Y-86 Processor Final Report

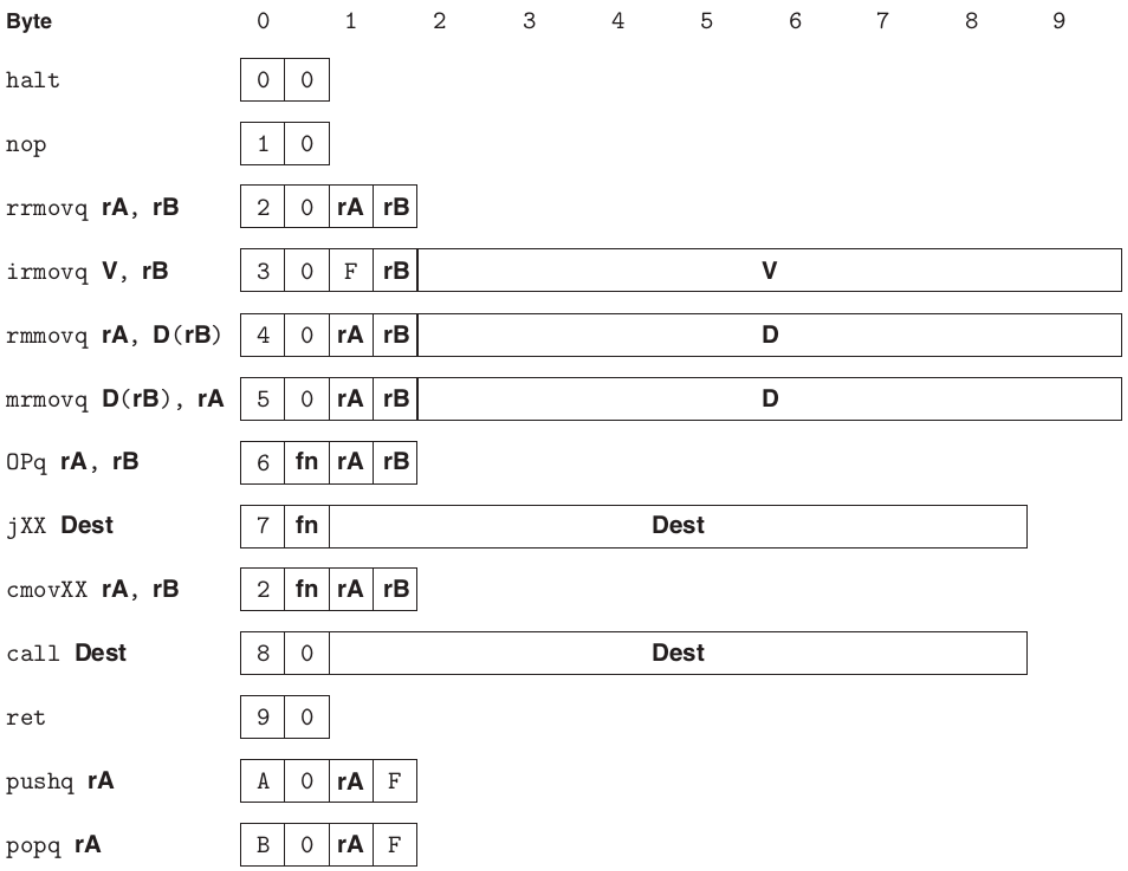
Submitted by:- Harshvardhan Singh (2022112004)
Janya Gupta (2022102033)

Objective:-

To implement Y86-Processor in Verilog using both the SEQ and Pipelining implementations.

Instruction Encodings:-

The instructions were addressed according to the following encodings:-

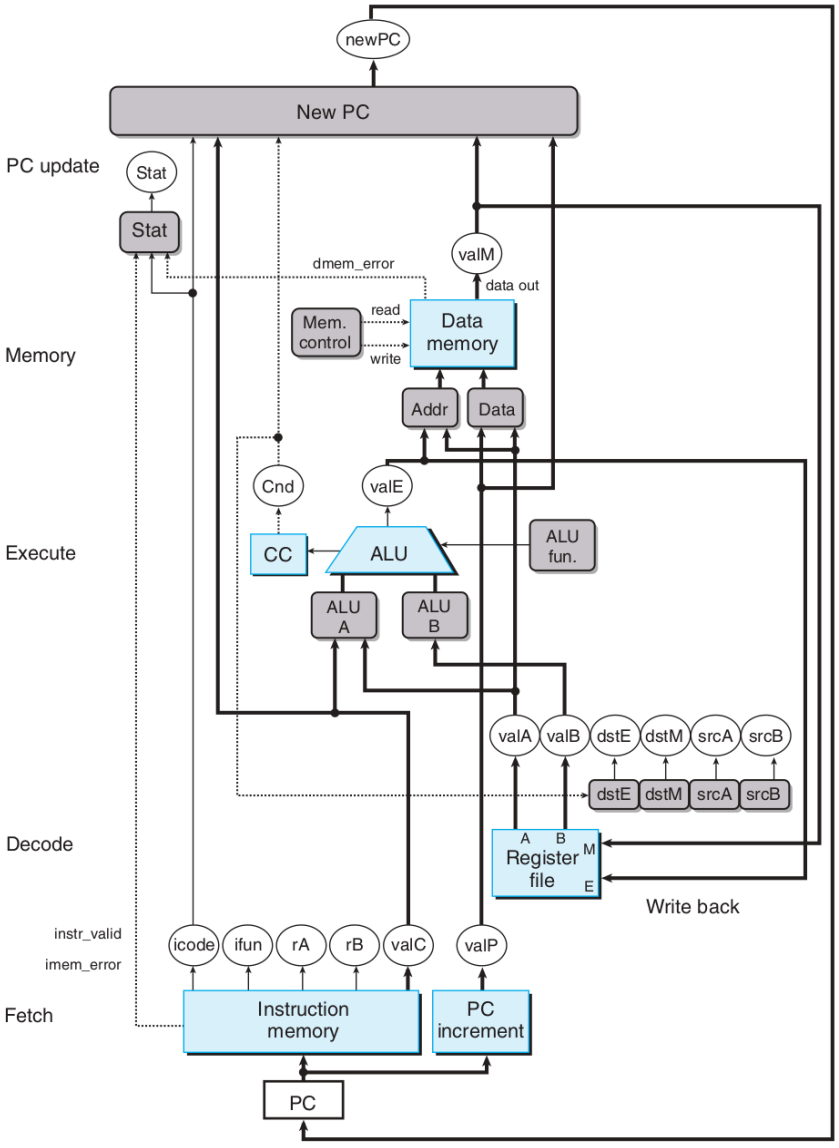


Y86-64 Instruction set

Operations			Branches				Moves					
addq	6	0	jmp	7	0	jne	7	4	rrmovq	2	0	cmovne	2	4
subq	6	1	jle	7	1	jge	7	5	cmovle	2	1	cmovge	2	5
andq	6	2	jl	7	2	jg	7	6	cmovl	2	2	cmovg	2	6
xorq	6	3	je	7	3				cmove	2	3			

Function encodings for Y86-64 processor

Sequential Implementation

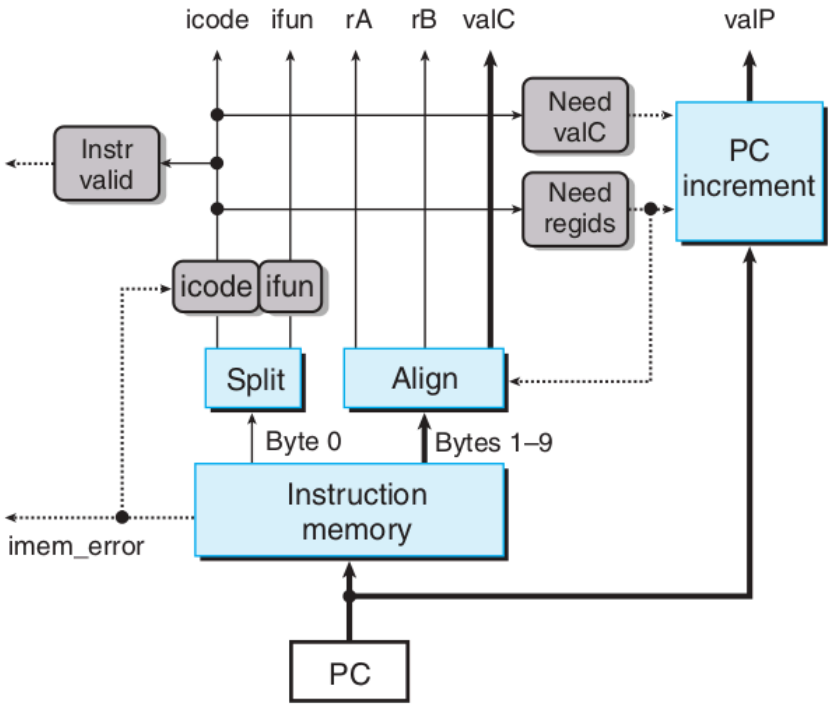


Hardware implementation of SEQ

Fetch:-

The fetch stage reads the bytes of an instruction from memory, using the program counter (PC) as the memory address.
From the instruction it extracts the two 4-bit portions of the instruction specifier byte, referred to as icode (the instruction code) and ifun (the instruction function).
It possibly fetches a register specifier byte, giving one or both of the register operand specifiers rA and rB.
It also possibly fetches an 8-byte constantword valC.
It computes valP to be the address of the instruction followingthe current one in sequential order. That is, valP equals the value of the PC plus the length of the fetched instruction.

Following is the hardware implementation of Fetch:-



Code snippets and explanation:-

```
module fetchmodule(input wire clk, input [63:0] pc,
    output reg [3:0] icode, output reg [3:0] ifun, output reg [3:0] rA, output reg [3:0] rB, output reg [63:0] valC,
    output reg [63:0] valP, output reg ins_address, output reg hlt, output reg adr_address);

    reg[7:0]instruction_memory[0:1023];
    reg [0:79] instruction;
    initial
    begin
        $readmemb("1.txt", instruction_memory);
        hlt=0;
        adr_address=0;
        ins_address=0;
    end
```

We're declaring input and output ports and creating an instruction memory to read to and store the instructions. We're reading instructions from a text file. Initially we take the error bits to be 0.

```
always @(posedge clk)
begin

 instruction= {instruction_memory[pc],instruction_memory[pc+1],instruction_memory[pc+2],instruction_memory[pc+3],
 instruction_memory[pc+4],instruction_memory[pc+5],instruction_memory[pc+6],
 instruction_memory[pc+7],instruction_memory[pc+8],instruction_memory[pc+9]};
 icode = instruction[0:3];
 ifun = instruction[4:7];
```

Extracting the first 10 bytes of the instruction.

```
if(pc>22)
begin
    adr_address=1;
end
```

In the snippet shown above the value 22 can be modified acc to the length of the given instruction memory.

```
else
begin
if(icode== 4'b0000 )
begin
   hlt=1;
 valP = pc +1;
end

 else if(icode== 4'b0001 || icode== 4'b1001)
 begin
   valP= pc+1;
 end

else if( icode== 4'b0010 || icode== 4'b0110 || icode== 4'b1010 || icode== 4'b1011)
 begin
rA= instruction[8:11];
rB= instruction[12:15];
valP= pc+2;
 end

else if(icode== 4'b0011 || icode== 4'b0100 || icode== 4'b0101)
begin
rA= instruction[8:11];
rB= instruction[12:15];
valC= {instruction[72:79],instruction[64:71],instruction[56:63],instruction[48:55],instruction[40:47],
instruction[32:39],instruction[24:31],instruction[16:23]};
valP= pc +10;
end

else if( icode== 4'b1000 || icode== 4'b0111)
begin
valC= {instruction[64:71],instruction[56:63],instruction[48:55],instruction[40:47],instruction[32:39],
instruction[24:31],instruction[16:23],instruction[8:15]};
valP= pc+9;
end

else
begin
ins_address=1;
end
```

rA= instruction[8:11]
rB= instruction[12:15]

valC= because encoding is done in bit reversal format.
Instruction[72:79],instruction[64:71],instruction[56:63],instruction[48:55],instruction[4:47],instruction[32:39],instruction[24:31],instruction[16:23]

And for jmp/call statements:-

Instruction[64:71],instruction[56:63],instruction[48:55],instruction[40:47],instruction[32:39],instruction[24:31],instruction[16:23],instruction[8:15]

And PC update is done acc to the individual conditions as shown in the code.

If the icode doesn't has value in [0,11] then it means it's an invalid instruction and it's flagged through ins_address.

Decode:-

The decode stage reads up to two operands from the register file, giving values valA and/or valB. Typically, it reads the registers designated by instruction fields rA and rB, but for some instructions it reads register %rsp.

Declaring the inputs and outputs:-

```
module decodemodule(input wire clk, input [3:0] rA, input [3:0] rB, input [3:0] icode,
    input [63:0] rax,
    input [63:0] rcx,
    input [63:0] rdx,
    input [63:0] rbx,
    input [63:0] rsp,
    input [63:0] rbp,
    input [63:0] rsi,
    input [63:0] rdi,
    input [63:0] r8,
    input [63:0] r9,
    input [63:0] r10,
    input [63:0] r11,
    input [63:0] r12,
    input [63:0] r13,
    input [63:0] r14,
    output reg[63:0] valA,
    output reg[63:0] valB);
```

Storing all the registers in the register file:-

```
always @(*) begin

  register_file[0]= rax;
  register_file[1]= rcx;
  register_file[2]= rdx;
  register_file[3]= rbx;
  register_file[4]= rsp;
  register_file[5]= rbp;
  register_file[6]= rsi;
  register_file[7]= rdi;
  register_file[8]= r8;
  register_file[9]= r9;
  register_file[10]= r10;
  register_file[11]= r11;
  register_file[12]= r12;
  register_file[13]= r13;
  register_file[14]= r14;
```

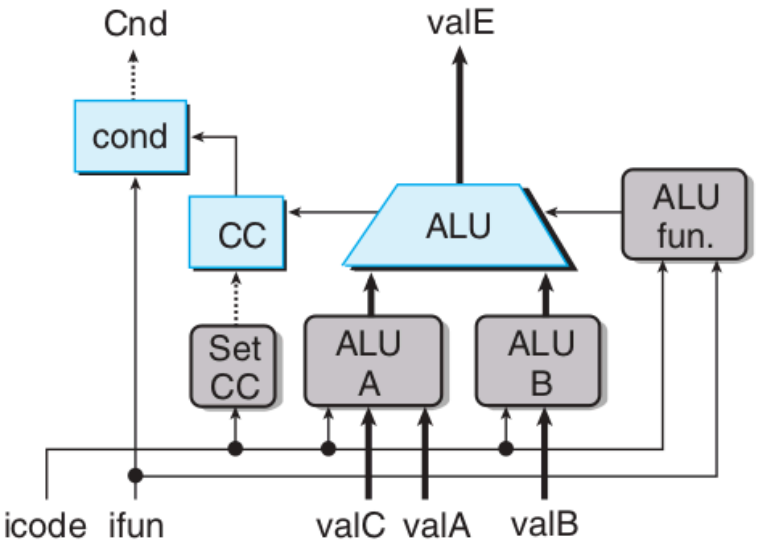
The icode of the instruction decides the values of valA and valB as follows:
− valA = Reg_File[rA] for icode in {RRMOVQ, RMOVQ, OPQ, PUSHQ}
− valA = Reg_File[RRSP] for icode in {POPQ, RET}
− valB = Reg_File[rB] for icode in {RRMOVQ, RMMOVQ, MROVQ, OPQ}
− valB = Reg_File[RRSP] for icode in {CALL, PUSHQ, POPQ, RET}
Implementing the following conditions as:-

```
if(icode== 4'b0010)
begin
  valA= register_file[rA];

end
else if(icode== 4'b0011)
begin

end
else if(icode== 4'b0100)
begin
  valA= register_file[rA];
  valB= register_file[rB];
end
else if(icode== 4'b0101)
begin
  valB= register_file[rB];

end
else if(icode== 4'b0110)begin
  valA= register_file[rA];
  valB= register_file[rB];
end
else if(icode== 4'b0111)
begin

end
else if(icode== 4'b1000)
begin

  valB= register_file[4];
end
else if(icode== 4'b1001)begin
  valA= register_file[4];
  valB= register_file[4];
end
else if(icode== 4'b1010)begin
  valA= register_file[rA];
  valB= register_file[4];
end
else if(icode== 4'b1011)begin
  valA= register_file[4];
  valB= register_file[4];
end
else
```

Execute:-

In the execute stage, the arithmetic/logic unit (ALU) either performs the operation specified by the instruction (according to the value of ifun),

computes the effective address of a memory reference, or increments or decrements the stack pointer. We refer to the resulting value as valE. The condition codes are possibly set. For a conditional move instruction, the stage will evaluate the condition codes and move condition (given by ifun) and enable the updating of the destination register only if the condition holds. Similarly, for a jump instruction, it determines whether or not the branch should be taken.



Hardware implementation of the execute stage

Conditional codes are set in this stage:-

```
always @(*)
begin
    if (icode==4'b0110 && clk ==1)
    begin
        zf = (res==1'b0)&&(ifun==4'b0000||ifun==4'b0001);
        sf = (res[63]==1);
        if (ifun == 4'b0000)
        begin
            of = (a[63]==b[63])&&(res[63]!=a[63]);
        end
        else if (ifun == 4'b0001)
        begin
            of = (a[63]!=b[63])&&(res[63] != a[63]);
        end
    end
end
```

```
if(ifun==4'b0000)//uncon
begin
    cnd = 1;
end
else if(ifun==4'b0001)//le
begin
    cnd = (sf^of)|(zf);
end
else if (ifun==4'b0010)//l
begin
    cnd = (sf^of);
end
else if (ifun==4'b0011)//e
begin
    cnd = zf;
end
else if (ifun==4'b0100)//ne
begin
    cnd = ~zf;
end
else if (ifun==4'b0101)//ge
begin
    cnd = ~(sf^of);
end
else if (ifun==4'b0110)//g
begin
    cnd = ~(sf^of)&(~zf);
end
ValE = 64'd0 + ValA;
```

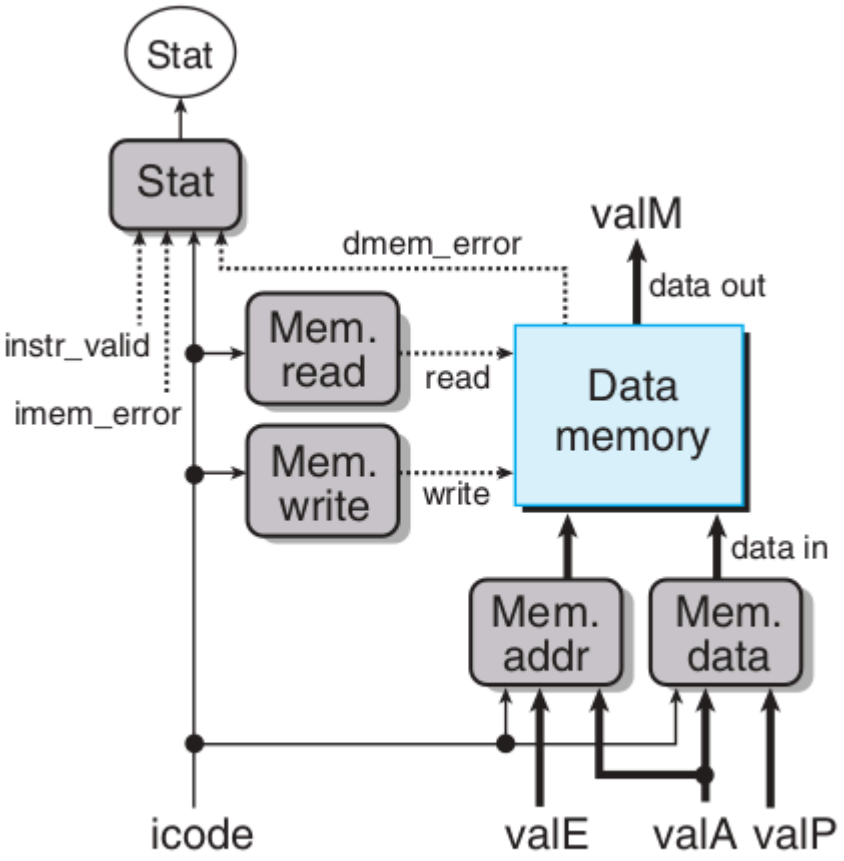
Implementation and working:-

->For ‘cmov’ instruction we set the cnd output according to the condition codes and ifun as specified above. We also set valE = valA.
->For ‘irmovq’ instruction we set the output valE = valC
->For ‘mrmovq’ and ‘rmmovq’ instructions we set the output valE = valB + valC.
-> For ‘opx’ instruction we check ifun code and perform the arthmetic and logical function accordingly. We set valE = ans which is the output from the alu. We also set the condition codes in this instruction.
->For ‘jXX’ instruction we again set the cnd bit according to the condition codes and ifun as specified above.
->For ‘call’ and ‘pushq’ we set the output valE = -64’d1+valB.
-> For ‘ret’ and ‘popq’ instructions we set the output valE = 64’d1+valB.

```
else if (icode==4'b0110)//OP
begin
    if(ifun==4'b0000)//add
    begin
        S0=0;
        S1=0;
        a = ValA;
        b = ValB;
        assign result = res;
    end
    else if (ifun==4'b0001)//sub
    begin
        S0=1;
        S1=0;
        a = ValA;
        b = ValB;
        assign result = res;
    end
    else if (ifun==4'b0010)//and
    begin
        S0=0;
        S1=1;
        a = ValA;
        b = ValB;
        assign result = anded;
    end
    else if (ifun==4'b0011)//xor
    begin
        S0=1;
        S1=1;
        a = ValA;
        b = ValB;
        assign result = xored;
    end
    ValE = result;
end
else if (icode==4'b1000 || icode==4'b1010)//call //push
begin
    ValE = vdec;
end
else if (icode==4'b1001 || icode==4'b1011)//ret //pop
begin
    ValE = vinc;
end
else
begin
    ValE = 64'd0;//default case
end
```

Memory:-

The memory stage may write data to memory, or it may read data from memory. We refer to the value read as valM.



Hardware implementation of memory block

We read and write data to the data memory as follows:

-> We write to the data memory for ‘rmmovq’, ‘call’ and ‘pushq’ instructions

-> We do data_memory[valE] = valA for rmmovq and pushq

-> We do data_memory[valE] = valP for call

-> We read from the data memory for ‘mrmovq’, ‘ret’ and ‘popq’ instructions

▪ We do valM = data_memory[valE] for mrmovq and popq

▪ We do valM = data_memory[valE] for ret

-> If the memory address accessed by the memory block is out of bounds (here greater than 8192) then we set the dmem_error =1.

In this block we declared the data memory as a register array:-

```
reg[63:0] mem[0:8192];
```

Code snippet:-

```
always @(negedge clk)
begin
    if(icode==4'b1001)
    begin
      ValM=mem[ValA];
    end
        if (icode==4'b1010)
    begin
        mem[ValE] = ValA;
    end
        if (icode==4'b1000)
    begin
        mem[ValE] = ValP;
    end
        if (icode==4'b0100)
    begin
        mem[ValE] = ValA;
    end
end
```

For writing and reading from memory we're using the negative edge of the block.

```
always @(*)
begin
    adr_memory=0;
    if(ValE>8192)
    begin
      adr_memory=1;
    end
    else
    begin
    // if (icode==4'b0100)
    // begin
    //     mem[ValE] = ValA;
    // end
    if (icode==4'b0101)
    begin
        ValM = mem[ValE];
    end
    // if (icode==4'b1000)
    // begin
    //     mem[ValE] = ValP;
    // end
    // if (icode==4'b1001)
    // begin
    //     ValM = mem[ValA];
    // end
    if (icode==4'b1011)
    begin
        ValM = mem[ValA];
    end
    // if (icode==4'b1010)
    // begin
    //     mem[ValE] = ValA;
    // end
    //     if(icode==4'b1001)
    // begin
    //   ValM=mem[ValA];
    // end
    data = mem[ValE];
end
end
```

Writeback:-

The register file has two write ports. Port E is used to write values computed by the ALU, while port M is used to write values read from the data memory that is the values valE, or valM are written into the registers with addresses rA, rB or RRSP depending on the icode.

Implementation:-

The values valE or valM are written into register with register code rA, Rb or RRSP depending on the icode as follows:
o Reg_File[rA] = valM for icode in {IMRMOVQ, IPOPQ}
o Reg_File[rB] = valE for icode in {IRRMOVQ & cnd = 1, IIRMOVQ, IOPQ}
o Reg_File[RRSP] = valE for icode in {ICALL, IRET, IPUSHQ, IPOPQ}

```
if (icode==4'b0010)//cmov
begin
    if (cnd==1'b1)
    begin
        register_file[rB] = ValE;
    end
end
if (icode==4'b0011)//irmov
begin
    register_file[rB] = ValE;
end
if (icode==4'b0101)//mrmov
begin
    register_file[rA] = ValM;
end
if (icode==4'b0110)//op
begin
    register_file[rB] = ValE;
end
if (icode==4'b1000)//call
begin
    register_file[4] = ValE;
end
if (icode==4'b1001)//return
begin
    register_file[4] = ValE;
end
if (icode==4'b1010)//push
begin
    register_file[4] = ValE;
end
if (icode==4'b1011)//pop
begin
    register_file[4] = ValE;
    register_file[rA] = ValM;
end
```

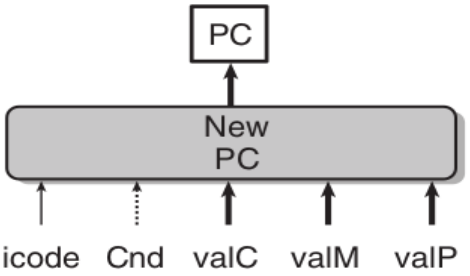
We’re taking the input registers and outputting the registers again which are then written back into the input registers:-

```
input [3:0] rA;
input [3:0] rB;
input [63:0] ValE;
input [63:0] ValM;
input [63:0] rax;
input [63:0] rcx;
input [63:0] rdx;
input [63:0] rbx;
input [63:0] rsp;
input [63:0] rbp;
input [63:0] rsi;
input [63:0] rdi;
input [63:0] r8;
input [63:0] r9;
input [63:0] r10;
input [63:0] r11;
input [63:0] r12;
input [63:0] r13;
input [63:0] r14;

output [63:0] reg0;
output [63:0] reg1;
output [63:0] reg2;
output [63:0] reg3;
output [63:0] reg4;
output [63:0] reg5;
output [63:0] reg6;
output [63:0] reg7;
output [63:0] reg8;
output [63:0] reg9;
output [63:0] reg10;
output [63:0] reg11;
output [63:0] reg12;
output [63:0] reg13;
output [63:0] reg14;

reg [63:0] register_file[0:15];
always @(negedge clk)
begin
    register_file[0] = rax;
    register_file[1] = rcx;
    register_file[2] = rdx;
    register_file[3] = rbx;
    register_file[4] = rsp;
    register_file[5] = rbp;
    register_file[6] = rsi;
    register_file[7] = rdi;
    register_file[8] = r8;
    register_file[9] = r9;
    register_file[10] = r10;
    register_file[11] = r11;
    register_file[12] = r12;
    register_file[13] = r13;
    register_file[14] = r14;
```

```
assign reg0 = register_file[0];
assign reg1 = register_file[1];
assign reg2 = register_file[2];
assign reg3 = register_file[3];
assign reg4 = register_file[4];
assign reg5 = register_file[5];
assign reg6 = register_file[6];
assign reg7 = register_file[7];
assign reg8 = register_file[8];
assign reg9 = register_file[9];
assign reg10 = register_file[10];
assign reg11 = register_file[11];
assign reg12 = register_file[12];
assign reg13 = register_file[13];
assign reg14 = register_file[14];
```

PC Update:-

In this stage, the PC is set to the address of the next instruction. The value of updated_pc depends on the values of icode, valC, valM, valP and cnd.



Hardware Implementation of PC update.

The value of the updated_pc is decided as follows

->For ‘jXX’ instruction if the cnd = 1 then updated_pc = valC otherwise updated_pc = valP

-> For ‘call’ instruction updated_pc = valC

-> For ‘ret’ instruction updated_pc = valM

-> For all the other instructions updated_pc = valP

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.

```
initial begin
newPc=0;
end
always @(*)
begin
    // if (icode==4'b0000)//halt
    // begin
    //     newPc = 0;
    // end
    if (icode==4'b0111)//jump
    begin
        if (cnd==1)
        begin
            newPc = ValC;
        end
        else
        begin
            newPc = ValP;
        end
    end
    else if (icode==4'b1000)//call
    begin
        newPc = ValC;
    end
    else if (icode==4'b1001)//ret
    begin
        newPc = ValM;
    end
    else//base case
    begin
        newPc = ValP;
    end

end
endmodule
```

Sequential Processor:-

The compiled version of all of the stages executing together:-

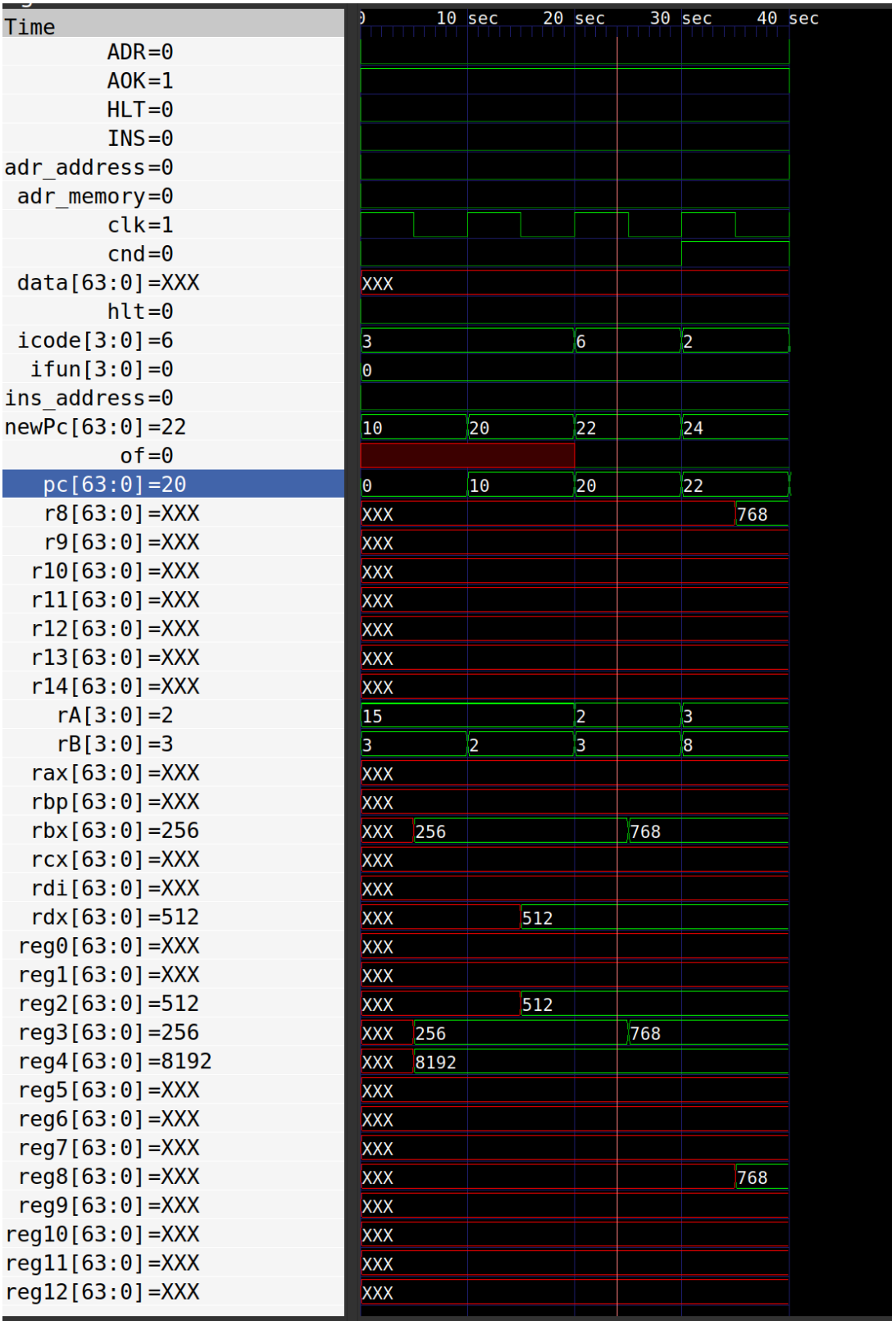
```
module finalmodule;
    reg clk;
    reg [63:0]pc;

    wire[3:0]icode;
    wire[3:0]ifun;
    wire[3:0]rA;
    wire[3:0]rB;
    wire signed [63:0]valC;
    wire [63:0]valP;
    wire ins_address;
    wire adr_address;
    wire hlt;
    wire signed [63:0]valA;
    wire signed [63:0]valB;
    wire signed [63:0]valE;
    wire cnd;
    wire zf;
    wire sf;
    wire of;
    wire signed [63:0]valM;
    wire adr_memory;
    wire signed [63:0]data;
    wire[63:0]newPc;

    reg AOK;
    reg HLT;
    reg ADR;
    reg INS;

    reg signed [63:0] rax;
    reg signed [63:0] rcx;
    reg signed [63:0] rdx;
    reg signed [63:0] rbx;
    reg signed [63:0] rsp;
    reg signed [63:0] rbp;
    reg signed [63:0] rsi;
    reg signed [63:0] rdi;
    reg signed [63:0] r8;
    reg signed [63:0] r9;
    reg signed [63:0] r10;
    reg signed [63:0] r11;
    reg signed [63:0] r12;
    reg signed [63:0] r13;
    reg signed [63:0] r14;
```

```
wire signed [63:0] reg0;
wire signed [63:0] reg1;
wire signed [63:0] reg2;
wire signed [63:0] reg3;
wire signed [63:0] reg4;
wire signed [63:0] reg5;
wire signed [63:0] reg6;
wire signed [63:0] reg7;
wire signed [63:0] reg8;
wire signed [63:0] reg9;
wire signed [63:0] reg10;
wire signed [63:0] reg11;
wire signed [63:0] reg12;
wire signed [63:0] reg13;
wire signed [63:0] reg14;

fetchmodule fetchmodule1(clk,pc,icode,ifun,rA,rB,valC,valP,ins_address,hlt,adr_address);
decodemodule decodemodule1(clk,rA,rB,icode,rax,rcx,rdx,rbx,rsp,rbp,rsi,rdi,r8,r9,r10,r11,r12,r13,r14, valA,valB);
execute execute1(clk,icode,ifun,valA,valB,valC,valE,cnd,zf,sf,of);
memory memory1(clk,icode ,valA, valE,valP,data ,valM,adr_memory);
writeback writeback1(clk,icode,cnd,rA,rB,valE,valM,rax,rcx,rdx,rbx,rsp,rbp,rsi,rdi,r8,r9,r10,r11,r12,r13,r14,reg0,reg1,reg2,reg3,
reg4,reg5,reg6,reg7,reg8,reg9,reg10,reg11,reg12,reg13,reg14);
pc_update pc_update1(clk,cnd,icode,valP,valC,valM,newPc);

initial
begin
    $dumpfile("seq.vcd");
$dumpvars(0,finalmodule);
  pc=0;
  AOK=1;
  HLT=0;
  ADR=0;
  INS=0;
  clk=1;
  rsp=8192;
end
 always #5 clk=~clk;
always @*
begin
    if(hlt==1)
    begin
        HLT=1'b1;
        AOK=1'b0;
        #1 $finish;
    end
    if(adr_address==1)
    begin
        AOK= 1'b0;
        ADR= 1'b1;
        $finish;
    end
    if(ins_address==1)
    begin
        INS= 1'b1;
        AOK= 1'b0;
    end
end
```

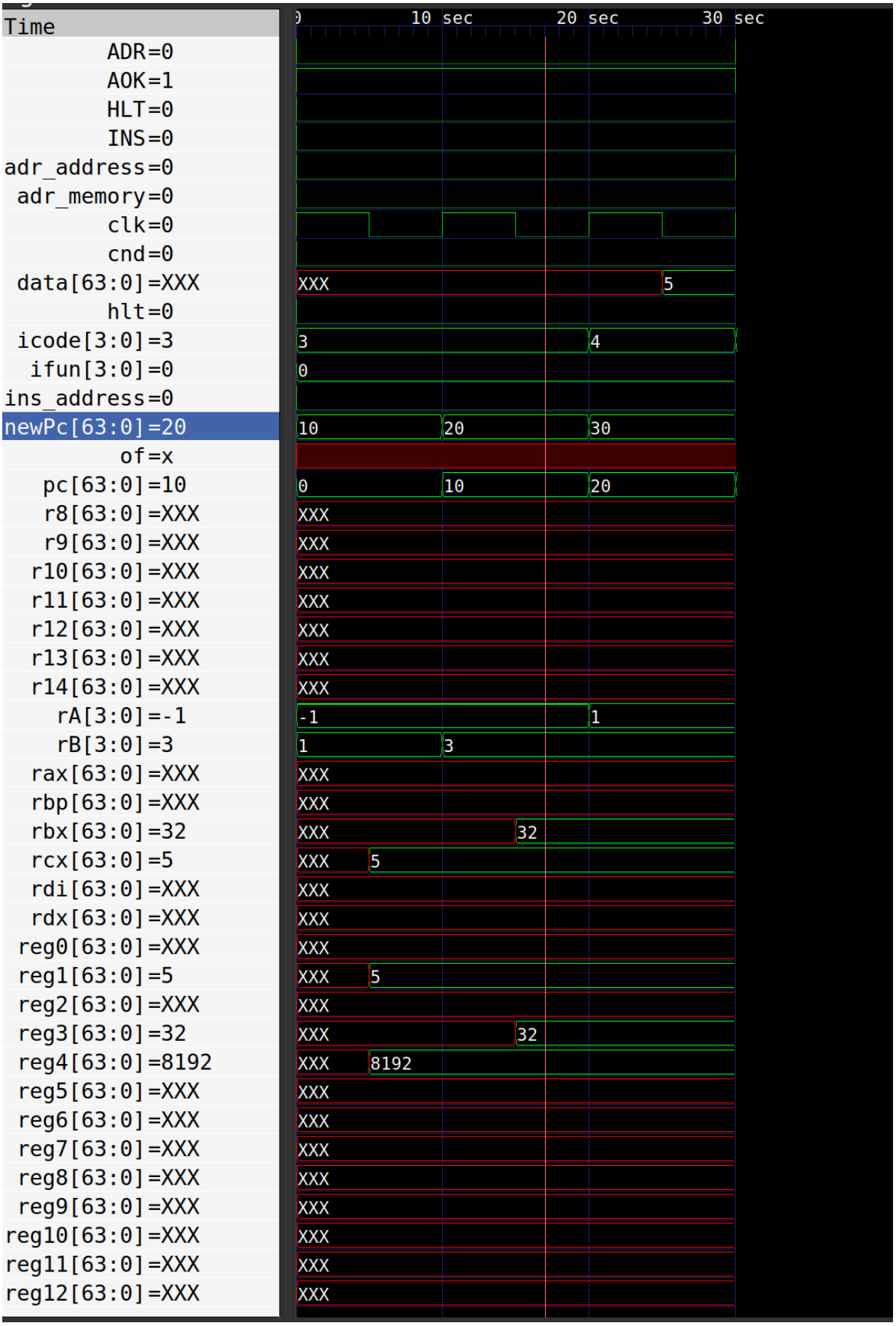
```
 always @*
  begin
  rax=reg0;
  rcx=reg1;
  rdx=reg2;
  rbx=reg3;
  rsp=reg4;
  rbp=reg5;
  rsi=reg6;
  rdi=reg7;
  r8=reg8;
  r9=reg9;
  r10=reg10;
  r11=reg11;
  r12=reg12;
  r13=reg13;
  r14=reg14;
  end
```

Outputs:-

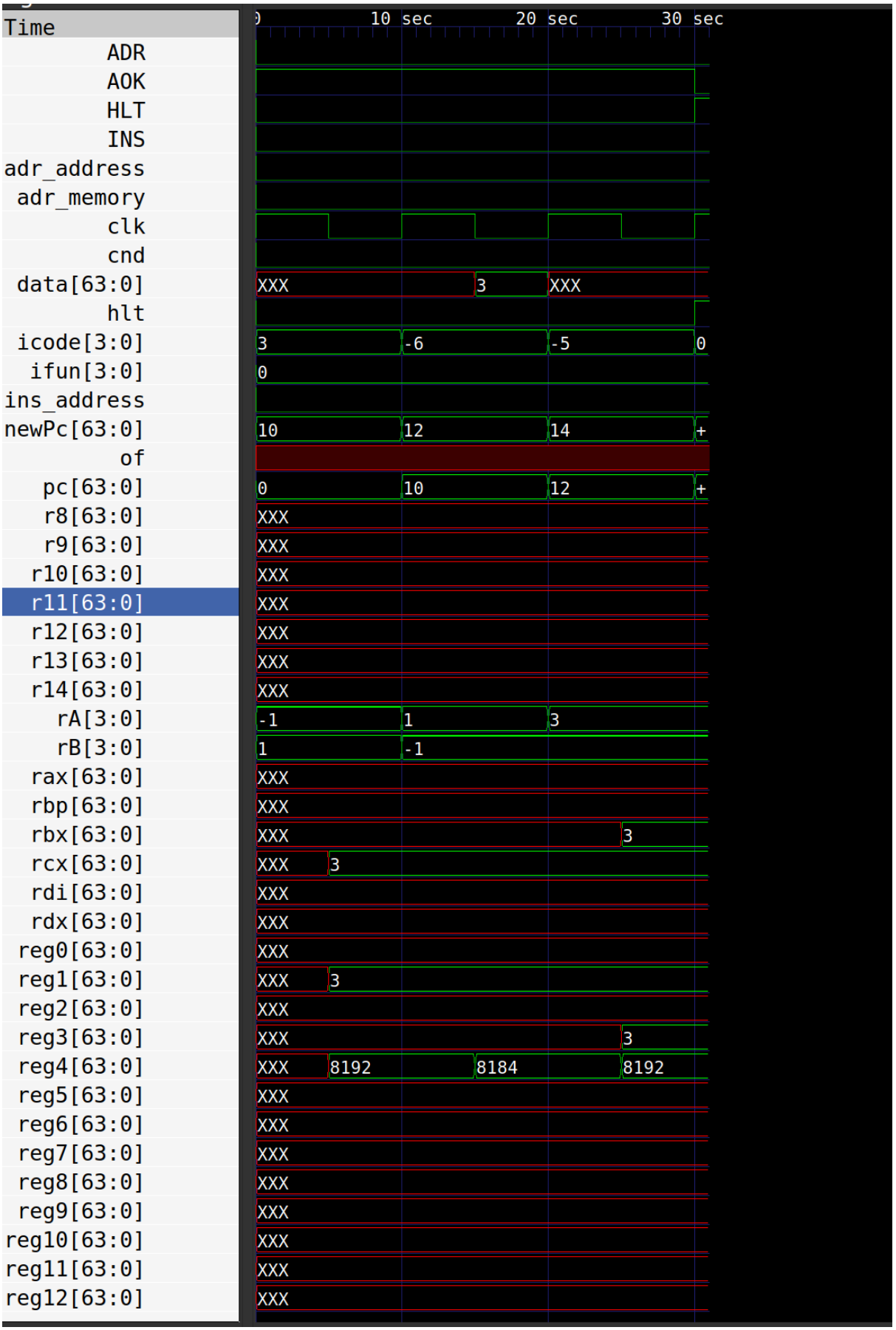
1)Sample.txt



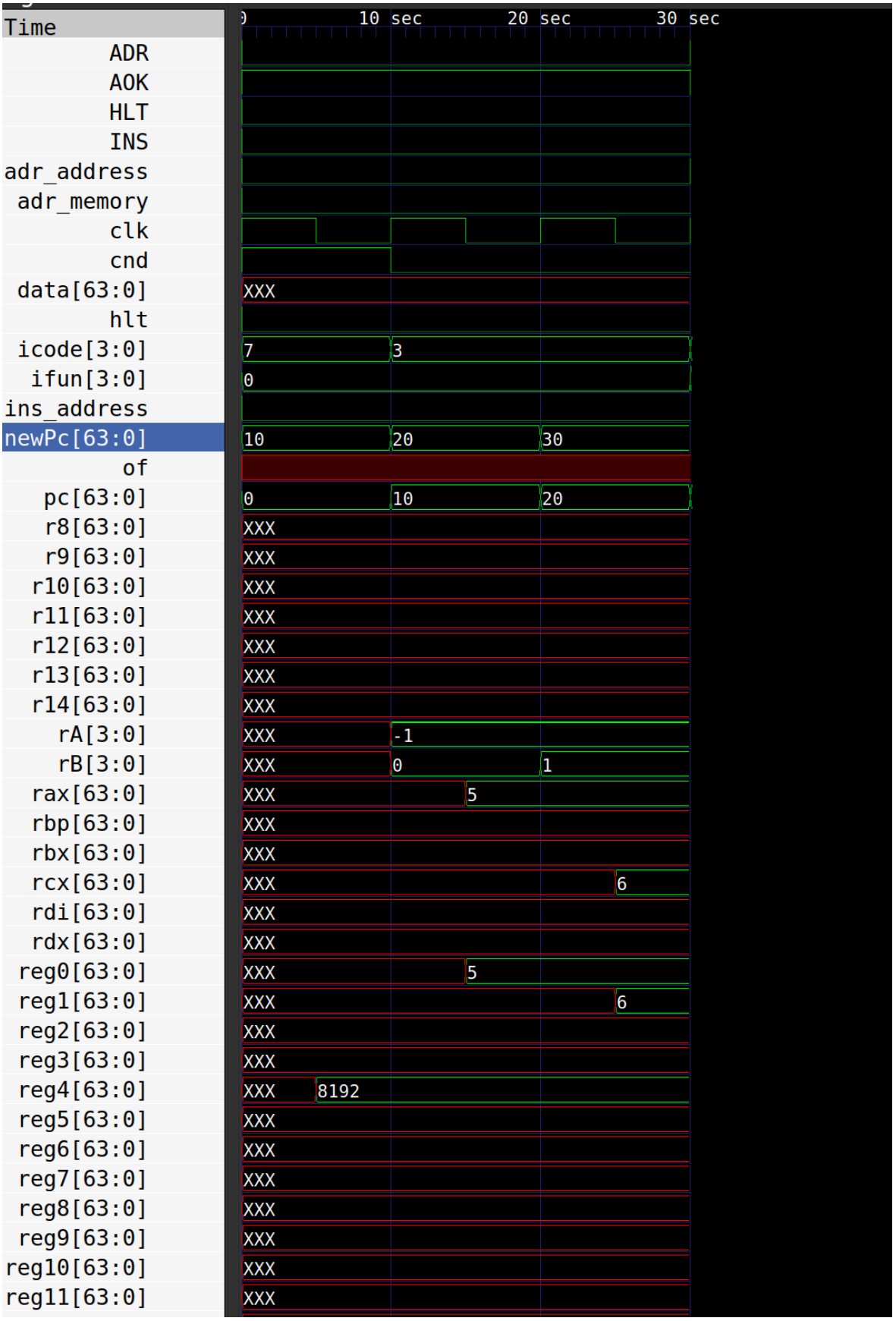
2)RMMR


Time
ADR=0
AOK=1
HLT=0
INS=0
adr_address=0
adr_memory=0
clk=0
cnd=0
data[63:0]=XXX
hlt=0
icode[3:0]=3
ifun[3:0]=0
ins_address=0
newPc[63:0]=20
of=x
pc[63:0]=10
r8[63:0]=XXX
r9[63:0]=XXX
r10[63:0]=XXX
r11[63:0]=XXX
r12[63:0]=XXX
r13[63:0]=XXX
r14[63:0]=XXX
rA[3:0]=-1
rB[3:0]=3
rax[63:0]=XXX
rbp[63:0]=XXX
rbx[63:0]=32
rcx[63:0]=5
rdi[63:0]=XXX
rdx[63:0]=XXX
reg0[63:0]=XXX
reg1[63:0]=5
reg2[63:0]=XXX
reg3[63:0]=32
reg4[63:0]=8192
reg5[63:0]=XXX
reg6[63:0]=XXX
reg7[63:0]=XXX
reg8[63:0]=XXX
reg9[63:0]=XXX
reg10[63:0]=XXX
reg11[63:0]=XXX
reg12[63:0]=XXX
10 sec
20 sec
30 sec

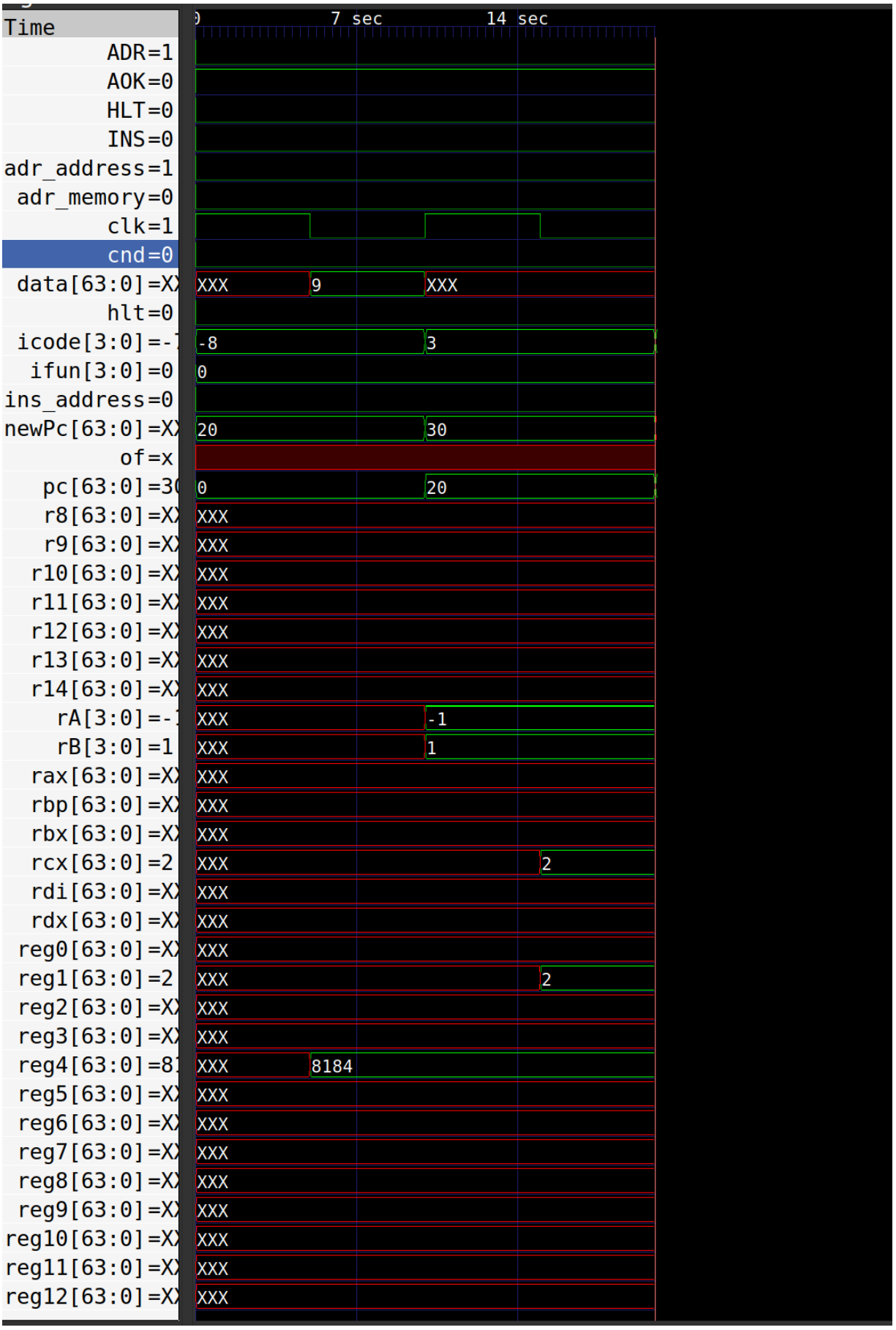
3)Pushpop


Time
ADR
AOK
HLT
INS
adr_address
adr_memory
clk
cnd
data[63:0]
hlt
icode[3:0]
ifun[3:0]
ins_address
newPc[63:0]
of
pc[63:0]
r8[63:0]
r9[63:0]
r10[63:0]
r11[63:0]
r12[63:0]
r13[63:0]
r14[63:0]
rA[3:0]
rB[3:0]
rax[63:0]
rbp[63:0]
rbx[63:0]
rcx[63:0]
rdi[63:0]
rdx[63:0]
reg0[63:0]
reg1[63:0]
reg2[63:0]
reg3[63:0]
reg4[63:0]
reg5[63:0]
reg6[63:0]
reg7[63:0]
reg8[63:0]
reg9[63:0]
reg10[63:0]
reg11[63:0]
reg12[63:0]
10 sec
20 sec
30 sec

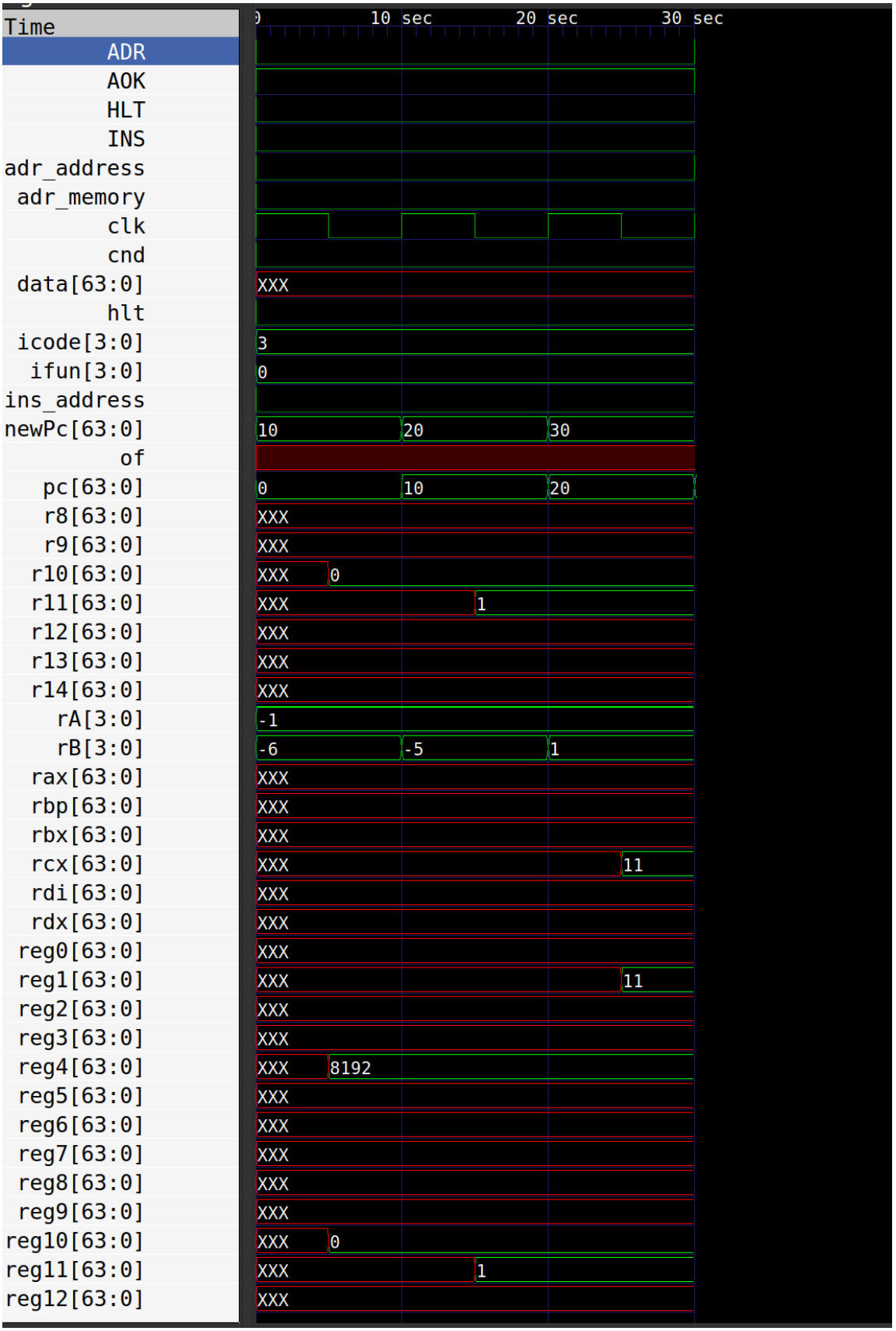
4)JmpCmov


Time
ADR
AOK
HLT
INS
adr_address
adr_memory
clk
cnd
data[63:0]
hlt
icode[3:0]
ifun[3:0]
ins_address
newPc[63:0]
of
pc[63:0]
r8[63:0]
r9[63:0]
r10[63:0]
r11[63:0]
r12[63:0]
r13[63:0]
r14[63:0]
rA[3:0]
rB[3:0]
rax[63:0]
rbp[63:0]
rbx[63:0]
rcx[63:0]
rdi[63:0]
rdx[63:0]
reg0[63:0]
reg1[63:0]
reg2[63:0]
reg3[63:0]
reg4[63:0]
reg5[63:0]
reg6[63:0]
reg7[63:0]
reg8[63:0]
reg9[63:0]
reg10[63:0]
reg11[63:0]
10 sec
20 sec
30 sec

5)CallRet

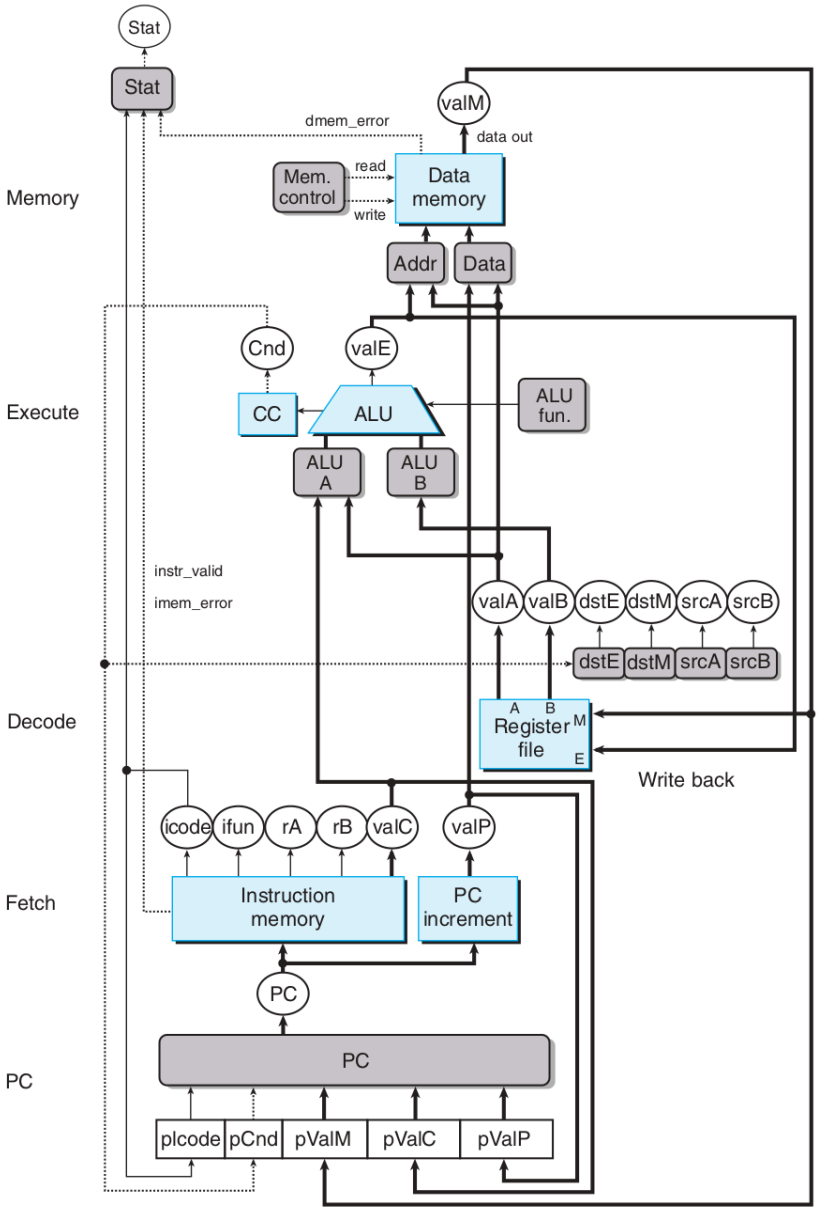

Time
7 sec
14 sec
ADR=1
AOK=0
HLT=0
INS=0
adr_address=1
adr_memory=0
clk=1
cnd=0
data[63:0]=X
hlt=0
icode[3:0]=-
ifun[3:0]=0
ins_address=0
newPc[63:0]=X
of=x
pc[63:0]=3
r8[63:0]=X
r9[63:0]=X
r10[63:0]=X
r11[63:0]=X
r12[63:0]=X
r13[63:0]=X
r14[63:0]=X
rA[3:0]=-
rB[3:0]=1
rax[63:0]=X
rbp[63:0]=X
rbx[63:0]=X
rcx[63:0]=2
rdi[63:0]=X
rdx[63:0]=X
reg0[63:0]=X
reg1[63:0]=2
reg2[63:0]=X
reg3[63:0]=X
reg4[63:0]=8
reg5[63:0]=X
reg6[63:0]=X
reg7[63:0]=X
reg8[63:0]=X
reg9[63:0]=X
reg10[63:0]=X
reg11[63:0]=X
reg12[63:0]=X
XXX
9
-8
3
0
20
30
0
-1
1
2
8184

6) Fibonacci


Time
ADR
AOK
HLT
INS
adr_address
adr_memory
clk
cnd
data[63:0]
hlt
icode[3:0]
ifun[3:0]
ins_address
newPc[63:0]
of
pc[63:0]
r8[63:0]
r9[63:0]
r10[63:0]
r11[63:0]
r12[63:0]
r13[63:0]
r14[63:0]
rA[3:0]
rB[3:0]
rax[63:0]
rbp[63:0]
rbx[63:0]
rcx[63:0]
rdi[63:0]
rdx[63:0]
reg0[63:0]
reg1[63:0]
reg2[63:0]
reg3[63:0]
reg4[63:0]
reg5[63:0]
reg6[63:0]
reg7[63:0]
reg8[63:0]
reg9[63:0]
reg10[63:0]
reg11[63:0]
reg12[63:0]
10 sec
20 sec
30 sec

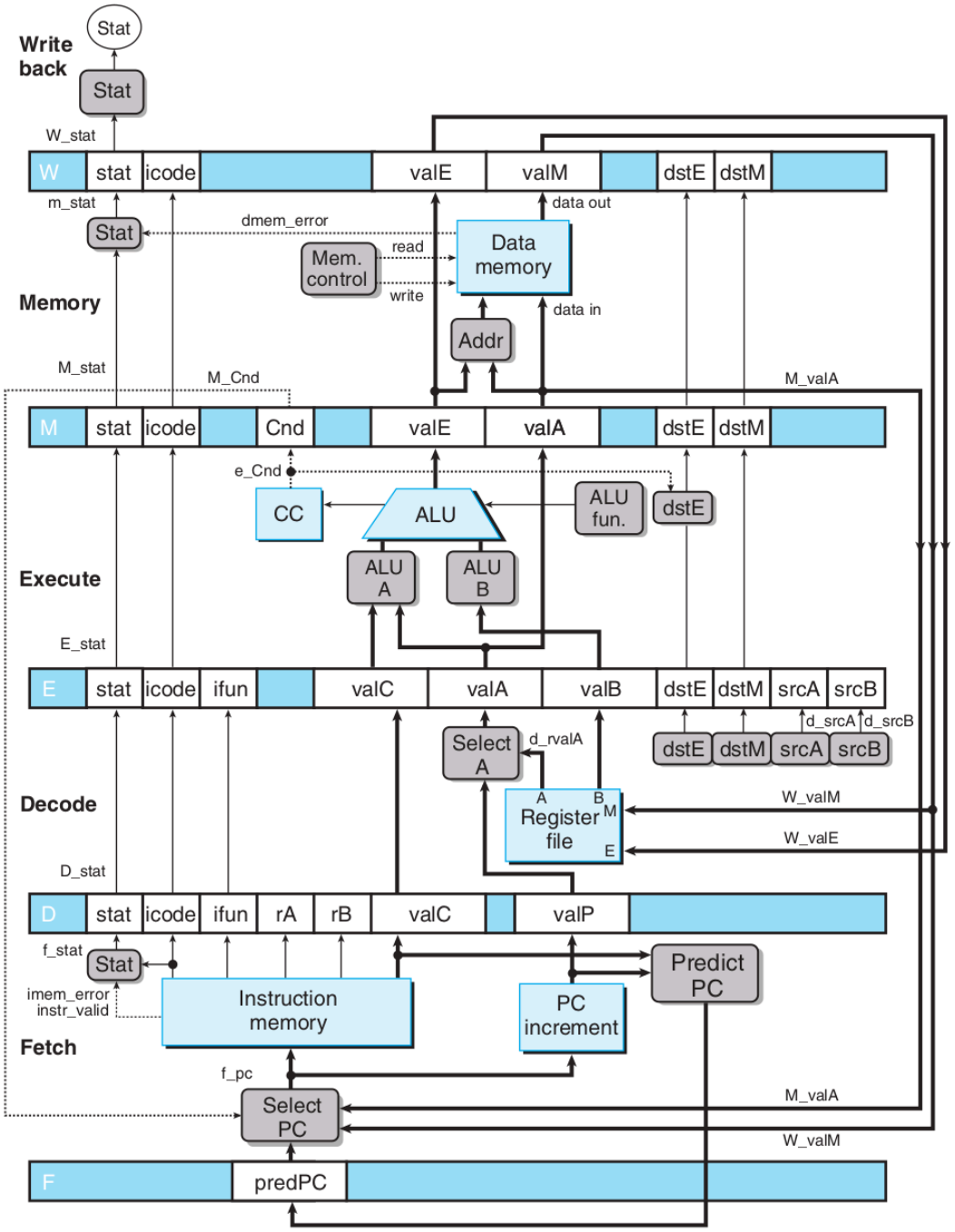
5-Stage Pipelined Processor

A pipelined processor is a type of computer processor architecture that allows multiple instructions to be executed simultaneously, by breaking the instruction execution cycle into smaller stages. As a result, multiple instructions can be in various stages of execution at the same time. This overlapping of instructions allows the processor to achieve higher performance and throughput.
We insert pipeline registers between the stages of SEQ+ and rearrange signals somewhat, yielding the PIPE− processor, where the "−" in the name signifies that this processor has somewhat less performance than our ultimate processor design.
The pipeline registers are shown in this figure as blue boxes, each containing different fields that are shown as white boxes. As indicated by the multiple fields, each pipeline register holds multiple bytes and words.



SEQ+ hardware structure

The PIPE− uses nearly the same set of hardware units as our sequential design SEQ but with the pipeline registers separating the stages. These registers stop the signals from flowing into the next stage and affect the processing happening there.
->F
Holds a predicted value of the program counter, as will be discussed shortly.
-> D
Sits between the fetch and decode stages. It holds information about the most recently
fetched instruction for processing by the decode stage.
->E
Sits between the decode and execute stages. It holds information about the most recently decoded instruction and the values read from the register file for processing by the execute stage.
-> M
Sits between the execute and memory stages. It holds the results of the most recently
executed instruction for processing by the memory stage. It also holds information about
branch conditions and branch targets for processing conditional jumps.
-> W
Sits between the memory stage and the feedback paths that supply the computed results to
the register file for writing and the return address to the PC selection logic when completing a ret instruction.



Hardware implementation of pipelined processor

PC Selection and Fetch

This stage selects a current value for the program counter and predicts the next PC value.

Implementation:

-> The logic to obtain the values of icode, ifun, rA, rB, and valC are the same as in the fetch stage of sequential processor.

-> The PC selection logic chooses between three program counter sources.

o f_PC = M_valA if M_icode = IJXX and M_cnd = 0

o f_PC = W_valM if W_icode == IRET

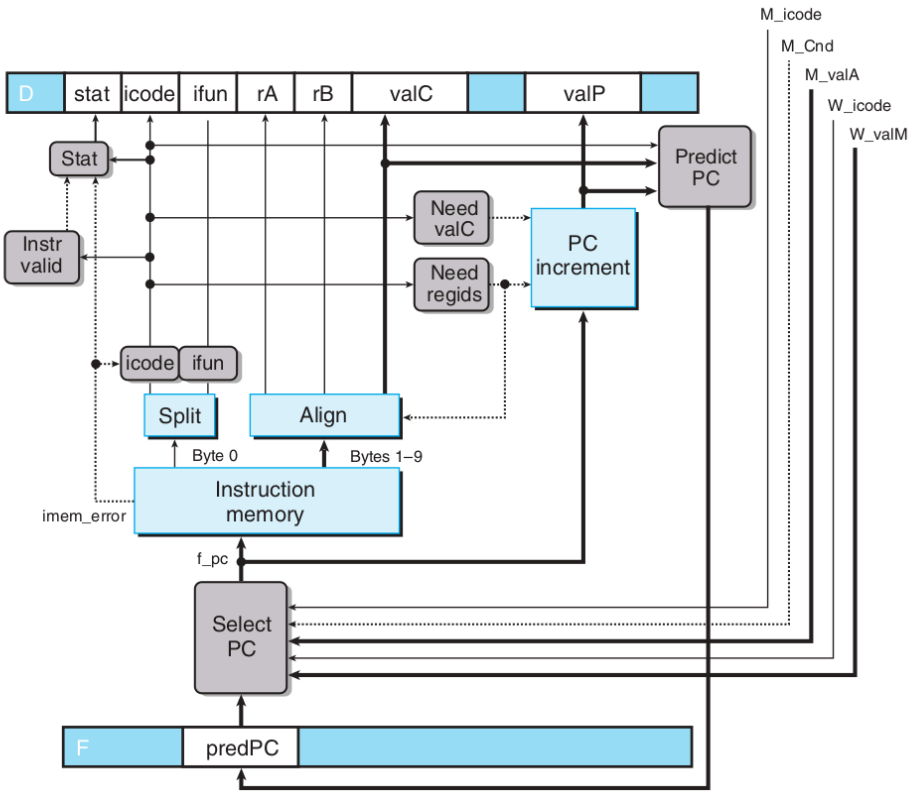
o f_PC = F_predPC in all other cases

o The PC prediction logic is as follows:

o F_predPC = f_valC if icode is in {IJXX, ICALL}

o F_predPC = f_valP in all other cases

-> At positive edge all the values are stored into the pipelined register variables.



Hardware implementation of PC selection and fetch stage

The above mentioned three steps have been written in three different files to make the code more modular.

Fetch:-

```
module fetchmodule(input wire clk, input[63:0] pc, input M_cnd, input[63:0] M_valA, input[63:0] W_valM,
input F_stall, input D_stall, input D_bubble, input[3:0] M_icode, input[3:0] W_icode,
output reg [63:0] f_pc, output reg[3:0] D_icode, output reg[3:0] D_ifun, output reg[3:0] D_rA, output reg[3:0] D_rB,
output reg [63:0] D_valC, output reg [63:0] D_valP, output reg[1:0] D_stat);

//reg[63:0] pc;
reg[3:0] rA;
reg[3:0] rB;
reg[63:0] valC;
reg[63:0] valP;
reg[1:0] stat;
reg ins_address;
reg adr_address;
reg[3:0] icode;
reg[3:0] ifun;

reg[0:79] instruction;
reg[7:0]instruction_memory[0:1023];
initial
    begin
        $readmemb("loaduse.txt", instruction_memory);
        stat=2'd0;
        adr_address=0;
        ins_address=0;
    end

always @(*)
    begin
    if(pc>120)
      begin
        adr_address=1;
        stat= 2'd2;
      end
```

```
    else
    begin

    instruction= {instruction_memory[pc],instruction_memory[pc+1],instruction_memory[pc+2],instruction_memory[pc+3],
    instruction_memory[pc+4],instruction_memory[pc+5],instruction_memory[pc+6],instruction_memory[pc+7],instruction_memory[pc+8],instruction_memory[pc+9]};
    icode = instruction[0:3];
    ifun = instruction[4:7];

    if(icode== 4'b0000 )
    begin
       stat=2'd1;
     valP = pc +1;
     f_pc= valP;
    end

     else if(icode== 4'b0001 || icode== 4'b1001)
     begin
       valP= pc+1;
       f_pc= valP;
     end

    else if( icode== 4'b0010 || icode== 4'b0110 || icode== 4'b1010 || icode== 4'b1011)
    begin
   rA= instruction[8:11];
   rB= instruction[12:15];
   valP= pc+2;
   f_pc= valP;
    end

   else if(icode== 4'b0011 || icode== 4'b0100 || icode== 4'b0101)
   begin
   rA= instruction[8:11];
   rB= instruction[12:15];
   valC= {instruction[72:79],instruction[64:71],instruction[56:63],instruction[48:55],instruction[40:47],instruction[32:39],instruction[24:31],instruction[16:23]};
   valP= pc +10;
   f_pc= valP;
   end

   else if( icode== 4'b1000 || icode== 4'b0111)
   begin
   valC= {instruction[64:71],instruction[56:63],instruction[48:55],instruction[40:47],instruction[32:39],instruction[24:31],instruction[16:23],instruction[8:15]};
   valP= pc+9;
   f_pc= valC;
   end

   else
   begin
   ins_address=1;
   stat= 2'd3;
   end

  end
```

```
  else
  begin
  ins_address=1;
  stat= 2'd3;
  end

nd



f(F_stall)
egin
 f_pc = pc;
nd


  always @(posedge(clk))
  begin
    if(D_stall)
    begin
    end
    else if(D_bubble)
    begin
      D_icode<= 4'b0001;
      D_ifun<= 4'b0000;
      D_rA<= 4'b1111;
      D_rB<= 4'b1111;
      D_valC<= 4'b0000;
      D_valP<= 4'b0000;
      D_stat<= 2'b00;
    end
    else
    begin
      D_icode<= icode;
      D_ifun<= ifun;
      D_rA<= rA;
      D_rB<= rB;
      D_valC<= valC;
      D_valP<= valP;
      D_stat<= stat;
    end
  end
module
```

This part of the processor decides which value should the newPc take:-

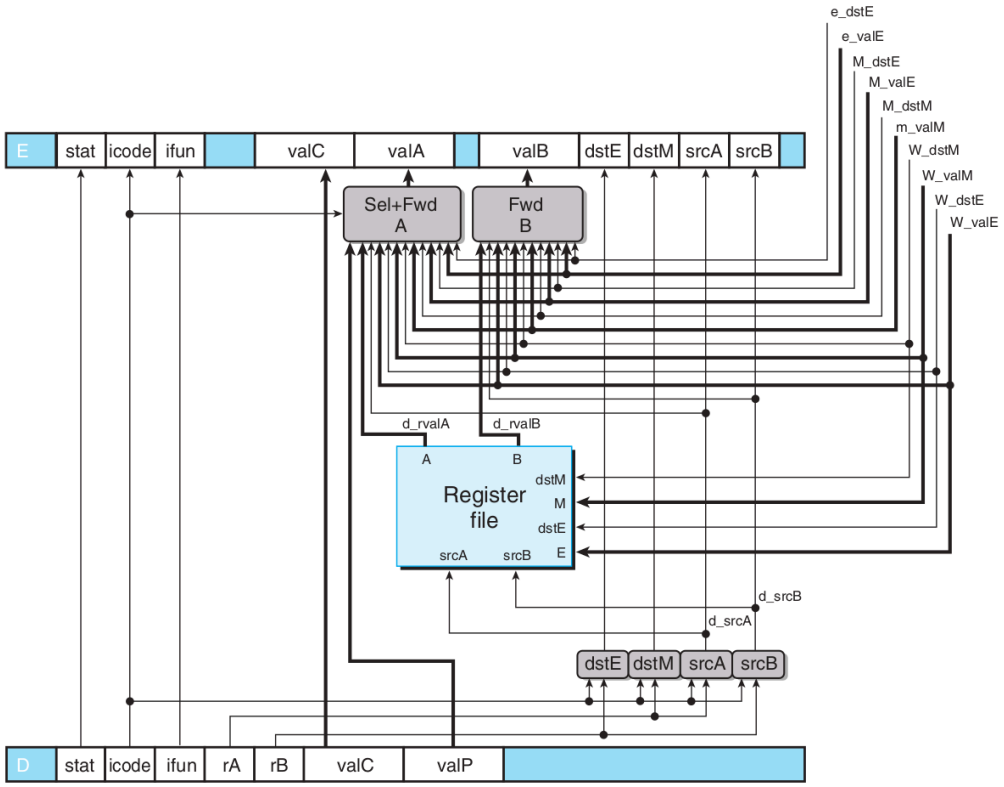
```
always @(*)
begin
if(M_icode== 4'd7 && M_cnd==0)
    begin
        newPc= M_valA;
    end
    else if(W_icode== 4'd9)
    begin
        newPc= W_valM;
    end
    else
    begin
        newPc= F_predPC;
    end
end
endmodule
```

This part of the code takes the predicted value from the fetch stage and feeds as the new pc value:-

```
module freg(input clk, input [63:0] f_pc, output reg [63:0] F_predPC);
always @(posedge clk)
begin
    F_predPC <= f_pc;
end
endmodule
```

Decode

The implementation of this stage is like that of the decode stage in the sequential processor. Most of the complexity of this stage is associated with the forwarding logic.
Although decode and writeback stages can be combined as shown in the diagram but we have implemented them individually:-



Hardware Implementation of Decode Stage

Implementation:

The forwarding logic is as follows:
-> d_valA = e_valE if d_srcA = e_dstE (Forward valE from execute)
-> d_valA = m_valM if d_srcA = M_dstM (Forward valM from memory)
-> d_valA = M_valE if d_srcA = M_dstE (Forward valE from memory)
-> d_valA = W_valE if d_srcA = W_dstE (Forward valM from write back)
-> d_valA = W_valE if d_srcA = W_dstE (Forward valE from write back)

```
module decodemodule(input wire clk, input [3:0] D_icode, input[3:0] D_ifun, input[3:0] D_rA, input[3:0] D_rB,
input[63:0] D_valC, input[63:0] D_valP, input[1:0] D_stat, input[3:0] e_dstE, input[3:0] M_dstE, input[3:0] M_dstM,
input[3:0] W_dstM, input[3:0] W_dstE, input[63:0] e_valE, input[63:0] M_valE, input[63:0] m_valM, input[63:0] W_valM,
input[63:0] W_valE,
input [63:0] rax,
input [63:0] rcx,
input [63:0] rdx,
input [63:0] rbx,
input [63:0] rsp,
input [63:0] rbp,
input [63:0] rsi,
input [63:0] rdi,
input [63:0] r8,
input [63:0] r9,
input [63:0] r10,
input [63:0] r11,
input [63:0] r12,
input [63:0] r13,
input [63:0] r14,
input E_bubble,
output reg[3:0] E_icode, output reg[3:0] E_ifun, output reg[63:0] E_valA, output reg[63:0] E_valB, output reg[63:0] E_valC,
output reg[3:0] E_dstE, output reg[3:0] E_dstM, output reg[3:0] E_srcA, output reg[3:0] E_srcB, output reg[1:0] E_stat,
output reg[63:0] d_valA, output reg[63:0] d_valB, output reg[3:0] d_srcA, output reg[3:0] d_srcB);

reg[3:0] dstE, dstM;
reg[63:0] register_file [0:14];


always @(*) begin

    register_file[0]= rax;
    register_file[1]= rcx;
    register_file[2]= rdx;
    register_file[3]= rbx;
    register_file[4]= rsp;
    register_file[5]= rbp;
    register_file[6]= rsi;
    register_file[7]= rdi;
    register_file[8]= r8;
    register_file[9]= r9;
    register_file[10]= r10;
    register_file[11]= r11;
    register_file[12]= r12;
    register_file[13]= r13;
    register_file[14]= r14;
```

```
    if(D_icode== 4'b0010)
    begin
      d_srcA= D_rA;
      d_srcB= 4'b1111;
      dstE= D_rB;
      dstM= 4'b1111;
      d_valA= register_file[d_srcA];
    end

    else if(D_icode== 4'b0011)
    begin
      dstE= D_rB;
      dstM= 4'b1111;
      d_srcA= 4'b1111;
      d_srcB= 4'b1111;
    end

    else if(D_icode== 4'b0100)
    begin
      d_srcA= D_rA;
      d_srcB= 4'b1111;
      dstE= D_rB;
      dstM= 4'b1111;

      d_valA= register_file[d_srcA];
      d_valB= register_file[dstE];
    end


    else if(D_icode== 4'b0101)
    begin
      d_srcA= 4'b1111;
      d_srcB= D_rB;
      dstM= D_rA;
      dstE= 4'b1111;

      d_valB= register_file[d_srcB];
    end

    else if(D_icode== 4'b0110)
    begin
      d_srcA= D_rA;
      d_srcB= D_rB;
      dstE= D_rB;
      dstM= 4'b1111;

      d_valA= register_file[d_srcA];
      d_valB= register_file[d_srcB];
    end
```

```
    else if(D_icode== 4'b1000)
    begin
      d_srcA= 4'b1111;
      d_srcB= 4'b0100;
      dstE= 4'b0100;
      dstM= 4'b1111;

      d_valB= register_file[d_srcB];
    end

    else if(D_icode== 4'b1001)
    begin
      d_srcA= 4'b0100;
      d_srcB= 4'b0100;
      dstE= 4'b0100;
      dstM= 4'b1111;

      d_valA= register_file[d_srcA];
      d_valB= register_file[d_srcB];
    end


    else if(D_icode== 4'b1010)
    begin
      d_srcA= D_rA;
      d_srcB= 4'b0100;
      dstE= 4'b0100;
      dstM= 4'b1111;

      d_valA= register_file[d_srcA];
      d_valB= register_file[d_srcB];

    end


    else if(D_icode== 4'b1011)
    begin
      d_srcA= 4'b0100;
      d_srcB= 4'b0100;
      dstE= 4'b0100;
      dstM= D_rA;

      d_valA= register_file[d_srcA];
      d_valB= register_file[d_srcB];
    end

    else
    begin

      d_srcA= 4'b1111;
      d_srcB= 4'b1111;
      dstE= 4'b1111;
      dstM= 4'b1111;

    end
  end
```

Data Forwarding:-

```
always @(*)
begin
  if(D_icode == 4'b1000 || D_icode == 4'b0111)
  begin
    d_valA= D_valP;
  end
  else if(d_srcA!=4'b1111)
  begin
    if(d_srcA== e_dstE)
    begin
      d_valA= e_valE;
    end
    else if(d_srcA== M_dstM)
    begin
      d_valA= m_valM;
    end
    else if(d_srcA== M_dstE)
    begin
      d_valA= M_valE;
    end
    else if(d_srcA== W_dstM)
    begin
      d_valA= W_valM;
    end
    else if(d_srcA== W_dstE)
    begin
      d_valA= W_valE;
    end
  end
end

always @(*)
begin
  if(d_srcB!=4'b1111)
  begin
    if(d_srcB== e_dstE)
    begin
      d_valB= e_valE;
    end
    else if(d_srcB== M_dstM)
    begin
      d_valB= m_valM;
    end
    else if(d_srcB== M_dstE)
    begin
      d_valB= M_valE;
    end
    else if(d_srcB== W_dstM)
    begin
      d_valB= W_valM;
    end
    else if(d_srcB== W_dstE)
    begin
      d_valB= W_valE;
    end
  end
end
```

Using the pipeline registers:-

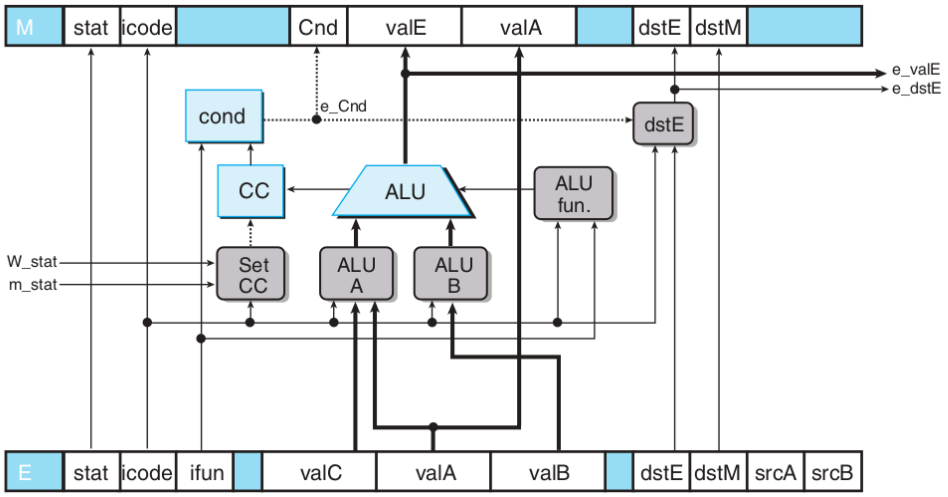
```
always @(posedge(clk))
begin

    if(E_bubble==0)
    begin
      E_icode <= D_icode;
      E_ifun <= D_ifun;
      E_valA <= d_valA;
      E_valB <= d_valB;
      E_valC <= D_valC;
      E_srcA <= d_srcA;
      E_srcB <= d_srcB;
      E_dstE <= dstE;
      E_dstM <= dstM;
      E_stat <= D_stat;
    end
    else
    begin
      E_icode <= 4'b0001;
      E_ifun <= 4'b0000;
      E_valA <= 4'b0000;
      E_valB <= 4'b0000;
      E_valC <= 4'b0000;
      E_srcA <= 4'b1111;
      E_srcB <= 4'b1111;
      E_dstE <= 4'b1111;
      E_dstM <= 4'b1111;
      E_stat <= 2'd0;
    end
end
endmodule
```

Execute

The implementation of this stage is almost the same as that of the execute stage in SEQ.

Implementation:-

1. The values of e_valE and e_cnd is set in the similar way as done in the sequential implementation of processor.
2. The values from the execute register and the values calculated in this stage are stored in the memory register.



Hardware implementation of Execute Stage

```
module execute(input wire clk,input [3:0]E_icode, input [3:0]E_ifun , input [63:0] E_valA , input [63:0] E_valB , input [63:0] E_valC ,
input [3:0] E_dstE , input [3:0] E_dstM ,input [1:0] E_stat ,input set_cc, output reg [1:0] M_stat , output reg [3:0] M_icode, output reg [3:0] M_ifun,
output reg M_cnd , output reg [63:0] M_valA , output reg [63:0] M_valE , output reg [3:0] M_dstE , output reg [3:0] M_dstM ,
output reg [63:0] e_valE , output reg [3:0] e_dstE , output reg e_cnd);



always @(*)
begin
    if (icode==4'b0110 && clk ==1)
    begin
        zf = (res==1'b0)&&(ifun==4'b0000||ifun==4'b0001);
        sf = (res[63]==1);
        if (ifun == 4'b0000)
        begin
            of = (a[63]==b[63])&&(res[63]!=a[63]);
        end
        else if (ifun == 4'b0001)
        begin
            of = (a[63]!=b[63])&&(res[63] != a[63]);
        end
    end
end
reg of;
reg zf;
reg sf;
reg [63:0] ValA;
reg [63:0] ValB;
reg [63:0] ValC;
reg [3:0] icode;
reg [3:0] ifun;
reg signed [63:0] ValE;
reg cnd;
wire [63:0] res;
wire overfl ;
wire [63:0] anded;
wire [63:0] movand;
wire [63:0] incand;
wire [63:0] decand;
wire [63:0] xored;
wire [63:0] movxor;
wire [63:0] incxor;
wire [63:0] decxor;
reg signed [63:0] result;
reg signed [63:0] a;
reg signed [63:0] b;
reg S0;
reg S1;
wire signed [63:0] vinc;
wire signed [63:0] vdec;
wire signed [63:0] vmov;
alu alu1(a,b,S0,S1,res,overfl,anded,xored);
alu alu2(ValB,ValC,1'b0,1'b0,vmov,overfl,movand,movxor);
alu alu3(ValB,64'd8,1'b0,1'b0,vinc,overfl,incand,incxor);
alu alu4(ValB,64'd8,1'b1,1'b0,vdec,overfl,decand,decxor);
initial
```

```
initial
begin
    S0=0;
    S1=0;
    a = 64'b0;
    b = 64'b0;
end

always @(*)
begin
    ValA = E_valA;
    ValB = E_valB;
    ValC = E_valC;
    icode = E_icode;
    ifun = E_ifun;
    e_dstE = E_dstE;
    e_valE = ValE;
    e_cnd = cnd;
end


always @(*)
begin
    if(clk==1)
    begin
        cnd = 0;
        if (icode==4'b0010 || icode==4'b0111)//cmov //jump
        begin
            if(ifun==4'b0000)//uncon
            begin
                cnd = 1;
            end
            else if(ifun==4'b0001)//le
            begin
                cnd = (sf^of)|(zf);
            end
            else if (ifun==4'b0010)//l
            begin
                cnd = (sf^of);
            end
            else if (ifun==4'b0011)//e
            begin
                cnd = zf;
            end
            else if (ifun==4'b0100)//ne
            begin
                cnd = ~zf;
            end
            else if (ifun==4'b0101)//ge
            begin
                cnd = ~(sf^of);
            end
            else if (ifun==4'b0110)//g
            begin
                cnd = ~(sf^of)&(~zf);
            end
            ValE = 64'd0 + ValA;
        end
```

```
    else if (icode==4'b0011) //irmov
    begin
        ValE = 64'd0 + ValC;
    end
    else if (icode==4'b0100 || icode == 4'b0101) //rmmov //mrmov
    begin
        ValE = vmov;
    end
    else if (icode==4'b0110)//OP
    begin
        if(ifun==4'b0000)//add
        begin
            S0=0;
            S1=0;
            a = ValA;
            b = ValB;
            assign result = res;
        end
        else if (ifun==4'b0001)//sub
        begin
            S0=1;
            S1=0;
            a = ValA;
            b = ValB;
            assign result = res;
        end
        else if (ifun==4'b0010)//and
        begin
            S0=0;
            S1=1;
            a = ValA;
            b = ValB;
            assign result = anded;
        end
        else if (ifun==4'b0011)//xor
        begin
            S0=1;
            S1=1;
            a = ValA;
            b = ValB;
            assign result = xored;
        end
        ValE = result;
    end
    else if (icode==4'b1000 || icode==4'b1010)//call //push
    begin
        ValE = vdec;
    end
    else if (icode==4'b1001 || icode==4'b1011)//ret //pop
    begin
        ValE = vinc;
    end
    else
    begin
        ValE = 64'd0;//default case
    end
end
```

```
end

always@(posedge clk)
begin
    M_stat <= E_stat;
    M_icode <= E_icode;
    M_cnd <= e_cnd;
    M_valE <= e_valE;
    M_valA <= E_valA;
    M_dstE <= e_dstE;
    M_dstM <= E_dstM;
end

endmodule
```

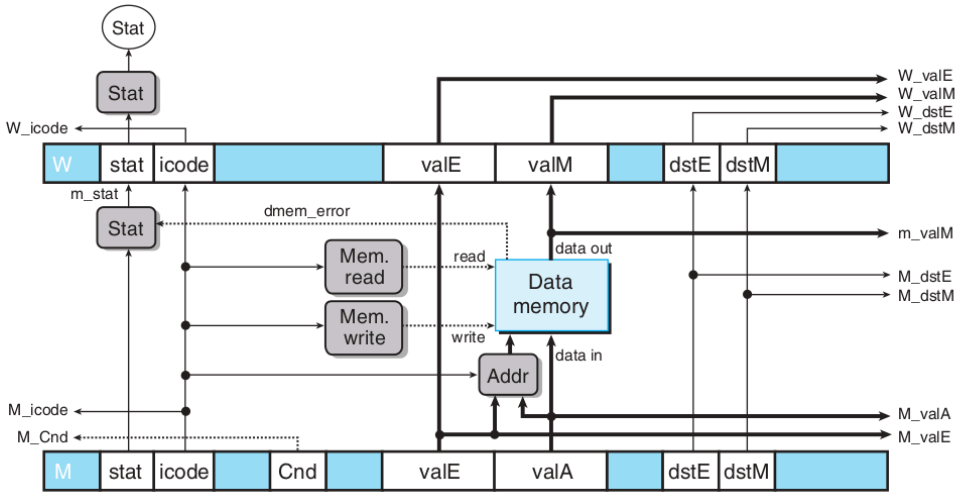
Memory

The implementation of the memory stage remains almost the same as that in sequential processor.

Implementation:

->We read from and write to the data memory in same way as we did for the sequential processor.

▪ We store the values from the memory register and the output values from memory stage in the write back register .



Hardware Implementation of Memory stage

```
module memory(input clk , input [1:0] M_stat , input [3:0] M_icode, input [3:0] M_ifun , input M_cnd , input [3:0] M_dstE,
input [3:0] M_dstM, input [63:0] M_valA , input [63:0] M_valE , output reg [1:0] W_stat , output reg [3:0] W_icode,
output reg [3:0] W_ifun , output reg [63:0] W_valE , output reg [63:0] W_valM , output reg [3:0] W_dstE , output reg [3:0] W_dstM
,output reg [1:0] m_stat , output reg [63:0] m_valM, output reg [63:0] data , output reg W_cnd);


reg [3:0] icode;
reg [3:0] ifun;
reg [63:0] ValA;
reg [63:0] ValM;
reg [63:0] ValE;
reg adr_memory;
always @(*)
begin
    icode = M_icode;
    ifun = M_ifun;
    ValA = M_valA;
    m_valM = ValM;
    ValE = M_valE;
    m_stat = M_stat;
end


reg[63:0] mem[0:8192];

always @(*)
begin
    if(icode==4'b1001)
    begin
      ValM=mem[ValA];
    end
        if (icode==4'b1010)
    begin
        mem[ValE] = ValA;
    end
        if (icode==4'b1000)
    begin
        mem[ValE] = ValA;
    end
        if (icode==4'b0100)
    begin
        mem[ValE] = ValA;
    end
end
```

```
always @(*)
begin
    adr_memory=0;
    if(ValE>8192)
    begin
      adr_memory=1;
      m_stat = 2'd2;
    end
    else
    begin
    // if (icode==4'b0100)
    // begin
    //     mem[ValE] = ValA;
    // end
    if (icode==4'b0101)
    begin
        ValM = mem[ValE];
    end
    // if (icode==4'b1000)
    // begin
    //     mem[ValE] = ValP;
    // end
    // if (icode==4'b1001)
    // begin
    //     ValM = mem[ValA];
    // end
    if (icode==4'b1011)
    begin
        ValM = mem[ValA];
    end
    // if (icode==4'b1010)
    // begin
    //     mem[ValE] = ValA;
    // end
    //     if(icode==4'b1001)
    // begin
    //   ValM=mem[ValA];
    // end
    data = mem[ValE];
end
end

always @(posedge clk)
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;
    W_cnd =M_cnd;
end


endmodule
```

Writeback:-

The implementation of this stage is same as that of the write back stage in SEQ. The register file has two write ports. Port E is used to write values computed by the ALU, while port M is used to write values read from the data memory that is the values valE, or valM are written into the registers with addresses rA, rB or RRSP depending on the icode.

```
module writeback(clk , W_icode ,W_ifun , W_cnd ,W_dstM , W_dstE , W_valE , W_valM,m_stat,rax,rcx,rdx,rbx,rsp,rbp,rsi,rdi,r8,r9,r10,r11,r12,r13,
r14,W_stat,reg0,reg1,reg2,reg3,reg4,reg5,reg6,reg7,reg8,reg9,reg10,reg11,reg12,reg13,reg14);


input clk;
    input [3:0] W_icode ;
    input [3:0] W_ifun;
    input W_cnd;
    input [3:0] W_dstM;
    input [3:0] W_dstE;
    input [63:0] W_valE;
    input [63:0] W_valM;
    input [1:0] m_stat;
    input [63:0] rax;
    input [63:0] rcx;
    input [63:0] rdx;
    input [63:0] rbx;
    input [63:0] rsp;
    input [63:0] rbp;
    input [63:0] rsi;
    input [63:0] rdi;
    input [63:0] r8;
    input [63:0] r9;
    input [63:0] r10;
    input [63:0] r11;
    input [63:0] r12;
    input [63:0] r13;
    input [63:0] r14;

    output reg [1:0] W_stat;
    output [63:0] reg0;
    output [63:0] reg1;
    output [63:0] reg2;
    output [63:0] reg3;
    output [63:0] reg4;
    output [63:0] reg5;
    output [63:0] reg6;
    output [63:0] reg7;
    output [63:0] reg8;
    output [63:0] reg9;
    output [63:0] reg10;
    output [63:0] reg11;
    output [63:0] reg12;
    output [63:0] reg13;
    output [63:0] reg14;
reg [63:0] register_file[0:15];

reg [63:0] ValE;
reg [63:0] ValM;
reg [3:0] rA;
reg [3:0] rB;
reg [3:0] icode;
reg cnd;
```

```
always @(*)
begin
    ValE = W_valE;
    ValM = W_valM;
    rA = W_dstM;
    rB = W_dstE;
    icode =W_icode;
    cnd = W_cnd;
end
    always @(posedge clk)
    begin
    W_stat=m_stat;
    end
    always @(*)
    begin
        register_file[0] = rax;
        register_file[1] = rcx;
        register_file[2] = rdx;
        register_file[3] = rbx;
        register_file[4] = rsp;
        register_file[5] = rbp;
        register_file[6] = rsi;
        register_file[7] = rdi;
        register_file[8] = r8;
        register_file[9] = r9;
        register_file[10] = r10;
        register_file[11] = r11;
        register_file[12] = r12;
        register_file[13] = r13;
        register_file[14] = r14;
    end
```

```
    always @(*)
    begin
        if (icode==4'b0010)//cmov
        begin
            if (cnd==1'b1)
            begin
                register_file[rB] = ValE;
            end
        end
        if (icode==4'b0011)//irmov
        begin
            register_file[rB] = ValE;
        end
        if (icode==4'b0101)//mrmov
        begin
            register_file[rA] = ValM;
        end
        if (icode==4'b0110)//op
        begin
            register_file[rB] = ValE;
        end
        if (icode==4'b1000)//call
        begin
            register_file[4] = ValE;
        end
        if (icode==4'b1001)//return
        begin
            register_file[4] = ValE;
        end
        if (icode==4'b1010)//push
        begin
            register_file[4] = ValE;
        end
        if (icode==4'b1011)//pop
        begin
            register_file[4] = ValE;
            register_file[rA] = ValM;
        end
    end
    assign reg0 = register_file[0];
    assign reg1 = register_file[1];
    assign reg2 = register_file[2];
    assign reg3 = register_file[3];
    assign reg4 = register_file[4];
    assign reg5 = register_file[5];
    assign reg6 = register_file[6];
    assign reg7 = register_file[7];
    assign reg8 = register_file[8];
    assign reg9 = register_file[9];
    assign reg10 = register_file[10];
    assign reg11 = register_file[11];
    assign reg12 = register_file[12];
    assign reg13 = register_file[13];
    assign reg14 = register_file[14];

endmodule
```

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 and
(2) control hazards.

Method-1) by Stalling

One very general technique for avoiding hazards involves stalling, where the processor holds back one or more instructions in the pipeline until the hazard condition no longer holds.Stalling involves holding back one group of instructions in their stages while allowing other instructions to continue flowing through the pipeline. The stages that should normally be processing would be injected with a bubble. A bubble is like a dynamically generated nop instruction—it does not cause any changes to the registers, the memory, the condition codes, or the program status.

Method-2) by Forwarding

The technique of passing a result value directly from one pipeline stage to an earlier one is commonly known as data forwarding. Data forwarding requires adding additional data connections and control logic to the basic hardware structure.
-> It can also use the ALU output (signal e_valE) for operand valA or valB.
-> It can use the value that has just been read from the data memory (signal m_valM) for operand valA or valB.
-> It can use the value in the memory stage (signal M_valE) for operand valA or valB.
-> It can use the value in the write-back stage (signal W_valE or signal W_valM) for operand valA or valB.

Method-3) Load/Use Data Hazards:

One class of data hazards cannot be handled purely by forwarding, because memory reads occur late in the pipeline. These are called load/use hazards and they occur when one instruction reads a value from memory for register while the next instruction needs this value as a source operand. We can avoid a load/use data hazard with a combination of stalling and forwarding. This requires modifications of the control logic.

Avoiding Control Hazards:

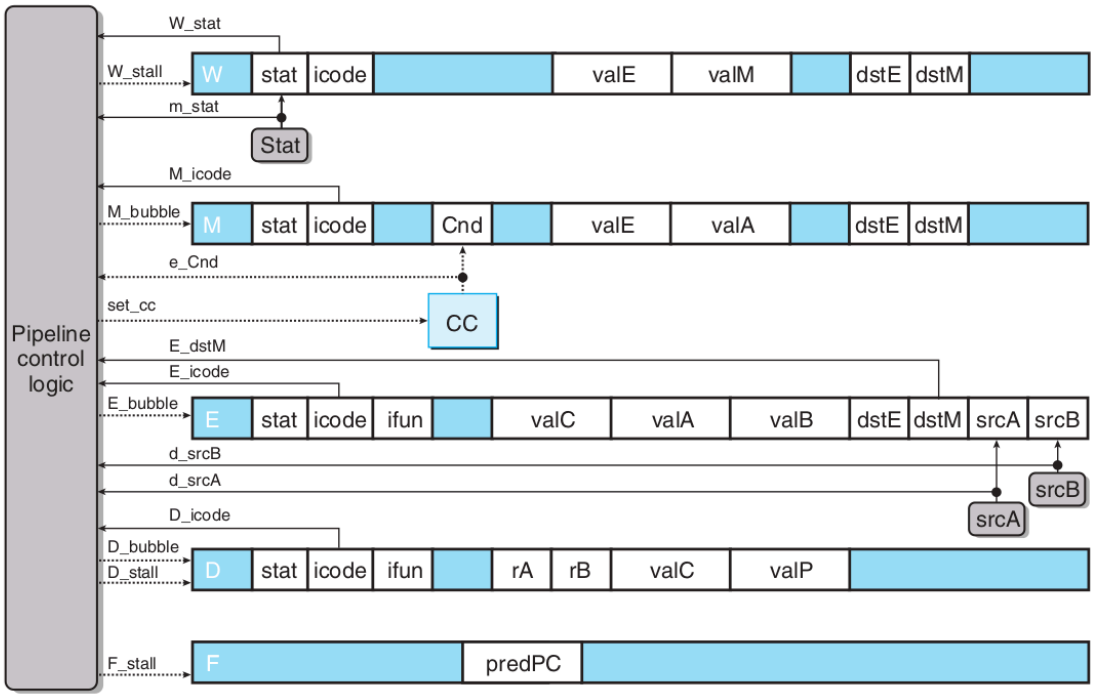
Control hazards arise when the processor cannot reliably determine the address of the next instruction based on the current instruction in the fetch stage. control hazards can only occur in our pipelined processor for ret and jump instructions. In case of ret, the processor is stalled for 3 clock cycles after the ret instruction. While in case of jump misprediction, the pipeline can simply cancel the two misfetched instructions by injecting bubbles into the decode and execute stages on the following cycle while also fetching the instruction following the jump instruction.

Pipeline Control Logic:-

In reference to the above mentioned hazards,This block is used for handling data and control hazards.

In cases of return, mispredicted branches and load/use hazards, we can observe misbehaviour and incorrect results. To handle such cases, we might need to stall some stage and insert a bubble in some.

-> The return instruction can be detected using the condition given below,
IRET in { D_icode, E_icode, M_icode}
-> Branch misprediction can be detected using the condition given below,
E_icode = IJXX & !e_cnd
-> Load/use hazard can be detected using the condition given below,
E_icode in { MRMOVQ, IPOPQ } && E_dstM in {d_srcA, d_srcB}



Hardware Implementation of Pipeline Control logic

```
module pipe_logic(input [3:0]D_icode, input[3:0]d_srcA, input[3:0]d_srcB, input[3:0]E_icode,
input[3:0]E_dstM, input[3:0]M_icode, input[1:0]m_stat, input[1:0]W_stat, input e_cnd,
output reg W_stall, output reg M_bubble, output reg set_cc, output reg E_bubble,
output reg D_bubble, output reg D_stall, output reg F_stall);

always @(*)
begin
  set_cc=1'b1;
  W_stall= 1'b0;
  M_bubble= 1'b0;
  E_bubble= 1'b0;
  D_bubble= 1'b0;
  D_stall= 1'b0;
  F_stall= 1'b0;

  F_stall= (((E_icode==4'b0101 || E_icode== 4'b1011)&&(E_dstM== d_srcA || E_dstM== d_srcB))|||(D_icode==4'b1001||E_icode==4'b1001||M_icode==4'b1001));
  D_bubble= ((E_icode==4'b0111 && !e_cnd)|||((!((E_icode==4'b0101 || E_icode== 4'b1011)&&(E_dstM== d_srcA || E_dstM== d_srcB)))&&(D_icode==4'b1001||E_icode==4'b1001||M_icode==4'b1001)));
  D_stall= ((E_icode==4'b0101 || E_icode== 4'b1011)&&(E_dstM== d_srcA || E_dstM== d_srcB));
  E_bubble= ((E_icode==4'b0111 && !e_cnd)|||((E_icode==4'b0101 || E_icode== 4'b1011)&&(E_dstM== d_srcA || E_dstM== d_srcB)));
  set_cc= ((E_icode==4'b0110)&&(m_stat==2'b00)&&(W_stat==2'b00));
  M_bubble= (!(m_stat==2'b00))|||(!(W_stat==2'b00));
  W_stall= !(W_stat==2'b00);
end
endmodule
```

Pipeline Processor:-

Here is the compiled version of 5-stage pipeline.

```
`include "fetch.v"
`include "f.v"
`include "selectpc.v"
`include "decode.v"
`include "execute.v"
`include "memory.v"
`include "writeback.v"
`include "pipe_ctrl.v"
module final;

reg clk;
reg [63:0] pc;

wire F_stall;
wire [63:0] F_predPC ,f_pc , newPc;

wire D_stall,D_bubble;
wire [1:0] D_stat;
wire [3:0] D_icode,D_ifun,D_rA,D_rB,d_srcA,d_srcB;
wire [63:0] D_valC,D_valP,d_valA,d_valB;

reg [63:0] rax, rcx, rdx, rbx, rsp, rbp ,rsi, rdi ,r8 ,r9 ,r10 ,r11 ,r12 ,r13 ,r14;


wire E_bubble,set_cc ,e_cnd;
wire [1:0] E_stat;
wire [3:0] E_icode, E_ifun , E_srcA, E_srcB, E_dstE ,E_dstM ,e_dstE;
wire [63:0] E_valA , E_valB ,E_valC, e_valE;

wire M_cnd;
wire [1:0] M_stat, m_stat;
wire [3:0] M_icode, M_ifun ,M_dstE ,M_dstM;
wire [63:0] M_valA, M_valE, m_valM ,data;

wire W_cnd;
wire [1:0] W_stat;
wire [3:0] W_icode , W_ifun , W_dstE, W_dstM;
wire [63:0] W_valE,W_valM;
wire [63:0] reg0,reg1,reg2,reg3,reg4,reg5,reg6,reg7,reg8,reg9,reg10,reg11,reg12,reg13,reg14;
```

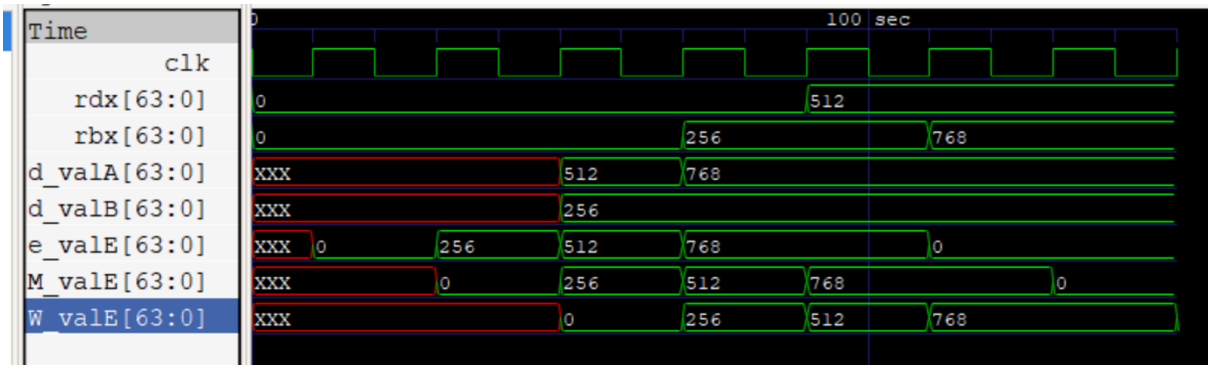
```
freg f(clk,f_pc,F_predPC);

selectpc select(clk,F_predPC,M_cnd ,M_valA,W_valM,M_icode,W_icode,newPc);

fetchmodule fetch(clk,pc,M_cnd,M_valA,W_valM,F_stall,D_stall,D_bubble,M_icode,W_icode,f_pc,D_icode,D_ifun,D_rA,D_rB,D_valC,D_valP,D_stat);

decodemodule decode(clk,D_icode,D_ifun,D_rA,D_rB,D_valC,D_valP,D_stat,e_dstE,M_dstE,M_dstM,W_dstM,W_dstE,e_valE,M_valE,m_valM,W_valM,
W_valE,rax,rcx,rdx,rbx,rsp,rbp,rsi,rdi,r8,r9,r10,r11,r12,r13,r14,E_bubble,E_icode,E_ifun,E_valA,E_valB,E_valC,E_dstE,E_dstM,E_srcA,E_srcB,
E_stat,d_valA,d_valB,d_srcA,d_srcB);

execute exec(clk,E_icode,E_ifun,E_valA,E_valB ,E_valC ,E_dstE ,E_dstM ,E_stat ,set_cc,M_stat ,M_icode,M_ifun,
M_cnd ,M_valA ,M_valE ,M_dstE ,M_dstM , e_valE ,e_dstE ,e_cnd);

memory mem(clk ,M_stat ,M_icode,M_ifun ,M_cnd ,M_dstE,M_dstM,M_valA ,M_valE ,W_stat ,W_icode,W_ifun ,W_valE ,W_valM ,W_dstE ,W_dstM
,m_stat ,m_valM,data ,W_cnd);

writeback wb(clk , W_icode ,W_ifun , W_cnd ,W_dstM , W_dstE , W_valE , W_valM,m_stat,rax,rcx,rdx,rbx,rsp,rbp,rsi,rdi,r8,r9,r10,r11,r12,r13,
r14,W_stat,reg0,reg1,reg2,reg3,reg4,reg5,reg6,reg7,reg8,reg9,reg10,reg11,reg12,reg13,reg14);

pipe_logic pl(D_icode,d_srcA,d_srcB,E_icode, E_dstM,M_icode,m_stat,W_stat,e_cnd,
W_stall,M_bubble,set_cc,E_bubble, D_bubble,D_stall,F_stall);
```

```
always #10 clk=~clk;
initial
begin
rax=0;
rcx=0;
rdx=0;
rbx=0;
rsp=8192;
rbp=0;
rsi=0;
rdi=0;
r8=0;
r9=0;
r10=0;
r11=0;
r12=0;
r13=0;
r14=0;
end

always @(*)
begin
   pc = newPc;
end
initial
begin
$dumpfile("final.vcd");
$dumpvars(0,final);
pc = 0;
clk=0;
end

   always @(W_stat)
   begin
     if(W_stat==2'd1)
     begin
       $finish;
     end
   end
```

```
always @(*)
begin

rax=reg0;
rcx=reg1;
rdx=reg2;
rbx=reg3;
rsp=reg4;
rbp=reg5;
rsi=reg6;
rdi=reg7;
r8=reg8;
r9=reg9;
r10=reg10;
r11=reg11;
r12=reg12;
r13=reg13;
r14=reg14;
end

always @*
   begin

//     $monitor("clk=%d\n icode=%difun=%d\nrA=%d rB=%d\n valC=%dvalP=%d\n ins_address=%d adr_address=%d hlt=%d\n valA=%d valB=%d \n valE=%d \n cnd=%d zf=%d sf=%
//    end
   $monitor("time = %d , e_cnd = %d ,d_valA = %d, d_valB = %d , e_valE = %d , m_valM = %d , W_valM = %d ,W_valE =%d ,M_valE= %d, pc = %d reg0=%d reg1=%d reg2=%d
   reg3=%d reg4=%d reg5=%d reg6=%d reg7=%d reg8=%d reg9=%d reg10=%d reg11=%d reg12=%d reg13=%d reg14=%d \n\n",$time,
   e_cnd,d_valA,d_valB,e_valE,m_valM,W_dstM,W_valE,M_valE,pc,rax,rcx,rdx,rbx,rsp,rbp,rsi,rdi,r8,r9,r10,r11,r12,r13,r14);
   end
endmodule
```

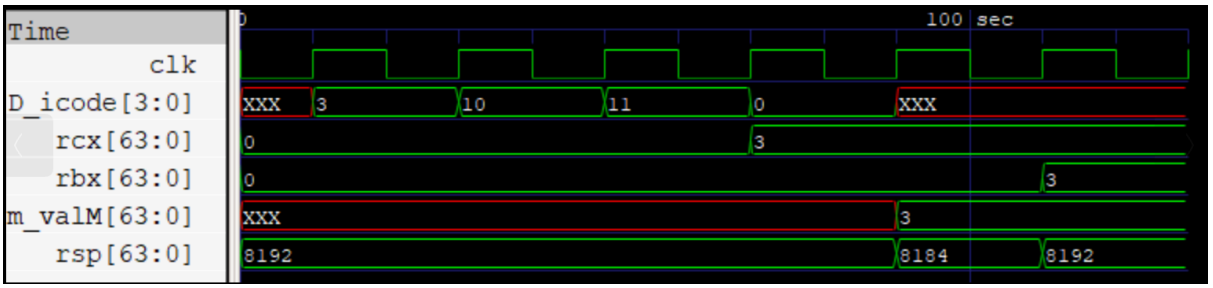
Outputs:-

1)Sample(Data Forwarding):-

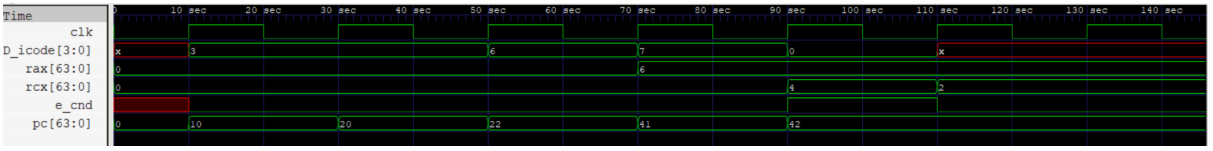


2)RMMR

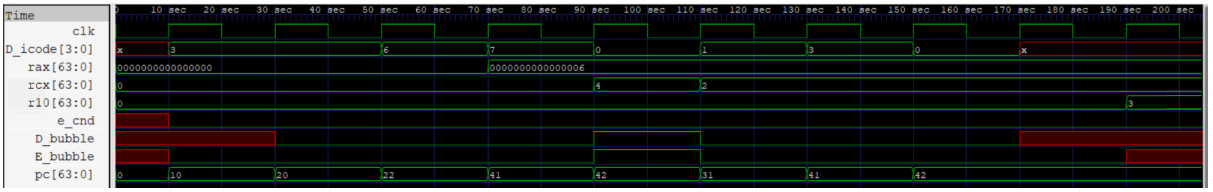
3)PushPop



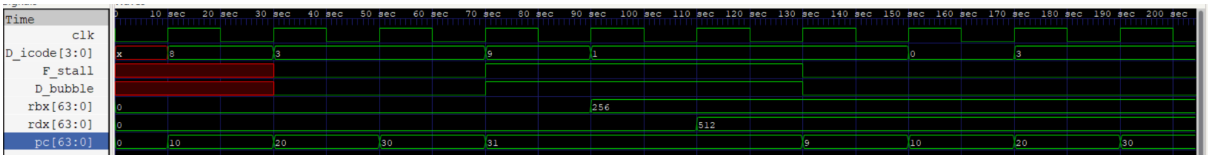
4)Predicted Jump



5)Mispredicted Jump



6)Call Return Jump



7)Load Use Hazards

Challenges faced:-

SEQUENTIAL:-

- Understanding of how different stages on different edges of the clock cycle was difficult.

PIPELINE:-

- We faced difficulty in implementing data forwarding,stalls and bubbles.
- It also took time while transitioning from sequential to pipeline implementation.

BOTH:-

- Managing so many inputs and output ports was difficult.

Overall the project helped us gain a better understanding of the processor.