Y86-64 ISA Implementation

Intro To Processor Architecture

Srihari **(2021112006)** Hanish **(2021102005)**

Objective -

Objective of this project is to develop a processor architecture design based on the Y86 ISA using Verilog. The processor should be implemented in both sequential and pipelined manner. The design approach is modular. The processor should be able to execute all the instructions in Y86-64 ISA.

Supported Instructions -

Instruction	What does it do?
IHALT	Code for halt instruction
INOP	Code for nop instruction
IRRMOVQ	Code for rrmovq (register to register) instruction
IIRMOVQ	Code for irmovq (immediate to register) instruction
IRMMOVQ	Code for rmmovq (register to memory) instruction
IMRMOVQ	Code for mrmovq (memory to register) instruction
IOPL	Code for integer operation instructions
IJXX	Code for jump instructions
ICALL	Code for call instruction
IRET	Code for ret instruction
IPUSHQ	Code for pushq instruction
IPOPQ	Code for popq instruction

halt

- halt stops instruction execution.
- It requires only a single byte.

nop

- nop stands for no operation
- It also requires a single byte as in halt.

X86-64 data movement instruction movq is split into four cases - rrmovq , irmovq , rmmovq , mrmovq. Memory referencing uses a register plus displacement address computation.

rrmovq

- This is register to register instruction.
- rrmovq instruction is a special case of a conditional move,where the move condition always holds.
- rrmovq has same format as cmovXX, but the destination register updated only if the condition codes satisfy the required constraints.
- **cmovXX** instruction represents seven different branch Instructions with different branch conditions. Branching is based on the setting of the condition codes by the arithmetic instructions.
- The seven instructions include - **rrmovq, cmovle, cmovl, cmove, cmovne, cmovge and cmovg.**

irmovq, rmmovq, mrmovq

- irmovq is immediate to register instruction.
- rmmovq is register to memory instruction.
- mrmovq is memory to register instruction.

OPq

- OPq instruction performs four different arithmetic and logical operations
- ADD,
- SUBTRACT
- XOR
- AND

jXX

- jXX instructions represents seven different branch

Instructions with different branch conditions. Branches are taken according to the type of branch and the settings of the conditional codes. The branch conditions are the same as with x86-64.

call

- call instruction pushes the return address onto the stack and then jumps to the designation.

ret

- ret instruction pops the return address from the stack and jumps to that location.

push

- pushq instruction pushes 8 byte words onto the stack.
- Pushing involves first decrementing the stack pointer by eight and then writing a word to the address given by the stack pointer.

pop

- popq instruction pops 8 byte words off the stack and involves reading the top word on the stack and then incrementing the stack pointer by eight.

Sequential Y86-64 Implementation

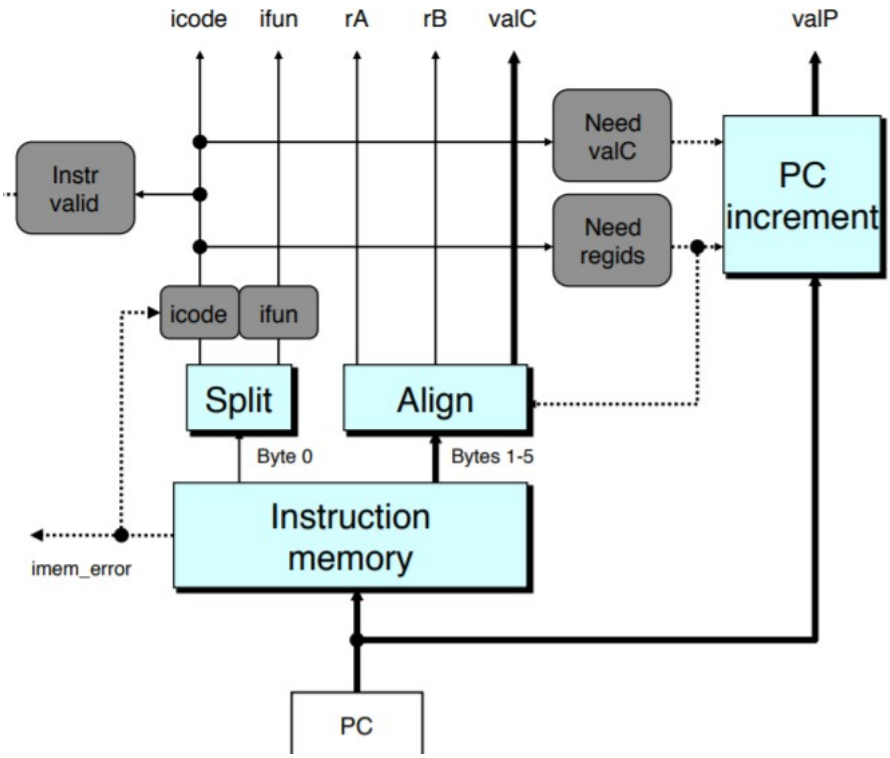
We build the Y86-64 processor in stages. On each clock cycle, SEQ performs all the steps required to process a complete instruction.

The steps and operations performed on each step are described below –

1. **Fetch** - Read instruction from instruction memory.
2. **Decode** - Read program registers
3. **Execute** - Compute value or address
4. **Memory** - Read or write back data.

5. **Write Back -** Write program registers.
6. **PC Update -** Update the program counter

Fetch



-> This stage reads the bytes of an instruction from memory using Program Counter (PC) as the memory address.

Computed Values in this stage are -

icode – Instruction Code

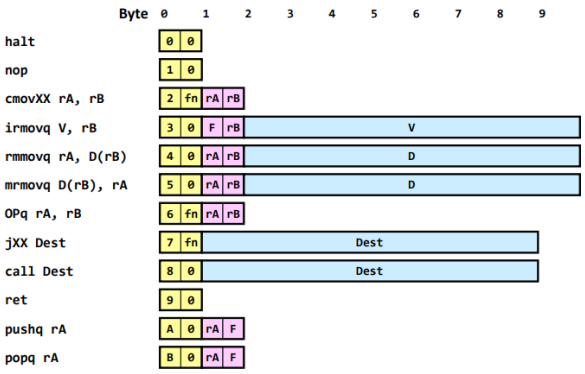
ifun – Function Code **rA** – Inst. Register A **rB** – Inst. Register B

valC – Instruction Constant

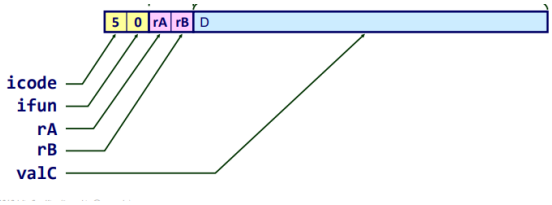
valP – Incremented Program Counter

Implementation Of Fetch Stage -

Below is Y86-64 Instruction Set -



Below is the method to determine icode, ifun, rA, rB and valC of a given instruction.



Based on the above data, we can determine icode, ifun, rA, rB, valC values.

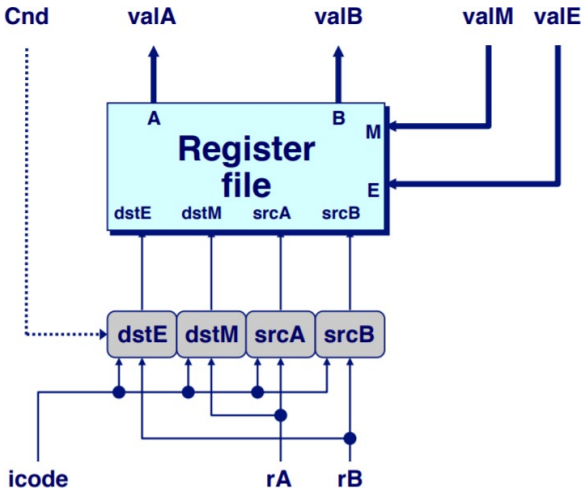
- **In case of halt, icode = 0, ifun = 0, valP = PC+64'd1**
- **In case of nop, icode = 1, ifun = 0, valP = PC+64'd2**
- **In case of cmovXX, icode = 2, ifun for - rrmovq = 0, cmovle = 1, cmovl = 2, cmove = 3 , cmovne = 4, cmovge = 5, cmovg = 6 valP = PC+64'd2**
- **In case of irmovq, icode = 3, ifun = 0, valP = PC+64'd10**
- **In case of rmmovq, icode = 4, ifun = 0, valP = PC+64'd10.**
- **In case of mrmovq, icode = 5, ifun = 0, valP = PC+64'd10.**
- **In case of OPq, icode = 6, ifun for - addq = 0, subq = 1, andq = 2, xorq = 3. valP = PC+64'd2.**
- **In case of jXX, icode = 7, ifun for - jmp = 0, jle = 1, jl = 2, je = 3, jne = 4,jge = 5, jg = 6 , valP = PC+64'd9.**
- **In case of call, icode = 8, ifun = 0, valP = PC+64'd9.**
- **In case of ret, icode = 9, ifun = 0, valP = PC+64'd1.**
- **In case of pushq, icode = 10, ifun = 0, valP = PC+64'd2**
- **In case of popq, icode = 11, ifun = 0, valP = PC+64'd2.**

```
module Fetch(clk, PC , icode , ifun , rA , rB , valC , valP , halt_prog , is_instruction_valid,pcvalid,current_instruction) ;

// 1. In the fetch stage, we need to read the instruction from current_instruction using the PC value
// 2. The first instruction byte is divided into two 4-bits referred to as icode and ifun
//    icode tells us the instruction
//    ifun tells the function of instruction ,else it is 0


// The inputs
input [63:0] PC ;
input clk ;
input [0:79] current_instruction; // max 10 bytes
// The outputs
output reg [3:0] ifun ;
output reg [3:0] icode ;
output reg [3:0] rA ;
output reg [3:0] rB ;
output reg signed[63:0] valC ;
output reg [63:0] valP ;
output reg is_instruction_valid ;
output reg halt_prog ;
output reg pcvalid ;

// Registers
reg [0:7] byte1 ;//ifun icode
reg [0:7] byte2 ;//rA rB

// always@(posedge clk)
always@(*)

  begin

    byte1 = {current_instruction[0:7]} ;
    byte2 = {current_instruction[8:15]} ;

    icode = byte1[0:3];
    ifun  = byte1[4:7];

    is_instruction_valid = 1'b1 ;

    halt_prog = 0 ;

    // icode gives the instruction type
    if(icode == 4'b0000) // Halt instruction should be called
    begin
      halt_prog = 1;
```

```
      valP = PC + 64'd1;  // since only 1byte
      $finish;
    end

    else if(icode == 4'b0001) //nop
    begin
      valP = PC + 64'd1;
    end

    else if(icode == 4'b0010) //cmovxx
    begin
      rA = byte2[0:3];
      rB = byte2[4:7];
      valP = PC + 64'd2;
    end

    else if(icode == 4'b0011) //irmovq
    begin
      rA = byte2[0:3];
      rB = byte2[4:7];
      valC = current_instruction[16:79];
      valP = PC + 64'd10;
    end

    else if(icode == 4'b0100) //rmmovq
    begin
      rA = byte2[0:3];
      rB = byte2[4:7];
      valC = current_instruction[16:79];

      valP = PC + 64'd10;
    end

    else if(icode == 4'b0101) //mrmovq
    begin
      rA = byte2[0:3];
      rB = byte2[4:7];
      valC = current_instruction[16:79];
      valP = PC + 64'd10;
    end

    else if(icode == 4'b0110) //OPq
    begin
      rA = byte2[0:3];
      rB = byte2[4:7];
      valP = PC + 64'd2;
    end

    else if(icode==4'b0111) //jxx
    begin
      valC = current_instruction[8:71];
      valP = PC + 64'd9;
    end

    else if(icode == 4'b1000) //call
    begin
      valC = current_instruction[8:71];
      valP = PC + 64'd9;
    end

    else if(icode == 4'b1001) //ret
    begin
      valP = PC+64'd1;
    end

    else if(icode == 4'b1010) //pushq
    begin
      rA = byte2[0:3];
      rB = byte2[4:7];
      valP = PC + 64'd2;
    end

    else if(icode==4'b1011) //popq
    begin
      rA = byte2[0:3];
      rB = byte2[4:7];
      valP = PC + 64'd2;
    end

    else
    begin
      is_instruction_valid = 1'b0;
    end

    pcvalid = 0;
    if(PC > 1023)
    begin
```

```
      pcvalid = 1 ;
    end

  end

endmodule
```

Decode and Write-Back



Decode reads the registers designated by rA and rB and output values valA and valB but for some instructions it reads register %rsp.

Write-Back write program registers.

During the decode stage, we read both operands. These are supplied to the ALU in the execute stage, along with the function specifier ifun, so that valE becomes the instruction result.

Computed Values in this stage are -

valA - Register Value A

valB - Register Value B

Implementation Of Decode and Write Back Stage -

OPq	Decode	valA ← R[rA]	Read operand A
rmmovq	Decode	valA ← R[rA]	Read operand A
mrmovq	Decode		
irmovq	Decode		
pushq	Decode	valA ← R[rA]	Read operand A
popq	Decode	valA ← R[%rsp]	Read stack pointer
cmovXX	Decode	valA ← R[rA]	Read operand A
jXX	Decode		
call	Decode		
ret	Decode	valA ← R[%rsp]	Read stack pointer

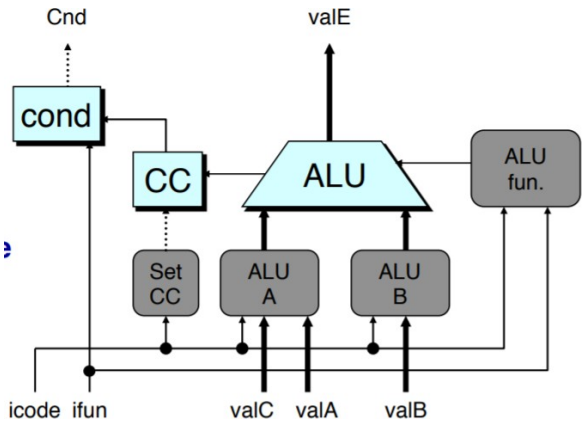
OPq	Write Back	R[rB] ← valE	Write back result
rmmovq	Write Back		
mrmovq	Write Back		
irmovq	Write Back	R[rB] ← valE	Write back result
pushq	Write Back	R[%rsp] ← valE	Update stack pointer
popq	Write Back	R[%rsp] ← valE	Update stack pointer
cmovXX	Write Back	R[rB] ← valE	Write back result
jXX	Write Back		
call	Write Back	R[%rsp] ← valE	Update stack pointer
ret	Write Back	R[%rsp] ← valE	Update stack pointer

```
module decode_and_writeback(valA , valB , valE , valM , clk , rA , rB , icode , cnd , register_memory0 , register_memory1 , register_m

input clk ;
input [3:0] icode,rA,rB ;
input cnd ;
```

```
input signed [63:0] valE;
input [63:0] valM;
output reg signed [63:0] valA,valB ;
reg signed [63:0] register_memory[0:14] ;

// If we were to consider that we have 15 register_memory from %rax to %r14, the stack pointer is %rsp and it is in the 4th place
output reg signed [63:0] register_memory0;
output reg signed [63:0] register_memory1;
output reg signed [63:0] register_memory2;
output reg signed [63:0] register_memory3;
output reg signed [63:0] register_memory4;
output reg signed [63:0] register_memory5;
output reg signed [63:0] register_memory6;
output reg signed [63:0] register_memory7;
output reg signed [63:0] register_memory8;
output reg signed [63:0] register_memory9;
output reg signed [63:0] register_memory10;
output reg signed [63:0] register_memory11;
output reg signed [63:0] register_memory12;
output reg signed [63:0] register_memory13;
output reg signed [63:0] register_memory14;

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

always@(*)
  begin

    if(icode == 4'b0010) //cmovxx
    begin
      valA = register_memory[rA] ;
      valB = 0 ;
    end

    else if(icode == 4'b0100) //rmmovq
    begin
      valA = register_memory[rA] ;
      valB = register_memory[rB] ;
    end

    else if(icode == 4'b0101) //mrmovq
    begin
      valA = 0 ;
      valB = register_memory[rB] ;
    end

    else if(icode == 4'b0110) //OPq
    begin
      valA = register_memory[rA] ;
      valB = register_memory[rB] ;
    end

    else if(icode == 4'b1000) //call
    begin
      // valA = 0;
      valB = register_memory[4] ;
    end

    else if(icode == 4'b1001) //ret
    begin
      valA = register_memory[4] ;
      valB = register_memory[4] ;
    end

    else if(icode == 4'b1010) //pushq
    begin
      valA = register_memory[rA] ;
      valB = register_memory[4] ;
    end
```

```
    else if(icode == 4'b1011) //popq
    begin
      valA = register_memory[4] ;
      valB = register_memory[4] ;
    end
  end
  // Write back stage

always@(posedge clk)
  begin

    if(icode == 4'b0010) //cmovxx
    begin
        if(cnd == 1'b1)  // cnd =1 when condition like < or  = or > or le etc are satisfied
        begin
          register_memory[rB] = valE ;
        end
    end
    else if(icode==4'b0011) //irmovq
      begin
        register_memory[rB] = valE;
      end
    else if(icode == 4'b0101) //mrmovq
      begin
          register_memory[rA] = valM ;
      end

    else if(icode == 4'b0110) //OPq
      begin
          register_memory[rB] = valE ;
      end

    else if(icode == 4'b1000) //call
      begin
          register_memory[4] = valE ;
      end

    else if(icode == 4'b1001) //ret
      begin
        register_memory[4] = valE ;
      end

    else if(icode == 4'b1010) //pushq
      begin
          register_memory[4] = valE ;
      end

    else if(icode == 4'b1011) //popq
      begin
          register_memory[4] = valE ;
          register_memory[rA] = valM ;
      end

    register_memory0 <= register_memory[0];
    register_memory1 <= register_memory[1];
    register_memory2 <= register_memory[2];
    register_memory3 <= register_memory[3];
    register_memory4 <= register_memory[4];
    register_memory5 <= register_memory[5];
    register_memory6 <= register_memory[6];
    register_memory7 <= register_memory[7];
    register_memory8 <= register_memory[8];
    register_memory9 <= register_memory[9];
    register_memory10 <= register_memory[10];
    register_memory11 <= register_memory[11];
    register_memory12 <= register_memory[12];
    register_memory13 <= register_memory[13];
    register_memory14 <= register_memory[14];

  end
endmodule
```

Execute



This stage performs either of the following two actions -

1. ALU performs the operation specified by ifun and computes effective address of memory.
2. Increments (or) Decrements the stack pointer.

Computed Values in this stage are -

valE - ALU Result

Cnd - Constant to determine whether to take a branch or not.

Implementation Of Execute Stage -

OPq	Execute	valE ← valB OP valA	Perform ALU operation
rmmovq	Execute	valE ← valB + valC	Compute effective address
mrmovq	Execute	valE ← valB + valC	Compute effective address
irmovq	Execute	valE ← valB + valC	Pass valC through ALU
pushq	Execute	valE ← valB + (-8)	Decrement stack pointer
popq	Execute	valE ← valB + 8	Increment stack pointer
cmovXX	Execute	valE ← valB + valA	Pass valA through ALU
jXX	Execute		
call	Execute	valE ← valB + (-8)	Decrement stack pointer
ret	Execute	valE ← valB + 8	Increment stack pointer

Condition Codes -

1. **Carry Flag (CF) -** This is used to detect overflow for unsigned operations . This is determined by the most recent operation generated by carry out of the most significant bit.
2. **Zero Flag (ZF) -** This flag comes into effect when the most recent operation yields zero.
3. **Sign Flag (SF) -** This flag comes into effect when the most recent operation yields a negative value.
4. **Overflow Flag (OF) -** This flag comes into effect if the most recent operation caused a two's complement overflow - either positive or negative.

- In case of logical operations, the carry and overflow flags are set to zero.
- In case of shift operations, the carry flag is set the last shifted out, while the overflow flag is set to zero.
- INC and DEC instructions set the overflow flag and zero flag but leave the carry flag unchanged.

Jump instructions and condition codes -

Instruction		Synonym	Jump condition	Description
jmp	*Label*		1	Direct jump
jmp	**Operand*		1	Indirect jump
je	*Label*	jz	ZF	Equal / zero
jne	*Label*	jnz	~ZF	Not equal / not zero
js	*Label*		SF	Negative
jns	*Label*		~SF	Nonnegative
jg	*Label*	jnle	~(SF ^ OF) & ~ZF	Greater (signed >)
jge	*Label*	jnl	~(SF ^ OF)	Greater or equal (signed >=)
jl	*Label*	jnge	SF ^ OF	Less (signed <)
jle	*Label*	jng	(SF ^ OF) | ZF	Less or equal (signed <=)
ja	*Label*	jnbe	~CF & ~ZF	Above (unsigned >)
jae	*Label*	jnb	~CF	Above or equal (unsigned >=)
jb	*Label*	jnae	CF	Below (unsigned <)
jbe	*Label*	jna	CF | ZF	Below or equal (unsigned <=)

cmovXX instructions and condition codes -

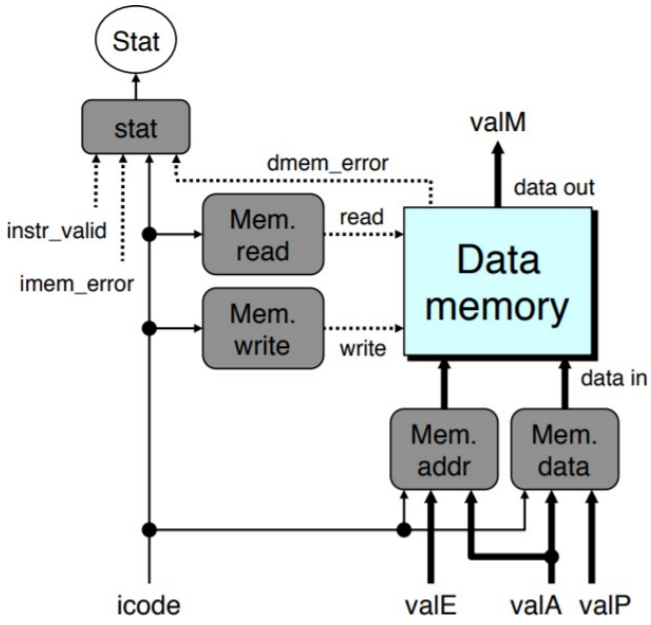
Instruction		Synonym	Move condition	Description
cmove	*S*, *R*	cmovz	ZF	Equal / zero
cmovne	*S*, *R*	cmovnz	~ZF	Not equal / not zero
cmovs	*S*, *R*		SF	Negative
cmovns	*S*, *R*		~SF	Nonnegative
cmovg	*S*, *R*	cmovnle	~(SF ^ OF) & ~ZF	Greater (signed >)
cmovge	*S*, *R*	cmovnl	~(SF ^ OF)	Greater or equal (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 <=)

```
module Execute(icode,ifun,valA,valB,valC,valE,clk,cnd,cc_out,cc_in);

  input clk;
  input [3:0] icode,ifun;
  input [2:0] cc_in;
  input signed [63:0] valA,valB,valC;
  output reg [63:0] valE;
  output reg cnd;
  output reg [2:0] cc_out;
  reg in1,in2,in3,in4,in5,in6,in7;
  wire OUTP1,OUTP2,OUTP3,OUTP4;
  reg [1:0] CONTROL;
  reg signed [63:0] Input1,Input2,Op;
  wire signed [63:0] Output;
  wire OVERFLOW;

  ALU alu1(Input1,Input2,CONTROL,Output,OVERFLOW);

  always @(*)
    begin
      if (icode == 4'b0010) begin //cmovXX-rrmovq,cmovle,cmovl,cmove,cmovne,cmovge,cmovg
        valE <= valA;
      end
      else if (icode == 4'b0111) begin //jmp
      end
      else if (icode == 4'b0011) begin //irmovq
        valE <= valC;
      end

      else if (icode == 4'b0100) begin //rmmovq
        // valE <= valB + valC;
        CONTROL = 2'b00;
        Input1 = valB;
        Input2 = valC;
        valE <= Output ;
      end

      else if (icode == 4'b0101) begin //mrmovq
        // valE <= valB + valC;
        CONTROL = 2'b00;
        Input1 = valB;
        Input2 = valC;
        valE <= Output ;
      end
```

```
    else if (icode == 4'b0110) begin //OPq - Addition, Subtraction, AND, XOR

      cc_out[2] <= OVERFLOW;
      cc_out[1] <=valE[63];
      cc_out[0] <= (valE ==0) ? 1'b1:1'b0;


      if (ifun == 4'b0000) begin //ADD
        CONTROL = 2'b00;
        Input1 = valA;
        Input2 = valB;
      end

      else if (ifun == 4'b0001) begin //SUBTRACT
        CONTROL = 2'b01;
        Input1 = valA;
        Input2 = valB;
      end

      else if (ifun == 4'b0010) begin //AND
        CONTROL = 2'b10;
        Input1 = valA;
        Input2 = valB;
      end

      else if (ifun == 4'b0011) begin //XOR
        CONTROL = 2'b11;
        Input1 = valA;
        Input2 = valB;
      end

      valE <= Output;
    end

    else if (icode == 4'b1000) begin //Call
      // valE <= valB - 64'd1;
      CONTROL = 2'b01;
      Input1 = valB;
      Input2 = 64'd1;
      valE <= Output ;
    end

    else if (icode == 4'b1001) begin //Ret
      // valE <= valB + 64'd1;
      CONTROL = 2'b00;
      Input1 = valB;
      Input2 = 64'd1;
      valE <= Output ;
    end

    else if (icode == 4'b1010) begin //pushq
      // valE <= valB - 64'd1;
      CONTROL = 2'b01;
      Input1 = valB;
      Input2 = 64'd1;
      valE <= Output ;
    end

    else if (icode == 4'b1011) begin //popq
      // valE <= valB + 64'd1;
      CONTROL = 2'b00;
      Input1 = valB;
      Input2 = 64'd1;
      valE <= Output ;
    end
  end
wire zf,sf,of;
assign zf = cc_in[0];
assign sf = cc_in[1];
assign of = cc_in[2];

always @(posedge clk)
begin
    if(icode == 4'b0010 || icode == 4'b0111) //cmovXX && jgXX
    begin
        if(ifun == 4'h0)begin //unconditional
            cnd = 1;
        end
        else if(ifun== 4'h1)begin //le
            cnd = (of^sf)|zf;
        end
        else if(ifun == 4'h2)begin //l
            cnd = (of^sf);
        end
        else if(ifun == 4'h3)begin //e
            cnd = zf;
        end
```

```
          else if(ifun == 4'h4)begin //ne
              cnd = ~zf;
          end
          else if(ifun == 4'h5)begin //ge
              cnd = ~(of^sf);
          end
          else if(ifun == 4'h6)begin //g
              cnd = ~(of^sf) & ~(zf);
          end
      end
  end
endmodule
```

Memory



Memory either reads data from memory or writes data to memory.

Computed Values in this stage are -

valM - Value read from memory

Implementation Of Memory Stage -

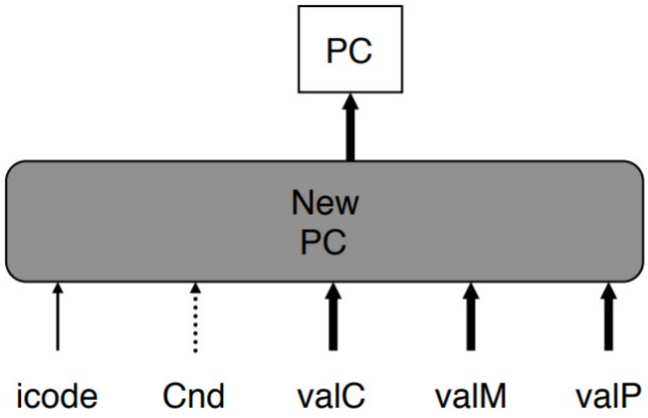
OPq	Memory		
rmmovq	Memory	M_8[valE] ← valA	Write value to memory
mrmovq	Memory	valM ← M_8[valE]	Read value from memory
irmovq	Memory		
pushq	Memory	M_8[valE] ← valA	Write to stack
popq	Memory	valM ← M_8[valA]	Read from stack
cmovXX	Memory		
jXX	Memory		
call	Memory	M_8[valE] ← valP	Update stack pointer
ret	Memory	valM ← M_8[valA]	Update stack pointer

In case of rmovq, call and pushq we write to memory. Whereas, in case of mrmovq, ret and popq we read from memory.

```
module data_memory( clk , icode , valE , valA , valM , valP,memvalid) ;

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

  output reg signed [63:0] valM;
  output reg memvalid;
  reg [63:0] data_memory[255:0];
```

```
  always@(*)
  begin

    //mrmovq
    if(icode == 4'b0101)
    begin
      if(valE > 255)
        begin
            memvalid = 1;
        end
      valM = data_memory[valE] ;
    end

    // ret
    else if(icode == 4'b1001)
    begin
      if(valA > 255)
          begin
              memvalid = 1;
          end
      valM = data_memory[valA];
    end

    // popq
    else if(icode == 4'b1011)
    begin
      if(valE > 255)
        begin
            memvalid = 1;
        end
      valM = data_memory[valA];
    end
    else
      begin
        memvalid =0;
      end
  end

  always @ (posedge clk) begin
    memvalid=0;
    // rmmovq
    if(icode == 4'b0100)
    begin
      if(valE > 255)
        begin
            memvalid = 1;
        end
      data_memory[valE] <= valA;
    end

    // call
    else if(icode == 4'b1000)
    begin
      if(valE > 255)
        begin
            memvalid = 1;
        end
      data_memory[valE] <= valP;
    end

    // pushq
    else if(icode == 4'b1010)
    begin
      if(valE > 255)
        begin
            memvalid = 1;
        end
      data_memory[valE] <= valA;
    end
  end
endmodule
```

PC Update



New value of the PC is taken in one of valC, valM, valP.

Computed Values in this stage are -

PC Update - Updated Program Counter

Implementation Of PC Update Stage -

OPq	PC Update	PC ← valP	Update PC
rmmovq	PC Update	PC ← valP	Update PC
mrmovq	PC Update	PC ← valP	Update PC
irmovq	PC Update	PC ← valP	Update PC
pushq	PC Update	PC ← valP	Update PC
popq	PC Update	PC ← valP	Update PC
cmovXX	PC Update	PC ← valP	Update PC
jXX	PC Update	PC ← Cnd? valC : valP	Update PC
call	PC Update	PC ← valC	Update PC
ret	PC Update	PC ← valM	Update PC

```
module PC_UPDATE(clk,valP,valC,valM,cnd,icode,PC,PC__Update);
  input clk;
  input [63:0] valP; //Incremented PC
  input signed [63:0] valC; //Insruction Constant
  input signed [63:0] valM; //Value from Memory
  input cnd;         //Branch Flag
  input [3:0] icode; //Instruction Code
  input [63:0] PC;
  output reg [63:0] PC__Update;


  always@(posedge clk) begin
    if (icode == 4'b0111) //jxx - jmp, jle, jl, je, jne, jge, and jg
      begin
        //Program Counter is set to Dest if branch is taken (Takes valC)
        //Otherwise PC is incremented by 9 (Takes valP)
        if  (cnd == 1)
          begin
            PC__Update <= valC;
          end
        else
          begin
            PC__Update <= valP;
          end
    end

    else if (icode == 4'b1000) //call
      begin
        //Program Counter is set to Dest (Takes valC)
        PC__Update <= valC;
      end

    else if (icode == 4'b1001) //ret
      begin
      //Program Counter is set to return address (Takes valP)
        PC__Update <= valM;
     end

    else
      begin
        PC__Update <= valP;
      end
```

```
  end
endmodule
```

Processor

```
`include "../ALU/ADD/fulladder1bit.v"
`include "../ALU/ADD/add_64.v"
`include "../ALU/AND/and_64.v"
`include "../ALU/SUB/sub_64.v"
`include "../ALU/XOR/xor_64.v"
`include "../ALU/alu.v"
`include "fetch.v"
`include "execute.v"
`include "decode_and_writeback.v"
`include "memory.v"
`include "PC_Update.v"

module processor;
    reg [0:3]stat_con;
    wire [63:0] PC__Update;
    reg clk;
    reg [63:0]PC;
    wire [3:0] icode,ifun,rA,rB;
    wire signed [63:0] valA, valB, valC,valE,valM;
    wire  [63:0]valP;
    wire halt_prog,is_instruction_valid,pcvalid,cnd,memvalid;
    wire [2:0] cc_out;
    reg [2:0] cc_in;
    reg [7:0] Instruction_memory[0:255];
    reg [0:79] current_instruction;
    // wire ZF,SF,OF;
    wire signed [63:0] register_memory0 , register_memory1 , register_memory2 ,register_memory3 , register_memory4 , register_memory5

    Fetch fetch1(clk, PC , icode , ifun , rA , rB , valC , valP , halt_prog , is_instruction_valid , pcvalid , current_instruction) ;
    decode_and_writeback decode1(valA , valB , valE , valM , clk , rA , rB , icode , cnd , register_memory0 , register_memory1 , regis
    Execute execute1(icode,ifun,valA,valB,valC,valE,clk,cnd,cc_out,cc_in);

    always @(posedge clk) begin
        if(icode == 4'h6)
            begin
                cc_in <=cc_out;
            end
    end

    data_memory memory1( clk , icode , valE , valA , valM , valP,memvalid) ;
    PC_UPDATE update1(clk,valP,valC,valM,cnd,icode,PC,PC__Update);

 always@(PC) begin

    current_instruction = {
      Instruction_memory[PC],
      Instruction_memory[PC+1],
      Instruction_memory[PC+2],
      Instruction_memory[PC+3],
      Instruction_memory[PC+4],
      Instruction_memory[PC+5],
      Instruction_memory[PC+6],
      Instruction_memory[PC+7],
      Instruction_memory[PC+8],
      Instruction_memory[PC+9]
    };
  end

always @(halt_prog,is_instruction_valid,memvalid,pcvalid)begin
    if(halt_prog == 1)begin
        stat_con = 4'b0100;//halt//HLT
        $display("halt");
        $finish;
    end
    if((pcvalid == 1) || (memvalid == 1))begin
        stat_con = 4'b0010;//Memory error//ADR
        $display("mem_error");
        $finish;
    end
    if(is_instruction_valid == 0)begin
        stat_con = 4'b0001;//invalid instruction//INS
        $display("instr_invalid");
        $finish;
    end
end

initial begin
    $dumpfile("processor.vcd");
```

```
    $dumpvars(0, processor);
    stat_con = 4'b1000;//Normal Operation //AOK
    clk = 0;
    PC = 64'd1;

end
always #10  begin
        clk = ~clk;
    end
always @(*) begin
    PC <= PC__Update;
end

always @(posedge clk)
    begin
        $display(" > rsp = %d,PC = %d clk=%d \n > icode=%h ifun=%h rA=%d rB=%d,valC=%d,valP=%d,\n > valA = %d,valB = %d,valE = %d,valM
    end

initial begin
    cc_in = 3'd0;
Instruction_memory[1]  = 8'h10; //nop

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

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

Instruction_memory[14] = 8'h40;//rmmovq
Instruction_memory[15] = 8'h24;
{Instruction_memory[16],Instruction_memory[17],Instruction_memory[18],Instruction_memory[19],Instruction_memory[20],Instruction_memory

Instruction_memory[24] = 8'h40;//rmmovq
Instruction_memory[25] = 8'h53;
{Instruction_memory[26],Instruction_memory[27],Instruction_memory[28],Instruction_memory[29],Instruction_memory[30],Instruction_memory

Instruction_memory[34] = 8'h50;//mrmovq
Instruction_memory[35] = 8'h53;
{Instruction_memory[36],Instruction_memory[37],Instruction_memory[38],Instruction_memory[39],Instruction_memory[40],Instruction_memory

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

Instruction_memory[46] = 8'h73;//je
{Instruction_memory[47],Instruction_memory[48],Instruction_memory[49],Instruction_memory[50],Instruction_memory[51],Instruction_memory

Instruction_memory[55] = 8'h00;

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

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

Instruction_memory[60] = 8'h80;//call
{Instruction_memory[61],Instruction_memory[62],Instruction_memory[63],Instruction_memory[64],Instruction_memory[65],Instruction_memory

Instruction_memory[69] = 8'h60;//OP
Instruction_memory[70] = 8'h56;

Instruction_memory[71] = 8'h70;//jump unconditional
{Instruction_memory[72],Instruction_memory[73],Instruction_memory[74],Instruction_memory[75],Instruction_memory[76],Instruction_memory

Instruction_memory[80] = 8'h30;//irmovq
Instruction_memory[81] = 8'hF2;
Instruction_memory[82] = 8'h00;
Instruction_memory[83] = 8'h00;
Instruction_memory[84] = 8'h00;
Instruction_memory[85] = 8'h00;
Instruction_memory[86] = 8'h00;
Instruction_memory[87] = 8'h00;
Instruction_memory[88] = 8'h00;
Instruction_memory[89] = 8'b00000010;

Instruction_memory[90] = 8'h60;//OPq
Instruction_memory[91] = 8'h9A;

Instruction_memory[92] = 8'h10;//no op
```

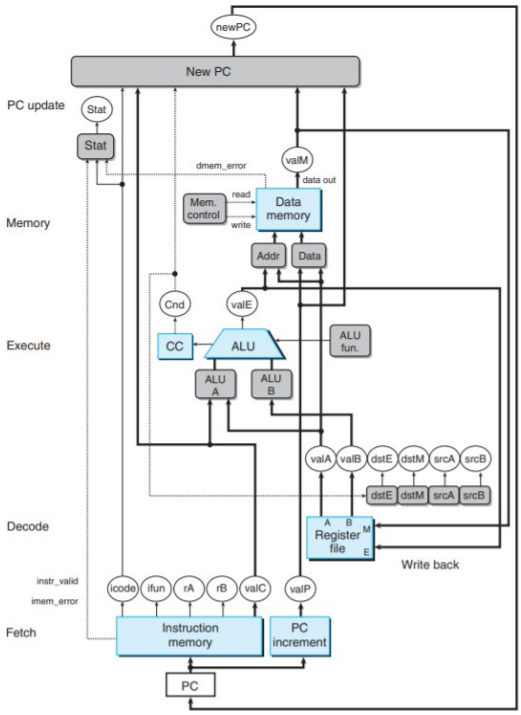
```
Instruction_memory[93] = 8'h90;// return
end
endmodule
```

Output

```
> rsp =                   x,PC =                   1 clk=1
 > icode=1 ifun=0 rA= x rB= x,valC=                   x,valP=                   2,
 > valA =                    x,valB =                      x,valE =                        x,valM =                    x
 > ccodes OF,SF,ZF cc_in = 000 || cc_out=xxx and cnd = x
-----------------------------------------------------------------

 > rsp =                 254,PC =                   2 clk=1
 > icode=2 ifun=0 rA= 1 rB= 2,valC=                   x,valP=                   4,
 > valA =                   10,valB =                      0,valE =                       10,valM =                    x
 > ccodes OF,SF,ZF cc_in = 000 || cc_out=xxx and cnd = 1
-----------------------------------------------------------------

 > rsp =                 254,PC =                   4 clk=1
 > icode=3 ifun=0 rA=15 rB= 2,valC=                   2,valP=                  14,
 > valA =                   10,valB =                      0,valE =                        2,valM =                    x
 > ccodes OF,SF,ZF cc_in = 000 || cc_out=xxx and cnd = 1
-----------------------------------------------------------------

 > rsp =                 254,PC =                  14 clk=1
 > icode=4 ifun=0 rA= 2 rB= 4,valC=                   1,valP=                  24,
 > valA =                    2,valB =                    254,valE =                      255,valM =                    x
 > ccodes OF,SF,ZF cc_in = 000 || cc_out=xxx and cnd = 1
-----------------------------------------------------------------

 > rsp =                 254,PC =                  24 clk=1
 > icode=4 ifun=0 rA= 5 rB= 3,valC=                   0,valP=                  34,
 > valA =                   50,valB =                      3,valE =                        3,valM =                    x
 > ccodes OF,SF,ZF cc_in = 000 || cc_out=xxx and cnd = 1
-----------------------------------------------------------------

 > rsp =                 254,PC =                  34 clk=1
 > icode=5 ifun=0 rA= 5 rB= 3,valC=                   0,valP=                  44,
 > valA =                    0,valB =                      3,valE =                        3,valM =                   50
 > ccodes OF,SF,ZF cc_in = 000 || cc_out=xxx and cnd = 1
-----------------------------------------------------------------

 > rsp =                 254,PC =                  44 clk=1
 > icode=6 ifun=0 rA= 9 rB=10,valC=                   0,valP=                  46,
 > valA =               -12345,valB =                  12345,valE =                        0,valM =                   50
 > ccodes OF,SF,ZF cc_in = 000 || cc_out=001 and cnd = 1
-----------------------------------------------------------------

 > rsp =                 254,PC =                  46 clk=1
 > icode=7 ifun=3 rA= 9 rB=10,valC=                  56,valP=                  55,
 > valA =               -12345,valB =                      0,valE =                   -12345,valM =                   50
 > ccodes OF,SF,ZF cc_in = 001 || cc_out=010 and cnd = 1
-----------------------------------------------------------------

 > rsp =                 254,PC =                  56 clk=1
 > icode=a ifun=0 rA= 9 rB=15,valC=                  56,valP=                  58,
 > valA =               -12345,valB =                    254,valE =                      253,valM =                   50
 > ccodes OF,SF,ZF cc_in = 001 || cc_out=010 and cnd = 1
-----------------------------------------------------------------

 > rsp =                 253,PC =                  58 clk=1
 > icode=b ifun=0 rA= 9 rB=15,valC=                  56,valP=                  60,
 > valA =                  253,valB =                    253,valE =                      254,valM =               -12345
 > ccodes OF,SF,ZF cc_in = 001 || cc_out=010 and cnd = 1
-----------------------------------------------------------------

 > rsp =                 254,PC =                  60 clk=1
 > icode=8 ifun=0 rA= 9 rB=15,valC=                  80,valP=                  69,
 > valA =                  254,valB =                    254,valE =                      253,valM =                    x
 > ccodes OF,SF,ZF cc_in = 001 || cc_out=010 and cnd = 1
-----------------------------------------------------------------

 > rsp =                 253,PC =                  80 clk=1
 > icode=3 ifun=0 rA=15 rB= 2,valC=                   2,valP=                  90,
 > valA =                  254,valB =                    253,valE =                        2,valM =                    x
 > ccodes OF,SF,ZF cc_in = 001 || cc_out=010 and cnd = 1
-----------------------------------------------------------------

 > rsp =                 253,PC =                  90 clk=1
 > icode=6 ifun=0 rA= 9 rB=10,valC=                   2,valP=                  92,
 > valA =               -12345,valB =                      0,valE =                   -12345,valM =                    x
 > ccodes OF,SF,ZF cc_in = 001 || cc_out=010 and cnd = 1
-----------------------------------------------------------------
```

```
 > rsp =                  253,PC =                  92 clk=1
 > icode=1 ifun=0 rA= 9 rB=10,valC=                  2,valP=                 93,
 > valA =               -12345,valB =              -12345,valE =              -24690,valM =                   x
 > ccodes OF,SF,ZF cc_in = 010 || cc_out=010 and cnd = 1
-----------------------------------------------------------------

 > rsp =                  253,PC =                  93 clk=1
 > icode=9 ifun=0 rA= 9 rB=10,valC=                  2,valP=                 94,
 > valA =                 253,valB =                253,valE =                  254,valM =                  69
 > ccodes OF,SF,ZF cc_in = 010 || cc_out=010 and cnd = 1
-----------------------------------------------------------------

 > rsp =                  254,PC =                  69 clk=1
 > icode=6 ifun=0 rA= 5 rB= 6,valC=                  2,valP=                 71,
 > valA =                  50,valB =                -143,valE =                  -93,valM =                   x
 > ccodes OF,SF,ZF cc_in = 010 || cc_out=010 and cnd = 1
-----------------------------------------------------------------

 > rsp =                  254,PC =                  71 clk=1
 > icode=7 ifun=0 rA= 5 rB= 6,valC=                 46,valP=                 80,
 > valA =                  50,valB =                 -93,valE =                  -43,valM =                   x
 > ccodes OF,SF,ZF cc_in = 010 || cc_out=010 and cnd = 1
-----------------------------------------------------------------

 > rsp =                  254,PC =                  46 clk=1
 > icode=7 ifun=3 rA= 5 rB= 6,valC=                 56,valP=                 55,
 > valA =                  50,valB =                 -93,valE =                  -43,valM =                   x
 > ccodes OF,SF,ZF cc_in = 010 || cc_out=010 and cnd = 0
-----------------------------------------------------------------

halt
[Done] exit with code=0 in 0.174 seconds
```

Hardware Structure Of SEQ Implementation -



Y86-64 Pipeline Implementation-

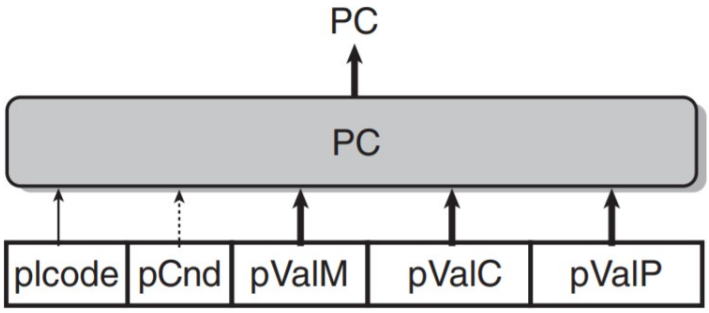
The term Pipelining refers to a technique of decomposing a sequential process into sub-operations, with each sub-operation being executed in a dedicated segment that operates concurrently with all other segments.

A key feature of pipelining is that it increases the throughput of the system. This is the simplest technique for improving performance through hardware parallelism with smaller cycles time.

Steps to design a pipeline –

1. **Rearranging the computation stage-**

- As a transitional step toward a pipelined design, we must slightly rearrange the order of the five stages in SEQ so that the PC update stage comes at the beginning of the clock cycle, rather than at the end.
- This step is also called circuit retiming as we can continuously fetch the next instruction without having to wait for the PC Update stage of the previous instruction.
- Retiming changes the state representation for a system without changing its logical behavior. It is often used to balance the delays between the different stages of a pipelined system.



Inserting Pipeline Registers-

- Second step of creating a pipelined Y86-64 processor is inserting pipeline registers.
- We insert pipeline registers between each stage and rearrange signals.

Pipeline registers are labeled as follows -

F -> Holds the 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.

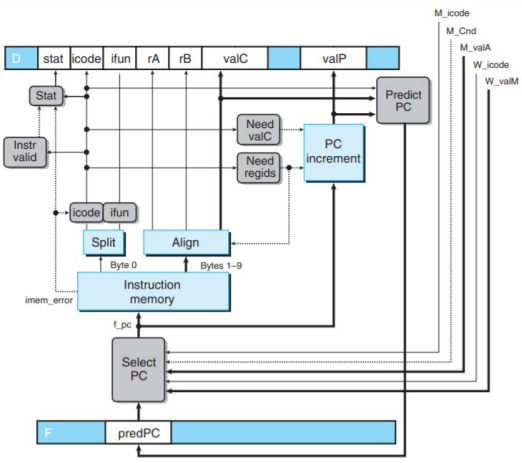
W -> Sits between the memory stage and the feedback paths that supply the computed results to the register file for writing and the return address to the PC selection logic when completing a ret instruction.

Rearranging and Relabeling Signals-

- In this type of implementation, signals pass through every stage one by one.
- We adopt a naming scheme where a signal stored in a pipeline register can be uniquely identified by prefixing its name with that of the pipe register written in uppercase.
- Signals that have just been computed within a stage are labeled by prefixing the signal name with the first character of the stage name, written in lowercase.

PC Selection and Fetch Stage

- Select current PC
- Read instruction
- Compute incremented PC



PC selection Logic -

- The Program Counter, or PC, is a register that holds the address that is presented to the instruction memory. At the start of a cycle, the address is presented to instruction memory. Then during the cycle, the instruction is being read out of instruction memory, and at the same time a calculation is done to determine the next PC.
- As a mispredicted branch enters the memory stage, the value of valP for this instruction (indicating the address of the following instruction) is read from pipeline register M (signal M_valA).
- When a ret instruction enters the write-back stage, the return address is read from pipeline register W (signal W_valM).
- All other cases use the predicted value of the PC, stored in pipeline register F (signal F_predPC) -

```
word f_pc = [
        # Mispredicted branch.  Fetch at incremented PC
        M_icode == IJXX && !M_Cnd : M_valA;
        # Completion of RET instruction
        W_icode == IRET : W_valM;
        # Default: Use predicted value of PC
        1 : F_predPC;
];
```

- The PC prediction logic chooses valC for the fetched instruction when it is either a call or a jump, and valP otherwise:

```
word f_predPC = [
        f_icode in { IJXX, ICALL } : f_valC;
        1 : f_valP;
];
```

- Unlike in SEQ, we must split the computation of the instruction status into two parts. In the fetch stage, we can test for a memory error due to an out-of-range instruction address, and we can detect an illegal instruction or a halt instruction. Detecting an invalid data address must be deferred to the memory stage.

```
module Fetch(clk,F_predPC,f_predPC,M_valA,W_valM,M_Cnd,M_icode,W_icode,F_stall,D_stall,D_bubble,D_stat,D_icode,D_ifun,D_rA,D_rB,D_valC


// The inputs
input [63:0] F_predPC;
input clk ;
input [3:0] M_icode;
input [3:0] W_icode;
input signed [63:0] M_valA;
input signed [63:0] W_valM;
input M_Cnd;
input F_stall;
input D_stall;
input D_bubble;
input [0:79] current_instruction;

output reg [63:0] f_predPC;
```

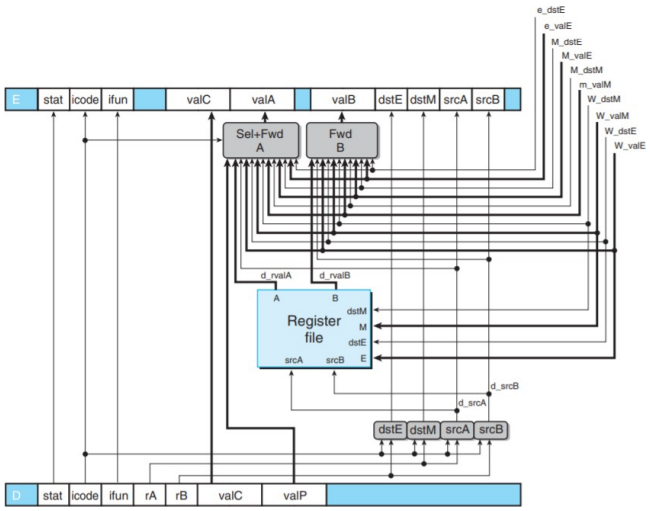
```
output reg [3:0] D_ifun ;
output reg [3:0] D_icode ;
output reg [3:0] D_rA ;
output reg [3:0] D_rB ;
output reg signed[63:0] D_valC ;
output reg [63:0] D_valP ;
output reg [0:3] D_stat;

// Registers
reg [63:0] PC;
reg [0:7] byte1 ;//ifun icode
reg [0:7] byte2 ;//rA rB
reg [3:0] icode,ifun;
reg signed [63:0] valC;
reg [63:0] valP;
reg is_instruction_valid = 1'b1;
reg pcvalid = 1'b0;
reg halt_prog=1'b0;
reg [0:3] stat;
reg [3:0] rA,rB;

// initial begin
//   PC = F_predPC;
// end

always @(*)
  begin
    if(M_icode == 4'b0111 & !M_Cnd)
      PC = M_valA;
    else if(W_icode == 4'b1001 )
      PC = W_valM;
    else
      PC = F_predPC;
  end

always@(*)

  begin
    byte1 = {current_instruction[0:7]} ;
    byte2 = {current_instruction[8:15]} ;

    icode = byte1[0:3];
    ifun  = byte1[4:7];

    // halt_prog = 0 ;

    // icode gives the instruction type
    if(icode == 4'b0000) // Halt instruction should be called
    begin
      halt_prog = 1;
      valP = PC;  // since only 1byte
      f_predPC =  valP;
      // $finish;
    end

    else if(icode == 4'b0001) //nop
    begin
      valP = PC + 64'd1;
      f_predPC =  valP;
        // $display("77");

    end

    else if(icode == 4'b0010) //cmovxx
    begin
      rA = byte2[0:3];
      rB = byte2[4:7];
      valP = PC + 64'd2;
      f_predPC =  valP;
    end

    else if(icode == 4'b0011) //irmovq
    begin
      rA = byte2[0:3];
      rB = byte2[4:7];
      valC = current_instruction[16:79];
      valP = PC + 64'd10;
      f_predPC =  valP;
    end

    else if(icode == 4'b0100) //rmmovq
    begin
      rA = byte2[0:3];
      rB = byte2[4:7];
      valC = current_instruction[16:79];
      valP = PC + 64'd10;
      f_predPC =  valP;
```

```
    end

    else if(icode == 4'b0101) //mrmovq
    begin
      rA = byte2[0:3];
      rB = byte2[4:7];
      valC = current_instruction[16:79];
      valP = PC + 64'd10;
      f_predPC =  valP;
    end

    else if(icode == 4'b0110) //OPq
    begin
      rA = byte2[0:3];
      rB = byte2[4:7];
      valP = PC + 64'd2;
      f_predPC =  valP;
    end

    else if(icode==4'b0111) //jxx
    begin
      valC = current_instruction[8:71];
      valP = PC + 64'd9;
      f_predPC =  valC;
    end

    else if(icode == 4'b1000) //call
    begin
      valC = current_instruction[8:71];
      valP = PC + 64'd9;
      // $display("valC =",valC);
      f_predPC =  valC;
    end

    else if(icode == 4'b1001) //ret
    begin
      valP = PC+64'd1;
    end

    else if(icode == 4'b1010) //pushq
    begin
      rA = byte2[0:3];
      rB = byte2[4:7];
      valP = PC + 64'd2;
      f_predPC =  valP;
    end

    else if(icode==4'b1011) //popq
    begin
      rA = byte2[0:3];
      rB = byte2[4:7];
      valP = PC + 64'd2;
      f_predPC =  valP;
    end

    else
    begin
      is_instruction_valid = 1'b0;
    end

    if(PC > 1023)
    begin
      pcvalid = 1'b1 ;
    end
  end

always @(*)begin
    stat = 4'b1000;
    if(halt_prog == 1)begin
        stat = 4'b0100;//halt//HLT
        $display("halt");
        $finish;
    end
    if((pcvalid == 1))begin
        stat = 4'b0010;//Memory error//ADR
        $display("mem_error");
        $finish;
    end
    if(is_instruction_valid == 0)begin
        stat = 4'b0001;//invalid instruction//INS
        $display("instr_invalid");
        $finish;
    end
  end

  always @(posedge clk) begin
```

```
    if (F_stall)
      begin
        // PC = F_predPC;
      end
    else if (D_stall)
      begin
      end
    else if (D_bubble)
      begin
        D_icode <= 4'b0001;
        D_ifun <= 4'b0000;
        D_rA <= 4'b1111;
        D_rB <= 4'b1111;
        D_valC <= 64'b0;
        D_valP <= 64'b0;
        D_stat <= 4'b1000;
      end
    else
      begin
        D_icode <= icode;
        D_ifun <= ifun;
        D_rA <= rA;
        D_rB <= rB;
        D_valC <= valC;
        D_valP <= valP;
        D_stat <= stat;
      end
  end

endmodule
```

Decode and Write-Back Stages

- Read program registers
- Update register file



- Register has four ports in which two are read ports and two are write ports.
- It supports two simultaneous reads and two simultaneous writes.
- The two read ports have address inputs srcA and srcB and the two write ports have address inputs dstE and dstM.

srcA - Indicate which register should be read to generate valA.

srcB - Indicate which register should be read to generate valB

dstE - Indicate the destination register for write port E where valE is stored.

dstM - Indicate the destination register for write port M where valM is stored.

These four blocks dstE, dstM, srcA, srcB, generate the four different register IDs for the register file, based on the instruction code icode, the register specifiers rA and rB, and possibly the condition signal Cnd computed in the execute stage.

- Data forwarding takes place in this stage.

- Block “Sel+Fwd A” merges valP into valA for later stages in order to reduce the amount of state in the pipeline register as only call and jump instructions need valP in further stages instead of valA.
- This block also implements the forwarding logic for source operand valA.
- Block “Fwd B” implements the forwarding logic for source operand valB.
- We also introduce a status register “stat” to indicate whether the program is executing normally or an exception occurred.

This is needed as the code should indicate either AOK or one of the three exception conditions. Exceptional conditions include when an Invalid instruction is fetched, or a halt instruction is executed.

- Consider bubble in the Write-Back stage as AOK.

```
module decode_and_writeback(clk,D_bubble,E_bubble,D_stat,D_icode,D_ifun,D_rA,D_rB,D_valC,D_valP,
                            e_destE,e_valE,M_destE,M_valE,M_destM,m_valM,W_destM,W_valM,W_destE,W_valE,
                            E_stat,E_icode,E_ifun,E_valC,E_valA,E_valB,E_destE,E_destM,E_srcA,E_srcB,
                            W_icode,d_srcA,d_srcB,
                            register_memory0 , register_memory1 , register_memory2 ,register_memory3 , register_memory4 , register_mem

input clk ;
input [3:0] D_icode,D_ifun,D_rA,D_rB,e_destE,M_destE,M_destM,W_destE,W_destM,W_icode;
input [0:3] D_stat;
input signed [63:0] D_valC,e_valE,M_valE,m_valM,W_valE,W_valM;
input [63:0] D_valP;
input E_bubble,D_bubble;

output reg signed [63:0] E_valC,E_valA,E_valB;
output reg [3:0] E_icode,E_ifun,E_destE,E_destM,E_srcA,E_srcB,d_srcA,d_srcB;
output reg [0:3] E_stat;
reg [3:0] d_destE,d_destM;
reg signed [63:0] d_rvalA,d_rvalB,d_valA,d_valB;
reg [63:0] register_memory[0:14];

// If we were to consider that we have 15 register_memory from %rax to %r14, the stack pointer is %rsp and it is in the 4th place
output reg signed [63:0] register_memory0;
output reg signed [63:0] register_memory1;
output reg signed [63:0] register_memory2;
output reg signed [63:0] register_memory3;
output reg signed [63:0] register_memory4;
output reg signed [63:0] register_memory5;
output reg signed [63:0] register_memory6;
output reg signed [63:0] register_memory7;
output reg signed [63:0] register_memory8;
output reg signed [63:0] register_memory9;
output reg signed [63:0] register_memory10;
output reg signed [63:0] register_memory11;
output reg signed [63:0] register_memory12;
output reg signed [63:0] register_memory13;
output reg signed [63:0] register_memory14;

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

always@(*)
  begin

    if(D_icode == 4'b0010) //cmovxx
    begin
      d_srcA = D_rA;
      d_destE = D_rB;
      d_rvalA = register_memory[D_rA] ;
      d_rvalB = 0 ;
    end

    else if(D_icode == 4'b0011) //irmovq
```

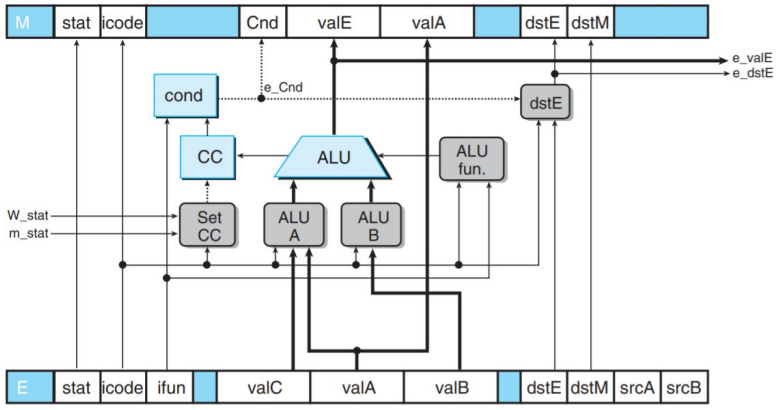
```
begin
// // // d_rvalB=64'b0 //
  d_destE=D_rB;
end

else if(D_icode == 4'b0100) //rmmovq
begin
  d_srcA = D_rA;
  d_srcB = D_rB;
  d_rvalA = register_memory[D_rA] ;
  d_rvalB = register_memory[D_rB] ;
end

else if(D_icode == 4'b0101) //mrmovq
begin
  d_srcB = D_rB;
  d_destM = D_rA;
  //d_rvalA = 0 ; // // //
  d_rvalB = register_memory[D_rB] ;
end

else if(D_icode == 4'b0110) //OPq
begin
  d_srcA = D_rA;
  d_srcB = D_rB;
  d_destE = D_rB;
  d_rvalA = register_memory[D_rA] ;
  d_rvalB = register_memory[D_rB] ;
end

else if(D_icode == 4'b1000) //call
begin
  d_srcB = 4;
  d_destE = 4;
  d_rvalB = register_memory[4] ;
end

else if(D_icode == 4'b1001) //ret
begin
  d_srcA = 4;
  d_srcB = 4;
  d_destE = 4;
  d_rvalA = register_memory[4] ;
  d_rvalB = register_memory[4] ;
end

else if(D_icode == 4'b1010) //pushq
begin
  d_srcA = D_rA;
  d_srcB = 4;
  d_destE = 4;
  d_rvalA = register_memory[D_rA] ;
  d_rvalB = register_memory[4] ;
end

else if(D_icode == 4'b1011) //popq
begin
  d_srcA = 4;
  d_srcB = 4;
  d_destE = 4;
  d_destM = D_rA;
  d_rvalA = register_memory[4] ;
  d_rvalB = register_memory[4] ;
end

else
  begin
  d_srcA = 4'hF;
  d_srcB = 4'hF;
  d_destE = 4'hF;
  d_destM = 4'hF;
  end

  // Forwarding A
if(D_icode==4'h7 | D_icode == 4'h8) //jxx or call
  d_valA = D_valP; //here we are using valA to hold value of valP
else if(d_srcA==e_destE & e_destE!=4'hF)
  d_valA = e_valE;//data forwarding from execute to src register
else if(d_srcA==M_destM & M_destM!=4'hF)
  d_valA = m_valM; //data forwarding from memory to src register
else if(d_srcA==W_destM & W_destM!=4'hF)
  d_valA = W_valM; //data forwarding from write back stage
else if(d_srcA==M_destE & M_destE!=4'hF)
  d_valA = M_valE; //data forwarding from memory stage
else if(d_srcA==W_destE & W_destE!=4'hF)
  d_valA = W_valE;//data forwarding from writeback stage
else
```

```
      d_valA = d_rvalA;//no Data forwading

    // Forwarding B
    if(d_srcB==e_destE & e_destE!=4'hF)
      d_valB = e_valE;// data forwarding from execute stage
    else if(d_srcB==M_destM & M_destM!=4'hF)
      d_valB = m_valM;// data forwarding from memory stage
    else if(d_srcB==W_destM & W_destM!=4'hF)
      d_valB = W_valM; // data forwarding memory value from write back stage
    else if(d_srcB==M_destE & M_destE!=4'hF)
      d_valB = M_valE; // data forwarding execute value from memory stage
    else if(d_srcB==W_destE & W_destE!=4'hF)
      d_valB = W_valE; // data forwarding execute value from write back stage
    else
      d_valB = d_rvalB;// no Data forwarding
  end

always@(posedge clk)
  begin
    if(E_bubble)
    begin
    // $display("179");
      E_stat <= 4'b1000;
      E_icode <= 4'b0001;
      E_ifun <= 4'b0000;
      E_valC <= 4'b0000;
      E_valA <= 4'b0000;
      E_valB <= 4'b0000;
      E_destE <= 4'hF;
      E_destM <= 4'hF;
      E_srcA <= 4'hF;
      E_srcB <= 4'hF;
    end
    else
    begin
      // Execute register update
      E_stat <= D_stat;
      E_icode <= D_icode;
      E_ifun <= D_ifun;
      E_valC <= D_valC;
      E_valA <= d_valA;
      E_valB <= d_valB;
      E_srcA <= d_srcA;
      E_srcB <= d_srcB;
      E_destE <= d_destE;
      E_destM <= d_destM;
    end

  end

// Write back stage
always@(posedge clk)
  begin

    if(W_icode == 4'b0010) //cmovxx
      begin
        register_memory[W_destE] = W_valE ;
      end
    else if(W_icode==4'b0011) //irmovq
      begin
        register_memory[W_destE] = W_valE;
      end
    else if(W_icode == 4'b0101) //mrmovq
      begin
          register_memory[W_destM] = W_valM ;
      end

    else if(W_icode == 4'b0110) //OPq
      begin
          register_memory[W_destE] = W_valE ;
      end

    else if(W_icode == 4'b1000) //call
      begin
          register_memory[W_destE] = W_valE ;
      end

    else if(W_icode == 4'b1001) //ret
      begin
        register_memory[W_destE] = W_valE ;
      end

    else if(W_icode == 4'b1010) //pushq
      begin
          register_memory[W_destE] = W_valE ;
      end
```

```
    else if(W_icode == 4'b1011) //popq
      begin
          register_memory[W_destE] = W_valE;
          register_memory[W_destM] = W_valM;
      end

    register_memory0 <= register_memory[0];
    register_memory1 <= register_memory[1];
    register_memory2 <= register_memory[2];
    register_memory3 <= register_memory[3];
    register_memory4 <= register_memory[4];
    register_memory5 <= register_memory[5];
    register_memory6 <= register_memory[6];
    register_memory7 <= register_memory[7];
    register_memory8 <= register_memory[8];
    register_memory9 <= register_memory[9];
    register_memory10 <= register_memory[10];
    register_memory11 <= register_memory[11];
    register_memory12 <= register_memory[12];
    register_memory13 <= register_memory[13];
    register_memory14 <= register_memory[14];

  end
endmodule
```

Execute

- Operate ALU



- Pipeline implementation of execute stage is similar to the sequential implementation.
- In pipeline implementation, the logic “Set CC” has signals m_stat and W_stat as inputs.
- The signals e_valE and e_dstE are directed towards the decode stage as forwarding sources.

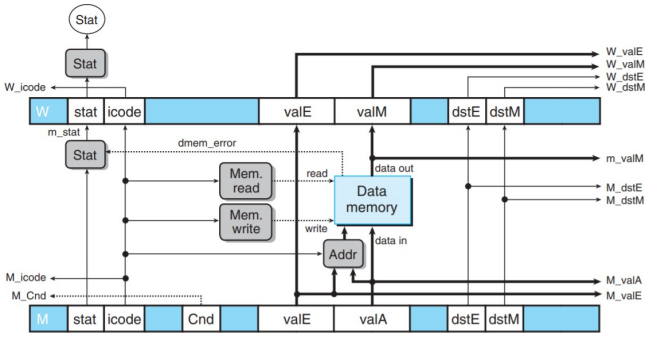
```
module Execute(clk,E_stat,E_icode,E_ifun,E_valA,E_valB,E_valC,E_destE,E_destM,M_bubble,setcc,e_valE,e_destE,e_Cnd,M_stat,M_icode,M_Cnd

  input clk;
  input [0:3] E_stat;
  input [3:0] E_icode,E_ifun,E_destE,E_destM;
  input signed [63:0] E_valA,E_valB,E_valC;
  input M_bubble,setcc;

  output reg signed [63:0] e_valE,M_valE,M_valA;
  output reg [3:0] e_destE,M_destE,M_destM,M_icode;
  output reg e_Cnd;
  output reg [0:3] M_stat;
  output reg M_Cnd;

  output reg [2:0] cc_in = 3'b000;
  // reg [2:0] cc_out;
  reg [1:0] CONTROL;
  reg signed [63:0] Input1,Input2;
  wire signed [63:0] Output;
  wire OVERFLOW;

  ALU alu1(Input1,Input2,CONTROL,Output,OVERFLOW);

  always @(*)
    begin
      if (E_icode == 4'b0010) begin //cmovXX-rrmovq,cmovle,cmovl,cmove,cmovne,cmovge,cmovg
```

```
        e_valE <= E_valA;
      end
      else if (E_icode == 4'b0111) begin //jmp
      end
      else if (E_icode == 4'b0011) begin //irmovq
        e_valE <= E_valC;
      end

      else if (E_icode == 4'b0100) begin //rmmovq
        // valE <= valB + valC;
        CONTROL = 2'b00;
        Input1 = E_valB;
        Input2 = E_valC;
        e_valE <= Output ;
      end

      else if (E_icode == 4'b0101) begin //mrmovq
        // valE <= valB + valC;
        CONTROL = 2'b00;
        Input1 = E_valB;
        Input2 = E_valC;
        e_valE <= Output ;
      end

      else if (E_icode == 4'b0110) begin //OPq - Addition, Subtraction, AND, XOR

        if (E_ifun == 4'b0000) begin //ADD
          CONTROL = 2'b00;
          Input1 = E_valA;
          Input2 = E_valB;
        end

        else if (E_ifun == 4'b0001) begin //SUBTRACT
          CONTROL = 2'b01;
          Input1 = E_valA;
          Input2 = E_valB;
        end

        else if (E_ifun == 4'b0010) begin //AND
          CONTROL = 2'b10;
          Input1 = E_valA;
          Input2 = E_valB;
        end

        else if (E_ifun == 4'b0011) begin //XOR
          CONTROL = 2'b11;
          Input1 = E_valA;
          Input2 = E_valB;
        end

        e_valE <= Output;

        if(setcc)
        begin
        // // //
        cc_in[2] <= OVERFLOW;
        cc_in[1] <= e_valE[63];
        cc_in[0] <= (e_valE ==0) ? 1'b1:1'b0;
        end
      end

      else if (E_icode == 4'b1000) begin //Call
        // valE <= valB - 64'd1;
        CONTROL = 2'b01;
        Input1 = E_valB;
        Input2 = 64'd1;
        e_valE <= Output ;
      end

      else if (E_icode == 4'b1001) begin //Ret
        // valE <= valB + 64'd1;
        CONTROL = 2'b00;
        Input1 = E_valB;
        Input2 = 64'd1;
        e_valE <= Output ;
      end

      else if (E_icode == 4'b1010) begin //pushq
        // valE <= valB - 64'd1;
        CONTROL = 2'b01;
        Input1 = E_valB;
        Input2 = 64'd1;
        e_valE <= Output ;
      end

      else if (E_icode == 4'b1011) begin //popq
        // valE <= valB + 64'd1;
```

```
        CONTROL = 2'b00;
        Input1 = E_valB;
        Input2 = 64'd1;
        e_valE <= Output ;
      end
    end

  wire zf,sf,of;
  assign zf = cc_in[0];
  assign sf = cc_in[1];
  assign of = cc_in[2];

  always @(*)
  begin
      if(E_icode == 4'b0010 || E_icode == 4'b0111) //cmovXX && jgXX
      begin
          if(E_ifun == 4'h0)begin //unconditional
              e_Cnd = 1;
          end
          else if(E_ifun== 4'h1)begin //le
              e_Cnd = (of^sf)|zf;
          end
          else if(E_ifun == 4'h2)begin //l
              e_Cnd = (of^sf);
          end
          else if(E_ifun == 4'h3)begin //e
              e_Cnd = zf;
          end
          else if(E_ifun == 4'h4)begin //ne
              e_Cnd = ~zf;
          end
          else if(E_ifun == 4'h5)begin //ge
              e_Cnd = ~(of^sf);
          end
          else if(E_ifun == 4'h6)begin //g
              e_Cnd = ~(of^sf) & ~(zf);
          end
        e_destE = (e_Cnd == 1) ? E_destE : 4'b1111;    //empty register
      end
      else
      begin
        e_destE =E_destE;
        e_Cnd=0;
      end
  end

  always@(posedge clk)
  begin
    if(M_bubble)
    begin
      M_stat <= 4'b1000;
      M_icode <= 4'b0001;
      M_Cnd <= 1;
      M_valE <= 0;
      M_valA <= 0;
      M_destE <= 4'hF;
      M_destM <= 4'hF;
    end
    else
    begin
      M_stat <= E_stat;
      M_icode <= E_icode;
      M_Cnd <= e_Cnd;
      M_valE <= e_valE;
      M_valA <= E_valA;
      M_destE <= e_destE;
      M_destM <= E_destM;
    end
  end
endmodule
```

Memory

- Read or write data memory

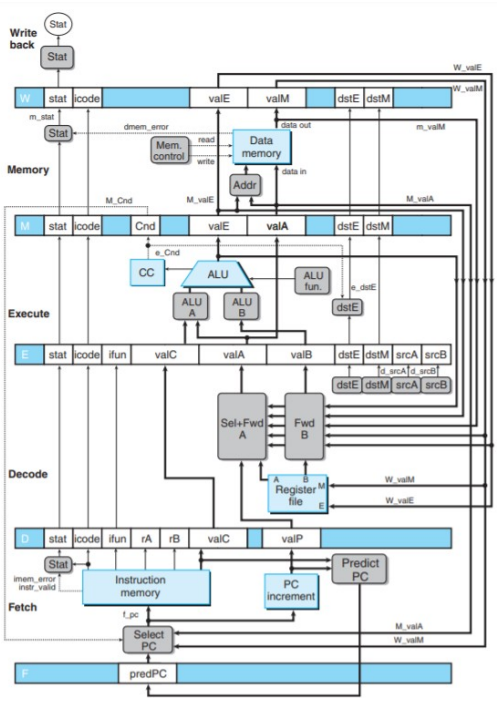


- Memory block either reads or writes the program data.
- Memory stage in pipeline lacks “Mem.data” block present in SEQ as the task is performed by “Sel+Fwd A” block in decode stage.

```
module data_memory(clk,M_stat,M_icode,M_Cnd,M_valE,M_valA,M_destE,M_destM,m_stat,m_valM,W_stall,W_stat,W_icode,W_valE,W_valM,W_destE,W

  input clk;
  input [0:3] M_stat;
  input [3:0] M_icode;
  input M_Cnd;
  input [63:0] M_valE;
  input [63:0] M_valA;
  input [3:0] M_destE;
  input [3:0] M_destM;
  input W_stall;

  output reg [0:3] m_stat;
  output reg signed [63:0] m_valM;
  output reg [0:3] W_stat;
  output reg [3:0] W_icode;
  output reg signed [63:0] W_valE;
  output reg signed [63:0] W_valA;
  output reg signed [63:0] W_valM;

  output reg [3:0] W_destE;
  output reg [3:0] W_destM;

  reg [63:0] data_memory [255:0];
  reg memvalid = 0;

  always @(*) begin
      if(memvalid)
      m_stat = 4'b0010;
      else
      m_stat = M_stat;
  end

  always@(*)
  begin

    //mrmovq
    if(M_icode == 4'b0101)
    begin
      if(M_valE > 255)
        begin
            memvalid = 1;
        end
      m_valM = data_memory[M_valE] ;
    end

    // ret
    else if(M_icode == 4'b1001)
    begin
      if(M_valA > 255)
          begin
              memvalid = 1;
          end
      m_valM = data_memory[M_valA];
    end

    // popq
    else if(M_icode == 4'b1011)
    begin
      if(M_valE > 255)
        begin
            memvalid = 1;
        end
      m_valM = data_memory[M_valA];
```

```
    end
    else
      begin
        memvalid =0;
      end
  end

  always @ (posedge clk) begin
    memvalid=0;
    // rmmovq
    if(M_icode == 4'b0100)
    begin
      if(M_valE > 255)
        begin
            memvalid = 1;
        end
      data_memory[M_valE] <= M_valA;
    end

    // call
    else if(M_icode == 4'b1000)
    begin
      if(M_valE > 255)
        begin
            memvalid = 1;
        end
      data_memory[M_valE] <= M_valA;
    end

    // pushq
    else if(M_icode == 4'b1010)
    begin
      if(M_valE > 255)
        begin
            memvalid = 1;
        end
      data_memory[M_valE] <= M_valA;
    end
  end

  always @(posedge clk)
    begin
      W_stat <= m_stat;
      W_icode <= M_icode;
      W_valE <= M_valE;
      W_valM <= m_valM;
      W_destE <= M_destE;
      W_destM <= M_destM;
    end
endmodule
```

Overall implementation of Y86-64 processor (5 Stage Pipeline)



Data Forwarding-

- In Naïve Pipeline, Register isn't written until completion of write-back stage and Source operands read from register file in decode stage.
- In data forwarding, we take the result from the earliest point that it exists in any of the pipeline state registers and forward it to the functional units that need it that cycle.
- In case of multiple forwarding choices, use matching value from the earliest pipeline stage.

Implementation-

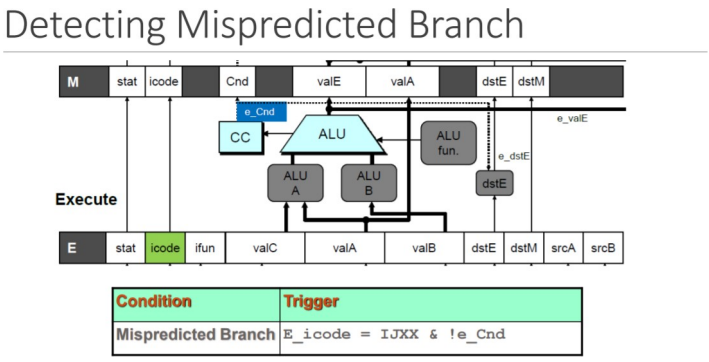
- Add additional feedback paths from E, M, and W pipeline registers into decode stage
- Create logic blocks to select from multiple sources for valA and valB in decode stage
- Forwarding Sources -

Data word	Register ID	Source description
e_valE	e_dstE	ALU output
m_valM	M_dstM	Memory output
M_valE	M_dstE	Pending write to port E in memory stage
W_valM	W_dstM	Pending write to port M in write-back stage
W_valE	W_dstE	Pending write to port E in write-back stage

Branch Misprediction Case -

Branch misprediction occurs mainly in the case of jump (jXX).

A misprediction can incur a serious penalty causing a serious degradation of program performance.

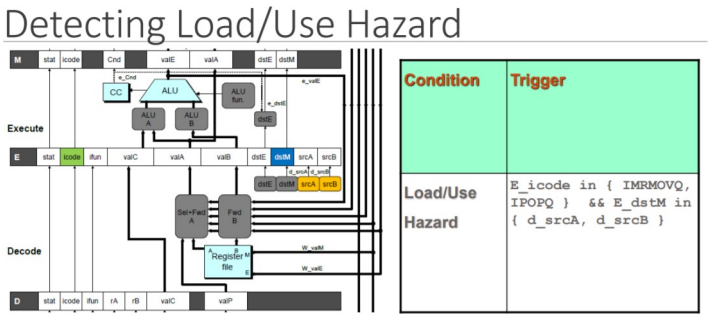


Handling Misprediction -

- Fetch 2 instructions at the target where branch is taken
- In execute stage, detect whether branch is taken or not, cancel When mispredicted.
- For no side effects, on the following cycle, replace instructions in execute and decode bubbles.

Load/Use Hazard Case -

- A load-use hazard requires delaying the execution of the using instruction until the result from the loading instruction can be made available to the using instruction.



> Control for Load/Use Hazard -

- Stall instructions in fetch and decode stages
- Inject bubble into execute stage

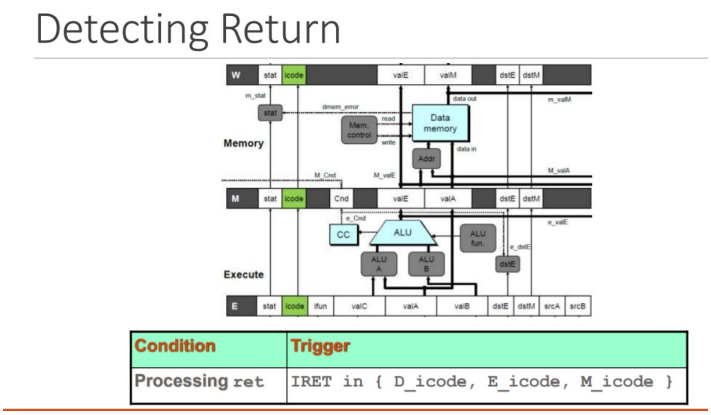
PipeLine Control

```
module pipeline_ctrl(D_icode,d_srcA,d_srcB,E_icode,E_destM,e_Cnd,M_icode,m_stat,W_stat,setcc,F_stall,D_stall,D_bubble,E_bubble,M_bubbl

input [3:0] D_icode;
input [3:0] d_srcA;
input [3:0] d_srcB;
input [3:0] E_icode;
input [3:0] E_destM;
input e_Cnd;
input [3:0] M_icode;
input [0:3] m_stat;
input [0:3] W_stat;

output reg setcc;
output reg F_stall;
output reg D_stall;
output reg D_bubble;
output reg E_bubble;
output reg M_bubble;
output reg W_stall;

always @(*) begin
    setcc = 1'b1;
    F_stall = 1'b0;
    D_stall = 1'b0;
    D_bubble = 1'b0;
```

```
    E_bubble = 1'b0;
    M_bubble = 1'b0;
    W_stall = 1'b0;

    F_stall <= ( ((E_icode == 4'h3 || E_icode == 4'hB) && (E_destM == d_srcA || E_destM == d_srcB)) || (D_icode == 4'h9 || E_icode ==
    D_stall <= ((E_icode == 4'h3 || E_icode == 4'hB) && (E_destM == d_srcA || E_destM == d_srcB));
    D_bubble <= (( E_icode == 4'h7 && !e_Cnd ) || (!(E_icode == 4'h3 || E_icode == 4'hB)  && (E_destM == d_srcA || E_destM== d_srcB) &
    E_bubble <= (( E_icode == 4'h7 && !e_Cnd ) || ((E_icode == 4'h3 || E_icode == 4'hB)  && (E_destM == d_srcA || E_destM== d_srcB)));

    if(E_icode == 4'h0 | m_stat != 4'b1000 | W_stat != 4'b1000)
        begin
            setcc <= 1'b0;
        end
    end

endmodule
```

Return Condition -

- For the return condition to be implemented, the return point should be known.
- Before the instruction's return point is executed, next instructions are fetched in between which should not be executed.
- Return point is known in the memory stage of the return instruction. Hence we need to handle this special control case.



Handling ret case -

-> As ret passes through the pipeline, stall at the fetch stage.

- > Inject bubble into the decode stage.
- > Release stall when reach write-back stage.

INPUT

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

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

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

Instruction_memory[14] = 8'h40;//rmmovq
Instruction_memory[15] = 8'h24;
{Instruction_memory[16],Instruction_memory[17],Instruction_memory[18],Instruction_memory[19],Instruction_memory[20],Instruction_memory

Instruction_memory[24] = 8'h40;//rmmovq
Instruction_memory[25] = 8'h53;
{Instruction_memory[26],Instruction_memory[27],Instruction_memory[28],Instruction_memory[29],Instruction_memory[30],Instruction_memory
```

```
Instruction_memory[34] = 8'h50;//mrmovq
Instruction_memory[35] = 8'h53;
{Instruction_memory[36],Instruction_memory[37],Instruction_memory[38],Instruction_memory[39],Instruction_memory[40],Instruction_memory

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

Instruction_memory[46] = 8'h73;//je
{Instruction_memory[47],Instruction_memory[48],Instruction_memory[49],Instruction_memory[50],Instruction_memory[51],Instruction_memory

Instruction_memory[55] = 8'h00;

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

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

Instruction_memory[60] = 8'h80;//call
{Instruction_memory[61],Instruction_memory[62],Instruction_memory[63],Instruction_memory[64],Instruction_memory[65],Instruction_memory

Instruction_memory[69] = 8'h60;//OP
Instruction_memory[70] = 8'h56;

Instruction_memory[71] = 8'h70;//jump unconditional
{Instruction_memory[72],Instruction_memory[73],Instruction_memory[74],Instruction_memory[75],Instruction_memory[76],Instruction_memory

Instruction_memory[80] = 8'h30;//irmovq
Instruction_memory[81] = 8'hF2;
Instruction_memory[82] = 8'h00;
Instruction_memory[83] = 8'h00;
Instruction_memory[84] = 8'h00;
Instruction_memory[85] = 8'h00;
Instruction_memory[86] = 8'h00;
Instruction_memory[87] = 8'h00;
Instruction_memory[88] = 8'h00;
Instruction_memory[89] = 8'b00000010;

Instruction_memory[90] = 8'h60;//OPq
Instruction_memory[91] = 8'h9A;

Instruction_memory[92] = 8'h10;//no op
Instruction_memory[93] = 8'h00;//halt

Instruction_memory[94] = 8'h90;// return
```

OUTPUT

```
[Running] processor.v
VCD info: dumpfile processor.vcd opened for output.

------------------------------------------------------------
Fetch stage :- clk=1 D_stat= x F_predPC=                  0 f_predPC=                  1 icode= x ifun= x
     > rsp =                  x rA= x rB= x valC=                  x D_valP=                  x
Decode stage :- clk=1 E_stat= x icode= x ifun= x rsp =                     x
     > valA=                  x valB=                 x valC=                  x destE= x destM= x srcA= x srcB= x
Execute stage :- clk=1 M_stat= x icode= x rsp =                    x
     > CND=0 OF0 SF0 ZF0 valA =                  x valE=                  x destE= x destM= x
Memory stage :- clk=1 W_stat= x icode= x rsp=                   x
     > valM =                   x valE=                  x destE= x destM= x
fetch_stallx D_bubblex D_stallx E_bubblex M_bubble0
------------------------------------------------------------

------------------------------------------------------------
Fetch stage :- clk=1 D_stat= 8 F_predPC=                   1 f_predPC=                  2 icode= 1 ifun= 0
     > rsp =                254 rA= x rB= x valC=                  x D_valP=                  1
Decode stage :- clk=1 E_stat= x icode= x ifun= x rsp =                   254
     > valA=                  x valB=                 x valC=                  x destE=15 destM=15 srcA=15 srcB=15
Execute stage :- clk=1 M_stat= x icode= x rsp =                  254
     > CND=0 OF0 SF0 ZF0 valA =                  x valE=                  x destE= x destM= x
Memory stage :- clk=1 W_stat= x icode= x rsp=                 254
     > valM =                   x valE=                  x destE= x destM= x
fetch_stallx D_bubblex D_stallx E_bubblex M_bubble0
------------------------------------------------------------

------------------------------------------------------------
Fetch stage :- clk=1 D_stat= 8 F_predPC=                   2 f_predPC=                  4 icode= 1 ifun= 0
     > rsp =                254 rA= x rB= x valC=                  x D_valP=                  2
Decode stage :- clk=1 E_stat= 8 icode= 1 ifun= 0 rsp =                   254
     > valA=                  x valB=                 x valC=                  x destE=15 destM=15 srcA=15 srcB=15
Execute stage :- clk=1 M_stat= x icode= x rsp =                  254
     > CND=0 OF0 SF0 ZF0 valA =                  x valE=                  x destE=15 destM=15
Memory stage :- clk=1 W_stat= x icode= x rsp=                 254
```

```
     > valM =                   x valE=                  x destE= x destM= x
fetch_stallx D_bubblex D_stall0 E_bubble0 M_bubble0
------------------------------------------------------------

------------------------------------------------------------
Fetch stage :- clk=1 D_stat= 8 F_predPC=                   4 f_predPC=                  14 icode= 2 ifun= 0
     > rsp =                 254 rA= 1 rB= 2 valC=                   x D_valP=                   4
Decode stage :- clk=1 E_stat= 8 icode= 1 ifun= 0 rsp =                 254
     > valA=                   x valB=                  x valC=                  x destE=15 destM=15 srcA=15 srcB=15
Execute stage :- clk=1 M_stat= 8 icode= 1 rsp =                254
     > CND=0 OF0 SF0 ZF0 valA =                   x valE=                   x destE=15 destM=15
Memory stage :- clk=1 W_stat= x icode= x rsp=               254
     > valM =                   x valE=                  x destE=15 destM=15
fetch_stall0 D_bubble0 D_stall0 E_bubble0 M_bubble0
------------------------------------------------------------

------------------------------------------------------------
Fetch stage :- clk=1 D_stat= 8 F_predPC=                  14 f_predPC=                  24 icode= 3 ifun= 0
     > rsp =                 254 rA=15 rB= 2 valC=                   2 D_valP=                  14
Decode stage :- clk=1 E_stat= 8 icode= 2 ifun= 0 rsp =                 254
     > valA=                  10 valB=                   0 valC=                  x destE= 2 destM=15 srcA= 1 srcB=15
Execute stage :- clk=1 M_stat= 8 icode= 1 rsp =                254
     > CND=1 OF0 SF0 ZF0 valA =                   x valE=                   x destE=15 destM=15
Memory stage :- clk=1 W_stat= 8 icode= 1 rsp=               254
     > valM =                   x valE=                  x destE=15 destM=15
fetch_stall0 D_bubble0 D_stall0 E_bubble0 M_bubble0
------------------------------------------------------------

------------------------------------------------------------
Fetch stage :- clk=1 D_stat= 8 F_predPC=                  24 f_predPC=                  34 icode= 4 ifun= 0
     > rsp =                 254 rA= 2 rB= 4 valC=                   1 D_valP=                  24
Decode stage :- clk=1 E_stat= 8 icode= 3 ifun= 0 rsp =                 254
     > valA=                  10 valB=                   0 valC=                   2 destE= 2 destM=15 srcA= 1 srcB=15
Execute stage :- clk=1 M_stat= 8 icode= 2 rsp =                254
     > CND=0 OF0 SF0 ZF0 valA =                  10 valE=                  10 destE= 2 destM=15
Memory stage :- clk=1 W_stat= 8 icode= 1 rsp=               254
     > valM =                   x valE=                  x destE=15 destM=15
fetch_stall0 D_bubble0 D_stall0 E_bubble0 M_bubble0
------------------------------------------------------------

------------------------------------------------------------
Fetch stage :- clk=1 D_stat= 8 F_predPC=                  34 f_predPC=                  44 icode= 4 ifun= 0
     > rsp =                 254 rA= 5 rB= 3 valC=                   0 D_valP=                  34
Decode stage :- clk=1 E_stat= 8 icode= 4 ifun= 0 rsp =                 254
     > valA=                   2 valB=                 254 valC=                   1 destE= 2 destM=15 srcA= 2 srcB= 4
Execute stage :- clk=1 M_stat= 8 icode= 3 rsp =                254
     > CND=0 OF0 SF0 ZF0 valA =                  10 valE=                   2 destE= 2 destM=15
Memory stage :- clk=1 W_stat= 8 icode= 2 rsp=               254
     > valM =                   x valE=                  10 destE= 2 destM=15
fetch_stall0 D_bubble0 D_stall0 E_bubble0 M_bubble0
------------------------------------------------------------

------------------------------------------------------------
Fetch stage :- clk=1 D_stat= 8 F_predPC=                  44 f_predPC=                  46 icode= 5 ifun= 0
     > rsp =                 254 rA= 5 rB= 3 valC=                   0 D_valP=                  44
Decode stage :- clk=1 E_stat= 8 icode= 4 ifun= 0 rsp =                 254
     > valA=                  50 valB=                   3 valC=                   0 destE= 2 destM=15 srcA= 5 srcB= 3
Execute stage :- clk=1 M_stat= 8 icode= 4 rsp =                254
     > CND=0 OF0 SF0 ZF0 valA =                   2 valE=                 255 destE= 2 destM=15
Memory stage :- clk=1 W_stat= 8 icode= 3 rsp=               254
     > valM =                   x valE=                   2 destE= 2 destM=15
fetch_stall0 D_bubble0 D_stall0 E_bubble0 M_bubble0
------------------------------------------------------------

------------------------------------------------------------
Fetch stage :- clk=1 D_stat= 8 F_predPC=                  46 f_predPC=                  56 icode= 6 ifun= 0
     > rsp =                 254 rA= 9 rB=10 valC=                   0 D_valP=                  46
Decode stage :- clk=1 E_stat= 8 icode= 5 ifun= 0 rsp =                 254
     > valA=                  50 valB=                   3 valC=                   0 destE= 2 destM= 5 srcA= 5 srcB= 3
Execute stage :- clk=1 M_stat= 8 icode= 4 rsp =                254
     > CND=0 OF0 SF0 ZF0 valA =                  50 valE=                   3 destE= 2 destM=15
Memory stage :- clk=1 W_stat= 8 icode= 4 rsp=               254
     > valM =                   x valE=                 255 destE= 2 destM=15
fetch_stall0 D_bubble0 D_stall0 E_bubble0 M_bubble0
------------------------------------------------------------

------------------------------------------------------------
Fetch stage :- clk=1 D_stat= 8 F_predPC=                  56 f_predPC=                  58 icode= 7 ifun= 3
     > rsp =                 254 rA= 9 rB=10 valC=                  56 D_valP=                  55
Decode stage :- clk=1 E_stat= 8 icode= 6 ifun= 0 rsp =                 254
     > valA=              -12345 valB=               12345 valC=                   0 destE=10 destM= 5 srcA= 9 srcB=10
Execute stage :- clk=1 M_stat= 8 icode= 5 rsp =                254
     > CND=0 OF0 SF0 ZF1 valA =                  50 valE=                   3 destE= 2 destM= 5
Memory stage :- clk=1 W_stat= 8 icode= 4 rsp=               254
     > valM =                   x valE=                   3 destE= 2 destM=15
fetch_stall0 D_bubble0 D_stall0 E_bubble0 M_bubble0
------------------------------------------------------------
```

```
------------------------------------------------------------
Fetch stage :- clk=1 D_stat= 8 F_predPC=                  58 f_predPC=                  60 icode=10 ifun= 0
     > rsp =                 254 rA= 9 rB=15 valC=                  56 D_valP=                  58
Decode stage :- clk=1 E_stat= 8 icode= 7 ifun= 3 rsp =                 254
     > valA=                  55 valB=              12345 valC=                 56 destE=15 destM=15 srcA=15 srcB=15
Execute stage :- clk=1 M_stat= 8 icode= 6 rsp =                254
     > CND=1 OF0 SF0 ZF1 valA =             -12345 valE=                   0 destE=10 destM= 5
Memory stage :- clk=1 W_stat= 8 icode= 5 rsp=                254
     > valM =                 50 valE=                 3 destE= 2 destM= 5
fetch_stall0 D_bubble0 D_stall0 E_bubble0 M_bubble0
------------------------------------------------------------

------------------------------------------------------------
Fetch stage :- clk=1 D_stat= 8 F_predPC=                  60 f_predPC=                  80 icode=11 ifun= 0
     > rsp =                 254 rA= 9 rB=15 valC=                  56 D_valP=                  60
Decode stage :- clk=1 E_stat= 8 icode=10 ifun= 0 rsp =                 254
     > valA=             -12345 valB=                254 valC=                 56 destE= 4 destM=15 srcA= 9 srcB= 4
Execute stage :- clk=1 M_stat= 8 icode= 7 rsp =                254
     > CND=0 OF0 SF0 ZF1 valA =                  55 valE=                   0 destE=15 destM=15
Memory stage :- clk=1 W_stat= 8 icode= 6 rsp=                254
     > valM =                 50 valE=                 0 destE=10 destM= 5
fetch_stall0 D_bubble0 D_stall0 E_bubble0 M_bubble0
------------------------------------------------------------

------------------------------------------------------------
Fetch stage :- clk=1 D_stat= 8 F_predPC=                  80 f_predPC=                  90 icode= 8 ifun= 0
     > rsp =                 254 rA= 9 rB=15 valC=                  80 D_valP=                  69
Decode stage :- clk=1 E_stat= 8 icode=11 ifun= 0 rsp =                 254
     > valA=                 253 valB=                253 valC=                 56 destE= 4 destM= 9 srcA= 4 srcB= 4
Execute stage :- clk=1 M_stat= 8 icode=10 rsp =                254
     > CND=0 OF0 SF0 ZF1 valA =             -12345 valE=                253 destE= 4 destM=15
Memory stage :- clk=1 W_stat= 8 icode= 7 rsp=                254
     > valM =                 50 valE=                 0 destE=15 destM=15
fetch_stall0 D_bubble0 D_stall0 E_bubble0 M_bubble0
------------------------------------------------------------

------------------------------------------------------------
Fetch stage :- clk=1 D_stat= 8 F_predPC=                  90 f_predPC=                  92 icode= 3 ifun= 0
     > rsp =                 254 rA=15 rB= 2 valC=                   2 D_valP=                  90
Decode stage :- clk=1 E_stat= 8 icode= 8 ifun= 0 rsp =                 254
     > valA=                  69 valB=                254 valC=                 80 destE= 4 destM= 9 srcA= 4 srcB= 4
Execute stage :- clk=1 M_stat= 8 icode=11 rsp =                254
     > CND=0 OF0 SF0 ZF1 valA =                253 valE=                254 destE= 4 destM= 9
Memory stage :- clk=1 W_stat= 8 icode=10 rsp=                254
     > valM =                 50 valE=                253 destE= 4 destM=15
fetch_stall0 D_bubble0 D_stall0 E_bubble0 M_bubble0
------------------------------------------------------------

------------------------------------------------------------
Fetch stage :- clk=1 D_stat= 8 F_predPC=                  92 f_predPC=                  93 icode= 6 ifun= 0
     > rsp =                 253 rA= 9 rB=10 valC=                   2 D_valP=                  92
Decode stage :- clk=1 E_stat= 8 icode= 3 ifun= 0 rsp =                 253
     > valA=                 253 valB=                253 valC=                  2 destE= 2 destM= 9 srcA= 4 srcB= 4
Execute stage :- clk=1 M_stat= 8 icode= 8 rsp =                253
     > CND=0 OF0 SF0 ZF1 valA =                  69 valE=                253 destE= 4 destM= 9
Memory stage :- clk=1 W_stat= 8 icode=11 rsp=                253
     > valM =              -12345 valE=                254 destE= 4 destM= 9
fetch_stall1 D_bubble0 D_stall1 E_bubble1 M_bubble0
------------------------------------------------------------
halt
```

Challenges Faced -

-> We faced difficulty in implementing data forwarding.

- > We also faced difficulty in implementing stalls and bubbles.
- > Initially we took time to understand how the 5 stages of the pipeline are implemented in a single clock cycle.
- > It also took time while transitioning from sequential to pipeline implementation.

Acknowledgment -

Working on this project is interesting and provided us with a great learning experience. Over the last month, we have learnt a lot about Processor Architecture. However, it would not have been possible without the kind support and help of our TA Akshith Gureja. He never hesitated to reply to our messages and correct our mistakes. We are highly indebted to him.

We would also like to express our gratitude towards Prof. Deepak Gangadharan for his support in completing the project.