Y86 - ISA IMPLEMENTATION

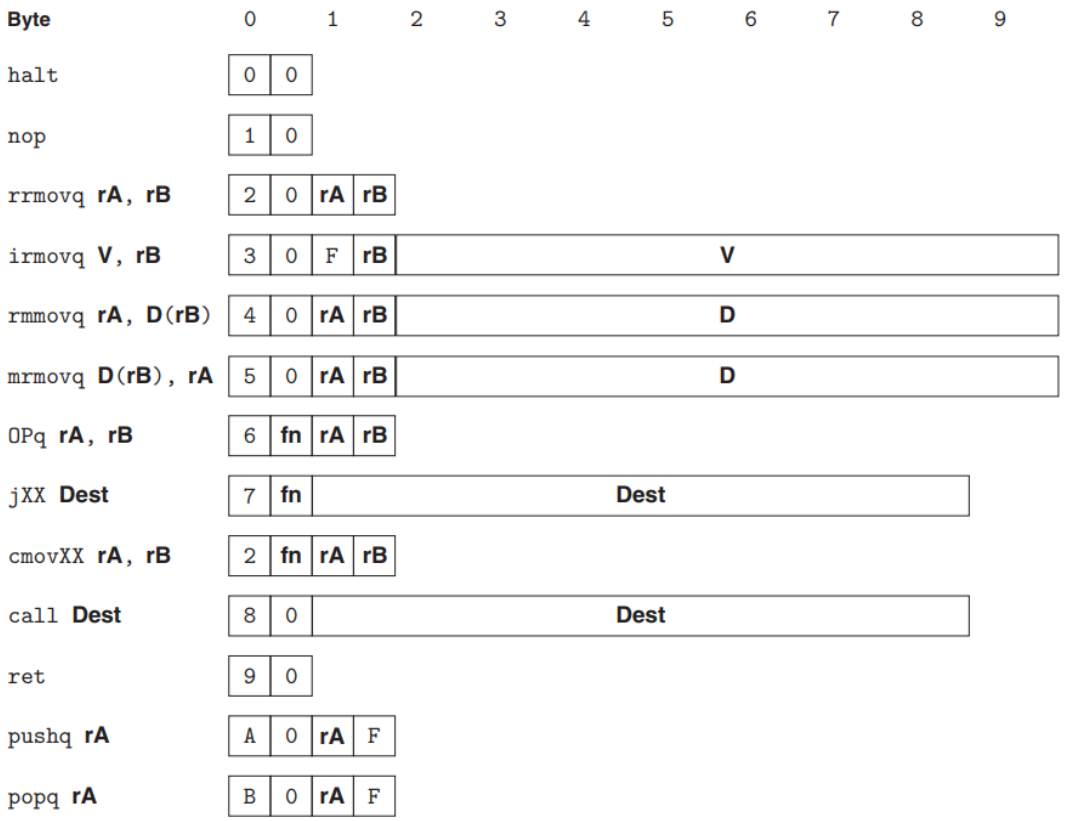
Juturu Dhanush 2022102006 M.V.R.Jeevesh 2022102009

https://www.notion.so/Y86-ISA-IMPLEMENTATION-4f913cc778e5471997f15b5dc520cc9e?pvs=4

objective:

develop a processor architecture design based on the Y86 ISA using Verilog. The design to be thoroughly tested to satisfy all the specification requirements using simulations.The processor should be implemented in both sequential and pipelined manner. The design approach is modular.

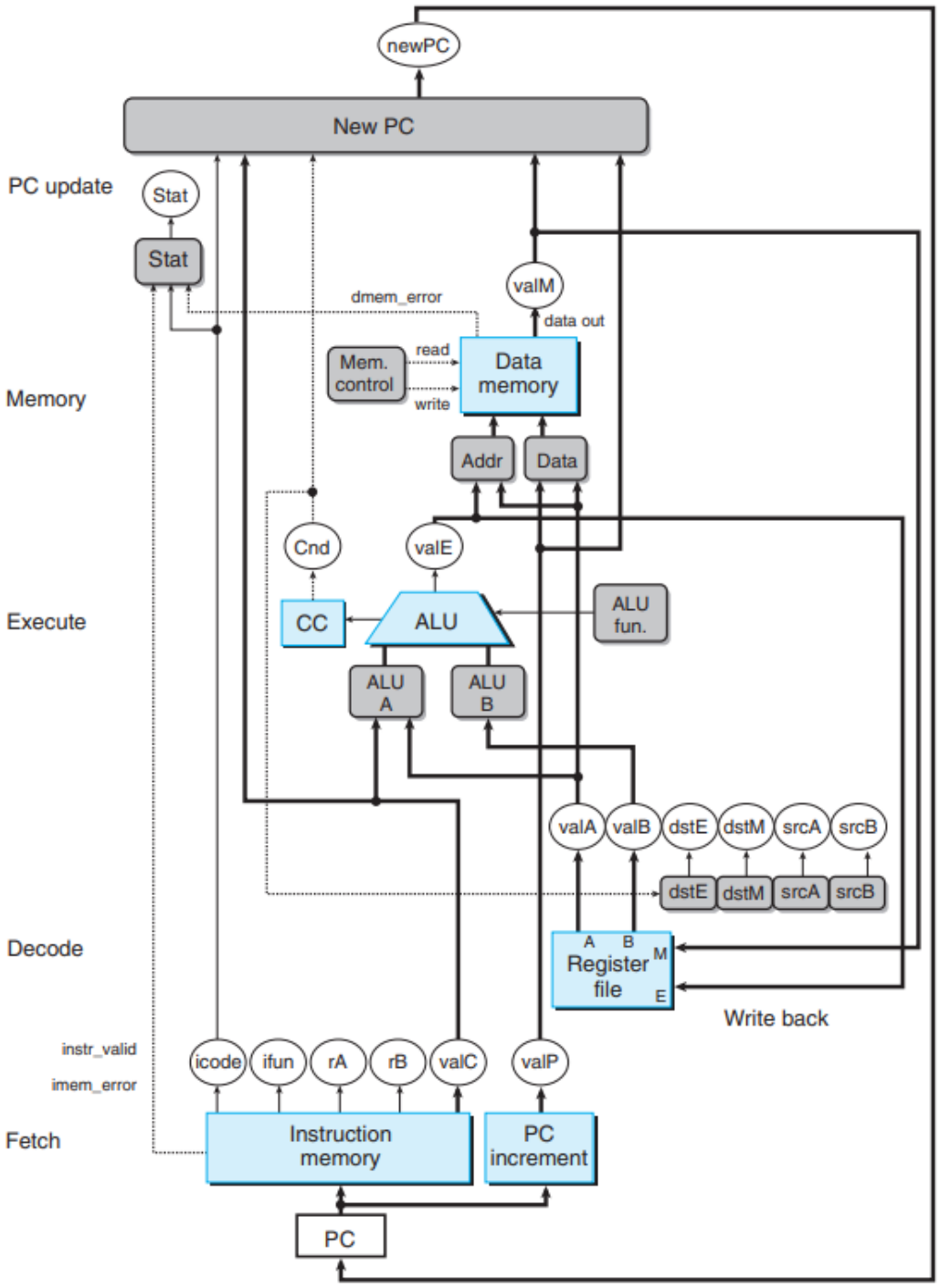
supported instructions:



sequential implementation of Y86 64 bit processor

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 –

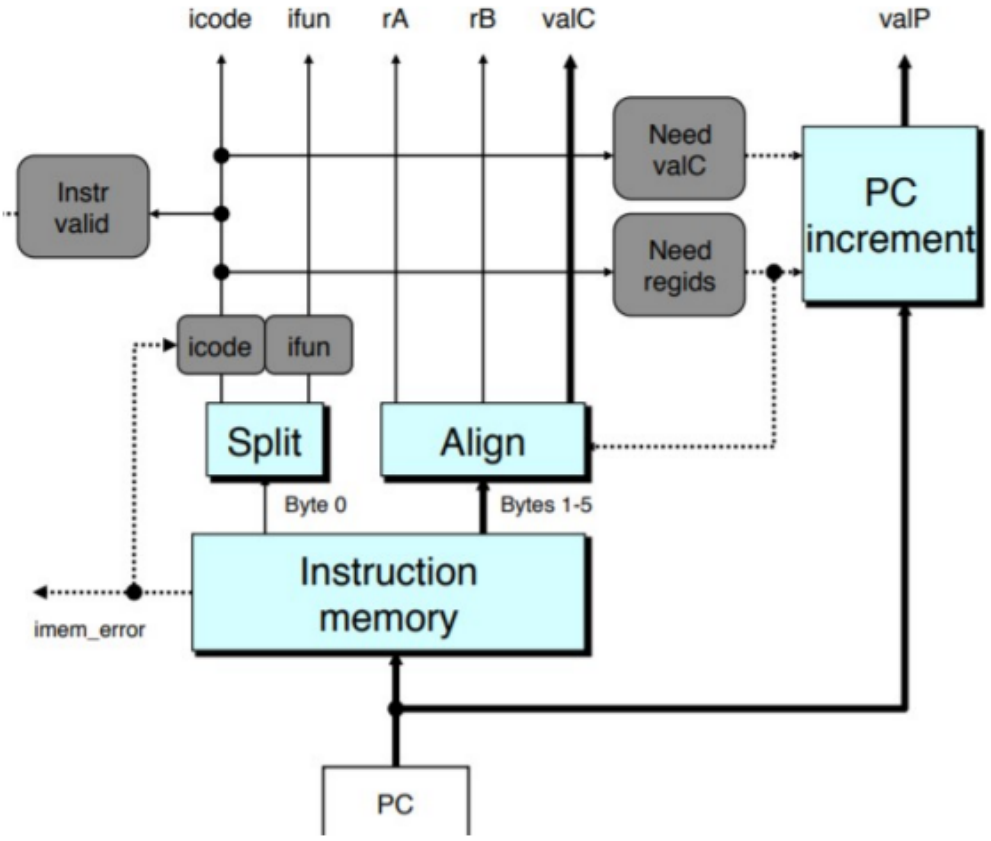
1. FETCH → read instruction from instruction memory.
2. DECODE → read program registers.
3. EXECUTE → compute the adress and values using ALU.
4. MEMORY → read and write back data into memory.
5. WRITE BACK →write program registers.
6. PC UPDATE →Update the program counter.


newPC
New PC
PC update
Stat
Stat
dmem_error
valM
data out
Memory
Mem. control
read
write
Data memory
Addr
Data
Cnd
valE
Execute
CC
ALU
ALU fun.
ALU A
ALU B
valA
valB
dstE
dstM
srcA
srcB
dstE
dstM
srcA
srcB
Decode
A
B
Register file
M
E
Write back
instr_valid
imem_error
icode
ifun
rA
rB
valC
valP
Fetch
Instruction memory
PC increment
PC

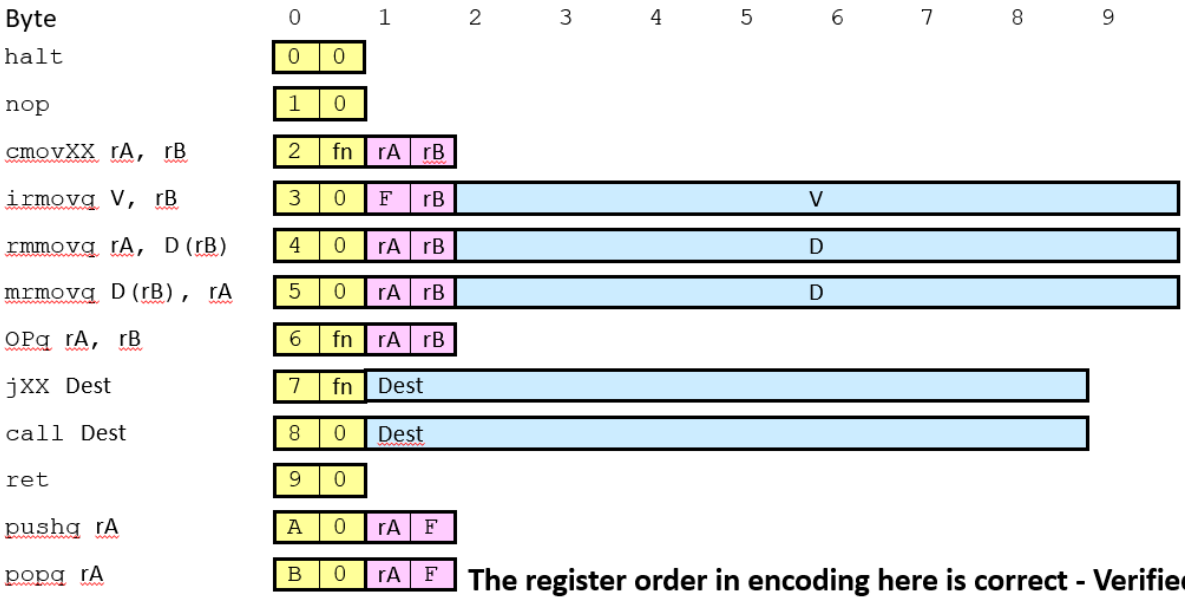
Stage	Computation	`OPq` rA, rB	`mrmovq` D(rB), rA
Fetch	icode, ifun	icode:ifun $\leftarrow$ M_1[PC]	icode:ifun $\leftarrow$ M_1[PC]
	rA, rB	rA:rB $\leftarrow$ M_1[PC + 1]	rA:rB $\leftarrow$ M_1[PC + 1]
	valC		valC $\leftarrow$ M_8[PC + 2]
	valP	valP $\leftarrow$ PC + 2	valP $\leftarrow$ PC + 10
Decode	valA, srcA	valA $\leftarrow$ R[rA]	
	valB, srcB	valB $\leftarrow$ R[rB]	valB $\leftarrow$ R[rB]
Execute	valE	valE $\leftarrow$ valB OP valA	valE $\leftarrow$ valB + valC
	Cond. codes	Set CC	
Memory	Read/write		valM $\leftarrow$ M_8[valE]
Write back	E port, dstE	R[rB] $\leftarrow$ valE	
	M port, dstM		R[rA] $\leftarrow$ valM
PC update	PC	PC $\leftarrow$ valP	PC $\leftarrow$ valP

the above fig shows the process in each stage of the processor for an MRMOVq operation.

Fetch



Fetch 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



in sequential implementation we implemented fetch module like this

```
module fetch (
    input clk,
    input [63:0] PC,
    input [0:79] instr,

    output reg [3:0] icode,
    output reg [3:0] ifun,
    output reg [3:0] ra,
    output reg [3:0] rb,
    output reg signed [63:0] valC,
    output reg [63:0] valP,
    output reg imem_error = 1'b0,
    output reg instr_invalid = 1'b0,
    output reg HLT=1'b0
);
    reg need_reg;
    reg need_valc;

    initial begin
    need_reg=1'b0;
    need_valc=1'b0;
    end
    always @(*) begin
```

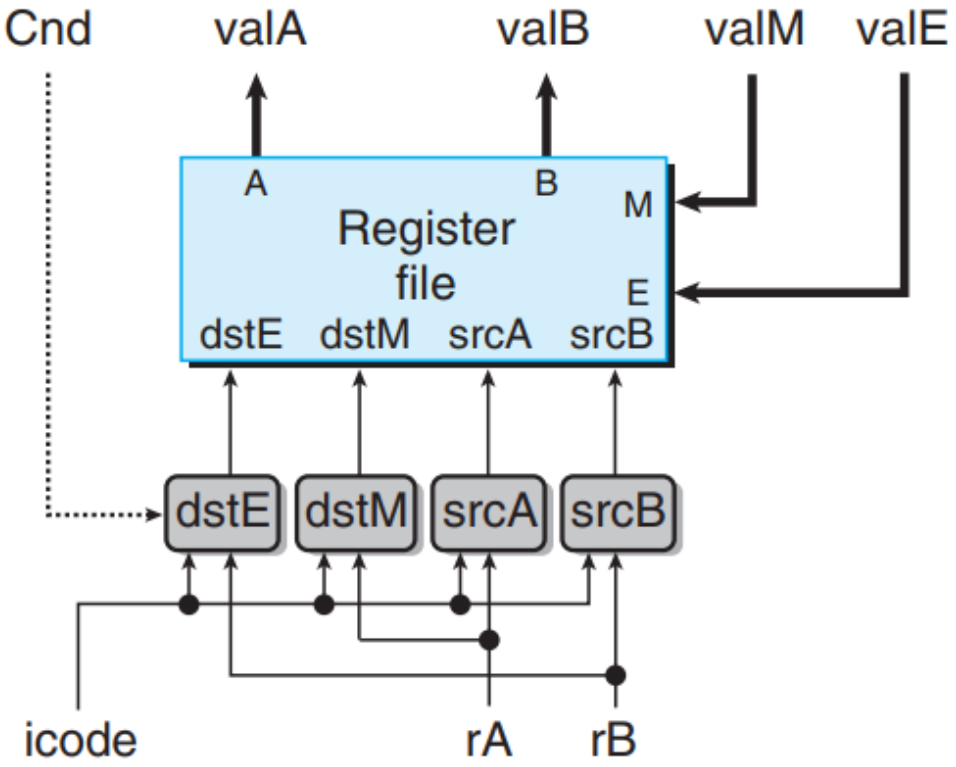
```
        if(PC>64'd1023)begin
            imem_error=1'b1;
        end
        else begin
            imem_error=1'b0;
        end
    end
    always @(*) begin
    icode = instr[0:3];
    ifun = instr[4:7];
        case (icode)
        4'b0000: begin                      // Halt
            need_reg <=1'b0;
            need_valc <=1'b0;
            HLT=1'b1;
            instr_invalid = 1'b0;
        end
        4'b0001:  begin                    // NOP
            need_reg <=1'b0;
            need_valc <=1'b0;
            HLT=1'b0;instr_invalid = 1'b0;
        end
        4'b0010: begin                    // cmov
            need_reg=1'b1;
            need_valc=1'b0;
            HLT=1'b0;instr_invalid = 1'b0;
        end
        4'b0011:  begin                    // irmov
            need_reg=1'b1;
            need_valc=1'b1;
            HLT=1'b0;instr_invalid = 1'b0;
        end
        4'b0100:  begin                    // rmmov
            need_reg=1'b1;
            need_valc=1'b1;
            HLT=1'b0;instr_invalid = 1'b0;
        end
        4'b0101:  begin                    //mrmov
```

```
            need_reg=1'b1;
            need_valc=1'b1;
            HLT=1'b0;instr_invalid = 1'b0;
        end
        4'b0110:  begin                    // opq
            need_reg=1'b1;
            need_valc=1'b0;
            HLT=1'b0;instr_invalid = 1'b0;
        end
        4'b0111:  begin                   // jxx
            need_reg=1'b0;
            need_valc=1'b1;
            HLT=1'b0;instr_invalid = 1'b0;
        end
        4'b1000:  begin                   // call
            need_reg=1'b0;
            need_valc=1'b1;
            HLT=1'b0;instr_invalid = 1'b0;
        end
        4'b1001:  begin                   // ret
            need_reg=1'b0;
            need_valc=1'b0;
            HLT=1'b0;instr_invalid = 1'b0;
        end
        4'b1010:  begin                   // push
            need_reg=1'b1;
            need_valc=1'b0;
            HLT=1'b0;instr_invalid = 1'b0;
        end
        4'b1011:  begin                   // pop
            need_reg=1'b1;
            need_valc=1'b0;
            HLT=1'b0;instr_invalid = 1'b0;
        end
        default: instr_invalid = 1'b0;        // Default case
    endcase
    if(need_reg == 1'b1)begin
        ra = instr[8:11];
```

```
        rb = instr[12:15];
    end
    else begin
      ra=4'b1111;
      rb=4'b1111;
    end
    if(need_valc == 1'b1 && need_reg == 1'b0)begin
        valC = instr[8:71];
    end
    else if(need_valc == 1'b1 && need_reg == 1'b1)begin
        valC=instr[16:79];
    end
    else begin
        valC=64'd0;
    end
    valP = PC + 1 + need_reg + 8*need_valc;
    end
endmodule
```

DECODE AND WRITEBACK

As both decode and write back stages asceses the program registers we implement both of them in same block



The instruction fields are decoded to generate register identifiers for four addresses (two read and two write) used by the register file. The values read from the register file become the signals valA and valB. The two write-back values valE and valM serve as the data for the writes.

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_writeback (
    input clk,
    input [3:0] icode,
    input [3:0] rA,
    input [3:0] rB,
    input cnd,
    output reg [63:0] valA,
    output reg [63:0] valB,
    input [63:0] valE,
    input [63:0] valM,

    output reg [63:0] rax,rbx,rcx,rdx,rsp,rbp,rsi,rbi,r8,r9,r
);

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

```
initial begin
  reg_file[0]  =  64'd0;
  reg_file[1]  =  64'd0;
  reg_file[2]  =  64'd0;
  reg_file[3]  =  64'd0;
  reg_file[4]  =  64'd50;
  reg_file[5]  =  64'd0;
  reg_file[6]  =  64'd0;
  reg_file[7]  =  64'd0;
  reg_file[8]  =  64'd0;
  reg_file[9]  =  64'd0;
  reg_file[10] =  64'd0;
  reg_file[11] =  64'd0;
  reg_file[12] =  64'd0;
  reg_file[13] =  64'd0;
  reg_file[14] =  64'd0;
end

//Decode
always @(*) begin
      if (icode == 4'b0000) begin // halt
      end
      else if (icode == 4'b0001) begin // nop
      end
      else if (icode == 4'b0010) begin // cmovXX
          valA = reg_file[rA];
          valB = 64'd0;
      end
      else if (icode == 4'b0011) begin // irmov
      end
      else if (icode == 4'b0100) begin // rmmovq
          valA = reg_file[rA];
          valB = reg_file[rB];
      end
      else if (icode == 4'b0101) begin // mrmovq
          valB = reg_file[rB];
      end
      else if (icode == 4'b0110) begin // opq
```

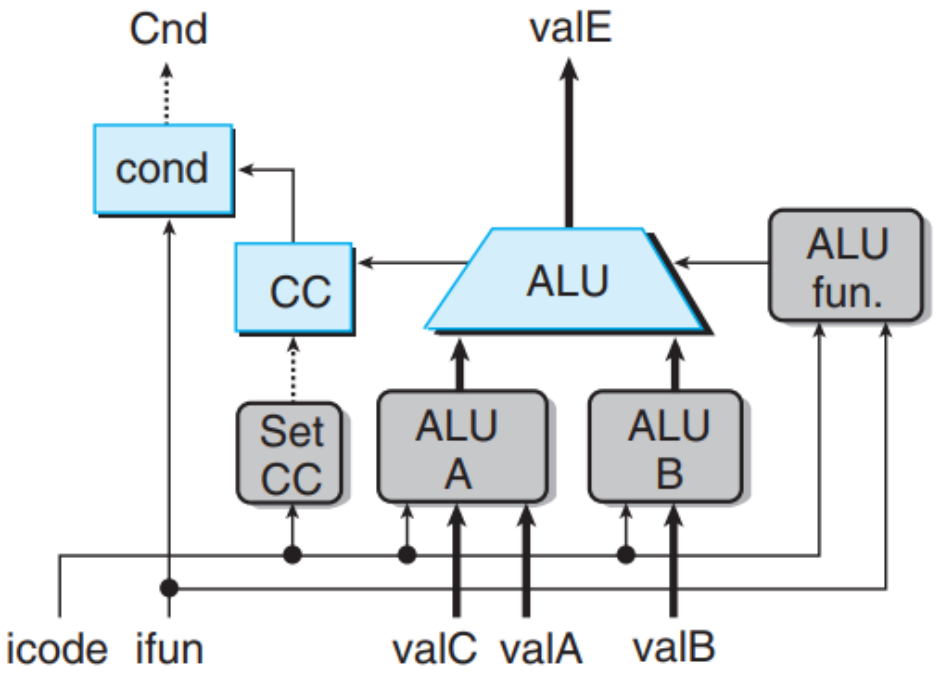
```
            valA = reg_file[rA];
            valB = reg_file[rB];
        end
        else if (icode == 4'b0111) begin // jxx
        end
        else if (icode == 4'b1000) begin // call
            valB = reg_file[4];
        end
        else if (icode == 4'b1001) begin // ret
            valA = reg_file[4];
            valB = reg_file[4];
        end
        else if (icode == 4'b1010) begin // pushq
            valA = reg_file[rA];
            valB = reg_file[4];
        end
        else if (icode == 4'b1011) begin // popq
            valA = reg_file[4];
            valB = reg_file[4];
        end
end
//Write Back
always @(posedge clk) begin
        if (icode == 4'b0000) begin // halt
        end
        else if (icode == 4'b0001) begin // nop
        end
        else if (icode == 4'b0010) begin // cmovXX
          if(cnd ==1'b1)begin
            reg_file[rB] = valE;
          end
        end
        else if (icode == 4'b0011) begin // irmov
            reg_file[rB] = valE;
        end
        else if (icode == 4'b0100) begin // rmmovq
        end
        else if (icode == 4'b0101) begin // mrmovq
```

```
            reg_file[rA] = valM;
        end
        else if (icode == 4'b0110) begin // opq
            reg_file[rB] = valE;
        end
        else if (icode == 4'b0111) begin // jxx
        if(cnd ==1'b1)begin
            reg_file[rB] = valE;
          end
        end
        else if (icode == 4'b1000) begin // call
            reg_file[4] = valE;
        end
        else if (icode == 4'b1001) begin // ret
            reg_file[4] = valE;
        end
        else if (icode == 4'b1010) begin // pushq
            reg_file[4] = valE;
        end
        else if (icode == 4'b1011) begin // popq
            reg_file[4] = valE;
            reg_file[rA] = valM;
        end

        rax = reg_file[0];
        rcx = reg_file[1];
        rdx = reg_file[2];
        rbx = reg_file[3];
        rsp = reg_file[4];
        rbp = reg_file[5];
        rsi = reg_file[6];
        rbi = reg_file[7];
        r8  = reg_file[8];
        r9  = reg_file[9];
        r10 = reg_file[10];
        r11 = reg_file[11];
        r12 = reg_file[12];
        r13 = reg_file[13];
```

```
        r14 = reg_file[14];
end

endmodule
```

EXECUTE:

The execute stage includes the arithmetic/logic unit (ALU). This unit performs the operation add, subtract, and, or exclusive-or on inputs aluA and aluB based on the setting of the alufun signal. These data and control signals are generated by three control blocks.The ALU output becomes
the signal valE.



The ALU either performs the operation for an integer operation instruction or acts as an adder. The
condition code registers are set according to the ALU value. The condition code values are tested
to determine whether a branch should be taken.

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.

- 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 <=)

Conditional Branches with Conditional Move

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 <=)

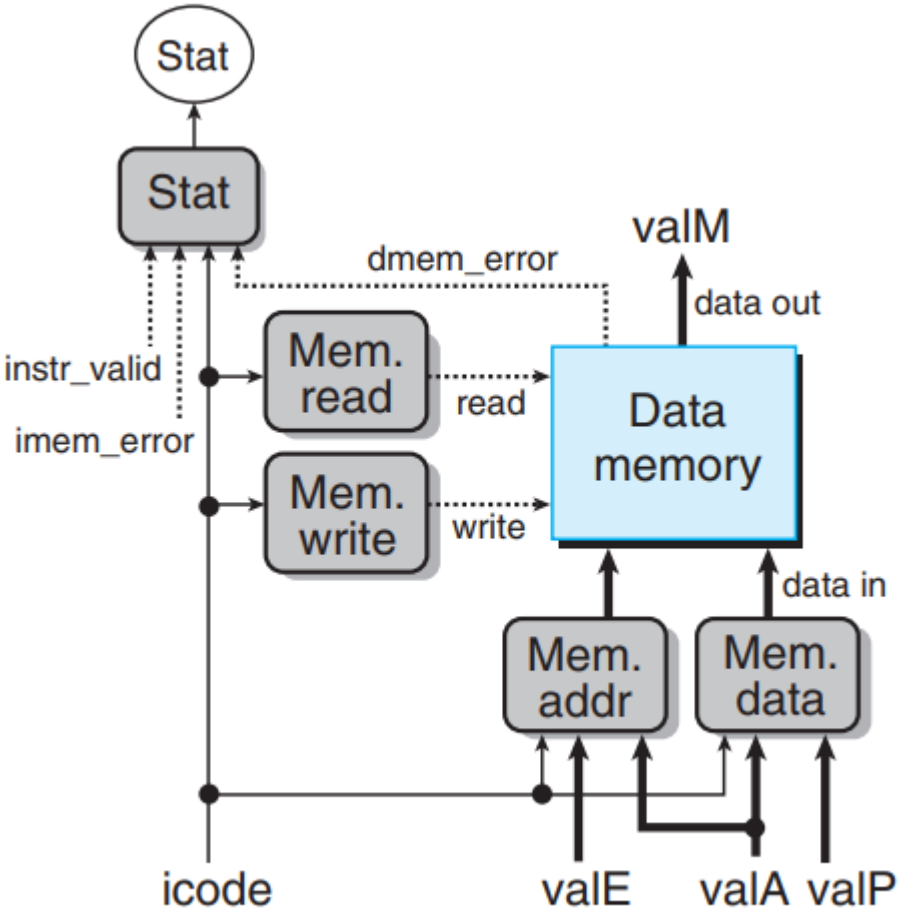
```
`include "../ALU/ALU.v"
module execute (
  input clk,
  input [3:0] icode,
  input [3:0] ifun,
  input [63:0] valA,valB,valC,
  output reg [63:0] valE,
  output reg cnd,
```

```
  output reg cf,
  output reg zf ,
  output reg sf ,
  output reg of
);

  reg [1:0]control;
  reg signed [63:0]ALUa;
  reg signed [63:0]ALUb;
  wire signed [63:0]Alu_out;
  wire overflow;
reg  condition[6:0];

ALU ALU(ALUa,ALUb,control,Alu_out,overflow);

 always@(*)
  begin
        cf = 1'b0;
        zf = (Alu_out == 1'b0);
        sf = (Alu_out<1'b0);
        of = (ALUa<1'b0 == ALUb<1'b0)&&(Alu_out<1'b0 != ALUa<

        condition[0]  =    1'b1;                      //
        condition[1]  =    (sf^of)|zf ;               // le
        condition[2]  =    (sf^of);                   // l
        condition[3]  =    zf;                        // e
        condition[4]  =    ~zf;                       // ne
        condition[5]  =    ~(sf^of);                  // ge
        condition[6]  =    (~(sf^of))&(~zf);          // g
  end
   initial begin
    cf = 1'b0;
    zf = 1'b0;
    sf = 1'b0;
    of = 1'b0;
   end
  always@(*)begin
```

```
        case (icode)
        4'b0010: begin                    // cmov
            cnd = condition[ifun];
            valE = valA+64'd0;
        end
        4'b0011:  begin                    // irmov
            valE = 64'd0 + valC;
        end
        4'b0100:  begin                    // rmmov
            valE = valB + valC;
        end
        4'b0101:  begin                    // mrmov
           valE = valB + valC;
        end
        4'b0110:  begin                 // opq
            control = ifun[1:0];
            ALUa = valA;
            ALUb = valB;
            valE = Alu_out;
        end
        4'b0111:  begin                    // jxx
            cnd = condition[ifun];
        end
        4'b1000:  begin                    // call
            valE = -64'd8+valB;
        end
        4'b1001:  begin                    // ret
            valE = 64'd8+valB;
        end
        4'b1010:  begin                    // push
            valE = -64'd8+valB;
        end
        4'b1011:  begin                    // pop
            valE = 64'd8+valB;
        end
        default:;        // Default case
    endcase
  end
```

```
endmodule
```

MEMORY STAGE



The data memory can either write or read memory values. The value read from memory forms the signal valM.

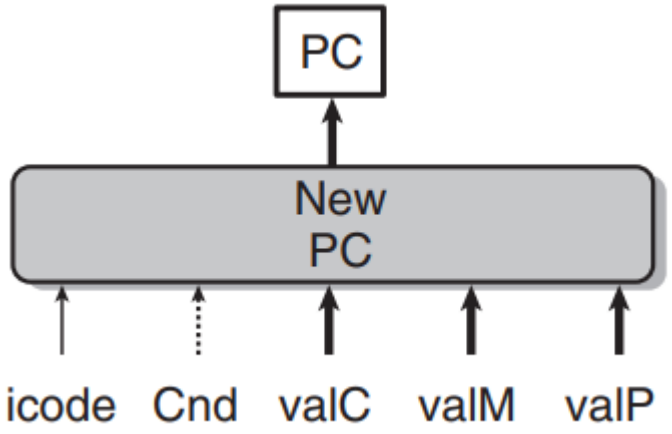
implementation of memory stage:

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

```
module memory(
    input [63:0] data,
    input [63:0] addr,
    input we,
    input re,
    input clk,
    output reg [63:0] q ,
    output reg dmem_error
);
always @(*) begin
    if(addr>1023)begin
                dmem_error=1'b1;
            end
            else begin
                dmem_error=1'b0;
            end
end
    reg [63:0] ram [1023:0];
    always @(clk==0) begin
        if (we)
            ram[addr] = data;
    end
    always @(clk==0) begin
        if (re)
            q = ram[addr];
        else
            q = 64'bx;
```

```
    end

endmodule
```

PC UPDATE STAGE:



The next value of the PC is selected from among the signals valC, valM, and valP, depending on the
instruction code and the branch flag.

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 (
    input clk,
    input [3:0] icode,
    input cnd,
```

```
    input [63:0] valC, valM, valP,
    output reg [63:0] PC_new
);
always @(posedge clk) begin
    case (icode)
        4'b0000: PC_new = 64'b0;         // Halt
        4'b0001: PC_new = valP;          // NOP
        4'b0010: PC_new = valP;          // CMOVXX
        4'b0011: PC_new = valP;          // IRMOVQ
        4'b0100: PC_new = valP;          // RMMOVQ
        4'b0101: PC_new = valP;          // MRMOVQ
        4'b0110: PC_new = valP;          // OPQ
        4'b0111: PC_new = (cnd == 1) ? valC : valP; // JXX
        4'b1000: PC_new = valC;          // CALL
        4'b1001: PC_new = valM;          // RET
        4'b1010: PC_new = valP;          // PUSHQ
        4'b1011: PC_new = valP;          // POPQ
        default: PC_new = valP;        // Default case
    endcase
end
endmodule
```

THE PROCESSOR IN SEQUENTIAL IMPLEMENTATION

```
`include "fetch.v"
`include "decode_writeback.v"
`include "execute.v"
`include "memory.v"
`include "pc_upd.v"
module seq (
    input clk,
    input [63:0] PC,
    input [0:79] instr,
    output [63:0] new_PC,
    output [63:0] rax,rbx,rcx,rdx,rsp,rbp,rsi,rbi,r8,r9,r10,r
    output reg [3:0]stat
);
```

```
    wire [3:0] icode,ifun,ra,rb;
    wire [63:0] valA,valB,valC,valE,valP,valM;
    wire imem_error,instr_invalid,HLT;
    always@(*) begin
    //   $display("\nFetch\nicode=%0h,ifun=%0h,ra=%0h,rb=%0h,
      if (instr_invalid==1'b1)begin
        assign stat=4'd3;
        $display("Invalid Instruction");
        $finish;
      end
      else if(HLT==1'b1) begin
        assign stat=4'd4;
        $display("Halt");
        $finish;
      end
      else if(imem_error==1'b1 || dmem_error==1'b1)begin
        assign stat=4'd2;
        $display("Invalid Address");
        $finish;
      end
      else begin
        assign stat=4'd1;
      end
    end
    fetch u1(clk,PC,instr,icode,ifun,ra,rb,valC,valP,imem_err
    wire cnd;
    decode_writeback u2(clk,icode,ra,rb,cnd,valA,valB,valE,va

    wire cf,zf,sf,of;
    execute u3 (clk,icode,ifun,valA,valB,valC,valE,cnd,cf,zf,

    reg [63:0]data,addr;
    wire [63:0] data_out;
    reg we,re;
    wire dmem_error;
    always@(negedge clk)begin
    case (icode)
        4'b0100:  begin                 // rmmov
```

```
            we=1;
            re=0;
            addr=valE;
            data=valA;
        end
        4'b0101:  begin                    // mrmov
           re=1;
           we=0;
           addr=valE;
           data=valM;
        end
        4'b1000:  begin                    // call
            we=1;
            re=0;
            addr=valE;
            data=valP;
        end
        4'b1001:  begin                    // ret
           re=1;
           we=0;
           addr=valE-64'd8;
           data=valM;
        end
        4'b1010:  begin                    // push
            we=1;
            re=0;
            addr=valE;
            data=valA;
        end
        4'b1011:  begin                    // pop
           re=1;
           we=0;
           addr=valE;
           data=valM;
        end
        default:;        // Default case
    endcase
    end
```

```
    memory u4(data,addr,we,re,clk,valM,dmem_error);
    pc_update u5(clk,icode,cnd,valC,valM,valP,new_PC);


endmodule
```

TEST BENCH

```
`include "seq.v"
module seq_tb();

reg clk;
reg [79:0] instr;
reg [63:0] PC;
wire [63:0] new_PC;
wire [63:0] rax,rbx,rcx,rdx,rsp,rbp,rsi,rbi,r8,r9,r10,r11,r12
wire [3:0] stat;
seq proc(clk,PC,instr,new_PC,rax,rbx,rcx,rdx,rsp,rbp,rsi,rbi,
reg [0:7] instr_mem[74:0];
always begin
    #5 clk = ~clk;
end
always @(PC) begin

    instr={instr_mem[PC],instr_mem[PC+1],instr_mem[PC+2],inst
 end
 always@(*)begin
    PC=new_PC;
 end
initial begin
    clk = 1'b0;
    PC=16'h0;
    $monitor("\n\nAt T = %4t,clk = %d, \n\ninstr = %b,\t\t\t|

        instr_mem[0] = 8'b0011_0000;//irmovq
        instr_mem[1] = 8'b1111_0011;//rbx
        instr_mem[2] = 8'b0000_0000;//val
        instr_mem[3] = 8'b0000_0000;
```

```
        instr_mem[4] = 8'b0000_0000;
        instr_mem[5] = 8'b0000_0000;
        instr_mem[6] = 8'b0000_0000;
        instr_mem[7] = 8'b0000_0000;
        instr_mem[8] = 8'b0000_0000;
        instr_mem[9] = 8'b0000_1000;

        instr_mem[10] = 8'b0011_0000;//irmovq
        instr_mem[11] = 8'b1111_0010;//rdx
        instr_mem[12] = 8'b0000_0000;//val
        instr_mem[13] = 8'b0000_0000;
        instr_mem[14] = 8'b0000_0000;
        instr_mem[15] = 8'b0000_0000;
        instr_mem[16] = 8'b0000_0000;
        instr_mem[17] = 8'b0000_0000;
        instr_mem[18] = 8'b0000_0000;
        instr_mem[19] = 8'b0000_0010;

        instr_mem[20] = 8'b0110_0000;//addq
        instr_mem[21] = 8'b0010_0011;//rdx , rbx

        instr_mem[22] = 8'b0100_0000;               //rmmovq
        instr_mem[23] = 8'b0010_0000;//rdx
        instr_mem[24] = 8'b0000_0000;//val
        instr_mem[25] = 8'b0000_0000;
        instr_mem[26] = 8'b0000_0000;
        instr_mem[27] = 8'b0000_0000;
        instr_mem[28] = 8'b0000_0000;
        instr_mem[29] = 8'b0000_0000;
        instr_mem[30] = 8'b0000_0000;
        instr_mem[31] = 8'b0000_1000;//-->16

        instr_mem[32] = 8'b1010_0000;//push
        instr_mem[33] = 8'b0010_0000;//rdx

        instr_mem[34] = 8'b1000_0000;//call
        instr_mem[35] = 8'b0000_0000;//val
        instr_mem[36] = 8'b0000_0000;
```

```
        instr_mem[37] = 8'b0000_0000;
        instr_mem[38] = 8'b0000_0000;
        instr_mem[39] = 8'b0000_0000;
        instr_mem[40] = 8'b0000_0000;
        instr_mem[41] = 8'b0000_0000;
        instr_mem[42] = 8'b0011_0000;//----->48

        instr_mem[43] = 8'b0000_0000;//HALT
        instr_mem[44] = 8'b0000_0000;
        instr_mem[45] = 8'b0000_0010;
        instr_mem[46] = 8'b0000_0000;
        instr_mem[47] = 8'b0000_0000;

        instr_mem[48] = 8'b0101_0000;//mrmovq
        instr_mem[49] = 8'b1100_0000;//r12
        instr_mem[50] = 8'b0000_0000;//val
        instr_mem[51] = 8'b0000_0000;
        instr_mem[52] = 8'b0000_0000;
        instr_mem[53] = 8'b0000_0000;
        instr_mem[54] = 8'b0000_0000;
        instr_mem[55] = 8'b0000_0000;
        instr_mem[56] = 8'b0000_0000;
        instr_mem[57] = 8'b0000_1000;//--->16

        instr_mem[58] = 8'b0111_0001;//jmp
        instr_mem[59] = 8'b0000_0000;
        instr_mem[60] = 8'b0000_0000;
        instr_mem[61] = 8'b0000_0000;
        instr_mem[62] = 8'b0000_0000;
        instr_mem[63] = 8'b0000_0000;
        instr_mem[64] = 8'b0000_0000;
        instr_mem[65] = 8'b0000_0000;
        instr_mem[66] = 8'b0011_0000;//----->48
        instr_mem[67] = 8'b1001_1000;//ret

    #1000;
    $finish;
```

```
end
endmodule
```

OUTPUT

```
[Running] seq_tb.v


At T =    0,clk = 0,

instr = 00110000111100110000000000000000000000000000000000000000

rax = x         ||      rbx= x          ||      rcx= x

rsi= x          ||      rbi= x          ||      r8= x

r10= x          ||      r11= x          ||      r12= x

stat= 1



At T =    5,clk = 1,

instr = 00110000111100100000000000000000000000000000000000000000

rax = 0         ||      rbx= 8          ||      rcx= 0

rsi= 0          ||      rbi= 0          ||      r8= 0

r10= 0          ||      r11= 0          ||      r12= 0

stat= 1
```

```
At T =   10,clk = 0,
instr = 0011000011110010000000000000000000000000000000000000000000
rax = 0         ||      rbx= 8          ||      rcx= 0
rsi= 0          ||      rbi= 0          ||      r8= 0
r10= 0          ||      r11= 0          ||      r12= 0
stat= 1



At T =   15,clk = 1,
instr = 0110000000100011010000000100000000000000000000000000000000
rax = 0         ||      rbx= 8          ||      rcx= 0
rsi= 0          ||      rbi= 0          ||      r8= 0
r10= 0          ||      r11= 0          ||      r12= 0
stat= 1



At T =   20,clk = 0,
instr = 0110000000100011010000000100000000000000000000000000000000
rax = 0         ||      rbx= 8          ||      rcx= 0
rsi= 0          ||      rbi= 0          ||      r8= 0
```

```
r10= 0          ||      r11= 0          ||      r12= 0

stat= 1



At T =   25,clk = 1,

instr = 0100000000100000000000000000000000000000000000000000000000000000

rax = 0         ||      rbx= 10         ||      rcx= 0

rsi= 0          ||      rbi= 0          ||      r8= 0

r10= 0          ||      r11= 0          ||      r12= 0

stat= 1



At T =   30,clk = 0,

instr = 0100000000100000000000000000000000000000000000000000000000000000

rax = 0         ||      rbx= 10         ||      rcx= 0

rsi= 0          ||      rbi= 0          ||      r8= 0

r10= 0          ||      r11= 0          ||      r12= 0

stat= 1
```

```
At T =   35,clk = 1,

instr = 10100000001000001000000000000000000000000000000000000

rax = 0         ||      rbx= 10         ||      rcx= 0

rsi= 0          ||      rbi= 0          ||      r8= 0

r10= 0          ||      r11= 0          ||      r12= 0

stat= 1



At T =   40,clk = 0,

instr = 10100000001000001000000000000000000000000000000000000

rax = 0         ||      rbx= 10         ||      rcx= 0

rsi= 0          ||      rbi= 0          ||      r8= 0

r10= 0          ||      r11= 0          ||      r12= 0

stat= 1



At T =   45,clk = 1,

instr = 10000000000000000000000000000000000000000000000000000

rax = 0         ||      rbx= 10         ||      rcx= 0

rsi= 0          ||      rbi= 0          ||      r8= 0
```

```
r10= 0          ||      r11= 0          ||      r12= 0

stat= 1



At T =   50,clk = 0,

instr = 1000000000000000000000000000000000000000000000000000000000000000

rax = 0         ||      rbx= 10         ||      rcx= 0

rsi= 0          ||      rbi= 0          ||      r8= 0

r10= 0          ||      r11= 0          ||      r12= 0

stat= 1



At T =   55,clk = 1,

instr = 0101000011000000000000000000000000000000000000000000000000000000

rax = 0         ||      rbx= 10         ||      rcx= 0

rsi= 0          ||      rbi= 0          ||      r8= 0

r10= 0          ||      r11= 0          ||      r12= 0

stat= 1



At T =   60,clk = 0,
```

```
instr = 0101000011000000000000000000000000000000000000000000000
rax = 0         ||      rbx= 10         ||      rcx= 0
rsi= 0          ||      rbi= 0          ||      r8= 0
r10= 0          ||      r11= 0          ||      r12= 0
stat= 1



At T =   65,clk = 1,
instr = 0111000100000000000000000000000000000000000000000000000
rax = 0         ||      rbx= 10         ||      rcx= 0
rsi= 0          ||      rbi= 0          ||      r8= 0
r10= 0          ||      r11= 0          ||      r12= 2
stat= 1



At T =   70,clk = 0,
instr = 0111000100000000000000000000000000000000000000000000000
rax = 0         ||      rbx= 10         ||      rcx= 0
rsi= 0          ||      rbi= 0          ||      r8= 0
r10= 0          ||      r11= 0          ||      r12= 2
```

```
stat= 1



At T =   75,clk = 1,

instr = 10011000xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

rax = 0          ||      rbx= 10          ||      rcx= 0

rsi= 0           ||      rbi= 0           ||      r8= 0

r10= 0           ||      r11= 0           ||      r12= 2

stat= 1



At T =   80,clk = 0,

instr = 10011000xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

rax = 0          ||      rbx= 10          ||      rcx= 0

rsi= 0           ||      rbi= 0           ||      r8= 0

r10= 0           ||      r11= 0           ||      r12= 2

stat= 1


Halt
./seq.v:27: $finish called at 85 (1s)
```

```
At T =   85,clk = 1,

instr = 0000000000000000000001000000000000000000010100001100

rax = 0         ||      rbx= 10         ||      rcx= 0

rsi= 0          ||      rbi= 0          ||      r8= 0

r10= 0          ||      r11= 0          ||      r12= 2

stat= 4


[Done] exit with code=0 in 0.983 seconds
```

Y86 -64 BIT PIPELINE IMPLEMENTATION

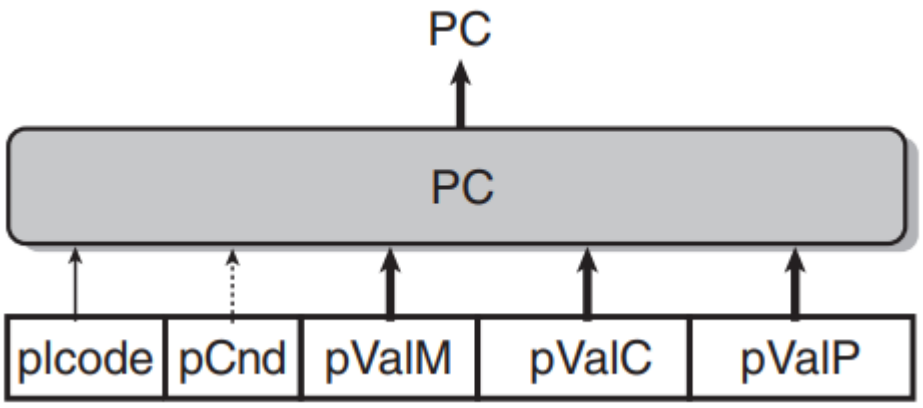
The term Pipelining refers to a technique of decomposing a sequential process into sub-operations, with each sub-operation being executed in a dedicated segment that operates concurrently with all other segments.
A key feature of pipelining is that it increases the throughput of the system.
This is the 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

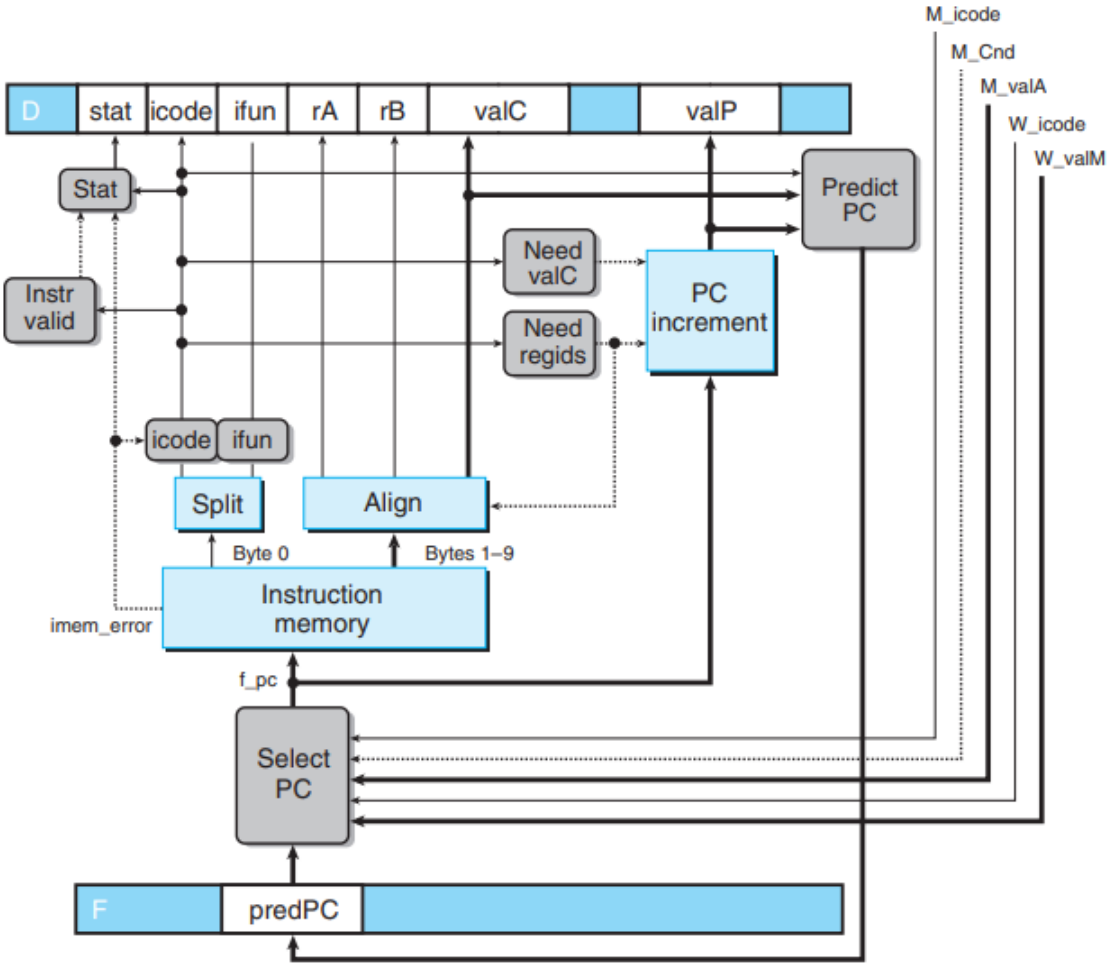
In this type of implementation, signals pass through every stage one by one.

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_Pipe (
input clk,
input F_stall, D_stall, D_bubble, M_cnd,
input [3:0] M_icode, W_icode,
input [63:0] M_valA, W_valM, F_predPC,

//for next intsruction
output reg [63:0] f_predPC,
// for next stage pipe line registers
output reg [3:0] D_icode, D_ifun, D_rA, D_rB,
output reg signed [63:0] D_valC, D_valP,
output reg [0:3] D_stat
);
```

```
    reg [0:7] instr_mem[1023:0];
    reg need_reg;
    reg need_valc;
    reg [63:0] f_PC;
    reg [3:0] f_icode,f_ifun,f_rA,f_rB;
    reg [63:0] f_valC,f_valP;
    reg [0:79] instruction;
    reg imem_error ,instr_invalid;
    reg [0:3] f_stat;

    initial begin
        need_reg=1'b0;
        need_valc=1'b0;
        f_PC=F_predPC;
        $readmemb("input.txt",instr_mem);
        // clk=1'b0;
    end
    // always #5 clk=~clk;


    //imem error
    always @(*) begin
        if(f_PC>64'd1023)begin
            imem_error=1'b1;
        end
        else begin
            imem_error=1'b0;
        end
    end


    //Handling PC changes;
    always@(*) begin
        if(W_icode==4'b1001)
            f_PC = W_valM;            // getting return PC va
        else if(M_icode==4'b0111 & !M_cnd)
            f_PC = M_valA;            // misprediction of bra
        else
```

```
            f_PC = F_predPC;
    end


    // assigning stat code
    always @(*)begin
        if (instr_invalid)
        f_stat = 4'h2;
        else if (imem_error)
        f_stat = 4'h3;
        else if (f_icode==4'h0)
        f_stat = 4'h4;
        else
        f_stat = 4'h1;
    end

    //assigning instruction based on PC
    always @(*)begin
        instruction = {instr_mem[f_PC],instr_mem[f_PC+1],inst
        f_icode = instruction[0:3];
        f_ifun = instruction[4:7];
    end

    //  assign icode,ifun,need_reg,need_valc
    always @(*) begin
        case (f_icode)
            4'b0000: begin                      // Halt
                need_reg =1'b0;
                need_valc =1'b0;
                instr_invalid = 1'b0;
            end
            4'b0001:  begin                    // NOP
                need_reg =1'b0;
                need_valc =1'b0;
                instr_invalid = 1'b0;
            end
            4'b0010: begin                    // cmov
                need_reg=1'b1;
```

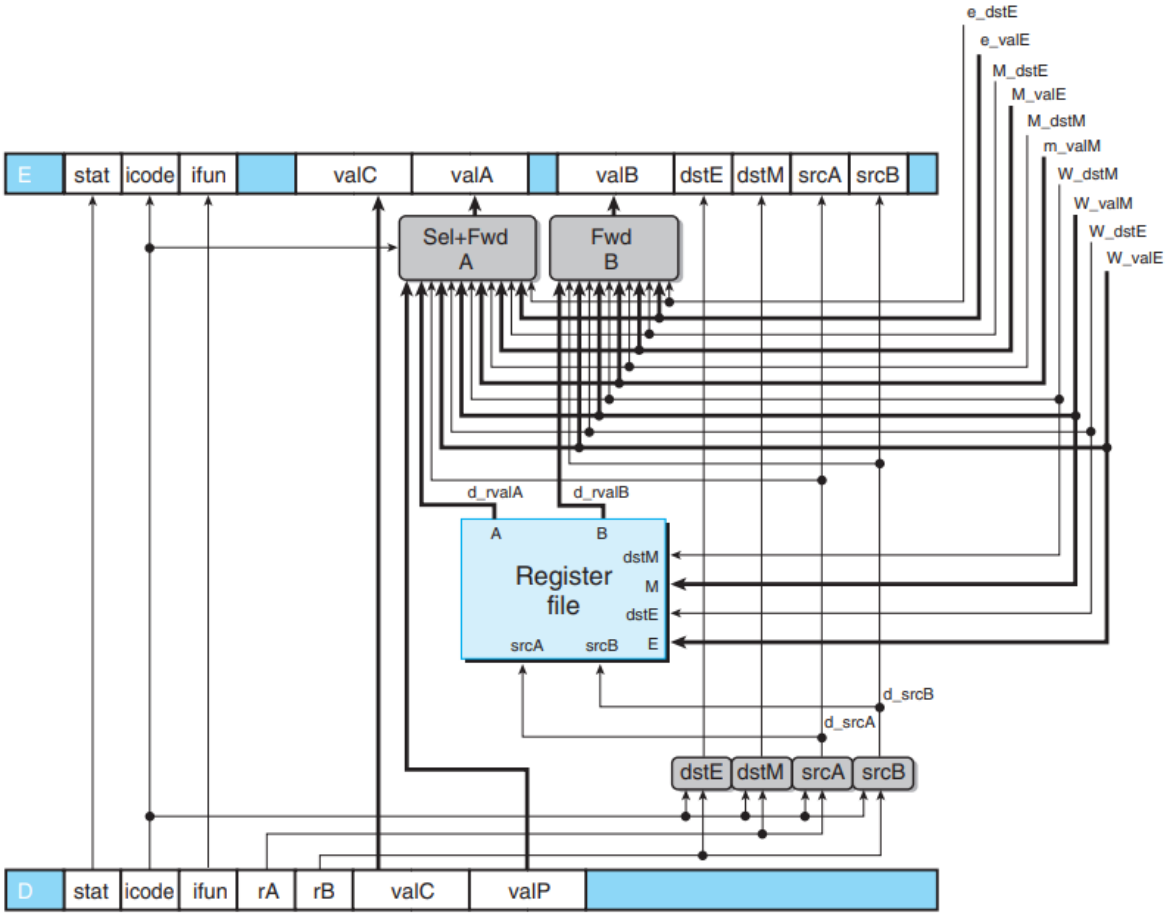
```
                need_valc=1'b0;
                instr_invalid = 1'b0;
            end
            4'b0011:  begin                    // irmov
                need_reg=1'b1;
                need_valc=1'b1;
                instr_invalid = 1'b0;
            end
            4'b0100:  begin                    // rmmov
                need_reg=1'b1;
                need_valc=1'b1;
                instr_invalid = 1'b0;
            end
            4'b0101:  begin                    //mrmov
                need_reg=1'b1;
                need_valc=1'b1;
                instr_invalid = 1'b0;
            end
            4'b0110:  begin                     // opq
                need_reg=1'b1;
                need_valc=1'b0;
                instr_invalid = 1'b0;
            end
            4'b0111:  begin                    // jxx
                need_reg=1'b0;
                need_valc=1'b1;
                instr_invalid = 1'b0;
            end
            4'b1000:  begin                    // call
                need_reg=1'b0;
                need_valc=1'b1;
                instr_invalid = 1'b0;
            end
            4'b1001:  begin                    // ret
                need_reg=1'b0;
                need_valc=1'b0;
                instr_invalid = 1'b0;
            end
```

```
            4'b1010:  begin                    // push
                need_reg=1'b1;
                need_valc=1'b0;
                instr_invalid = 1'b0;
            end
            4'b1011:  begin                    // pop
                need_reg=1'b1;
                need_valc=1'b0;
                instr_invalid = 1'b0;
            end
            default: instr_invalid = 1'b1;         // Default
        endcase


        //assign rA,rB,valC,valP,predicted_PC;
        f_valP = f_PC + 1 + need_reg + 8*need_valc;
        if(need_reg == 1'b1)begin
            f_rA = instruction[8:11];
            f_rB = instruction[12:15];
            f_predPC=f_valP;        // predicted PC will be v
        end
        else begin
        f_rA=4'b1111;
        f_rB=4'b1111;
        end
        if(need_valc == 1'b1 && need_reg == 1'b0)begin
            f_valC = instruction[8:71];
            f_predPC=f_valC;                    // Predicti
        end
        else if(need_valc == 1'b1 && need_reg == 1'b1)begin
            f_valC=instruction[16:79];
        end
        else begin
            f_valC=64'd0;
        end
    end
```

```
    // writing to pipeline registers;
    always @(posedge clk ) begin
        if (F_stall==1'b1)
        begin
            f_PC = F_predPC;
        end

        if (D_bubble==1'b1)
        begin
            D_icode <= 4'h1;
            D_ifun <= 4'h0;
            D_rA <= 4'hF;
            D_rB <= 4'hF;
            D_valC <= 64'b0;
            D_valP <= 64'b0;
            D_stat <= 4'h1;
        end
        else if(D_stall==1'b1)
        begin

        end
        else
        begin
            D_icode <= f_icode;
            D_ifun <= f_ifun;
            D_rA <= f_rA;
            D_rB <= f_rB;
            D_valC <= f_valC;
            D_valP <= f_valP;
            D_stat <= f_stat;
        end

    end


endmodule
```

DECODE and WRITEBACK stages:

- read program registers.
- update program registers.



- 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.

```
module Decode_Pipe (
    input clk,
    input [3:0] D_icode,D_ifun,D_rA,D_rB,
    input [63:0] D_valC,D_valP,
    input [0:3] D_stat,
    input [3:0] e_dstE,M_dstE,M_dstM,W_dstE,W_dstM,
    input [63:0] e_valE,m_valM,M_valE,W_valM,W_valE,
    input E_bubble,
    input [3:0] W_icode,

    output reg signed [63:0] rax,rbx,rcx,rdx,rsp,rbp,rsi,rbi,
    // for next stage pipe line registers
    output reg [3:0] E_icode, E_ifun, E_dstE, E_dstM, E_srcA,
    output reg signed [63:0] E_valC, E_valA, E_valB,
    output reg [0:3] E_stat,
    output reg [3:0] d_srcA,d_srcB
);
```

```
    reg [3:0] d_dstE, d_dstM;
    reg [63:0] d_valA, d_valB;
    reg [63:0] d_rvalA, d_rvalB;
    reg signed [63:0] reg_file[0:14];

// assigning values to the reg_files initially
    initial begin
        reg_file[0] = 64'd0;        //rax
        reg_file[1] = 64'd0;        //rcx
        reg_file[2] = 64'd0;       //rdx
        reg_file[3] = 64'd0;       //rbx
        reg_file[4] = 64'd256;       //rsp
        reg_file[5] = 64'd64;        //rbp
        reg_file[6] = 64'd0;      //rsi
        reg_file[7] = 64'd0;     //rdi
        reg_file[8] = 64'd0;    //r8
        reg_file[9] = 64'd0;    //r9
        reg_file[10] = 64'd0;    //r10
        reg_file[11] = 64'd0;    //r11
        reg_file[12] = 64'd0;        //r12
        reg_file[13] = 64'd0;     //r13
        reg_file[14] = 64'd0;     //r14
    end


      //Decode
    always @(*) begin
        case(D_icode)
            4'b0000: begin // halt
            end
            4'b0001: begin // nop
            end
            4'b0010: begin // cmovXX
                d_srcA = D_rA;
                d_dstE = D_rB;
                d_rvalA = reg_file[d_srcA];
                d_rvalB = 64'd0;
            end
```

```
            4'b0011: begin // irmov
                d_dstE = D_rB;
            end
            4'b0100: begin // rmmovq
                d_srcA = D_rA;
                d_srcB = D_rB;
                d_rvalA = reg_file[d_srcA];
                d_rvalB = reg_file[d_srcB];
            end
            4'b0101: begin // mrmovq
                d_srcB = D_rB;
                d_dstM = D_rA;
                d_rvalB = reg_file[d_srcB];
            end
            4'b0110: begin // opq
                d_srcA = D_rA;
                d_srcB = D_rB;
                d_dstE = D_rB;
                d_rvalA = reg_file[d_srcA];
                d_rvalB = reg_file[d_srcB];
            end
            4'b0111: begin // jxx
            end
            4'b1000: begin // call
                d_srcB = 4;
                d_dstE = 4;
                d_rvalB = reg_file[4];
            end
            4'b1001: begin // ret
                d_srcA = 4;
                d_srcB = 4;
                d_dstE = 4;
                d_rvalA = reg_file[4];
                d_rvalB = reg_file[4];
            end
            4'b1010: begin // pushq
                d_srcA = D_rA;
                d_srcB = 4;
```

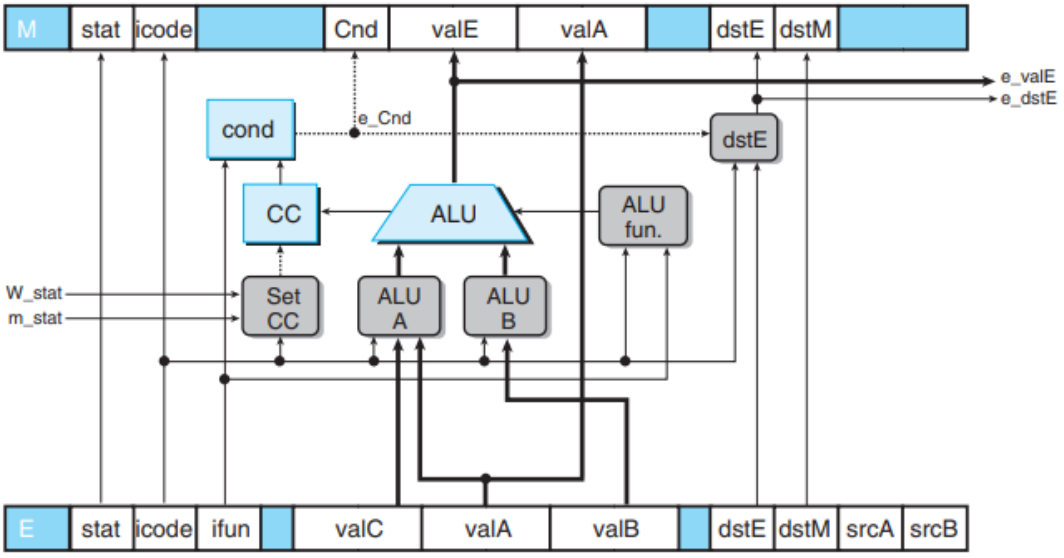
```
                d_dstE = 4;
                d_rvalA = reg_file[d_srcA];
                d_rvalB = reg_file[4];
            end
            4'b1011: begin // popq
                d_srcA = 4;
                d_srcB = 4;
                d_dstE = 4;
                d_dstM = D_rA;
                d_rvalA = reg_file[4];
                d_rvalB = reg_file[4];
            end
            default : begin
                d_srcA = 4'hF;
                d_srcB = 4'hF;
                d_dstE = 4'hF;
                d_dstM = 4'hF;
            end
        endcase
    end

    //Data Forwarding
    always @(*) begin

        // for A;
        if(D_icode == 4'b1000 || D_icode == 4'b0111)
            d_valA = D_valP;
        else if(d_srcA == e_dstE)
            d_valA = e_valE;
        else if(d_srcA == M_dstM)
            d_valA = m_valM;
        else if(d_srcA == M_dstE)
            d_valA = M_valE;
        else if(d_srcA == W_dstM)
            d_valA = W_valM;
        else if(d_srcA == W_dstE)
            d_valA = W_valE;
        else
```

```
            d_valA = d_rvalA;

        //for B;
        if(d_srcB == e_dstE)
            d_valB = e_valE;
        else if(d_srcB == M_dstM)
            d_valB = m_valM;
        else if(d_srcB == M_dstE)
            d_valB = M_valE;
        else if(d_srcB == W_dstM)
            d_valB = W_valM;
        else if(d_srcB == W_dstE)
            d_valB = W_valE;
        else
            d_valB = d_rvalB;
    end


    //Write Back
    always @(*) begin
        case(W_icode)
            4'b0000: begin // halt
            end
            4'b0001: begin // nop
            end
            4'b0010: begin // cmovXX
                reg_file[W_dstE] = W_valE;
            end
            4'b0011: begin // irmov
                reg_file[W_dstE] = W_valE;
            end
            4'b0100: begin // rmmovq
            end
            4'b0101: begin // mrmovq
                reg_file[W_dstM] = W_valM;
            end
            4'b0110: begin // opq
                reg_file[W_dstE] = W_valE;
```

```
            end
            4'b0111: begin // jxx
                reg_file[W_dstE] = W_valE;
            end
            4'b1000: begin // call
                reg_file[W_dstE] = W_valE;
            end
            4'b1001: begin // ret
                reg_file[W_dstE] = W_valE;
            end
            4'b1010: begin // pushq
                reg_file[W_dstE] = W_valE;
            end
            4'b1011: begin // popq
                reg_file[W_dstE] = W_valE;
                reg_file[W_dstM] = W_valM;
            end
        endcase

    end
    always @(*) begin
        rax <= reg_file[0];
        rcx <= reg_file[1];
        rdx <= reg_file[2];
        rbx <= reg_file[3];
        rsp <= reg_file[4];
        rbp <= reg_file[5];
        rsi <= reg_file[6];
        rbi <= reg_file[7];
        r8  <= reg_file[8];
        r9  <= reg_file[9];
        r10 <= reg_file[10];
        r11 <= reg_file[11];
        r12 <= reg_file[12];
        r13 <= reg_file[13];
        r14 <= reg_file[14];
    end
```

```
    // writing to pipeline registers;
    always @(posedge clk ) begin
        if (E_bubble == 1'b1) begin
            E_icode <= 4'h1;
            E_ifun <= 4'h0;
            E_valC <= 4'h0;
            E_valA <= 4'h0;
            E_valB <= 4'h0;
            E_dstE <= 4'hF;
            E_dstM <= 4'hF;
            E_srcA <= 4'hF;
            E_srcB <= 4'hF;
            E_stat <= 4'h1;
        end
        else   begin
            E_icode <= D_icode;
            E_ifun <= D_ifun;
            E_srcA <= d_srcA;
            E_srcB <= d_srcB;
            E_dstE <= d_dstE;
            E_dstM <= d_dstM;
            E_valA <= d_valA;
            E_valB <= d_valB;
            E_valC <= D_valC;
            E_stat <= D_stat;
        end
    end
endmodule
```

EXECUTE:



- 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.

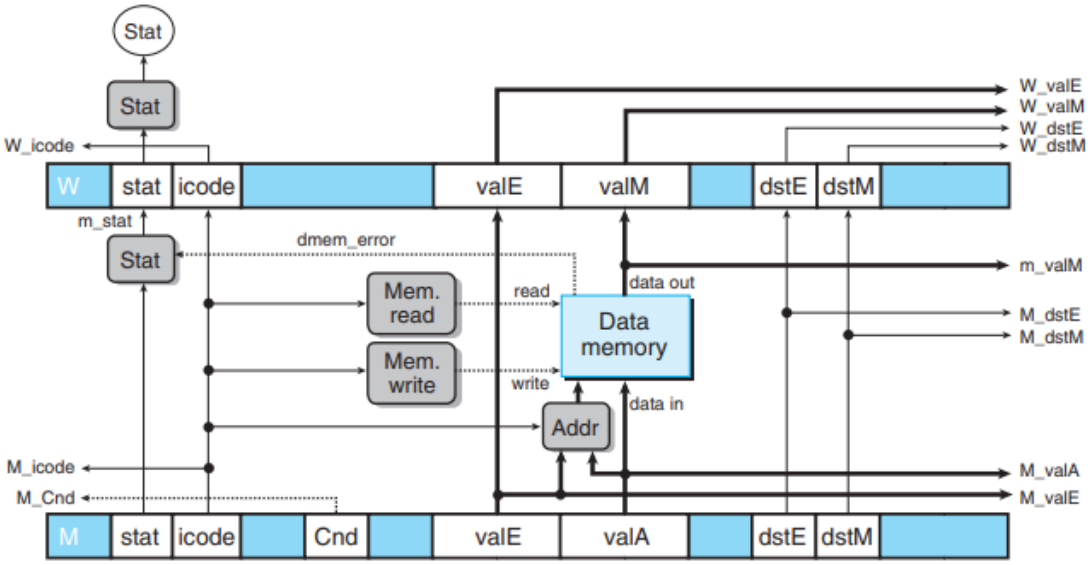
```
module Execute_Pipe (
    input clk,
    input [3:0] E_icode, E_ifun, E_dstE, E_dstM, E_srcA, E_sr
    input [63:0] E_valC, E_valA, E_valB,
    input [0:3] E_stat,

    input [0:3] W_stat,m_stat,

    output reg [3:0] e_dstE,
    output reg [63:0] e_valE,


    output reg [3:0] M_icode, M_dstE, M_dstM,
    output reg [63:0] M_valE, M_valA,
    output reg [0:3] M_stat,
    output reg M_cnd,e_cnd
```

```
);

    reg [1:0]control;
    reg signed [63:0]ALUa;
    reg signed [63:0]ALUb;
    wire signed [63:0]Alu_out;
    wire overflow;
    reg  condition[6:0];
    reg cf,zf,sf,of;

    initial begin
        cf = 1'b0;
        zf = 1'b0;
        sf = 1'b0;
        of = 1'b0;
    end


    ALU ALU(ALUa,ALUb,control,Alu_out,overflow);

    always@(*)
    begin
        cf = 1'b0;
        zf = (Alu_out == 1'b0);
        sf = (Alu_out<1'b0);
        of = (ALUa<1'b0 == ALUb<1'b0)&&(Alu_out<1'b0 != ALUa<

        condition[0]  =    1'b1;                      //
        condition[1]  =    (sf^of)|zf ;               // le
        condition[2]  =    (sf^of);                   // l
        condition[3]  =    zf;                        // e
        condition[4]  =    ~zf;                       // ne
        condition[5]  =    ~(sf^of);                  // ge
        condition[6]  =    (~(sf^of))&(~zf);          // g
    end
```

```
    always@(*)begin
        case (E_icode)
            4'b0010: begin                    // cmov
                e_cnd = condition[E_ifun];
                e_valE = E_valA+64'd0;
            end
            4'b0011:  begin                    // irmov
                e_valE = 64'd0 + E_valC;
            end
            4'b0100:  begin                    // rmmov
                e_valE = E_valB + E_valC;
            end
            4'b0101:  begin                    // mrmov
            e_valE = E_valB + E_valC;
            end
            4'b0110:  begin                 // opq
                control = E_ifun[1:0];
                ALUa = E_valA;
                ALUb = E_valB;
                e_valE = Alu_out;
            end
            4'b0111:  begin                    // jxx
                e_cnd = condition[E_ifun];
            end
            4'b1000:  begin                    // call
                e_valE = -64'd8+E_valB;
            end
            4'b1001:  begin                    // ret
                e_valE = 64'd8+E_valB;
            end
            4'b1010:  begin                    // push
                e_valE = -64'd8+E_valB;
            end
            4'b1011:  begin                    // pop
                e_valE = 64'd8+E_valB;
            end
            default:;        // Default case
        endcase
```

```
    end


    //checking for condition;
    always @(*)
    begin
        if(E_icode == 2 || E_icode == 7)
        begin
            e_dstE = (e_cnd == 1) ? E_dstE : 4'b1111;
        end
        else
        begin
            e_dstE = E_dstE;
        end
    end

    // writiing to pipeline registers;
    always @(posedge clk)
    begin
        M_stat <= E_stat;
        M_icode <= E_icode;
        M_cnd <= e_cnd;
        M_valE <= e_valE;
        M_valA <= E_valA;
        M_dstE <= e_dstE;
        M_dstM <= E_dstM;
    end
endmodule
```

MEMORY:



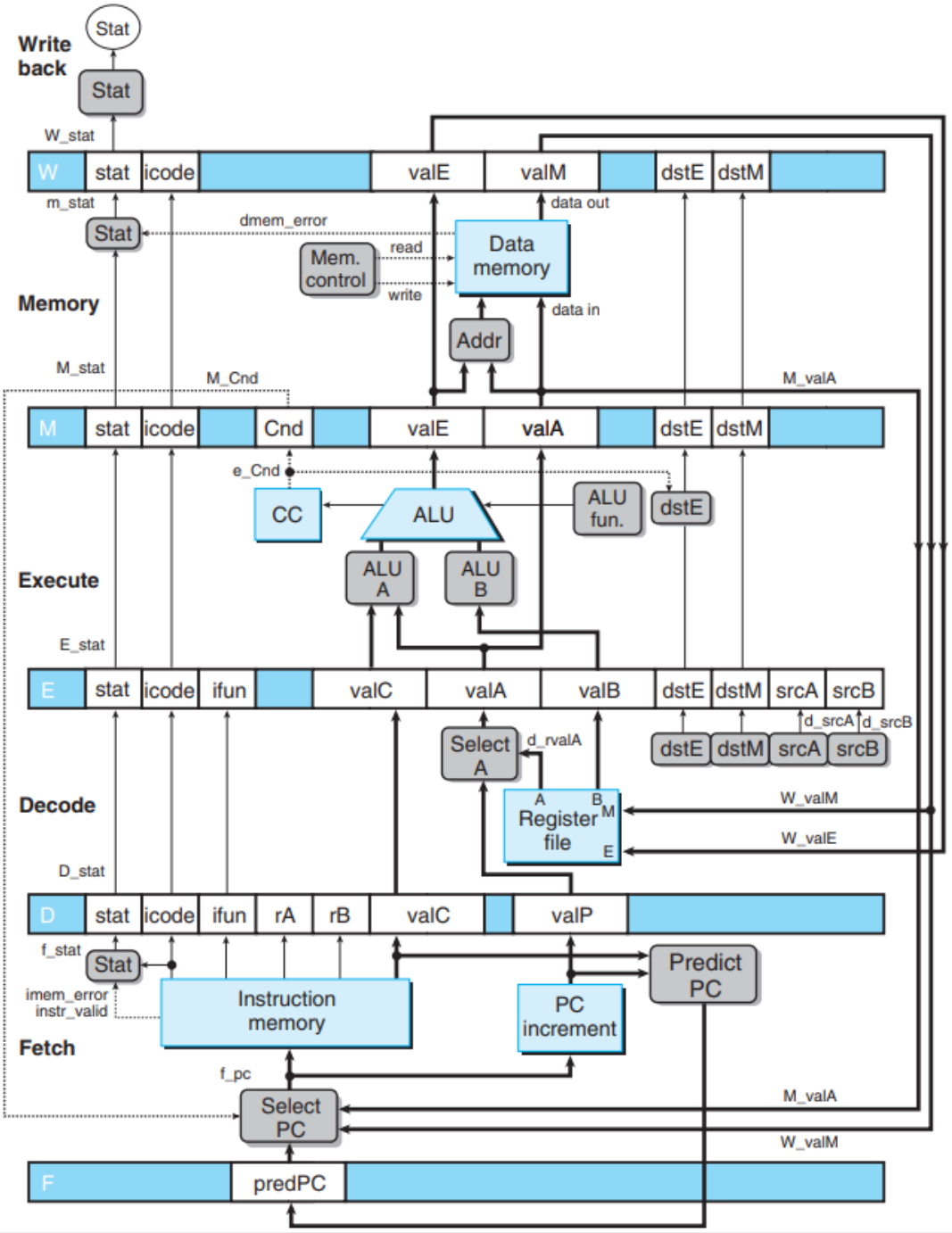
- Memory block either reads or writes the program data.
- Memory stage in pipeline lacks “Mem.data” block present in SEQ as the task is performed by “Sel+Fwd A” block in decode stage.

```
module Memory_Pipe (
    input clk,
    input [3:0] M_icode, M_dstE, M_dstM,
    input [63:0] M_valE, M_valA,
    input [0:3] M_stat,
    input M_cnd,

    output reg [3:0] W_icode, W_dstE, W_dstM,
    output reg [63:0] W_valE, W_valM,
    output reg [0:3] W_stat,

    output reg [63:0] m_valM,
    output reg [0:3] m_stat
);
    reg [63:0] ram [8191:0];
    reg dmem_error;


    always @(*)begin
        if(M_valE > 8191 || M_valA > 8191)
```

```
        begin
            dmem_error = 1;
        end
        if(dmem_error == 1)
        m_stat = 4'h3;
        else
        m_stat = M_stat;
        case(M_icode)
            4'b0101: m_valM = ram [M_valE];   //mrmovq
            4'b1001: m_valM = ram [M_valA];   //return
            4'b1011: m_valM = ram [M_valA];   //popq
        endcase

    end

    always@(posedge clk)
    begin
        case(M_icode)
            4'b0100: ram [M_valE] = M_valA;   //rmmovq
            4'b1000: ram [M_valE] = M_valA;   //call
            4'b1010: ram [M_valE] = M_valA;   //pushq
        endcase
    end

    // writing to pipeline registers;
    always @(posedge clk)
    begin
        W_icode <= M_icode;
        W_stat <= m_stat;
        W_valE <= M_valE;
        W_valM <= m_valM;
        W_dstM <= M_dstM;
        W_dstE <= M_dstE;
    end
endmodule
```

```
module Pipeline_Control (
    input [3:0] D_icode, d_srcA, d_srcB,E_icode, E_dstM,M_ico
    input e_cnd,
    input [0:3] m_stat,W_stat,

    output reg  F_stall, D_stall, D_bubble, E_bubble
);

always @(*)
    begin
        if (D_icode == 4'b1001 || E_icode == 4'b1001 || M_ico
        begin
            F_stall = 1'b1;
            D_bubble = 1'b1;
        end
        else if((E_icode == 4'b0101 || E_icode == 4'b1011) &&
        begin
            F_stall = 1'b1;
            D_stall = 1'b1;
            E_bubble = 1'b1;
        end
        else if (E_icode == 4'b0111 && !e_cnd) //Jump mispred
        begin
            D_bubble = 1'b1;
            E_bubble = 1'b1;
        end
        else begin
            F_stall = 1'b0;
            D_stall = 1'b0;
            D_bubble = 1'b0;
            E_bubble = 1'b0;
        end
    end

endmodule
```

Overall implementation of pipeline Y86 architecture




```
`include "fetch_pipe.v"
`include "decode_pipe.v"
```

```
`include "execute_pipe.v"
`include "memory_pipe.v"
`include "control_logic.v"
module processor;

reg clk;

reg [63:0] F_predPC;
wire [63:0] f_predPC;
reg [0:3] stat = 4'h0;



wire [3:0] D_icode, D_ifun, D_rA, D_rB;
wire signed [63:0] D_valC, D_valP;
wire [0:3] D_stat;

wire [3:0] d_srcA,d_srcB;




wire [3:0] E_icode, E_ifun;
wire signed [63:0] E_valA, E_valB, E_valC;
wire [3:0] E_srcA, E_srcB, E_dstE, E_dstM;
wire [0:3] E_stat;

wire [3:0] e_dstE;
wire signed [63:0] e_valE;
wire e_cnd;




wire [3:0] M_icode, M_dstE, M_dstM;
wire signed [63:0] M_valA, M_valE;
wire [0:3] M_stat;
wire M_cnd;
```

```
wire signed [63:0] m_valM;
wire [0:3] m_stat;




wire [0:3] W_stat;
wire [3:0] W_icode, W_dstE, W_dstM;
wire signed [63:0] W_valE, W_valM;



//registers
wire signed [63:0] rax,rbx,rcx,rdx,rsp,rbp,rsi,rbi,r8,r9,r10,


//control
wire F_stall, D_stall, D_bubble, E_bubble, M_bubble, W_stall,


always #10 clk = ~clk;


Fetch_Pipe fetch(clk,F_stall,D_stall,D_bubble,M_cnd,M_icode,W

Decode_Pipe decode(clk,D_icode,D_ifun,D_rA,D_rB,D_valC,D_valP

Execute_Pipe execute(clk,E_icode, E_ifun, E_dstE, E_dstM, E_s


Memory_Pipe memory(clk,M_icode, M_dstE, M_dstM,M_valE, M_valA

Pipeline_Control pipe_control(D_icode, d_srcA, d_srcB,E_icode
```

```
// stopping program based on error flags from stat
always @(stat)
begin
    case (stat)
        4'h2:
        begin
            $display("----------------------------------------
            $finish;
        end
        4'h3:
        begin
            $display("----------------------------------------
            $finish;
        end
        4'h4:
        begin
            $display("----------------------------------------
            $finish;
        end
        4'h1:
        begin

        end
    endcase
end

always @(W_stat)
begin
    stat = W_stat;
end

always @(posedge clk )
begin
    F_predPC = f_predPC;
end

initial begin
```

```
    $dumpfile("processor.vcd");
    $dumpvars(0,processor);
    F_predPC = 64'd0;
    clk = 0;

    // $monitor("Time=%0t\tclk=%0d\n\n ,M_icode=%0d , M_dstE=
    // $monitor("Time=%0t\tclk=%0d\n\n E_icode=%0d, \t\t E_if
    // $monitor("Time=%0t\tclk=%0d\n\nD_icode=%0d,\t\tD_ifun=
    // $monitor("Time=%0t\tclk=%0d\n\nf_predPC=%0d \t\tF_pred

    $monitor("Time = %0t \t\t\t clk = %b \n\n ---------------

    // $monitor("Time=%0t\tclk=%0d\n\nf_predPC=%0d \t\tF_pred

end

endmodule
```

Data Forwarding

- Data Forwarding in Naïve Pipeline, Register isn’t written until completion of write-backstage 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

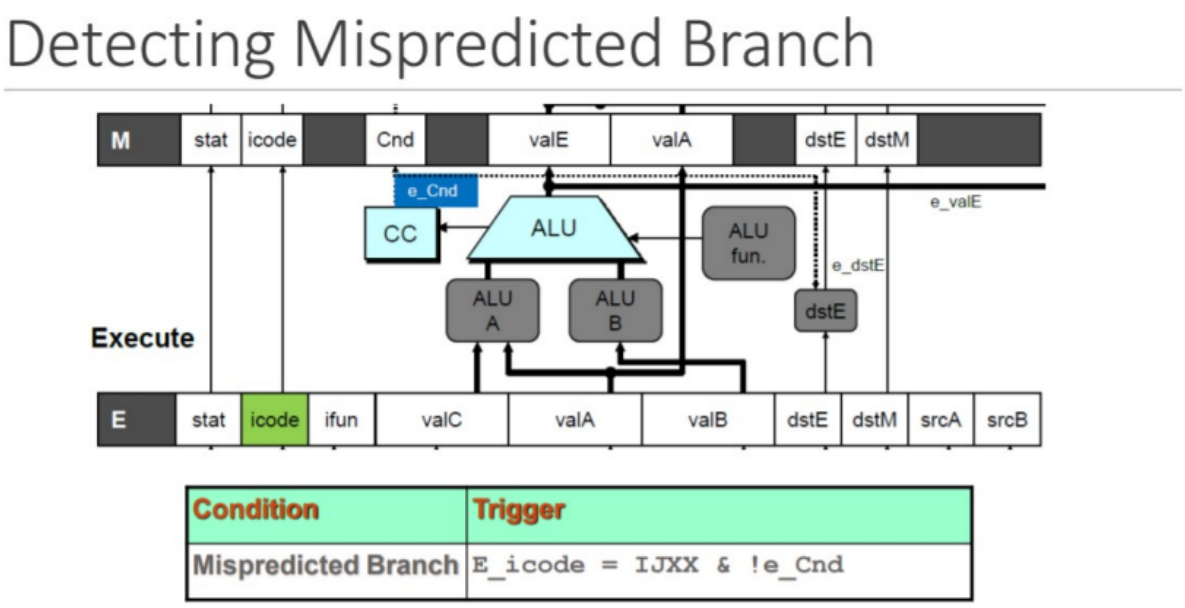
- Add additional feedback paths from E, M, and W pipeline registers into decode stage.
- Create logic blocks to select from multiple sources for valA and valB in decode stage.

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.

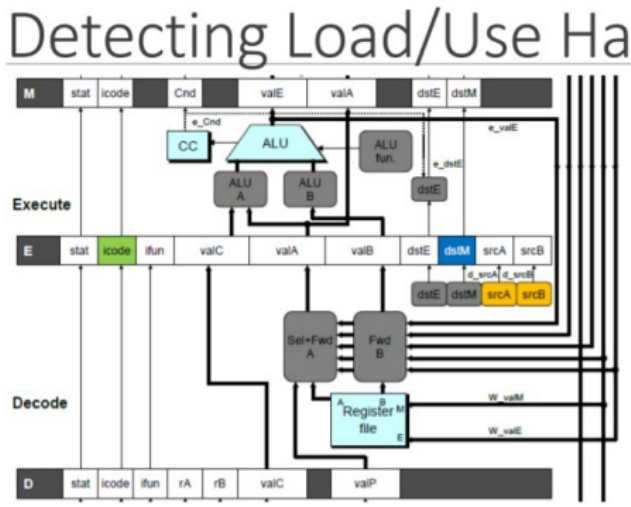


Handling Misprediction -

- 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:

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.



Condition	Trigger
Load/Use Hazard	`E_icode in { IMRMOVQ, IPOPQ } && E_dstM in { d_srcA, d_srcB }`

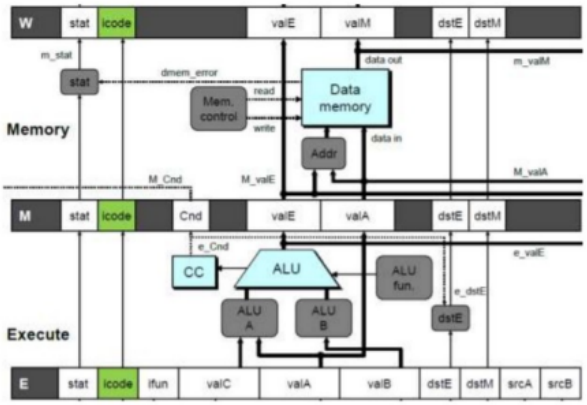
Control for Load/Use Hazard -

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

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.

Detecting Return



Condition	Trigger
Processing `ret`	`IRET in { D_icode, E_icode, M_icode }`

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-backstage.