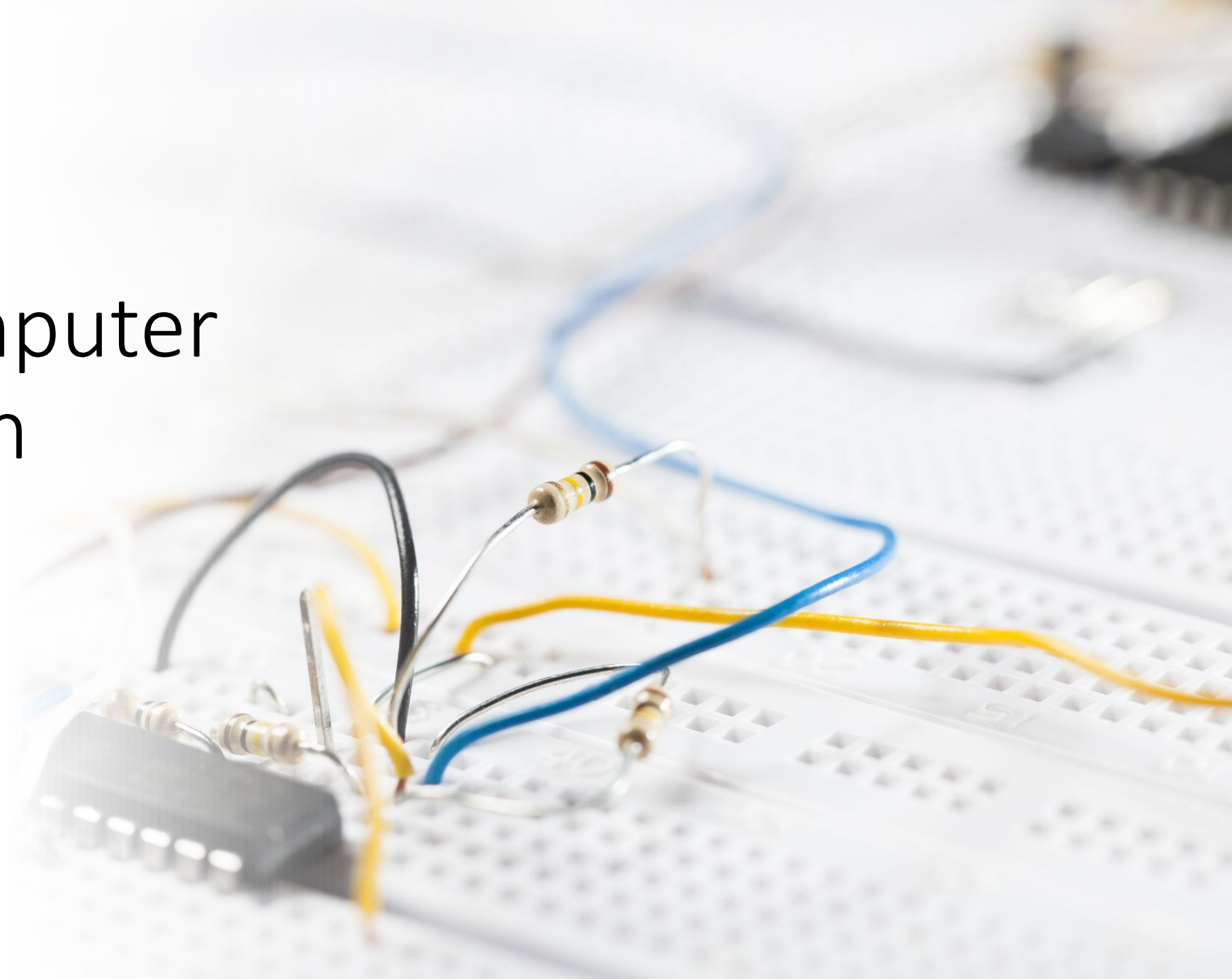
CS 201-Computer Organization

Narmit Kumar- B20218



Problem Statement

- To Design a processor which contains the 8-bit registers to perform the required operation.
- Data can be loaded through 8-bit registers such as R0,R1….R7 and A
- Various Operations like Addition /Subtraction can be performed using the multiplexer with different operations in each clock cycle
- Each instruction can be encoded and stored in the IR register using the 9-bit format IIIXXXYYY, where III represents the instruction, XXX gives the RX register, and YYY gives the RY register

What is Pipelining?

A mechanism for overlapped execution of several input sets by partitioning some computation into a set of k sub-computations (or stages).

- Very nominal increase in the cost of implementation.
- Very significant speedup (ideally, k).

By associating a register with every segment in the pipeline, the process of computation can be made overlapping. The registers provide separation and isolation among every segment, allowing each to work on different data at the same time.

Where is pipelining used in a computer system?

Instruction execution: Several instructions executed in some sequence.

Arithmetic computation: Same operation carried out on several data sets.

Memory access: Several memory accesses to consecutive locations are made.

Advantages and Disadvantages of Pipelining

Pros:

- Multiple instructions are being processed at same time.
- This works because stages are isolated by registers.
- Best case speedup of N

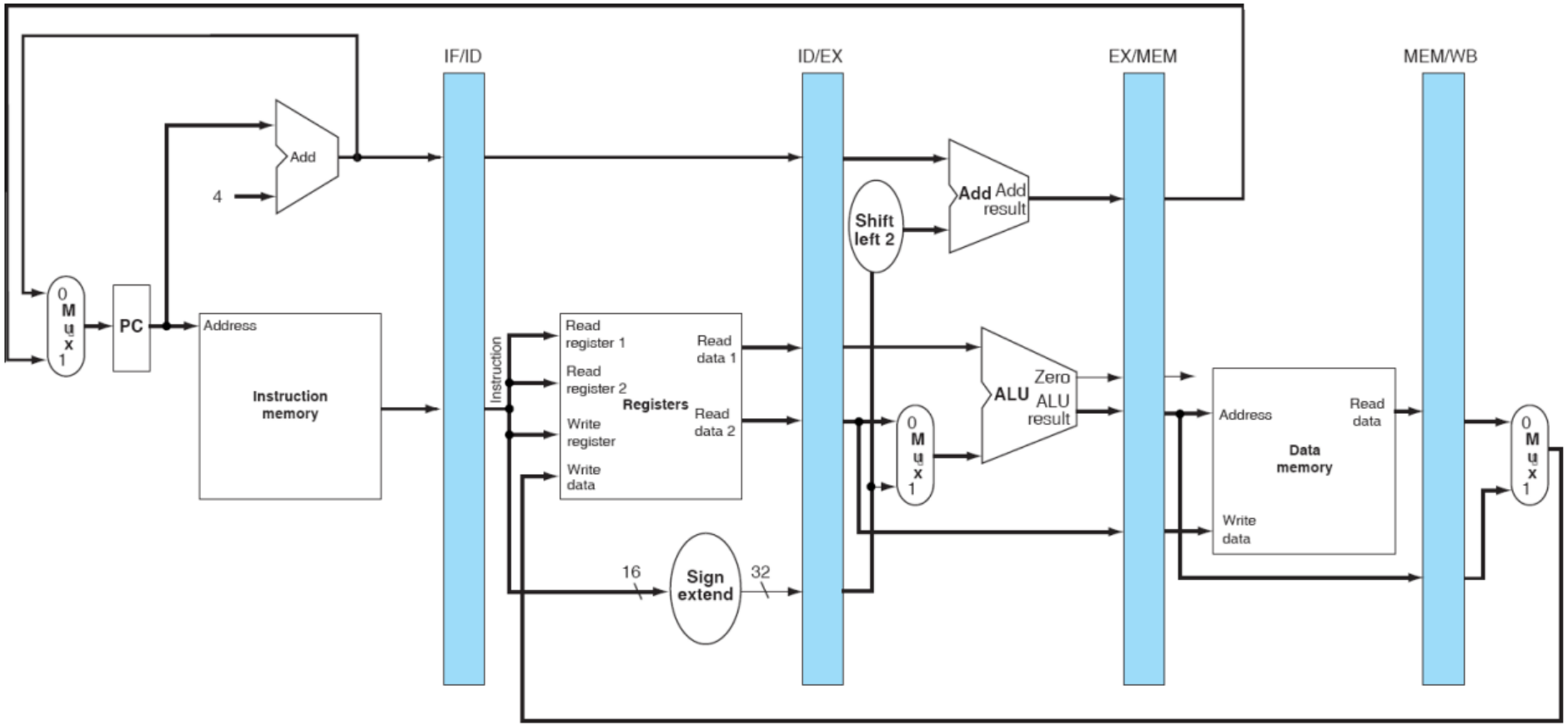
Cons:

- Interference of instructions with each other known as pipelining hazards.
- These hazards prevent next instruction from executing during its designated clock cycle

5 steps in instruction execution in Pipeline

A. Instruction Fetch (IF)
B. Instruction Decode and Register Read (ID)
C. Execution operation or calculate address (EX)
D. Memory access (MEM)
E. Write result into register (WB)

Basic Pipelined Processor



Instructions considered with opcode

Instruction	Function	Opcode
mv Rx, Ry	Rx ← [Ry]	000
mvi Rx, #D	Rx ← D	001
add Rx, Ry	Rx ← [Rx] + [Ry]	010
sub Rx, Ry	Rx ← [Rx] – [Ry]	011
and Rx, Ry	Rx ← [Rx] & [Ry]	100
or Rx, Ry	Rx ← [Rx] | [Ry]	101
xor Rx,Ry	Rx ← [Rx] ^ [Ry]	110
not Rx, Ry	Rx ← ~[Rx]	111

Clock Cycles:

Instruction	1	2	3	4	5	6	7	8
i	IF	ID	EX	MEM	WB			
i + 1		IF	ID	EX	MEM	WB		
i + 2			IF	ID	EX	MEM	WB	
i + 3				IF	ID	EX	MEM	WB

Verilog Code

```
`timescale 1ns / 1ps
module Processor(clk1,clk2);
input clk1, clk2;
reg [8:0] PC, IF_ID_IR, IF_ID_NPC;
reg [8:0] ID_EX_IR, ID_EX_NPC, ID_EX_A, ID_EX_B, ID_EX_Imm;
reg [1:0] ID_EX_type, EX_MEM_type, MEM_WB_type;
reg [8:0] EX_MEM_IR, EX_MEM_ALUOut, EX_MEM_B;
reg [8:0] MEM_WB_IR, MEM_WB_ALUOut;

reg [8:0] Reg [0:11];   //12 Registers storing 9 bit data
reg [8:0] Mem [0:1023]; //1024 x 9 bit memory

parameter MV = 3'b000,MVI = 3'b001 ,ADD = 3'b010,SUB = 3'b011,
          AND = 3'b100, OR = 3'b101, XOR = 3'b110,NOT=3'b111;

parameter RR_ALU=1'b00, RM_ALU=1'b01;
```

Instruction Fetch Stage

```
//Instruction Fetch stage
always @(posedge clk1)
        begin
                IF_ID_IR  <= #2 Mem[PC];                    //picking instruction from memory
                IF_ID_NPC <= #2 PC+1;                       //updating new program counter
                PC        <= #2 PC+1;                       //updating program counter
        end
```

Instruction Decode Stage

```
//Instruction Decode Stage
always @(posedge clk2)

    begin
        if (IF_ID_IR[5:3] ==5'b000)                          //updating register A
        ID_EX_A <=0;
        else
        ID_EX_A <= #2 Reg[IF_ID_IR[5:3]];  //rx

        if (IF_ID_IR[2:0] ==5'b000)                          //updating register B
        ID_EX_B <=0;
        else
        ID_EX_B <= #2 Reg[IF_ID_IR[2:0]];  //ry

         ID_EX_NPC  <= #2 IF_ID_NPC;                             //forwarding NPC to execution stage
         ID_EX_IR   <= #2 IF_ID_IR;                              //forwarding instruction to execution stage
         ID_EX_Imm  <= #2 {{6{IF_ID_IR[2]}}, {IF_ID_IR[2:0]}};        //sign extending immediate value to 9bits

        case (IF_ID_IR[8:6])                                  //declaring type of opcode

        MV,ADD,SUB,AND,OR,XOR,NOT:   ID_EX_type <= #2 RR_ALU;
        MVI :                        ID_EX_type <= #2 RM_ALU;

        endcase
    end
```

```
//Execution Stage
always @(posedge clk1)
    begin
        EX_MEM_type   <= #2 ID_EX_type;                    //forwarding type of opcode memory stage
        EX_MEM_IR     <= #2 ID_EX_IR;                      //forwarding instruction to memory stage

    case (ID_EX_type)                                      //executing function according to type
        RR_ALU: begin
            case (ID_EX_IR[8:6])
                MV:     begin
                          ID_EX_A <= #1 ID_EX_B ;
                          EX_MEM_ALUOut <= #1 ID_EX_A;
                        end
                ADD:      EX_MEM_ALUOut <= #2 ID_EX_A + ID_EX_B;
                SUB:      EX_MEM_ALUOut <= #2 ID_EX_A - ID_EX_B;
                AND:      EX_MEM_ALUOut <= #2 ID_EX_A & ID_EX_B;
                OR:       EX_MEM_ALUOut <= #2 ID_EX_A | ID_EX_B;
                XOR:      EX_MEM_ALUOut <= #2 ID_EX_A ^ ID_EX_B;

                default: EX_MEM_ALUOut <= #2 9'bxxxxxxxx;
            endcase
            end

        RM_ALU: begin
            case (ID_EX_IR[8:6])
                        MVI: begin
                             ID_EX_A       <= #1 ID_EX_Imm;
                             EX_MEM_ALUOut <= #1 ID_EX_A;
                             end
                        default: EX_MEM_ALUOut <= #2 9'bxxxxxxxx;
            endcase
            end
    endcase
    end
```

Execution Stage

Memory Stage

```
//Memory Stage
always @(posedge clk2)

    begin
        MEM_WB_type <= #2 EX_MEM_type;                    //forwarding type to write back stage
        MEM_WB_IR   <= #2 EX_MEM_IR;                      //forwarding instruction to write back stage

    case (EX_MEM_type)
        RR_ALU, RM_ALU: MEM_WB_ALUOut <= #2 EX_MEM_ALUOut;

    endcase
    end
```

Write Back Stage

```
//Write Back Stage
always @(posedge clk1)
    begin


    case ( MEM_WB_type)
        RR_ALU: Reg[MEM_WB_IR[5:3]] <= #2 MEM_WB_ALUOut;          //writing back in destination register(rd)

        RM_ALU: Reg[MEM_WB_IR[2:0]] <= #2 MEM_WB_ALUOut;          //writing back in destination register(rt)




    endcase
    end


endmodule
```

Adding R_3 and R_4: Rx ← [Rx]+[Ry]

Subtracting R_5 from R_6 : Rx ← [Rx] - [Ry]

```
Vivado Simulator 2017.1
Time resolution is 1 ps
R1:     1
R2:     2
R3:     3
R4:     4
R5:     5
R6:     6
R7:     7
R3:     7
R6:     1
relaunch_sim: Time (s): cpu = 00:00:01 ; elapsed = 00:00:07 . Memory (MB): peak = 869.543 ; gain = 0.000
run 1 s
R0:     0
R1:     1
R2:     2
R3:     7
R4:     4
R5:     5
R6:     1
R7:     7
R8:     8
R9:     9
R10:    10
R11:    11
```

Code for:
Not R_1: $R_1 \leftarrow \sim[R_1]$

XOR of R_3 with R_3 : $R_3 \leftarrow [R_3]\wedge[R_3]$

```
initial begin
for (k=0; k<8; k=k+1)
    mips.Reg[k] =k;                 //initialising all the register values with k

    mips.Mem[0] = 9'b111001000;     //NOT R1
    mips.Mem[1] = 9'b110011011;     //XOR R3 with R3









    mips.Reg[7] =10;              //initialsing memory data at 7 memory address
    mips.PC = 0;
```

Output for:
Not R_1: $R_1 \leftarrow \sim[R_1]$

XOR of R_3 with R_3 : $R_3 \leftarrow [R_3]\wedge[R_3]$

```
Time resolution is 1 ps
relaunch_sim: Time (s): cpu = 00:00:01 ; elapsed = 00:00:07 . Memory (MB): peak = 869.543 ; gain = 0.000
run 1 s
R0:    0
R1: x
R2:    2
R3:    0
R4:    4
R5:    5
R6:    6
R7:   10
R8: x
R9: x
R10: x
R11: x
$finish called at time : 4500 ns : File "C:/Users/Narmit/project_3/project_3.srcs/sim_1/new/testbench.v" Line 51
```

THANK YOU