IPA Final-Evals

Date: 8/3/2022

INTERNATIONAL INSTITUTE OF INFORMATION TECHNOLOGY

HYDERABAD

# Pipelined - y86 Implementation

Team: 211

Aditya Sehgal 2020112013

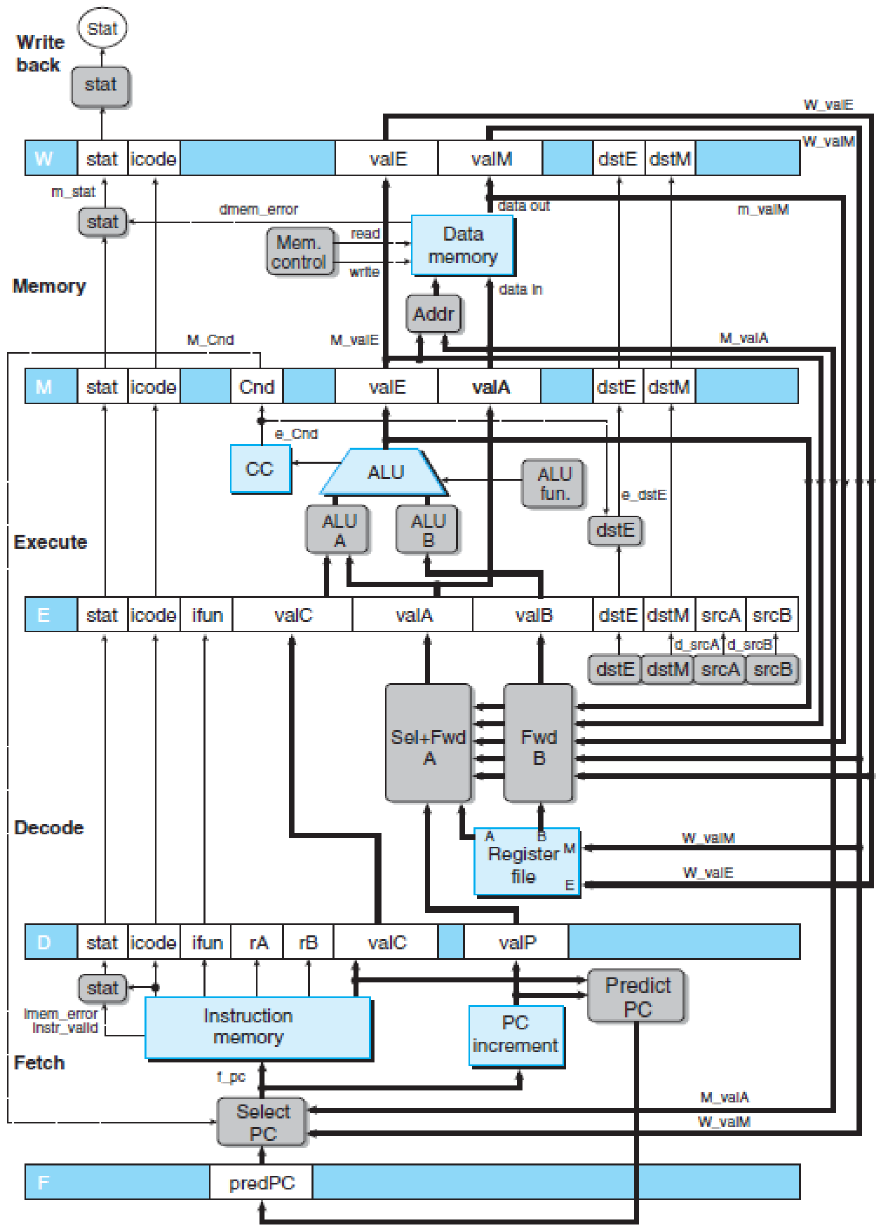
Rishabh Agrawal 2020102038

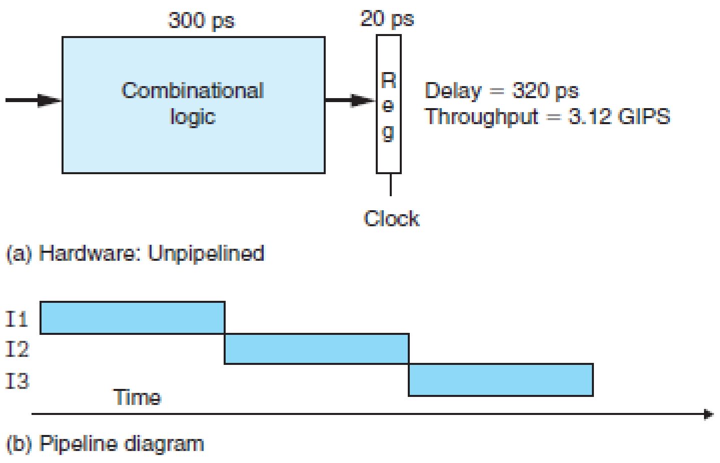
# Index

# Overview

The required specifications in the processor design are as follows:

A 5-stage pipelined y86 processor with data forwarding and control hazards.





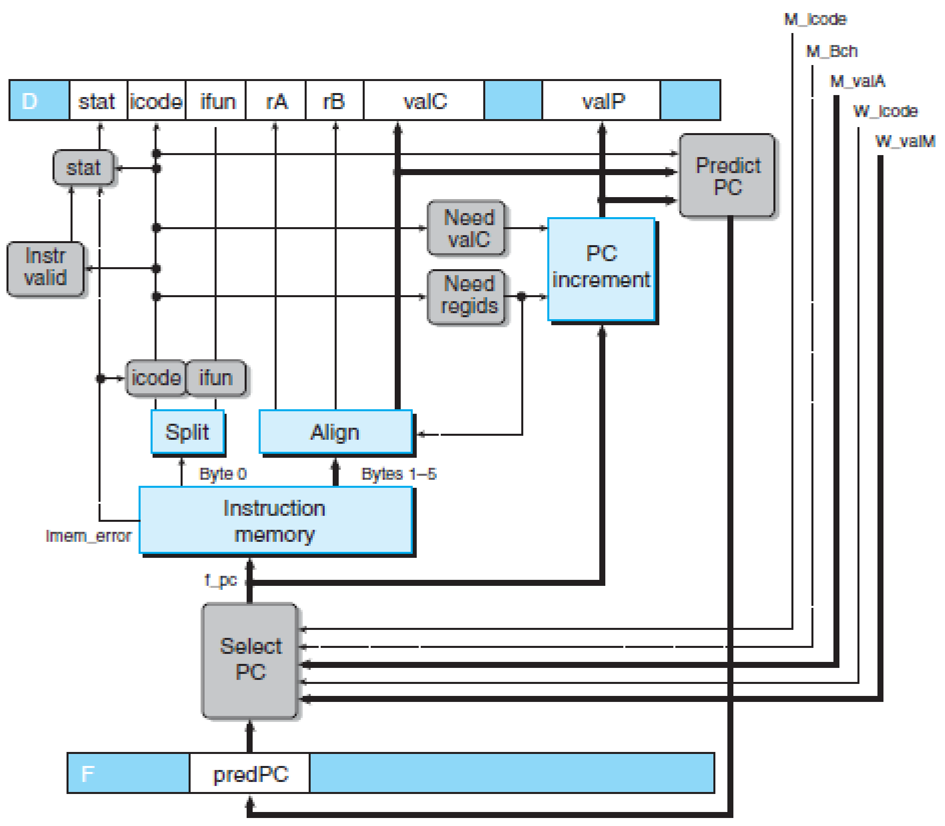
A key feature of pipelining is that it increases the throughput of the system. Above figure shows an example of a simple nonpipelined hardware system. It consists of some logic that performs a computation, followed by a register to hold the results of this computation.

## Implementing Registers:

```
//fetch register
f_reg r_f(.clk(clk),.upred_PC(upred_PC),.F_predPC(F_predPC));

//decode register
decode_reg r_d(.clk(clk),.f_stat(f_stat),.f_icode(f_icode),.f_ifun(f_ifun),.f_valC(f_valC),
             .f_valP(f_valP),.f_rA(f_rA),.f_rB(f_rB),.D_stat(D_stat),.D_icode(D_icode),
             .D_ifun(D_ifun),.D_valC(D_valC),.D_valP(D_valP),.D_rA(D_rA),.D_rB(D_rB));

execute_reg
r_e(.clk(clk),.d_stat(D_stat),.d_icode(d_icode),.d_ifun(d_ifun),.d_valC(d_valC),.d_valA(d_val
A),.d_valB(d_valB),.d_destE(d_destE),.d_destM(d_destM),.d_srcA(d_srcA),.d_srcB(d_srcB),.E_sta
t(E_stat),.E_icode(E_icode),.E_ifun(E_ifun),.E_valC(E_valC),.E_valA(E_valA),.E_valB(E_valB),.
E_destE(E_destE),.E_destM(E_destM),.E_srcA(E_srcA),.E_srcB(E_srcB));

memory_reg
r_m(.clk(clk),.e_stat(E_stat),.e_icode(e_icode),.e_cnd(e_cnd),.e_valE(e_valE),.e_valA(e_valA)
,.e_destE(e_destE),.e_destM(e_destM),.M_stat(M_stat),.M_icode(M_icode),.M_cnd(M_cnd),.M_valE(
M_valE),.M_valA(M_valA),.M_destE(M_destE),.M_destM(M_destM));

writeback_reg
r_w(.clk(clk),.m_stat(M_stat),.m_icode(m_icode),.m_valE(m_valE),.m_valM(m_valM),.m_destE(m_de
stE),.m_destM(M_destM),.W_stat(W_stat),.W_icode(W_icode),.W_valE(W_valE),.W_valM(W_valM),.W_d
estE(W_destE),.W_destM(W_destM));
```

# Modules

## Fetch



We have implemented it by taking input/output as:

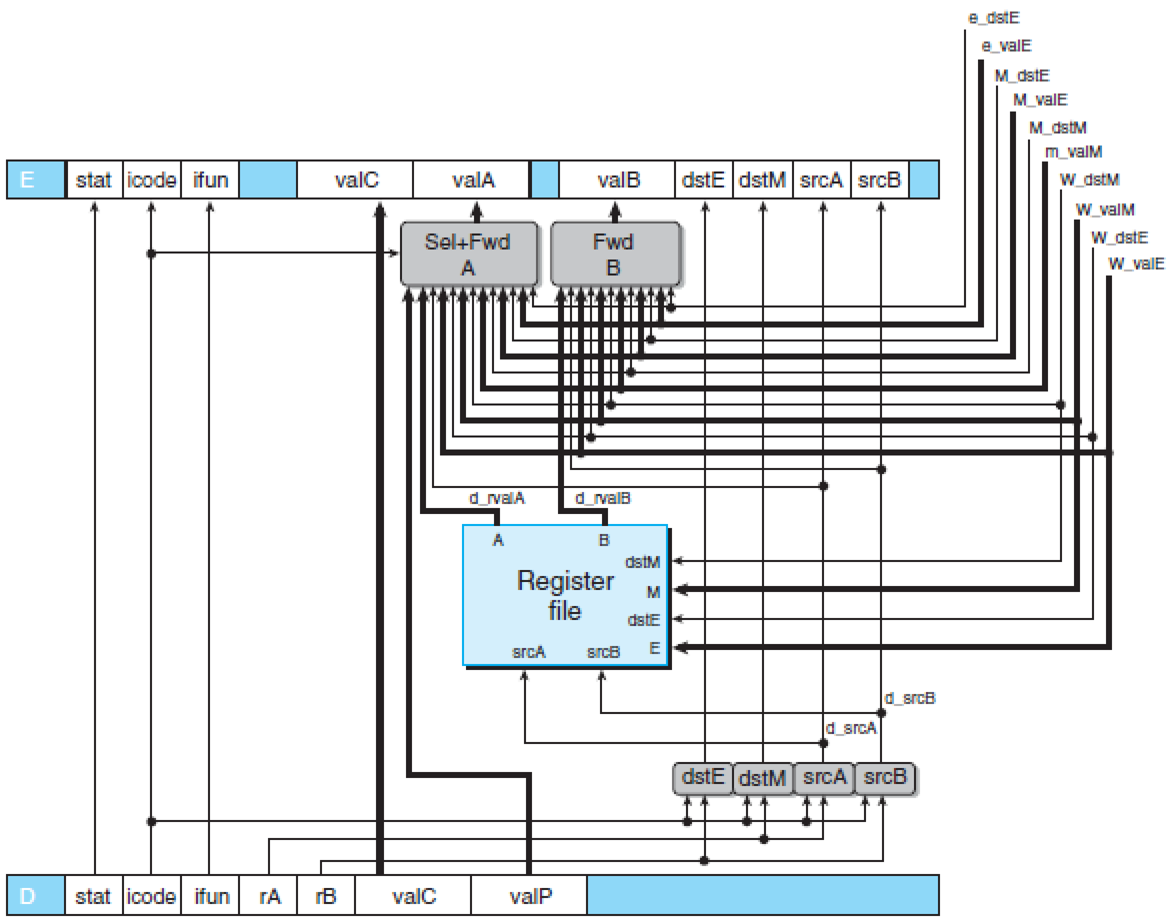
```
input clk;
input [64:1]W_valM;
input [64:1]M_valA;
input [4:1]M_icode;
input [4:1]W_icode;
input M_cond;
input [64:1]F_predPC;


output reg [4:1] icode;
output reg [4:1] ifun;
output reg [4:1] rA;
output reg [4:1] rB;
output reg [64:1] valC;
output reg [64:1] valP;
output reg [64:1]upred_PC;
reg inv_inst;
reg mem_error;
reg hlt_inst;
output reg [3:1]stat;
wire [64:1] f_pc;
wire [64:1]tupred_PC;
reg [8:1]instr_mem[100:1];
```

Fetch conditions are implemented using muxes with if statements comparing values of icode for respective functions as:

```
 else if(icode == 4'b0111) //jXX
    begin
          valC = {instr_mem[f_pc+8],
                instr_mem[f_pc+7],
                instr_mem[f_pc+6],
                instr_mem[f_pc+5],
                instr_mem[f_pc+4],
                instr_mem[f_pc+3],
                instr_mem[f_pc+2],
                instr_mem[f_pc+1]
                };
        valP = f_pc + 64'd9;
    end
```

Since it’s a combinational circuit, always block (values updating) is independent of the rising or falling of clock.

# Decode



We have implemented it by taking input/output as:

```
    input clk,
    input cnd,
    input [4:1]icode,
    input [4:1]ifun,
    input [4:1] rA,
    input [4:1] rB,
    input [64:1]D_valP,
    input [64:1]D_valC,
    input [4:1] e_destE,
    input [64:1] e_valE,
    input [4:1] M_destE,
    input [64:1] M_valE,
    input [4:1] M_destM,
    input [64:1] m_valM,
    input [4:1] W_destM,
    input [64:1] W_valM,
    input [4:1] W_destE,
    input [64:1] W_valE,
    input [4:1] W_icode,
    output reg [4:1] d_icode,
    output reg [4:1] d_ifun,
    output reg [64:1] valA,
    output reg [64:1] valB,
    output reg [64:1]d_valC,
    output reg [3:0]d_destE,
    output reg [3:0]d_destM,

    output reg [64:1] register_mem1,
    output reg [64:1] register_mem2,
    output reg [64:1] register_mem3,
    output reg [64:1] register_mem4,
    output reg [64:1] register_mem5,
    output reg [64:1] register_mem6,
    output reg [64:1] register_mem7,
    output reg [64:1] register_mem8,
    output reg [64:1] register_mem9,
    output reg [64:1] register_mem10,
    output reg [64:1] register_mem11,
    output reg [64:1] register_mem12,
    output reg [64:1] register_mem13,
    output reg [64:1] register_mem14,
    output reg [64:1] register_mem15
```

Here, all the registers are sent independently as we can’t send a 2-D array through a function call in Verilog.

## *Decode Block:*

**Decode conditions** are implemented using muxes with if statements comparing values of icode for respective functions as:

```
else if(icode == 4'b0110) //Op
    begin
        rvalA = register_mem[rA];
        rvalB = register_mem[rB];
        srcA = rA;
        srcB = rB;
    end
    else if(icode == 4'b1000) //call
    begin
        rvalB = register_mem[5]; //rsp register
        srcB = 5;
    end
```

Since it’s a combinational circuit (only reading of values is occurring), always block (values updating) is independent of the rising or falling of clock.

## *Sel+Fwd A and Fwd B Block:*

Sel+Fwd A block helps in detecting values from further stages which are not yet executed and directly fetch it to implement it in decode stage:

```
//Sel+Fwd A
always @(*)
begin
    if(icode == 4'b1000 || icode == 4'b0111)
    begin
        valA = D_valP;
    end
    else if (srcA == e_destE)
    begin
        valA = e_valE;
    end
    else if (srcA == M_destM)
    begin
        valA = m_valM;
    end
    else if (srcA == M_destE)
    begin
        valA = M_valE;
    end
    else if (srcA == W_destM)
    begin
        valA = W_valM;
    end
    else if (srcA == W_destE)
    begin
        valA = W_valE;
    end
    else
    begin
        valA = rvalA;
    end
end
```

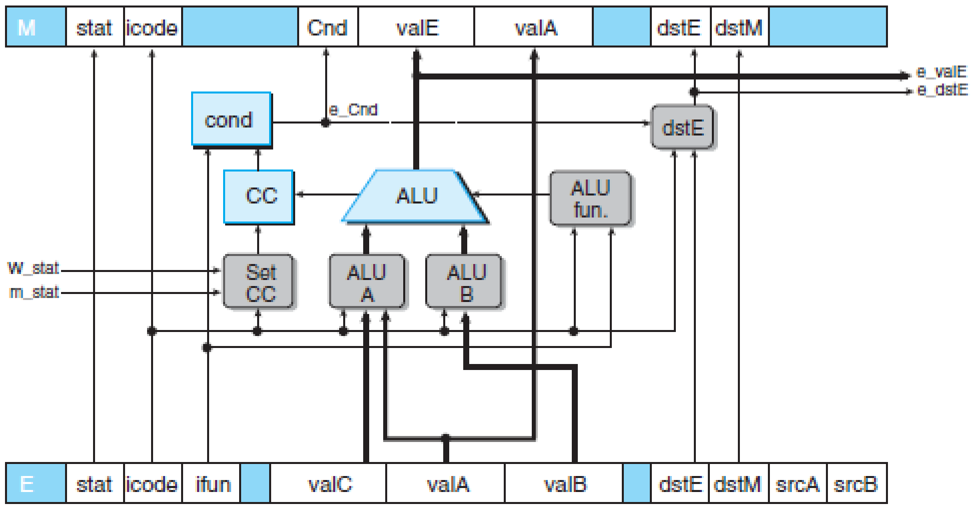
### *Write Back Block:*

**Write Back** is a sequential block, it gets updated on very positive edge of the clock:

```
always @(negedge clk)
begin
    register_mem[W_destE] <= W_valE;
    register_mem[W_destM] = W_valM;
end
```

Since both depends on register memory, they are written in a single file.

## Execute



We have implemented it by taking input/output as:

```
input clk;
input  [4:1]icode;
input  [4:1]ifun;
input  [64:1]valA;
input [64:1]valB;
input [64:1]valC;
input [4:1] destE;
output reg [4:1] e_destE; //might be of 4 bit
output reg [64:1] valE;
output reg cnd;
output reg [4:1] e_icode;
output reg [4:1] e_ifun;

reg [2:1]sel;
reg [64:1]a;
reg [64:1]b;
wire [64:1]result;
wire overflow;
reg zf,of,sf;
reg cin;

alu dut(.sel(sel),.a(a),.b(b),.cin(cin),.result(result),.overflow(overflow));
```

Here, ALU is been called for all Operations carried out through any function. Also, we can see that there’s not much change in the structure of execute.

## *Setting Condition Code:*

Whenever the icode is equal to Operation function, condition codes zf, sf and of are generated. Since they are sequential blocks, they get updated on positive edge of clock:

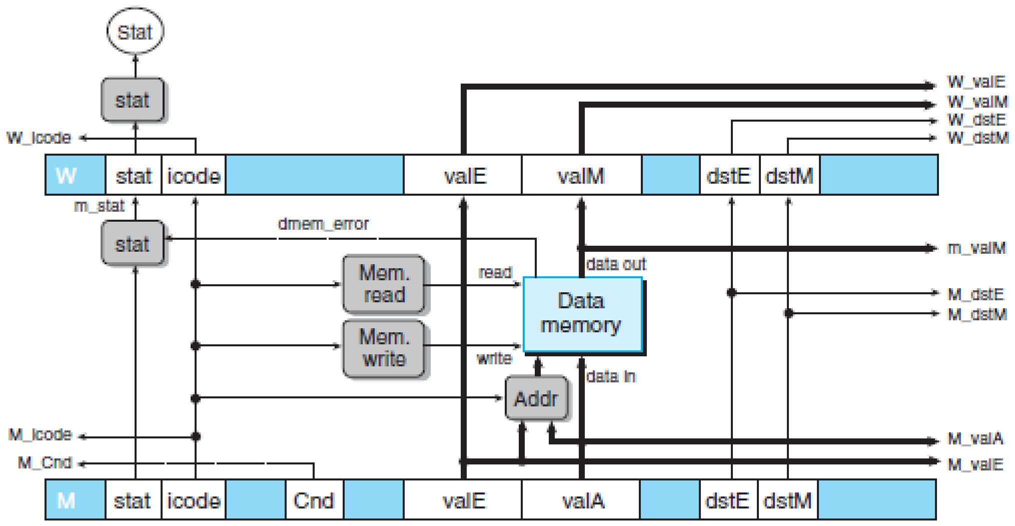
```
always @(posedge clk)
begin
    if(icode == 4'b0110 || clk==1)  //OP
    begin
        if(result == 64'b0)
            begin
                zf = 1;
            end
        else
            begin
                zf = 0;
            end
        if(result[64] == 1 )
            begin
                sf = 1;
            end
        else
            begin
                sf = 0;
            end
        if((a<64'b0 == b<64'b0) && (result<64'b0 != a<64'b0))
            begin
                of = 1;
            end
        else
            begin
                of = 0;
            end
    end
end
```

## *Setting values and Cnd for other Functions:*

Values and Cnd (for cmovxx and jxx) are set for all the functions as:

```
if(icode == 4'b0110) //OP
    begin
        b = valA;
        a = valB;
      if(ifun == 4'b0000) //ADD
        begin
            sel = 2'b00;
        end
      if(ifun == 4'b0001) //SUB
        begin
            sel = 2'b01;
        end
      if(ifun == 4'b0100) //AND
        begin
            sel = 2'b10;
        end
      if(ifun == 4'b0011) //XOR
      begin
            sel = 2'b11;
      end
      valE = result; //set_CC

    end

    // var assign problem
    else if(icode == 4'b0010) //cmovXX
    begin
        if(ifun == 4'b0000) //rrmovq
        begin
           cnd=1'b1;
        end

        if(ifun == 4'b0001) //cmovle
        begin
            if((sf^of)|zf)
            begin
                cnd=1'b1;
            end
        end

        if(ifun == 4'b0010) //cmovl
        begin
            if(sf^of)
            begin
                cnd=1'b1;
            end
        end

        if(ifun == 4'b0011) //cmove
        begin
            if(zf)
            begin
                cnd=1'b1;
            end
        end

        if(ifun == 4'b0100) //cmone
        begin
            if(~zf)
            begin
                cnd=1'b1;
            end
        end

        if(ifun == 4'b0101) //cmovge
        begin
            if(~(sf^of))
            begin
                cnd=1'b1;
            end
        end

        if(ifun == 4'b0110) //cmovg
        begin
            if((~(sf^of)&(~zf)))  //note here
            begin
                cnd=1'b1;
            end
        end

        a = valA;
        b = 0;
        sel = 2'b00;    //add
        valE = result;

    end

    else if (icode == 4'b0111) //jxx
    begin
        if(ifun == 4'b0000) //jmp
        begin
            cnd=1'b1;
        end

        if(ifun == 4'b0001) //jle
        begin
            if((sf^of)|zf)
            begin
                cnd=1'b1;
            end
        end

        if(ifun == 4'b0010) //jl
        begin
            if(sf^of)
            begin
                cnd=1'b1;
            end
        end
        if(ifun == 4'b0011) //je
          begin
            if(zf)
            begin
                cnd=1'b1;
            end
        end
        if(ifun == 4'b0100) //jne
        begin
            if(~zf)
            begin
                cnd=1'b1;
            end
        end
        if(ifun == 4'b0101) //jge
        begin
            if(~(sf^of))
            begin
                cnd=1'b1;
            end
        end
        if(ifun == 4'b0110) //jg
        begin
            if((~(sf^of)&(~zf)))
            begin
                cnd=1'b1;
            end
        end

    end

    else if(icode == 4'b0100) //rmmovq
    begin
        a = valB;
        b = valC;
        sel = 2'b00;
        valE = result;
    end

    else if(icode == 4'b0101) //mrmovq
    begin
        a = valB;
        b = valC;
        sel = 2'b00;
        valE = result;
    end

    else if(icode == 4'b0011) //irmovq
    begin
        a = valC;
        b = 64'b0;
        sel = 2'b00;
        valE = result;
    end

    else if(icode == 4'b1011 || icode == 4'b1001) //popq || ret
    begin
        a = valB;
        b = 64'b1000;
        sel = 2'b00;
        valE = result;
    end

    else if(icode == 4'b1010 || icode == 4'b1000) //pushq || call
    begin
        a = valB;
        b = 64'b1000;
        sel = 2'b01;
        valE = result;
    end
```

## Memory



We have implemented it by taking input/output as:

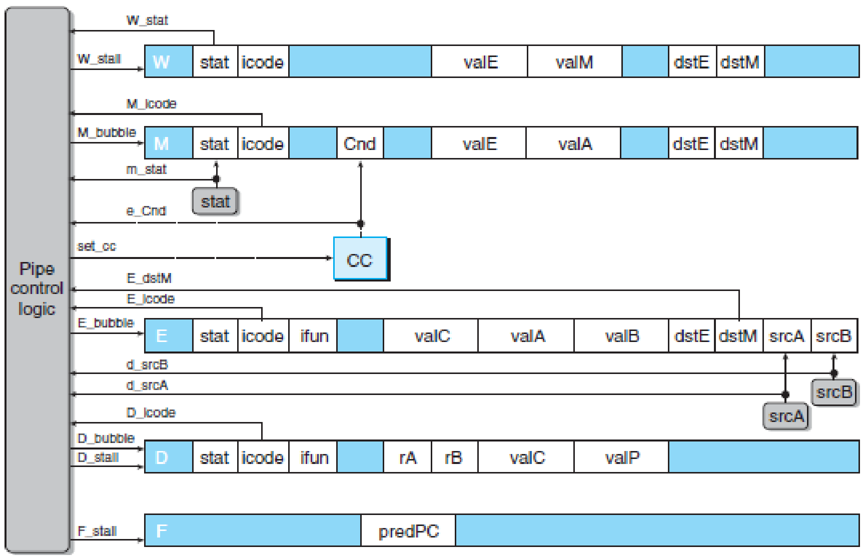
```
input clk;
input [4:1]icode;
input [64:1] valA;
input [64:1] valE;
input [64:1] valP;
input [4:1] M_dstE;

output reg [64:1] valM;
output reg [64:1]dram_val;
output reg [64:1] m_valE;
output reg [4:1] m_icode;
output reg [4:1] m_dstE;

reg [64:1] ram[1:1024];
```

Values are either taken or written back to memory or register using either valE or valM depending on the location. For eg:

```
if(icode == 4'b0100) //rmmovq
    begin
        ram[valE] = valA;
        dram_val = ram[valE];
    end
    else if(icode == 4'b0101) //mrmovq
    begin
        valM = ram[valE];
        dram_val = valM;
    end
```

# Control Logic

## Overview:



The pipeline control logic allows us to hold back an instruction in a pipeline register or to inject a bubble into the pipeline.

| | Pipeline register | | | | |
|---|---|---|---|---|---|
| Condition | F | D | E | M | W |
| Processing `ret` | stall | bubble | normal | normal | normal |
| Load/use hazard | stall | stall | bubble | normal | normal |
| Mispredicted branch | normal | bubble | bubble | normal | normal |

Based on signals from the pipeline registers and pipeline stages, the control logic generates stall and bubble control signals for the pipeline registers, and also determines whether the condition code registers should be updated. We can combine the detection conditions.

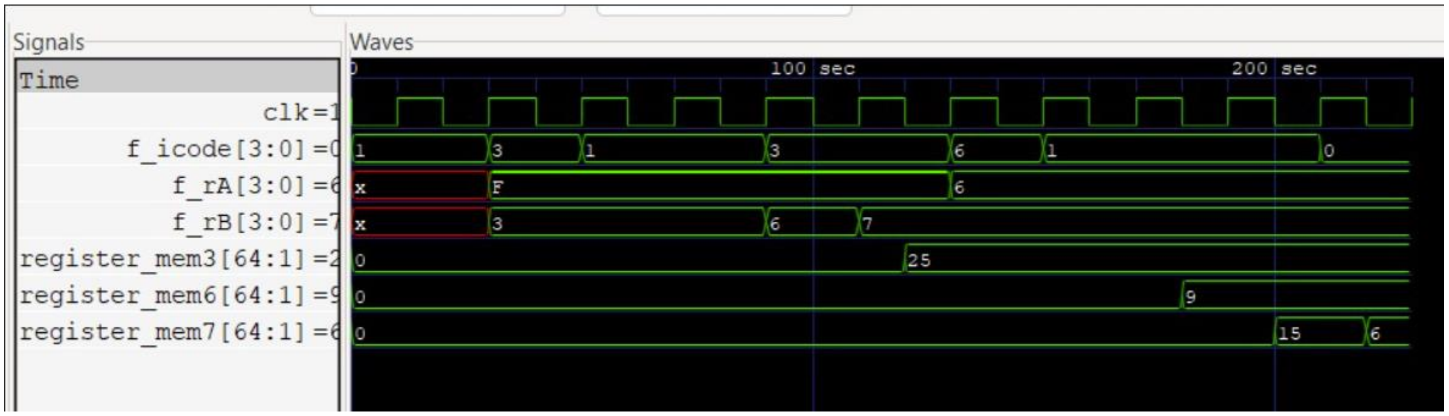
# GTKWave Results:

## Code 1:

Here, we are implementing 2 irmovq(s) and 1 subq without any use of nop operation. Hence, checking working of **data forwarding.**

```
//irmove
    instr_mem[30]=8'b00010000;
    instr_mem[31]=8'b00010000;
    instr_mem[32]=8'b00110000;
    instr_mem[33]=8'b11110011;  //5 to 6
    instr_mem[34]=8'b00011001;
    instr_mem[35]=8'b00000000;
    instr_mem[36]=8'b00000000;
    instr_mem[37]=8'b00000000;
    instr_mem[38]=8'b00000000;
    instr_mem[39]=8'b00000000;
    instr_mem[40]=8'b00000000;
    instr_mem[41]=8'b00000000;
    instr_mem[42]=8'b00010000;
    instr_mem[43]=8'b00010000;
//irmove
    instr_mem[44]=8'b00110000;
    instr_mem[45]=8'b11110110;  //5 to 6
    instr_mem[46]=8'b00001001;
    instr_mem[47]=8'b00000000;
    instr_mem[48]=8'b00000000;
    instr_mem[49]=8'b00000000;
    instr_mem[50]=8'b00000000;
    instr_mem[51]=8'b00000000;
    instr_mem[52]=8'b00000000;
    instr_mem[53]=8'b00000000;
//irmove
    instr_mem[54]=8'b00110000;
    instr_mem[55]=8'b11110111;  //5 to 6
    instr_mem[56]=8'b00001111;
    instr_mem[57]=8'b00000000;
    instr_mem[58]=8'b00000000;
    instr_mem[59]=8'b00000000;
    instr_mem[60]=8'b00000000;
    instr_mem[61]=8'b00000000;
    instr_mem[62]=8'b00000000;
    instr_mem[63]=8'b00000000;
//subq
    instr_mem[64]=8'b01100001;
    instr_mem[65]=8'b01100111;
    instr_mem[66]=8'b00010000;
    instr_mem[67]=8'b00010000;
    instr_mem[68]=8'b00010000;
    instr_mem[69]=8'b00000000;
```

## *GTKWave:*

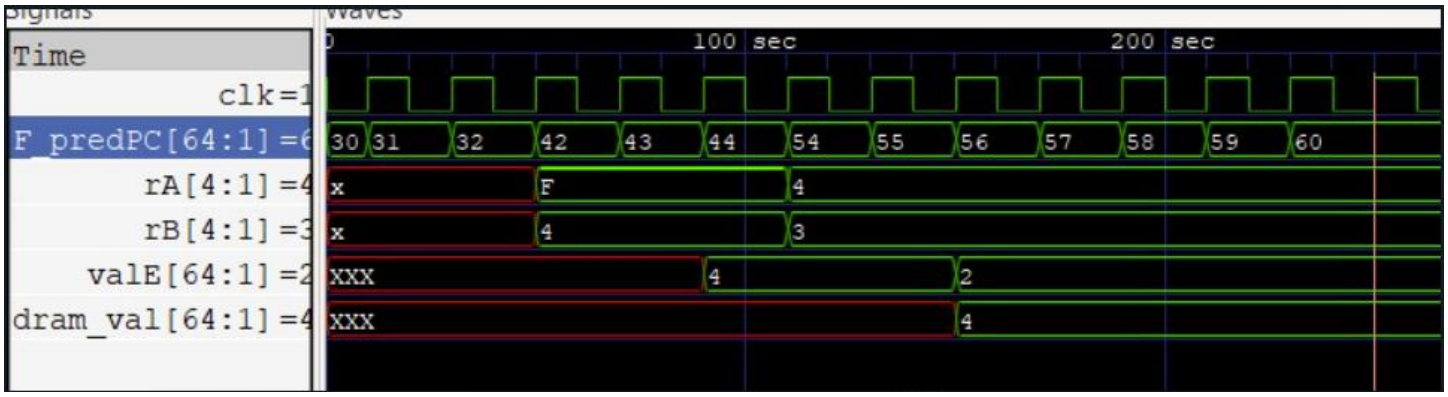


## Code 2:

Here, we are implementing first irmovq then rmmovq. Hence, checking working of memory module.

```
//irmove
    instr_mem[30]=8'b00010000;
    instr_mem[31]=8'b00010000;
    instr_mem[32]=8'b00110000;
    instr_mem[33]=8'b11110100;
    instr_mem[34]=8'b00000100;
    instr_mem[35]=8'b00000000;
    instr_mem[36]=8'b00000000;
    instr_mem[37]=8'b00000000;
    instr_mem[38]=8'b00000000;
    instr_mem[39]=8'b00000000;
    instr_mem[40]=8'b00000000;
    instr_mem[41]=8'b00000000;

    instr_mem[42]=8'b00010000;
    instr_mem[43]=8'b00010000;
//rmmove
    instr_mem[44]=8'b01000000;
    instr_mem[45]=8'b01000011;
    instr_mem[46]=8'b00000010;
    instr_mem[47]=8'b00000000;
    instr_mem[48]=8'b00000000;
    instr_mem[49]=8'b00000000;
    instr_mem[50]=8'b00000000;
    instr_mem[51]=8'b00000000;
    instr_mem[52]=8'b00000000;
    instr_mem[53]=8'b00000000;

    instr_mem[54]=8'b00010000;
    instr_mem[55]=8'b00010000;
    instr_mem[56]=8'b00010000;
    instr_mem[57]=8'b00010000;
    instr_mem[58]=8'b00010000;
    instr_mem[59]=8'b00010000;
    instr_mem[60]=8'b00000000;
```

*GTKWave:*

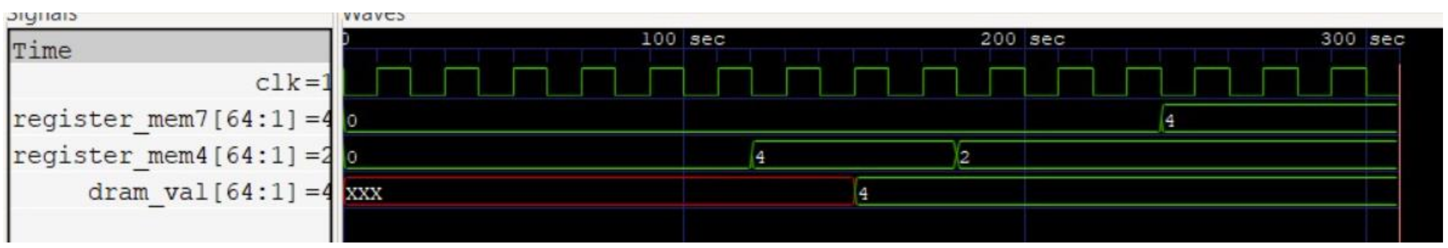


## Code 3:

Here, we are implementing irmovq, rmmovq and a mrmovq to check the working of load/write hazard and working of memory block.

```
//irmove
    instr_mem[30]=8'b00010000;
    instr_mem[31]=8'b00010000;
    instr_mem[32]=8'b00110000;
    instr_mem[33]=8'b11110100;
    instr_mem[34]=8'b00000100;
    instr_mem[35]=8'b00000000;
    instr_mem[36]=8'b00000000;
    instr_mem[37]=8'b00000000;
    instr_mem[38]=8'b00000000;
    instr_mem[39]=8'b00000000;
    instr_mem[40]=8'b00000000;
    instr_mem[41]=8'b00000000;

    instr_mem[42]=8'b00010000;
    instr_mem[43]=8'b00010000;
//rmmove
    instr_mem[44]=8'b01000000;
    instr_mem[45]=8'b01000011;
    instr_mem[46]=8'b00000010;
    instr_mem[47]=8'b00000000;
    instr_mem[48]=8'b00000000;
    instr_mem[49]=8'b00000000;
    instr_mem[50]=8'b00000000;
    instr_mem[51]=8'b00000000;
    instr_mem[52]=8'b00000000;
    instr_mem[53]=8'b00000000;
    instr_mem[54]=8'b00010000;
    instr_mem[55]=8'b00010000;
//mrmove
    instr_mem[56]=8'b01010000;
    instr_mem[57]=8'b01110011;
    instr_mem[58]=8'b00000010;
    instr_mem[59]=8'b00000000;
    instr_mem[60]=8'b00000000;
    instr_mem[61]=8'b00000000;
    instr_mem[62]=8'b00000000;
    instr_mem[63]=8'b00000000;
    instr_mem[64]=8'b00000000;
    instr_mem[65]=8'b00000000;

    instr_mem[66]=8'b00010000;
    instr_mem[67]=8'b00010000;
    instr_mem[68]=8'b00010000;
    instr_mem[69]=8'b00010000;
    instr_mem[70]=8'b00010000;
    instr_mem[71]=8'b00000000;
```

## *GTKWave:*

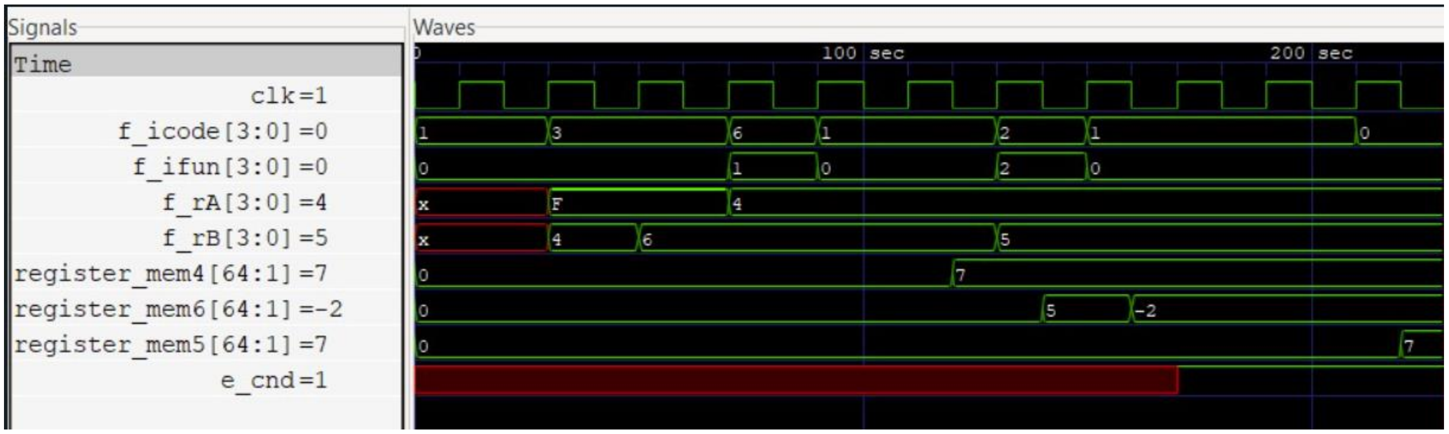


## Code 4:

Here, we are implementing two irmovq, subq and a cmovxx to check the working of condition codes and set_cc.

```
    instr_mem[30]=8'b00010000;
    instr_mem[31]=8'b00010000;
//irmove
    instr_mem[32]=8'b00110000;
    instr_mem[33]=8'b11110100; //%7 to 4
    instr_mem[34]=8'b00000111;
    instr_mem[35]=8'b00000000;
    instr_mem[36]=8'b00000000;
    instr_mem[37]=8'b00000000;
    instr_mem[38]=8'b00000000;
    instr_mem[39]=8'b00000000;
    instr_mem[40]=8'b00000000;
    instr_mem[41]=8'b00000000;

//irmove
    instr_mem[42]=8'b00110000;
    instr_mem[43]=8'b11110110;  //5 to 6
    instr_mem[44]=8'b00000101;
    instr_mem[45]=8'b00000000;
    instr_mem[46]=8'b00000000;
    instr_mem[47]=8'b00000000;
    instr_mem[48]=8'b00000000;
    instr_mem[49]=8'b00000000;
    instr_mem[50]=8'b00000000;
    instr_mem[51]=8'b00000000;

//subq
    instr_mem[52]=8'b01100001;
    instr_mem[53]=8'b01000110;


//cmovl
    instr_mem[54]=8'b00010000;
    instr_mem[55]=8'b00010000;
    instr_mem[56]=8'b00100010;
    instr_mem[57]=8'b01000101;
    instr_mem[58]=8'b00010000;

    instr_mem[59]=8'b00010000;
    instr_mem[60]=8'b00010000;
    instr_mem[61]=8'b00000000;
```

## *GTKWave:*


Signals
Waves
Time
clk=1
f_icode[3:0]=0
f_ifun[3:0]=0
f_rA[3:0]=4
f_rB[3:0]=5
register_mem4[64:1]=7
register_mem6[64:1]=-2
register_mem5[64:1]=7
e_cnd=1
100 sec
200 sec
1
3
6
1
2
1
0
0
1
0
2
0
x
F
4
x
4
6
5
0
7
0
5
-2
0
7

*******