4'd2://cmovl
                begin
                    cnd1 = SF; cnd2= OF;
                    if(xoro)begin
                        e_Cnd = 1'b1;
                    end
            e_valE= result;

                end
                4'd3://cmove
                begin
                    if(ZF)begin
                        e_Cnd = 1'b1;
                    end
            e_valE= result;

                end
                4'd4://cmovne
                begin
                    cnd1 = ZF;
                    if(ZF)begin
                        e_Cnd = 1'b1;
                    end
            e_valE= result;

                end
                4'd5://cmovge
                begin
                    cnd1= SF;
                    cnd2=OF;
                    temp1 = xoro;
                    e_Cnd = temp1;
                    if(noto)begin
                        e_Cnd = 1'b1;
                    end
            e_valE= result;

                end
                4'd6://cmovg
                begin
                    cnd1 = SF; cnd2= OF;
                    temp1 = xoro;
                    cnd1 = temp1;
                    if(noto)begin
                        e_Cnd = 1'b1;
                    end
            e_valE= result;

                end
            endcase
            inpA = E_valA;
            inpB = 64'd0;
            select = 2'd0;
            e_valE= result;
    end
    else if (E_icode == 4'd3) begin //irmove
```

```
        inpA = 64'b0;
        inpB = E_valC;
        select = 2'd0;
        e_valE= result;
    end
    else if (E_icode == 4'd4) begin //rmmove
        inpA = E_valB;
        inpB = E_valC;
        select = 2'd0;
        e_valE= result;
    end
    else if (E_icode == 4'd5) begin //mrmove
        inpA = E_valB;
        inpB = E_valC;
        select = 2'd0;
        e_valE= result;
    end
    else if (E_icode == 4'd6) begin //operation
        inpA = E_valA;
            inpB = E_valB;
            case (E_ifun)
                4'd0://addq
                begin
                    select = 2'd0;
                end
                4'd1://sub
                begin
                    select = 2'd1;
                end
                4'd2://and
                begin
                    select = 2'd2;
                end
                4'd3:
                begin
                    select = 2'd3;
                end
            endcase
            e_valE= result;
    end
    else if (E_icode == 4'd7) begin //jxx
        case (E_ifun)

            4'd0://jmp
            begin
                e_Cnd = 1'b1;
            end
            4'd1://jle
            begin
                cnd1 = SF;
                cnd2 = OF;
                temp1 = xoro;
                cnd1 = temp1;
                cnd2 = ZF;
                if(oro)begin
                    e_Cnd = 1'b1;
                end
                else begin
                    e_Cnd = 1'b0;
                end
            end
            4'd2://jl
            begin
                cnd1= SF;
                cnd2 =OF;
                if(xoro)
                begin
                    e_Cnd = 1'b1;
                end
                else begin
                    e_Cnd = 1'b0;
                end
            end
            4'd3://je
            begin
                if(ZF)
                begin
                    e_Cnd = 1'b1;
                end
                else begin
                    e_Cnd = 1'b0;
                end
            end
            4'd4://jne
            begin
                cnd1 = ZF;
                if(noto)begin
```

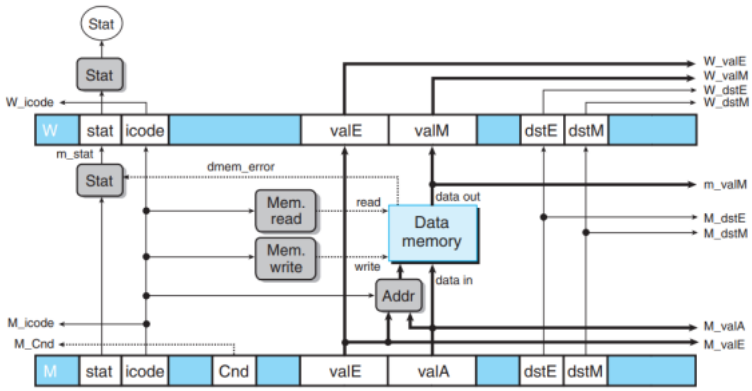
```
                    e_Cnd = 1'b1;
                end
                else begin
                    e_Cnd = 1'b0;
                end
            end
            4'd5://jge
            begin
                cnd1 = SF;
                cnd2 = OF;
                temp1 = xoro;
                cnd1 = temp1;
                if(noto)begin
                    e_Cnd = 1'b1;
                end
                else begin
                    e_Cnd = 1'b0;
                end
            end
            4'd6://jg
            begin
                cnd1 = SF;
                cnd2 = OF;
                temp1 = xoro;
                cnd1 = temp1;
                if(noto)begin
                    e_Cnd = 1'b1;
                end
                else begin
                    e_Cnd = 1'b0;
                end
            end
    endcase
    end
    else if (E_icode == 4'd8) begin //call Dest
        // e_valE= E_valB - 8
            inpA = -64'd8;
            inpB = E_valB;
            select = 2'b00; // to decrement the stack pointer by 8 on call
            e_valE= result;
    end
    else if (E_icode == 4'd9) begin //ret
        // e_valE= E_valB + 8
            inpA = 64'd8;
            inpB = E_valB;
            select = 2'b00; // to increment the stack pointer by 8 on ret
            e_valE= result;
    end
    else if (E_icode == 4'd10) begin //pushq rA
        // e_valE= E_valB - 8
            inpA = -64'd8;
            inpB = E_valB;
            select = 2'b00; // to decrement the stack pointer by 8 on pushq
            e_valE= result;
    end
    else if (E_icode == 4'd11) begin //popq rA
        // e_valE= E_valB + 8
            inpA = 64'd8;
            inpB = E_valB;
            select = 2'b00; // to increment the stack pointer by 8 on popq
            e_valE= result;
    end
end
```

Destination Value

```
always @(*) begin

        if (E_icode == 4'b0010 && e_Cnd == 4'b0000) begin

            e_dstE = 15;
        end
        else begin
            e_dstE = E_dstE;
        end
        e_valA <= E_valA;
    end
endmodule
```

Memory

The memory stage is where the data is read from or written to the memory. The appropriate information is sent from M to W. The memory is declared separate from the instruction memory and does not see use in the other stages as it is accessed only in this stage. A striking feature of this stage is the large number of signals that are sent to the earlier instructions (for potential data problems of later instructions to take care of) from M, the stage itself and W.



Initiating the data memory

```
reg [63:0] memory [0:255];
    reg mem_error;
    integer i ;
    initial begin
        for(i=0;i<255;i = i+1 )begin
            memory[i]=0;
        end
    end
```

Error and Status Block

```
always @(*) begin
        if (m_dstM > 255 ) begin
            mem_error = 1;
        end
        else begin
            mem_error = 0;
        end
    end

     always @(*) begin

        if (mem_error == 1) begin

            m_stat <= 3;
        end
        else begin

            m_stat <= M_stat;
        end
    end
    endmodule
```

Read Block

```
    always@(*)
    begin
        //memory[0] = 64'b2;
        if(M_icode == 4'b0101) // mrmovq
        begin
            m_valM = memory[M_valE];
        end
        else if(M_icode == 4'b1001) //ret
```

```
        begin
            m_valM = memory[M_valA];
        end
        else if(M_icode == 4'b1011) //popq
        begin
            m_valM = memory[M_valA];
        end
    end
```

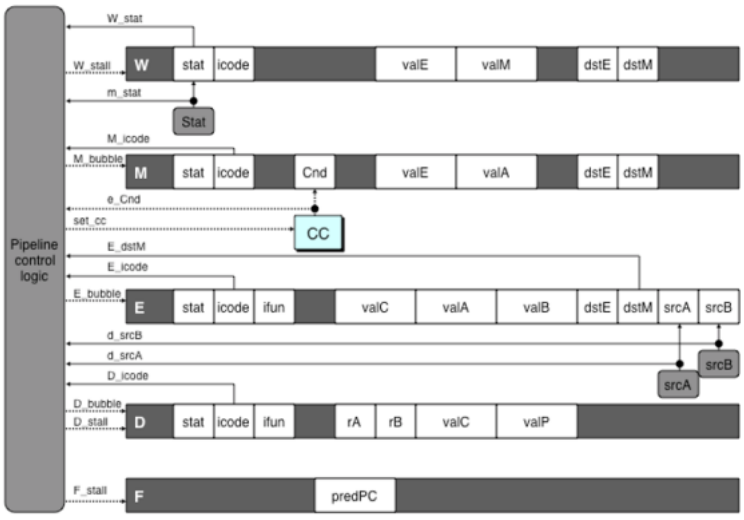
Write Block

```
always@(posedge clk)
begin
    if(M_icode == 4'b0100) // rmmovq
    begin
        memory[M_valE] = M_valA;
    end
    else if(M_icode == 4'b1000) //call
    begin
        memory[M_valE] = M_valA;
    end
    else if(M_icode == 4'b1010) //pushq
    begin
        memory[M_valE] = M_valA;
    end
end
```

M block (reg)

```
always @(*) begin
    m_icode <= M_icode;
    m_valE <= M_valE;
    m_dstE <= M_dstE;
    m_dstM <= M_dstM;
end
```

Control



There are certain control cases which cannot completely be handled by data forwarding and branch prediction. These cases are listed below: -

1. Load/use hazards: For two consecutive instructions where the first reads a value from memory and the second uses that value requires the pipeline to stall for a single cycle.
2. Processing ret: While the ret instruction is being run, the pipeline must be stalled until ret reaches write back.
3. Mispredicted branches: If a jump that was not supposed to happen occurs, the pipeline must cancel all the instructions that have already entered the pipeline and fetch the instruction just after the jump instruction.
4. Exceptions

Predict Hazard

```
  wire Return,pred_miss,haz_L_U;
  assign Return = (D_icode == 9 || E_icode == 9 || M_icode == 9) ? 1 : 0;
  assign haz_L_U = ((E_icode == 5 || E_icode == 11) && (E_dstM == d_srcA || E_dstM == d_srcB)) ? 1 : 0;
  assign pred_miss = (E_icode == 7 && e_Cnd == 0) ? 1 : 0;
```

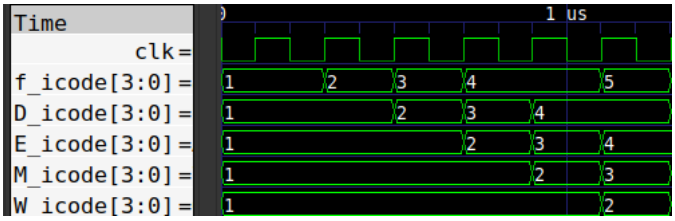
Generating Stalls and Bubbles

```
// Assigning F_stall according to the hazards
  always @(*) begin
      if ((Return== 1 && haz_L_U == 1) || (Return== 1 && pred_miss == 1) || (Return== 1) || (haz_L_U == 1)) begin
          F_stall <= 1;
      end
      else begin
          F_stall <= 0;
      end
  end

  // Assigning D_stall according to the hazards
  always @(*) begin
      if ((haz_L_U == 1 && Return== 1) || (haz_L_U == 1)) begin
          D_stall <= 1;
      end
      else begin
          D_stall <= 0;
      end
  end

  // Assigning D_bubble according to the hazards
  always @(*) begin
      if (D_stall == 0) begin
          if ((Return== 1 && pred_miss == 1) || (Return== 1) || (pred_miss == 1)) begin
              D_bubble <= 1;
          end
          else begin
              D_bubble <= 0;
          end
      end
      else begin
          D_bubble <= 0;
      end
  end

  // Assigning E_bubble according to the hazards
  always @(*) begin
      if ((haz_L_U == 1 && Return== 1) || (Return== 1 && pred_miss == 1) || (haz_L_U == 1) || (pred_miss == 1)) begin
          E_bubble <= 1;
      end
      else begin
          E_bubble <= 0;
      end
  end

  // Assigning M_bubble according to the hazards
  always @(*) begin
      if (m_stat == 2 || m_stat == 3 || m_stat == 4 || W_stat == 2 || W_stat == 3 || W_stat == 4) begin
          M_bubble <= 1;
      end
      else begin
          M_bubble <= 0;
      end
  end

  // Assigning W_stall according to the hazards
```

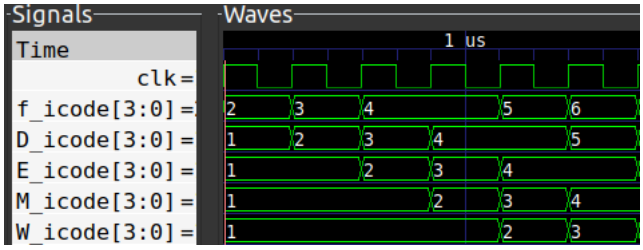
```
    always @(*) begin
        if (W_stat == 2 || W_stat == 3 || W_stat == 4) begin
            W_stall <= 1;
        end
        else begin
            W_stall <= 0;
        end
    end

endmodule
```

Testing and Output:

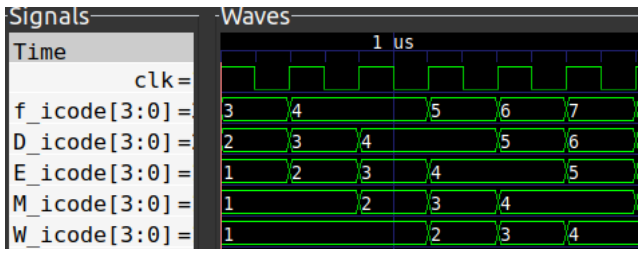
```
inst_memo[0] = 8'h10;//nop
inst_memo[1]  = 8'h10; //nop
```



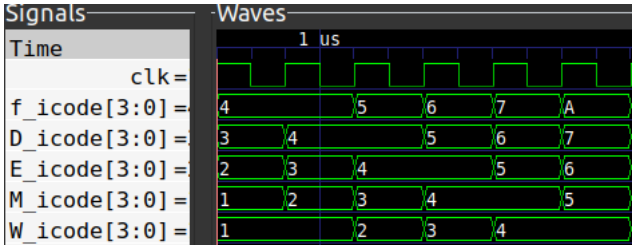

```
inst_memo[2] = 8'h20; //rrmovq
inst_memo[3] = 8'h12;
```



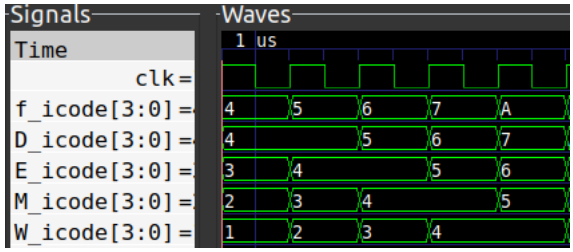

```
inst_memo[4] = 8'h30; //irmovq
inst_memo[5] = 8'hF2;
inst_memo[6] = 8'h00;
inst_memo[7] = 8'h00;
inst_memo[8] = 8'h00;
inst_memo[9] = 8'h00;
inst_memo[10] = 8'h00;
inst_memo[11] = 8'h00;
inst_memo[12] = 8'h00;
inst_memo[13] = 8'b00000010;
```



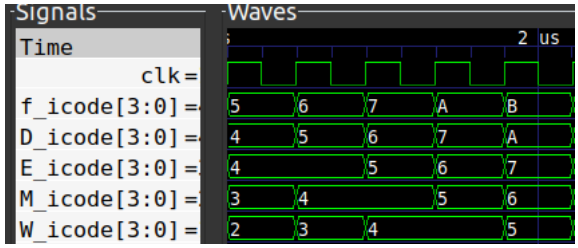

```
inst_memo[14] = 8'h40; //rmmovq
inst_memo[15] = 8'h24;
```



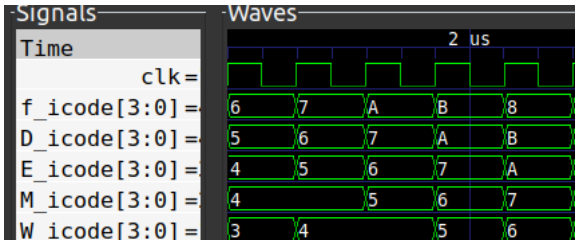

```
inst_memo[24] = 8'h40; //rmmovq
inst_memo[25] = 8'h53;
```



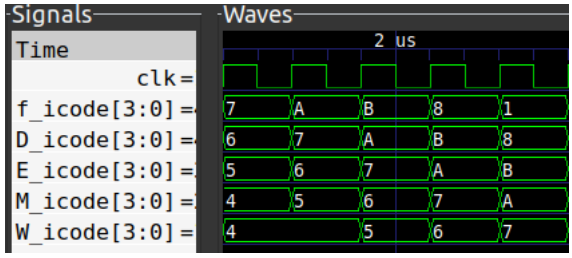

```
inst_memo[34] = 8'h50; //mrmovq
  inst_memo[35] = 8'h53;
```

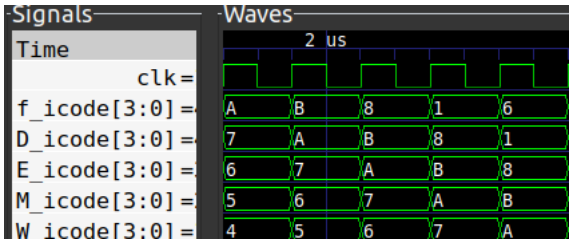



```
inst_memo[44] = 8'h60; //opq
inst_memo[45] = 8'h9A;
```

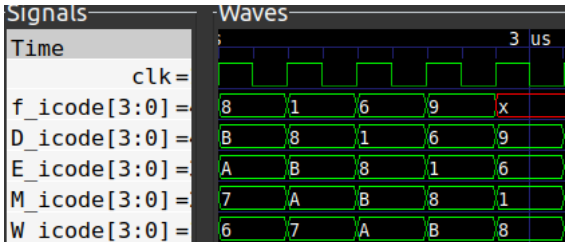


inst_memo[46] = 8'h73; //jmp

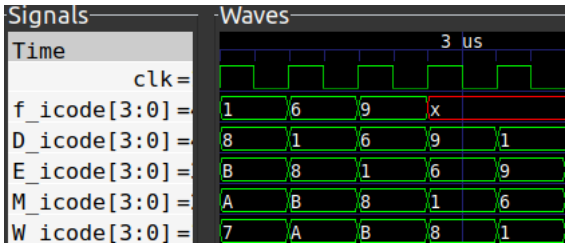




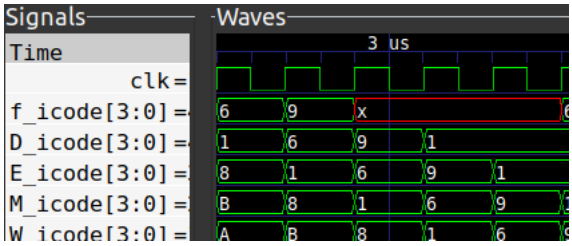

```
inst_memo[54]=8'd56;
{inst_memo[47],inst_memo[48],inst_memo[49],inst_memo[50],inst_memo[51],inst_memo[52],inst_memo[53]}=56'd0;

inst_memo[55] = 8'h00; //halt

inst_memo[56] = 8'hA0; // pushq
inst_memo[57] = 8'h9F;

inst_memo[58] = 8'hB0; //popq
inst_memo[59] = 8'h9F;

inst_memo[60] = 8'h80; //call
```



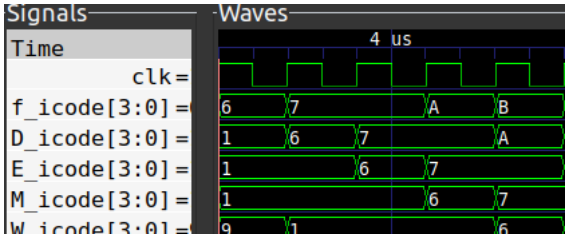

```
{inst_memo[61],inst_memo[62],inst_memo[63],inst_memo[64],inst_memo[65],inst_memo[66],inst_memo[67],inst_memo[68]} = 64'd80;
```



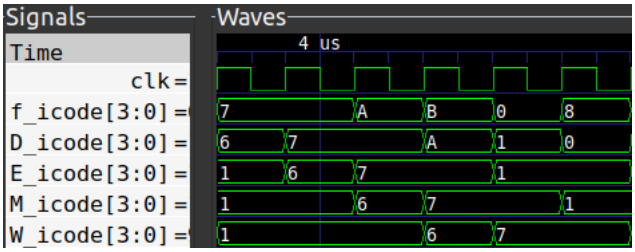

```
inst_memo[69] = 8'h60; //opq
inst_memo[70] = 8'h56;
```



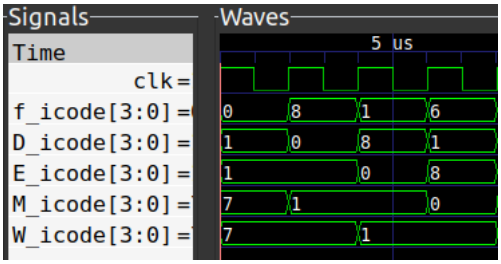
After Misprediction:




```
inst_memo[71] = 8'h72; //jmp
```



before Completion of program



Challenges Faced:

1. Figuring out where to implement always@(*) and where to implement always@(posedge clk).
2. Differentiating between the various PC signals in the fetch stage proved to be confusing.
3. Because the pipelined implementation more than doubles the number of wires, keeping track of all of them in the various modules is difficult.
4. Keeping decode and write back in the pipelined implementation in separate files for clarity's sake proved to be difficult.
5. Figuring out the control logic and implementing it correctly was also rather challenging especially in case of the pipelined processor

Acknowledgement

Working on this project has been a great learning experience. Over the last few weeks, we developed a deeper understanding of processor architecture design, instruction set architecture, memory and many of the other concepts required for this project.
I would like to thank Prof. Deepak Gangadharan and the TAs for guiding us throughout the duration of this project.