Y86 Architecture

Singularity

Table of Contents

1. Introduction

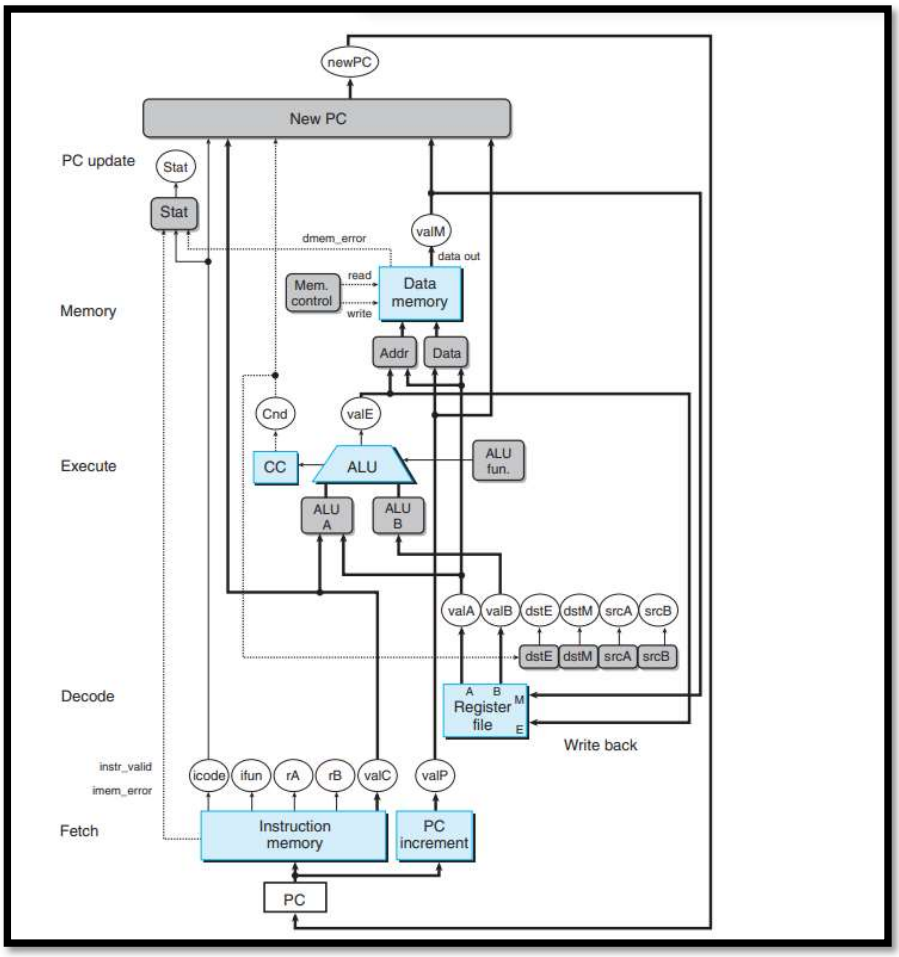
Y86-64 bits is an instruction set architecture derived from Intel's X86 architecture. Y86 architecture has intermediate level of complexity which falls between RISC and CISC. Differences between X86 and Y86 are listed below:

	Y86 architecture	X86 architecture
1	Fewer data types, instructions and addressing modes.	More data types, instructions and addressing modes.
2	Simple byte-level encoding.	Complex instruction encoding.
3	Less compact machine code and easy decoding logic.	Highly compact machine code and complex decoding.
4	15 program registers with 16^{th} register redundant.	All 16 registers are functions.

2. Sequential

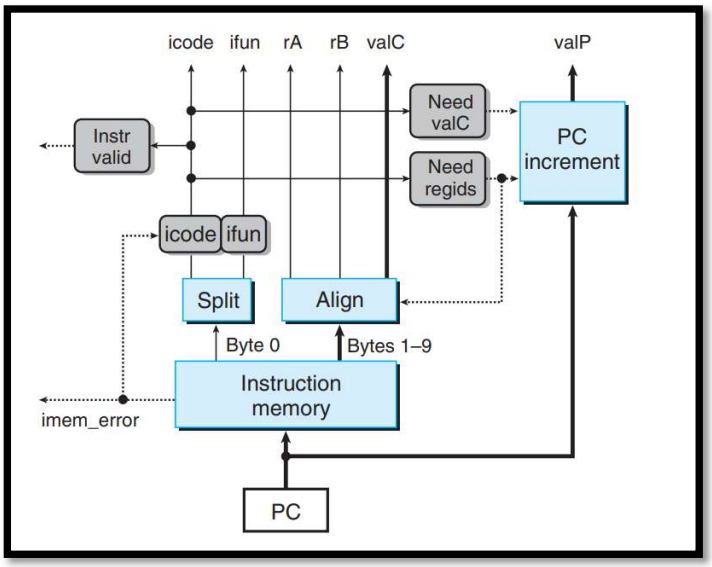
Sequential implementation performs all the instruction and serves as a first step towards making a full-fledged pipelined processor.

Sequential implementation consists of 6 blocks, namely *fetch, decode, execute, memory, write-back and PC-update*.



a. Fetch Stage:

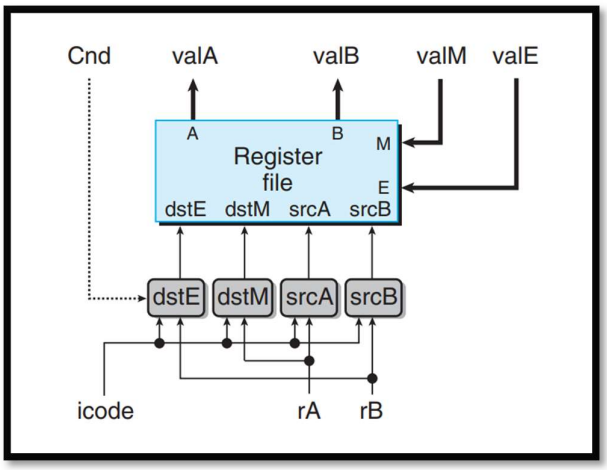
In fetch stage we, take PC value as input and derive instructions from ‘instruction memory’. Depending upon the instruction value we obtain values for ‘icode’ & ‘ifun’. Again depending upon instruction we may also obtain ‘rA’ & ‘rB’ and ‘valC’ (immediate value) and ‘valP’ (incremented PC). To follow the ‘Little Endian’ convention in Y86-64 we perform ‘Split’ and ‘Align’. Also we need to specify whether received instruction is correct or not, so we have an instr_valid flag which is ‘1’ in normal operation. We also have to check whether PC value id correct, so for that too we define a flag ‘imem_error’.



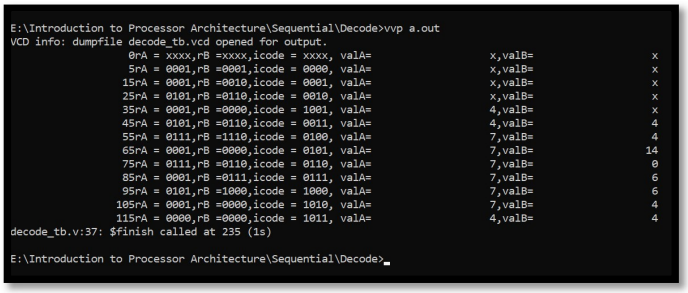

```
E:\Introduction to Processor Architecture\Sequential\Fetch>iverilog fetch_tb.v fetch.v

E:\Introduction to Processor Architecture\Sequential\Fetch>vvp a.out
VCD info: dumpfile fetch_tb.vcd opened for output.
pc=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx,icode=xxxx,ifun=xxxx,ra=xxxx,rb=xxxx,valc=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xx,valp=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx,inst_valid=x,imem_error=x
pc=0000000000000000000000000000000000000000000000000000000000001010,icode=0110,ifun=0000,ra=1010,rb=0000,valc=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xx,valp=0000000000000000000000000000000000000000000000000000000000001100,inst_valid=1,imem_error=0
pc=0000000000000000000000000000000000000000000000000000000000001100,icode=0110,ifun=0000,ra=1000,rb=0111,valc=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xx,valp=0000000000000000000000000000000000000000000000000000000000001110,inst_valid=1,imem_error=0
pc=0000000000000000000000000000000000000000000000000000000000001110,icode=0110,ifun=0001,ra=1001,rb=0110,valc=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xx,valp=0000000000000000000000000000000000000000000000000000000000010000,inst_valid=1,imem_error=0
pc=0000000000000000000000000000000000000000000000000000000000010000,icode=1001,ifun=0000,ra=1001,rb=0110,valc=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xx,valp=0000000000000000000000000000000000000000000000000000000000010001,inst_valid=1,imem_error=0
pc=0000000000000000000000000000000000000000000000000000000000010001,icode=0100,ifun=0000,ra=0110,rb=0100,valc=0101101001011010010110100101101001011010010110100101101001011010
10,valp=0000000000000000000000000000000000000000000000000000000000011011,inst_valid=1,imem_error=0
fetch_tb.v:45: $finish called at 165 (1s)

E:\Introduction to Processor Architecture\Sequential\Fetch>gtkwave fetch_tb.vcd

GTKWave Analyzer v3.3.108 (w)1999-2020 BSI

[0] start time.
[165] end time.
```

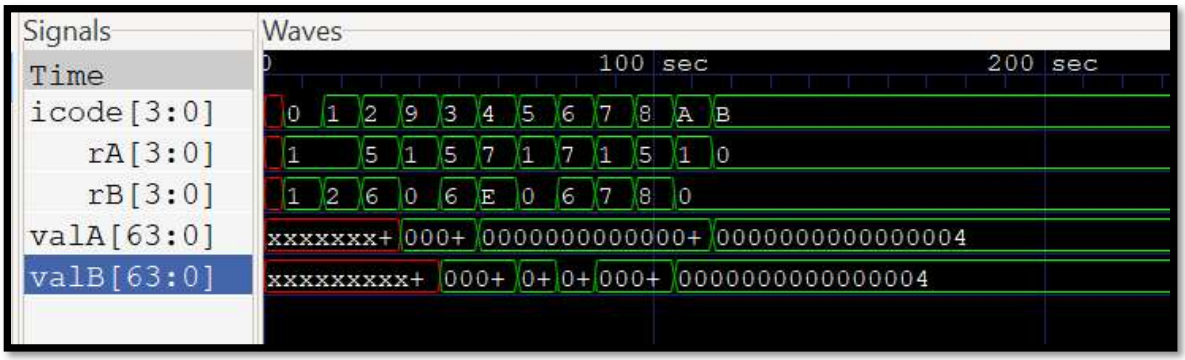
b. Decode:

We have implemented the decode stage by creating individual modules for srcA, srcB and register file. The register file is also accessed by decode thus image contains both decode and writeback stages. However, we have implemented them separately. srcA and srcB take either rA, rB or 4 as its value depends on instruction specified by icode. srcA, srcB are register identifiers (IDs) which specifies which register is to be accessed out of 15 registers (16 in X86). In Y86-64, 16th register is set to 'F' and is never used.



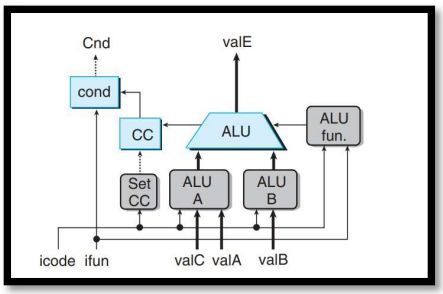
Output:

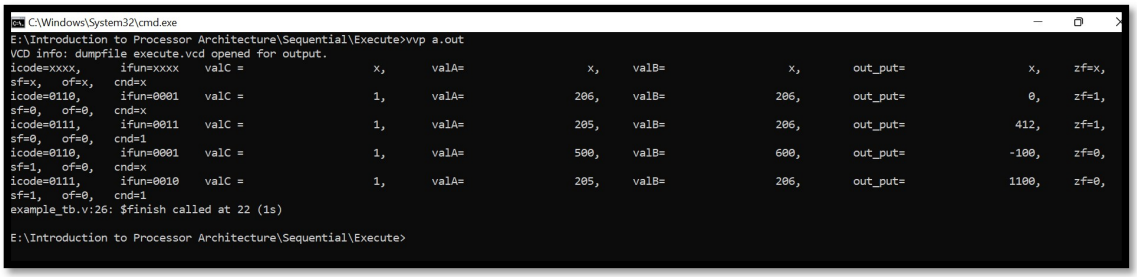


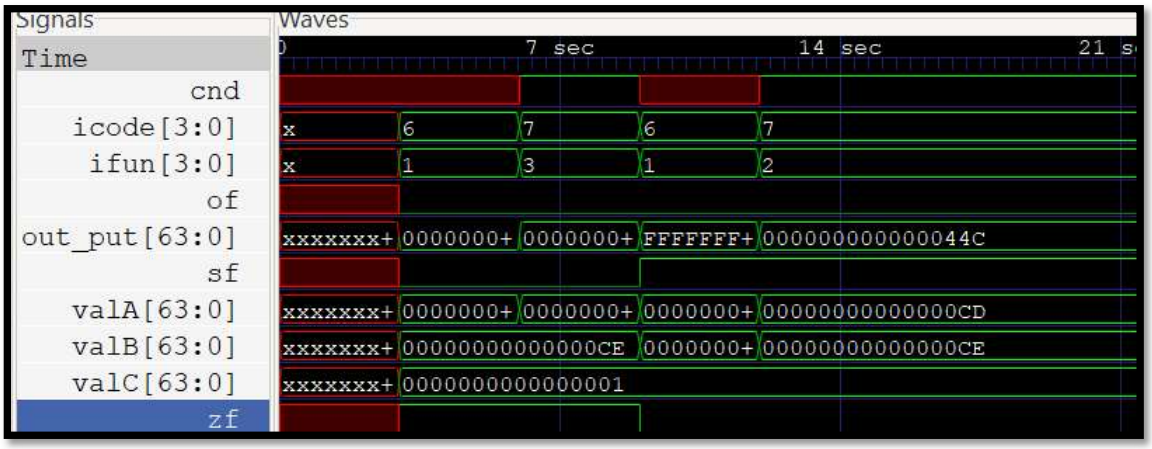


c. Execute:

In this stage we either compute some operation or we verify the conditions for cmov or jmp instructions. If set_cc is ‘1’, we perform arithmetic operation, otherwise verify the condition and set CND flag accordingly. ALU_A and ALU_B take (valC or valA) and valB values respectively depeding upon ‘icode’ value. ALU_fun block specifies the exact operation which is to be performed by ALU in case of operation mode. We made ALU block with structural modelling previously, it computes values depending upon contrl_signal.

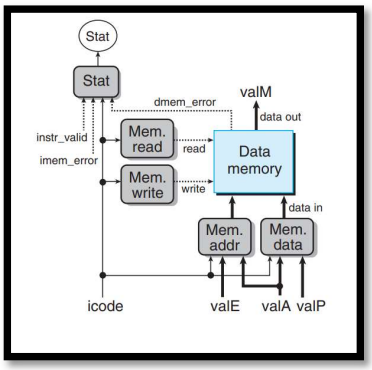




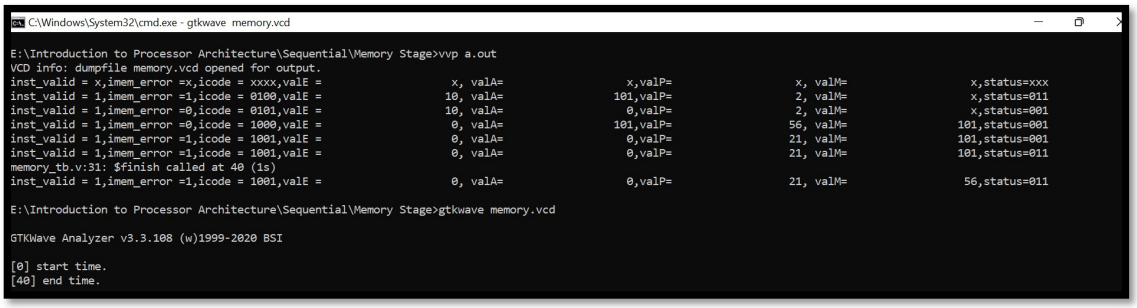


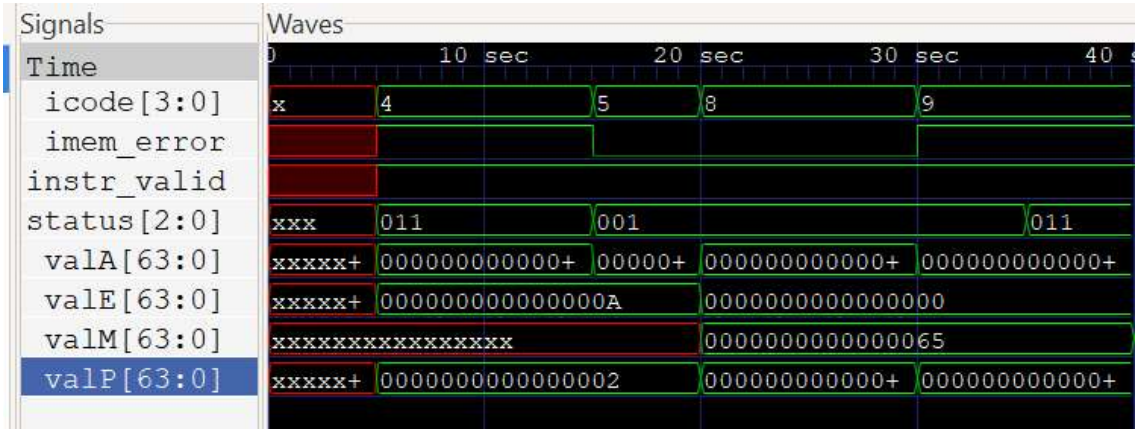
Memory:

We have implemented each block visible in the below figure. ‘Memory address’ and ‘Memory data’ blocks provide the address and data respectively. That address and data can be used for read or write. This signals for read and write are provided by ‘Memory read’ and ‘Memory write’ depending upon icode value. Data memory is 65536 bytes of storage device which gives signal valM whevever read is ‘1’. It also gives out dmem_error which specifies whether memory address is valid or not. The last module is of status which gives out 2 bit output setting the status code (AOK, ADR, INS, HLT) for processor.



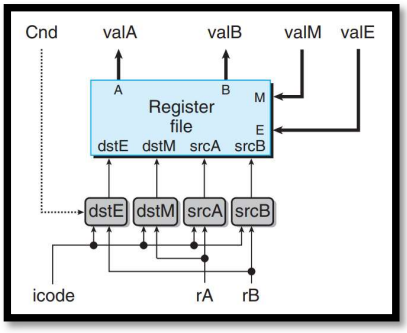
Output:





d. Write back:

In write_back stage, we store the values of valM and valE into the registers present in register file. Write back is performed only for specific values of icode, based on which we define the logic for ‘dstE’ and ‘dstM’ . Similar to ‘srcA’ and ‘srcB’ in decode stage, here we have ‘dstE’ and ‘dstM’ which are register identifiers (IDs) and specify the register in which we want to write back. So, write_back performs write operation while decode performed read operations. Unlike decode, writeback stage has CND input (coming from execute stage). CND is ‘1’ if condition for cmov or jmp instructions is satisfied. rB is assigned to dstE if CND is ‘1’.

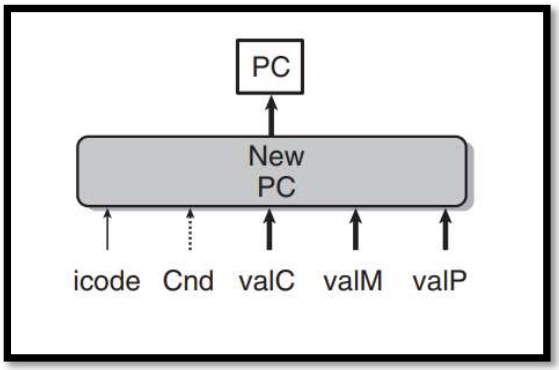


Output:

```
5rA=4'b0001; rB=4'b0000; icode=4'b0000; valM=64'b1010; valE=64'b1111; rcx=x rax=x
10rA=4'b0001; rB=4'b0000; icode=4'b0001; valM=64'b1010; valE=64'b1111; rcx=x rax=x
15rA=4'b0001; rB=4'b0000; icode=4'b0010; valM=64'b1010; valE=64'b1111; rcx=x rax=x
20rA=4'b0001; rB=4'b0000; icode=4'b0011; valM=64'b1010; valE=64'b1111; rcx=x rax=9
25rA=4'b0001; rB=4'b0000; icode=4'b0100; valM=64'b1010; valE=64'b1111; rcx=x rax=9
30rA=4'b0001; rB=4'b0000; icode=4'b0101; valM=64'b1010; valE=64'b1111; rcx=7 rax=9
35rA=4'b0001; rB=4'b0110; icode=4'b0110; valM=64'b1010; valE=64'b0010; rcx=7 rax=2
40rA=4'b0001; rB=4'b0000; icode=4'b0111; valM=64'b1010; valE=64'b1111; rcx=7 rax=2
45rA=4'b0001; rB=4'b0000; icode=4'b1000; valM=64'b1010; valE=64'b1111; rcx=7 rax=2
50rA=4'b0001; rB=4'b0000; icode=4'b1001; valM=64'b1010; valE=64'b1111; rcx=7 rax=2
55rA=4'b0001; rB=4'b0000; icode=4'b1010; valM=64'b1010; valE=64'b1111; rcx=7 rsp=1
60rA=4'b0001; rB=4'b0000; icode=4'b1011; valM=64'b1010; valE=64'b1111; rcx=7 rsp=0
```

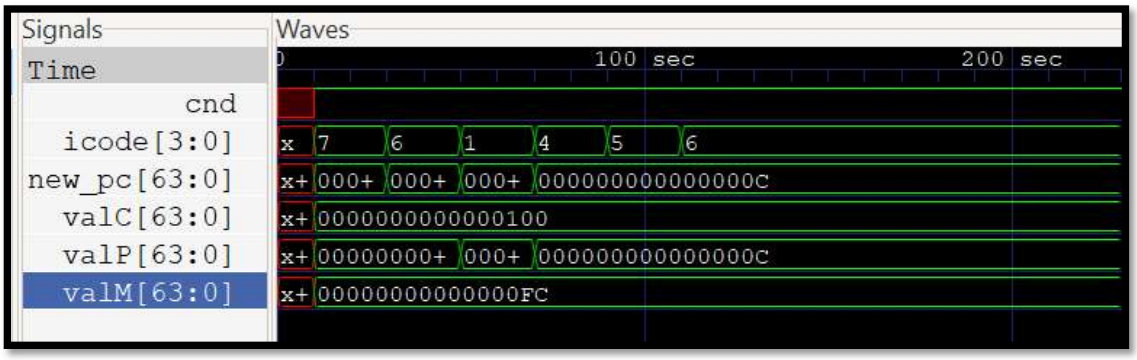
e. PC Update:

In this stage we increment the PC value, so that processor starts processing the next instruction in 'instruction memory'. In this block we give 'icode', 'CND', 'valC', 'valM', and 'valP' depending on which we decide whether we need to increment the PC or not. In case of a satified jump instruction, PC should contain address where we need to jump. Based of such requirements we build this stage.



Output:

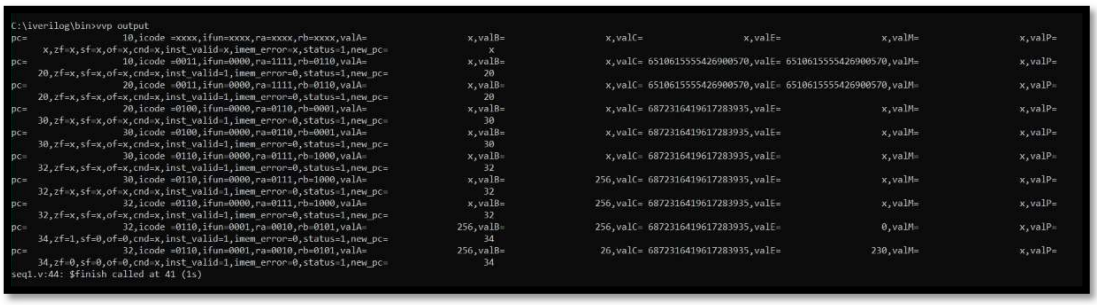
```
E:\Introduction to Processor Architecture\Sequential\PC Update>iverilog pcupdate_tb.v pcupdate.v

E:\Introduction to Processor Architecture\Sequential\PC Update>vvp a.out
VCD info: dumpfile pcupdate.vcd opened for output.
icode=xxxx,     cnd=x,    valC=                x,   valM=                x,     valP=               x,     new_pc=               x
icode=0111,     cnd=1,    valC=              256,   valM=              252,     valP=              12,     new_pc=             256
icode=0110,     cnd=1,    valC=              256,   valM=              252,     valP=              12,     new_pc=              12
icode=0001,     cnd=1,    valC=              256,   valM=              252,     valP=               1,     new_pc=               1
icode=0100,     cnd=1,    valC=              256,   valM=              252,     valP=              12,     new_pc=              12
icode=0101,     cnd=1,    valC=              256,   valM=              252,     valP=              12,     new_pc=              12
icode=0110,     cnd=1,    valC=              256,   valM=              252,     valP=              12,     new_pc=              12
pcupdate_tb.v:33: $finish called at 230 (1s)

E:\Introduction to Processor Architecture\Sequential\PC Update>gtkwave pcupdate.vcd

GTKWave Analyzer v3.3.108 (w)1999-2020 BSI
```

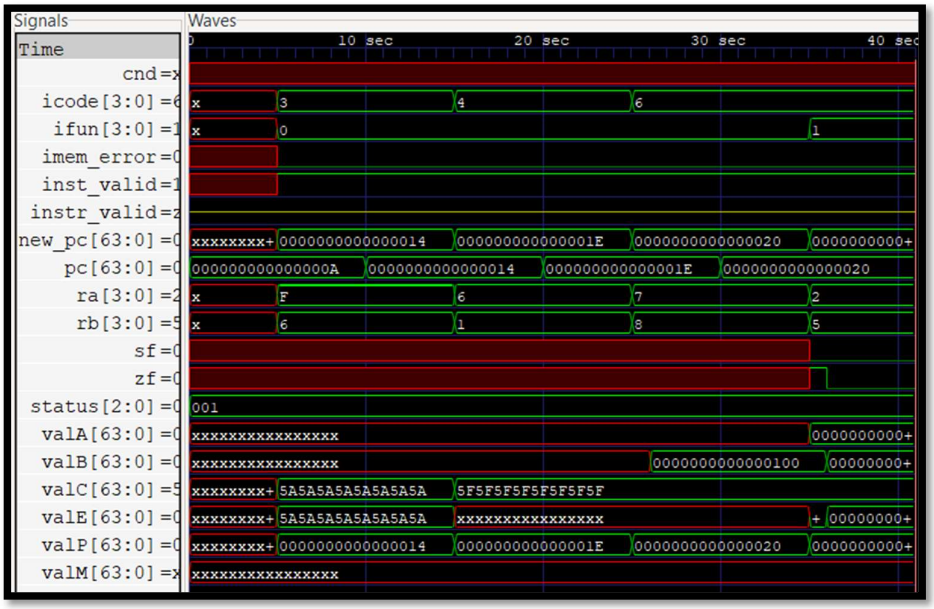


Integration:

We instantiate the above stages in a single module to form a 'Sequential Hardware' for Y86-64. We give the PC value and monitor the outputs of individual stages.

Output:

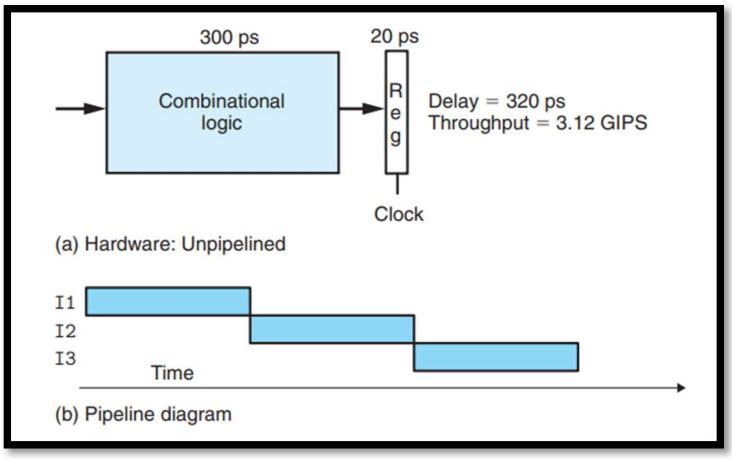


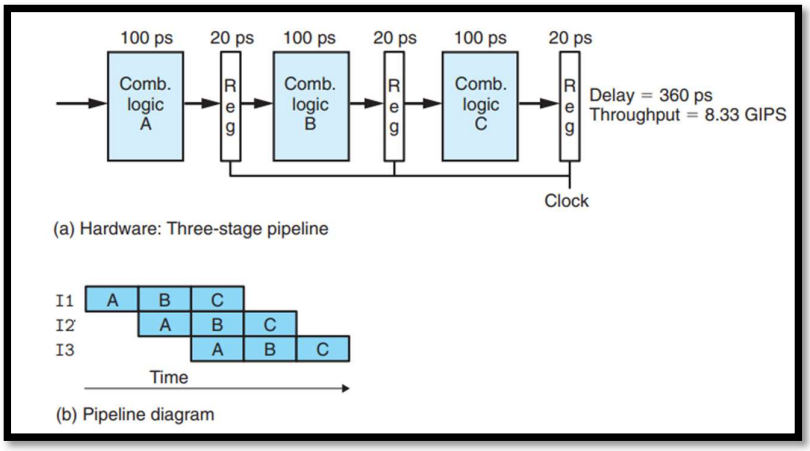


3. Pipelining

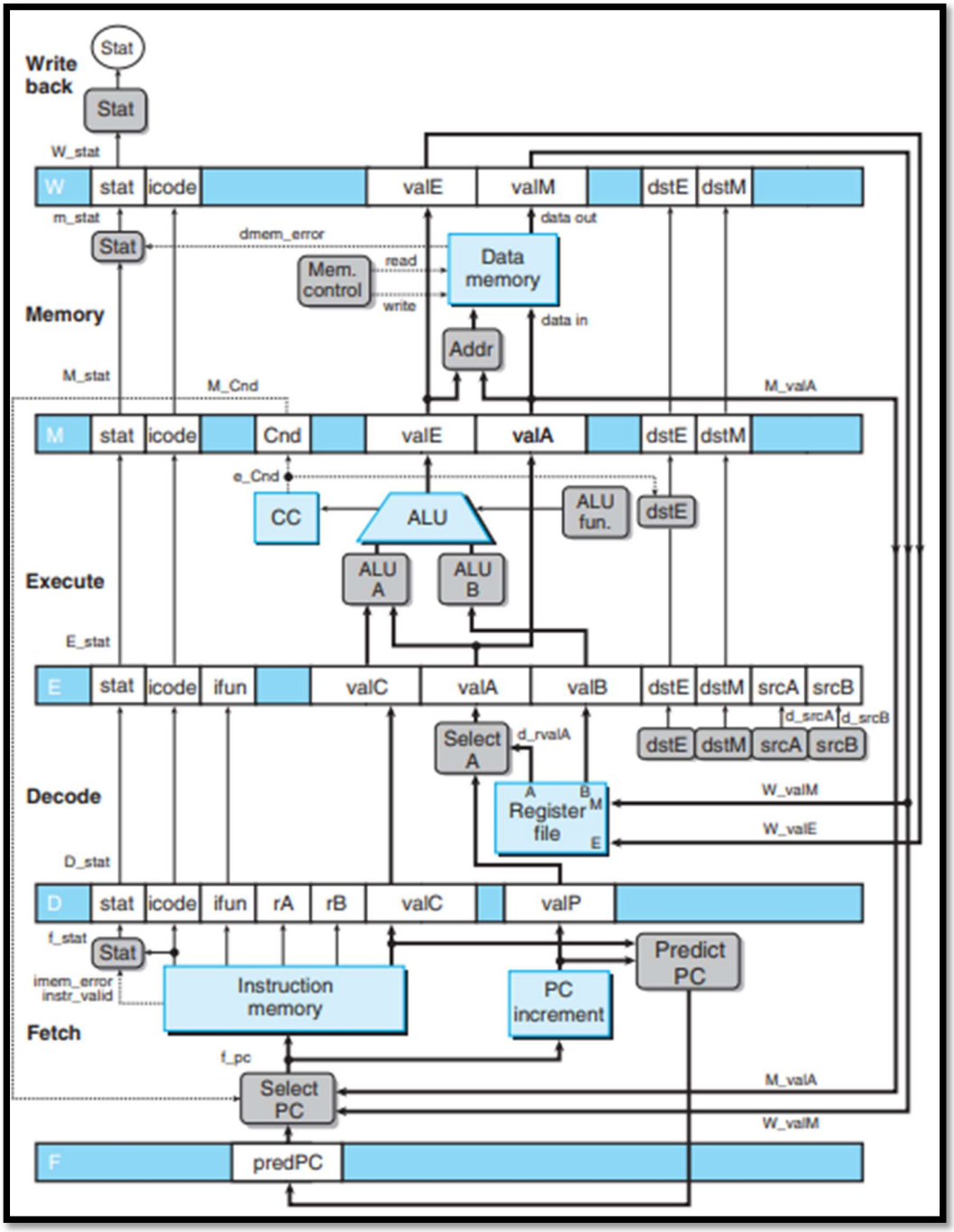
In a pipelined system, task to be performed is divided into series of discrete stages. Pipelining increases throughput of the system however it may increase the latency.

	Pipelined system	Sequential system
1	Multiple instructions processed at same time.	One instruction processed at a single time.
2	Registers contribute to latency.	No pipelined registers involved.
3	High throughput and latency.	Lower throughput and latency.
4	Complex logic and hardware design.	Simpler logic and hardware design.





Upgrading sequential to pipelined system by combining Fetch and PC update stage and inserting registers between individual stages.



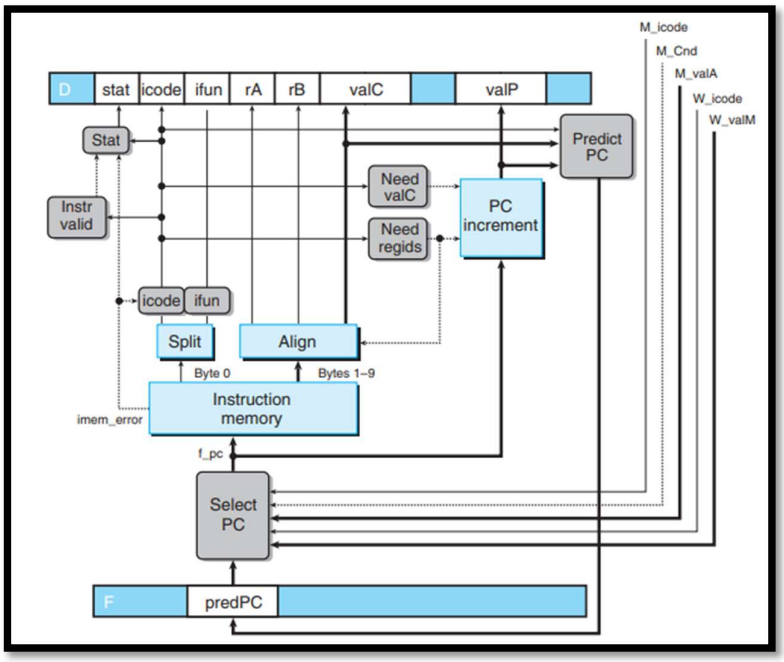
Note that the registers are triggered at positive clock edge. After this we upgrade individual stages. With our naming system, the uppercase prefixes ‘D’, ‘E’, ‘M’, and ‘W’ refer to

pipeline registers, and so M_stat refers to the status code field of pipeline register M. The lowercase prefixes ‘f’, ‘d’, ‘e’, ‘m’, and ‘w’ refer to the pipeline stages, and so m_stat refers to the status signal generated in the memory stage by a control logic block.

a. Fetch

We initiate the processor by providing an initial value to the PC (predPC) after which processor will run in a loop until halt instruction has arrived or status code changes to anything other than AOK. The selected PC value is used to obtain instruction from instruction memory and this instruction is decoded to get some values. Depending upon the current instruction, pipeline control logic and instructions further in the loop, we decide the PC value. The output from this stage is stored in D pipeline register.

This stage is of utmost important when instructions may take branch. We initially assume that a branch will be taken and thus set the next PC value to valC instead of valP.



Code:

```
module fetch (predpc,W_icode,M_icode,M_Cnd,W_valM,M_valA,E_valA,e_cnd,e_icode,
              icode,
              ifun,
              ra,
              rb,
              valc,
              valp,
              status,
              nextpredpc);

input [63:0] predpc;
input [63:0] M_valA,W_valM;
input [3:0] W_icode,M_icode;
input M_Cnd;
input e_cnd;
input [63:0] E_valA;
input [3:0] e_icode;
output reg [2:0] status;
output reg [3:0] icode,ifun,ra,rb;
output reg [63:0] valc;
output reg [63:0] valp;
output reg [63:0] nextpredpc;
reg inst_valid;
reg imem_error;
reg [63:0] pc;
```

```
reg [7:0] split;
reg [7:0] align;
integer i;
reg [7:0] inst_mem[65535:0];


 initial begin
 //instruction memory will be included here
 end



always @ (*)
   begin
   if (W_icode==4'b1001)
      pc=W_valM;

   else if(M_icode==4'b0111 && M_Cnd==1'b0)
      pc=M_valA;


   else
      pc=predpc;

   end
always @ (pc)
  begin

    split = inst_mem[pc];
    icode = split[3:0];
    ifun = split[7:4];

    if((icode ==4'b0000) && (ifun == 4'b0000))  //Halt
      begin
      valp = pc+1;
      inst_valid = 1'b1;
      end
    else if((icode ==4'b0001) && (ifun == 4'b0000))   //nop
      begin
      valp = pc+1;
      inst_valid = 1'b1;
      end

    else if(icode == 4'b0010)     //cmovXX
      begin

       align[7:0]=inst_mem[pc+1];
       ra=align[3:0];
       rb=align[7:4];
```

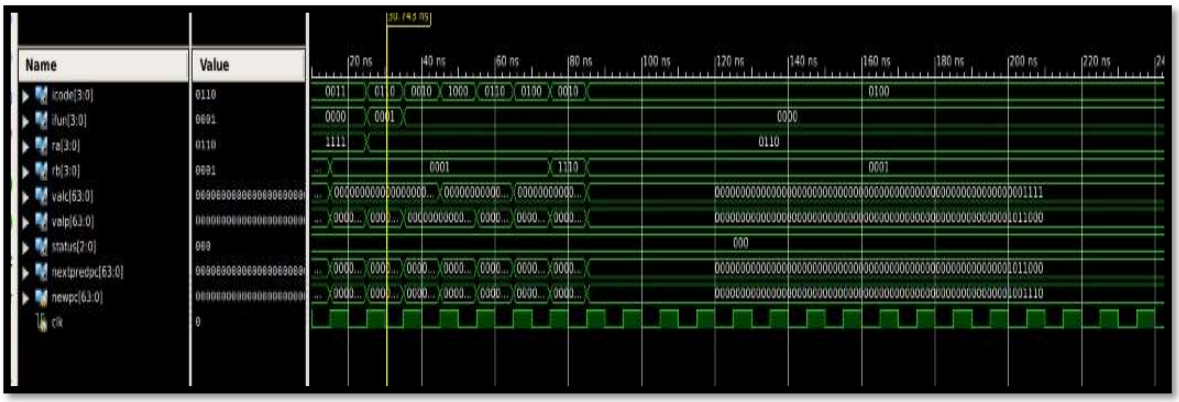
```
      valp=pc+2;
      inst_valid=1'b1;

    end

  else if((icode == 4'b0011)&&(ifun == 4'b0000))   //irmovq
    begin
    align[7:0]=inst_mem[pc+1];
    ra=align[3:0];
    rb=align[7:4];
    valc[7:0]<=inst_mem[pc+2];
    valc[15:8]<=inst_mem[pc+3];
    valc[23:16]<=inst_mem[pc+4];
    valc[31:24]<=inst_mem[pc+5];
    valc[39:32]<=inst_mem[pc+6];
    valc[47:40]<=inst_mem[pc+7];
    valc[55:48]<=inst_mem[pc+8];
    valc[63:56]<=inst_mem[pc+9];
    valp=pc+10;
    inst_valid=1'b1;
    end

  else if((icode == 4'b0100)&&(ifun == 4'b0000))   //rmmovq
    begin
    align[7:0]=inst_mem[pc+1];
    ra=align[3:0];
    rb=align[7:4];
    valc[7:0]<=inst_mem[pc+2];
    valc[15:8]<=inst_mem[pc+3];
    valc[23:16]<=inst_mem[pc+4];
    valc[31:24]<=inst_mem[pc+5];
    valc[39:32]<=inst_mem[pc+6];
    valc[47:40]<=inst_mem[pc+7];
    valc[55:48]<=inst_mem[pc+8];
    valc[63:56]<=inst_mem[pc+9];
    valp=pc+10;
    inst_valid=1'b1;
    end


  else if((icode == 4'b0101)&&(ifun == 4'b0000))   //mrmovq
    begin
    align[7:0]=inst_mem[pc+1];
    ra=align[3:0];
    rb=align[7:4];
    valc[7:0]<=inst_mem[pc+2];
    valc[15:8]<=inst_mem[pc+3];
    valc[23:16]<=inst_mem[pc+4];
```

```
    valc[31:24]<=inst_mem[pc+5];
    valc[39:32]<=inst_mem[pc+6];
    valc[47:40]<=inst_mem[pc+7];
    valc[55:48]<=inst_mem[pc+8];
    valc[63:56]<=inst_mem[pc+9];
    valp=pc+10;
    inst_valid=1'b1;
    end

  else if(icode == 4'b0110)   //opq
    begin
     if(ifun==4'b0000)       //addq
      begin
      align[7:0]=inst_mem[pc+1];
      ra=align[3:0];
      rb=align[7:4];
      valp=pc+2;
      inst_valid=1'b1;
      end
     else if(ifun==4'b0001)   //subq
      begin
      align[7:0]=inst_mem[pc+1];
      ra=align[3:0];
      rb=align[7:4];
      valp=pc+2;
      inst_valid=1'b1;
      end
     else if(ifun==4'b0010)   //andq
      begin
      align[7:0]=inst_mem[pc+1];
      ra=align[3:0];
      rb=align[7:4];
      valp=pc+2;
      inst_valid=1'b1;
      end
     else if(ifun==4'b0011)   //xorq
      begin
      align[7:0]=inst_mem[pc+1];
      ra=align[3:0];
      rb=align[7:4];
      valp=pc+2;
      inst_valid=1'b1;
      end
    end


  else if(icode==4'b0111)   //jXX
    begin
```

```
    valc[7:0]<=inst_mem[pc+1];
    valc[15:8]<=inst_mem[pc+2];
    valc[23:16]<=inst_mem[pc+3];
    valc[31:24]<=inst_mem[pc+4];
    valc[39:32]<=inst_mem[pc+5];
    valc[47:40]<=inst_mem[pc+6];
    valc[55:48]<=inst_mem[pc+7];
    valc[63:56]<=inst_mem[pc+8];
    valp =pc+9;
    inst_valid=1'b1;
    end

  else if((icode==4'b1000)&&(ifun==4'b0000))   //call
    begin
    valc[7:0]<=inst_mem[pc+1];
    valc[15:8]<=inst_mem[pc+2];
    valc[23:16]<=inst_mem[pc+3];
    valc[31:24]<=inst_mem[pc+4];
    valc[39:32]<=inst_mem[pc+5];
    valc[47:40]<=inst_mem[pc+6];
    valc[55:48]<=inst_mem[pc+7];
    valc[63:56]<=inst_mem[pc+8];
    valp= pc+9;
    inst_valid=1'b1;
    end

  else if((icode==4'b1001)&&(ifun==4'b0000))   //ret
    begin
    valp = pc+1;
    inst_valid = 1'b1;
    end

  else if((icode==4'b1010)&&(ifun==4'b0000))   //pushq
    begin
    align[7:0]=inst_mem[pc+1];
    ra=align[3:0];
    rb=align[7:4];
    valp=pc+2;
    inst_valid=1'b1;
    end

  else if((icode==4'b1011)&&(ifun==4'b0000))   //popq
    begin
    align[7:0]=inst_mem[pc+1];
    ra=align[3:0];
    rb=align[7:4];
    valp=pc+2;
    inst_valid=1'b1;
```

```
      end

    else
      inst_valid=1'b0;

    if (pc>65535)
       imem_error=1'b1;
    else
       imem_error=1'b0;

    if(inst_valid==1'b0)
       status=3'b011;              //INS  3'b011

    else if(imem_error==1'b1)
       status=3'b010;             //ADR  3'b010

    else if(icode==4'b0000)
       status=3'b100;               //HLT 3'b100
    else
       status=3'b000;              //AOK 3'b000

    end
    always@(*)
    begin
    if ((icode==4'b0001) || (icode==4'b0010) ||(icode==4'b0011)
||(icode==4'b0100) ||(icode==4'b0101 )||(icode==4'b0110) ||(icode==4'b1001)
||(icode==4'b1010 )||(icode==4'b1011))
      nextpredpc = valp;
    if ((icode==4'b0111 )||(icode==4'b1000))
      nextpredpc=valc;


   if((e_icode==4'b0111)&&(e_cnd==1'b0))
      nextpredpc=E_valA;

   if(W_icode==4'b000)
      nextpredpc=W_valM;

    end
endmodule
```

Output:



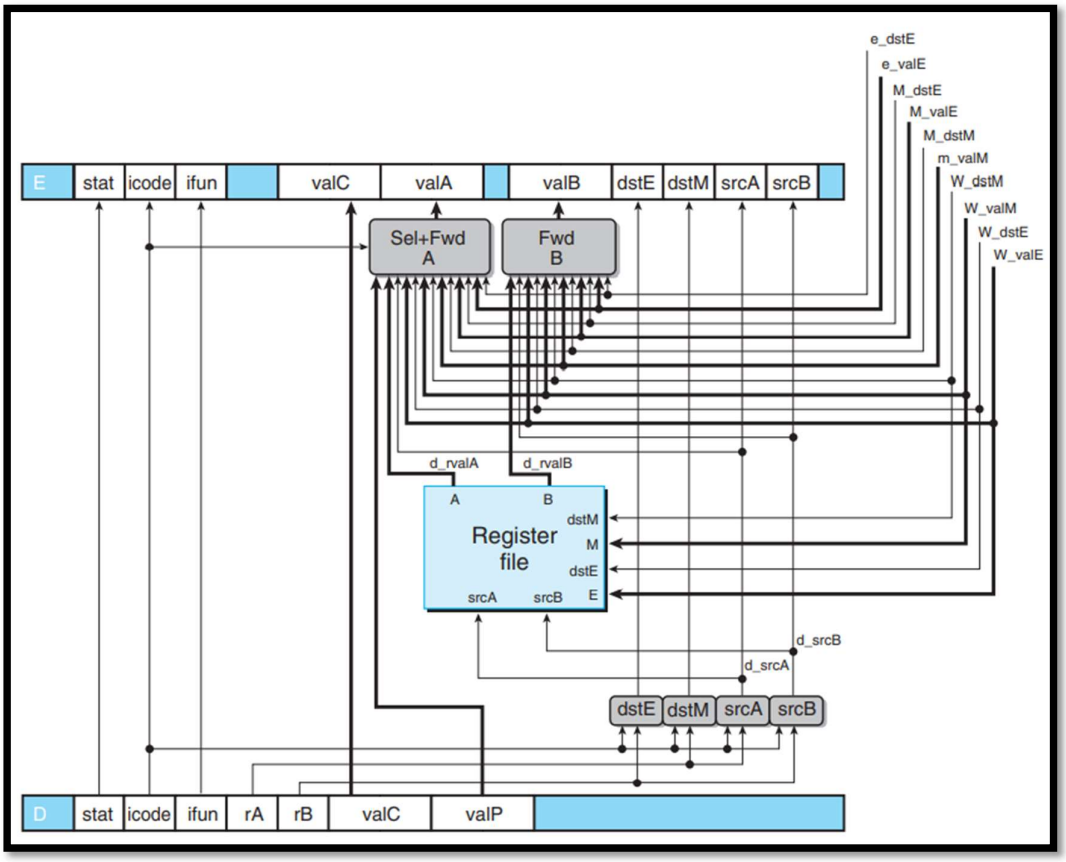

```
x,icode=xxxx,ifun=xxxx,ra=xxxx,rb=xxxx,valc=        x,valp=        x,status=xxx,nextpredpc=        x
 x,icode=0011,ifun=0000,ra=1111,rb=0110,valc=       8,valp=       20,status=000,nextpredpc=       20
 x,icode=0011,ifun=0000,ra=1111,rb=0001,valc=      15,valp=       30,status=000,nextpredpc=       30
 x,icode=0110,ifun=0001,ra=0110,rb=0001,valc=      15,valp=       32,status=000,nextpredpc=       32
 x,icode=0010,ifun=0000,ra=0110,rb=0001,valc=      15,valp=       34,status=000,nextpredpc=       34
 x,icode=1000,ifun=0000,ra=0110,rb=0001,valc=      64,valp=       34,status=000,nextpredpc=       64
 x,icode=0110,ifun=0000,ra=0110,rb=0001,valc=      64,valp=       66,status=000,nextpredpc=       66
 x,icode=0100,ifun=0000,ra=0110,rb=0001,valc=      31,valp=       76,status=000,nextpredpc=       76
 x,icode=0010,ifun=0000,ra=0110,rb=1110,valc=      31,valp=       78,status=000,nextpredpc=       78
 x,icode=0100,ifun=0000,ra=0110,rb=0001,valc=      15,valp=       88,status=000,nextpredpc=       88
```

b. Decode and Write-back

In decode stage we take inputs from decode pipeline register and the bypass baths (consisting of e_dstE, e_valE, M_dstE, M_valE, M_dstM, m_valM, W_dstM, W_valM, W_dstE, W_valE) and essentially give out d_valA, d_valB, d_dstE, d_dstM, d_srcA, d_srcB values which will be stored in E pipeline register. The hardware is same as that in sequential except for data forwarding logic which detects data dependency based on D_icode and destination register IDs received using bypass. In case of any data dependency the forwarding logic satisfies the requirement.

We create write-back in the same stage in order to access register file. Other method is to instantiate register file in both the stages separately but that

would lead to two copies of register file decreasing the hardware efficiency. Write-back stage is same as its sequential counterpart.



Code:

```
module decode_wb (
D_stat, D_icode, D_ifun, D_rA, D_rB, D_valC, D_valP,
e_dstE, e_valE,
M_dstE, M_valE, M_dstM, m_valM,
W_dstM, W_valM, W_dstE, W_valE, W_stat, W_icode,
d_stat,d_icode, d_ifun, d_valC, d_valA, d_valB, d_dstE, d_dstM, d_srcA,
d_srcB,
rax,rcx,rdx,rbx,rsp,rbp,rsi,rdi,r8,r9,r10,r11,r12,r13,r14,r15
);

//decode input
//D reg
input [2:0] D_stat;
input [3:0] D_icode, D_ifun;
input [3:0] D_rA, D_rB;
input [63:0] D_valC, D_valP;
```

```
//bypass path
input [3:0] e_dstE, M_dstE, M_dstM;
input [3:0] W_dstM, W_dstE; //w_reg
input [63:0] e_valE, M_valE, m_valM;
input [63:0] W_valM, W_valE; //w_reg
//decode wires
reg [63:0] d_rvalA, d_rvalB;
//decode output
//E reg
output reg [2:0] d_stat;
output reg [3:0] d_icode, d_ifun;
output reg [63:0] d_valC, d_valA, d_valB;
output reg [3:0] d_dstE, d_dstM, d_srcA, d_srcB;
//writeback
//input
input [2:0] W_stat;
input [3:0] W_icode;
//register file
output reg [63:0] rax;
output reg [63:0] rcx;
output reg [63:0] rdx;
output reg [63:0] rbx;
output reg [63:0] rsp;
output reg [63:0] rbp;
output reg [63:0] rsi;
output reg [63:0] rdi;
output reg [63:0] r8 ;
output reg [63:0] r9 ;
output reg [63:0] r10;
output reg [63:0] r11;
output reg [63:0] r12;
output reg [63:0] r13;
output reg [63:0] r14;
output reg [63:0] r15;


    //decode
    always @(*) begin
    //d_srcA
    if (( (D_icode==4'b0010) || (D_icode==4'b0100) ) || ( (D_icode==4'b0110)
|| (D_icode==4'b1010) ))
        begin
            d_srcA<=D_rA;
        end
    else if( (D_icode==4'b1001) || (D_icode==4'b1011))
        begin
            d_srcA<=4'h4;
        end
    else
```

```
        begin
            d_srcA<=4'hF;
        end

    //d_srcB

    if ( (D_icode==4'b0110)||((D_icode==4'b0100)||(D_icode==4'b0101)))
        begin
            d_srcB<=D_rB;
        end
    else if(( (D_icode==4'b1010)||(D_icode==4'b1011)
)||((D_icode==4'b1000)||(D_icode==4'b1001)))
        begin
            d_srcB<=4'h4;
        end
    else
        d_srcB<=4'hF;
//////////////////////////
    //d_dstE
    if ( (D_icode==4'b0011) ||(D_icode==4'b0110) || (D_icode==4'h2))
        d_dstE<=D_rB;
    else
if(((D_icode==4'b1010)||(D_icode==4'b1011))||((D_icode==4'b1000)||(D_icode==4'
b1001)))
        d_dstE<=4'h4;
    else
        d_dstE=4'hF;

    //d_dstM
    if ((D_icode==4'b0101)||(D_icode==4'b1011))
        d_dstM<=D_rA;
    else
        d_dstM<=4'hF;
//////////////////////////
    //reg file
        case (d_srcA)
        4'h0:d_rvalA <= rax;
        4'h1:d_rvalA <= rcx;
        4'h2:d_rvalA <= rdx;
        4'h3:d_rvalA <= rbx;
        4'h4:d_rvalA <= rsp;
        4'h5:d_rvalA <= rbp;
        4'h6:d_rvalA <= rsi;
        4'h7:d_rvalA <= rdi;
        4'h8:d_rvalA <= r8 ;
        4'h9:d_rvalA <= r9 ;
        4'hA:d_rvalA <= r10;
        4'hB:d_rvalA <= r11;
```

```
        4'hC:d_rvalA <= r12;
        4'hD:d_rvalA <= r13;
        4'hE:d_rvalA <= r14;
        default: r15 <= 64'hF;
    endcase
    case (d_srcB)
        4'h0:d_rvalB <= rax;
        4'h1:d_rvalB <= rcx;
        4'h2:d_rvalB <= rdx;
        4'h3:d_rvalB <= rbx;
        4'h4:d_rvalB <= rsp;
        4'h5:d_rvalB <= rbp;
        4'h6:d_rvalB <= rsi;
        4'h7:d_rvalB <= rdi;
        4'h8:d_rvalB <= r8 ;
        4'h9:d_rvalB <= r9 ;
        4'hA:d_rvalB <= r10;
        4'hB:d_rvalB <= r11;
        4'hC:d_rvalB <= r12;
        4'hD:d_rvalB <= r13;
        4'hE:d_rvalB <= r14;
        default: r15 <= 64'hF;
    endcase

    //sel+fwd A

    if ((D_icode==4'b1000)||(D_icode==4'b0111)) begin
        d_valA = D_valP;
    end
    else begin
        case (d_srcA)
            e_dstE: d_valA = e_valE;
            M_dstM: d_valA = m_valM;
            M_dstE: d_valA = M_valE;
            W_dstM: d_valA = W_valM;
            W_dstE: d_valA = W_valE;
            default:d_valA = d_rvalA;
        endcase
    end

    //Fwd B

    begin
        case (d_srcB)
            e_dstE: d_valB = e_valE;
            M_dstM: d_valB = m_valM;
            M_dstE: d_valB = M_valE;
            W_dstM: d_valB = W_valM;
```

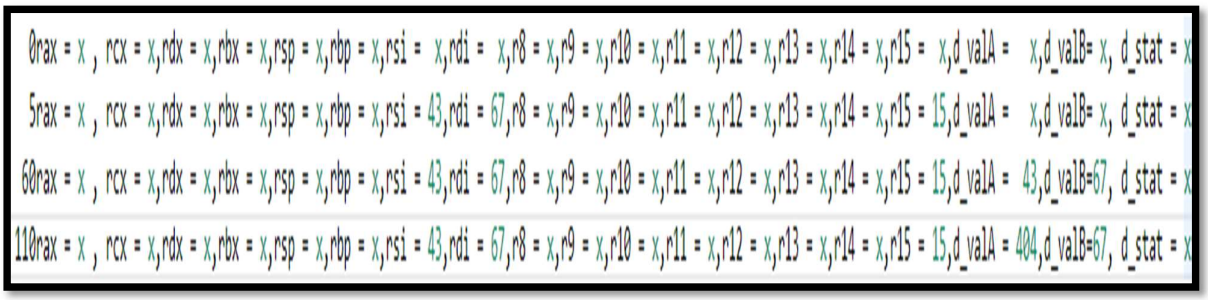
```
            W_dstE: d_valB = W_valE;
            default:d_valB = d_rvalB;
        endcase
    end
    //Reg assign
    d_stat = D_stat;
    d_icode = D_icode;
    d_ifun = D_ifun;
    d_valC = D_valC;
    end

    //writeback
    always @(*) begin

            //reg file
            case (W_dstM)
                4'h0:rax <= W_valM;
                4'h1:rcx <= W_valM;
                4'h2:rdx <= W_valM;
                4'h3:rbx <= W_valM;
                4'h4:rsp <= W_valM;
                4'h5:rbp <= W_valM;
                4'h6:rsi <= W_valM;
                4'h7:rdi <= W_valM;
                4'h8:r8  <= W_valM;
                4'h9:r9  <= W_valM;
                4'hA:r10 <= W_valM;
                4'hB:r11 <= W_valM;
                4'hC:r12 <= W_valM;
                4'hD:r13 <= W_valM;
                4'hE:r14 <= W_valM;
                default: r15 <= 64'hF;
            endcase
            case (W_dstE)
                4'h0:rax <= W_valE;
                4'h1:rcx <= W_valE;
                4'h2:rdx <= W_valE;
                4'h3:rbx <= W_valE;
                4'h4:rsp <= W_valE;
                4'h5:rbp <= W_valE;
                4'h6:rsi <= W_valE;
                4'h7:rdi <= W_valE;
                4'h8:r8  <= W_valE;
                4'h9:r9  <= W_valE;
                4'hA:r10 <= W_valE;
                4'hB:r11 <= W_valE;
                4'hC:r12 <= W_valE;
                4'hD:r13 <= W_valE;
```

```
                4'hE:r14 <= W_valE;
                default: r15 <= 64'hF;
            endcase
    end
endmodule
```

Output:

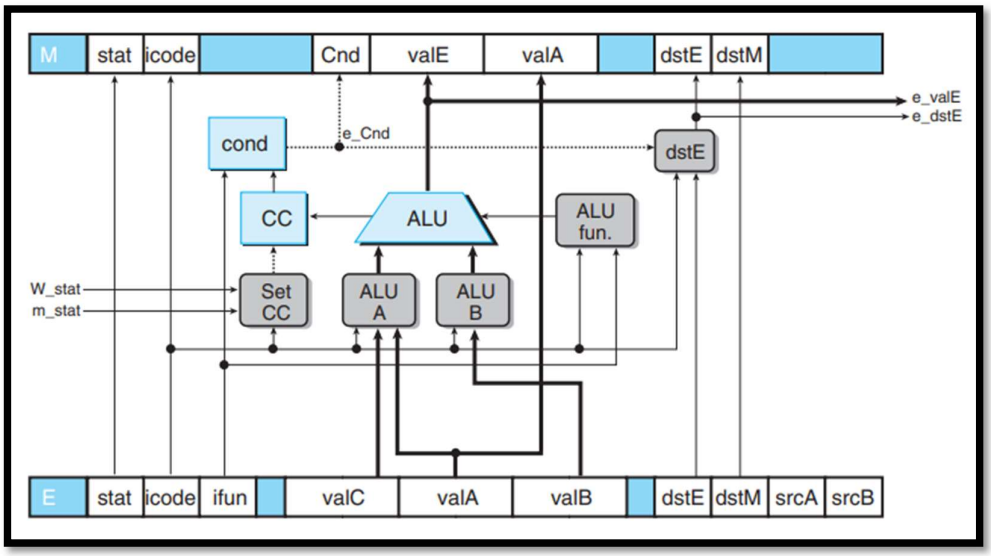
In the output, we are able to see three processes:

1. Reading of the register content when required. (decode)
2. Providing the values to decode stage. (bypass paths)
3. Writing to the registers in register file. (write-back)



c. Execute

This stage is same as that in sequential but the set_cc value is determined by combinational block in the pipeline control logic. This tells us whether instructions in later pipeline stages are causing any exceptions. The outputs of this stage are stored in M pipeline register.



Code

```
`include "ALU_64.v"


module
execute(E_stat,E_icode,E_ifun,E_valC,E_valA,E_valB,E_dstE,E_dstM,E_srcA,E_srcB
,e_valE,e_cnd,EE_valA,e_dstE,EE_stat,EE_icode,EE_dstM,zf,sf,of);
input [2:0] E_stat;
input [3:0] E_icode,E_ifun;
input signed [63:0] E_valA,E_valB,E_valC;
input [3:0] E_dstE,E_dstM,E_srcA,E_srcB;

output signed [63:0] e_valE;
output reg e_cnd;
output reg [3:0] EE_icode,EE_dstM;
output reg [2:0] EE_stat;
output reg signed [63:0] EE_valA;
output reg [3:0] e_dstE;
output reg  zf,sf,of;
reg signed [63:0] alu_A,alu_B;
reg [1:0] alu_fun;
reg set_cc;
wire overflow;

ALU_64 instant(alu_fun[1:0],alu_A,alu_B,e_valE,overflow);


always @ (*)
begin
if (set_cc==1'b1)
    of = overflow;
end
```

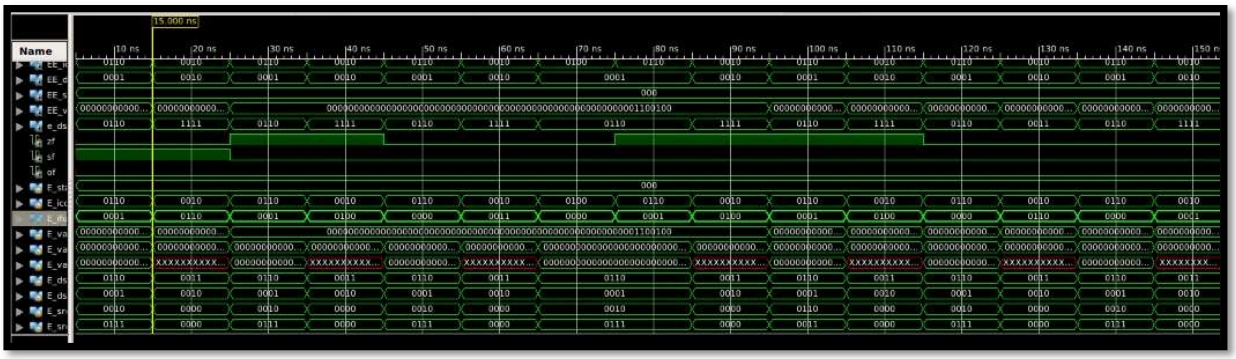
```
always @(e_valE)
 begin
       if(set_cc==1'b1)
        begin
        if(e_valE==64'd0)
            zf=1'b1;
        else
            zf=1'b0;

        if(e_valE[63]==1'b1)
            sf=1'b1;
        else
            sf=1'b0;

        end



 end


always @(E_icode,E_valC,E_valA,E_valB,E_ifun)
begin
if((E_icode==4'b0010)||(E_icode==4'b0110))          //aluA
      alu_A = E_valA;

   else if((E_icode==4'b0011)||(E_icode==4'b0100)||(E_icode==4'b0101))
      alu_A = E_valC;

   else if((E_icode==4'b1000)||(E_icode==4'b1010))
      alu_A = -64'd8;

   else if((E_icode==4'b1001)||(E_icode==4'b1011))
      alu_A = 64'd8;


if((E_icode==4'b0010)||(E_icode==4'b0011))                  //aluB
      alu_B = 64'b0;

   else
if((E_icode==4'b0110)||(E_icode==4'b0100)||(E_icode==4'b0101)||(E_icode==4'b10
11)||(E_icode==4'b1001)||(E_icode==4'b1010)||(E_icode==4'b1000))
      alu_B = E_valB;
```

```
if (E_icode==4'b0110)                          //alu_fun   and set_cc

      begin
      set_cc=1'b1;
      alu_fun=E_ifun;
      end
   else
      begin
      alu_fun=4'b0000;
      set_cc=1'b0;
      end

end
    always@(*)
    begin
          if((E_icode==4'b0010)||(E_icode==4'b0111))
          begin
              case(E_ifun)
              4'b0000:e_cnd<=1'b1;
              4'b0001:e_cnd<=(sf^of)|zf;
              4'b0010:e_cnd<=sf^of;
              4'b0011:e_cnd<=zf;
              4'b0100:e_cnd<=~zf;
              4'b0101:e_cnd<=~(sf^of);
              4'b0110:e_cnd<=~(sf^of)&~zf;
              endcase

          end

          else
          begin
          e_cnd=1'bx;
          e_dstE=E_dstE;
          end

    end

    always@(*)
    begin
             if (e_cnd==1'b0)
                 e_dstE=4'b1111;
              else if(e_cnd==1'b1)
                 e_dstE=E_dstE;
     end



    always@(E_stat,E_icode,E_dstM)
```

```
    begin
    EE_stat=E_stat;
    EE_icode=E_icode;
    EE_dstM=E_dstM;
    EE_valA=E_valA;
    end
endmodule
```

Output:

The output is checked for subq, ccmovne, subq, ccmovne, addq, ccmove, rmmovq, subq, ccmovne, xorq, ccmovne, addq, ccmovg, addq and ccmovle as inputs.

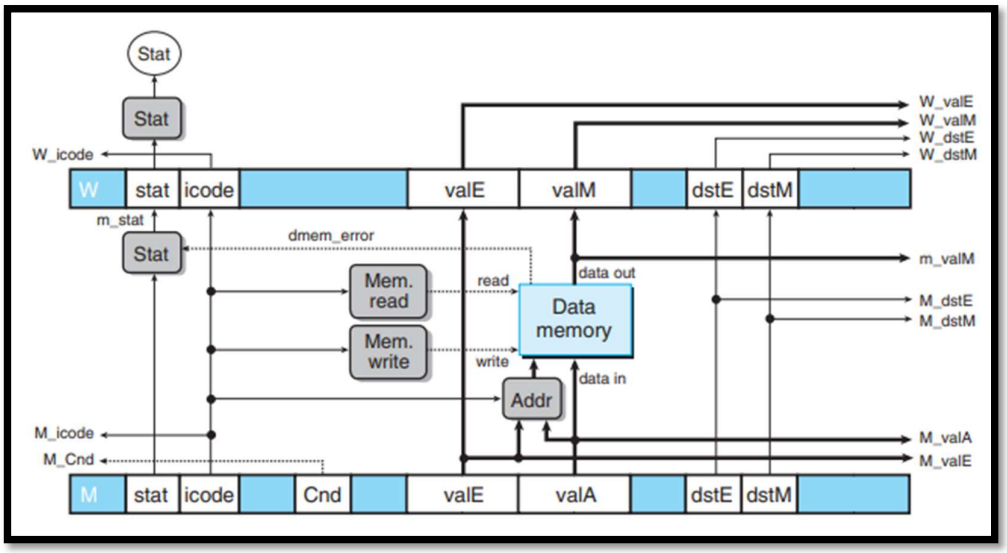


```
E_icode=0110,E_ifun=0001,E_valC=256,E_valA=100,E_valB=200,E_dstE=0110,E_dstM=0001,E_srcA=0010,E_srcB=0111,e_valE=-100,e_cnd=x,EE_valA=100,e_dstE=0110,EE_stat=000,EE_icode=0110,EE_dstM=0001,zf=0,sf=1,of=0
E_icode=0010,E_ifun=0110,E_valC=  x,E_valA= 25,E_valB= 36,E_dstE=0011,E_dstM=0010,E_srcA=0000,E_srcB=0000,e_valE=  25,e_cnd=0,EE_valA= 25,e_dstE=1111,EE_stat=000,EE_icode=0010,EE_dstM=0010,zf=0,sf=1,of=0
E_icode=0110,E_ifun=0001,E_valC=256,E_valA=100,E_valB=100,E_dstE=0110,E_dstM=0001,E_srcA=0010,E_srcB=0111,e_valE=   0,e_cnd=x,EE_valA=100,e_dstE=0110,EE_stat=000,EE_icode=0110,EE_dstM=0001,zf=1,sf=0,of=0
E_icode=0010,E_ifun=0100,E_valC=  x,E_valA=100,E_valB=200,E_dstE=0011,E_dstM=0010,E_srcA=0000,E_srcB=0000,e_valE= 100,e_cnd=0,EE_valA=100,e_dstE=1111,EE_stat=000,EE_icode=0010,EE_dstM=0010,zf=1,sf=0,of=0
E_icode=0110,E_ifun=0000,E_valC=256,E_valA=100,E_valB=100,E_dstE=0110,E_dstM=0001,E_srcA=0010,E_srcB=0111,e_valE= 200,e_cnd=x,EE_valA=100,e_dstE=0110,EE_stat=000,EE_icode=0110,EE_dstM=0001,zf=0,sf=0,of=0
E_icode=0010,E_ifun=0011,E_valC=  x,E_valA=100,E_valB=200,E_dstE=0011,E_dstM=0010,E_srcA=0000,E_srcB=0000,e_valE= 100,e_cnd=0,EE_valA=100,e_dstE=1111,EE_stat=000,EE_icode=0010,EE_dstM=0010,zf=0,sf=0,of=0
E_icode=0100,E_ifun=0000,E_valC=256,E_valA=100,E_valB=100,E_dstE=0110,E_dstM=0001,E_srcA=0010,E_srcB=0111,e_valE= 356,e_cnd=x,EE_valA=100,e_dstE=0110,EE_stat=000,EE_icode=0100,EE_dstM=0001,zf=0,sf=0,of=0
E_icode=0110,E_ifun=0001,E_valC=256,E_valA=100,E_valB=100,E_dstE=0110,E_dstM=0001,E_srcA=0010,E_srcB=0111,e_valE=   0,e_cnd=x,EE_valA=100,e_dstE=0110,EE_stat=000,EE_icode=0110,EE_dstM=0001,zf=1,sf=0,of=0
E_icode=0010,E_ifun=0100,E_valC=  x,E_valA=100,E_valB=200,E_dstE=0011,E_dstM=0010,E_srcA=0000,E_srcB=0000,e_valE= 100,e_cnd=0,EE_valA=100,e_dstE=1111,EE_stat=000,EE_icode=0010,EE_dstM=0010,zf=1,sf=0,of=0
E_icode=0110,E_ifun=0001,E_valC=256,E_valA=250,E_valB=250,E_dstE=0110,E_dstM=0001,E_srcA=0110,E_srcB=0011,e_valE=   0,e_cnd=x,EE_valA=250,e_dstE=0110,EE_stat=000,EE_icode=0110,EE_dstM=0001,zf=1,sf=0,of=0
E_icode=0010,E_ifun=0100,E_valC=  x,E_valA=100,E_valB=200,E_dstE=0011,E_dstM=0010,E_srcA=0000,E_srcB=0000,e_valE= 100,e_cnd=0,EE_valA=100,e_dstE=1111,EE_stat=000,EE_icode=0010,EE_dstM=0010,zf=1,sf=0,of=0
E_icode=0110,E_ifun=0000,E_valC=256,E_valA=200,E_valB=300,E_dstE=0110,E_dstM=0001,E_srcA=0010,E_srcB=0111,e_valE= 500,e_cnd=x,EE_valA=200,e_dstE=0110,EE_stat=000,EE_icode=0110,EE_dstM=0001,zf=0,sf=0,of=0
E_icode=0010,E_ifun=0110,E_valC=  x,E_valA= 25,E_valB= 36,E_dstE=0011,E_dstM=0010,E_srcA=0000,E_srcB=0000,e_valE=  25,e_cnd=1,EE_valA= 25,e_dstE=0011,EE_stat=000,EE_icode=0010,EE_dstM=0010,zf=0,sf=0,of=0
E_icode=0110,E_ifun=0000,E_valC=256,E_valA=500,E_valB=100,E_dstE=0110,E_dstM=0001,E_srcA=0010,E_srcB=0111,e_valE= 600,e_cnd=x,EE_valA=500,e_dstE=0110,EE_stat=000,EE_icode=0110,EE_dstM=0001,zf=0,sf=0,of=0
E_icode=0010,E_ifun=0001,E_valC=  x,E_valA=100,E_valB=200,E_dstE=0011,E_dstM=0010,E_srcA=0000,E_srcB=0000,e_valE= 100,e_cnd=0,EE_valA=100,e_dstE=1111,EE_stat=000,EE_icode=0010,EE_dstM=0010,zf=0,sf=0,of=0
```

d. Memory

In this stage we either read or write from or to the memory respectively. It is same as its sequential counterpart, only difference is that the ‘Mem data’

block is removed as its function is performed by forwarding block in decode stage.



Code:

```
module memory
(M_stat,M_icode,M_cnd,M_valE,M_valA,M_dstE,M_dstM,m_valM,m_stat,MM_icode,MM_va
lE,MM_dstE,MM_dstM);
input [2:0] M_stat;
input [3:0] M_icode,M_dstE,M_dstM;
input M_cnd;
input [63:0] M_valE,M_valA;

output reg [63:0] m_valM;
output reg [2:0] m_stat;
output reg [3:0] MM_icode;
output reg [63:0] MM_valE;
output reg [3:0] MM_dstE,MM_dstM;


reg [63:0] mem_ad;
reg [63:0] mem_dt;
reg m_read;
reg m_write;
reg [7:0] mem[65535:0];
reg dmem_error;




always @(M_icode,M_valE,M_valA)
    begin
```

```
        mem_dt=M_valA;
        MM_icode=M_icode;
        MM_valE=M_valE;
        MM_dstE=M_dstE;
        MM_dstM=M_dstM;

if((M_icode==4'b0100)||(M_icode==4'b1010)||(M_icode==4'b1000)||(M_icode==4'b01
01))
            mem_ad=M_valE;
        else if((M_icode==4'b1011)||(M_icode==4'b1001))
            mem_ad=M_valA;
    end

always @(M_icode,M_valE,M_valA)
    begin

        if((M_icode==4'b0101)||(M_icode==4'b1011)||(M_icode==4'b1001))
            m_read=1;
        else
            m_read=0;

        if((M_icode==4'b0100)||(M_icode==4'b1010)||(M_icode==4'b1000))
            m_write=1;
        else
            m_write=0;

        if ((mem_ad+7>65536)||(m_read & m_write))
            dmem_error=1'b1;
        else
            dmem_error=1'b0;


    end

always @(dmem_error)
    begin
        if(dmem_error==1'b1)
            m_stat=2'b10;              //ADR  2'b10
        else
            m_stat=M_stat;
    end


always @(m_read,m_write,mem_ad,mem_dt)
    begin
       if(m_write&(~(dmem_error)))
        begin
          mem[mem_ad]   <= mem_dt[7:0];
```

```
          mem[mem_ad+1] <= mem_dt[15:8];
          mem[mem_ad+2] <= mem_dt[23:16];
          mem[mem_ad+3] <= mem_dt[31:24];
          mem[mem_ad+4] <= mem_dt[39:32];
          mem[mem_ad+5] <= mem_dt[47:40];
          mem[mem_ad+6] <= mem_dt[55:48];
          mem[mem_ad+7] <= mem_dt[63:56];
        end
        if(m_read&(~(dmem_error)))
        begin
          m_valM[7:0]   <=mem[mem_ad];
          m_valM[15:8]  <=mem[mem_ad+1];
          m_valM[23:16] <=mem[mem_ad+2];
          m_valM[31:24] <=mem[mem_ad+3];
          m_valM[39:32] <=mem[mem_ad+4];
          m_valM[47:40] <=mem[mem_ad+5];
          m_valM[55:48] <=mem[mem_ad+6];
          m_valM[63:56] <=mem[mem_ad+7];
        end
      if (m_read&dmem_error)
        begin
          m_valM[7:0]   <=8'h0;
          m_valM[15:8]  <=8'h0;
          m_valM[23:16] <=8'h0;
          m_valM[31:24] <=8'h0;
          m_valM[39:32] <=8'h0;
          m_valM[47:40] <=8'h0;
          m_valM[55:48] <=8'h0;
          m_valM[63:56] <=8'h0;
        end
    end
endmodule
```

Output:

The memory module is checked for following commands:

rmmovq

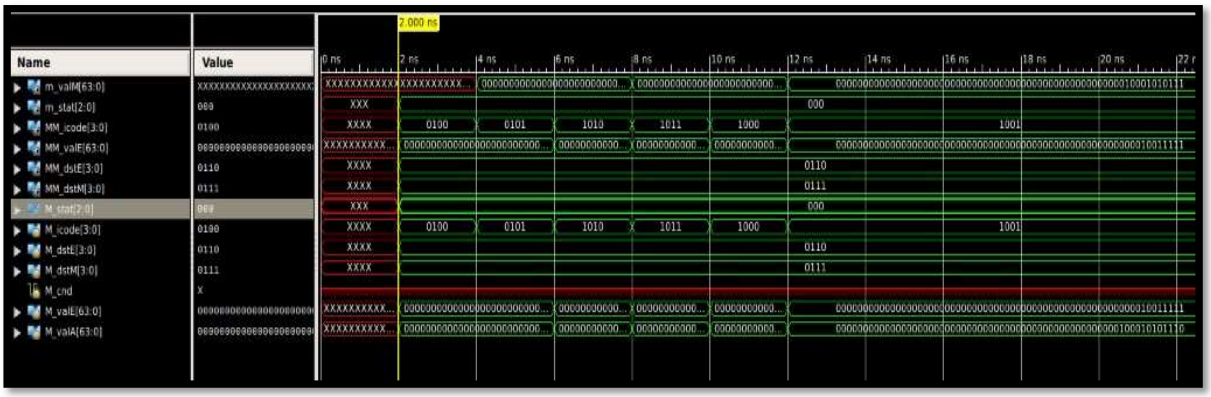
mrmovq

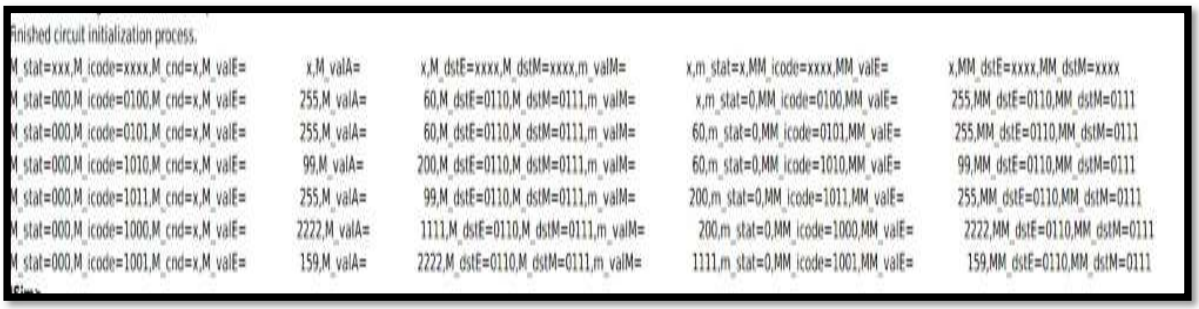
pushq

popq

call

ret





Pipelined Registers.

F – It holds the predicted value of program counter, one in each clock cycle.

Code:

```
module fetch_reg (clk,F_stall,ipredpc,opredpc);
input clk;
input [63:0] ipredpc;
input  F_stall;
output reg [63:0] opredpc;



always@(posedge clk)
   begin
    if(F_stall==1'b0)
       opredpc=ipredpc;
    end
endmodule
```

D – It holds the information about most recently fetched instruction required by decode stage for processing.

Code:

```
module decode_reg (
    clk,
    D_stall, D_bubble,
    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 D_stall, D_bubble;

    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
        if ((D_stall==1'b0) && (D_bubble==1'b0))
            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
        else if (D_bubble==1'b1)
            begin
                D_stat  = 3'bx;
                D_icode = 4'bx;
                D_ifun  = 4'bx;
                D_rA    = 4'hF;
                D_rB    = 4'hF;
                D_valC  = 64'bx;
                D_valP  = 64'bx;
            end
```

```
    end
endmodule
```

E – It holds the output of decode stage along with some values from decode register.

Code:

```
module execute_reg (
    clk,
    E_bubble,
    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 E_bubble;
input [2:0] d_stat;
input [3:0] d_icode, d_ifun;
input [63:0] d_valC, d_valA, d_valB;
input [3:0] d_dstE, d_dstM, d_srcA, d_srcB;

output reg [2:0] E_stat;
output reg [3:0] E_icode, E_ifun;
output reg [63:0] E_valC, E_valA, E_valB;
output reg [3:0] E_dstE, E_dstM, E_srcA, E_srcB;



    always @(posedge clk ) begin
    if(E_bubble==1'b0)
    begin
        E_stat = d_stat;
        E_icode = d_icode;
        E_ifun = d_ifun;
        E_valC = d_valC;
        E_valA = d_valA;
        E_valB = d_valB;
        E_dstE = d_dstE;
        E_dstM = d_dstM;
        E_srcA = d_srcA;
        E_srcB = d_srcB;
    end
    else if(E_bubble==1'b1)
    begin
        E_stat = 3'bx;
```

```
        E_icode= 4'bx;
        E_ifun = 4'bx;
        E_valC = 64'bx;
        E_valA = 64'bx;
        E_valB = 64'bx;
        E_dstE = 4'hF;
        E_dstM = 4'hF;
        E_srcA = 4'hF;
        E_srcB = 4'hF;
    end

    end
endmodule
```

M – It holds output of the execute stage.

Code:

```
module memory_reg (
    clk,
    M_bubble,
    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 M_bubble;
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
    if (M_bubble==1'b0)
    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
    else if (M_bubble==1'b1)
    begin
        M_stat  = 3'bx;
        M_icode = 4'bx;
        M_CND   = 1'bx;
        M_valE  = 64'bx;
        M_valA  = 64'bx;
        M_dstE  = 4'hF;
        M_dstM  = 4'hF;
    end
end

endmodule
```

W – It holds output of memory stage.

Code:

```
module write_reg (
    clk,
    W_stall,
    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 W_stall;
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
    if (W_stall==1'b0)
        begin
```

```
            W_stat  = m_stat;
            W_icode = m_icode;
            W_valE  = m_valE;
            W_valM  = m_valM;
            W_dstE  = m_dstE;
            W_dstM  = m_dstM;
        end
end

endmodule
```

4. Integrating Pipelined Stages

Pipelined Processor:

ASM code:

Irmovq $100, %rsp

Irmovq $6, %rax

Irmovq $1, %rcx

Irmovq $5, %rbx

Rmmovq %rbx, 0(%rcx)

Addq %rcx, %rbx

Ccmove %rax, %rcx

Pushq %rax

Subq %rcx, %rbx

Code:

```
`include "fetch_regpl.v"
`include "fetch_pipepl.v"
`include "decode_regpl.v"
`include "decode_writeb.v"
`include "execute_regpl.v"
`include "execute_pipepl.v"
`include "memory_regpl.v"
`include "memory_pipe.v"
`include "writeback_regpl.v"
`include "pipe_control.v"

module Pipeline;

reg clk;
```

```
reg [63:0] ipredpc;

wire [63:0] opredpc,f_valC,f_valP,f_nextpredpc;
wire [3:0] f_icode,f_ifun,f_ra,f_rb;
wire [2:0] f_stat;

wire [2:0] D_stat,d_stat;
wire [3:0]
D_icode,D_ifun,D_ra,D_rb,d_icode,d_ifun,d_dstE,d_dstM,d_srcA,d_srcB;
wire [63:0] rax,rcx,rdx,rbx,rsp,rbp,rsi,rdi,r8,r9,r10,r11,r12,r13,r14,r15;
wire signed [63:0] D_valC,D_valP,d_valC,d_valA,d_valB;

wire [2:0] E_stat,e_stat;
wire [3:0] E_icode,E_ifun,E_dstE,E_dstM,E_srcA,E_srcB,e_icode,e_dstE,e_dstM;
wire signed [63:0] E_valC,E_valA,E_valB,e_valA,e_valE;
wire e_cnd;


wire [2:0] M_stat,m_stat;
wire [3:0] M_icode,M_dstE,M_dstM,m_icode,m_dstE,m_dstM;
wire M_cnd;
wire signed [63:0] M_valE,M_valA,m_valM,m_valE;


wire [2:0] W_stat;
wire [3:0] W_icode,W_dstE,W_dstM;
wire signed [63:0] W_valE,W_valM;




wire  F_stall, D_stall, D_bubble, E_bubble, M_bubble, W_stall;


pipe_control uut(D_icode, d_srcA, d_srcB, E_icode, E_dstM, e_cnd, M_icode,
m_stat, W_stat,
                F_stall, D_stall, D_bubble, E_bubble, M_bubble, W_stall);


fetch_reg regf(clk,F_stall,ipredpc,opredpc);

fetch fet(opredpc,W_icode,M_icode,M_Cnd,W_valM,M_valA,e_valA,e_cnd,E_icode,
             f_icode,
             f_ifun,
             f_ra,
             f_rb,
             f_valC,
             f_valP,
             f_stat,
```

```
             f_nextpredpc);        //////////////

decode_reg regd(clk,D_stall, D_bubble,
                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);

decode_wb d_and_wb(D_stat, D_icode, D_ifun, D_ra, D_rb, D_valC, D_valP,
                   e_dstE, e_valE,
                   M_dstE, M_valE, M_dstM, m_valM,
                   W_dstM, W_valM, W_dstE, W_valE, W_stat, W_icode,
                   d_stat,d_icode, d_ifun, d_valC, d_valA, d_valB, d_dstE,
d_dstM, d_srcA, d_srcB,

 rax,rcx,rdx,rbx,rsp,rbp,rsi,rdi,r8,r9,r10,r11,r12,r13,r14,r15);

execute_reg rege(clk,E_bubble,

 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);


execute
exe(E_stat,E_icode,E_ifun,E_valC,E_valA,E_valB,E_dstE,E_dstM,E_srcA,E_srcB,e_v
alE,e_cnd,e_valA,e_dstE,e_stat,e_icode,e_dstM);

memory_reg regm(clk,M_bubble,
                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);

memory
mem(M_stat,M_icode,M_cnd,M_valE,M_valA,M_dstE,M_dstM,m_valM,m_stat,m_icode,m_v
alE,m_dstE,m_dstM);

write_reg regw(clk,W_stall,
                  m_stat, m_icode, m_valE, m_valM, m_dstE, m_dstM,
                  W_stat, W_icode, W_valE, W_valM, W_dstE, W_dstM);
initial
/////ipredpc=64'd10;  // TO TEST LOAD/USE HAZARD
ipredpc=64'd10;
initial
clk=1'b0;
always
#5 clk=~clk;
always@(*)
ipredpc=f_nextpredpc;
initial
begin
```

```
$monitor("opredpc=%d,f_icode=%b,f_ifun=%b,f_ra=%b,f_rb=%b,f_valC=%d,f_valP=%d,
f_stat=%b,f_nextpredpc=%d,d_stat=%b,d_icode=%b, d_ifun=%b, d_valC=%d,
d_valA=%d, d_valB=%d, d_dstE=%b, d_dstM=%b, d_srcA=%b,
d_srcB=%b,e_valE=%d,e_cnd=%d,e_valA=%d,e_dstE=%b,e_stat=%b,e_icode=%b,e_dstM=%
b,m_valM=%d,m_stat=%b,m_icode=%b,m_valE=%d,m_dstE=%b,m_dstM=%b,W_dstM=%b,
W_valM=%d, W_dstE=%b, W_valE=%d, W_stat=%b,
W_icode=%b,rax=%d,rcx=%d,rdx=%d,rbx=%d,rsp=%d,rbp=%d,rsi=%d,rdi=%d,r8=%d,r9=%d
,r10=%d,r11=%d,r12=%d,r13=%d,r14=%d,r15=%d,F_stall=%b,D_stall=%b,D_bubble=%b,
E_bubble=%b, M_bubble=%b,
W_stall=%b",opredpc,f_icode,f_ifun,f_ra,f_rb,f_valC,f_valP,f_stat,f_nextpredpc
,d_stat,d_icode, d_ifun, d_valC, d_valA, d_valB, d_dstE, d_dstM, d_srcA,
d_srcB,e_valE,e_cnd,e_valA,e_dstE,e_stat,e_icode,e_dstM,m_valM,m_stat,m_icode,
m_valE,m_dstE,m_dstM,W_dstM, W_valM, W_dstE, W_valE, W_stat,
W_icode,rax,rcx,rdx,rbx,rsp,rbp,rsi,rdi,r8,r9,r10,r11,r12,r13,r14,r15,F_stall,
D_stall,D_bubble, E_bubble, M_bubble, W_stall);
#90 $finish;
end
endmodule
```

Instruction Memory:

```
//---------Data-forwarding-----------------//
     inst_mem[10]=8'b0000_0011;     //irmovq
     inst_mem[11]=8'b0100_1111;
     inst_mem[10+2+0]=8'b0110_0100;
     inst_mem[10+2+1]=8'b0000_0000;
     inst_mem[10+2+2]=8'b0000_0000;
     inst_mem[10+2+3]=8'b0000_0000;
     inst_mem[10+2+4]=8'b0000_0000;
     inst_mem[10+2+5]=8'b0000_0000;
     inst_mem[10+2+6]=8'b0000_0000;
     inst_mem[10+2+7]=8'b0000_0000;



     inst_mem[20]=8'b0000_0011;     //irmovq
     inst_mem[21]=8'b0000_1111;
     inst_mem[20+2+0]=8'b0000_0110;
     inst_mem[20+2+1]=8'b0000_0000;
     inst_mem[20+2+2]=8'b0000_0000;
     inst_mem[20+2+3]=8'b0000_0000;
     inst_mem[20+2+4]=8'b0000_0000;
     inst_mem[20+2+5]=8'b0000_0000;
     inst_mem[20+2+6]=8'b0000_0000;
     inst_mem[20+2+7]=8'b0000_0000;

     inst_mem[30]=8'b0000_0011;     //irmovq
```

```
        inst_mem[31]=8'b0001_1111;
        inst_mem[30+2+0]=8'b0000_0001;
        inst_mem[30+2+1]=8'b0000_0000;
        inst_mem[30+2+2]=8'b0000_0000;
        inst_mem[30+2+3]=8'b0000_0000;
        inst_mem[30+2+4]=8'b0000_0000;
        inst_mem[30+2+5]=8'b0000_0000;
        inst_mem[30+2+6]=8'b0000_0000;
        inst_mem[30+2+7]=8'b0000_0000;

        inst_mem[40]=8'b0000_0011;     //irmovq
        inst_mem[41]=8'b0101_1111;
        inst_mem[40+2+0]=8'b0000_0101;
        inst_mem[40+2+1]=8'b0000_0000;
        inst_mem[40+2+2]=8'b0000_0000;
        inst_mem[40+2+3]=8'b0000_0000;
        inst_mem[40+2+4]=8'b0000_0000;
        inst_mem[40+2+5]=8'b0000_0000;
        inst_mem[40+2+6]=8'b0000_0000;
        inst_mem[40+2+7]=8'b0000_0000;



        inst_mem[50]=8'b0000_0100;            //rmmovq
        inst_mem[51]=8'b0001_0011;
        inst_mem[50+2+0]=8'b0000_0000;
        inst_mem[50+2+1]=8'b0000_0000;
        inst_mem[50+2+2]=8'b0000_0000;
        inst_mem[50+2+3]=8'b0000_0000;
        inst_mem[50+2+4]=8'b0000_0000;
        inst_mem[50+2+5]=8'b0000_0000;
        inst_mem[50+2+6]=8'b0000_0000;
        inst_mem[50+2+7]=8'b0000_0000;

        inst_mem[60]=8'b0000_0110;    //addq
        inst_mem[60+1]=8'b0011_0001;

        inst_mem[62]=8'b0011_0010;    //ccmove
        inst_mem[62+1]=8'b0001_0000;

        inst_mem[64]=8'b0000_1010;    //pushq
        inst_mem[64+1]=8'b1111_0000;

        inst_mem[66]=8'b0001_0110;    //subq
        inst_mem[66+1]=8'b0011_0001;
    //---------Data-forwarding-----------------//
```

Output:

5. Instructions Supported

All the instructions in Y86 ISA are supported by the processor. The processor is also capable of handling data hazards and combinations of control hazards.

6. Pipeline hazards.

Introducing pipelining into a system with feedback can lead to problems when there are dependencies between successive instructions. These dependencies can take two forms: (1) data dependencies, where the results computed by one instruction are used as the data for a following instruction, and (2) control dependencies, where one instruction determines the location of the following instruction, such as when executing a jump, call, or return. When such dependencies have the potential to cause an erroneous computation by the pipeline, they are called hazards. Like dependencies, hazards can be classified as either data hazards or control hazards.

In order to defy the effects of dependency, we have implemented (1) Instruction Stalling and (2) Data forwarding which will be explained later.

1. **Load/Use Hazard:**

```
Irmovq $8, %rdx
Irmovq $15, %rcx
Rmmovq %rcx, 0(%rdx)
Irmovq $3, %rbx
Mrmovq 0(%rdx), %rax
Addq %rbx, %rax
Subq %rdx, %rcx
```

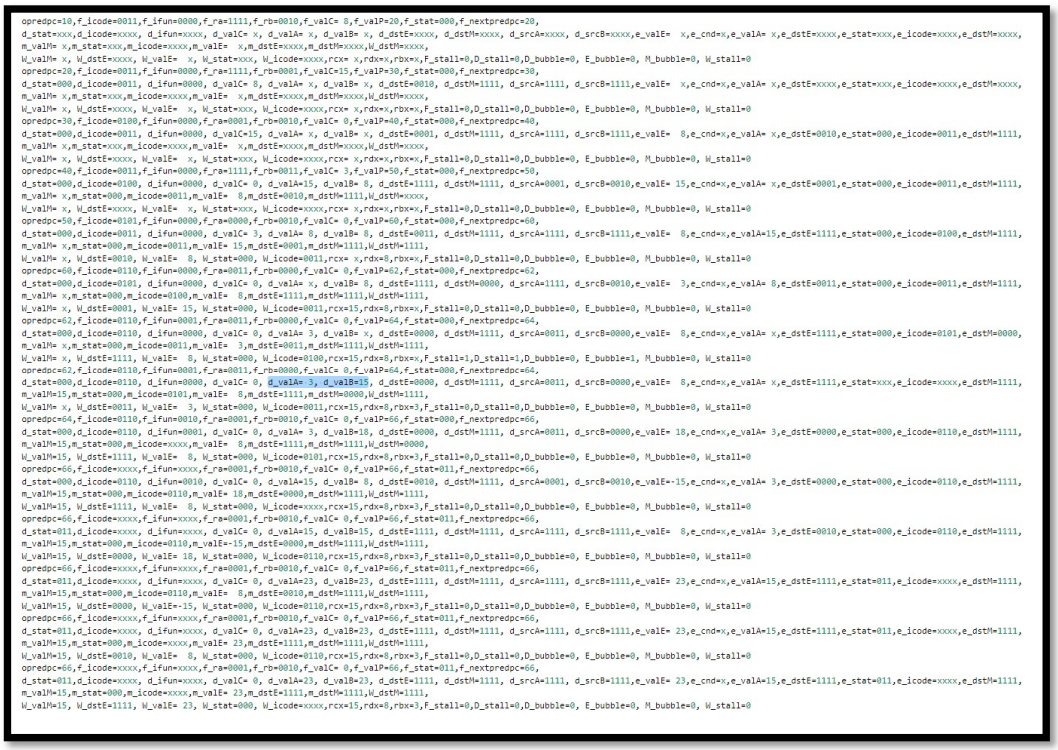
Code:

//Instruction Memory

```
///---------------Load/Use Hazard---------------------
     ///////inst_mem[10]=8'b0000_0011;            //irmovq
     ///////inst_mem[11]=8'b0010_1111;
     ///////inst_mem[10+2+0]=8'b0000_1000;
     ///////inst_mem[10+2+1]=8'b0000_0000;
     ///////inst_mem[10+2+2]=8'b0000_0000;
     ///////inst_mem[10+2+3]=8'b0000_0000;
     ///////inst_mem[10+2+4]=8'b0000_0000;
     ///////inst_mem[10+2+5]=8'b0000_0000;
     ///////inst_mem[10+2+6]=8'b0000_0000;
     ///////inst_mem[10+2+7]=8'b0000_0000;
///////
///////
 ///////
     ///////inst_mem[20]=8'b0000_0011;            //irmovq
     ///////inst_mem[21]=8'b0001_1111;
     ///////inst_mem[20+2+0]=8'b0000_1111;
     ///////inst_mem[20+2+1]=8'b0000_0000;
     ///////inst_mem[20+2+2]=8'b0000_0000;
     ///////inst_mem[20+2+3]=8'b0000_0000;
     ///////inst_mem[20+2+4]=8'b0000_0000;
     ///////inst_mem[20+2+5]=8'b0000_0000;
     ///////inst_mem[20+2+6]=8'b0000_0000;
     ///////inst_mem[20+2+7]=8'b0000_0000;
///////
     ///////inst_mem[30]=8'b0000_0100;            //rmmovq
     ///////inst_mem[31]=8'b0010_0001;
     ///////inst_mem[30+2+0]=8'b0000_0000;
     ///////inst_mem[30+2+1]=8'b0000_0000;
```

```
     ///////inst_mem[30+2+2]=8'b0000_0000;
     ///////inst_mem[30+2+3]=8'b0000_0000;
     ///////inst_mem[30+2+4]=8'b0000_0000;
     ///////inst_mem[30+2+5]=8'b0000_0000;
     ///////inst_mem[30+2+6]=8'b0000_0000;
     ///////inst_mem[30+2+7]=8'b0000_0000;
///////
     ///////inst_mem[40]=8'b0000_0011;            //irmovq
     ///////inst_mem[41]=8'b0011_1111;
     ///////inst_mem[40+2+0]=8'b0000_0011;
     ///////inst_mem[40+2+1]=8'b0000_0000;
     ///////inst_mem[40+2+2]=8'b0000_0000;
     ///////inst_mem[40+2+3]=8'b0000_0000;
     ///////inst_mem[40+2+4]=8'b0000_0000;
     ///////inst_mem[40+2+5]=8'b0000_0000;
     ///////inst_mem[40+2+6]=8'b0000_0000;
     ///////inst_mem[40+2+7]=8'b0000_0000;
///////
     ///////inst_mem[50]=8'b0000_0101;            //mrmovq
     ///////inst_mem[51]=8'b0010_0000;
     ///////inst_mem[50+2+0]=8'b0000_0000;
     ///////inst_mem[50+2+1]=8'b0000_0000;
     ///////inst_mem[50+2+2]=8'b0000_0000;
     ///////inst_mem[50+2+3]=8'b0000_0000;
     ///////inst_mem[50+2+4]=8'b0000_0000;
     ///////inst_mem[50+2+5]=8'b0000_0000;
     ///////inst_mem[50+2+6]=8'b0000_0000;
     ///////inst_mem[50+2+7]=8'b0000_0000;
///////
     ///////inst_mem[60]=8'b0000_0110;           //addq
     ///////inst_mem[61]=8'b0000_0011;
///////
///////
     ///////inst_mem[62]=8'b0001_0110;           //subq
     ///////inst_mem[63]=8'b0000_0011;
///////
     ///////inst_mem[64]=8'b0010_0110;           //andq
     ///////inst_mem[65]=8'b0010_0001;
///////
///////-----------------End of LOAD USE HAZARD---------------------
```

Output:



2. **Mis-predicted branch**

   ASM Code:

   Irmovq $1, %rax

   Xorq %rax, %rax

   Jne .p

   Irmovq $1, %rax

   .p :

   Irmovq $2, %rdx

   Irmovq $3, %rbx

   Verilog code:

   //Instruction Memory

```
//----------------Start of Branch Misprediction-------------
    /// inst_mem[10]=8'b0000_0011;     //irmovq
    /// inst_mem[11]=8'b0000_1111;
    /// inst_mem[10+2+0]=8'b0000_0001;
    /// inst_mem[10+2+1]=8'b0000_0000;
    /// inst_mem[10+2+2]=8'b0000_0000;
    /// inst_mem[10+2+3]=8'b0000_0000;
    /// inst_mem[10+2+4]=8'b0000_0000;
    /// inst_mem[10+2+5]=8'b0000_0000;
    /// inst_mem[10+2+6]=8'b0000_0000;
```

```
    /// inst_mem[10+2+7]=8'b0000_0000;
///
    /// inst_mem[20]=8'b0011_0110;    //xorq
    /// inst_mem[20+1]=8'b0000_0000;
///
    ///
    /// inst_mem[22]=8'b0100_0111;     //jne
    /// inst_mem[22+1]=8'b0100_0000;
    /// inst_mem[22+2]=8'b0000_0000;
    /// inst_mem[22+3]=8'b0000_0000;
    /// inst_mem[22+4]=8'b0000_0000;
    /// inst_mem[22+5]=8'b0000_0000;
    /// inst_mem[22+6]=8'b0000_0000;
    /// inst_mem[22+7]=8'b0000_0000;
    /// inst_mem[22+8]=8'b0000_0000;
///
    /// inst_mem[64]=8'b0000_0011;     //irmovq
    /// inst_mem[65]=8'b0010_1111;
    /// inst_mem[64+2+0]=8'b0000_0010;
    /// inst_mem[64+2+1]=8'b0000_0000;
    /// inst_mem[64+2+2]=8'b0000_0000;
    /// inst_mem[64+2+3]=8'b0000_0000;
    /// inst_mem[64+2+4]=8'b0000_0000;
    /// inst_mem[64+2+5]=8'b0000_0000;
    /// inst_mem[64+2+6]=8'b0000_0000;
    /// inst_mem[64+2+7]=8'b0000_0000;
///
    /// inst_mem[74]=8'b0000_0011;     //irmovq
    /// inst_mem[75]=8'b0011_1111;
    /// inst_mem[74+2+0]=8'b0000_0011;
    /// inst_mem[74+2+1]=8'b0000_0000;
    /// inst_mem[74+2+2]=8'b0000_0000;
    /// inst_mem[74+2+3]=8'b0000_0000;
    /// inst_mem[74+2+4]=8'b0000_0000;
    /// inst_mem[74+2+5]=8'b0000_0000;
    /// inst_mem[74+2+6]=8'b0000_0000;
    /// inst_mem[74+2+7]=8'b0000_0000;
///
    /// inst_mem[31]=8'b0000_0011;     //irmovq
    /// inst_mem[32]=8'b0000_1111;
    /// inst_mem[31+2+0]=8'b0000_0001;
    /// inst_mem[31+2+1]=8'b0000_0000;
    /// inst_mem[31+2+2]=8'b0000_0000;
    /// inst_mem[31+2+3]=8'b0000_0000;
    /// inst_mem[31+2+4]=8'b0000_0000;
    /// inst_mem[31+2+5]=8'b0000_0000;
    /// inst_mem[31+2+6]=8'b0000_0000;
    /// inst_mem[31+2+7]=8'b0000_0000;
```

```
 ///
///
    /// inst_mem[41]=8'b0000_0110;     //addq
    /// inst_mem[42]=8'b0000_0000;
////----------------End of Branch-Misprediction------------
```

Output:

3. **Ret control hazard**

   ASM code:

   Irmovq $100, %rsp

   Call p:

   Irmovq $5, %rsi

   P:

   Irmovq $5, %rdi

   Ret

   Irmovq $1, %rax

Verilog code:

```
 //////-----------------Ret---------------------
    // inst_mem[10]=8'b0000_0011;     //irmovq
    // inst_mem[11]=8'b0100_1111;
    // inst_mem[10+2+0]=8'b0110_0100;
    // inst_mem[10+2+1]=8'b0000_0000;
    // inst_mem[10+2+2]=8'b0000_0000;
    // inst_mem[10+2+3]=8'b0000_0000;
    // inst_mem[10+2+4]=8'b0000_0000;
    // inst_mem[10+2+5]=8'b0000_0000;
    // inst_mem[10+2+6]=8'b0000_0000;
    // inst_mem[10+2+7]=8'b0000_0000;
//
    // inst_mem[20]=8'b0000_1000;     //call
    // inst_mem[20+1]=8'b0100_0000;
    // inst_mem[20+2]=8'b0000_0000;
    // inst_mem[20+3]=8'b0000_0000;
    // inst_mem[20+4]=8'b0000_0000;
    // inst_mem[20+5]=8'b0000_0000;
    // inst_mem[20+6]=8'b0000_0000;
    // inst_mem[20+7]=8'b0000_0000;
    // inst_mem[20+8]=8'b0000_0000;
//
    // inst_mem[29]=8'b0000_0011;     //irmovq
    // inst_mem[30]=8'b0110_1111;
    // inst_mem[29+2+0]=8'b0000_0101;
    // inst_mem[29+2+1]=8'b0000_0000;
    // inst_mem[29+2+2]=8'b0000_0000;
    // inst_mem[29+2+3]=8'b0000_0000;
    // inst_mem[29+2+4]=8'b0000_0000;
    // inst_mem[29+2+5]=8'b0000_0000;
    // inst_mem[29+2+6]=8'b0000_0000;
    // inst_mem[29+2+7]=8'b0000_0000;
//
    // inst_mem[64]=8'b0000_0011;     //irmovq       Target
    // inst_mem[65]=8'b0111_1111;
    // inst_mem[64+2+0]=8'b0000_0101;
    // inst_mem[64+2+1]=8'b0000_0000;
    // inst_mem[64+2+2]=8'b0000_0000;
    // inst_mem[64+2+3]=8'b0000_0000;
    // inst_mem[64+2+4]=8'b0000_0000;
    // inst_mem[64+2+5]=8'b0000_0000;
    // inst_mem[64+2+6]=8'b0000_0000;
    // inst_mem[64+2+7]=8'b0000_0000;
//
    // inst_mem[74]=8'b0000_1001;   //ret
//
```

```
    // inst_mem[75]=8'b0000_0011;      //irmovq
    // inst_mem[76]=8'b0000_1111;
    // inst_mem[75+2+0]=8'b0000_0001;
    // inst_mem[75+2+1]=8'b0000_0000;
    // inst_mem[75+2+2]=8'b0000_0000;
    // inst_mem[75+2+3]=8'b0000_0000;
    // inst_mem[75+2+4]=8'b0000_0000;
    // inst_mem[75+2+5]=8'b0000_0000;
    // inst_mem[75+2+6]=8'b0000_0000;
    // inst_mem[75+2+7]=8'b0000_0000;

//--------------------End of Ret-------------------------------
```

Output:

7. Challenges faced

1. Selecting the location of register file.
   The register file is used by decode and write-back stages and thus needs to be instantiated separately if we create different modules for both stages. We tried this in sequential and it worked successfully. But in order to ease up the process of presenting register file as output, we combined both the stages.
2. Input delay in testbench.

We set out input delay to 5 (#5) for all combinations of instructions. In case the instruction is taking longer than delay provided, the stage would give incorrect output.

3. Ret instruction
   The return instruction worked perfectly with stages of all the output correct. The problem is that it displayed incorrect value of predicted PC but still successfully processed outputs for subsequent stages.
4. Full adder.
   Full adder which was required in ALU did not work for unknown reason. We created an identical full adder which worked correctly. We ran the first full adder on different machines but the error persisted. We look forward to understand Verilog better and find the reason behind it.