Y86-64 Processor

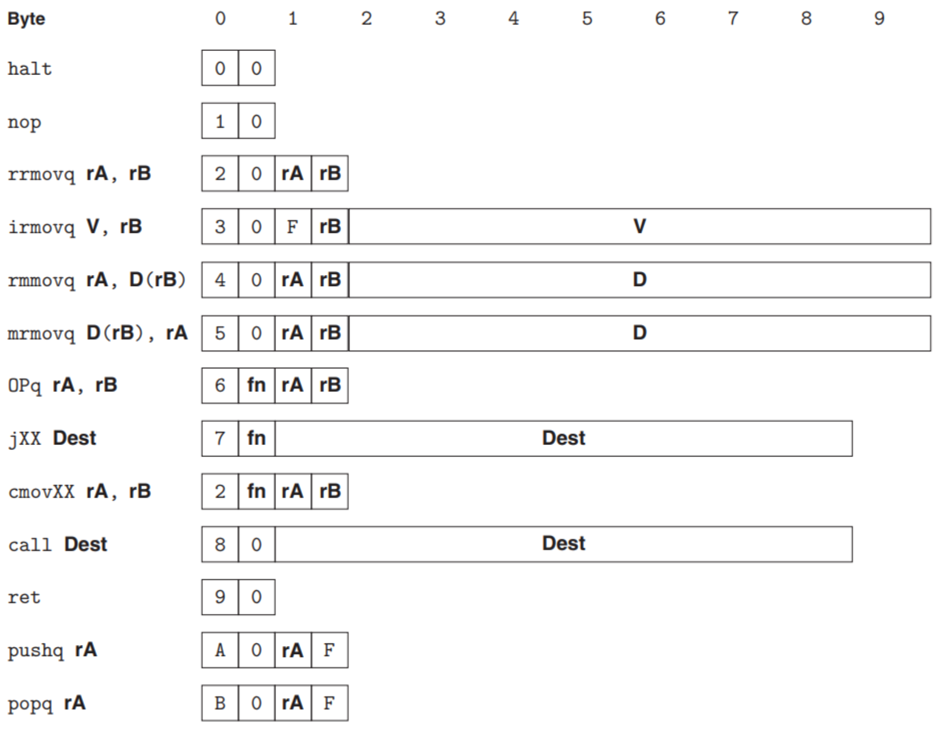
Team AHF

1. Agrim Rawat 2020102037
2. Jewel Benny 2020102057

Instructions

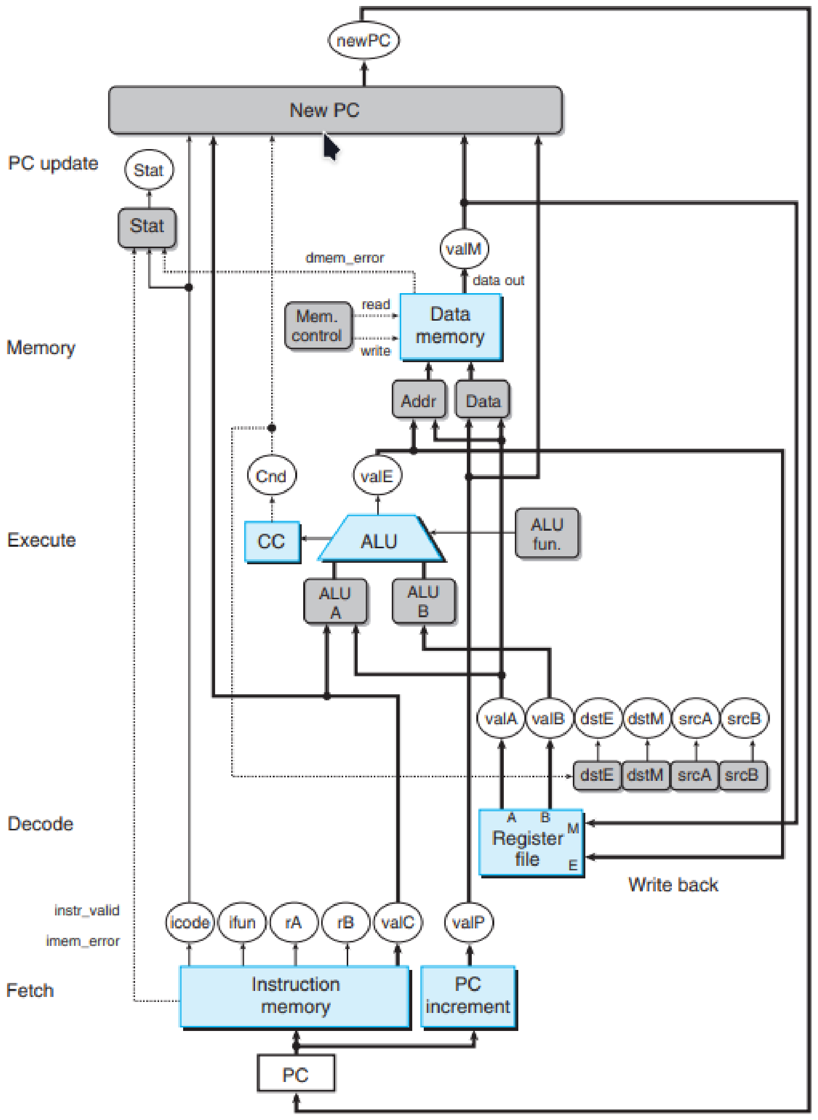
Y86-64 instruction can have a length of 1-10 bytes.

- Byte 0 is divided into two 4-bit blocks called `icode` and `ifun` which give the instruction and the specific function if needed.
- If the instruction requires registers, Byte 1 is divided into two 4-bit blocks called `rA` and `rB` which give the register ids.
- If the instruction requires a constant value (be it a memory address or a number) it is contained in the next 8 bytes.



Byte	0	1	2	3	4	5	6	7	8	9
`halt`	0 0									
`nop`	1 0									
`rrmovq` **rA, rB**	2 0	rA rB								
`irmovq` **V, rB**	3 0	F rB	V							
`rmmovq` **rA, D(rB)**	4 0	rA rB	D							
`mrmovq` **D(rB), rA**	5 0	rA rB	D							
`OPq` **rA, rB**	6 fn	rA rB								
`jXX` **Dest**	7 fn	Dest								
`cmovXX` **rA, rB**	2 fn	rA rB								
`call` **Dest**	8 0	Dest								
`ret`	9 0									
`pushq` **rA**	A 0	rA F								
`popq` **rA**	B 0	rA F								

SEQ : Sequential Y86-64 Implementation



In order to reach the final pipelined version of the Y86-64 processor, we have initially implemented the Sequential version of the processor.

The whole process of a single instruction can be broken down into 6 stages

- Fetch
- Decode
- Execute
- Memory
- Write Back
- PC Update

Fetch

Description

During the fetch stage, the instruction is read from memory. The length of the instruction is 1-10 bytes, depending on the instruction being read.

The instruction is then fed `split` and `align`.

- `split` contains the `icode` and `ifun`.
- `align` contains the rest of the instruction, which could include `rA`, `rB` and `valC`.

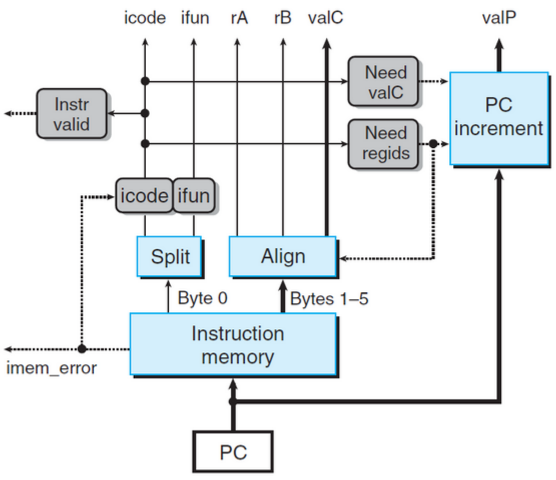
From `split`, we can determine whether to read from `align`, if it contains `rA` and `rB` and also `valC`. Internally,

- `need_valC` tells whether `valC` needs to be read.
- `need_regids` tells whether `rA` and `rB` are needed.

Depending on the instruction, the value of `valP` (the possible PC increment) is updated.

`instr_valid` tells whether the instruction read is valid. If valid, it is set to 1, else it is set to 0.

`imem_error` keeps track of whether the memory address read from fetch stage is valid.



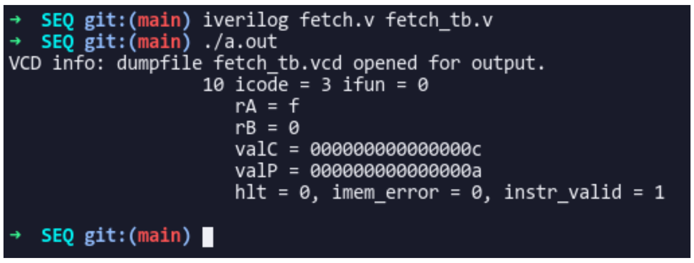
Testbench

```
`timescale 1ns / 1ps

module fetch_tb;

    reg clk = 0;
    reg [63:0] PC = 0;
    output [ 3:0] icode;
    output [ 3:0] ifun;
    output [ 3:0] rA;
    output [ 3:0] rB;
    output [63:0] valC;
    output [63:0] valP;
    output hlt;
    output imem_error;
    output instr_valid;

    fetch uut (clk, PC, icode, ifun, rA, rB, valC, valP, hlt,
imem_error, instr_valid);

    initial begin
        $dumpfile("fetch_tb.vcd");
        $dumpvars(0, fetch_tb);

        #20 $finish;
    end

    always begin
        #10 clk = ~clk;
    end

    always @(posedge clk) begin
        $monitor($time, " icode = %h ifun = %h\n\t\t     rA = %h\n\t\t
  rB = %h\n\t\t     valC = %h\n\t\t     valP = %h\n\t\t     hlt = %d,
imem_error = %d, instr_valid = %d\n", icode, ifun, rA, rB, valC, valP,
hlt, imem_error, instr_valid);
    end

endmodule
```

Results

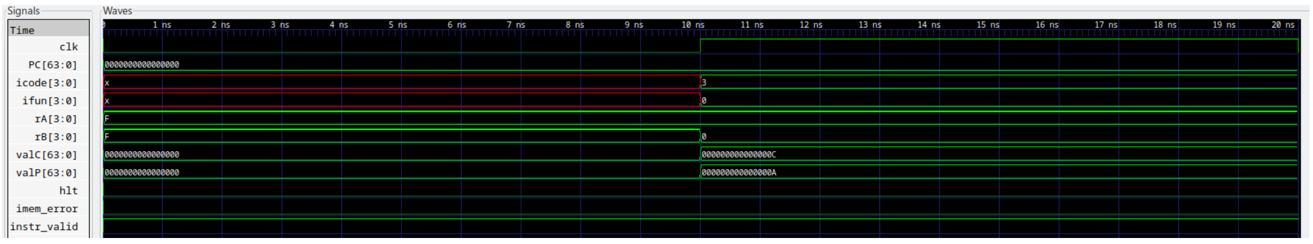
Output of `./a.out`

Instruction given `irmovq $12, %rax`(30 f0 0c 00 00 00 00 00 00 00)



```
→ SEQ git:(main) iverilog fetch.v fetch_tb.v
→ SEQ git:(main) ./a.out
VCD info: dumpfile fetch_tb.vcd opened for output.
                  10 icode = 3 ifun = 0
                     rA = f
                     rB = 0
                     valC = 000000000000000c
                     valP = 000000000000000a
                     hlt = 0, imem_error = 0, instr_valid = 1

→ SEQ git:(main) ▌
```

Output waveform



Decode

Description

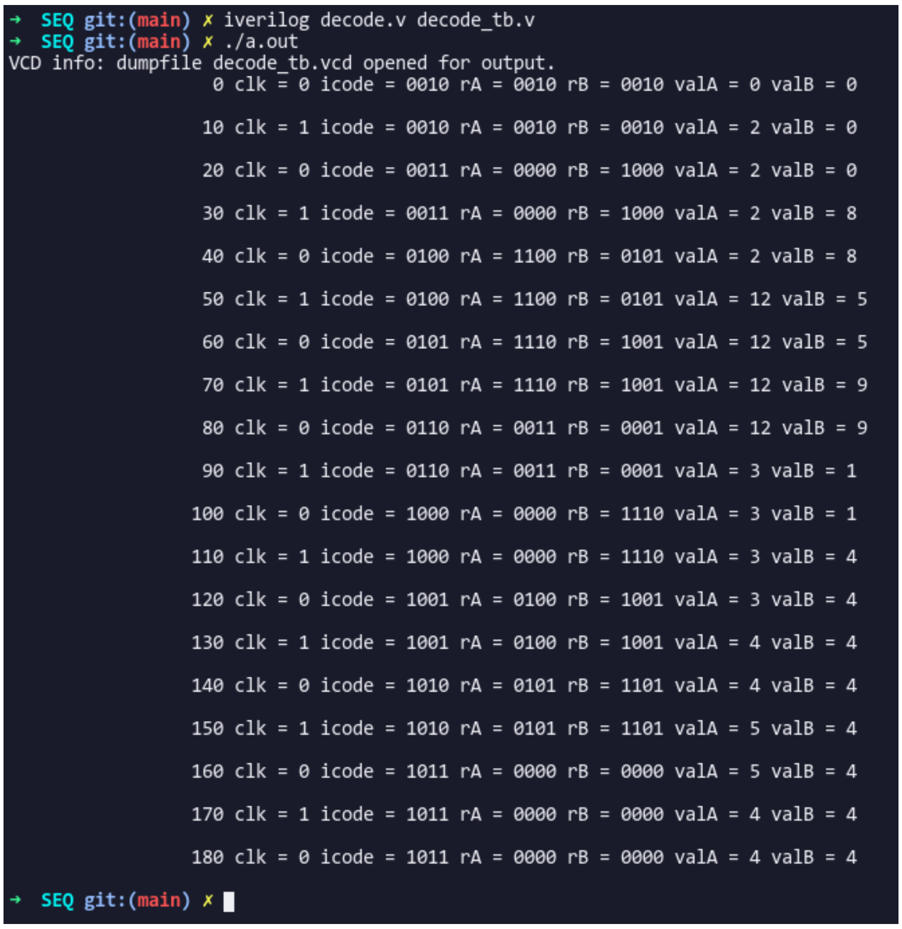
In the decode stage, the values of `valA` and `valB` are read from the registers with addresses `rA`, `rB`, or `%rsp`, depending on the `icode`.

Testbench

```
`timescale 1ns / 1ps

module decode_tb;

    reg clk;
    reg [3:0] icode;
    reg [3:0] rA;
    reg [3:0] rB;
    output [63:0] valA;
    output [63:0] valB;

    decode uut (clk, icode, rA, rB, valA, valB);

    initial begin
        $dumpfile("decode_tb.vcd");
        $dumpvars(0, decode_tb);

        clk=1'b0;
        icode = 4'b0010; rA = 4'b0010; rB = 4'b0010;
        #10 clk = ~clk;
        #10 clk = ~clk;
        icode = 4'b0011; rA = 4'b0000; rB = 4'b1000;
        #10 clk = ~clk;
        #10 clk = ~clk;
        icode = 4'b0100; rA = 4'b1100; rB = 4'b0101;
        #10 clk = ~clk;
        #10 clk = ~clk;
        icode = 4'b0101; rA = 4'b1110; rB = 4'b1001;
        #10 clk = ~clk;
        #10 clk = ~clk;
        icode = 4'b0110; rA = 4'b0011; rB = 4'b0001;
        #10 clk = ~clk;
        #10 clk = ~clk;
        icode = 4'b1000; rA = 4'b0000; rB = 4'b1110;
        #10 clk = ~clk;
        #10 clk = ~clk;
        icode = 4'b1001; rA = 4'b0100; rB = 4'b1001;
        #10 clk = ~clk;
        #10 clk = ~clk;
        icode = 4'b1010; rA = 4'b0101; rB = 4'b1101;
        #10 clk = ~clk;
        #10 clk = ~clk;
        icode = 4'b1011; rA = 4'b0000; rB = 4'b0000;
        #10 clk = ~clk;
        #10 clk = ~clk;
    end

    initial
        $monitor($time, " clk = %d icode = %b rA = %b rB = %b valA = %g
valB = %g\n", clk,icode, rA, rB, valA, valB);

endmodule
```

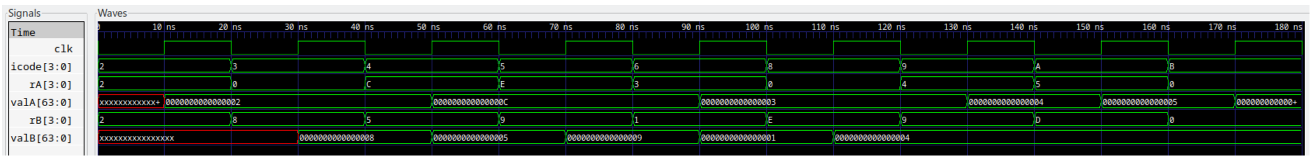
Results

Output of `./a.out`



```
→ SEQ git:(main) ✗ iverilog decode.v decode_tb.v
→ SEQ git:(main) ✗ ./a.out
VCD info: dumpfile decode_tb.vcd opened for output.
                   0 clk = 0 icode = 0010 rA = 0010 rB = 0010 valA = 0 valB = 0

                  10 clk = 1 icode = 0010 rA = 0010 rB = 0010 valA = 2 valB = 0

                  20 clk = 0 icode = 0011 rA = 0000 rB = 1000 valA = 2 valB = 0

                  30 clk = 1 icode = 0011 rA = 0000 rB = 1000 valA = 2 valB = 8

                  40 clk = 0 icode = 0100 rA = 1100 rB = 0101 valA = 2 valB = 8

                  50 clk = 1 icode = 0100 rA = 1100 rB = 0101 valA = 12 valB = 5

                  60 clk = 0 icode = 0101 rA = 1110 rB = 1001 valA = 12 valB = 5

                  70 clk = 1 icode = 0101 rA = 1110 rB = 1001 valA = 12 valB = 9

                  80 clk = 0 icode = 0110 rA = 0011 rB = 0001 valA = 12 valB = 9

                  90 clk = 1 icode = 0110 rA = 0011 rB = 0001 valA = 3 valB = 1

                 100 clk = 0 icode = 1000 rA = 0000 rB = 1110 valA = 3 valB = 1

                 110 clk = 1 icode = 1000 rA = 0000 rB = 1110 valA = 3 valB = 4

                 120 clk = 0 icode = 1001 rA = 0100 rB = 1001 valA = 3 valB = 4

                 130 clk = 1 icode = 1001 rA = 0100 rB = 1001 valA = 4 valB = 4

                 140 clk = 0 icode = 1010 rA = 0101 rB = 1101 valA = 4 valB = 4

                 150 clk = 1 icode = 1010 rA = 0101 rB = 1101 valA = 5 valB = 4

                 160 clk = 0 icode = 1011 rA = 0000 rB = 0000 valA = 5 valB = 4

                 170 clk = 1 icode = 1011 rA = 0000 rB = 0000 valA = 4 valB = 4

                 180 clk = 0 icode = 1011 rA = 0000 rB = 0000 valA = 4 valB = 4

→ SEQ git:(main) ✗ 
```

Output waveform


Signals
Waves
Time
0 10 ns 20 ns 30 ns 40 ns 50 ns 60 ns 70 ns 80 ns 90 ns 100 ns 110 ns 120 ns 130 ns 140 ns 150 ns 160 ns 170 ns 180 ns
clk
icode[3:0] 2 3 4 5 6 8 9 A B
rA[3:0] 2 0 C E 3 0 4 5 0
valA[63:0] xxxxxxxxxxxx+ 0000000000000002 000000000000000C 0000000000000003 0000000000000004 0000000000000005 000000000000+
rB[3:0] 2 8 5 9 1 E 9 D 0
valB[63:0] xxxxxxxxxxxxxxxx 0000000000000008 0000000000000005 0000000000000009 0000000000000001 0000000000000004

Execute

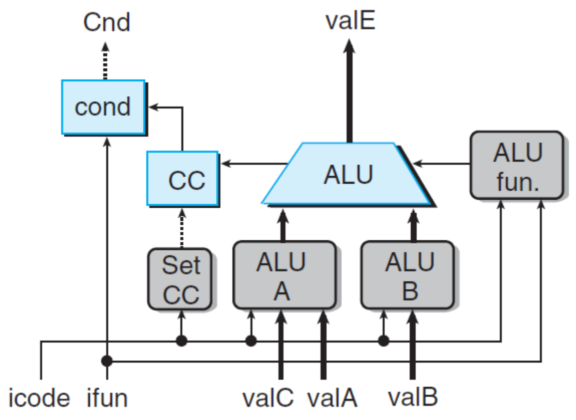
Description

In the execute stage, the operations `ADD`, `SUBTRACT`, `AND` or `EXCLUSIVE-OR` are performed on `aluA` and `aluB`, depending on the values of `icode` and `ifun`.

`icode` and `ifun` sets the value of `alufun`.

During this stage, the condition codes are also set, depending on the instruction. The output of this stage is `valE`, and `Cnd`, which depends on the condition codes. The value of `Cnd` is assigned as follows depending on the values of `ifun`, `ZF`, `OF` and `SF`.

```
assign Cnd =
(ifun == Branch_UNC) |
(ifun == Branch_LE & ((ZF ^ OF) | ZF)) |
(ifun == Branch_L & (SF ^ OF)) |
(ifun == Branch_E & ZF) |
(ifun == Branch_NE & ~ZF) |
(ifun == Branch_GE & (~SF ^ OF)) |
(ifun == Branch_G & (~SF ^ OF) & ~ZF)
```

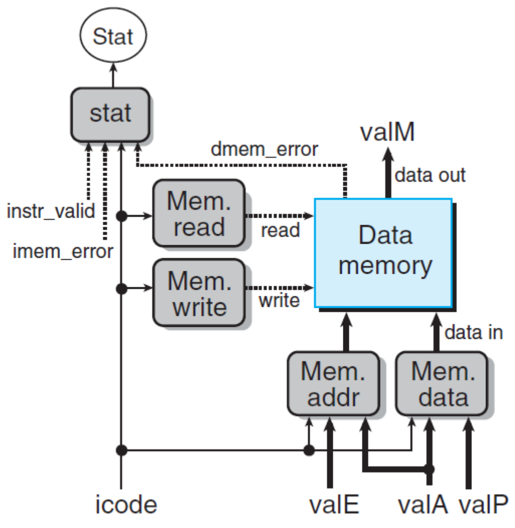


Memory

Description

In the memory stage, values are read from or written into the main memory, depending on the value of `icode`. If values are read, the values are outputted as `valM`. If values are to be written, then depending on the `icode`, `valA` or `valP` is written into the memory location with address `valE`.

`mem_write` tells whether values are to be written into memory, while `mem_read` tells whether values are to be written into memory.



Testbench

```
module memory_tb();

reg clk;
reg [3:0] icode;
reg [63:0] valA;
reg [63:0] valB;
reg [63:0] valE;
reg [63:0] valP;
output [63:0] valM;
output dmem_error;

memory memory(
     .clk(clk),
     .icode(icode),
     .valA(valA),
     .valB(valB),
     .valE(valE),
     .valP(valP),
     .dmem_error(dmem_error),
     .valM(valM)
   );
```

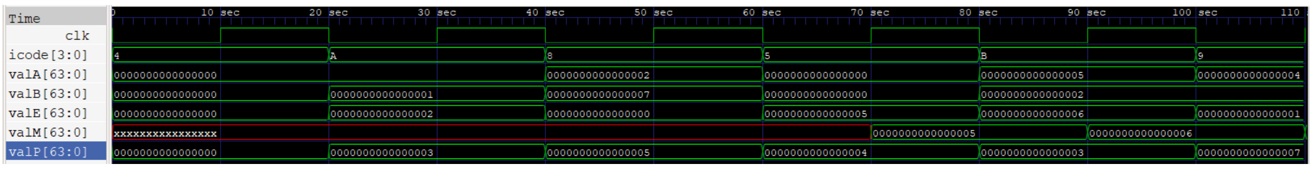
```
initial begin

    $dumpfile("memory_tb.vcd");
    $dumpvars(0, memory_tb);
    clk=1'b0;

    icode = 4'b0100; valA = 64'd0; valB = 64'd0; valE = 64'd0; valP =
64'd0;
    #10 clk = ~clk;
    #10 clk = ~clk;
    icode = 4'b1010; valA = 64'd0; valB = 64'd1; valE = 64'd2; valP =
64'd3;
    #10 clk = ~clk;
    #10 clk = ~clk;
    icode = 4'b1000; valA = 64'd2; valB = 64'd7; valE = 64'd0; valP =
64'd5;
    #10 clk = ~clk;
    #10 clk = ~clk;
    icode = 4'b0101; valA = 64'd0; valB = 64'd0; valE = 64'd5; valP =
64'd4;
    #10 clk = ~clk;
    #10 clk = ~clk;
    icode = 4'b1011; valA = 64'd5; valB = 64'd2; valE = 64'd6; valP =
64'd3;
    #10 clk = ~clk;
    #10 clk = ~clk;
    icode = 4'b1001; valA = 64'd4; valB = 64'd2; valE = 64'd1; valP =
64'd7;
    #10 clk = ~clk;

end

initial
    $monitor("clk = %d icode = %b valA = %g valB = %g valE = %g valP =
%g valM = %g\n",clk,icode,valA,valB,valE,valP,valM);

endmodule
```

Results

Output of `./a.out`

```
clk = 1 icode = 1011 valA = 5 valB = 2 valE = 6 valP = 3 valM = 6

clk = 0 icode = 1001 valA = 4 valB = 2 valE = 1 valP = 7 valM = 6

clk = 1 icode = 1001 valA = 4 valB = 2 valE = 1 valP = 7 valM = 4
```

Output waveform

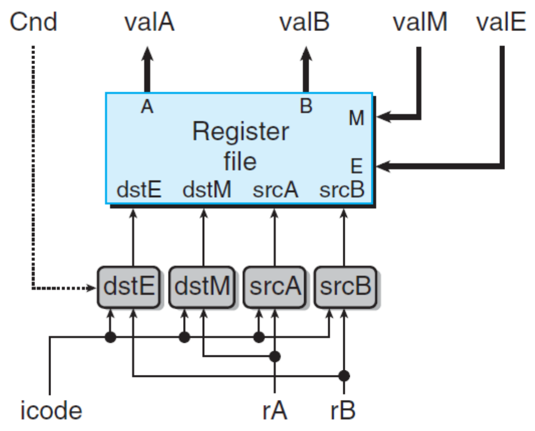


Write Back

Description

During the write back stage, values are written back into either of the 15 main registers.
The values `valE`, or `valM` are written into the registers with addresses `rA`, `rB` or `%rsp` depending on the value of `icode`.

The decode and write back stages can be combined into one as follows,



Testbench

```
module write_back_tb();

reg clk;
reg [3:0] icode;
reg [3:0] rA;
reg [3:0] rB;
reg [63:0] valA;
reg [63:0] valB;
reg [63:0] valE;
reg [63:0] valM;
```

```
output [63:0] dstE;
output [63:0] dstM;

write_back write_back(
    .clk(clk),
    .icode(icode),
    .rA(rA),
    .rB(rB),
    .valA(valA),
    .valB(valB),
    .valE(valE),
    .valM(valM),
    .dstE(dstE),
    .dstM(dstM)
  );

initial begin

    $dumpfile("write_back_tb.vcd");
        $dumpvars(0, write_back_tb);
    clk=1'b0;

    icode = 4'b0010; rA = 4'b0010; rB = 4'b0110; valA = 64'd3; valB =
64'd18; valE = 64'd140; valM = 64'd16;
    #10 clk = ~clk;
    #10 clk = ~clk;
    icode = 4'b0011; rA = 4'b0000; rB = 4'b0010; valA = 64'd16; valB =
64'd20; valE = 64'd10; valM = 64'd0;
    #10 clk = ~clk;
    #10 clk = ~clk;
    icode = 4'b0110; rA = 4'b1010; rB = 4'b1100; valA = 64'd2; valB =
64'd16; valE = 64'd0; valM = 64'd1;
    #10 clk = ~clk;
    #10 clk = ~clk;
    icode = 4'b1000; rA = 4'b1110; rB = 4'b0000; valA = 64'd57; valB =
64'd17; valE = 64'd0; valM = 64'd20;
    #10 clk = ~clk;
    #10 clk = ~clk;
    icode = 4'b1001; rA = 4'b0000; rB = 4'b0110; valA = 64'd23; valB =
64'd11; valE = 64'd0; valM = 64'd17;
    #10 clk = ~clk;
    #10 clk = ~clk;
    icode = 4'b1010; rA = 4'b0001; rB = 4'b0011; valA = 64'd0; valB =
64'd9; valE = 64'd10; valM = 64'd10;
    #10 clk = ~clk;
    #10 clk = ~clk;
    icode = 4'b1011; rA = 4'b0010; rB = 4'b0010; valA = 64'd15; valB =
64'd7; valE = 64'd40; valM = 64'd5;
    #10 clk = ~clk;
```

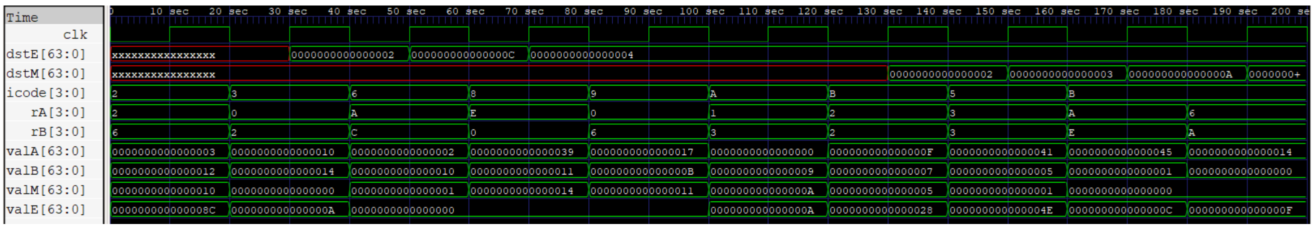
```
        #10 clk = ~clk;
        icode = 4'b0101; rA = 4'b0011; rB = 4'b0011; valA = 64'd65; valB =
64'd5; valE = 64'd78; valM = 64'd1;
        #10 clk = ~clk;
        #10 clk = ~clk;
        icode = 4'b1011; rA = 4'b1010; rB = 4'b1110; valA = 64'd69; valB =
64'd1; valE = 64'd12; valM = 64'd0;
        #10 clk = ~clk;
        #10 clk = ~clk;
        icode = 4'b1011; rA = 4'b0110; rB = 4'b1010; valA = 64'd20; valB =
64'd0; valE = 64'd15; valM = 64'd0;
        #10 clk = ~clk;
        #10 clk = ~clk;

end

initial
        $monitor("clk=%d icode=%b rA=%b rB=%b valA=%g valB=%g valE=%g
valM=%g dstE=%g
dstM=%g\n",clk,icode,rA,rB,valA,valB,valE,valM,dstE,dstM);
endmodule
```

Results

Output of `./a.out`

```
clk=0 icode=1011 rA=0110 rB=1010 valA=20 valB=0 valE=15 valM=0 dstE=4 dstM=10

clk=1 icode=1011 rA=0110 rB=1010 valA=20 valB=0 valE=15 valM=0 dstE=4 dstM=6

clk=0 icode=1011 rA=0110 rB=1010 valA=20 valB=0 valE=15 valM=0 dstE=4 dstM=6
```

Output waveform

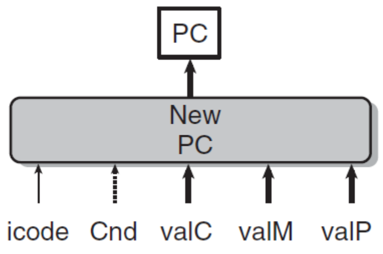

Time
clk
dstE[63:0] xxxxxxxxxxxxxxxx 0000000000000002 000000000000000C 0000000000000004
dstM[63:0] xxxxxxxxxxxxxxxx 0000000000000002 0000000000000003 000000000000000A 0000000+
icode[3:0] 2 3 6 8 9 A B 5 B
rA[3:0] 2 0 A E 0 1 2 3 A 6
rB[3:0] 6 2 C 0 6 3 2 3 E A
valA[63:0] 0000000000000003 0000000000000010 0000000000000002 0000000000000039 0000000000000017 0000000000000000 000000000000000F 0000000000000041 0000000000000045 0000000000000014
valB[63:0] 0000000000000012 0000000000000014 0000000000000010 0000000000000011 000000000000000B 0000000000000009 0000000000000007 0000000000000005 0000000000000001 0000000000000000
valM[63:0] 0000000000000010 0000000000000000 0000000000000001 0000000000000014 0000000000000011 000000000000000A 0000000000000005 0000000000000001 0000000000000000
valE[63:0] 000000000000008C 000000000000000A 0000000000000000 000000000000000A 0000000000000028 000000000000004E 000000000000000C 000000000000000F
10 sec 20 sec 30 sec 40 sec 50 sec 60 sec 70 sec 80 sec 90 sec 100 sec 110 sec 120 sec 130 sec 140 sec 150 sec 160 sec 170 sec 180 sec 190 sec 200 se

PC Update

Description

The PC update stage calculates the value of `new_pc`, the value of `PC` for the start of the next instruction. The value of `new_pc` depends on the values of `icode`, `valC`, `valM`, `valP` and `Cnd`.

After all the 6 stages are over, the fetch stage of the next instruction occurs and the cycle continues until the `HALT` instruction is encountered.

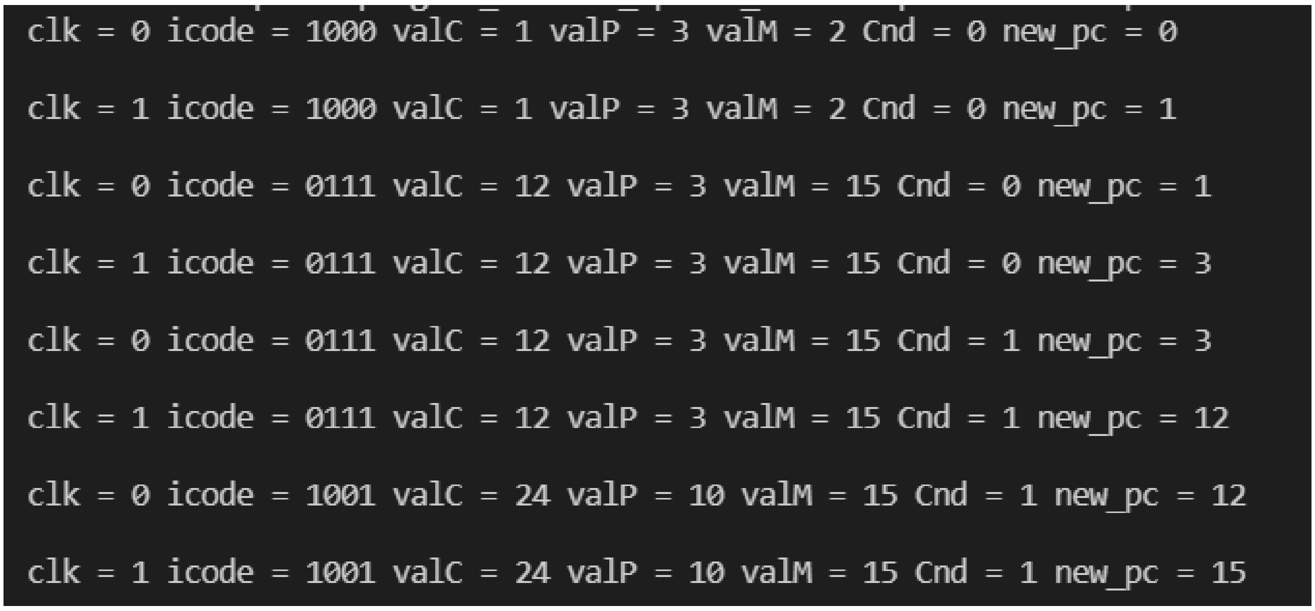


Testbench

```
module pc_update_tb();

reg clk;
reg [3:0] icode;
reg Cnd;
reg [63:0] PC;
reg [63:0] valC;
reg [63:0] valP;
reg [63:0] valM;
output [63:0] new_pc;

program_counter_update program_counter_update(
    .clk(clk),
    .PC(PC),
    .Cnd(Cnd),
    .icode(icode),
    .valC(valC),
    .valM(valM),
    .valP(valP),
    .new_pc(new_pc)
  );

initial begin

    $dumpfile("program_counter_update_tb.vcd");
        $dumpvars(0, pc_update_tb);
    clk=1'b0;
    PC = 64'h0000;
    icode = 4'b1000; valC = 64'd1; valM = 64'd2; valP = 64'd3; Cnd =
1'b0;
    #10 clk = ~clk;
    #10 clk = ~clk;
    icode = 4'b0111; valC = 64'd12; valM = 64'd15; valP = 64'd3; Cnd =
1'b0;
    #10 clk = ~clk;
    #10 clk = ~clk;
    icode = 4'b0111; valC = 64'd12; valM = 64'd15; valP = 64'd3; Cnd =
1'b1;
    #10 clk = ~clk;
    #10 clk = ~clk;
    icode = 4'b1001; valC = 64'd24; valM = 64'd15; valP = 64'd10; Cnd =
1'b1;
    #10 clk = ~clk;

end

initial
    $monitor("clk = %d icode = %b valC = %g valP = %g valM = %g Cnd = %g
new_pc = %g\n",clk,icode,valC,valP,valM,Cnd,new_pc);

endmodule
```

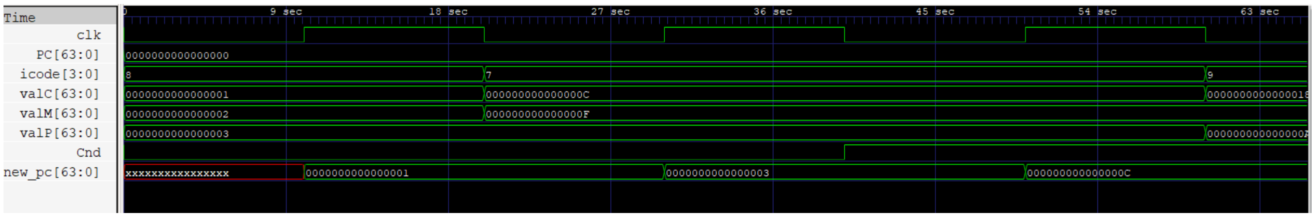
Results

Output of `./a.out`



```
clk = 0 icode = 1000 valC = 1 valP = 3 valM = 2 Cnd = 0 new_pc = 0

clk = 1 icode = 1000 valC = 1 valP = 3 valM = 2 Cnd = 0 new_pc = 1

clk = 0 icode = 0111 valC = 12 valP = 3 valM = 15 Cnd = 0 new_pc = 1

clk = 1 icode = 0111 valC = 12 valP = 3 valM = 15 Cnd = 0 new_pc = 3

clk = 0 icode = 0111 valC = 12 valP = 3 valM = 15 Cnd = 1 new_pc = 3

clk = 1 icode = 0111 valC = 12 valP = 3 valM = 15 Cnd = 1 new_pc = 12

clk = 0 icode = 1001 valC = 24 valP = 10 valM = 15 Cnd = 1 new_pc = 12

clk = 1 icode = 1001 valC = 24 valP = 10 valM = 15 Cnd = 1 new_pc = 15
```

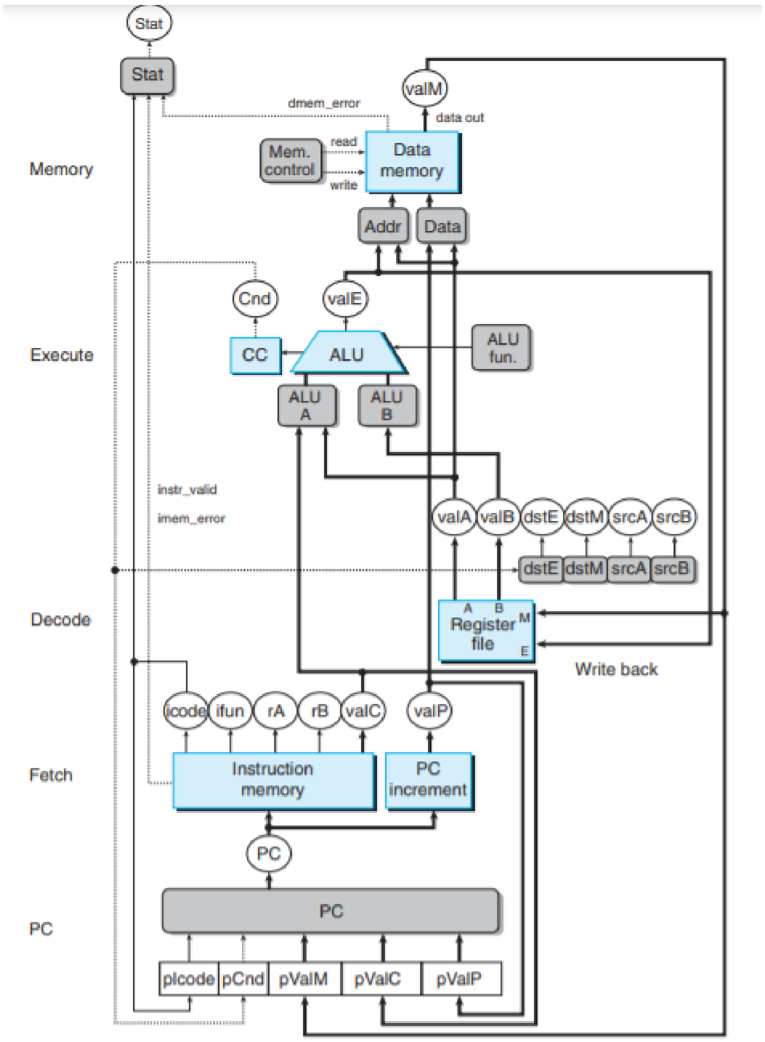
Output waveform



PIPE : 5-Staged Pipelined Y86-64

SEQ+

In order to start the implementation of pipelined processor, we start by rearranging the stages of SEQ. This is done by moving the PC update stage before Fetch stage. This new reordering is called SEQ+.



PIPE-

Next, using what we know about pipelining, we divide the whole of SEQ+ into different blocks with intermediate registers in between. They are as follows:

F

- Before the Fetch stage.
- Holds `predPC` (the predicted program counter)

D

- Lies between Fetch and Decode stages.
- Holds information about currently fetched instruction to be used by decode stage.

E

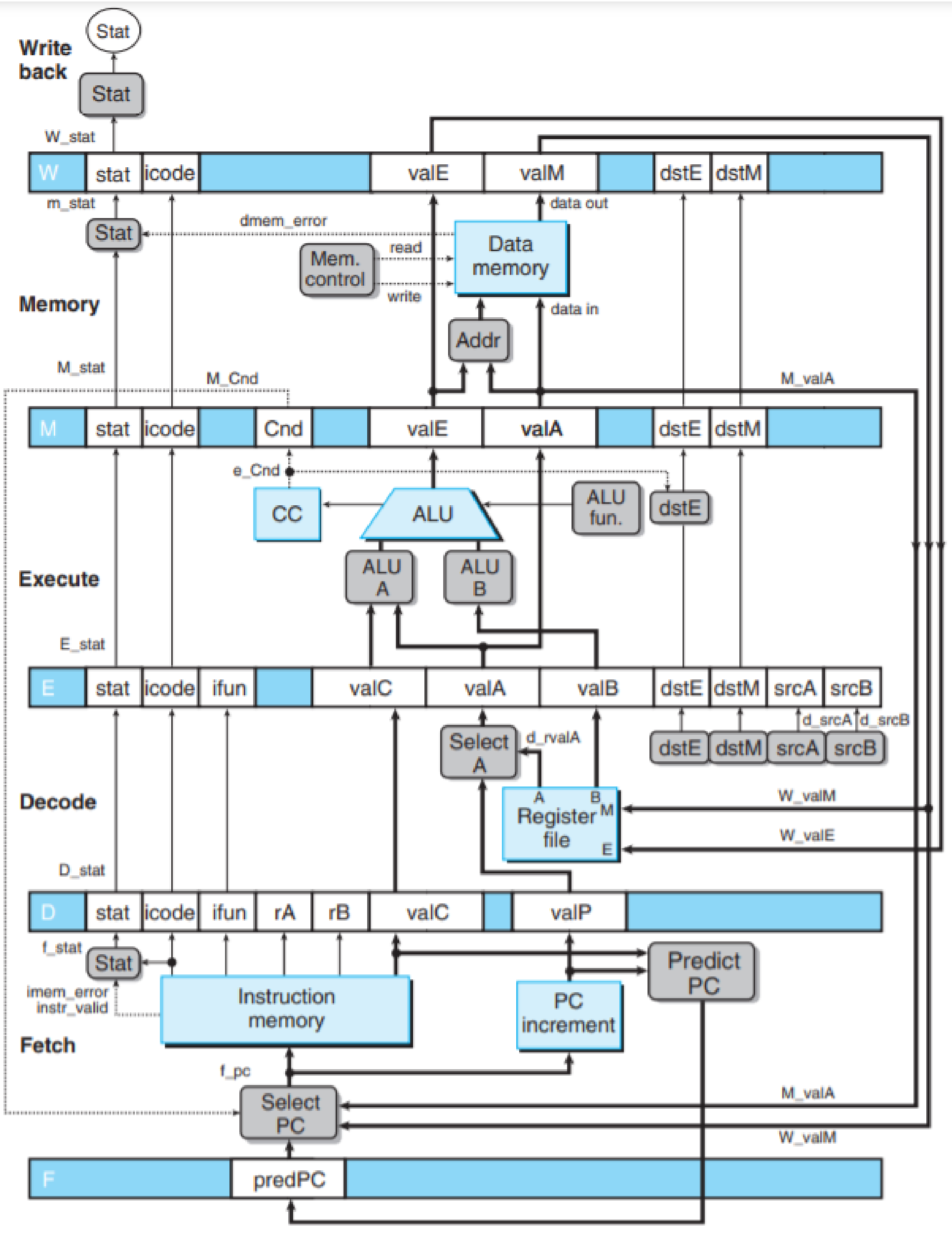
- Lies between Decode and Execute stages.
- Holds information about currently decoded instruction to be used by execute stage.

M

- Lies between Execute and Memory stages.
- Holds information about currently executed instruction to be used by memory stage.

W

- Lies between Memory and Writeback stages.
- It is connected to 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.



Verilog Code

```
module f_reg (
    clk,
    predPC,
    f_pc
);
    input              clk;
    input       [63:0] predPC;
    output reg  [63:0] f_pc;

    always @(posedge clk) begin
        f_pc <= predPC;
    end
endmodule

module d_reg (
    clk,
    f_stat, f_icode, f_ifun, f_rA, f_rB, f_valC, f_valP,
    d_stat, d_icode, d_ifun, d_rA, d_rB, d_valC, d_valP
);
    input             clk;

    input      [ 2:0] f_stat;
    input      [ 3:0] f_icode, f_ifun, f_rA, f_rB;
    input      [63:0] f_valC, f_valP;

    output reg [ 2:0] d_stat;
    output reg [ 3:0] d_icode, d_ifun, d_rA, d_rB;
    output reg [63:0] d_valC, d_valP;

    always @(posedge clk) begin
        d_stat  <= f_stat;
        d_icode <= f_icode;
        d_ifun  <= f_ifun;
        d_rA    <= f_rA;
        d_rB    <= f_rB;
        d_valC  <= f_valC;
        d_valP  <= f_valP;
    end
endmodule

module e_reg (
    clk,
    d_stat, d_icode, d_ifun, d_valC, d_valA, d_valB, d_dstE, d_dstM,
d_srcA, d_srcB,
    e_stat, e_icode, e_ifun, e_valC, e_valA, e_valB, e_dstE, e_dstM,
e_srcA, e_srcB
);
    input clk;

    input      [ 2:0] d_stat;
    input      [ 3:0] d_icode, d_ifun;
    input      [63:0] d_valC;
    input      [63:0] d_valA, d_valB;
    input      [ 3:0] d_dstE, d_dstM, d_srcA, d_srcB;

    output reg [ 2:0] d_stat;
    output reg [ 3:0] d_icode, d_ifun;
    output reg [63:0] d_valC;
    output reg [63:0] d_valA, d_valB;
    output reg [ 3:0] d_dstE, d_dstM, d_srcA, d_srcB;

    always @(posedge clk) begin
        d_stat  <= e_stat;
        d_icode <= e_icode;
```

```verilog
        d_ifun  <= e_ifun;
        d_valC  <= e_valC;
        d_valA  <= e_valA;
        d_valB  <= e_valB;
        d_dstE  <= e_dstE;
        d_dstM  <= e_dstM;
        d_srcA  <= e_srcA;
        d_srcB  <= e_srcB;
    end
endmodule

module m_reg (
    clk,
    e_stat, e_icode, e_Cnd, e_valE, e_valA, e_dstE, e_dstM,
    m_stat, m_icode, m_Cnd, m_valE, m_valA, m_dstE, m_dstM
);
    input clk;

    input      [ 2:0] e_stat;
    input      [ 3:0] e_icode;
    input             e_Cnd;
    input      [63:0] e_valE, e_valA;
    input      [ 3:0] e_dstE, e_dstM;

    output reg [ 2:0] m_stat;
    output reg [ 3:0] m_icode;
    output reg        m_Cnd;
    output reg [63:0] m_valE, m_valA;
    output reg [ 3:0] m_dstE, m_dstM;

    always @(posedge clk) begin
        e_stat  <= m_stat;
        e_icode <= m_icode;
        e_Cnd   <= m_Cnd;
        e_valE  <= m_valE;
        e_valA  <= m_valA;
        e_dstE  <= m_dstE;
        e_dstM  <= m_dstM;
    end
endmodule

module w_reg (
    clk,
    m_stat, m_icode, m_valE, m_valM, m_dstE, m_dstM,
    w_stat, w_icode, w_valE, w_valM, w_dstE, w_dstM
);
    input clk;

    input      [ 2:0] m_stat;
    input      [ 3:0] m_icode;
    input      [63:0] m_valE, m_valM;
    input      [ 3:0] m_dstE, m_dstM;

    output reg [ 2:0] w_stat;
    output reg [ 3:0] w_icode;
    output reg [63:0] w_valE, w_valM;
    output reg [ 3:0] w_dstE, w_dstM;

    always @(posedge clk) begin
        m_stat  <= w_stat;
        m_icode <= w_icode;
        m_valE  <= w_valE;
        m_valM  <= w_valM;
        m_dstE  <= w_dstE;
        m_dstM  <= w_dstM;
    end
endmodule
```

Hazards

This involves handling the hazards that come along with pipelining (data hazards and control hazards).

To handle these hazards we use the following methods.

1. Stall to avoid data hazards
2. Forwarding to avoid data hazards
3. Stall + Forwarding to avoid load/use hazards
4. Bubbling to avoid control hazards.
5. etc.

PIPE

PC Prediction Strategy

The prediction strategy for `predPC` can be broadly divided into four criterias.

1. For instructions that don't transfer control - Predict next PC to be `valP`. This is ALWAYS reliable since no other values are possible for `predPC`.
2. Call and unconditional jumps - Predict next PC to be `valC`. This again is always reliable since no other values are possible for `predPC`.
3. Conditional jumps - Predict next PC to be `valC`. This prediction will only come out to be correct if a branch is taken.
4. Return instruction - Don't try to predict.

Control for Return, mispredicted branches and load/use hazard

1. The return instruction can be detected using the condition given below,

```
IRET in { D_icode, E_icode, M_icode}
```

2. Branch misprediction can be detected using the condition given below,

```
E_icode = IJXX & !e_Cnd
```

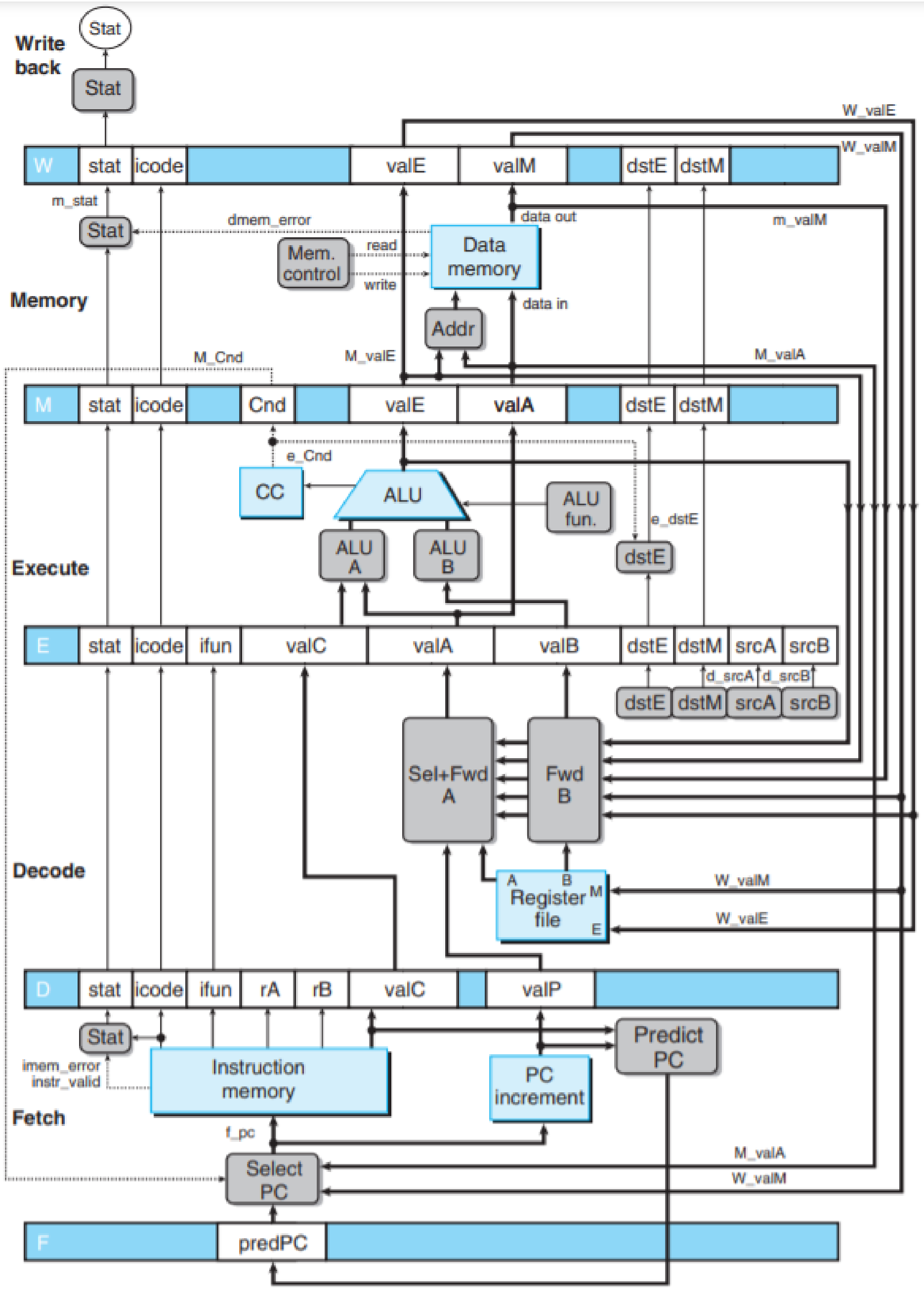
3. Load/use hazard can be detected using the condition given below,

```
E_icode in { IRMOVQ, IPOPQ } && E_dstM in { d_srcA, d_srcB }
```

The above three can be controlled using the scheme given below for each stage,

Condition	F	D	E	M	W
Processing `ret`	stall	bubble	normal	normal	normal
Load/Use Hazard	stall	stall	bubble	normal	normal
Mispredicted Branch	normal	bubble	bubble	normal	normal

Furthermore, combinations of these control cases are also taken care of.



Stat
Write back
Stat
W_valE
W_valM
W
stat
icode
valE
valM
dstE
dstM
m_stat
dmem_error
data out
m_valM
Stat
Mem. control
read
write
Data memory
Memory
data in
Addr
M_Cnd
M_valE
M_valA
M
stat
icode
Cnd
valE
valA
dstE
dstM
e_Cnd
CC
ALU
ALU fun.
e_dstE
Execute
ALU A
ALU B
dstE
E
stat
icode
ifun
valC
valA
valB
dstE
dstM
srcA
srcB
d_srcA
d_srcB
dstE
dstM
srcA
srcB
Sel+Fwd A
Fwd B
Decode
A
B
M
Register file
E
W_valM
W_valE
D
stat
icode
ifun
rA
rB
valC
valP
Stat
Predict PC
imem_error
instr_valid
Instruction memory
PC increment
Fetch
f_pc
Select PC
M_valA
W_valM
F
predPC