PROJECT REPORT
TEAM : NO NAME
ISHANYA SETHI 2020102014
LAKSH BALANI 2020102019

1. **FETCH STAGE**

   The fetch stage includes the instruction memory hardware unit. This unit reads 10 bytes from memory at a time, using the PC as the address of the first byte (byte 0). The 0th byte is interpreted as the instruction byte and is split (by the Split block) into two 4-bit quantities. The control logic blocks labelled "icode" and "ifun" then compute the instruction and function codes as equaling either the values read from memory or, in the event that the instruction address is not valid (as indicated by the signal imem_error from Instr Valid block), the values corresponding to a nop instruction. The control logic blocks labelled "icode" and "ifun" then compute the instruction 2 and function codes as equaling either the values read from memory or, in the event that the instruction address is not valid (as indicated by the signal imem_error), the values corresponding to a nop instruction. Based on the value of icode, we can compute three 1-bit signals

   - instr_valid. Does this byte correspond to a legal Y86-64 instruction? This signal is used to detect an illegal instruction.

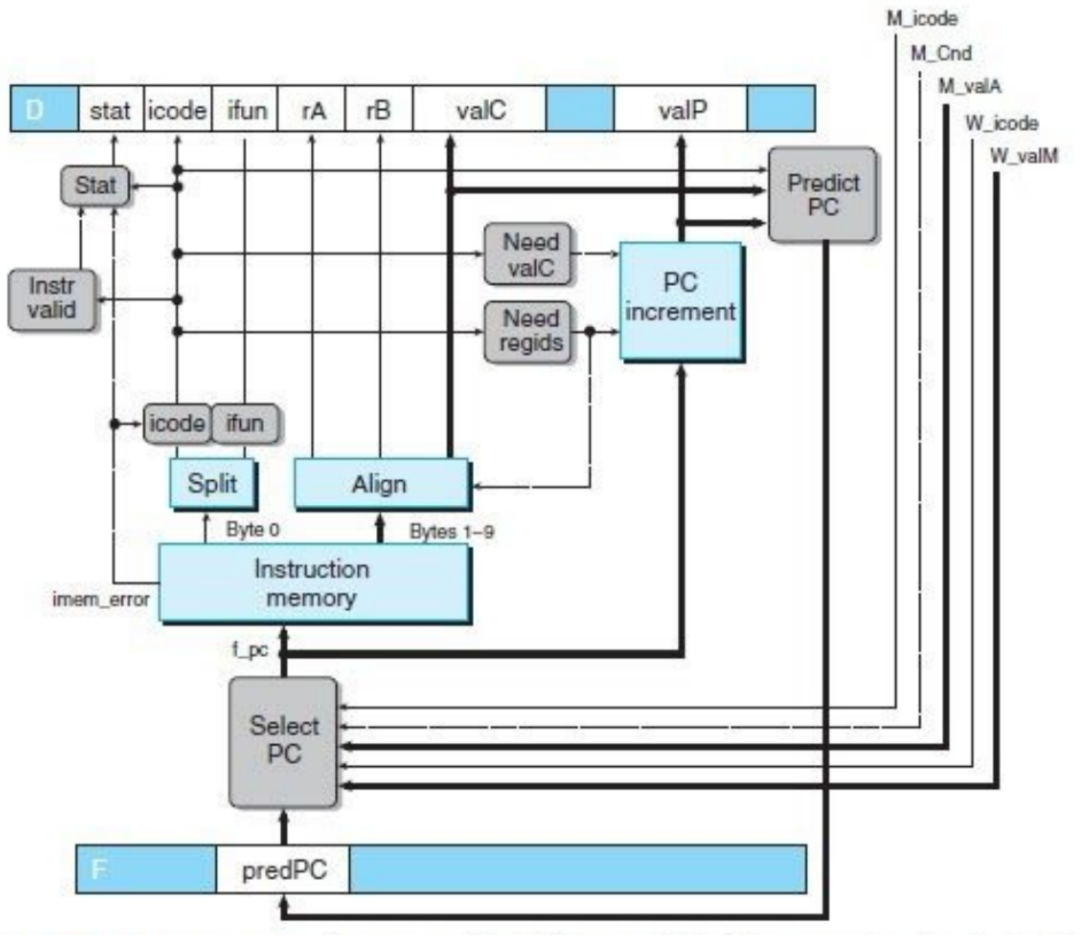
   - need_regids. Does this instruction include a register specifier byte?

   - need_valC. Does this instruction include a constant word?

The signals instr_valid and imem_error (generated when the instruction address is out of bounds) are used to generate the status code in the memory stage. The remaining 9 bytes read from the instruction memory encode some combination of the register specifier byte and the constant word. These bytes are processed by the hardware unit labelled "Align" into the register fields and the constant word. Byte 1 is split into register specifiers rA and rB when the computed signal need_regids is 1. If need_regids is 0, both register specifiers are set to 0xF (RNONE), indicating there are no registers specified by this instruction. For any instruction having only one register operand, the other field of the register specifier byte will be 0xF (RNONE). Thus, we can assume that the signals rA and rB either encode registers we want to access or indicate that register access is not required. The PC incrementer block generates the signal valP, based on the current value of the PC, and the two signals need_regids and need_valC. For PC value p, need_regids value r, and need_valC value i, the incrementer generates the value p + 1 + r + 8i. Instr_valid is a signal used to detect an illegal instruction. 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.

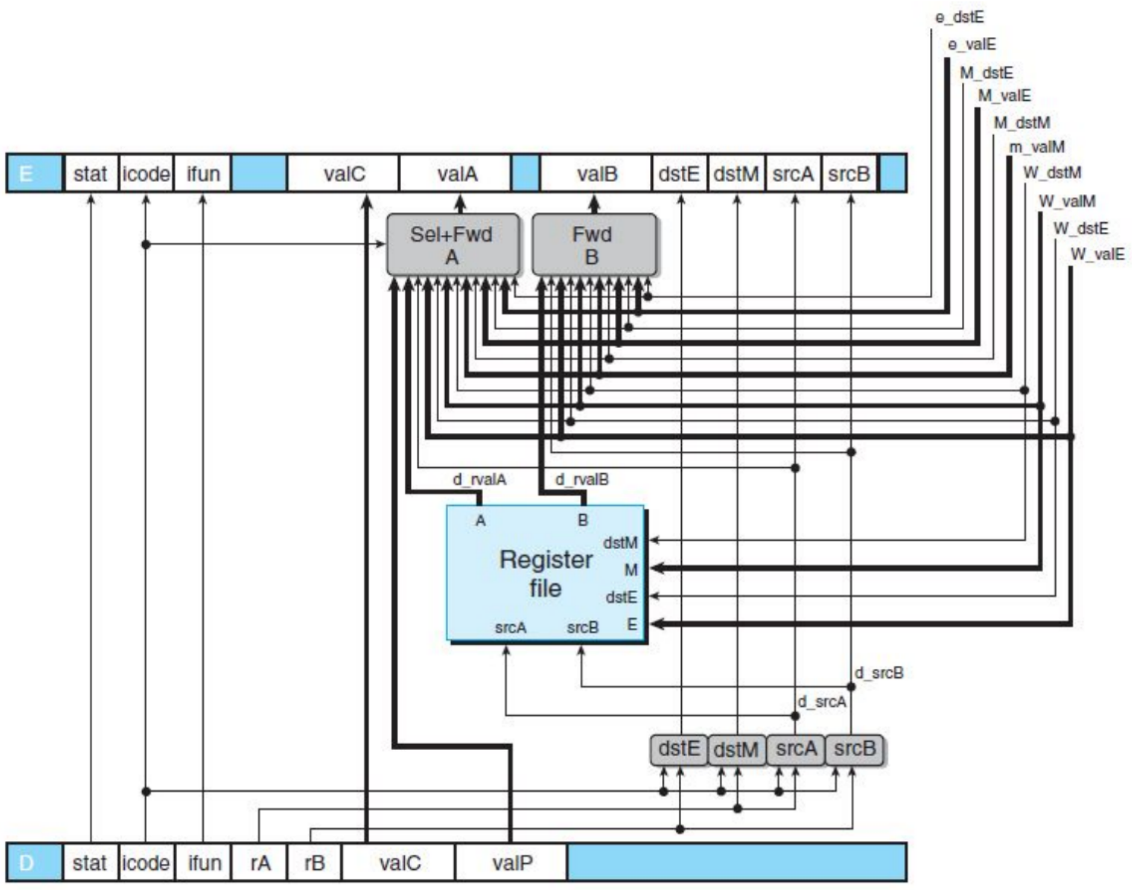
CODE

```
module Instr_memory(
    PC,
    __split,
    __align,
    imem_error
);
   initial begin
       $readmemh("./instructions.mem",instr_mem);
   end

   input [63:0] PC;
   reg [7:0] instr_mem[2047:0];
   output reg[71:0] __align;
   output reg[7:0] __split;
   output reg imem_error;


   always @(PC)
   begin
       imem_error <= (PC < 64'd0 || PC > 64'd9824) ? 1'b1:1'b0;
        __split <= instr_mem[PC];
        __align[71:64] <= instr_mem[PC+1];
        __align[63:56] <= instr_mem[PC+2];
        __align[55:48] <= instr_mem[PC+3];
        __align[47:40] <= instr_mem[PC+4];
        __align[39:32] <= instr_mem[PC+5];
        __align[31:24] <= instr_mem[PC+6];
        __align[23:16] <= instr_mem[PC+7];
        __align[15:8]  <= instr_mem[PC+8];
        __align[ 7:0]  <= instr_mem[PC+9];
   end

endmodule

module split(ibyte, icode, ifun);
    input [7:0] ibyte;
    output [3:0] icode, ifun;
    assign ifun = ibyte[3:0];
    assign icode = ibyte[7:4];

endmodule

module align(ibytes, need_regids, rA, rB, valC);
    input need_regids;
    input [71:0] ibytes;
    output [ 3:0] rA, rB;
    output [63:0] valC;
    assign rA = ibytes[71:68];
    assign rB = ibytes[67:64];
    assign valC = need_regids ? ibytes[63:0] : ibytes[71:8];

endmodule

module PC_increment(PC, need_regids, need_valC, valP);
    input [63:0] PC;
    input need_regids, need_valC;
    output [63:0] valP;
    assign valP = PC + 1 + 8*need_valC + need_regids;

endmodule
```



2. DECODE and WRITE BACK STAGE

There are four ports in the registry file. It can handle up to two reads (on ports A and B) and two writes at the same time (on ports E and M). Each port has an address connection and a data connection, with the address connection being a register ID and the data connection being a collection of 64 wires serving as either a read port's output word or a write port's input word. The two read ports have address inputs srcA and srcB, while the two write ports have address inputs dstE and dstM. The special identifier 0xF (RNONE) on an address port indicates that no register should be accessed. The 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. Register ID srcA indicates which register should be read to generate valA. Register ID dstE indicates the destination register for write port E, where the computed value valE is stored. But here, the register IDs supplied to the write ports come from the write-back stage (signals W_dstE and W_dstM), and not the one coming from the decode stage because we want the writes to occur to the destination registers specified by the instruction in the write-back stage.



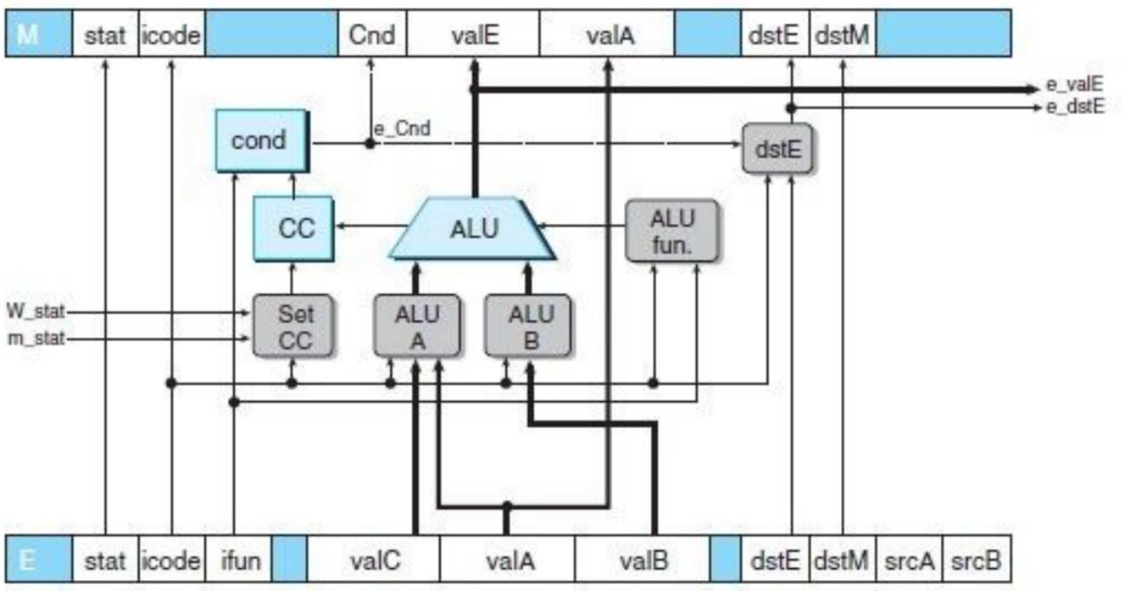
CODE

```
module decode(dstE,dstM,srcA,srcB,icode,rA,rB,Cnd);
 input [3:0] icode;
 input [3:0] rA;
 input [3:0] rB;
 input Cnd;
 output [3:0] srcA;
 output [3:0] srcB;
 output [3:0] dstE;
 output [3:0] dstM;

// These are the instructions
  parameter IHALT = 4'h0;
  parameter INOP = 4'h1;
  parameter IRRMOVQ = 4'h2;
  parameter IIRMOVQ = 4'h3;
  parameter IRMMOVQ = 4'h4;
  parameter IMRMOVQ = 4'h5;
  parameter IOPQ = 4'h6;
  parameter IJXX = 4'h7;
  parameter ICALL = 4'h8;
  parameter IRET = 4'h9;
  parameter IPUSHQ = 4'hA;
  parameter IPOPQ = 4'hB;

assign srcA =
  icode == IRMMOVQ ? rA:
  icode == IOPQ ? rA:
  icode == IPUSHQ ? rA:
  icode == IRET ? 4'h4:
  icode == IRRMOVQ ? rA:
  icode == IPOPQ ? 4'h4 : 4'hf;

assign srcB =
  icode == IMRMOVQ ? rB:
  icode == IOPQ ? rB:
  icode == IRMMOVQ ? rB:
  icode == IRET ? 4'h4:
  icode == ICALL ? 4'h4:
  icode == IPUSHQ ? 4'h4:
  icode == IPOPQ ? 4'h4 : 4'hf;

assign dstE =
  icode == IRRMOVQ ? (Cnd ? rB : 4'hf):
  icode == IIRMOVQ ? rB:
  icode == IOPQ ? rB:
  icode == IRET ? 4'h4:
```

```
  icode == ICALL ? 4'h4:
  icode == IPUSHQ ? 4'h4:
  icode == IPOPQ ? 4'h4 : 4'hf;

assign dstM =
  icode == IMRMOVQ ? rA:
  icode == IPOPQ ? rA: 4'hf;



endmodule
```

3. 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 computation for each instruction is shown as the first step in the execute stage. The operands are listed with aluB first, followed by aluA to make sure the subq instruction subtracts valA from valB. The value of aluA can be valA, valC, or either −8 or +8, depending on the instruction type. Our ALU generates the three signals on which the condition codes are based—zero, sign, and overflow—every time it operates. We also have "Set CC," which determines whether or not to update the condition codes, has signals m_stat and W_stat as inputs. The Cond block generates the Cnd signal used both for the setting of dstE with conditional moves and in the next PC logic for conditional branches. The signals e_valE and e_dstE are directed toward the decode stage as one of the forwarding sources. One difference is that the logic labelled "Set CC," which determines whether or not to update the condition codes, has signals m_stat and W_stat as inputs. 5 These signals are used to detect cases where an instruction causing an exception is passing through later pipeline stages, and therefore any updating of the condition codes should be suppressed. These signals are used to detect cases where an instruction causing an exception is passing through later pipeline stages, and therefore any updating of the condition codes should be suppressed.

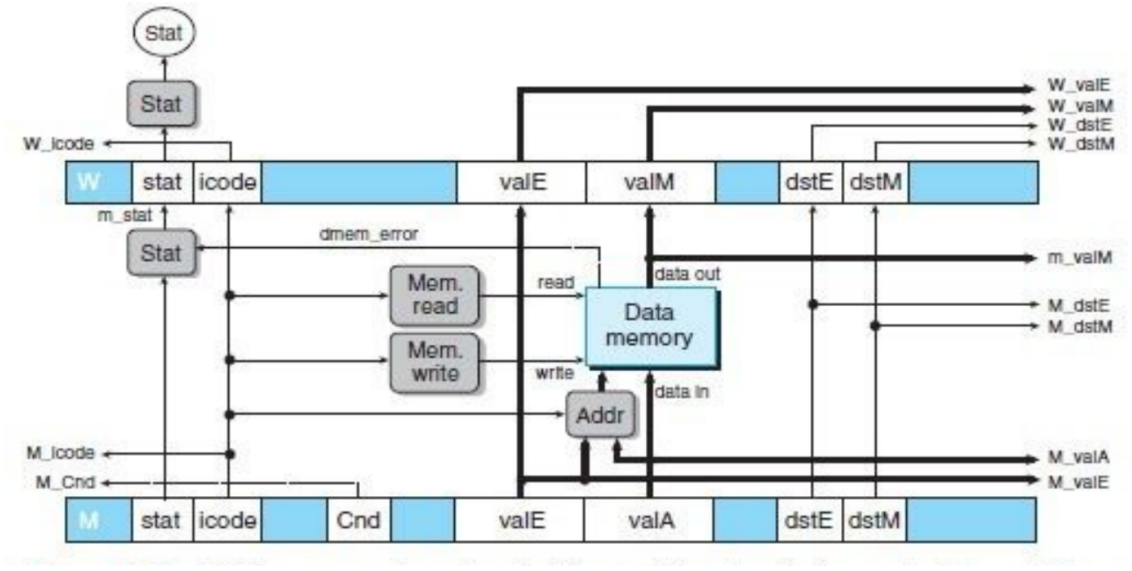


CODE

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


module execute(icode, ifun, valA, valB, valC, Cnd, valE);
// Branch condition
  input [3:0] ifun;
  input [3:0] icode;
  input [63:0] valA;
  input [63:0] valB;
  input [63:0] valC;
  output reg [63:0] valE;
  output reg Cnd;

  // Jump conditions.
  parameter J_YES = 4'h0;
  parameter J_LE = 4'h1;
  parameter J_L = 4'h2;
  parameter J_E = 4'h3;
  parameter J_NE = 4'h4;
  parameter J_GE = 4'h5;
  parameter J_G = 4'h6;

  // These are the instructions
  parameter IHALT = 4'h0;
  parameter INOP = 4'h1;
  parameter IRRMOVQ = 4'h2;
  parameter IIRMOVQ = 4'h3;
  parameter IRMMOVQ = 4'h4;
  parameter IMRMOVQ = 4'h5;
  parameter IOPQ = 4'h6;
  parameter IJXX = 4'h7;
  parameter ICALL = 4'h8;
  parameter IRET = 4'h9;
  parameter IPUSHQ = 4'hA;
  parameter IPOPQ = 4'hB;

  assign ALUA =
  icode == IRRMOVQ ?valA:
  icode == IOPQ ? valA :
  icode == IIRMOVQ ? valC:
  icode == IRMMOVQ ? valC:
  icode == IMRMOVQ ? valC :
  icode == ICALL ? -8:
  icode == IPUSHQ ? -8 :
  icode == IRET ? 8:
  icode == IPOPQ ? 8 : 0;

  assign ALUB =
  icode == IRMMOVQ ? valB:
  icode == IMRMOVQ ? valB:
  icode == IOPQ ? valB:
  icode == ICALL ? valB:
  icode == IPUSHQ ? valB:
  icode == IRET ? valB:
  icode == IPOPQ ? valB :
  icode == IRRMOVQ ? 0:
  icode == IIRMOVQ ? 0 : 0;

  wire [1:0] sel_line;
  assign sel_line =
  icode == IRMMOVQ ? 2'b0:
  icode == IMRMOVQ ? 2'b0:
  icode == IOPQ ? ifun:
  icode == ICALL ? 2'b0:
  icode == IPUSHQ ? 2'b0:
  icode == IRET ? 2'b0:
  icode == IPOPQ ? 2'b0 :
  icode == IRRMOVQ ? 2'b0:
  icode == IIRMOVQ ? 2'b0 : 0;



  wire overflow_flag;
  wire [63:0]ans;
  ALU alu_module
(.ans(ans),.overflow(overflow_flag),.sel(sel_line),.a(valA),.b(va
lB));
  reg ZF,SF,OF;


  always@(*)
  begin
  valE = ans;
  SF=valE[63];
  if(valE== 64'b0) begin
    ZF=1;
  end
  else begin
     ZF=0;
  end

  OF=overflow_flag;


 Cnd =
  (ifun == J_YES) |
  (ifun == J_LE & ((ZF^OF)|ZF)) |
  (ifun == J_L & (SF^OF)) |
  (ifun == J_E & ZF) |
  (ifun == J_NE & ~ZF) |
  (ifun == J_GE & (~SF^OF)) |
  (ifun == J_G & (~SF^OF)&~ZF);
   end

endmodule
```

4. MEMORY

The memory stage has the task of either reading or writing program data. Two control blocks generate the values for the memory address and the memory input data (for write operations). Two other blocks generate the control signals indicating whether to perform a read or a write operation. When a read operation is performed, the data memory generates the value valM. We set the control signal mem_read only for instructions that read data from memory. Many of the values in pipeline registers and M and W are supplied to other parts of the circuit as part of the forwarding and pipeline control logic.

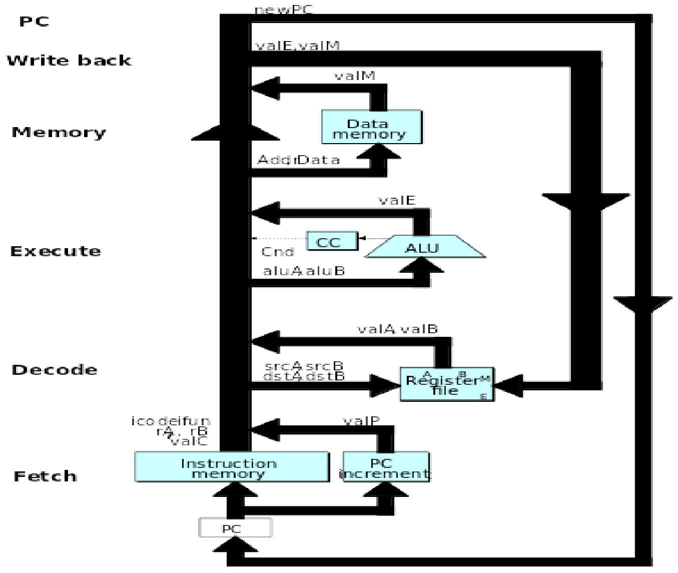


CODE

```
module mem_1 (mem_r,mem_w,mem_add,mem_data,icode,valE,valA,valP);

  // These are the instructions
  parameter IHALT = 4'h0;
  parameter INOP = 4'h1;
  parameter IRRMOVQ = 4'h2;
  parameter IIRMOVQ = 4'h3;
  parameter IRMMOVQ = 4'h4;
  parameter IMRMOVQ = 4'h5;
  parameter IOPQ = 4'h6;
  parameter IJXX = 4'h7;
  parameter ICALL = 4'h8;
  parameter IRET = 4'h9;
  parameter IPUSHQ = 4'hA;
  parameter IPOPQ = 4'hB;


 input [3:0] icode;
 input [63:0] valA;
 input [63:0] valE;
 input [63:0] valP;
 output mem_r;
 output mem_w;
 output [63:0] mem_add;
 output [63:0] mem_data;

 wire mem_r,mem_w;
 assign mem_r =(icode == IMRMOVQ | icode == IPOPQ | icode ==
IRET);
 assign mem_w =(icode == IRMMOVQ | icode == IPUSHQ | icode ==
ICALL);

 assign mem_addr =
 icode == IRMMOVQ ? valE :
 icode == IPUSHQ ? valE :
 icode == ICALL ? valE :
 icode == IMRMOVQ ? valE : valA;
 assign mem_data=(icode==IRMMOVQ||icode==IPUSHQ) ? valA : valP;

endmodule

module mem_2 (valM,dmem_error,mem_r,mem_w,mem_add,mem_data);
 input mem_r,mem_w;
 input [63:0] mem_add;
 input [63:0] mem_data;
 output dmem_error;
 output [63:0] valM;

 reg [63:0] mem [4095:0];
 reg [63:0] __valM;

always @ (*) begin
     if(mem_add<=64'h0FFF&&mem_add>=64'h0) begin
        if(mem_r ^ mem_w)
        begin

        if(mem_w == 1'b1)
        begin
        mem[mem_add] = mem_data;
        end

        if ( mem_r == 1'b1 ) begin
                 __valM [63:56] = mem[mem_add]   ;
                 __valM [55:48] = mem[mem_add+1]   ;
```

```
                    __valM [47:40] = mem[mem_add+2]  ;
                    __valM [39:32] = mem[mem_add+3]  ;
                    __valM [33:24] = mem[mem_add+4]  ;
                    __valM [23:16] = mem[mem_add+5]  ;
                    __valM [15:8 ] = mem[mem_add+6]  ;
                    __valM [ 7:0 ] = mem[mem_add+7]  ;
              end

         end
      end
end
assign valM=__valM;

endmodule
```

5. **Forward (Upward) Paths**
   Values passed from one stage to next.Cannot jump past stage e.g., valC passes through decode



OVERALL Y-86 PROCESSOR

CODE

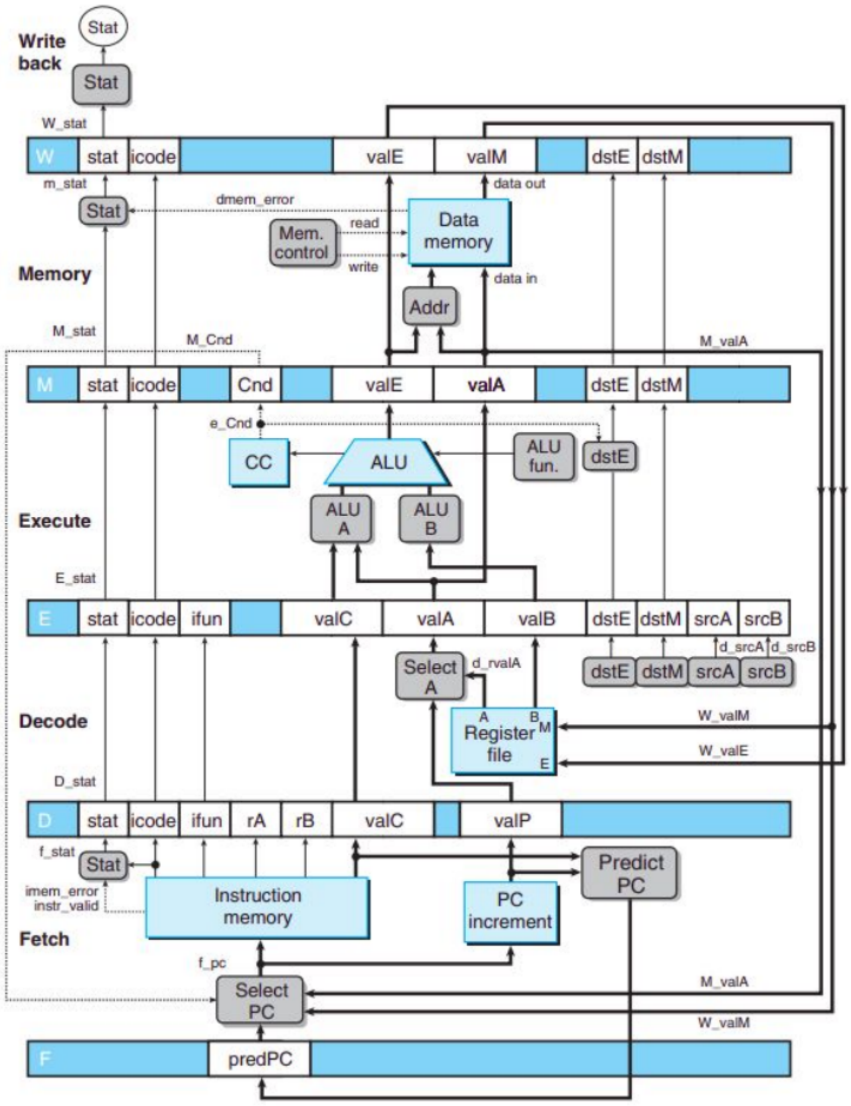
```
`include "fetch.v"
`include "decode.v"
`include "execute.v"
`include "memory.v"
`include "write_back.v"

module pipereg(out, in, stall, Bubbleval, clock);

     input stall, clock;
```

```
  parameter w=8;
     input [w-1:0] in;
     input [w-1:0] Bubbleval;
     output reg [w-1:0] out;
     initial begin
         assign out = Bubbleval;
     end

     always @(posedge clock)
     begin
         if (!stall)
             assign out = in;
     end

endmodule

module pipereg2(out, in, stall,Bubble, Bubbleval, clock);

  parameter w=8;
     output reg [w-1:0] out;
     input [w-1:0] in;
     input stall,Bubble;
     input [w-1:0] Bubbleval;
     input clock;

     initial begin
          assign out=Bubbleval;
     end

     always @(posedge clock) begin
         if (!stall && !Bubble)
             assign out = in;
         else if (!stall && Bubble)
             assign out = Bubbleval;
     end
endmodule


module y86 ( clk, W_stat );

  // These are the instructions
      parameter IHALT = 4'h0;
      parameter INOP = 4'h1;
      parameter IRRMOVQ = 4'h2;
      parameter IIRMOVQ = 4'h3;
      parameter IRMMOVQ = 4'h4;
      parameter IMRMOVQ = 4'h5;
      parameter IOPQ = 4'h6;
      parameter IJXX = 4'h7;
      parameter ICALL = 4'h8;
      parameter IRET = 4'h9;
      parameter IPUSHQ = 4'hA;
      parameter IPOPQ = 4'hB;
  // These are some Status conditions
     parameter BUB = 3'h0;
     parameter AOK = 3'h1;
     parameter HLT = 3'h2;
     parameter ADR = 3'h3;
     parameter INS = 3'h4;


input clk ;
output [2:0] W_stat;


wire [63:0] f_predPC, F_predPC, f_pc;
wire imem_error;
wire [ 2:0] f_stat;
wire [71:0] __align;
wire [7:0] __split;
wire [ 3:0] f_icode, f_ifun;
wire [ 3:0] f_rA, f_rB;
wire [63:0] f_valC;
wire [63:0] f_valP;
wire need_regids, need_valC, instr_valid;
wire F_stall, F_BUBble, F_reset;
wire nr;
wire nc;


wire [ 2:0] D_stat;
wire [63:0] D_pc;
wire [ 3:0] D_icode;
wire [ 3:0] D_ifun;
wire [ 3:0] D_rA, D_rB;
wire [63:0] D_valC;
wire [63:0] D_valP;
wire [63:0] d_valA;
wire [63:0] d_valB;
wire [63:0] d_rvalA, d_rvalB;
wire [3:0] d_dstE;
wire [3:0] d_dstM;
wire [3:0] d_srcA;
wire [3:0] d_srcB;
wire d_cnd;
wire D_stall, D_BUBble;
wire [ 2:0] E_stat;
wire [63:0] E_pc;
wire [ 3:0] E_icode, E_ifun;
wire [63:0] E_valC, E_valA, E_valB;
wire [ 3:0] E_dstE, E_dstM, E_srcA, E_srcB;
wire [63:0] ALUA, ALUB;
wire set_CC;
wire [ 2:0] CC;
wire [ 2:0] new_CC;
wire [ 3:0] ALUfun;
wire [63:0] e_valE, e_valA;
wire [ 3:0] e_dstE;
wire e_Cnd;
wire E_stall, E_BUBble;


wire mem_r;
wire mem_w;
wire [ 2:0] M_stat;
wire [63:0] M_pc;
wire [ 3:0] M_icode, M_ifun;
wire M_Cnd;
wire [63:0] m_valM;
wire [63:0] M_valE, M_valA,M_valP;
wire [ 3:0] M_dstE, M_dstM;
wire [ 2:0] m_stat;
wire [63:0] mem_add;
wire [63:0] mem_data;
wire dmem_error;

wire [ 2:0] W_stat;
wire [63:0] W_pc;
wire [ 3:0] W_icode;
wire signed [63:0] W_valA;
wire signed [63:0] W_valB;
wire signed [63:0] W_valC;
wire signed [63:0] W_valE;
wire signed [63:0] W_valM;
wire signed [63:0] W_valP;
wire [3:0] W_srcA,W_srcB;
wire [ 3:0] W_dstE, W_dstM;
wire [ 3:0] w_dstE, w_dstM;
wire W_stall, W_BUBble, resetting;

      assign Stat =((W_stat == BUB) ? AOK : W_stat);
     assign F_bubble = 0;
```

```
	assign F_stall =(((E_icode == IMRMOVQ | E_icode == IPOPQ) &
(E_dstM == d_srcA | E_dstM == d_srcB)) | (IRET == D_icode | IRET
== E_icode | IRET == M_icode));
	assign D_stall =((E_icode == IMRMOVQ | E_icode == IPOPQ) &
(E_dstM == d_srcA | E_dstM == d_srcB));
	assign D_bubble =(((E_icode == IJXX) & ~e_Cnd) | (~((E_icode
== IMRMOVQ | E_icode ==IPOPQ) & (E_dstM == d_srcA | E_dstM ==
d_srcB)) & (IRET ==D_icode | IRET == E_icode | IRET == M_icode)));
	assign E_stall =0;
	assign E_bubble =(((E_icode == IJXX) & ~e_Cnd) | ((E_icode ==
IMRMOVQ | E_icode == IPOPQ) & (E_dstM == d_srcA | E_dstM ==
d_srcB)));
	assign M_stall =0;
	assign M_bubble =((m_stat == ADR | m_stat == INS | m_stat ==
HLT) | (W_stat == ADR | W_stat == INS | W_stat == HLT));
	assign W_stall =(W_stat == ADR | W_stat == INS | W_stat ==
HLT);
	assign W_bubble =0;


pipereg #(64) F_predPC_reg(F_predPC, f_predPC, F_stall, 64'b0,
clock);

// D stage
	pipereg #(3) D_stat_reg(D_stat, f_stat, D_stall,   BUB,
clock);
	pipereg #(64) D_pc_reg(D_pc, f_pc, D_stall,   64'b0, clock);
	    pipereg2 #(4) D_icode_reg(D_icode, f_icode, D_stall,
D_BUBble,  INOP, clock);
	//pipereg #(4) D_icode_reg(D_icode, f_icode, D_stall,   INOP,
clock);
	pipereg #(4) D_ifun_reg(D_ifun, f_ifun, D_stall, 4'h0,
clock);
	pipereg #(4) D_rA_reg(D_rA, f_rA, D_stall, 4'hF, clock);
	pipereg #(4) D_rB_reg(D_rB, f_rB, D_stall, 4'hF, clock);
	pipereg #(64) D_valC_reg(D_valC, f_valC, D_stall, 64'b0,
clock);
	pipereg #(64) D_valP_reg(D_valP, f_valP, D_stall, 64'b0,
clock);

// E stage
	pipereg #(3) E_stat_reg(E_stat, D_stat, E_stall,  BUB,
clock);
	pipereg #(64) E_pc_reg(E_pc, D_pc, E_stall,  64'b0, clock);
	    pipereg #(4) E_icode_reg(E_icode, D_icode, E_stall,
INOP, clock);
	pipereg #(4) E_ifun_reg(E_ifun, D_ifun, E_stall,  4'h0,
clock);
```

```
	pipereg #(64) E_valC_reg(E_valC, D_valC, E_stall,  64'b0,
clock);
	pipereg #(64) E_valA_reg(E_valA, d_valA, E_stall,  64'b0,
clock);
	pipereg #(64) E_valB_reg(E_valB, d_valB, E_stall,  64'b0,
clock);
	pipereg #(4) E_dstE_reg(E_dstE, d_dstE, E_stall,  4'hF,
clock);
	pipereg #(4) E_dstM_reg(E_dstM, d_dstM, E_stall,  4'hF,
clock);
	pipereg #(4) E_srcA_reg(E_srcA, d_srcA, E_stall,  4'hF,
clock);
	pipereg #(4) E_srcB_reg(E_srcB, d_srcB, E_stall,  4'hF,
clock);

// M stage
	pipereg #(3) M_stat_reg(M_stat, E_stat, M_stall,  BUB,
clock);
	pipereg #(64) M_pc_reg(M_pc, E_pc, M_stall,  64'b0, clock);
	pipereg #(4) M_icode_reg(M_icode, E_icode, M_stall,  INOP,
clock);
	pipereg #(4) M_ifun_reg(M_ifun, E_ifun, M_stall,  4'h0,
clock);
	pipereg #(1) M_Cnd_reg(M_Cnd, e_Cnd, M_stall,  1'b0, clock);
	pipereg #(64) M_valE_reg(M_valE, e_valE, M_stall,  64'b0,
clock);
	pipereg #(64) M_valA_reg(M_valA, e_valA, M_stall,  64'b0,
clock);
	pipereg #(4) M_dstE_reg(M_dstE, e_dstE, M_stall,  4'hF,
clock);
	pipereg #(4) M_dstM_reg(M_dstM, E_dstM, M_stall,  4'hF,
clock);

// W stage
	pipereg #(3) W_stat_reg(W_stat, m_stat, W_stall,  BUB,
clock);
	pipereg #(64) W_pc_reg(W_pc, M_pc, W_stall,  64'b0, clock);
	pipereg #(4) W_icode_reg(W_icode, M_icode, W_stall,  INOP,
clock);
	pipereg #(64) W_valE_reg(W_valE, M_valE, W_stall,  64'b0,
clock);
	pipereg #(64) W_valM_reg(W_valM, m_valM, W_stall,  64'b0,
clock);
	pipereg #(4) W_dstE_reg(W_dstE, M_dstE, W_stall,  4'hF,
clock);
	pipereg #(4) W_dstM_reg(W_dstM, M_dstM, W_stall,  4'hF,
clock);
```

```
Instr_memory
ins_m(.PC(f_pc),.__split(__split),.__align(__align),.imem_error(im
em_error));

split sp(.ibyte(__split), .icode(f_icode), .ifun(f_ifun));

assign nr = ((f_icode == IRRMOVQ) || (f_icode == IIRMOVQ) ||
(f_icode == IRMMOVQ) || (f_icode == IMRMOVQ) || (f_icode == IOPQ)
|| (f_icode == IPUSHQ) || (f_icode == IPOPQ)) ? 1'b1 : 1'b0 ;
assign nc = ((f_icode == IIRMOVQ) || (f_icode == IRMMOVQ) ||
(f_icode == IMRMOVQ) || (f_icode == IJXX) || (f_icode == ICALL) )
? 1'b1 : 1'b0 ;

align al(.ibytes(__align), .need_regids(nr), .rA(f_rA), .rB(f_rB),
.valC(f_valC));

PC_increment inc_pc(.PC(f_pc), .need_regids(nr), .need_valC(nc),
.valP(f_valP));

decode dec(.dstE(d_dstE), .dstM(d_dstM), .srcA(d_srcA),
.srcB(d_srcB), .icode(D_icode), .rA(D_rA), .rB(D_rB),
.Cnd(d_cnd));

execute exe(.icode(E_icode), .ifun(E_ifun), .valA(E_valA),
.valB(E_valB), .valC(E_valC), .Cnd(e_cnd), .valE(e_valE));

mem_1
m1(.mem_r(mem_r),.mem_w(mem_w),.mem_add(mem_add),.mem_data(mem_dat
a),.icode(M_icode),.valE(M_valE),.valA(M_valA),.valP(M_valP));
mem_2 m2(
.valM(m_valM),.dmem_error(dmem_error),.mem_r(mem_r),.mem_w(mem_w),
.mem_add(mem_add),.mem_data(mem_data));

write_back wr_ba(.valA(W_valA), .valB(W_valB), .valM(W_valM),
.valE(W_valE), .dstE(W_dstE), .dstM(W_dstM), .srcA(W_srcA),
.srcB(W_srcB) );


endmodule
```



The processor supports the following set of commands (as of now) :

1. inop
2. ihalt
3. ioPq
4. ipopq
5. ipushq
6. iirmovq
7. irmmovq
8. imrmovq
9. ijxx
10. iret
11. icall

Some of the above instructions also have sub-instructions that can be performed.

● IOPL

1. ADDQ - Performs addition of 2 numbers
2. SUBQ - Performs subtraction between 2 numbers
3. ANDQ - Performs LOGICAL AND of 2 numbers

4. XORQ - Performs LOGICAL XOR of 2 numbers

- IJXX

1. C_YES - Jumps without condition
2. C_LE - Jumps if it is less than or equal to
3. C_L - Jumps if it is less than
4. C_E - Jumps if it is equal to
5. C_NE - Jumps if it is not equal to
6. C_GE - Jumps if it is greater than or equal to
7. C_G - Jumps if it is greater than