Y86-64 ISA Implementation

REPORT

VYSISHTYA KARANAM **2022102044**

SAHASRA VENEPALLY **2022102024**

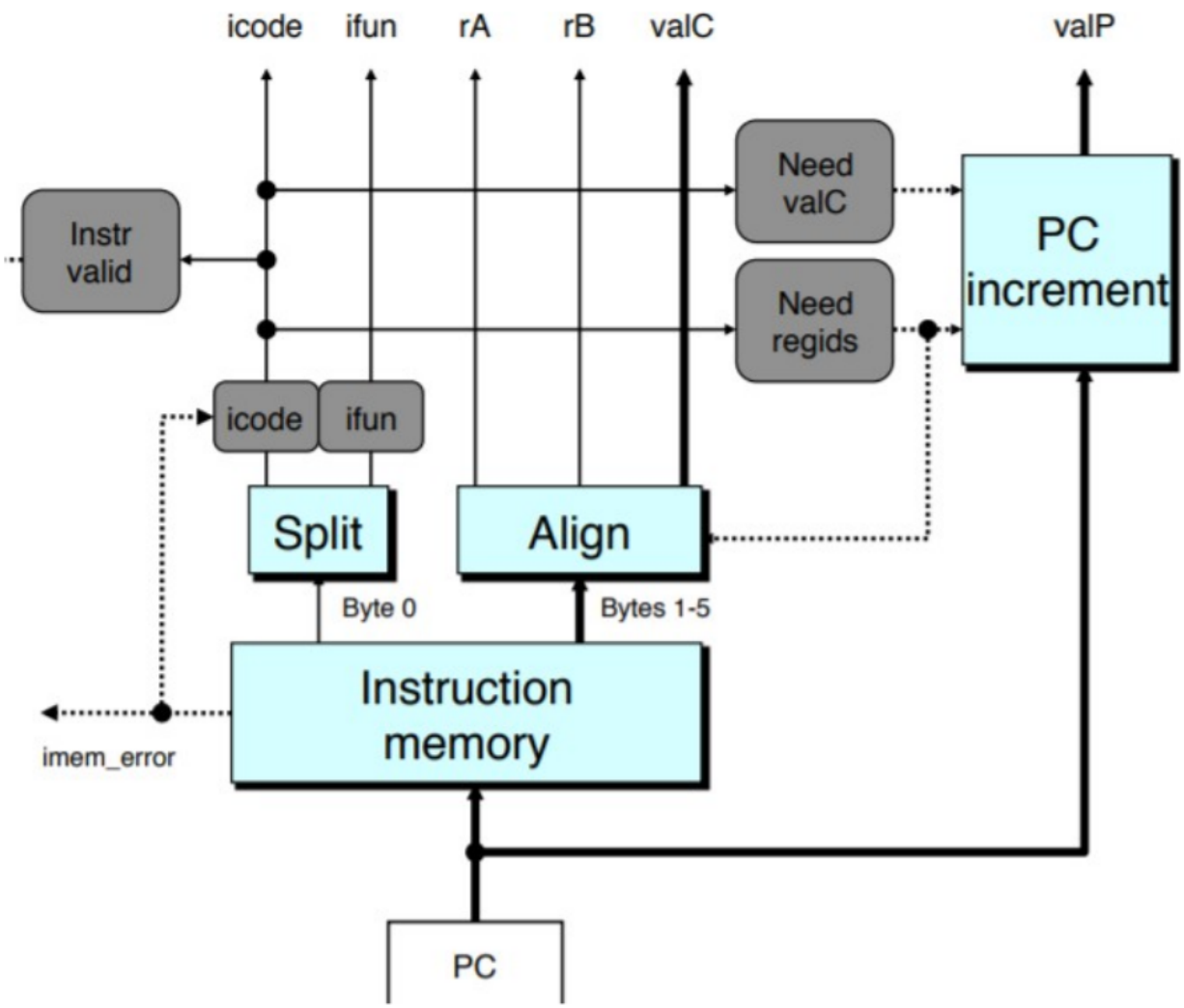
Aim :

The end goal 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.

Sequential:

We have six modules in sequential implementation of Y86-64 processor . The stages are as follows.

FETCH :

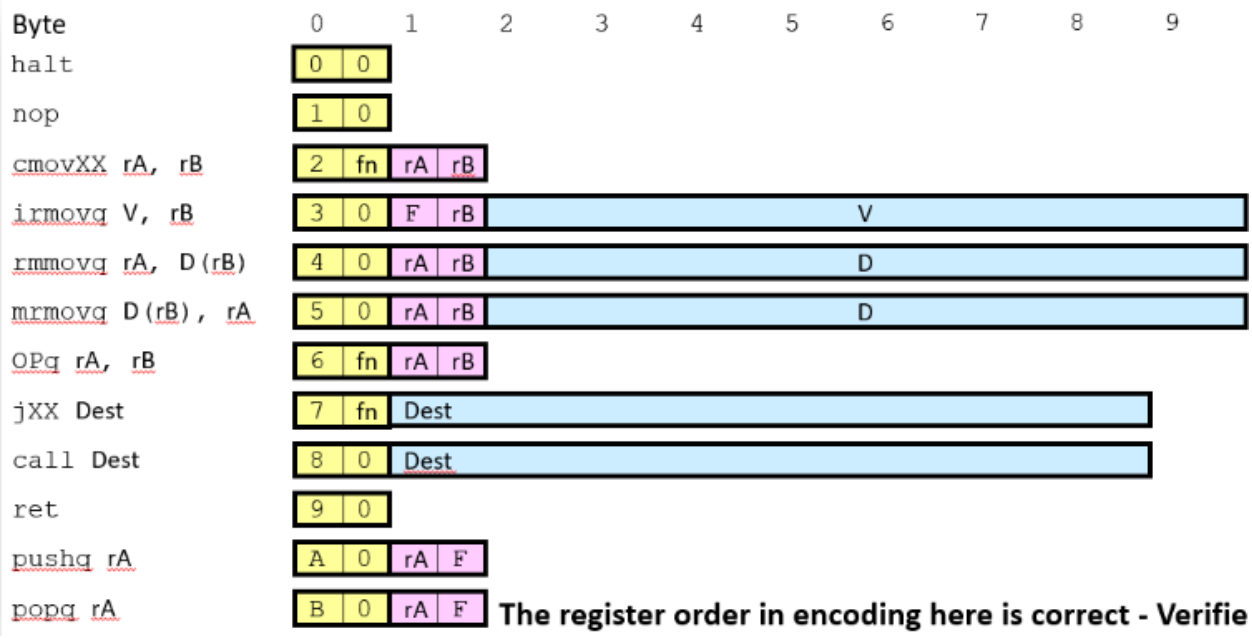


Stage	CALL	RET	PUSHQ	POPQ
Fch	icode:ifun ← M_1[PC] valC ← M_8[PC+1] valP ← PC + 9	icode:ifun ← M_1[PC] valP ← PC + 1	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valP ← PC + 2	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valP ← PC + 2

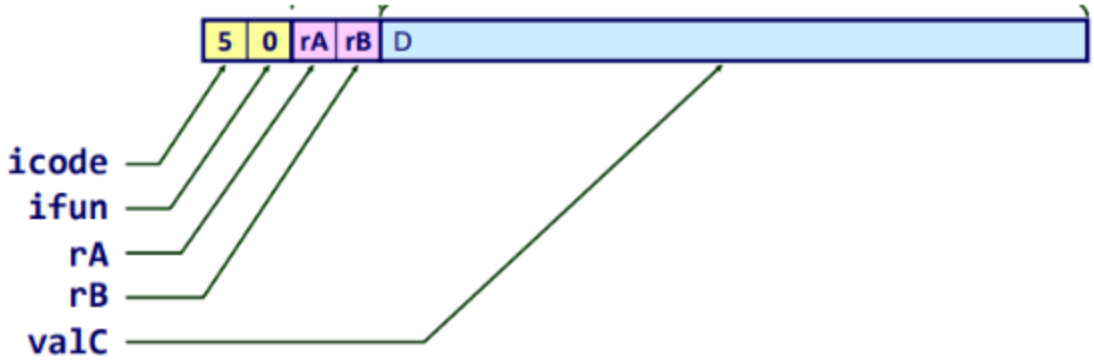
Stage	RMMOVQ	MRMOVQ	OPq	jXX
Fch	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valC ← M_8[PC+2] valP ← PC + 10	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valC ← M_8[PC+2] valP ← PC + 10	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valP ← PC + 2	icode:ifun ← M_1[PC] valC ← M_8[PC+1] valP ← PC + 9

Stage	HALT	NOP	CMOV	IRMOVQ
Fch	icode:ifun ← M_1[PC] valP ← PC + 1	icode:ifun ← M_1[PC] valP ← PC + 1	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valP ← PC + 2	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valC ← M_8[PC+2] valP ← PC + 10

Fetch block uses instruction array which may each have 10 bytes to compute icode , ifun , rA, rB, valC, valP, instruction error and halting first 1 byte represent icode : ifun where icode is of 4bits and ifun is of 4 bits. Second byte represents the registers rA, rB, then after all the bits represents destination offset.



ifun of the instruction represents the condition required to execute the respective instruction. Instruction is set to 0 if the instruction given is wrong. inst_mem error is set to 0 if the pc value stays within 1023. And when halt is encountered halt is set to 1.



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

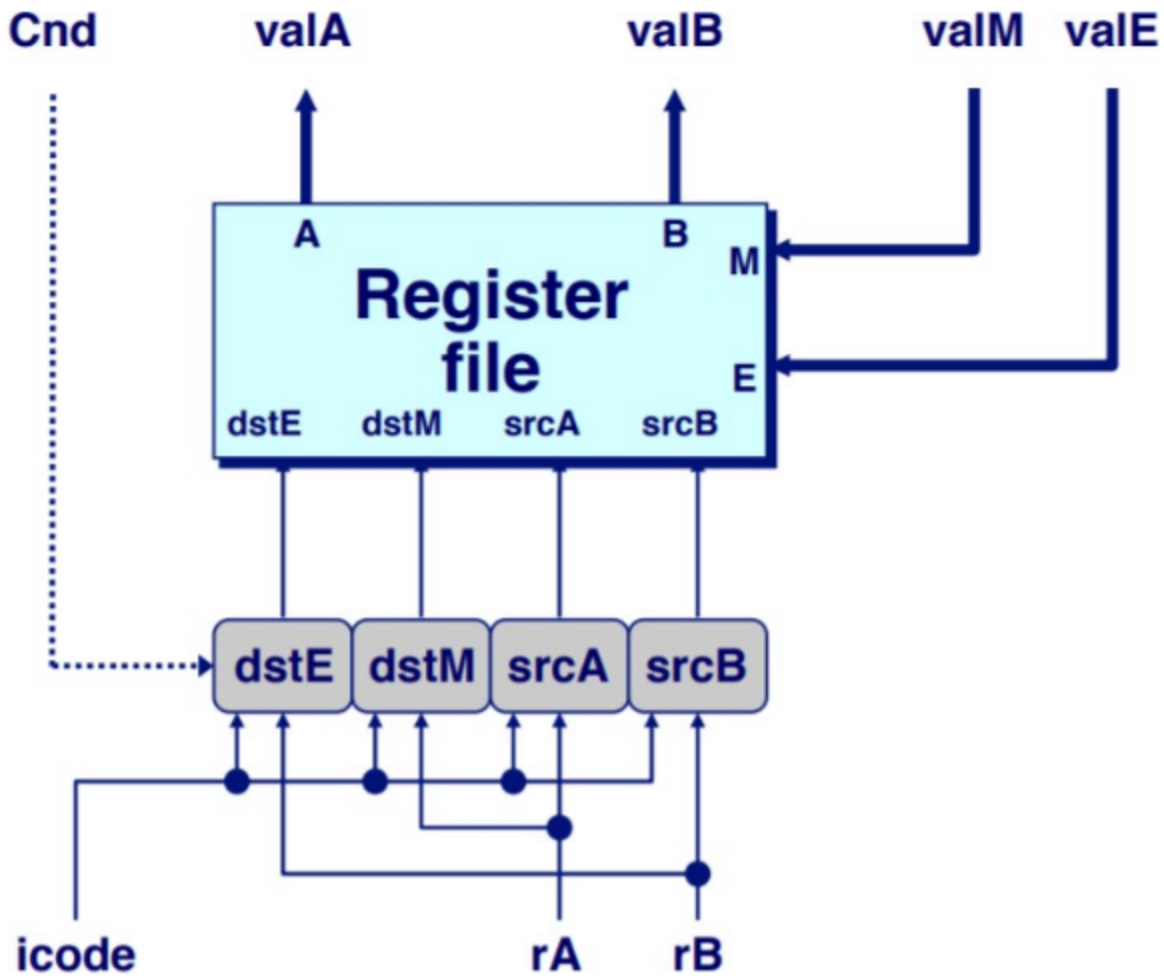
- In case of halt, icode = 0, ifun = 0, valP = PC+64'd1
- In case of nop, icode = 1, ifun = 0, valP = PC+64'd2
- In case of cmovXX, icode = 2, ifun for - rrmovq = 0, cmovle = 1, cmovl = 2, cmove = 3 , cmovne = 4, cmovge = 5, cmovg = 6 valP = PC+64'd2
- In case of irmovq, icode = 3, ifun = 0, valP = PC+64'd10
- 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 ,instruct,  icode , ifun , ra , rb , valC , valP,halt,instruct_err,mem_err ) ;
  reg [0:7] Split ;
  reg [0:71] Align ;
  reg [0:0] need_valC;
  input [63:0] PC ;
  input clk ;
  input [0:79] instruct;
  output reg [3:0] ifun,icode,ra ,rb ;
  output reg signed[63:0] valC,valP ;
  output reg instruct_err, halt,mem_err;
always@(*)
  begin
    Split = {instruct[0:7]} ;
    Align = {instruct[8:79]} ;
    icode = Split[0:3];
    ifun  = Split[4:7];
    instruct_err = 1'b0 ;
    halt=0;
    mem_err = 1'b0;
    if(PC> 64) begin
      mem_err=1;
    end
    if (icode == 4'b0011 || icode == 4'b0110 || icode == 4'b1010 || icode == 4'b1011 || icode ==
4'b0100 || icode == 4'b0101) begin
        need_valC = 1'b1;
    end
    if(icode == 4'b0000)  // halt
    begin
      halt=1;
      valP = PC + 64'd1;
      $finish;
    end
    else if(icode == 4'b0001) //nop
    begin
      valP = PC + 64'd1;
    end
    else if(icode == 4'b0010) //cmovxx
    begin
      ra = Align[0:3];
      rb = Align[4:7];
      valP = PC + 64'd2;
    end
    else if(icode == 4'b0011) //irmovq
    begin
      ra = Align[0:3];
      rb = Align[4:7];
      if (need_valC)
          valC = instruct[16:79];
      valP = PC + 64'd10;
    end
    else if(icode == 4'b0100) //rmmovq
    begin
      ra = Align[0:3];
      rb = Align[4:7];
      if (need_valC)
          valC = instruct[16:79];
      valP = PC + 64'd10;
    end
```

```
    else if(icode == 4'b0101) //mrmovq
    begin
      ra = Align[0:3];
      rb = Align[4:7];
      if (need_valC)
          valC = instruct[16:79];
      valP = PC + 64'd10;
    end
    else if(icode == 4'b0110) //OPq
    begin
      ra = Align[0:3];
      rb = Align[4:7];
      valP = PC + 64'd2;
    end
    else if(icode==4'b0111) //jxx
    begin
      valC = instruct[8:71];
      valP = PC + 64'd9;
    end
    else if(icode == 4'b1000) //call
    begin
      if (need_valC)
          valC = instruct[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 = Align[0:3];
      rb = Align[4:7];
      valP = PC + 64'd2;
    end
    else if(icode==4'b1011) //popq
    begin
      ra = Align[0:3];
      rb = Align[4:7];
      valP = PC + 64'd2;
    end
    else
    begin
      instruct_err = 1'b1;
    end
    end

endmodule
```

DECODE AND WRITEBACK :



We have 15 registers in the Y86-64 processor which are to be accessed in the decode stage. The register file represents those 15 registers.

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.

The implementation of Decode and WriteBack stages are :

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 , regis0 , regis1 ,
regis2 ,regis3 , regis4 , regis5 ,regis6 ,regis7 ,regis8 , regis9 , regis10 ,regis11 , regis12 ,
regis13 , regis14) ;
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] regis[0:14] ;
output reg signed [63:0] regis0;
output reg signed [63:0] regis1;
output reg signed [63:0] regis2;
output reg signed [63:0] regis3;
output reg signed [63:0] regis4;
output reg signed [63:0] regis5;
output reg signed [63:0] regis6;
output reg signed [63:0] regis7;
output reg signed [63:0] regis8;
output reg signed [63:0] regis9;
output reg signed [63:0] regis10;
output reg signed [63:0] regis11;
output reg signed [63:0] regis12;
output reg signed [63:0] regis13;
output reg signed [63:0] regis14;
initial
  begin
    regis[0] = 64'd12;        //rax
    regis[1] = 64'd10;        //rcx
    regis[2] = 64'd141;       //rdx
    regis[3] = 64'd3;         //rbx
    regis[4] = 64'd254;       //rsp
    regis[5] = 64'd50;        //rbp
    regis[6] = -64'd143;      //rsi
    regis[7] = 64'd10000;     //rdi
    regis[8] = 64'd990000;    //r8
    regis[9] = -64'd12345;    //r9
    regis[10] = 64'd12345;    //r10
    regis[11] = 64'd10112;    //r11
    regis[12] = 64'd8;        //r12
    regis[13] = 64'd1567;     //r13
    regis[14] = 64'd8643;     //r14
  end
always@(*)
  begin
    if(icode == 4'b0010) //cmovxx
    begin
      valA = regis[ra] ;
      valB = 0 ;
    end
    else if(icode == 4'b0100) //rmmovq
    begin
      valA = regis[ra] ;
      valB = regis[rb] ;
    end
    else if(icode == 4'b0101) //mrmovq
    begin
      valA = 0 ;
      valB = regis[rb] ;
    end
    else if(icode == 4'b0110) //OPq
    begin
      valA = regis[ra] ;
      valB = regis[rb] ;
    end
    else if(icode == 4'b1000) //call
    begin
      valB = regis[4] ;
    end
    else if(icode == 4'b1001) //ret
    begin
      valA = regis[4] ;
      valB = regis[4] ;
    end
    else if(icode == 4'b1010) //pushq
    begin
      valA = regis[ra] ;
      valB = regis[4] ;
    end
    else if(icode == 4'b1011) //popq
    begin
      valA = regis[4] ;
      valB = regis[4] ;
    end
  end
```

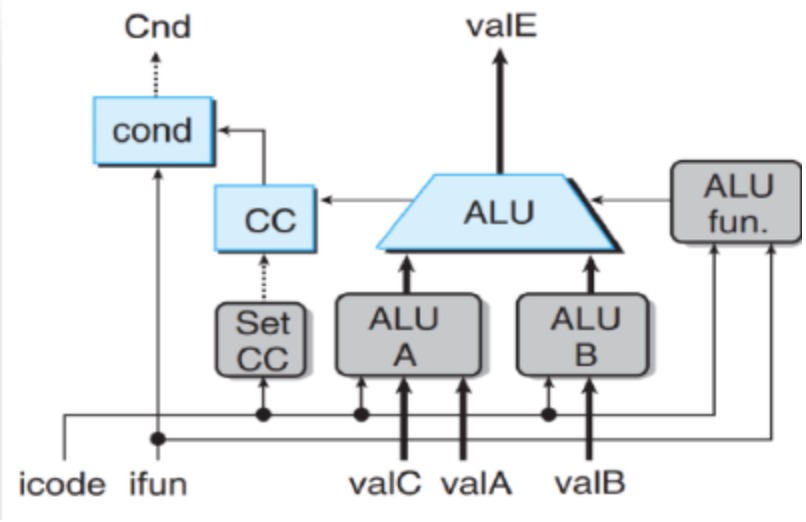
DECODE

```
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
          regis[rb] = valE ;
        end
    end
    else if(icode==4'b0011) //irmovq
      begin
        regis[rb] = valE;
      end
    else if(icode == 4'b0101) //mrmovq
      begin
          regis[ra] = valM ;
      end
    else if(icode == 4'b0010) //rrmovq
      begin
          regis[rb] = valE ;
      end
    else if(icode == 4'b0110) //OPq
      begin
          regis[rb] = valE ;
      end
    else if(icode == 4'b1000) //call
      begin
          regis[4] = valE ;
      end
    else if(icode == 4'b1001) //ret
      begin
        regis[4] = valE ;
      end

    else if(icode == 4'b1001) //ret
      begin
        regis[4] = valE ;
      end
    else if(icode == 4'b1010) //pushq
      begin
          regis[4] = valE ;
      end
    else if(icode == 4'b1011) //popq
      begin
          regis[4] = valE ;
          regis[ra] = valM ;
      end
    regis0 <= regis[0];
    regis1 <= regis[1];
    regis2 <= regis[2];
    regis3 <= regis[3];
    regis4 <= regis[4];
    regis5 <= regis[5];
    regis6 <= regis[6];
    regis7 <= regis[7];
    regis8 <= regis[8];
    regis9 <= regis[9];
    regis10 <= regis[10];
    regis11 <= regis[11];
    regis12 <= regis[12];
    regis13 <= regis[13];
    regis14 <= regis[14];
  end
endmodule
```

WRITEBACK

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

Codes of Conditions:

1. Carry Flag (CF): For unsigned operations, this is used to identify overflow. The most recent operation produced by executing the most important bit determines this.
2. Zero Flag (ZF): When the most recent operation gives zero, this flag is activated.
3. Sign Flag (SF) - This flag is activated when a negative result is obtained from the most recent procedure.
4. Overflow Flag (OF): If a two's complement overflow—positive or negative—was triggered by the most recent operation, this flag is activated.
   The carry and overflow flags are reset to zero in the event of logical operations.
   When there is a shift operation, the overflow flag is set to zero and the carry flag is set to the last shift out.

Jump instructions along with 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 along with condition codes -

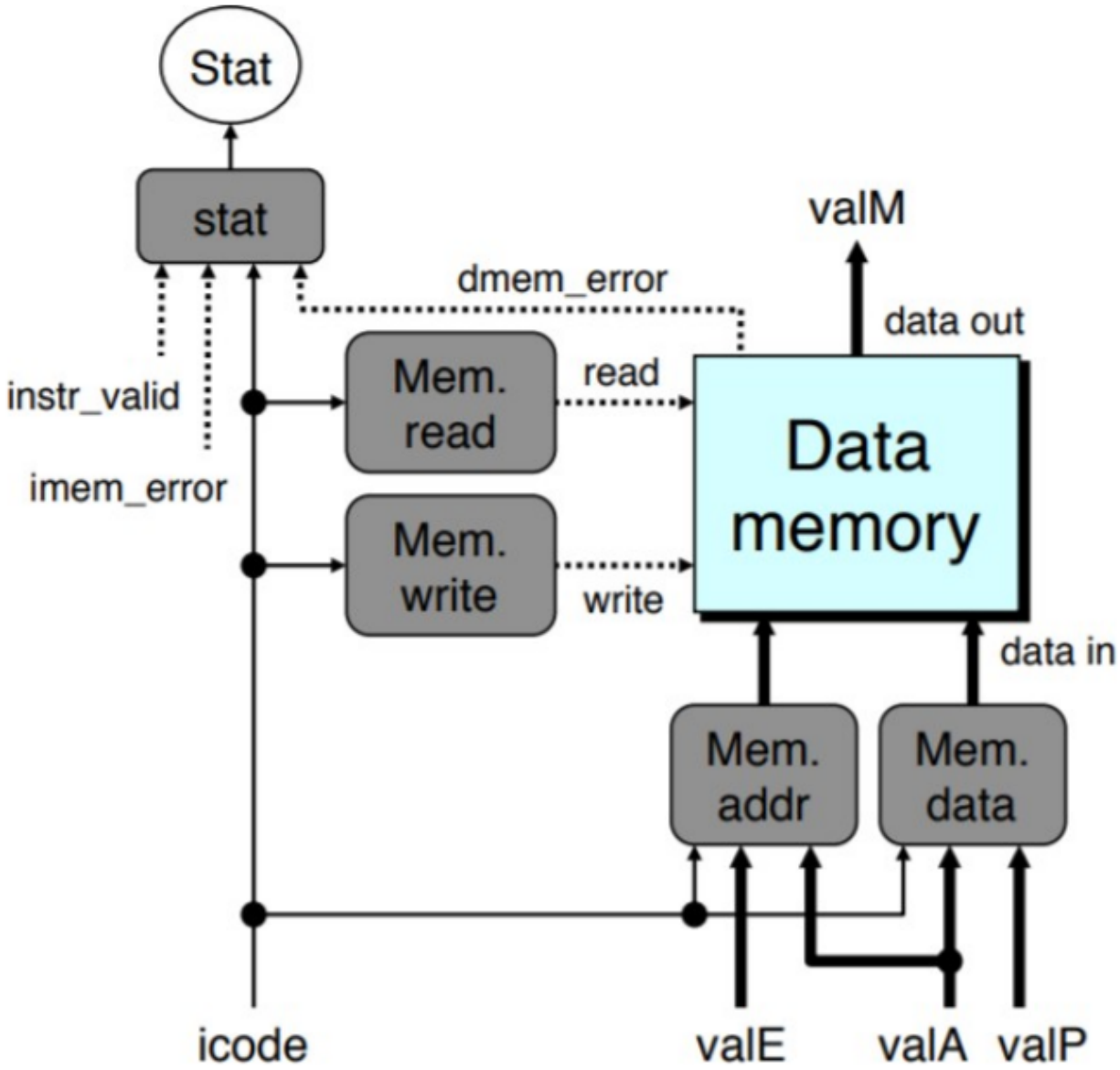
Instruction		Synonym	Move condition	Description
`cmove`	*S*, *R*	`cmovz`	`ZF`	Equal / zero
`cmovne`	*S*, *R*	`cmovnz`	`~ZF`	Not equal / not zero
`cmovs`	*S*, *R*		`SF`	Negative
`cmovns`	*S*, *R*		`~SF`	Nonnegative
`cmovg`	*S*, *R*	`cmovnle`	`~(SF ^ OF) & ~ZF`	Greater (signed >)
`cmovge`	*S*, *R*	`cmovnl`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

```
`include "alu.v"

module Execute(icode, ifun, valA, valB, valC, valE, clk, cnd, cc_in, cc_out, zf, sf, of);
  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, zf, sf, of;
  output reg [2:0] cc_out;
  reg [1:0] ctrl;
  reg signed [63:0] in1, in2, ans;
  wire signed [63:0] Output;
  wire overflo;
  ALU alu1(in1, in2, ctrl, Output, overflo);
  initial
  begin
    cnd = 0;
    zf <= cc_in[0];
    sf <= cc_in[1];
    of <= cc_in[2];
  end
  always @(*)
  begin
    case (icode)
      4'b0010: begin
        case (ifun)
          4'h0: cnd = 1;
          4'h1: cnd = sf | of; // le
          4'h2: cnd = (of ^ sf); // l
          4'h3: cnd = zf; // e
          4'h4: cnd = ~zf; // ne
          4'h5: cnd = ~(of ^ sf); // ge
          4'h6: cnd = (of ^ sf) | zf; // g
          default: cnd = 0;
        endcase
          ctrl = 2'b00;
          in1 = valA;
          in2 = 0;
      end
      4'b0011: begin // irmovq
        ctrl = 2'b00;
        in1 = valC;
        in2 = 0;
      end // irmovq
      4'b0100, 4'b0101: begin // rmmovq, mrmovq
        ctrl = 2'b00;
        in1 = valB;
        in2 = valC;
      end
      4'b0110: begin
        case (ifun)
          4'b0000: ctrl = 2'b00;
          4'b0001: ctrl = 2'b01;
          4'b0010: ctrl = 2'b10;
          4'b0011: ctrl = 2'b11;
          default: ctrl = 2'b00;
        endcase
        in1 <= valA;
        in2 <= valB;
      end
```

```
      4'b0111: begin // jmp
        case (ifun)
          4'h0: cnd = 1; // unconditional
          4'h1: cnd = sf | of; // le
          4'h2: cnd = (of ^ sf); // l
          4'h3: cnd = zf; // e
          4'h4: cnd = ~zf; // ne
          4'h5: cnd = ~(of ^ sf); // ge
          4'h6: cnd = (of ^ sf) | zf; // g
          default: cnd = 0;
        endcase
      end
      4'b1000:
          begin
              in1 = valB;
              in2 = 8;
              ctrl = 2'b01;
          end
      4'b1001:
          begin
              in1 = valB;
              in2 = 8;
              ctrl = 2'b00;
          end
      4'b1010:
          begin
              in1 = valB;
              in2 = 8;
              ctrl = 2'b01;
          end
      4'b1011:
          begin
              in1 = valB;
              in2 = 8;
              ctrl = 2'b00;
          end
    endcase
      ans = Output;
      valE <= Output;
    if(Output!=0)
      begin
        zf <= 0;
      end
      else
      begin
        zf <= 1;
      end
      sf <= Output[63];
      of <= overflo;
      cc_out[0] <= zf;
      cc_out[1] <= sf;
      cc_out[2] <= of;
  end
endmodule
```

MEMORY IMPLEMENTATION :



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 -

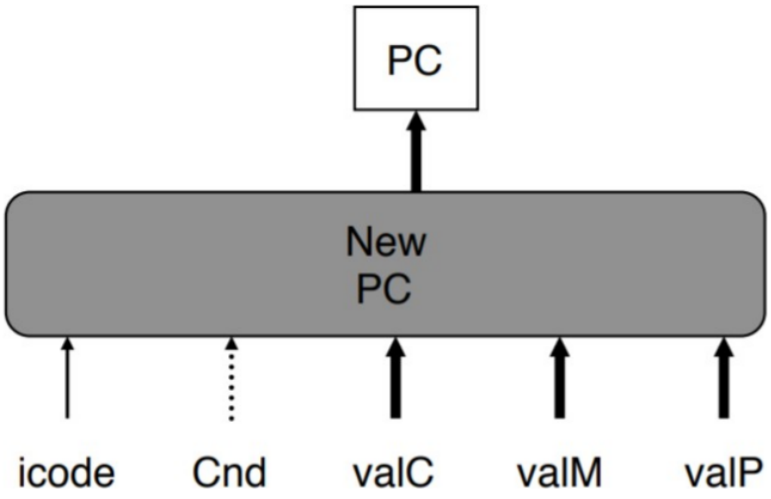
In case of rmovq, call and pushq we write to memory. Whereas, in case of mrmovq, ret and popq we read from memory.

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

```
module memory (clk,icode,valP,valA,valB,valM,valE,mem_err);
    input clk;                      //clock
    input [3:0] icode;              //instruction code
    input [63:0] valP,valA,valB,valE;   //constant, next PC, value A, va
    output reg mem_err;             //memory error
    output reg [63:0] valM;         //value M
    reg [63:0] memo_arr[0:1023];     //temporary memory for storing and
    initial
    begin
        mem_err=0;
        valM=64'b0;
        for(integer i=0;i<1024;i=i+1)
        begin
            memo_arr[i]=34;
        end
    end
  always@(*)
  begin
    //mrmovq
    if(icode == 4'b0101)
    begin
      if(valE > 1023)
        begin
            mem_err = 1;
        end
      valM = memo_arr[valE] ;
    end
    // ret
    else if(icode == 4'b1001)
    begin
      if(valA > 255)
          begin
              mem_err = 1;
          end
      valM = memo_arr[valA];
    end
    // popq
    else if(icode == 4'b1011)
    begin
      if(valE > 255)
        begin
            mem_err = 1;
        end
      valM = memo_arr[valA];
    end
    else
      begin
        mem_err =0;
      end
  end
```

```
  always @ (posedge clk) begin
    mem_err=0;
    // rmmovq
    if(icode == 4'b0100)
    begin
      if(valE > 255)
        begin
            mem_err = 1;
        end
      memo_arr[valE] <= valA;
    end
    // call
    else if(icode == 4'b1000)
    begin
      if(valE > 255)
        begin
            mem_err = 1;
        end
      memo_arr[valE] <= valP;
    end
    // pushq
    else if(icode == 4'b1010)
    begin
      if(valE > 255)
        begin
            mem_err = 1;
        end
      memo_arr[valE] <= valA;
    end
  end
endmodule
```

PC UPDATE :



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

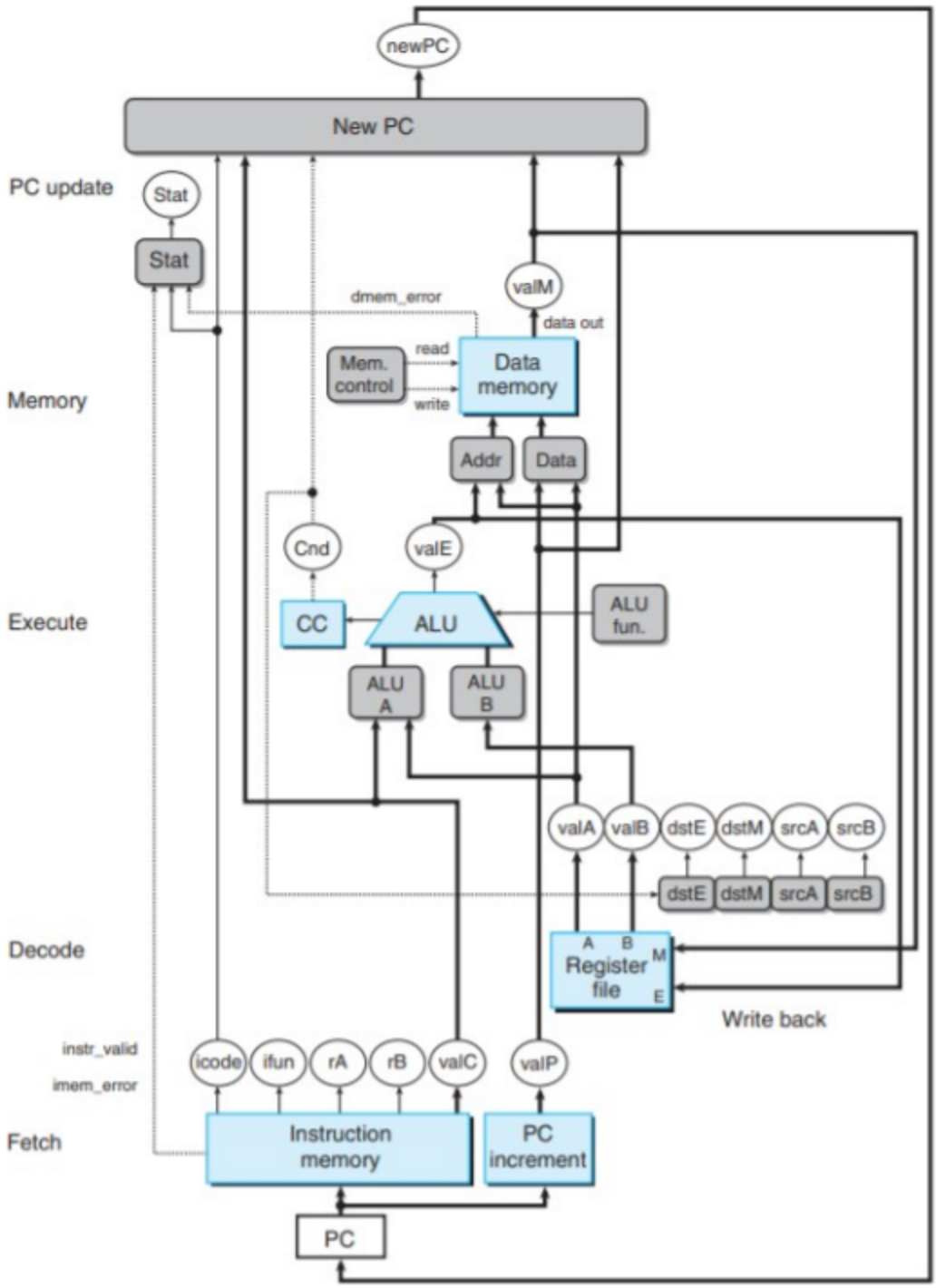
Computed Values in this stage are -

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, cnd, icode,  valC, valM, valP, PC_new);
input cnd;
input clk;
input [3:0] icode;
input [63:0] valC, valM, valP;
output reg [63:0] PC_new;
always@(*)
    begin
        case(icode)
        4'b0111:
        begin
            if(cnd)
                PC_new = valC;
            else
                PC_new = valP;
        end
        4'b1001:
        begin
            PC_new = valM;
        end
        4'b1000:
        begin
            PC_new = valC;
        end
        default:
        begin
            PC_new = valP;
        end
        endcase
    end
endmodule
```

HARDWARE IMPLEMENTATION OF SEQUENTIAL :

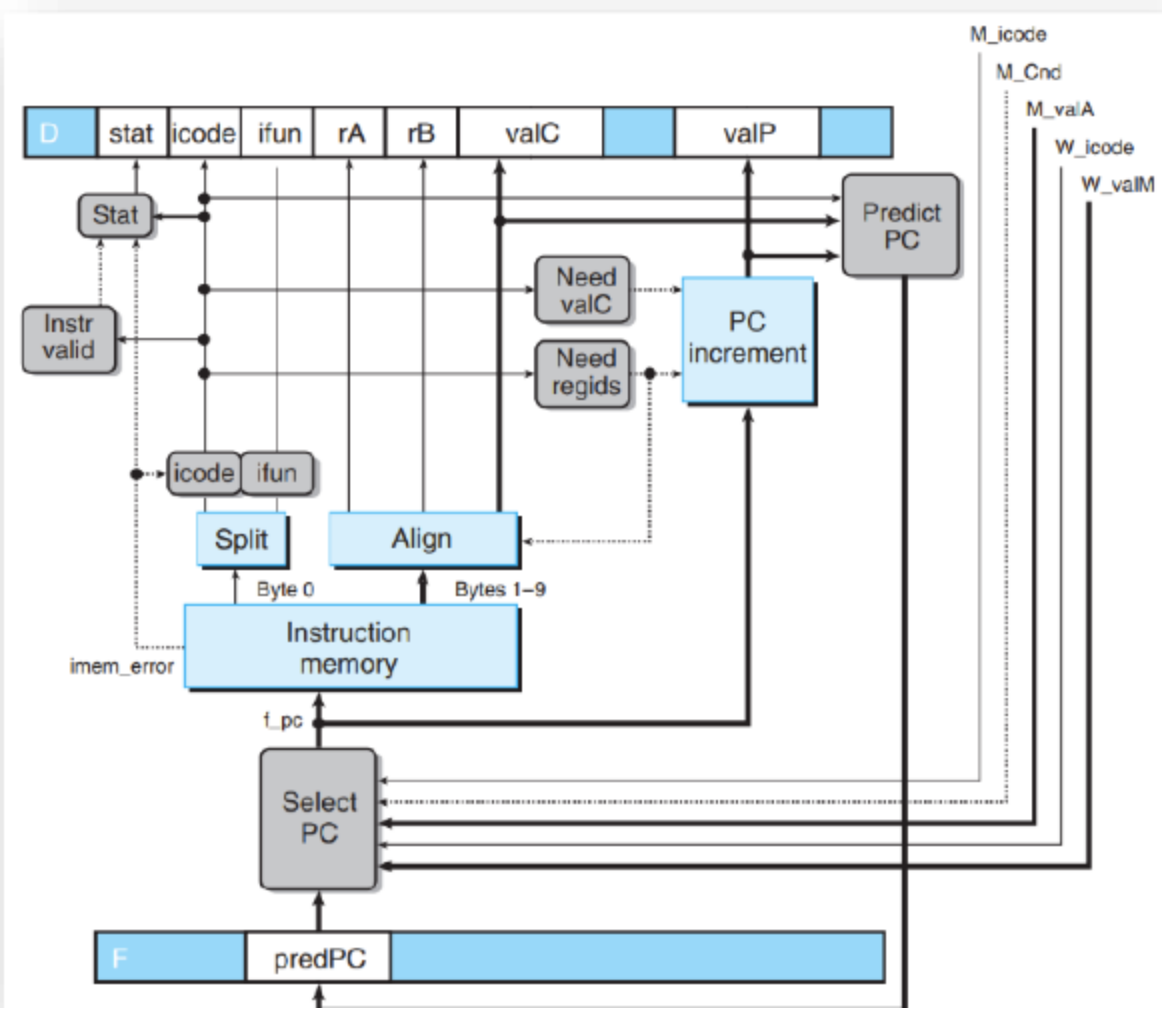


PIPELINE IMPLEMENTATION :

The processor Y86- 64's implementation uses the same modules as its sequential implementation,
with the addition of pipelined registers, a slight modification to the fetch and

decode blocks, also
addition of data forwarding and PC prediction to boost performance, and pipeline control logic to eliminate pipeline hazards.64's implementation uses the same modules as its sequential
Implementation, with the addition of pipelined registers, a slight modification to the fetch and decode blocks, also addition of data forwarding and PC prediction to boost performance, and pipeline control logic to eliminate pipeline hazards. As we want to fetch the next instruction constantly without having to wait for the PC update stage of the previous instruction to finish had
it been at the end of the cycle, we should move the PC update stage to the beginning of the cycle for the pipelined implementation. Circuit retiming is the term for this. This modifies the circuit's overall state while having no impact on its local behaviour. Additionally, it enables us to manage the delays in the pipelined system between phases. In a pipelined version, some hardware and signals in the SEQ implementation are moved around, and pipeline registers are added in between each step.

FETCH :



The starting value of PC from the processor.v block is used to create the field f pc in this case.

The values of stat, icode, ifun, rA, rB, valC, and valP are computed using the PC as the input, and since they will be sent into the decode register, they will be given the names D_icode, D_ifun, D

rA, D_rB, valC, and D_stat. The fetch portion operates in the same way as the sequential portion, but the inclusion of the sequential block boosts the processor's performance. Predict PC block capability is sent to the retrieve register in the processor.v block after the Predict PC block is added. New pc will be updated after each positive edge of the clock.

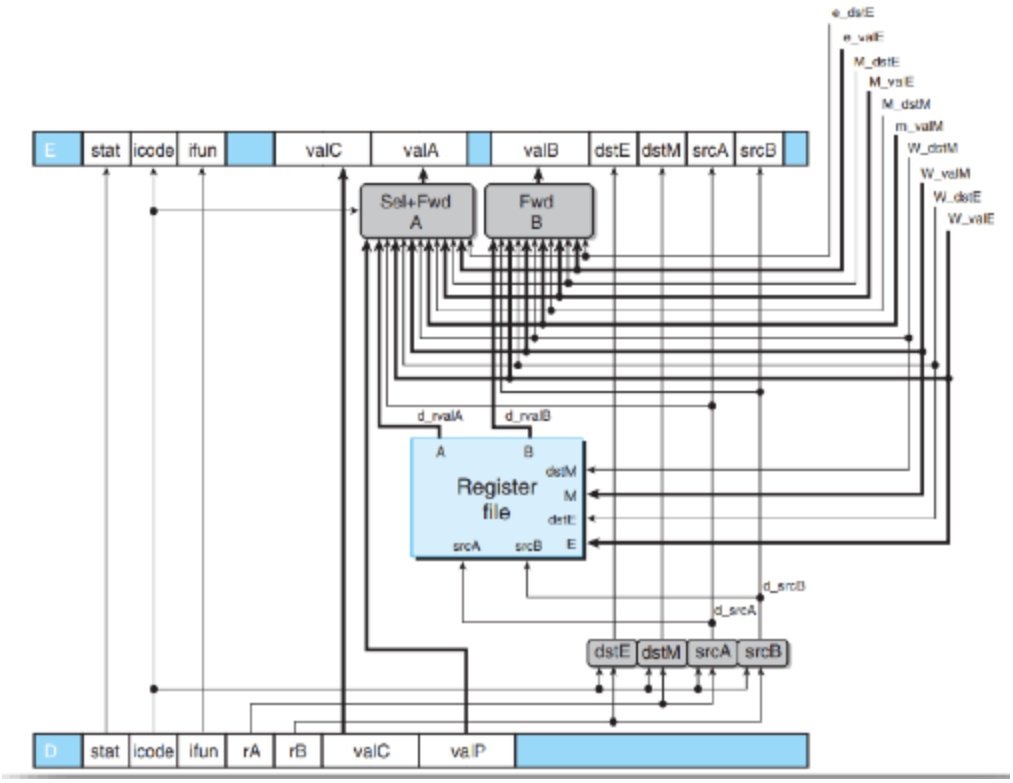
```
module
Fetch(clk,F_predPC,f_predPC,M_valA,W_valM,M_Cnd,M_icode,W_icode,F_stall,D_stall,D_bubble,D_stat,D_icode
,D_ifun,D_rA,D_rB,D_valC,D_valP,current_instruction,D_stat,PC);

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
output 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;
```

The input for this block are the clock signal and M_icode, M_cnd, M_valA, W_icode, W_valM which are present for the calculation of the next predicted PC i.e.output PC and input PC in which is used to calculate the values of the D_icode, D_ifun, D_rA, D_rB, D_valC, D stat and D valP. Once the clock's positive edge is reached, we changed the register and the output for D icode, D ifun, D rA, D rB, D valC, D stat, and D valP is made accessible as a register output.

DECODE AND WRITEBACK :



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. Along with the earlier blocks, there are two more blocks in this. These blocks, which are represented by the Sel+Fwd A and Fwd B that directly provide the value for valA or valB from the execute, memory, or writeback stages, aid in the forwarding of data.

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

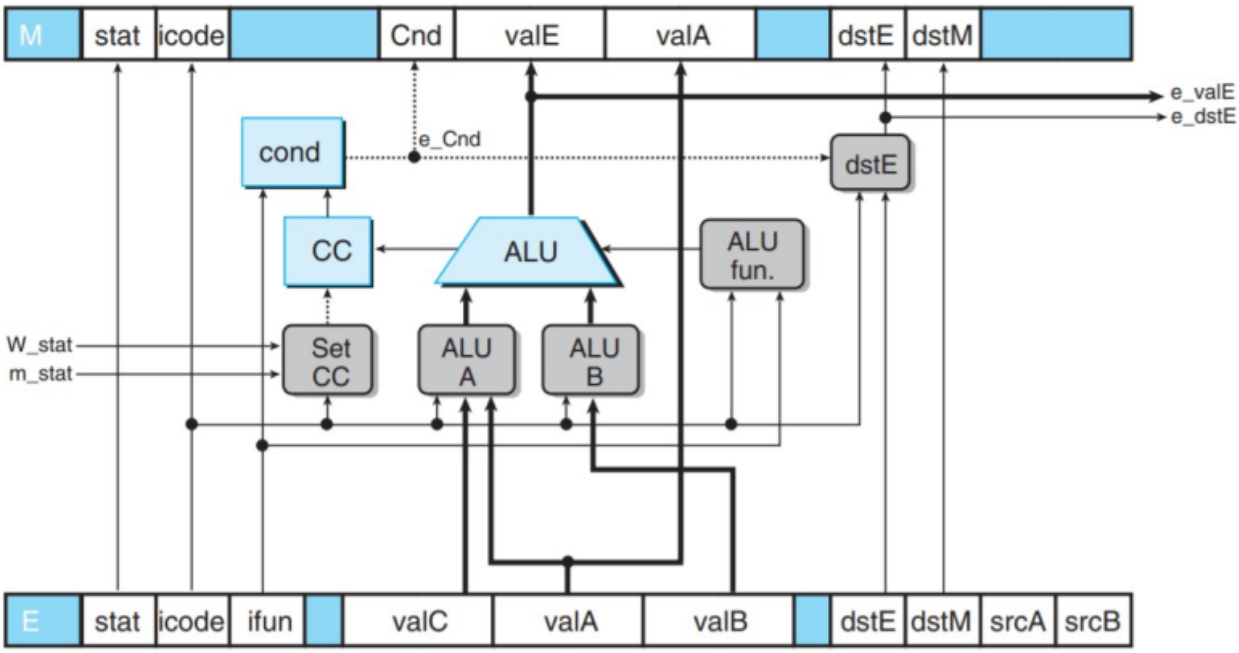
Forwarding Sources -

```
## What should be the A value?
int d_valA = [
  # Use incremented PC
     D_icode in { ICALL, IJXX } : D_valP;
  # Forward valE from execute
     d_srcA == e_dstE : e_valE;
  # Forward valM from memory
     d_srcA == M_dstM : m_valM;
  # Forward valE from memory
     d_srcA == M_dstE : M_valE;
  # Forward valM from write back  d_srcA
== W_dstM : W_valM;
  # Forward valE from write back
     d_srcA == W_dstE : W_valE;
  # Use value read from register file
     1 : d_rvalA;
];
```

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

EXECUTE ALU :

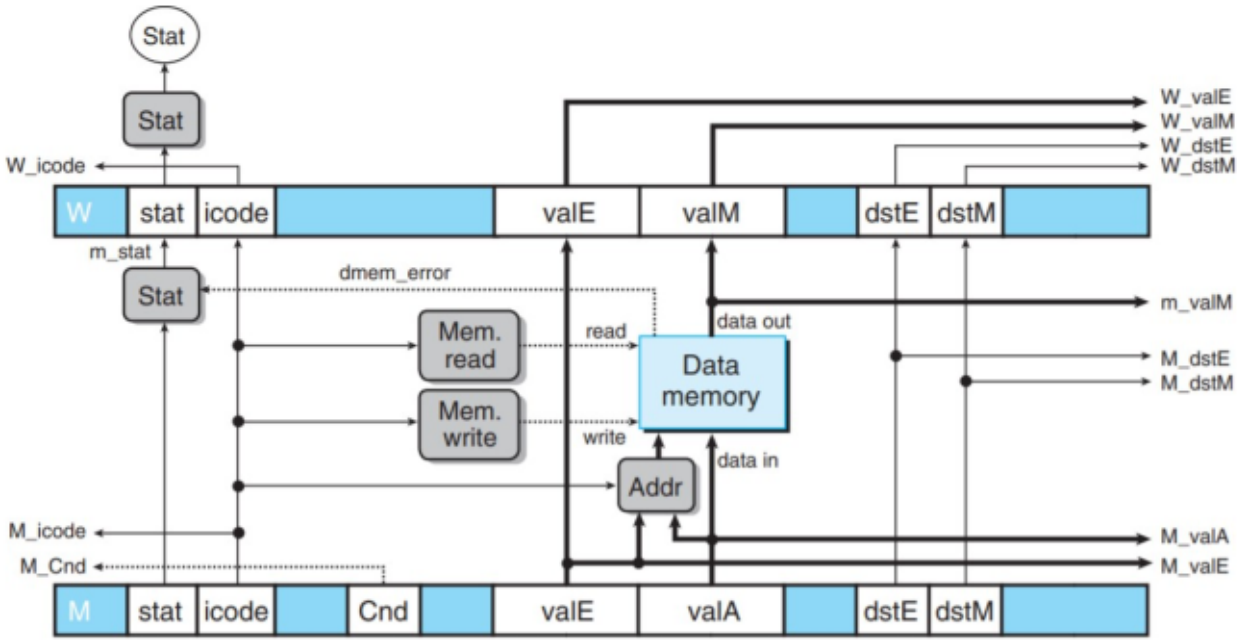
Operate ALU:



- 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

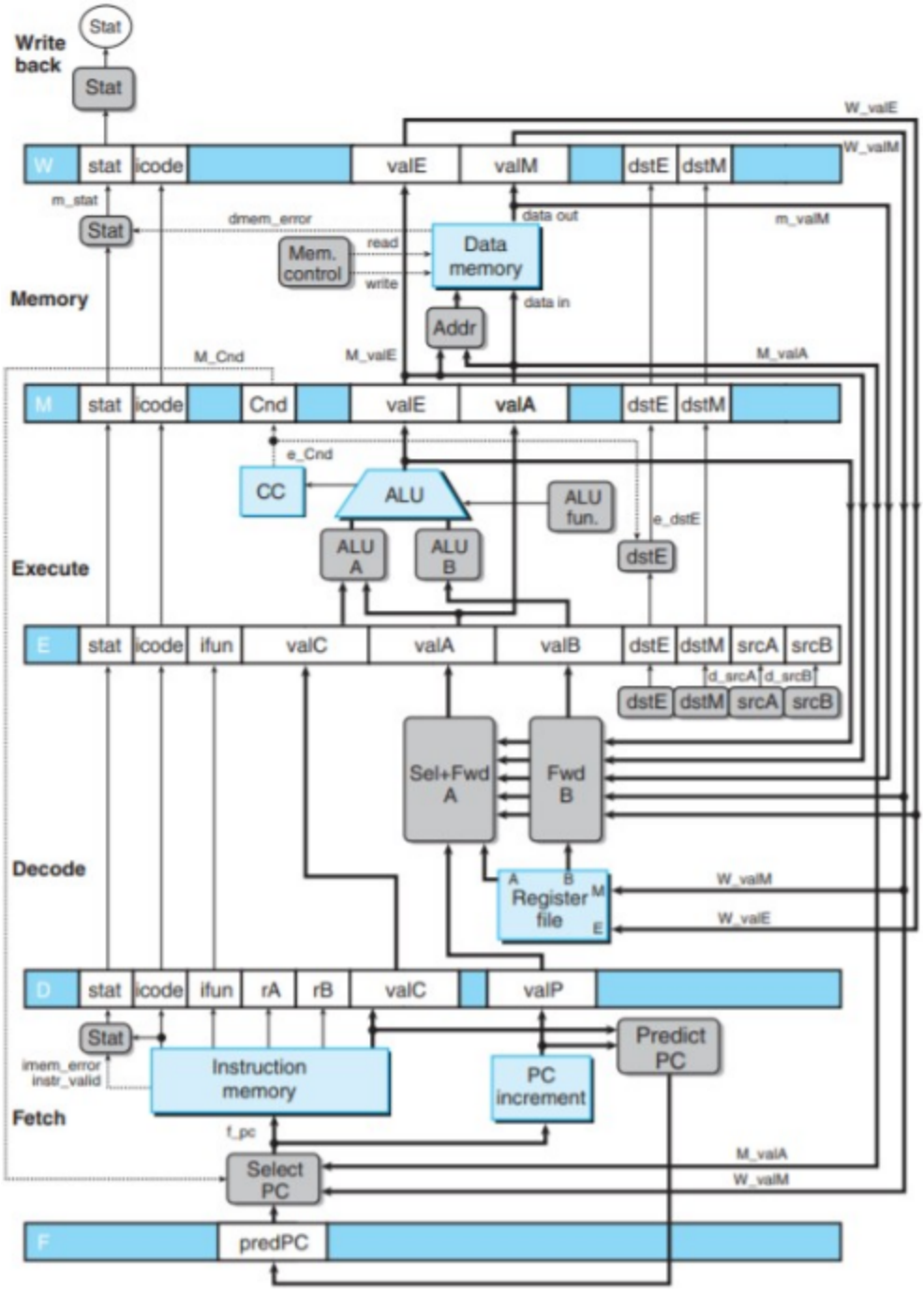
MEMORY :

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.

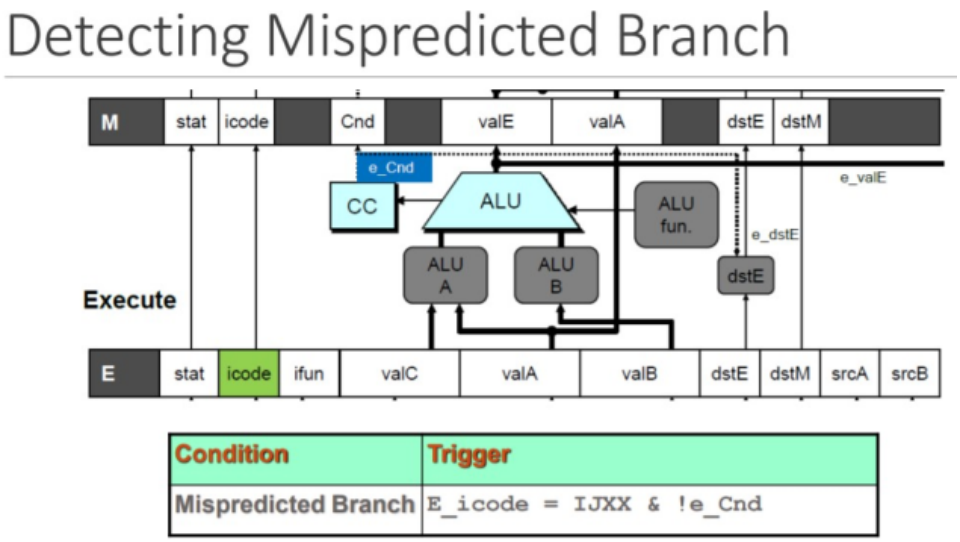
OVERALL PIPELINE IMPLEMENTATION :



Branch Misprediction Case -

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

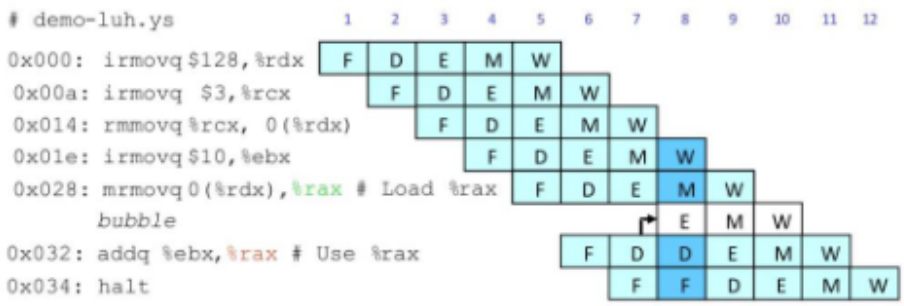
- A misprediction can incur a serious penalty causing a serious degradation of program performance



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.

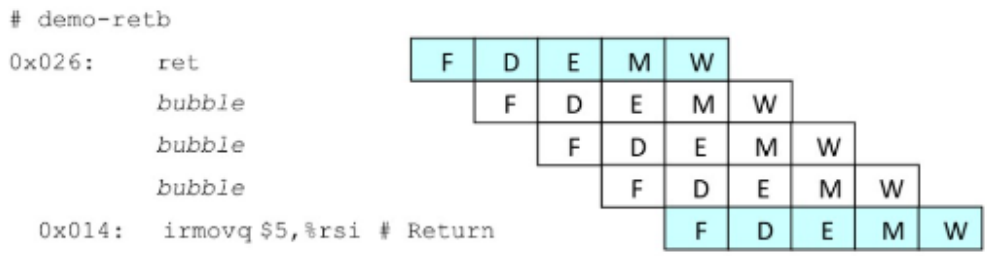
Control for Load/Use Hazard



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

Condition	F	D	E	M	W
Load/Use Hazard	stall	stall	bubble	normal	normal

Control for Return



Condition	F	D	E	M	W
Processing `ret`	stall	bubble	normal	normal	normal

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 initially to update the registers properly so that it hold correct value for teh next instruction