PROJECT REPORT

BANALA VISHWA TEJA REDDY - 2020102058

CHALLA THARUN. - 2020102047

CONTENTS

2.5-**PC UPDATE**

3. **5-STAGE PIPELINING**

1.GOAL OF THE PROJECT

- We need to prepare a Y86-64 Processor using verilog by implementing the sequential and alu and pipeline parts perfectly and allow all the types of the instruction structure architecture.
- A Y86-64 processor implemented using Verilog that is capable of running Y86-64 instructions.The project repository should contain both a sequential model as well as a 5 stage pipelined model.
- INSTRUCTIONS we includes are:

```
halt
nop
rrmovq
vmovle
cmovl
cmove
cmovne
cmovge
cmovg
irmovq
rmmovq
mrmovq
addq
subq
andq
xorq
jmp
jle
```

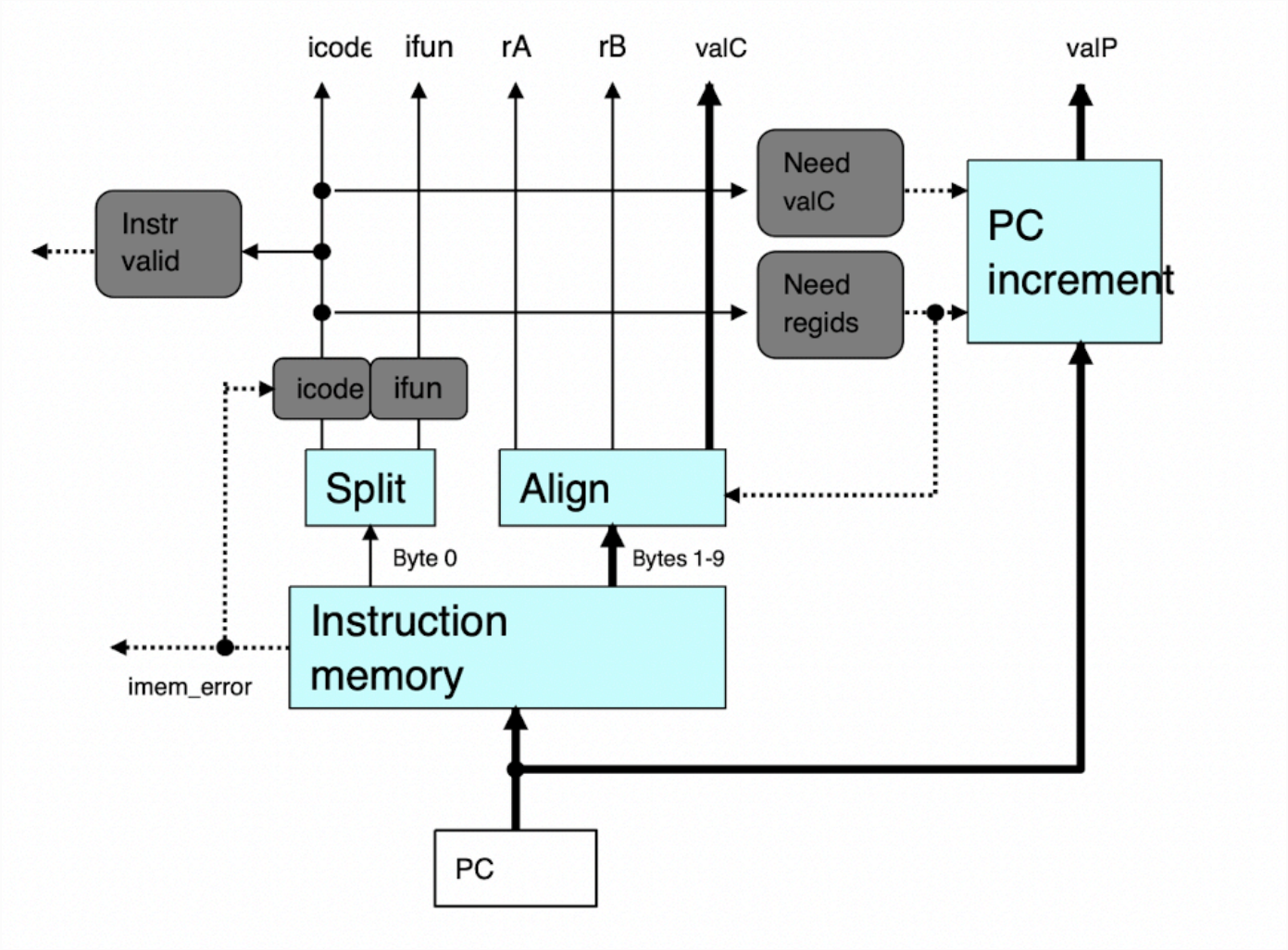
```
jl
je
jne
jge
jg
call
ret
pushq
popq
```

2.SEQUENTIAL MODULES

2.1 FETCH MODULE

2.1.1 BASIC OVERVIEW OF FETCH

- The main use of the fetch is same as suggested by the name it will fetch the data from the instruction memory stored by the instruction memory using PC value and continuously updating PC to the valP based on the conditions.



- Here in fetch we are taking inputs at the negative edge of the clock preferably and at clk=0 after modules will work and at clk=positive edge write back will rewrite the registers.
- Here in fetch we will take the inputs from instruction memory based on the icode.
- Icode will determine which instruction will happen for a certain inputs now we will take the instruction memory at PC byte and divide it into 2 -4 bites one containing MSB and other containing LSB Now we will call four MSB bits as Icode and 4-LSB bits as Ifun.
- Now depending on the Icode we may need to take data from instruction memory for various instructions sets.
- The inputs the fetch module are instruction memory and PC value from that we will deduce rA,rB,Icode,Ifun,valC,valP,instr_valid(validity),imem_error(error).

2.1.2 FETCH FOR DIFFERENT INSTRUCTIONS

- Here we discuss how fetch will take instructions from instruction memory we given to the fetch it make take upto PC or PC+1 or PC+9 it will be determined by the Icode mainly.
- rA and rB will show the adresses of the register memory we took in that 14 bytes of information we store in Y86-64 processor.
- Below we describe each and every instruction that fetch will take as input.

```
###### **2.1.2.1 FETCH FOR HALT **
```

- For halt instruction Fetch will first take information from PC it will encounter Icode as 0000 and Ifun as xxxx.
- As the fetch sees the Icode as 0000 then it will keep halt=1 and there will not be operation in execute module, decode module,memory module, so it wont allow next iteration to run which will halt the iterations.
- Finally we need to halt the iterations using halt function as stated above it will stop the iterations as no input of the execute,decode,memory blocks didnot changed as it uses always@(*)
- Now at fetch at halt will take Instruction_memory(PC) and divide icode and ifun and check the icode ==0000 and then will flag halt=1,and increase valP=PC+1.

  4'b0000 :
  begin // halt
  halt = 1'b1 ;
  valP = PC +1 ;
  end

2.1.2.2 FETCH FOR NOP

- For nop instruction Fetch will take information from Pc it will encounter Icode=0001 and ifun=xxxx.
- As the fetch sees the Icode as 0001 then it will just increase the value of valP as valP=PC+1 and there will be no operations will happen and it will directly skip for the next iteration as the PC is updated every time to PC+1.
- Now at fetch at nop instruction(icode=0001) will take instruction from PC and divide icode and ifun check the icode as 0001 and it will increase valP=PC+1.

```
4'b0001:                //nop
valP=PC+1;
```

2.1.2.3 FETCH FOR CMOVXX

- For cmovxx Fetch will first take Information from PC it will encounter icode as 00010 ifun as xxxx.
- As it seen icode as 0010 then fetch will take data from instruction memory(PC,PC+1) as it will take icode and ifun. rA and rB will take instruction memory(PC+1);
- Now at fetch it will define rA and rB and Icode and Ifun and it will increase the valP to PC+2 as it took instructions from PC and PC+1.

```
4'b0010:                //cmovxx
 begin
 rA = Instruction_memory (PC+1)(7:4) ;
 rB = Instruction_memory (PC+1)(3:0) ;
 valP = PC+64'd2;
 end
```

2.1.2.4 FETCH FOR IRMOVQ

- As the name suggests it will move immediately data from the valC to register as suggested by rA,rB.
- Now fetch will decode the instruction memory from PC to PC+9.
- PC gives ICode(0011) and IFun , PC+1 gives rA,rB, and (PC+2:PC+9) will give valC. and it took all 10 PC instructions so it will keep increment the PC by PC+10
- PC-Icode:Ifun,PC+1-rA,rB,PC+2:PC+9-valC and valP= PC+10

```
4'b0011:                          //irmovq
```

begin
rA = Instruction_memory (PC+1)(7:4) ;
rB = Instruction_memory (PC+1)(3:0) ;
valC = Instruction ;
valP = PC+64'd10;
end

2.1.2.5 FETCH FOR RMMOVQ

- As the name suggests it will move register data to memory as suggested by rA,rB.
- Now fetch will decode the instruction memory from PC to PC+9.
- PC gives ICode(0100) and IFun , PC+1 gives rA,rB, and (PC+2:PC+9) will give valC. and it took all 10 PC instructions so it will keep increment the PC by PC+10
- PC-Icode:Ifun,PC+1-rA,rB,PC+2:PC+9-valC and valP= PC+10

```
4'b0100:                          //rmmovq
```

begin
rA = Instruction_memory (PC+1)(7:4) ;
rB = Instruction_memory (PC+1)(3:0) ;
valC = Instruction ;
valP = PC+64'd10;
end

2.1.2.6 FETCH FOR MRMOVQ

- As the name suggests it will move memory data to register as suggested by rA,rB.
- Now fetch will decode the instruction memory from PC to PC+9.
- PC gives ICode(0100) and IFun , PC+1 gives rA,rB, and (PC+2:PC+9) will give valC. and it took all 10 PC instructions so it will keep increment the PC by PC+10
- PC-Icode:Ifun,PC+1-rA,rB,PC+2:PC+9-valC and valP= PC+10

```
4'b0100:                          //rmmovq
```

begin
rA = Instruction_memory (PC+1)(7:4) ;
rB = Instruction_memory (PC+1)(3:0) ;
valC = Instruction ;
valP = PC+64'd10;
end

2.1.2.7 FETCH FOR OPQ

- As the name suggests the instruction will take care of addition subtraction and and xor of the register adresses rA and rB based on the ifun
- For OPQ Icode must be 0110 and ifun will be matter in execution.
- Now it will take instruction memories of PC to PC+1 it will take icode and ifun from the PC and rA and rB from the PC+1 then increment the PC value by 2 so valP=PC+2

```
4'b0110:                        //OPq
begin
rA = Instruction_memory [PC+1][7:4] ;
rB = Instruction_memory [PC+1][3:0] ;
valP = PC+64'd2;
end
```

2.1.2.8 FETCH FOR JXX

- As the name suggests it will help to jump from one instruction memory to other one on the basis of the conditions.
- For JXX Icode must be 0111 and Ifun will not be matter in execution and PC update.
- Now it will take instruction memories of PC to PC+8 it will take Icode and Ifun from PC and from PC+1 to PC+8 it will take valC and it will increment PC by 9 so valP=PC+9.

```
4'b0111:                        //jxx
begin
valC={Instruction_memory[PC+1], Instruction[63:8]} ;
valP=PC+64'd9;
end
```

2.1.2.9 FETCH FOR CALL

- As the name suggests it will call a value from the register which is stack pointed in the 15 registers of the storage Y86-64.
- For Call icode should be 1000 and ifun as xxxx.
- Now fetch will take instructions from instrunction memory PC to PC+8 as PC gives Icode and Ifun and from PC to PC+8 will provide valC so we need to increment PC by 9 so valP=PC+9.

```
4'b1000:                    //call
begin
valC={Instruction_memory(PC+1), Instruction(63:8)} ;
valP=PC+64'd9;
end
```

2.1.2.10 FETCH FOR RETURN

- As the name suggests it will return the value which is stack pointed.
- it will take only one instruction PC as ICode(1001) and Ifun so it will increment PC by only once which implies valP=PC+1.

```
4'b1001:
begin                    //ret
valP = PC+64'd1;
end
```

2.1.2.11 FETCH FOR PUSHQ

- As the name suggests it wil push the data to lower value decrement is 8 normally so it should be stored on memory of initial memory - 8
- it will take PC and PC+1 from instruction memory and will keep PC as icode(1010):ifun and PC+1 as rA and rB and increment the pc by 2 so valP=PC+2.

```
4'b1010:                    //pushq
 begin
 rA = Instruction_memory (PC+1)(7:4) ;
 rB = Instruction_memory (PC+1)(3:0) ;
 valP = PC+64'd2;
 end
```

2.1.2.12 FETCH FOR POPQ

- As the name suggests it wil pop the data to higer value increment is 8 normally so it should be stored on memory of initial memory + 8
- it will take PC and PC+1 from instruction memory and will keep PC as icode(1011):ifun and PC+1 as rA and rB and increment the pc by 2 so valP=PC+2.

```
4'b1011:                    //popq
begin
rA = Instruction_memory (PC+1)(7:4) ;
rB = Instruction_memory (PC+1)(3:0) ;
valP = PC+64'd2;
end
```

2.1.3 FINALISED CODE FOR FETCH

- we took negedge fetch and stated the validity and imem error etc of the fetch and directly given instruction memory from fetch.

module fetch(clk,PC,rA,rB,valC,icode,ifun,halt,valP,instr_valid,imem_error);

```
module fetch(clk,PC,rA,rB,valC,icode,ifun,halt,valP,instr_valid,imem_error);

    input clk;
input  (63:0) PC;
output reg (3:0) rA;
output reg (3:0) rB ;
output reg (63:0) valC;
```

output reg (3:0) icode;
output reg (3:0) ifun;
output reg halt ;

output reg (63:0) valP; // it stores the new PC
output reg instr_valid; // it checks wheter itnstruction is true or false
output reg imem_error; // it checks where the PC is more than the instruction memeory or not
reg (7:0) Instruction_memory (0:1023) ; // it stores the insrtuction bytes
reg (63:0) Instruction; // it combines the needfull bytes for valC

initial begin

```
Instruction_memory[0]=8'b00010000; // 1 0
Instruction_memory[1]=8'b01100001; //6 fn
Instruction_memory[2]=8'b01110110; //rA rB
Instruction_memory[3]=8'b00010000;
Instruction_memory[4]=8'b00010000;

Instruction_memory[5]=8'b00100000;
Instruction_memory[6]=8'b00000010;
Instruction_memory[7]=8'b00010000;
Instruction_memory[8]=8'b00010000;
Instruction_memory[9]=8'b00000000;
```

end

always @(negedge clk)
begin

Instruction = {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) };

        imem_error = 0 ;
if (PC>1023)
imem_error = 1'b1 ;



        icode = Instruction_memory (PC)(7:4) ;
 ifun  = Instruction_memory (PC)(3:0) ;

       assign instr_valid = 1'b1 ;

       case (icode)
    4'b0000 :
    begin                  // halt
    halt = 1'b1 ;
    valP = PC +1 ;
    end
```

```
    4'b0001:                        //nop
    valP=PC+1;

    4'b0010:                        //cmovxx
    begin
    rA = Instruction_memory [PC+1][7:4] ;
    rB = Instruction_memory [PC+1][3:0] ;
    valP = PC+64'd2;
    end

    4'b0011:                         //irmovq
    begin
    rA = Instruction_memory [PC+1][7:4] ;
    rB = Instruction_memory [PC+1][3:0] ;
    valC = Instruction ;
    valP = PC+64'd10;
```

```
end

4'b0100:                      //rmmovq
begin
rA = Instruction_memory [PC+1][7:4] ;
rB = Instruction_memory [PC+1][3:0] ;
valC = Instruction ;
valP = PC+64'd10;
end

4'b0101:                     //mrmovq
begin
rA = Instruction_memory [PC+1][7:4] ;
rB = Instruction_memory [PC+1][3:0] ;
valC = Instruction ;
valP = PC+64'd10;
end

4'b0110:                      //OPq
begin
rA = Instruction_memory [PC+1][7:4] ;
rB = Instruction_memory [PC+1][3:0] ;
valP = PC+64'd2;
end

4'b0111:                      //jxx
begin
valC={Instruction_memory[PC+1], Instruction[63:8]} ;
valP=PC+64'd9;
end

4'b1000:                     //call
begin
valC={Instruction_memory[PC+1], Instruction[63:8]} ;
valP=PC+64'd9;
end

4'b1001:
begin                       //ret
valP = PC+64'd1;
end

4'b1010:                      //pushq
```

```
begin
rA = Instruction_memory [PC+1][7:4] ;
rB = Instruction_memory [PC+1][3:0] ;
valP = PC+64'd2;
end

4'b1011:                        //popq
begin
rA = Instruction_memory [PC+1][7:4] ;
rB = Instruction_memory [PC+1][3:0] ;
valP = PC+64'd2;
end

default:  instr_valid=1'b0;
```

endcase

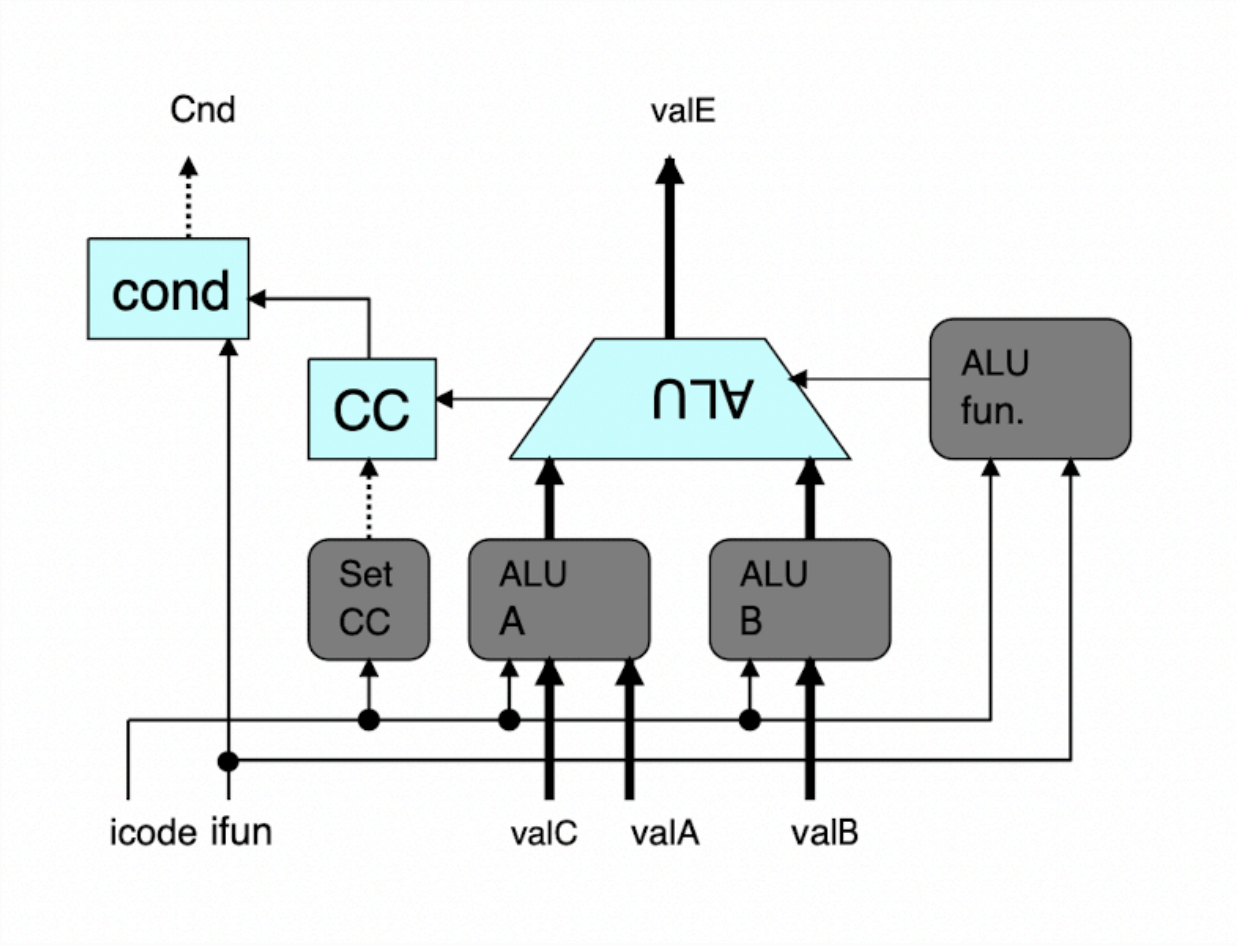
end

endmodule

2.2 EXECUTE MODULE

2.2.1 BASIC OVERVIEW OF EXECUTE

- Execute mainly focuses on the operations of alu that should be taken care of the valA and valB given in decode.
- it will give outputs as Zf,Sf,Of,valE mainly.



- above is the structure of execution of our ALU bit.

2.2.2 EXECUTION OF DIFFERENT INSTRUCTIONS

- Here we discuss about the executions of different instructions

2.2.2.1 EXECUTION FOR HALT

- As the halt name suggests there won't be any execution that will happen here

2.2.2.2 EXECUTION FOR NOP

- As the nop name suggests there won't be any executions that will happen in nop

2.2.2.3 EXECUTION FOR CMOVXX

- There wont be much executions happen here it will execute the condition control using the flags that we given sf zf and of on the basis of ifun.

Instruction		Synonym	Move condition	Description
cmove	S, R	cmovz	ZF	Equal / zero
cmovne	S, R	cmovnz	~ZF	Not equal / not zero
cmovs	S, R		SF	Negative
cmovns	S, R		~SF	Nonnegative
cmovg	S, R	cmovnle	~(SF ^ OF) & ~ZF	Greater (signed >)
cmovge	S, R	cmovnl	~(SF ^ OF)	Greater or equal (signed >=)
cmovl	S, R	cmovnge	SF ^ OF	Less (signed <)
cmovle	S, R	cmovng	(SF ^ OF) | ZF	Less or equal (signed <=)
cmova	S, R	cmovnbe	~CF & ~ZF	Above (unsigned >)
cmovae	S, R	cmovnb	~CF	Above or equal (Unsigned >=)
cmovb	S, R	cmovnae	CF	Below (unsigned <)
cmovbe	S, R	cmovna	CF | ZF	below or equal (unsigned <=)

- on the basis of the condition cnd is 1 if the above move condition satisfies.

2.2.2.4 EXECUTE FOR IRMOVQ

- As we directly give the valC to the register so there will be one operation that is valE= 64'd0+valC.

2.2.2.5 EXECUTE FOR RMMOVQ AND MRMOVQ

- AS the register to memory and memory to register will have the same execution cause it will calculate the adress of the memory
- Which is valE=valC+valB.

2.2.2.6 EXECUTE FOR OPQ

- As the name suggests there will be operations happening between the valA and valB on the basis of the ifun.
- we will be adding those 2 numbers if ifun=0000 and sub and multiply etc on the basis of the ifun i will be keeping raw code below

```
else if(icode4'b0110)
        begin
            if(ifun4'b0000)
                begin
                    control=2'b00;
```

```
            a=valA;
            b=valB;
        end
        else if(ifun4'b0001)
        begin
            control=2'b01;
            a=valB;
            b=valA;
        end
        else if(ifun4'b0010)
        begin
            control=2'b10;
            a=valA;
            b=valB;
        end
        else if(ifun==4'b0011)
        begin
            control=2'b11;
            a=valA;
            b=valB;
        end
        assign fans=ans;
        valE=fans;
    end
```

2.2.2.7 EXECUTE FOR JXX

- There wont be much executions happen here it will execute the conditional jump using the flags that we given sf zf and of on the basis of ifun.

Instruction		Synonym	Jump condition	Description
`jmp`	*Label*		1	Direct jump
`jmp`	**Operand*		1	Indirect jump
`je`	*Label*	`jz`	`ZF`	Equal / zero
`jne`	*Label*	`jnz`	`~ZF`	Not equal / not zero
`js`	*Label*		`SF`	Negative
`jns`	*Label*		`~SF`	Nonnegative
`jg`	*Label*	`jnle`	`~(SF ^ OF) & ~ZF`	Greater (signed >)
`jge`	*Label*	`jnl`	`~(SF ^ OF)`	Greater or equal (signed >=)
`jl`	*Label*	`jnge`	`SF ^ OF`	Less (signed <)
`jle`	*Label*	`jng`	`(SF ^ OF) | ZF`	Less or equal (signed <=)
`ja`	*Label*	`jnbe`	`~CF & ~ZF`	Above (unsigned >)
`jae`	*Label*	`jnb`	`~CF`	Above or equal (unsigned >=)
`jb`	*Label*	`jnae`	`CF`	Below (unsigned <)
`jbe`	*Label*	`jna`	`CF | ZF`	Below or equal (unsigned <=)

2.2.2.8 EXECUTE FOR CALL

- for call there will be simple execution that will just decrease the valC with a 8 and name it as valE.

2.2.2.9 EXECUTE FOR RETURN

- For call there will be simple execution that will just increase the valC with a 8 and name it as valE.

2.2.2.10 EXECUTE FOR PUSH

- here we use valE to determine the memory adress so we need to decrement the valB with 8.
- valE = -64'd8 +valB.

2.2.2.11 EXECUTE FOR POP

- Here we use valE to determine the memory adress so we need to increment the valB with 8.
- valE = 64'd8 +valB.

2.2.3 FINALISED CODE FOR EXECUTE

```
	`timescale 1ns/1ps

	`include"alu.v"

	module
execute(clk,icode,ifun,valA,valB,valC,valE,cnd,zeroflag,signflag,overflag);

	input clk;
input (3:0) icode;
input (3:0) ifun;
input signed (63:0) valA;
input signed (63:0) valB;
input signed (63:0) valC;

	output reg (63:0) valE;
output reg cnd;

	output reg zeroflag;
output reg signflag;
output reg overflag;

	always@(*)
begin
   if(clk==0 && icode == 4'b0110)
	 begin

		zeroflag=(ans==1'b0);

		signflag=(ans<1'b0);

		overflag=(a<1'b0==b<1'b0)&&(ans<1'b0!=a<1'b0);

	end

	end
```

```
    initial begin
   zeroflag=0;
   signflag=0;
   overflag=0;
end

    reg signed (63:0) fans;
reg (1:0) control;
reg signed (63:0) a;
reg signed (63:0) b;

    wire signed (63:0) ans;
wire signed overflow;

    alu module1(
```

```
  .ans(ans),
  .overflow(overflow),
  .control(control),
  .a(a),
  .b(b)
```

```
    );

    reg xor1;
reg xor2;
reg or1;
reg or2;
reg and1;
reg and2;
reg not1;

    wire xor_out;
wire or_out;
wire and_out;
wire not_out;
```

xor g1(xor_out,xor1,xor2);
or g2(or_out,or1,or2);
and g3(and_out,and1,and2);
not g4(not_out,not1);

//initialisation

initial
begin

```
control = 2'b00;
a=64'b0;
b=64'b0;
```

end

always@(*)
begin
if(clk==0)
begin

```
cnd=0;

 if(icode==4'b0010)
     begin

          if(ifun==4'b0000)
             begin
                 cnd=1;
             end
             else if(ifun==4'b0001)
             begin

                 xor1 = signflag;
                 xor2 = overflag;

                 if(xor_out)
                 begin
                     cnd=1;
```

```
        end
        else if(zeroflag)
        begin
            cnd=1;
        end
    end
    else if(ifun==4'b0010)
    begin
        xor1=signflag;
        xor2=overflag;
        if(xor_out)
        begin
            cnd=1;
        end
    end
    else if(ifun==4'b0011)
    begin
        if(zeroflag)
        begin
            cnd=1;
        end
    end
    else if(ifun==4'b0100)
    begin
        not1=zeroflag;
        if(not_out)
        begin
            cnd=1;
        end
    end
    else if(ifun==4'b0101)
    begin
        xor1=signflag;
        xor2=overflag;
        not1=xor_out;
        if(not_out)
        begin
            cnd=1;
        end
    end
    else if(ifun==4'b0110)
    begin
        xor1=signflag;
```

```
                xor2=overflag;
                not1=xor_out;
                if(not_out)
                begin
                    not1=zeroflag;
                    if(not_out)
                    begin
                        cnd=1;
                    end
                end
            end

            valE=64'd0+valA;
    end

       else if(icode==4'b0011)
       begin
            valE=64'd0+valC;
       end

       else if(icode==4'b0100)
       begin
            valE=valB+valC;
       end

       else if(icode==4'b0101)
       begin
            valE=valB+valC;
       end

    else if(icode==4'b0110)
    begin
            if(ifun==4'b0000)
            begin
                control=2'b00;
                a=valA;
                b=valB;
            end
            else if(ifun==4'b0001)
            begin
                control=2'b01;
                a=valB;
                b=valA;
```

```
            end
            else if(ifun==4'b0010)
            begin
                control=2'b10;
                a=valA;
                b=valB;
            end
            else if(ifun==4'b0011)
            begin
                control=2'b11;
                a=valA;
                b=valB;
            end
            assign fans=ans;
            valE=fans;
    end
    if(icode==4'b0111)
    begin
        if(ifun==4'b0000)
        begin
            cnd=1;
        end
        else if(ifun==4'b0001)
        begin
            xor1=signflag;
            xor2=overflag;
            if(xor_out)
            begin
                cnd=1;
            end
            else if(zeroflag)
            begin
                cnd=1;
            end
        end
        else if(ifun==4'b0010)
        begin
            xor1=signflag;
            xor2=overflag;
            if(xor_out)
            begin
                cnd=1;
            end
```

```
end
else if(ifun==4'b0011)
begin
    if(zeroflag)
    begin
        cnd=1;
    end
end
else if(ifun==4'b0101)
begin
    xor1=signflag;
    xor2=overflag;
    not1=xor_out;
    if(not_out)
    begin
        cnd=1;
    end
end
else if(ifun==4'b0110)
begin
    xor1=signflag;
    xor2=overflag;
    not1=xor_out;
    if(not_out)
    begin
        not1=zeroflag;
        if(not_out)
        begin
            cnd=1;
        end
    end
end
if(icode==4'b1000)
begin
    valE=-64'd8+valB;
end
if(icode==4'b1001)
begin
    valE=64'd8+valB;
end
if(icode==4'b1010)
begin
    valE=-64'd8+valB;
```

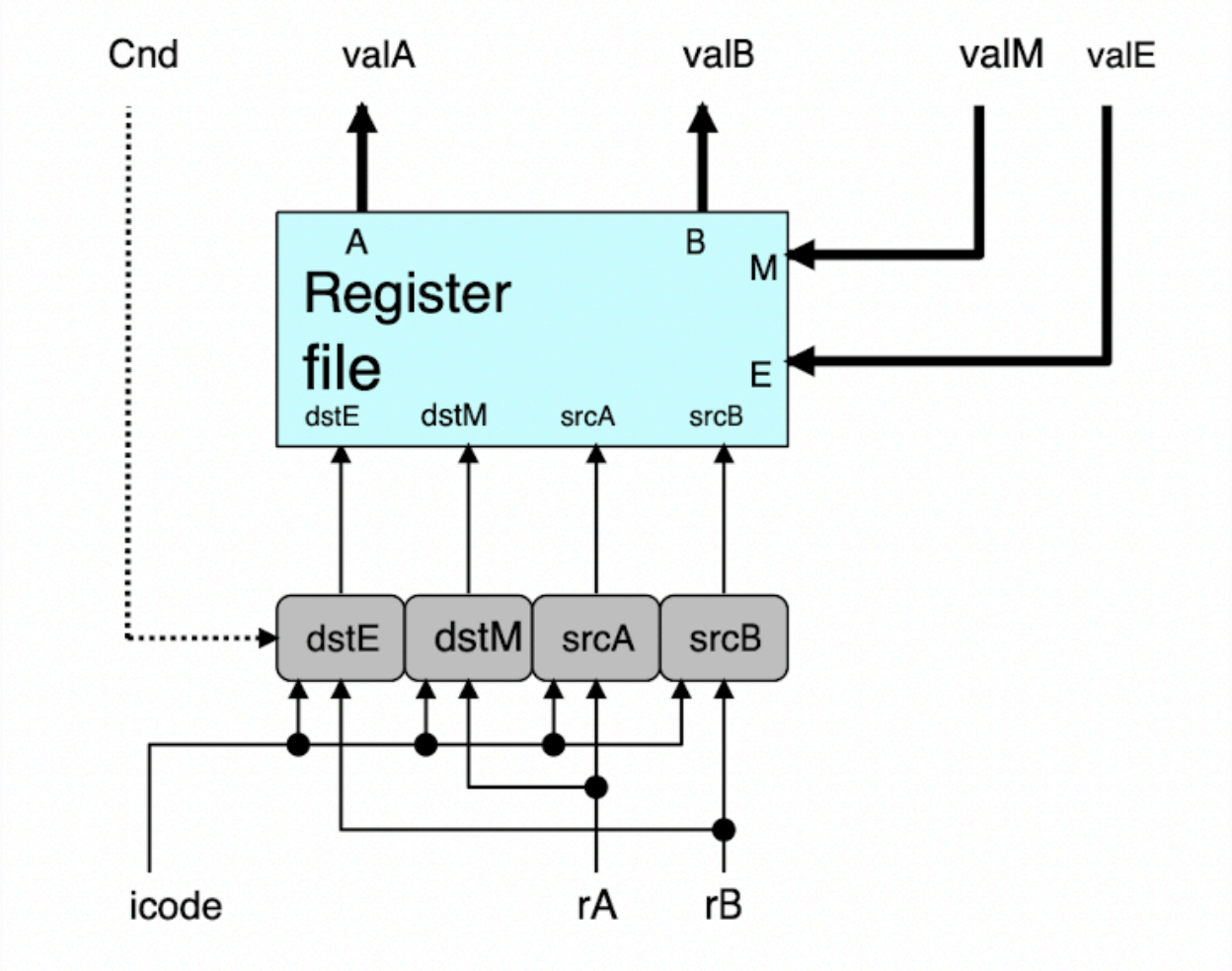
```
                    end
                    if(icode==4'b1011)
                    begin
                        valE=64'd8+valB;
                    end
                end

    end
end
```

endmodule

2.3 DECODE AND WRITE BACK MODULE

BASIC INFORMATION ABOUT DECODE BLOCK

- This block gives the values of valA and valB or only valA or only valB depending on the case used
- below figure will help you to understand the implementation better.



DECODE AND WRITEBACK IMPLEMENTATION ON VARIOUS INSTRUNCTION SETS

DECODE AND WRITEBACK FOR CMOVXX

- In cmovxx we need only the value of A which will be taken from register memory rA
- For writeback if the condition is 1 keep register memory rB as valE.

DECODE AND WRITEBACK FOR IRMOVQ

- Decode will not be required for the irmovq and for the write back we will be giving the register memory rB value of valE to it.

DECODE AND WRITE baCK FOR MRMOVQ

- Decode the valB take it from the register memory rB in MRMOVQ.
- Then write back the value of valM to the register memory rB.

DECODE AND WRITE BACK OPQ

- Decode the valA and valB from the register memory rA and rB respectively.
- write back the valE to the registermemory of rB.

DECODE AND WRITE BACK CALL

- Decode the valB from rsp stack pointer.
- write back the stack pointer with valE.

DECODE AND WRITE BACK RETURN

- Decode the valA and valB from the register memory of rsp.
- write back the stack pointer rsp with the valE(executed value).

DECODE AND WRITE BACK PUSHQ

- Decode the valA and valB from register memory rA and rsp respectively.
- Write back the stack point register rsp with the valE.

DECODE AND WRITE BACK POPQ

- Decode the valA and valB from the register memory of rsp.
- write back the rsp by valE and R(rA) with valM.

AND OTHERS WONT NEED ANY DECODE AND WRITE BACKS.

FINALISED CODE FOR DECODE WRITE BACK

`timescale 1ns/1ps

```
    module
decode(clk,icode,rA,rB,cnd,valA,valB,valM,valE,rax,rcx,rdx,rbx,
rsp,rbp,rsi,rdi,r8,r9,r10,r11,r12,r13,r14) ;

    input clk;
    input [3:0] icode;
    input  [3:0] rA;
    input [3:0] rB;
    input [63:0] valM;
    input [63:0] valE;
    input cnd;

    output reg signed [63:0] valA;
    output reg signed [63:0] valB;
```

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

reg [63:0] reg_mem[0:14];
initial begin
    reg_mem[0] = 64'd0;
    reg_mem[1] = 64'd2;
    reg_mem[2] = 64'd4;
    reg_mem[3] = 64'd6;
    reg_mem[4] = 64'd8;
    reg_mem[5] = 64'd16;
    reg_mem[6] = 64'd32;
    reg_mem[7] = 64'd64;
    reg_mem[8] = 64'd128;
    reg_mem[9] = 64'd256;
    reg_mem[10] = 64'd512;
    reg_mem[11] = 64'd1024;
    reg_mem[12] = 64'd2048;
    reg_mem[13] = 64'd4096;
    reg_mem[14] = 64'd8192;
end
```

```
always@(*)

begin

    case(icode)
```

```
4'b0010 : begin
//cmovxx
valA=reg_mem[rA];

end

4'b0100 : begin
    //rmmovq
    valA=reg_mem[rA];
    valB=reg_mem[rB];
end

4'b0101 : begin
    //mrmovq
    valB = reg_mem[rB];
end

4'b0110 : begin
    //OPq
    valA = reg_mem[rA];
    valB = reg_mem[rB];

end

4'b1000 : begin
    //call
    valB=reg_mem[4];
end

4'b1001 : begin
    //ret
    valA = reg_mem[4];
    valB = reg_mem[4];
end

4'b1010 : begin
    //pushq
    valA = reg_mem[rA];
    valB = reg_mem[4];
end

4'b1011 : begin
    //popq
```

```
        valA = reg_mem[4];
        valB = reg_mem[4];
    end

    endcase

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

always@(posedge clk)

    begin

    case(icode)

        4'b0010 : begin
            //cmovxx
            if(cnd==1'b1)
            begin
                reg_mem[rB] = valE;
            end
        end

        4'b0011 : begin
            //irmovq
            reg_mem[rB] = valE;
        end

    4'b0101 : begin
```

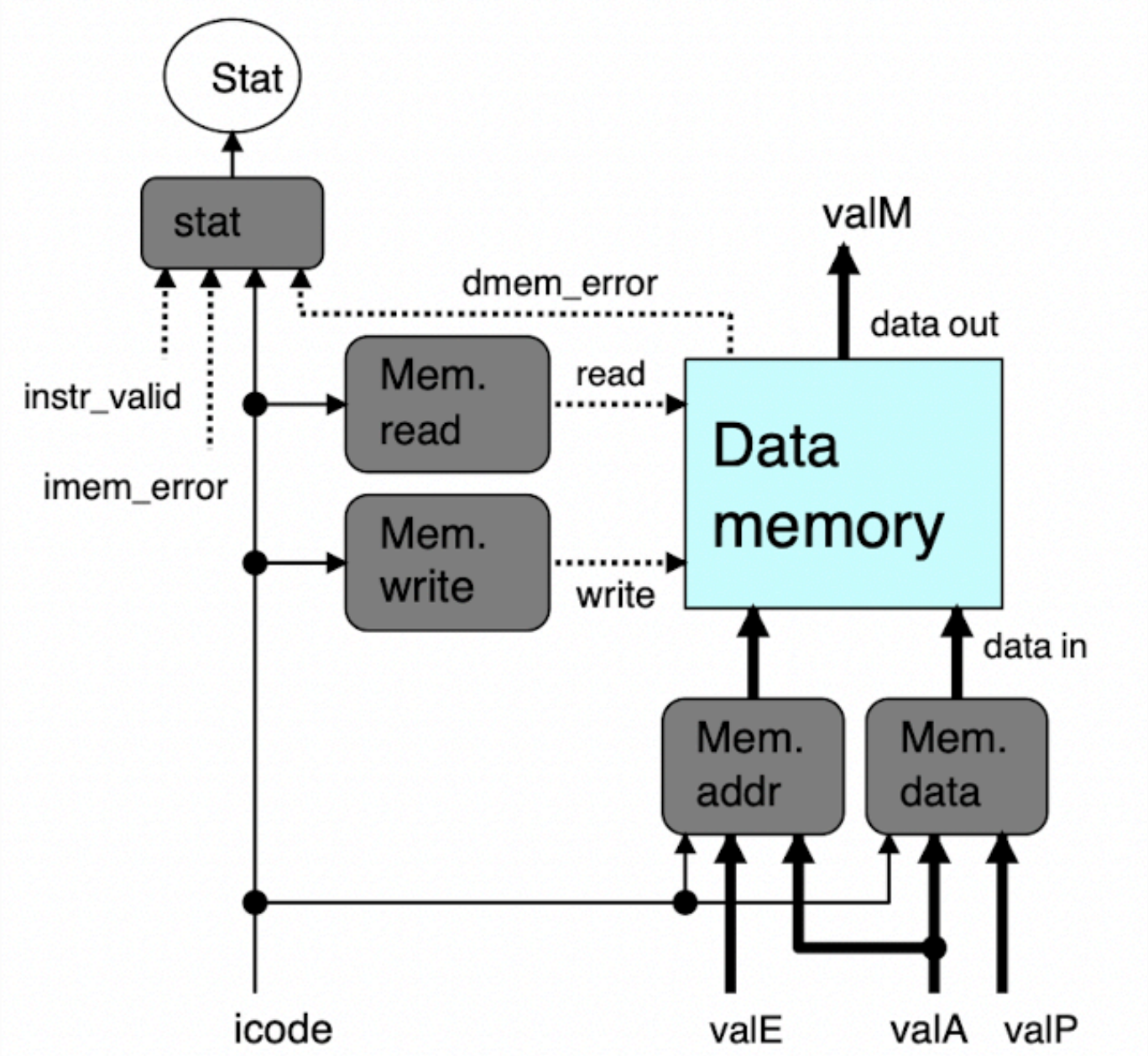
```
    //mrmovq
    reg_mem[rA] = valM;
end

4'b0110 : begin
    //OPq
    reg_mem[rB] = valE;
end

4'b1000 : begin
    //call
    reg_mem[4] = valE;
end
4'b1001 : begin
    //ret
    reg_mem[4] = valE;
end

4'b1010 : begin
    //pushq
    reg_mem[4] = valE;
end

4'b1011 : begin
    //popq
    reg_mem[4] = valE;
    reg_mem[rA] = valM;
end
endcase

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

```
        r13 = reg_mem[13];
        r14 = reg_mem[14];

    end

    endmodule
```

2.4 MEMORY MODULE

BASIC INFORMATION ABOUT MEMORY MODULE

- Memory module will store memory of bits and can be called or used any time we need it will need an adress to pull or push the data to memory it will mainly used in rmmovq mrmovq pop push etc etc.



- We decide the adress of memory by valE and depending on the instruction we will write or read the memory of the above data memory
- basically stores data of the processor

FINALISED CODE FOR THE MEMORY MODULE

`timescale 1ns/1ps

```
module memory(clk,icode,valA,valB,valE,valP,valM,data);

    integer i=0;
```

```
    input clk;
    input [3:0] icode;
    input signed [63:0] valA;
    input signed [63:0] valB;
    input [63:0] valE;
    input [63:0] valP;

    output reg [63:0] valM;
    output reg [63:0] data;
    reg [63:0] data_mem[0:1023];

    initial begin

     for(i=0;i<1024;i=i+1) begin

         data_mem[i]=64'd2;

     end
```

```
    end

always@(*)
    begin

        case(icode)

                4'b0100 : begin
                    //rmmovq
                    data_mem[valE] = valA;

                end

                4'b0101 : begin
                    //mrmovq
                    valM = data_mem[valE];
                end

                4'b1000 : begin
```

```
                    //call
                    data_mem[valE] = valP;
                end

                4'b1001 : begin
                    //ret
                    valM = data_mem[valA];
                end

                4'b1010 : begin
                    //pushq
                    data_mem[valE] = valA;
                end

                4'b1011 : begin
                    //popq
                    valM = data_mem[valE];
                end
        endcase

        data = data_mem[valE];
```

```
    end

endmodule
```

2.5 PCUPDATE MODULE

- In this module we will just update the PC with the respective valP for normal conditions and for CMOVXX JXX and for some instructions we will get valM or valC as PCUPDATE.
- Basic code for the PCUPDATAE.
- `timescale 1ns/1ps

  `timescale 1ns/1ps

  module pcupdate(clk,PC,cnd,icode,valC,valM,valP,PC_update);

input clk;

input cnd;

input (3:0) icode;

input (63:0) valC;

input (63:0) valP;

input (63:0) valM;

input (63:0) PC;

output reg (63:0) PC_update;

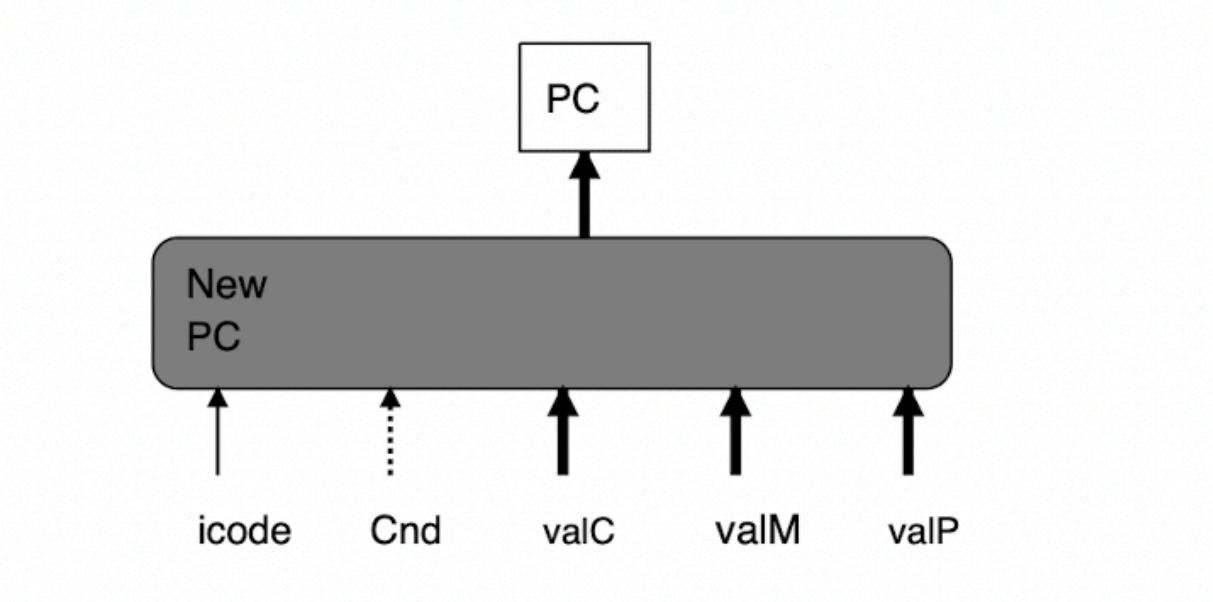
always@(*)

```
begin

      case(icode)

          4'b0000 : begin

              PC_update = valP;

          end

          4'b0001 : begin

              PC_update = valP;

          end

          4'b0010 : begin

              PC_update = valP;

          end

          4'b0010 : begin

              PC_update = valP;

          end

          4'b0011 : begin
```

```
            PC_update = valP;
        end
        4'b0100 : begin
            PC_update = valP;
        end
        4'b0101 : begin
            PC_update = valP;
        end
        4'b0110 : begin
            PC_update = valP;
        end
        4'b0111 : begin
            case(cnd)
             1'b1 : begin
                 PC_update = valC;
             end
             1'b0 : begin
                 PC_update = valP;
             end
            endcase
        end
```

```
            4'b1000 : begin
                PC_update = valC;
            end
            4'b1001 : begin
                PC_update = valM;
            end
            4'b1010 : begin
                PC_update = valP;
            end
            4'b1011 : begin
                PC_update = valP;
            end
        endcase
    end
```

endmodule



- **SEQUENTIAL MODULE**
    - in this module I Included all the above modules and wrote the test benches accordingly and printed the outputs.
    - We created below instruction memory and tested the sequential module and the PC update.
    - code for sequential model
    -

```
`timescale 1ns/1ps

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

module seq;

reg clk;

reg [63:0] PC;

reg stat[0:2];

wire [3:0] icode;
```

```
wire [3:0] ifun;
wire [3:0] rA;
wire [3:0] rB;

wire [63:0] valC;
wire [63:0] valP;
wire signed [63:0] valA;
wire signed [63:0] valB;
wire [63:0] valE;
wire [63:0] valM;
wire [63:0] PC_update;
wire signed [63:0] rax;
wire signed [63:0] rcx;
wire signed [63:0] rdx;
wire signed [63:0] rbx;
wire signed [63:0] rsp;
wire signed [63:0] rbp;
wire signed [63:0] rsi;
wire signed [63:0] rdi;
wire signed [63:0] r8;
wire signed [63:0] r9;
wire signed [63:0] r10;
wire signed [63:0] r11;
wire signed [63:0] r12;
wire signed [63:0] r13;
wire signed [63:0] r14;
wire signed [63:0] data;

wire cnd;
wire haltins;
wire Instr_Valid;
wire imem_error;
wire zeroflag;
wire signflag;
wire overflag;

fetch module1 (
    .clk(clk),
    .PC(PC),
    .rA(rA),
    .rB(rB),
    .valC(valC),
    .icode(icode),
```

```
    .ifun(ifun),
    .halt(haltins),
    .valP(valP),
    .instr_valid(instr_valid),
    .imem_error(imem_error)
    );

    execute module2(
        .clk(clk),
        .icode(icode),
        .ifun(ifun),
        .valA(valA),
        .valB(valB),
        .valC(valC),
        .valE(valE),
        .cnd(cnd),
        .zeroflag(zeroflag),
        .signflag(signflag),
        .overflag(overflag)
    );


//clk,icode,rA,rB,cnd,valA,valB,valM,valE,rax,rc
x,rdx,rbx,rsp,rbp,rsi,rdi,r8,r9,r10,r11,r12,r13,
r14
    decode module3(
        .clk(clk),
        .icode(icode),
        .rA(rA),
        .rB(rB),
        .cnd(cnd),
        .valA(valA),
        .valB(valB),
        .valM(valM),
        .valE(valE),
        .rax(rax),
        .rcx(rcx),
        .rdx(rdx),
        .rbx(rbx),
        .rsp(rsp),
        .rbp(rbp),
        .rsi(rsi),
        .rdi(rdi),
```

```
        .r8(r8),
        .r9(r9),
        .r10(r10),
        .r11(r11),
        .r12(r12),
        .r13(r13),
        .r14(r14)
    );

     //clk,icode,valA,valB,valE,valP,valM,data
    memory module4(
        .clk(clk),
        .icode(icode),
        .valA(valA),
        .valB(valB),
        .valE(valE),
        .valP(valP),
        .valM(valM),
        .data(data)
    );

    //clk,PC,cnd,icode,valC,valM,valP,PC_update
    pcupdate module5(
        .clk(clk),
        .PC(PC),
        .cnd(cnd),
        .icode(icode),
        .valC(valC),
        .valM(valM),
        .valP(valP),
        .PC_update(PC_update)
    );

always #20 clk=~clk;

initial begin

    $dumpfile("seq.vcd");
    $dumpvars(0,seq);
    stat[0]=1;
    stat[1]=0;
    stat[2]=0;
    clk=0;
```

```
    PC=64'd0;
 end

 always@(*)
 begin
     PC=PC_update;
 end

 always@(*)
 begin
     if(haltins)
     begin
         stat[2]=haltins;
         stat[1]=1'b0;
         stat[0]=1'b0;
     end
     else if(instr_valid)
     begin
         stat[1]=instr_valid;
         stat[2]=1'b0;
         stat[0]=1'b0;
     end
     else
     begin
         stat[0]=1'b1;
         stat[1]=1'b0;
         stat[2]=1'b0;
     end
 end
 always@(*)
begin
    if(stat[2]==1'b1)
    begin
        $finish;
    end
end

initial
```

```
  $monitor("PC=%d\n clk=%d\n icode=%b\n
ifun=%b\n rA=%b\n rB=%b\n valA=%d\n valB=%d\n
valC=%d\n valE=%d\n valM=%d\n instr_valid=%d\n
imem_error=%d\n cnd=%d\n halt=%d\n rax=%d\n
rcx=%d\n rdx=%d\n rbx=%d\n rsp=%d\n rbp=%d\n
rsi=%d\n rdi=%d\n r8=%d\n r9=%d\n r10=%d\n
r11=%d\n r12=%d\n r13=%d\n r14=%d\n data=%d \n
",PC,clk,icode,ifun,rA,rB,valA,valB,valC,valE,va
lM,instr_valid,imem_error,cnd,stat[2],rax,rcx,rd
x,rbx,rsp,rbp,rsi,rdi,r8,r9,r10,r11,r12,r13,r14,
data);

endmodule
```

Instruction_memory(0)=8'b00010000; // 1 0

Instruction_memory(1)=8'b01100001; //6 fn

Instruction_memory(2)=8'b01110110; //rA rB

Instruction_memory(3)=8'b00010000;

Instruction_memory(4)=8'b00010000;

```
Instruction_memory[5]=8'b00100000;
Instruction_memory[6]=8'b00000010;
Instruction_memory[7]=8'b00010000;
Instruction_memory[8]=8'b00010000;
Instruction_memory[9]=8'b00000000;
```

I Used OPQ ,NOP,CMOVXX,HALT in the above instruction memory

THE OUTPUT FOR THE ABOVE INSTRUCTION MEMORY IS GIVEN BELOW WE VERIFIED AND ITS WORKING ACCORDINGLY.

```
PC=                    1
 clk=0
 icode=0001
 ifun=0000
 rA=xxxx
 rB=xxxx
 valA=                  x
 valB=                  x
 valC=                  x
 valE=                  x
 valM=                  x
 instr_valid=1
 imem_error=0
 cnd=0
 halt=0
 rax=                   0
 rcx=                   2
 rdx=                   4
 rbx=                   6
 rsp=                   8
 rbp=                  16
 rsi=                  32
 rdi=                  64
 r8=                  128
 r9=                  256
 r10=                 512
 r11=                1024
 r12=                2048
 r13=                4096
 r14=                8192
 data=                  x

PC=                    1
 clk=1
 icode=0001
 ifun=0000
 rA=xxxx
 rB=xxxx
 valA=                  x
 valB=                  x
 valC=                  x
 valE=                  x
```

```
 valM=                    x
 instr_valid=1
 imem_error=0
 cnd=0
 halt=0
 rax=                     0
 rcx=                     2
 rdx=                     4
 rbx=                     6
 rsp=                     8
 rbp=                    16
 rsi=                    32
 rdi=                    64
 r8=                    128
 r9=                    256
 r10=                   512
 r11=                  1024
 r12=                  2048
 r13=                  4096
 r14=                  8192
 data=                    x

PC=                      3
 clk=0
 icode=0110
 ifun=0001
 rA=0111
 rB=0110
 valA=                   64
 valB=                   32
 valC=                    x
 valE=18446744073709551584
 valM=                    x
 instr_valid=1
 imem_error=0
 cnd=0
 halt=0
 rax=                     0
 rcx=                     2
 rdx=                     4
 rbx=                     6
 rsp=                     8
 rbp=                    16
```

```
 rsi=                  32
 rdi=                  64
 r8=                  128
 r9=                  256
 r10=                 512
 r11=                1024
 r12=                2048
 r13=                4096
 r14=                8192
 data=                   x

PC=                   3
 clk=1
 icode=0110
 ifun=0001
 rA=0111
 rB=0110
 valA=                 64
 valB=                -32
 valC=                  x
 valE=18446744073709551584
 valM=                  x
 instr_valid=1
 imem_error=0
 cnd=0
 halt=0
 rax=                   0
 rcx=                   2
 rdx=                   4
 rbx=                   6
 rsp=                   8
 rbp=                  16
 rsi=                 -32
 rdi=                  64
 r8=                  128
 r9=                  256
 r10=                 512
 r11=                1024
 r12=                2048
 r13=                4096
 r14=                8192
 data=                   x
```

```
PC=                    4
 clk=0
 icode=0001
 ifun=0000
 rA=0111
 rB=0110
 valA=                  64
 valB=                 -32
 valC=                   x
 valE=18446744073709551584
 valM=                   x
 instr_valid=1
 imem_error=0
 cnd=0
 halt=0
 rax=                    0
 rcx=                    2
 rdx=                    4
 rbx=                    6
 rsp=                    8
 rbp=                   16
 rsi=                  -32
 rdi=                   64
 r8=                   128
 r9=                   256
 r10=                  512
 r11=                 1024
 r12=                 2048
 r13=                 4096
 r14=                 8192
 data=                   x

PC=                    4
 clk=1
 icode=0001
 ifun=0000
 rA=0111
 rB=0110
 valA=                  64
 valB=                 -32
 valC=                   x
 valE=18446744073709551584
 valM=                   x
```

```
 instr_valid=1
 imem_error=0
 cnd=0
 halt=0
 rax=                   0
 rcx=                   2
 rdx=                   4
 rbx=                   6
 rsp=                   8
 rbp=                  16
 rsi=                 -32
 rdi=                  64
 r8=                  128
 r9=                  256
 r10=                 512
 r11=                1024
 r12=                2048
 r13=                4096
 r14=                8192
 data=                   x

PC=                   5
 clk=0
 icode=0001
 ifun=0000
 rA=0111
 rB=0110
 valA=                  64
 valB=                 -32
 valC=                   x
 valE=18446744073709551584
 valM=                   x
 instr_valid=1
 imem_error=0
 cnd=0
 halt=0
 rax=                   0
 rcx=                   2
 rdx=                   4
 rbx=                   6
 rsp=                   8
 rbp=                  16
 rsi=                 -32
```

```
 rdi=                  64
 r8=                 128
 r9=                 256
 r10=                512
 r11=               1024
 r12=               2048
 r13=               4096
 r14=               8192
 data=                 x

PC=                  5
 clk=1
 icode=0001
 ifun=0000
 rA=0111
 rB=0110
 valA=                64
 valB=               -32
 valC=                 x
 valE=18446744073709551584
 valM=                 x
 instr_valid=1
 imem_error=0
 cnd=0
 halt=0
 rax=                  0
 rcx=                  2
 rdx=                  4
 rbx=                  6
 rsp=                  8
 rbp=                 16
 rsi=                -32
 rdi=                 64
 r8=                 128
 r9=                 256
 r10=                512
 r11=               1024
 r12=               2048
 r13=               4096
 r14=               8192
 data=                 x

PC=                  7
```

```
 clk=0
 icode=0010
 ifun=0000
 rA=0000
 rB=0010
 valA=                   0
 valB=                 -32
 valC=                   x
 valE=                   0
 valM=                   x
 instr_valid=1
 imem_error=0
 cnd=1
 halt=0
 rax=                    0
 rcx=                    2
 rdx=                    4
 rbx=                    6
 rsp=                    8
 rbp=                   16
 rsi=                  -32
 rdi=                   64
 r8=                   128
 r9=                   256
 r10=                  512
 r11=                 1024
 r12=                 2048
 r13=                 4096
 r14=                 8192
 data=                   2

PC=                     7
 clk=1
 icode=0010
 ifun=0000
 rA=0000
 rB=0010
 valA=                   0
 valB=                 -32
 valC=                   x
 valE=                   0
 valM=                   x
 instr_valid=1
```

```
 imem_error=0
 cnd=1
 halt=0
 rax=                  0
 rcx=                  2
 rdx=                  0
 rbx=                  6
 rsp=                  8
 rbp=                 16
 rsi=                -32
 rdi=                 64
 r8=                 128
 r9=                 256
 r10=                512
 r11=               1024
 r12=               2048
 r13=               4096
 r14=               8192
 data=                 2

PC=                  8
 clk=0
 icode=0001
 ifun=0000
 rA=0000
 rB=0010
 valA=                 0
 valB=               -32
 valC=                 x
 valE=                 0
 valM=                 x
 instr_valid=1
 imem_error=0
 cnd=0
 halt=0
 rax=                  0
 rcx=                  2
 rdx=                  0
 rbx=                  6
 rsp=                  8
 rbp=                 16
 rsi=                -32
 rdi=                 64
```

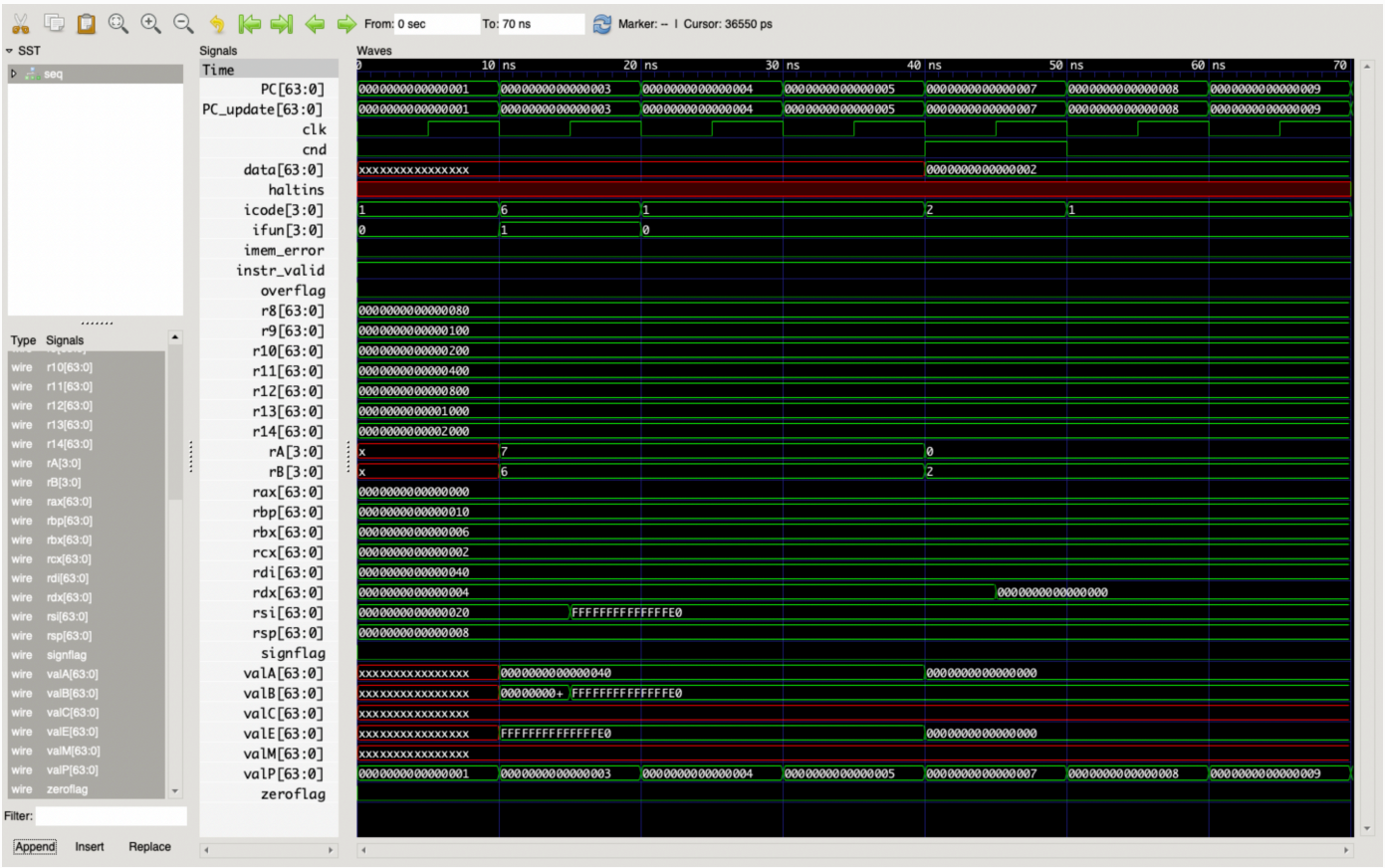
```
 r8=                  128
 r9=                  256
 r10=                  512
 r11=                 1024
 r12=                 2048
 r13=                 4096
 r14=                 8192
 data=                   2

PC=                   8
 clk=1
 icode=0001
 ifun=0000
 rA=0000
 rB=0010
 valA=                   0
 valB=                 -32
 valC=                   x
 valE=                   0
 valM=                   x
 instr_valid=1
 imem_error=0
 cnd=0
 halt=0
 rax=                    0
 rcx=                    2
 rdx=                    0
 rbx=                    6
 rsp=                    8
 rbp=                   16
 rsi=                  -32
 rdi=                   64
 r8=                  128
 r9=                  256
 r10=                  512
 r11=                 1024
 r12=                 2048
 r13=                 4096
 r14=                 8192
 data=                   2

PC=                   9
 clk=0
```

```
 icode=0001
 ifun=0000
 rA=0000
 rB=0010
 valA=                    0
 valB=                  -32
 valC=                    x
 valE=                    0
 valM=                    x
 instr_valid=1
 imem_error=0
 cnd=0
 halt=0
 rax=                     0
 rcx=                     2
 rdx=                     0
 rbx=                     6
 rsp=                     8
 rbp=                    16
 rsi=                   -32
 rdi=                    64
 r8=                    128
 r9=                    256
 r10=                   512
 r11=                  1024
 r12=                  2048
 r13=                  4096
 r14=                  8192
 data=                    2

PC=                      9
 clk=1
 icode=0001
 ifun=0000
 rA=0000
 rB=0010
 valA=                    0
 valB=                  -32
 valC=                    x
 valE=                    0
 valM=                    x
 instr_valid=1
 imem_error=0
```

```
 cnd=0
 halt=0
 rax=                  0
 rcx=                  2
 rdx=                  0
 rbx=                  6
 rsp=                  8
 rbp=                 16
 rsi=                -32
 rdi=                 64
 r8=                 128
 r9=                 256
 r10=                512
 r11=               1024
 r12=               2048
 r13=               4096
 r14=               8192
 data=                 2

PC=                  10
 clk=0
 icode=0000
 ifun=0000
 rA=0000
 rB=0010
 valA=                 0
 valB=               -32
 valC=                 x
 valE=                 0
 valM=                 x
 instr_valid=1
 imem_error=0
 cnd=0
 halt=1
 rax=                  0
 rcx=                  2
 rdx=                  0
 rbx=                  6
 rsp=                  8
 rbp=                 16
 rsi=                -32
 rdi=                 64
 r8=                 128
```

```
r9=                  256
r10=                  512
r11=                 1024
r12=                 2048
r13=                 4096
r14=                 8192
data=                   2
```

WAVEFORM WILL BE GIVEN AS BELOW



3. 5-STAGE PIPELINING

- We should design a 5 stage pipe line here with deriving each of the operations to a register and trying for removing any hazards if possible we added

- Data hazards

  - Instruction having register R as source follows shortly after instruction having register R as destination
  - Common condition, don't want to slow down pipeline

1. - Control hazards

     • Mispredicted conditional branch
     – Our design predicts all branches as being taken – Naïve pipeline executes two extra instructions

     • Getting return address for ret instruction – Naïve pipeline executes three extra instructions

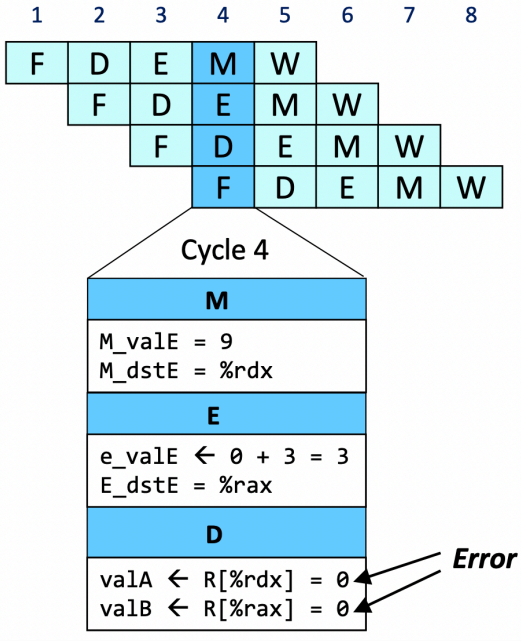
     - Making sure it really works

     • What if multiple special cases happen simultaneously?

Data Dependencies (Revisited)

- No nop

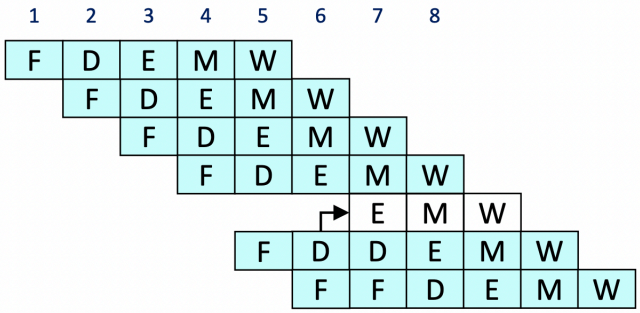
```
0x000:  irmovq   $9,%rdx
0x00a:  irmovq   $3,%rax
0x014:  addq     %rdx,%rax
0x016:  halt
```



(Both %rax and %rdx are initialized to 0)

Stalling for Data Dependencies

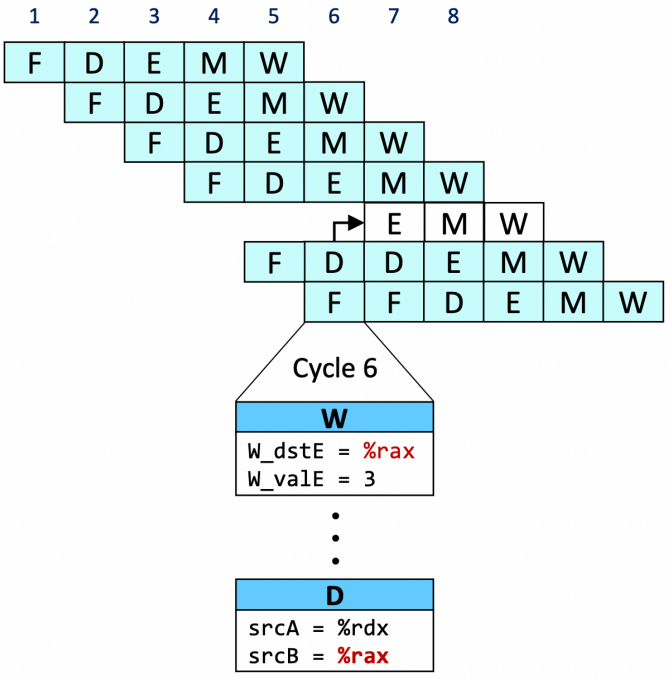
```
0x000:  irmovq   $9,%rdx
0x00a:  irmovq   $3,%rax
0x014:  nop
0x015:  nop
        bubble
0x016:  addq     %rdx,%rax
0x018:  halt
```



- If instruction follows too closely after one that writes register, slow it down
- Hold instruction in decode
- Dynamically inject nop into execute stage

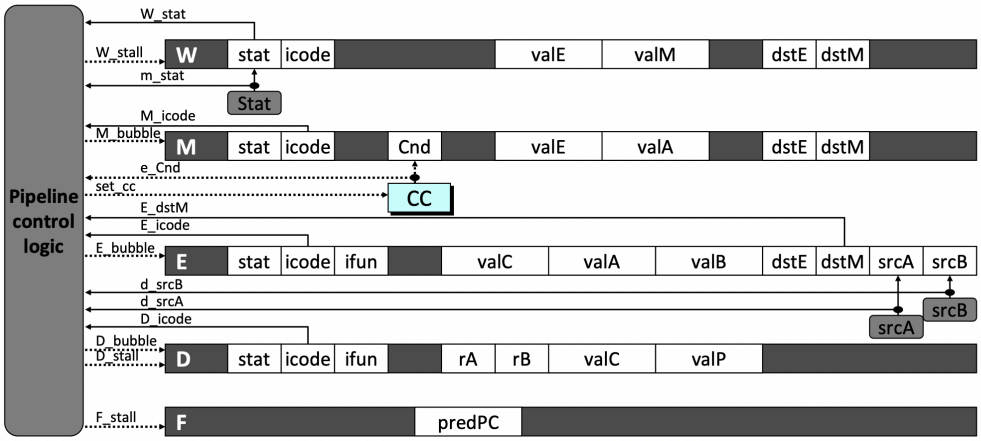
Detecting Stall Condition

```
0x000:  irmovq   $9,%rdx
0x00a:  irmovq   $3,%rax
0x014:  nop
0x015:  nop
        bubble
0x016:  addq     %rdx,%rax
0x018:  halt
```



- finally similar way use bubble which will do no operation in which it is activated.
- now final operations can be seen in below picture.

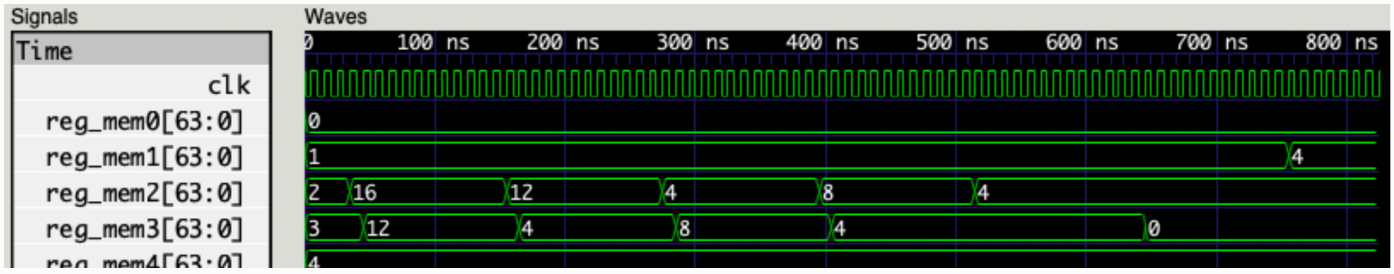
- **Pipeline control**
  - Combinational logic detects stall condition
  - Sets mode signals for how pipeline registers should be updated



- We implemented data forwarding and load/use hazards bubble and stall depending on the instruction memory

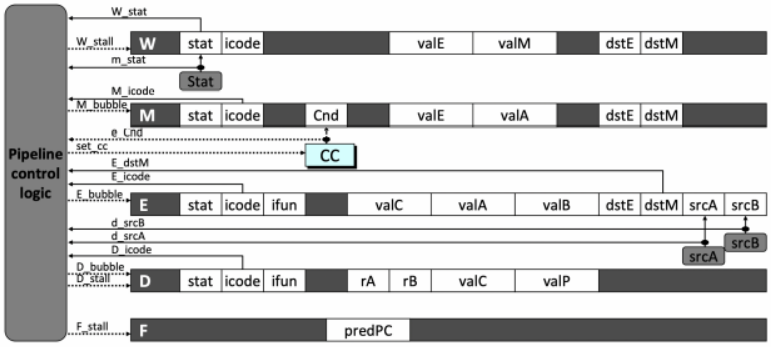
- we built 5 pipe lines and transfer data in between them
- Below re executions of codes of pipeline that we built.

WAVE OUTPUTS FOR PIPELINING



On manuall checking we verified the above output we got.

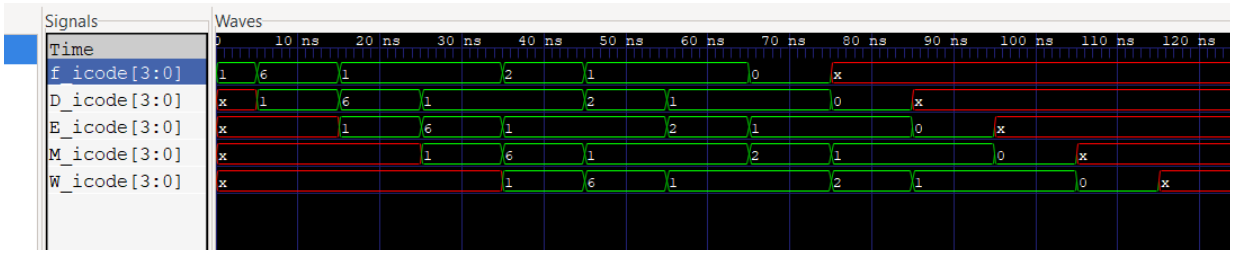
- **Pipeline control**
  - Combinational logic detects stall condition
  - Sets mode signals for how pipeline registers should be updated

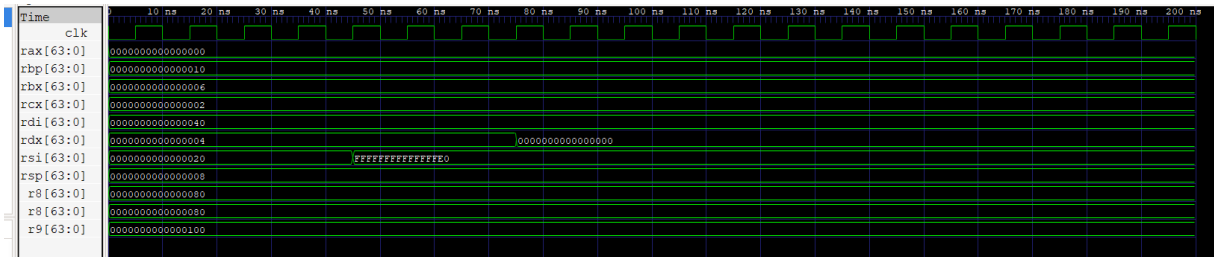


- We implemented data forwarding and load/use hazards bubble and stall depending on the instruction memory

- we built 5 pipe lines and transfer data in between them
- Below re executions of codes of pipeline that we built.

WAVE OUTPUTS FOR PIPELINING





On manuall checking we verified the above output we got.