PROJECT REPORT

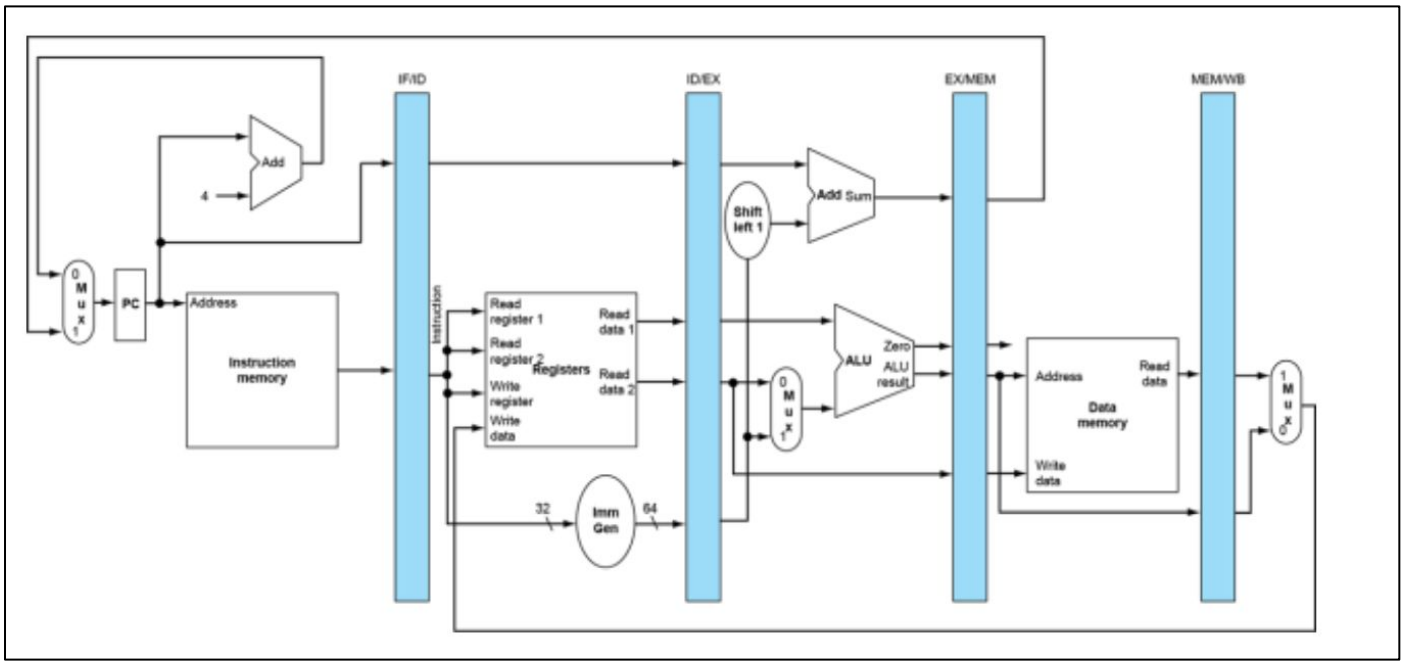
Title: RISC-V pipelined Data path

PAGES:

Designing Pipelined RISC-V Data path.

- **Objective:**
  The objective of this LAB is to design a 4 stage RISC-V pipelined data path.

- **Problem Statement:**
  Design a 5 stage RISC-V pipelined data path with the following features
  1. Store data and access data.
  2. Perform add, sub, mul, addi arithmetic operations.
  3. Apply forwarding logic to decrease number of stalls as much as possible thereby increasing performance.



RISC-V 5 stage pipeline.

You can store value 45 and -20 in the D-Cache initially,

- **Guiding Design principles in RISC-V ISA:**

1. Simplicity favours regularity.
2. Smaller is faster.
3. Good design demands good compromises.

- **Stages in RISC-V pipeline:**

Similar to MIPS architecture there are 5 stages in RISC-V ISA(Instruction Set Architecture) :

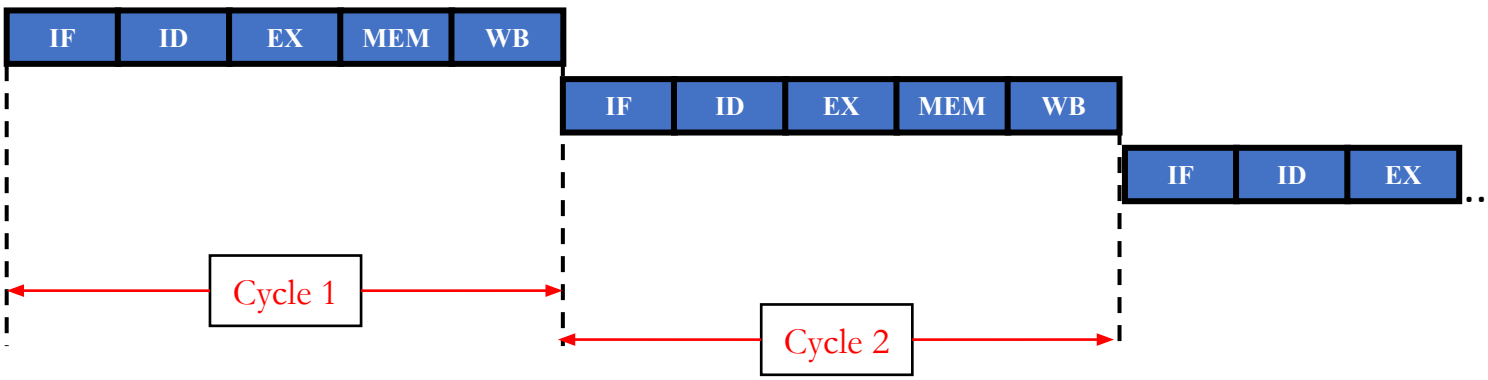
a) IF : Instruction fetch from Instruction cache.
b) ID : Instruction Decode and Register read.
c) EX: Execute Operations[eg: add, sub] (or) Calculate address[ld, sd].
d) MEM: Access memory operands from data cache.
e) WB: Write result back to the register in register file.

- **Why pipelining?:**

NOTE:
Execution time = IC * CPI * T
IC – Instruction count
CPI – Cycles Per Instruction
T – Clock Period.

In a single cycle processing unit the instructions are going to be executed sequentially one by one. Here CPI (Cycles Per Instruction) is low [for one instruction – one cycle] but here we are going to have a long clock period [the clock period corresponds to slowest instruction]. As a result the execution time is going to increase significantly due to long clock period, which results in less performance. This single cycle processing unit doesn't have any benefit except its simplicity.

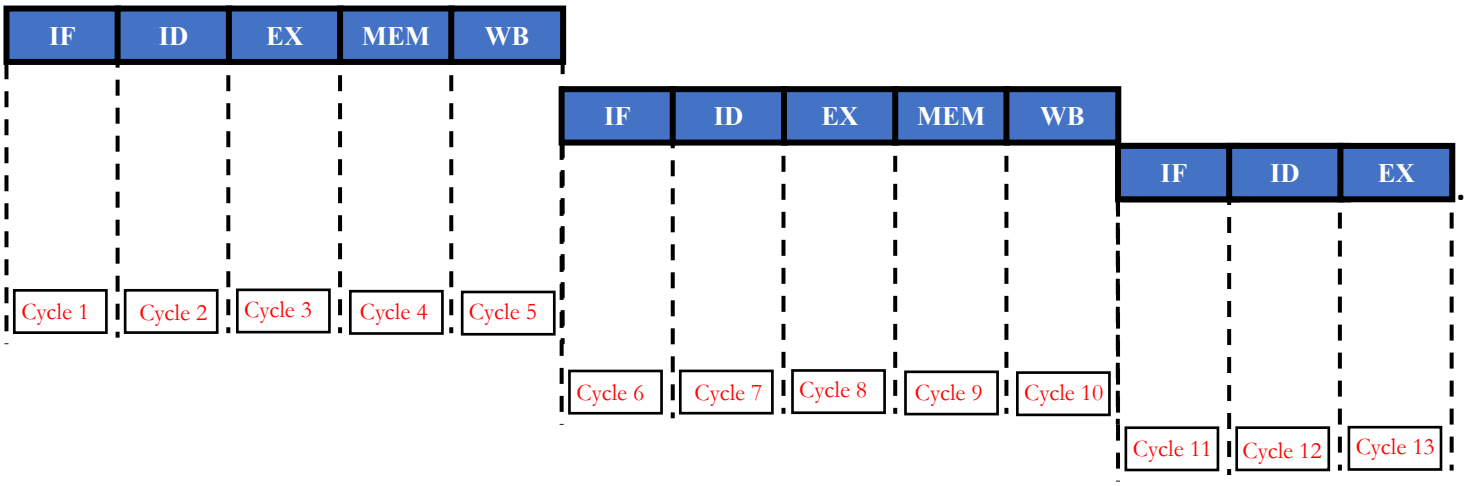
Picture title: Diagram representing single cycle processing unit.



CPI: 1
Clock period: IF_time + ID_time + EX_time + MEM_time + WB_time
(long clock period, critical path: ld instruction)

In a multi-cycle processing unit each stage is executed in one cycle without pipelining. This results high CPI[one instruction 5 stages – 5 cycles] and short clock period [clock period corresponding to the longest time taken among 5 stages]. As a result execution time is going to increase and due to which performance decreases. Here one important thing to note that at each cycle, all hardware except the hardware corresponding to the stage executed in that cycle remains idle (unused).

Figure : Multicycle processing unit without pipelining.



CPI : 5

Clock period: max{IF_time, ID_time, EX_time, MEM_time, WB_time}

- **Challenges in pipelining:**

1) Speed up issues when unbalanced.

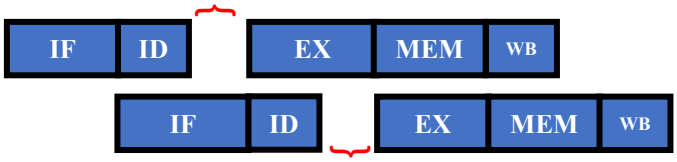
If the time taken by all stages is same (that is balanced) then we can say that

$$(time)_{pipelined} = \frac{(time)_{non-pipelined}}{\text{Number of stages}}$$

But in general the stages are not balanced, so we do not get the ideal speed up as written in the above formulae. The throughput will be less than expected. But the throughput of pipelined datapath will be far better than non-pipelined datapath.

Eg: if the time needed for ID stage is less than EX stage, when these both stages are executing in the same clock cycle, ID stage should wait for getting next computation until the EX stage gets completed.

Figure : ID , WB stages take less time(register operations) than other stages:



2) Hazards

a) Structural Hazards

Conflict for use of resource, IF and MEM stage requires memory or cache element but if there is only one cache then a requirement of bubble is happening, so this can be avoided by giving two different caches Instruction-cache and Data-cache.

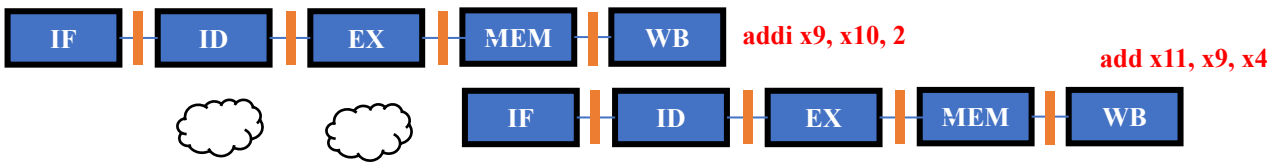
b) Data Hazards

An instruction depends on completion of previous instruction, this type of issue is known as **Data dependencies.**

Eg:

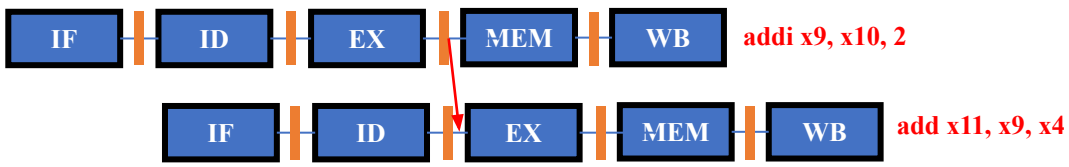
1) I1: addi x9, x10, 2
   I2: addi x11, x9, x4
   I2 depeneds on I1 because this is Write after read dependency

Figure : Bubbles needed (no forwarding) due to data dependency:



In the above case, WB is done in the first half of cycle and then ID.

Figure : Forwarding (aka bypassing) the above data dependency:

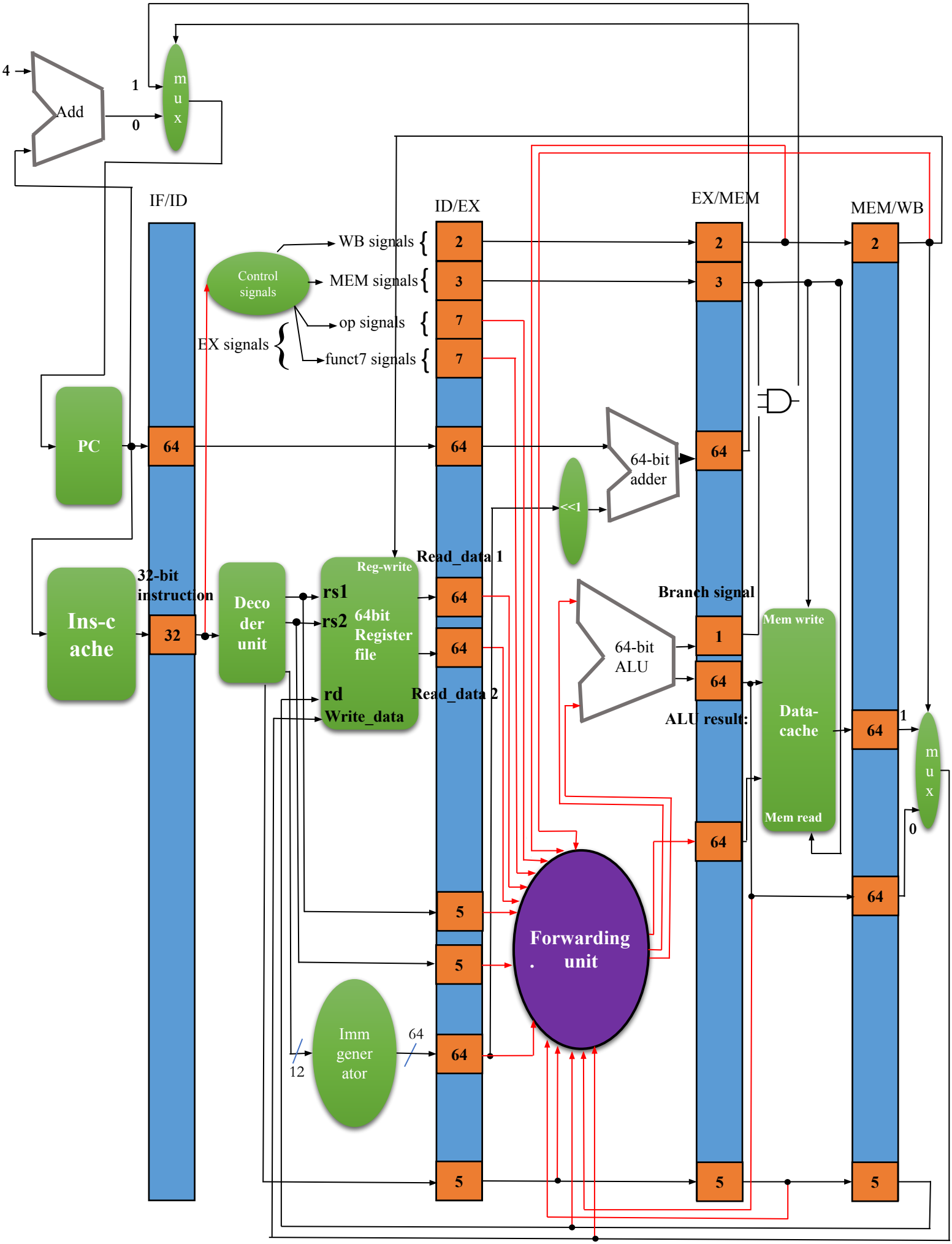


Not all data dependencies can be avoided by forwarding, if the instruction is ld instruction and next instruction is reading the register then there is a need of one bubble even when there is forwarding.

c) Control Hazards.

When a instruction is branch instruction the next instruction depends on the result of branch instruction. So there are many methods for overcoming this some of them are 1) Computing branch at the ID stage (this needs one bubble) 2) branch predictors (dynamic and static branch predictors). Handling control hazards is out of the scope of this project.

Figure : RISC-V pipelined data path



Note: Red wires depict the input and output of forwarding logic. The number in orange color boxes depict the number of bits.

- **Goal:**

1) Use hardware lying as much as possible.(running instructions in parallel though there is no actual parallel hardware)
2) Minimize the number of stalls
3) Thereby increasing the performance.
4) Retaining the simplicity of the hardware.

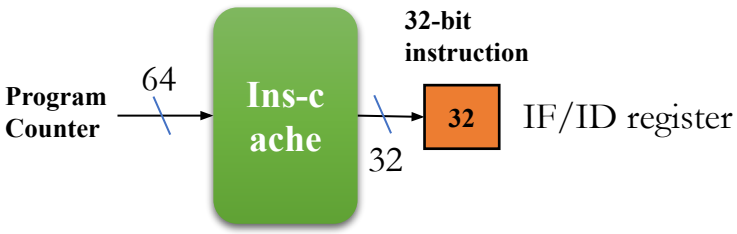
- **Features of this data path:**

Our main motive of this project is to come close as much as possible to the actual RISC-V pipelined data path. The following features are included in this data path:

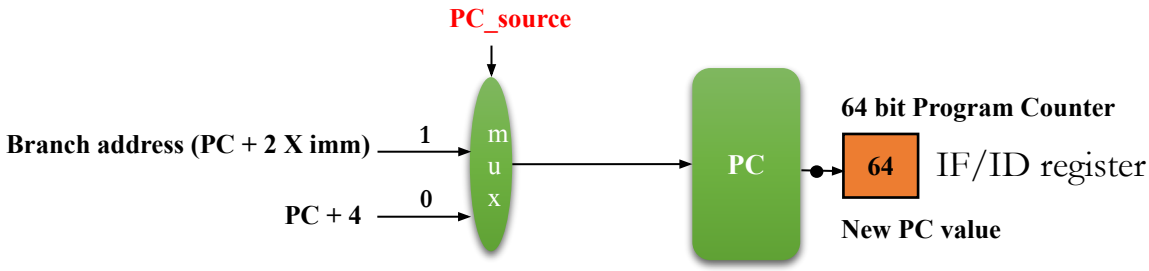
1. Pipelined data path: which allows multiple subtasks to be carried out simultaneously using independent resources.
2. Important components of Data path:
   a) 20 32-bit instruction cache. (can be increased based on requirements)
   b) Program Counter.
   c) Instruction decoder. (R-type, I-type, S-type)
   d) 32 64-bit register file.
   e) Immediate Generator.
   f) 64-bit ALU(addition, subtraction, multiplication)
   g) 20 64-bit Data cache. (can be increase based on requirements)
   h) Pipeline registers(IF/ID, ID/EX, EX/MEM, MEM/WB)
   i) Control Unit. (Generate control signals)
   j) Forwarding unit.

- **Description of Approach:**

Instruction cache stores instructions, when instruction is fetched from it using program counter.



Program counter is used for iterating through the instruction cache. Assuming the cache is byte addressable it is incremented by 4 to do to the next instruction and it is given PC + 2 X imm when there are branch instructions.



Role of decoder unit and register file in data path:

In this project the decoder unit decodes I, R, S type instructions of RISC-V. The register file has 32 64-bit registers where register read and register write gets executed.

32 64bit Registers in register file:

- x0: the constant value 0
- x1: return address
- x2: stack pointer
- x3: global pointer
- x4: thread pointer
- x5 – x7, x28 – x31: temporaries
- x8: frame pointer
- x9, x18 – x27: saved registers
- x10 – x11: function arguments/results
- x12 – x17: function arguments

R-format:

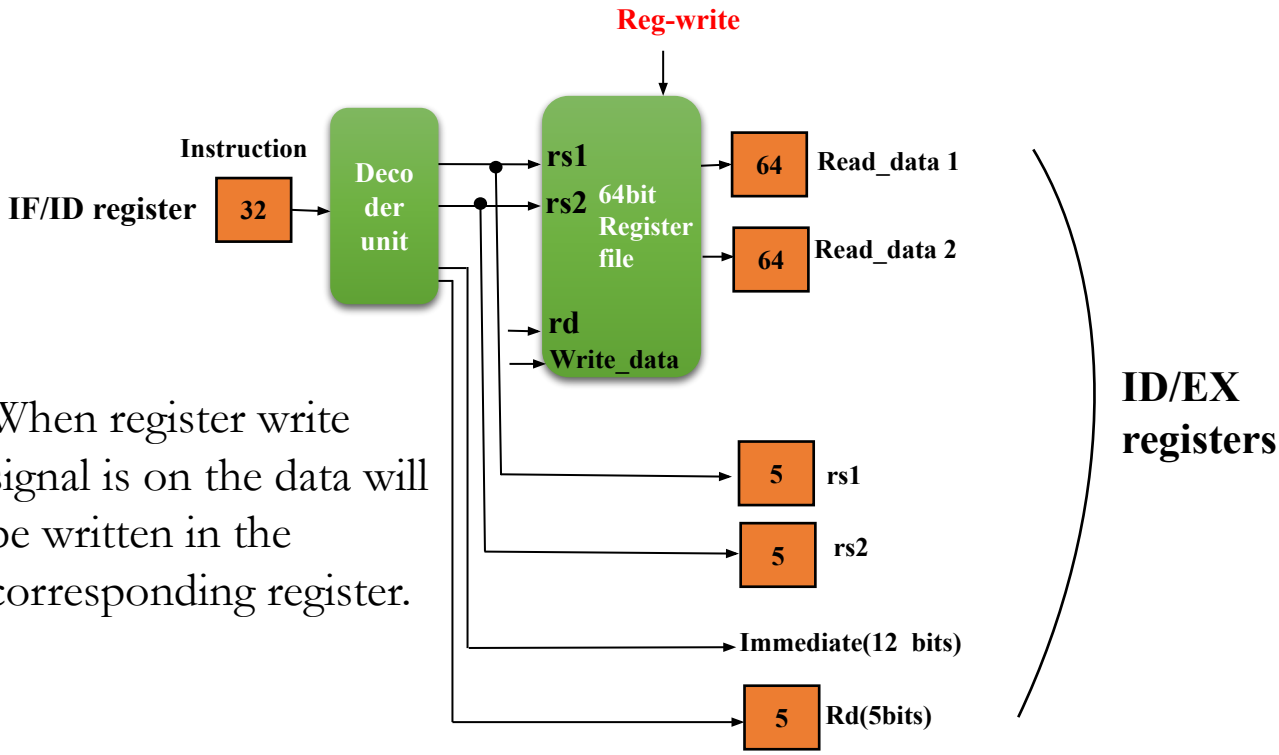
funct7	rs2	rs1	funct3	rd	opcode
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

I-format:

Immediate	rs1	funct3	rd	opcode
12 bits	5 bits	3 bits	5 bits	7 bits

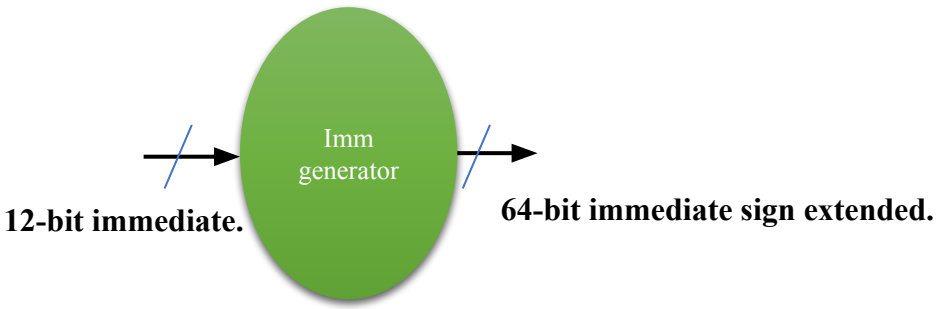
S-format:

Imm[11:5]	rs2	rs1	funct3	Imm[4:0]	opcode
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits



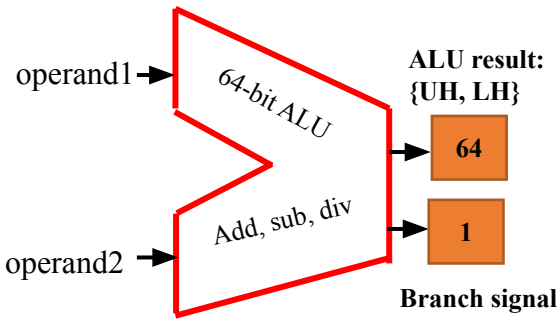
When register write signal is on the data will be written in the corresponding register.

Role of immediate generator:



This imm generator performs sign extension of 12-bit immediate to 64-bit immediate. The reason for extension is that other registers in register file are 64-bit and the ALU is designed for 64-bits.

Role of 64-bit ALU:



In the data path designed here the ALU is designed for 64-bits and it performs addition and subtraction by carry look ahead adder logic and multiplication by booth's algorithm. It gives one output of 64-bits size.

Role of Data cache:

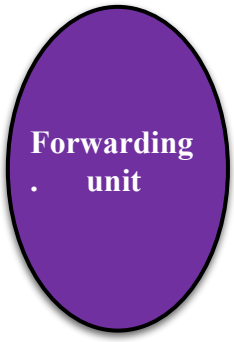
The data cache is used to store data values. The access time to access data from data cache is way more than access time to access data from register file. When data read signal is on data is read from data cache, whenever data write signal is on data is written in the data cache.

Control Unit and Control Signals:

S.No	Signals	Role
1)	Register write	If register write signal is 1 the data will be written in the register file. Otherwise data is not written in the register file.
2)	Mem_to_register	If mem_to_register signal is 1 the data read from the data cache (ld) will be write data value for register file. Otherwise alu_result will be the write data value for register file.
3)	Mem_read	If mem_read signal is 1, data value is read from the data cache using the address otherwise the data is not read from the data cache.
4)	Mem_write	If mem_write signal is 1, data value is written in the data cache else the data is not written in the data cache.
5)	PC_source	PC_source will be 1 when the instruction is branch instruction to select PC + 2 X imm Otherwise PC + 4 will be the updated program counter.
6)	ALU_source	This is execution signal, this depends on opcode and funct7 values. If instruction is R, based on funct7 value the type of operation like add, sub or mul is determined. And the operands will be data from rs1 and rs2. For other instructions like I format like addi, ld and S-type like sd only addition is needed to performed in execution stage, the addition involves immediate. One of the operand will be immediate.

Operation to be performed	Funct 7	Funct 3	Opcode
ADDITION(+)	1000000	000	0110011
SUBTRACTION(-)	0100000	000	0110011
MULTIPLICATION(X)	0000001	000	0110011

Working Of Forwarding Unit(aka by-passing):



The forwarding unit is represented by this block diagram but this forwarding unit can be viewed as several multiplexers placing together. The multiplexers involved in this forwarding logic can be seen from verilog code placed in page12 and page13 from this report.

Forwarding unit is added in this data path to decrease the number of stalls and there by decreasing CPI and increasing performance of the processing unit.
The forwarding unit decides the operands to be sent to the ALU. The below conditions make sure that the most recent value is forwarded. The most recent value to forward will be there in EX/MEM pipeline.

1. If rs1 in ID/EX pipeline is equal to rd of EX/MEM pipeline, with write back in EX/MEM, and registers not being zero:
   Then alu_result in EX/MEM is passed as operand A.
2. If the above condition is not satisfied and if rs1 in ID/EX is equal to rd of MEM/WB pipeline, with registers not being zero and write back is on.
   1. If write back signals in MEM/WB is indicating that the instruction in it was load then register value corresponding to the value read from data cache is passed.
   2. If not then alu_result from MEM/WB is passed.
3. If all the above conditions are not met then read_data1 from ID/EX is passed as operand A to ALU.

For operand B, the similar conditions are checked as above but additionally making sure that immediate is passed whenever instruction ID/EX is not R-type. As the conditions for passing B is complicated than A that can be verified from the Verilog code.

- **Pipelined data path Verilog code:**

```
module pipelined_datapath(
    input clk,
    input [31:0]instruction,
    input instruction_memory_cycle,
    input data_memory_cycle,
    input [64:0]data,
    input [3:0]data_location, // 10 locations needed

    output [63:0]PC,
    output [95:0]pipeline_IF_ID,
    output [289:0]pipeline_ID_EX,
    output [202:0]pipeline_EX_MEM,
    output [134:0]pipeline_MEM_WB,
    output reg [63:0]data_written_in_regfile,
    output reg [63:0]data_written_in_data_mem,
    output reg [31:0]data_written_in_ins_mem,
    output reg overflow
    );

    /*
    20 32-bit instructions can be stored in instruction memory
    */
    reg [31:0]instruction_memory[19:0];

    /*
    20 64-bit data can be stored in data memory
    */
    reg [63:0]data_memory[19:0];

    /*
    32 64-bit register file
    */
    reg [31:0]register_file[63:0];

    /*
    IF/ID pipeline
    PC - 64 bits
    instruction - 32 bits
    total bits = 96
    */
    reg [95:0]pipeline_IF_ID;

    /*
    ID/EX pipeline
    write back signals - 2 bits (register write signal , mem to reg signal)
    memory signals - 3 bits
                     (memory read, memory write, branch)
    execution : opcode - 7 bits
                funct7 - 7 bits
    PC - 64 bits
    read_data1 = 64 bits
    read_data2 = 64 bits
    rs1 = 5 bits
    rs2 = 5 bits
    immediate sign extended = 64 bits
    rd = 5 bits
    total = 2 + 3 + 7 + 7 + 64 + 64 + 64 + 5 + 5 + 64 + 5 = 290 bits
    */
    reg [289:0]pipeline_ID_EX;

    /*
    EX/MEM pipeline
    write back signals - 2 bits (register write signal , mem to reg signal)
    memory signals - 3 bits
                     (memory read, memory write, branch)
    new branch = 64 bits
    alu_branch_signal = 1 bit
    alu_result = 64 bits
    read_data2 = 64 bits
    rd = 5 bits

    total_bits = 203 bits
    */
    reg [202:0]pipeline_EX_MEM;

    /*
    MEM/WB pipeline
    write back signals - 2 bits (register write signal , mem to reg signal)
    mem data = 64 bits
    alu_result = 64 bits
    rd = 5 bits

    total bits = 135 bits
    */
    reg [134:0]pipeline_MEM_WB;

    integer iterator = 0;
    integer i;

    reg [63:0]PC;

    reg [63:0]A;
    reg [63:0]B;
    reg [63:0]B1;
    reg [63:0]G;
    reg [63:0]P;
    reg [63:0]sum;
    reg [64:0]carry;
    reg [127:0]mul;
    reg [1:0]temp;
    reg E1;
    reg [63:0]alu_result;

    reg PC_source;

    always @ (posedge clk)
    begin
        register_file[0] <= 0;
        if (instruction_memory_cycle == 1'b1)
```

Instruction cache.

Data cache.

Register file.

Pipeline IF/ID

Pipeline ID/EX.

Pipeline EX/MEM.

Pipeline MEM/WB.

```
        begin
            // initially to put instructions in instructiojn memory
            PC <= 0;
            for(i = 0; i <= 31; i = i + 1)
            begin
                register_file[i] = 0;
            end

            pipeline_IF_ID <= 0;
            pipeline_ID_EX <= 0;
            pipeline_EX_MEM <= 0;
            pipeline_MEM_WB <= 0;
            alu_result <= 0;

            instruction_memory[iterator] <= instruction;
            iterator <= iterator + 1;

            data_written_in_regfile <= 0;
            data_written_in_data_mem <= 0;
            data_written_in_ins_mem <= instruction;
        end
```

Initially writing some instructions into instruction-cache.

```
        else if (data_memory_cycle == 1'b1)
        begin
            // initially to put some data into the data memory
            pipeline_IF_ID <= 0;
            pipeline_ID_EX <= 0;
            pipeline_EX_MEM <= 0;
            pipeline_MEM_WB <= 0;
            alu_result <= 0;
            data_memory[data_location] <= data;

            data_written_in_regfile <= 0;
            data_written_in_data_mem <= data;
            data_written_in_ins_mem <= 0;
        end
```

Initially writing some data values into data cache.

```
        else
        begin

// |||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
            // instrction fetch
            pipeline_IF_ID[63:0] <= PC;
            pipeline_IF_ID[95:64] <= instruction_memory[PC >> 2];
// |||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
```

Instruction fetch stage.

```
           // Write back stage
           if (pipeline_MEM_WB[0] == 1'b1) // register write signal
           begin
                if(pipeline_MEM_WB[1] == 1'b1) // writing data memory result
                begin
                    register_file[pipeline_MEM_WB[134:130]] <= pipeline_MEM_WB[65:2];
                    data_written_in_regfile <= pipeline_MEM_WB[65:2];
                end
                else
                begin
                    register_file[pipeline_MEM_WB[134:130]] <= pipeline_MEM_WB[129:66];
                    data_written_in_regfile <= pipeline_MEM_WB[129:66];
                end
           end
           else
           begin
                data_written_in_regfile <= 0;
           end
```

Write back stage.

```
// |||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
           // program counter calculation and memory stage
            PC_source <= pipeline_EX_MEM[2] & pipeline_EX_MEM[69];

            if (PC_source == 1'b1)
            begin
                PC <= pipeline_EX_MEM[68:5];
            end
            else
            begin
                PC <= PC + 4;
            end

           if(pipeline_EX_MEM[4] == 1'b1) // read(ld)
           begin
                pipeline_MEM_WB[65:2] <= data_memory[pipeline_EX_MEM[133:70]];

                data_written_in_data_mem = 0;
           end
           else if(pipeline_EX_MEM[3] == 1'b1) // write(sd)
           begin
                pipeline_MEM_WB[65:2] <= 0;
                data_memory[pipeline_EX_MEM[133:70]] <= pipeline_EX_MEM[197:134]; // read data 2

                data_written_in_data_mem <= pipeline_EX_MEM[197:134];
           end
           else
           begin
                pipeline_MEM_WB[65:2] <= 0;

                data_written_in_data_mem = 0;
           end

           pipeline_MEM_WB[1:0] <= pipeline_EX_MEM[1:0]; // write back signals
           pipeline_MEM_WB[129:66] <= pipeline_EX_MEM[133:70]; // alu result
           pipeline_MEM_WB[134:130] <= pipeline_EX_MEM[202:198]; // destination register
```

Program counter and MEM stage.

```
// |||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
           // Execution stage

           // Forwarding Logic with multiplexers
           if((pipeline_EX_MEM[202:198] != 5'b00000) && (pipeline_EX_MEM[202:198] == pipeline_ID_EX[215:211]) && (pipeline_EX_MEM[0] == 1'b1) && (pipeline_EX_MEM[1] == 1'b0))
           begin
                A = pipeline_EX_MEM[133:70]; // alu result
           end
           else if((pipeline_MEM_WB[134:130] != 5'b00000) && (pipeline_MEM_WB[134:130] == pipeline_ID_EX[215:211]) && (pipeline_MEM_WB[0] == 1'b1))
           begin
                if(pipeline_MEM_WB[1] == 1'b1)
                begin
                    A = pipeline_MEM_WB[65:2]; // data from data_memory
                end
                else
                begin
```

Forwarding unit

```
                A = pipeline_MEM_WB[129:66]; // alu result
            end
        end
        else
        begin
            A = pipeline_ID_EX[146:83]; // read data 1
        end


        if ((pipeline_EX_MEM[202:198] != 5'b00000) && (pipeline_EX_MEM[202:198] == pipeline_ID_EX[220:216]) && (pipeline_EX_MEM[0] == 1'b1) && (pipeline_EX_MEM[1] == 1'b0) )
        begin
            if (pipeline_ID_EX[11:5]  != 7'b0110011)
            begin
                pipeline_EX_MEM[197:134] <= pipeline_EX_MEM[133:70]; // alu result
                B = pipeline_ID_EX[284:221]; // immediate
            end
            else
            begin
                pipeline_EX_MEM[197:134] <= pipeline_ID_EX[210:147]; // read data 2
                B = pipeline_EX_MEM[133:70]; // alu result
            end
        end
        else if((pipeline_MEM_WB[134:130] != 5'b00000) && (pipeline_MEM_WB[134:130] == pipeline_ID_EX[220:216]) &&  (pipeline_MEM_WB[0] == 1'b1))
        begin
            if(pipeline_MEM_WB[1] == 1'b1) // ld
            begin
                 if (pipeline_ID_EX[11:5]  != 7'b0110011)
                begin
                    pipeline_EX_MEM[197:134] <= pipeline_MEM_WB[65:2]; // data from data mem
                    B = pipeline_ID_EX[284:221]; // immediate
                end
                else
                begin
                    pipeline_EX_MEM[197:134] <= pipeline_ID_EX[210:147]; // read data 2
                    B = pipeline_MEM_WB[65:2]; // data from data memory
                end
            end
            else
            begin
                 if (pipeline_ID_EX[11:5]  != 7'b0110011)
                begin
                    pipeline_EX_MEM[197:134] <= pipeline_MEM_WB[129:66]; // data from data mem
                    B = pipeline_ID_EX[284:221]; // immediate
                end
                else
                begin
                    pipeline_EX_MEM[197:134] <= pipeline_ID_EX[210:147]; // read data 2
                    B = pipeline_MEM_WB[129:66]; // data from data memory
                end
            end
        end
        else
        begin
            if (pipeline_ID_EX[11:5] == 7'b0110011) // opcode (R-type instruction)
            begin
                B = pipeline_ID_EX[210:147]; // read_data 2
                pipeline_EX_MEM[197:134] = pipeline_ID_EX[210:147]; // read data 2
            end
            else // (S-type, I -type)
            begin
                B = pipeline_ID_EX[284:221];  // immediate
                pipeline_EX_MEM[197:134] = pipeline_ID_EX[210:147]; // read data 2
            end
        end

        // Performing operation on A and B
        if(pipeline_ID_EX[11:5] == 7'b0110011) // opcode
        begin
            // R-format
            if (pipeline_ID_EX[18:12] == 7'b1000000) // funct 7 values
            begin
                // addition
                carry[0] = 0;
                for(i = 0; i <= 63; i = i + 1)
                begin
                    G[i] = A[i] & B[i];
                    P[i] = A[i] ^ B[i];
                    sum[i] = carry[i] ^ P[i];
                    carry[i + 1] = G[i] | (carry[i] & P[i]);
                end
                alu_result = sum;
                if (A[63] == 0 && B[63] == 0 && sum[63] == 1) // adding two positive numbers giving a negative number is not possible
                begin
                    overflow = 1'b1;
                end
                else if (A[63] == 1 && B[63] == 1 && sum[63] == 0) // adding two negative numbers giving a positive numbers is not possible
                begin
                     overflow = 1'b1;
                end
                else
                begin
                    overflow = 1'b0;
                end
            end
            else if(pipeline_ID_EX[18:12] == 7'b0100000)
            begin
                // subtraction
                B1 = ~(B) + 1;
                carry[0] = 0;
                for(i = 0; i <= 63; i = i + 1)
                begin
                    G[i] = A[i] & B1[i];
                    P[i] = A[i] ^ B1[i];
                    sum[i] = carry[i] ^ P[i];
                    carry[i + 1] = G[i] | (carry[i] & P[i]);
                end
                alu_result = sum;
                if (A[63] == 0 && B1[63] == 0 && sum[63] == 1) // adding two positive numbers giving a negative number is not possible
                begin
                    overflow = 1'b1;
                end
                else if (A[63] == 1 && B1[63] == 1 && sum[63] == 0) // adding two negative numbers giving a postive numbers is not possible
```

Execution stage.

```
                    begin
                        overflow = 1'b1;
                    end
                    else
                    begin
                        overflow = 1'b0;
                    end
                end
                else if(pipeline_ID_EX[18:12] == 7'b0000001)
                begin
                    // multiplication
                    B1 = ~(B) + 1;
                    mul = 0;
                    E1 = 0;
                    for (i = 0; i <= 63; i = i + 1)
                    begin
                        temp = {A[i], E1};
                        case (temp)
                            2'd2 : mul[127:64] = mul[127:64] + B1;
                            2'd1 : mul[127:64] = mul[127:64] + B;
                        endcase
                        mul = mul >> 1;
                        mul[127] = mul[126];
                        E1 = A[i];
                    end
                    if (B == 64'd9223372036854775808)
                    begin
                        mul = -mul;
                    end
                    overflow = 1'b0;
                    alu_result = mul[63:0];
                end
            end
            else if(pipeline_ID_EX[11:5] == 7'b0010011 || pipeline_ID_EX[11:5] == 7'b0000011 || pipeline_ID_EX[11:5] == 7'b0100011)
            begin
                // I-format or S-format
                // addition
                carry[0] = 0;
                for(i = 0; i <= 63; i = i + 1)
                begin
                    G[i] = A[i] & B[i];
                    P[i] = A[i] ^ B[i];
                    sum[i] = carry[i] ^ P[i];
                    carry[i + 1] = G[i] | (carry[i] & P[i]);
                end
                alu_result = sum;
                if (A[63] == 0 && A[63] == 0 && sum[63] == 1) // adding two positive numbers giving a negative number is not possible
                begin
                    overflow = 1'b1;
                end
                else if (A[63] == 1 && B[63] == 1 && sum[63] == 0) // adding two negative numbers giving a postive numbers is not possible
                begin
                    overflow = 1'b1;
                end
                else
                begin
                    overflow = 1'b0;
                end
             end
             else
             begin
                 alu_result = 0;
                 overflow = 0;
             end

            // writing in the registers
            pipeline_EX_MEM[1:0] <= pipeline_ID_EX[1:0]; // write back signals
            pipeline_EX_MEM[4:2] <= pipeline_ID_EX[4:2]; // memory signals
            pipeline_EX_MEM[68:5] <= pipeline_ID_EX[82:19] + (pipeline_ID_EX[284:221] * 2); // branch calculation
            pipeline_EX_MEM[69] <= 0; // branch signal
            pipeline_EX_MEM[133:70] <= alu_result; // alu result
            pipeline_EX_MEM[202:198] <= pipeline_ID_EX[289:285]; // destination register
// ||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
            // INSTRUCTION DECODE
             if( pipeline_IF_ID[70:64] == 7'b0110011)
             begin
                // R type instruction
                pipeline_ID_EX[1:0] <= 2'b01; // write back signals
                pipeline_ID_EX[4:2] <= 3'b000; // memory read, memory write, branch signal
                pipeline_ID_EX[11:5] <=  pipeline_IF_ID[70:64]; // opcode values
                pipeline_ID_EX[18:12] <= pipeline_IF_ID[95:89]; // funct 7 values
                pipeline_ID_EX[82:19] <= pipeline_IF_ID[63:0]; // PC value
                pipeline_ID_EX[146:83] <= register_file[pipeline_IF_ID[83:79]]; // read_data 1
                pipeline_ID_EX[210:147] <= register_file[pipeline_IF_ID[88:84]]; // read_data 2
                pipeline_ID_EX[215:211] <= pipeline_IF_ID[83:79]; // rs1
                pipeline_ID_EX[220:216] <= pipeline_IF_ID[88:84]; // rs2
                pipeline_ID_EX[284:221] <= 0; // immediate
                pipeline_ID_EX[289:285] <= pipeline_IF_ID[75:71]; // destination register
            end
            else if(pipeline_IF_ID[70:64] == 7'b0010011)
            begin
                // addi
                pipeline_ID_EX[1:0] <= 2'b01; // write back signal
                pipeline_ID_EX[4:2] <= 3'b000; // memory read, memory write, branch signal
                pipeline_ID_EX[11:5] <=  pipeline_IF_ID[70:64]; // opcode values
                pipeline_ID_EX[18:12] <= 0; // funct 7 values
                pipeline_ID_EX[82:19] <= pipeline_IF_ID[63:0]; // PC value
                pipeline_ID_EX[146:83] <= register_file[pipeline_IF_ID[83:79]]; // read_data 1
                pipeline_ID_EX[210:147] <= 0; // read_data 2
                pipeline_ID_EX[215:211] <= pipeline_IF_ID[83:79]; // rs1
                pipeline_ID_EX[220:216] <= 0; // rs2
                // sign extending 12 bit immediate
                pipeline_ID_EX[232:221] <= pipeline_IF_ID[95:84];
                for(i = 233; i <= 284; i = i + 1)
                begin
                    pipeline_ID_EX[i] = pipeline_IF_ID[95];
                end
                pipeline_ID_EX[289:285] = pipeline_IF_ID[75:71]; // destination register
            end
            else if(pipeline_IF_ID[70:64] == 7'b0000011)
            begin
                // ld type instruction
                pipeline_ID_EX[1:0] <= 2'b11; // write back signal
                pipeline_ID_EX[4:2] <= 3'b100; // memory read, memory write, branch signal
```

Instruction decode stage.

```
                pipeline_ID_EX[11:5] <=  pipeline_IF_ID[70:64]; // opcode values
                pipeline_ID_EX[18:12] <= 0; // funct 7 values
                pipeline_ID_EX[82:19] <= pipeline_IF_ID[63:0]; // PC value
                pipeline_ID_EX[146:83] <= register_file[pipeline_IF_ID[83:79]]; // read_data 1
                pipeline_ID_EX[210:147] <= 0; // read_data 2
                pipeline_ID_EX[215:211] <= pipeline_IF_ID[83:79]; // rs1
                pipeline_ID_EX[220:216] <= 0; // rs2
                // immediate sign extension
                pipeline_ID_EX[232:221] <= pipeline_IF_ID[95:84];
                for(i = 233; i <= 284; i = i + 1)
                begin
                    pipeline_ID_EX[i] = pipeline_IF_ID[95];
                end
                pipeline_ID_EX[289:285] = pipeline_IF_ID[75:71]; // destination register
            end
            else if(pipeline_IF_ID[70:64] == 7'b0100011)
            begin
                // S type instruction
                pipeline_ID_EX[1:0] <= 2'b00; // write back signals
                pipeline_ID_EX[4:2] <= 3'b010; // memory read, memory write, branch signal
                pipeline_ID_EX[11:5] <=  pipeline_IF_ID[70:64]; // opcode values
                pipeline_ID_EX[18:12] <= 0; // funct 7 values
                pipeline_ID_EX[82:19] <= pipeline_IF_ID[63:0]; // PC value
                pipeline_ID_EX[146:83] <= register_file[pipeline_IF_ID[83:79]]; // read_data 1
                pipeline_ID_EX[210:147] <= register_file[pipeline_IF_ID[88:84]]; // read_data 2
                pipeline_ID_EX[215:211] <= pipeline_IF_ID[83:79]; // rs1
                pipeline_ID_EX[220:216] <= pipeline_IF_ID[88:84]; // rs2
                // sign extending imeediate
                pipeline_ID_EX[225:221] <= pipeline_IF_ID[75:71];
                pipeline_ID_EX[232:226] <= pipeline_IF_ID[95:89];
                for(i = 233; i <= 284; i = i + 1)
                begin
                    pipeline_ID_EX[i] = pipeline_IF_ID[95];
                end
                pipeline_ID_EX[289:285] = 0; // destination register
            end
            else
            begin
                pipeline_ID_EX = 0;
            end
        end
    end
endmodule
```

TEST Bench:

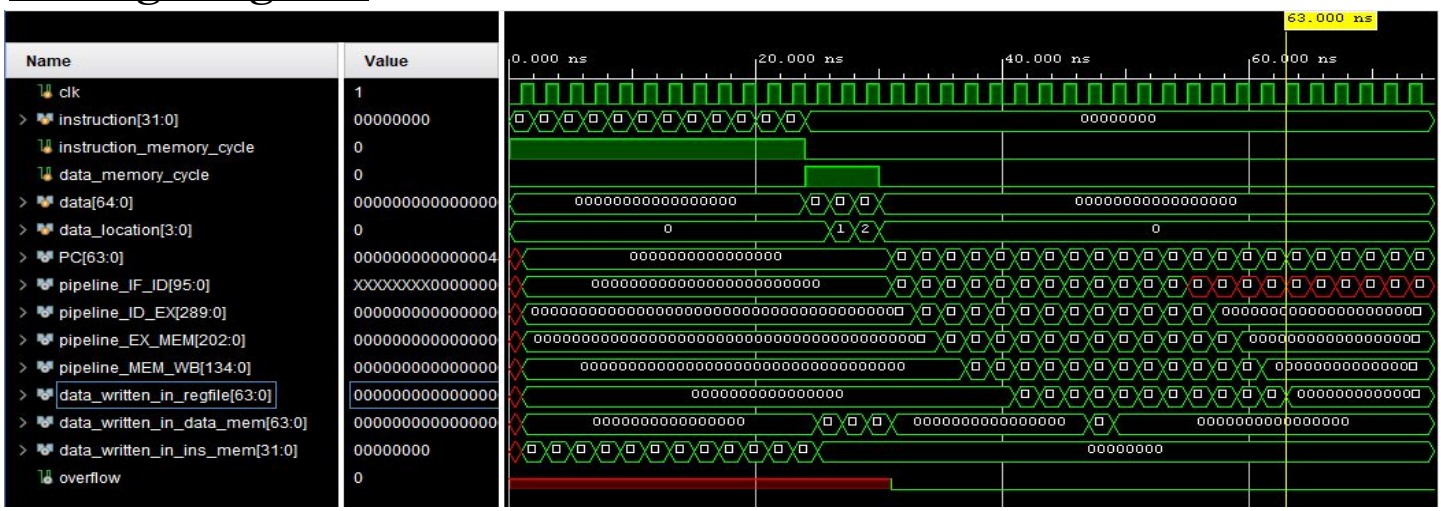
The below test bench is written to simulate different test cases.

```
/*
R-type instructions
add                 1000000_rs2_rs1_000_rd_0110011
sub                 0100000_rs2_rs1_000_rd_0110011
mul                 0000001_rs2_rs1_000_rd_0110011

I-type instructions
addi                imm(12bits)_rs1_000_rd_0010011
ld                  imm(12bits)_rs1_011_rs_0000011

S-type instructions
sd                  imm[11:5]_rs2_rs1_011_imm[4:0]_0100011
*/
`timescale 1ns / 1ps

module test_bench;
    reg clk;
    reg [31:0]instruction;
    reg instruction_memory_cycle;
    reg data_memory_cycle;
    reg [64:0]data;
    reg [3:0]data_location;

    wire [63:0]PC;
    wire [95:0]pipeline_IF_ID;
    wire [289:0]pipeline_ID_EX;
    wire [202:0]pipeline_EX_MEM;
    wire [134:0]pipeline_MEM_WB;
    wire [63:0]data_written_in_regfile;
    wire [63:0]data_written_in_data_mem;
    wire [31:0]data_written_in_ins_mem;
    wire overflow;

    pipelined_datapath uut(
    clk,
    instruction,
    instruction_memory_cycle,
    data_memory_cycle,
    data,
    data_location,


    PC,
    pipeline_IF_ID,
    pipeline_ID_EX,
    pipeline_EX_MEM,
    pipeline_MEM_WB,
    data_written_in_regfile,
    data_written_in_data_mem,
    data_written_in_ins_mem,
```

```
51          overflow
52          );
53
54          initial
55          begin
56              clk = 1'b0;
57
58              instruction = 32'b00000000000001110100011101001001 1;// addi x29, x29, 0 (I - type instruction)
59              instruction_memory_cycle = 1'b1;
60              data_memory_cycle = 1'b0;
61              data = 0;
62              data_location = 0;
63
64              #2
65              instruction = 32'b00000000000011111000011110001001 1;//  addi x30, x30, 1 (I - type instruction)
66              instruction_memory_cycle = 1'b1;
67              data_memory_cycle = 1'b0;
68              data = 0;
69              data_location = 0;
70
71              #2
72              instruction = 32'b00000000000001110101111100000001 1;// ld x28, 0(x29)
73              instruction_memory_cycle = 1'b1;
74              data_memory_cycle = 1'b0;
75              data = 0;
76              data_location = 0;
77
78               #2
79              instruction = 32'b00000000000011110011110110000001 1; // ld x27 0(x30)
80              instruction_memory_cycle = 1'b1;
81             data_memory_cycle = 1'b0;
82             data = 0;
83             data_location = 0;
84
85             #2
86             instruction = 32'b10000001110111100000010100110011; // add x10, x28, x29
87             instruction_memory_cycle = 1'b1;
88             data_memory_cycle = 1'b0;
89             data = 0;
90             data_location = 0;
91
92             #2
93             instruction = 32'b00000000101011101011000000100011;// sd x10, 0(x29)
94             instruction_memory_cycle = 1'b1;
95             data_memory_cycle = 1'b0;
96             data = 0;
97             data_location = 0;
98
99             #2
100            instruction = 32'b00000000000011101011010110000011; // ld x11 0(x29)
101            instruction_memory_cycle = 1'b1;
102            data_memory_cycle = 1'b0;
103            data = 0;
104            data_location = 0;
105
106            #2
107            instruction = 32'b10000000000000000000000000110011; // add x0, x0, x0 (R-type instruction) STALL
108            instruction_memory_cycle = 1'b1;
109            data_memory_cycle = 1'b0;
110            data = 0;
111            data_location = 0;
112
113            #2
114            instruction = 32'b00000010101111111000000110011001 1; // mul x6, x11, x31 (R-type instruction)
115            instruction_memory_cycle = 1'b1;
116            data_memory_cycle = 1'b0;
117            data = 0;
118            data_location = 0;
119
120            // Putting -20 from data to register x18
121            #2
122            instruction = 32'b00000000000011111001110010000001 1; // ld x18 1(x30)
123            instruction_memory_cycle = 1'b1;
124            data_memory_cycle = 1'b0;
125            data = 0;
126            data_location = 0;
127
128            #2
129            instruction = 32'b10000000000000000000000000110011; // add x0, x0, x0 (R-type instruction) STALL
130            instruction_memory_cycle = 1'b1;
131           data_memory_cycle = 1'b0;
132           data = 0;
133           data_location = 0;
134
135            #2
136            instruction = 32'b01000001001011011000110110110011;//  sub x19, x27, x18 (R - type instruction)
137            instruction_memory_cycle = 1'b1;
138            data_memory_cycle = 1'b0;
139            data = 0;
140            data_location = 0;
141
142
143            // Data 45 at data_cache[0]
144            #2
145            instruction = 0;
146            instruction_memory_cycle = 1'b0;
```

```
147         data_memory_cycle = 1'b1;
148         data = 45;
149         data_location = 0;
150
151         // Data -20 at data_cache[1]
152          #2
153         instruction = 0;
154         instruction_memory_cycle = 1'b0;
155         data_memory_cycle = 1'b1;
156         data = 20;
157         data_location = 1;
158
159         // Data 20 at data_cache[2]
160         #2
161         instruction = 0;
162         instruction_memory_cycle = 1'b0;
163         data_memory_cycle = 1'b1;
164         data = -20;
165         data_location = 2;
166
167         #2
168         instruction = 0;
169         instruction_memory_cycle = 1'b0;
170         data_memory_cycle = 1'b0;
171         data = 0;
172         data_location = 0;
173     end
174
175     always #1 clk = ~clk;
176
177     initial #75 $finish;
178 endmodule
```

Timing Diagram:



Description of the timing diagram for written test bench:

The register values in each cycle can be observed from the above timing diagram. **pipeline_IF_ID**, **pipeline_ID_EX**, **pipeline_EX_MEM**, **pipeline_MEM_WB**, represents the pipeline register values. **data_written_in_regfile** represents the data written back in the register file**. data_written_in_data_mem** represents the data written in the data cache. **data_written_in_ins_mem** represents the data written in the ins cache.

Initially 12 instructions are written in ins-cache including one stall. Data values 45, 20, -20 are written at the locations 0, 1, 2 of data cache. We can observe that in total 5 clock cycles are taken to completely run a instruction, ld – the critical instruction requires all the 5-stages completely.

- Analysis of timing diagram with the help of test bench written:

After writing the instructions in the instruction cache and data values in data cache the scenario is:

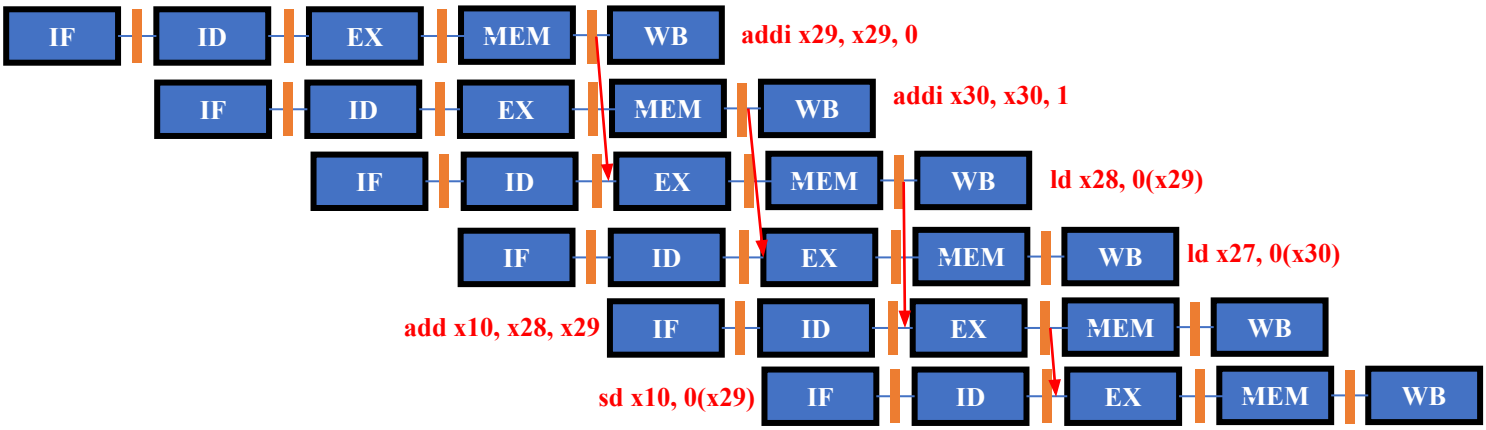
Instruction Cache:

Address	Instrucion	Description
0000	addi x29, x29, 0	# x29 = 0 + 0 = 0
0001	addi x30, x30, 1	# x30 = 0 + 1 = 1
0010	ld x28, 0(x29)	# x28 = MEM[0 + 0] =45
0011	ld x27, 0(x30)	# x27 = MEM[0 + 1] = 20
0100	add x10, x28, x29	# x10 = 45 + 0 = 45
0101	sd x10, 0(x29)	# MEM[0 + 0] = 45
0110	ld x11 0(x29)	# x11 = MEM[0 + 0] = 45
0111	add x0, x0, x0	# [STALL]
1000	mul x6, x11, x31	# x6 = 45 * 1 = 45
1001	ld x18, 1(x30)	# x18 = MEM[1 + 1] = -20
1010	add x0, x0, x0	# [STALL]
1011	sub x19, x27, x18	# x19 = 20 – (-20) = 40

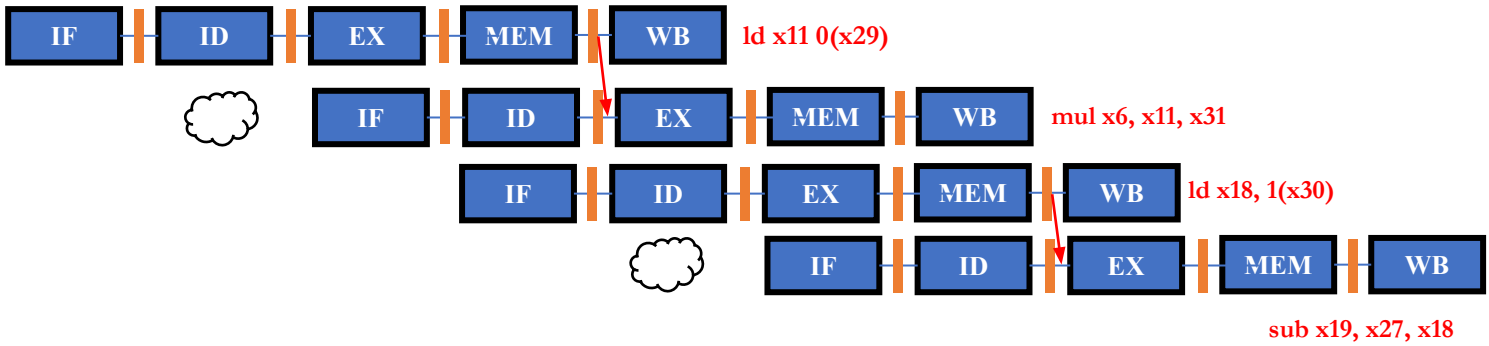
Data-cache:

Address	Data
0000	45
0001	20
0010	-20

The forwarding lines are indicated in red. The below picture depicts the execution of first 6 instructions. As there is full forwarding there is no requirement of STALLs in first 6 instructions.



In the below picture instructions and its computation is shown along with full forwarding:

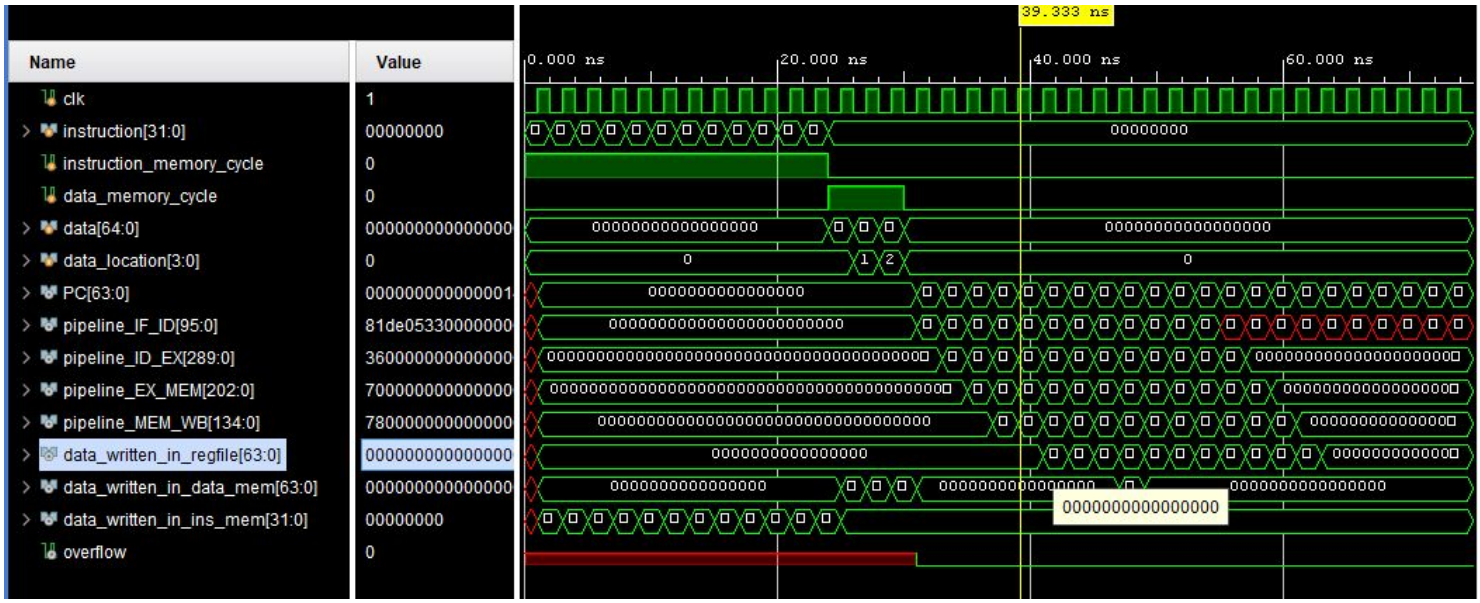


Including 2 stalls and 10 instructions on total we can say that 12 instructions are executed. Here stalls are implemented by add x0, x0, x0.

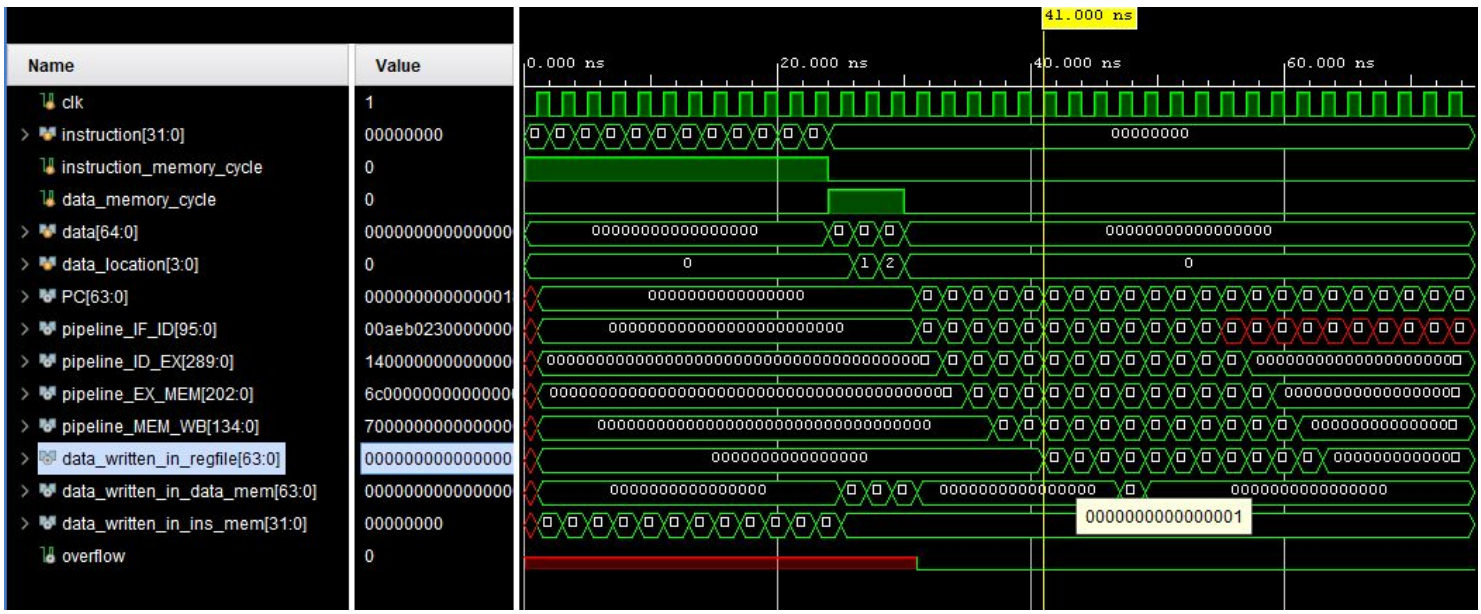
Total Number of cycles for this 10 instructions(2-stalls) = 16
Actual Number of instructions = 10
Stalls = 2
[average CPI]: CPI = (16) / (10) = 1.6 (2-stalls are not considered as instructions)

The CPI is not 1(ideal) because some cycles are wasted in stalling and also time required to get full pipelining.

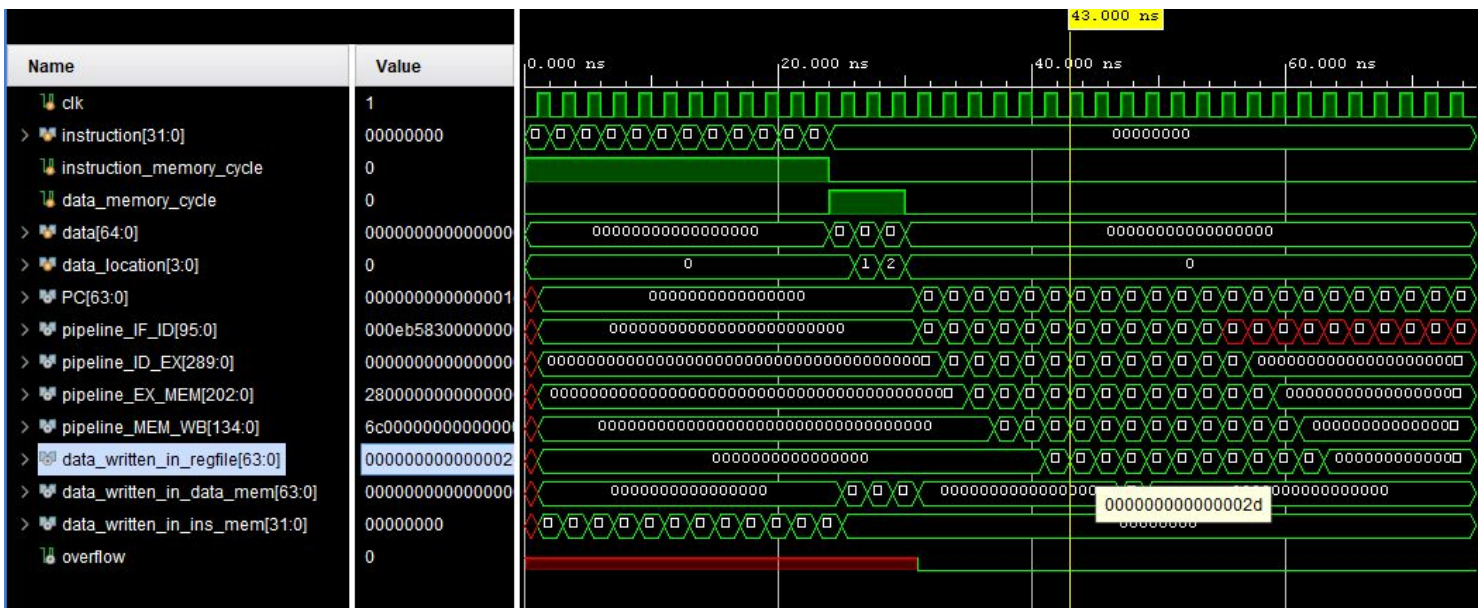
Instruction: **addi x29, x29, 0.** so data_written_in_regfile = 0 :



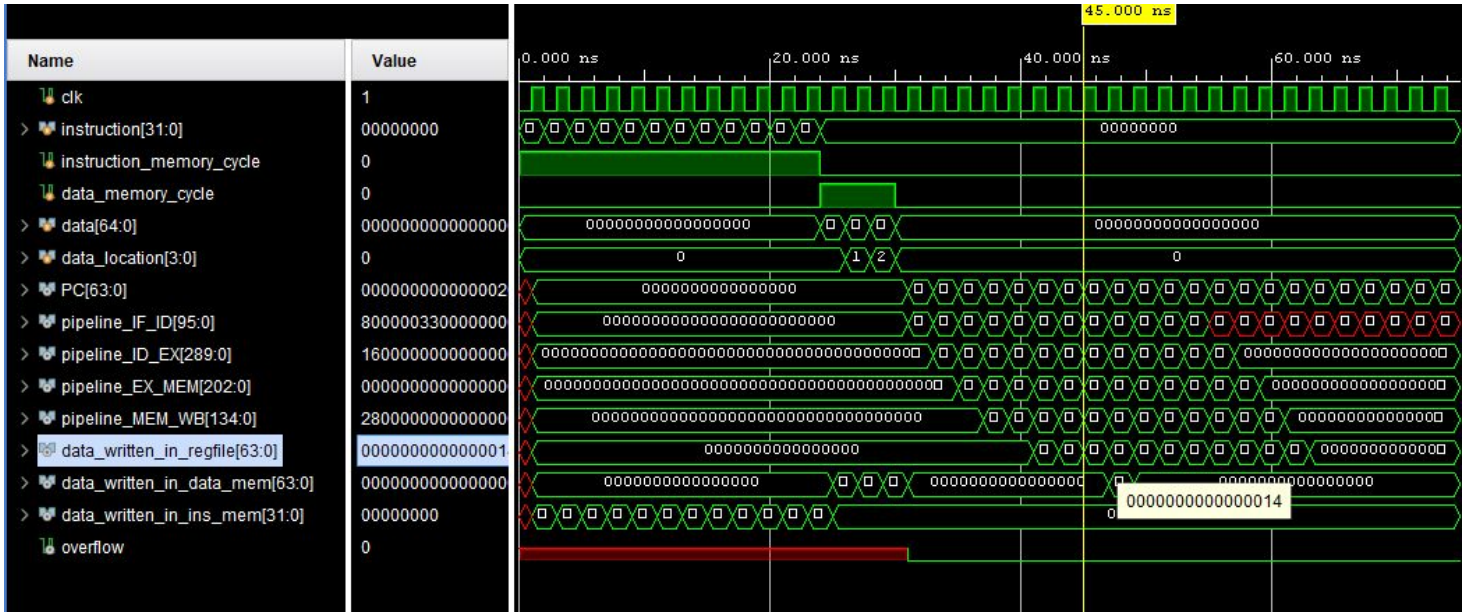
Instruction : **addi x30, x30, 1.** so data_written_in_regfile = 1.



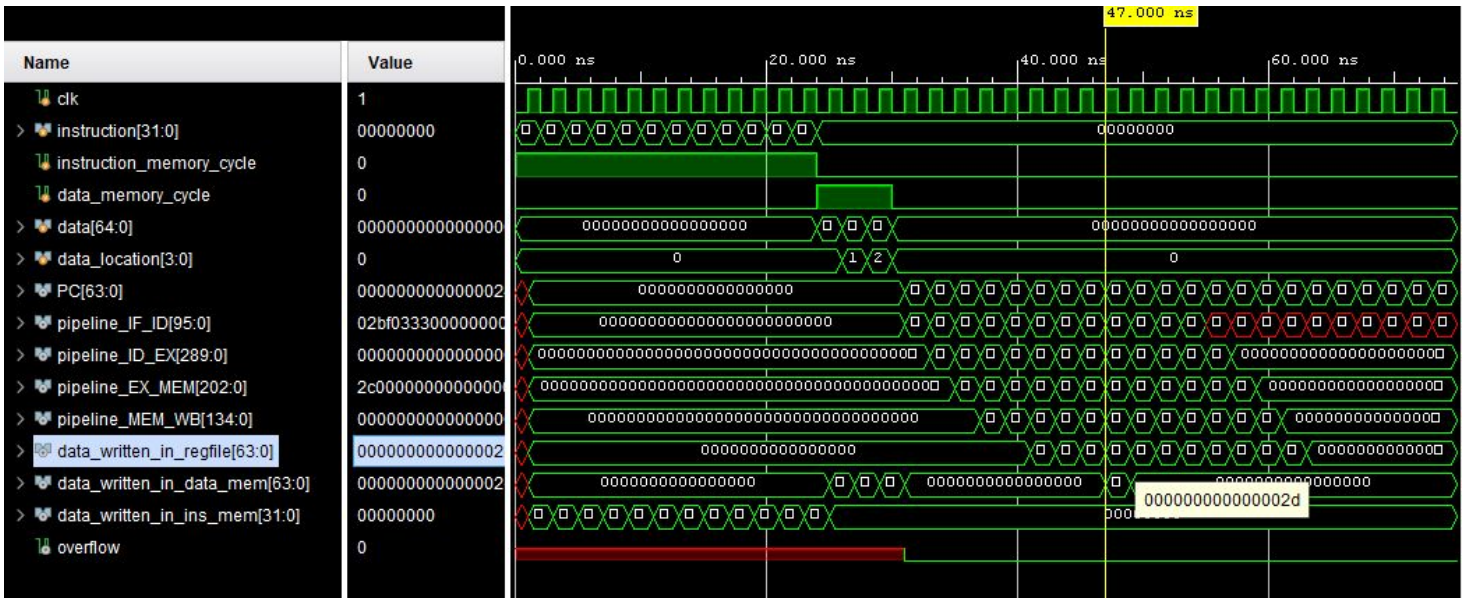
Instruction : **ld x28, 0(x29).** so data_written_in_regfile = MEM[0] = 45 (2d in hexadecimal)



Instruction: **ld x27, 0(x30).** so data_written_in_regfile = MEM[1] = 20 (14 in dexadecimal) :

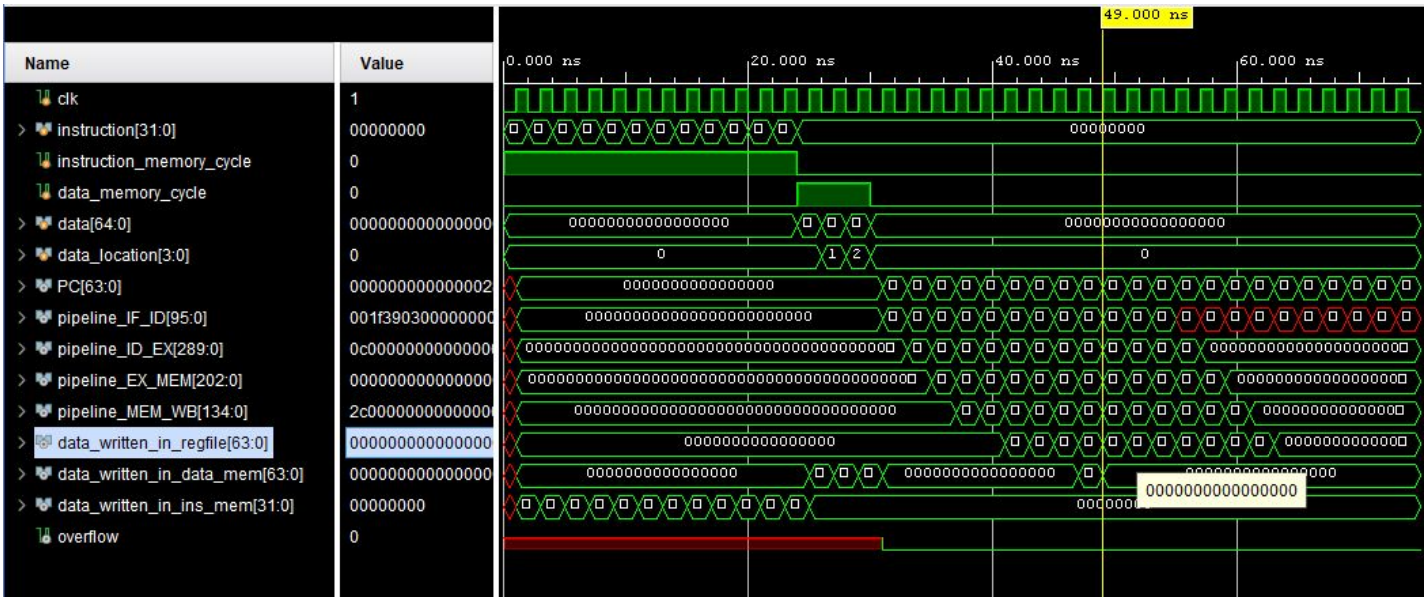


Instruction : **add x10, x28, x29.** so data_written_in_regfile = 45 + 0 = 45 . (2d in hexadecimal)

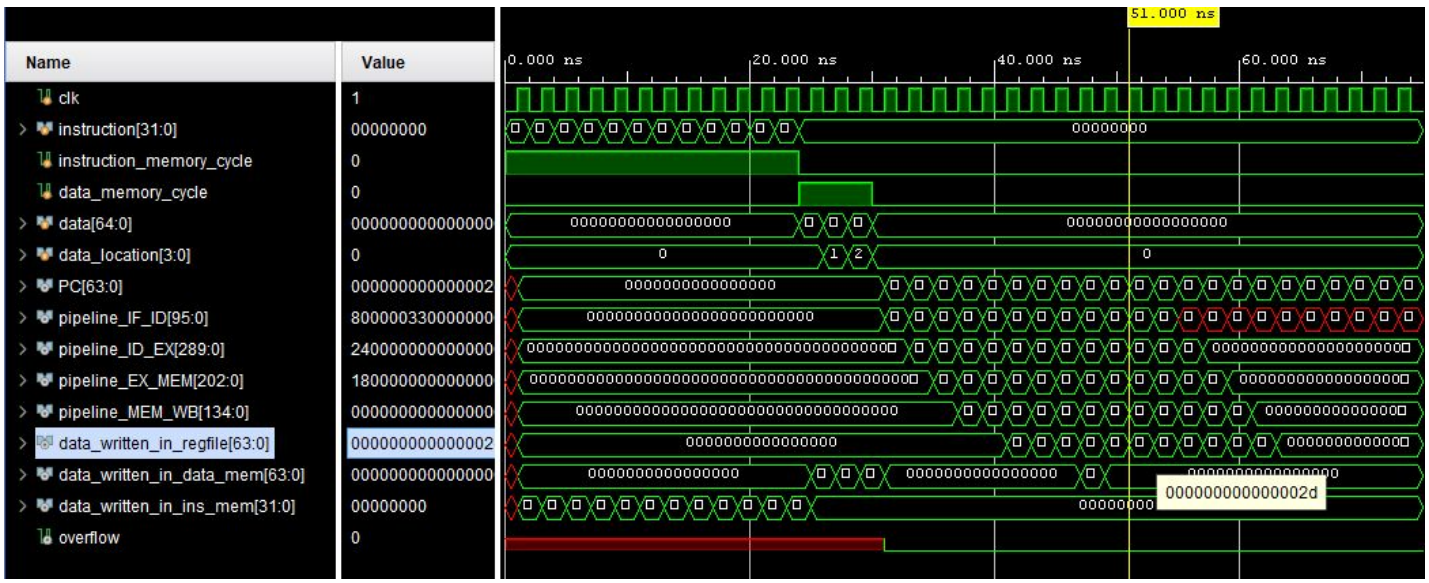


Instruction : **sd x10, 0(x29).** so data_written_in_datamem = 45(2d in hexadecimal):

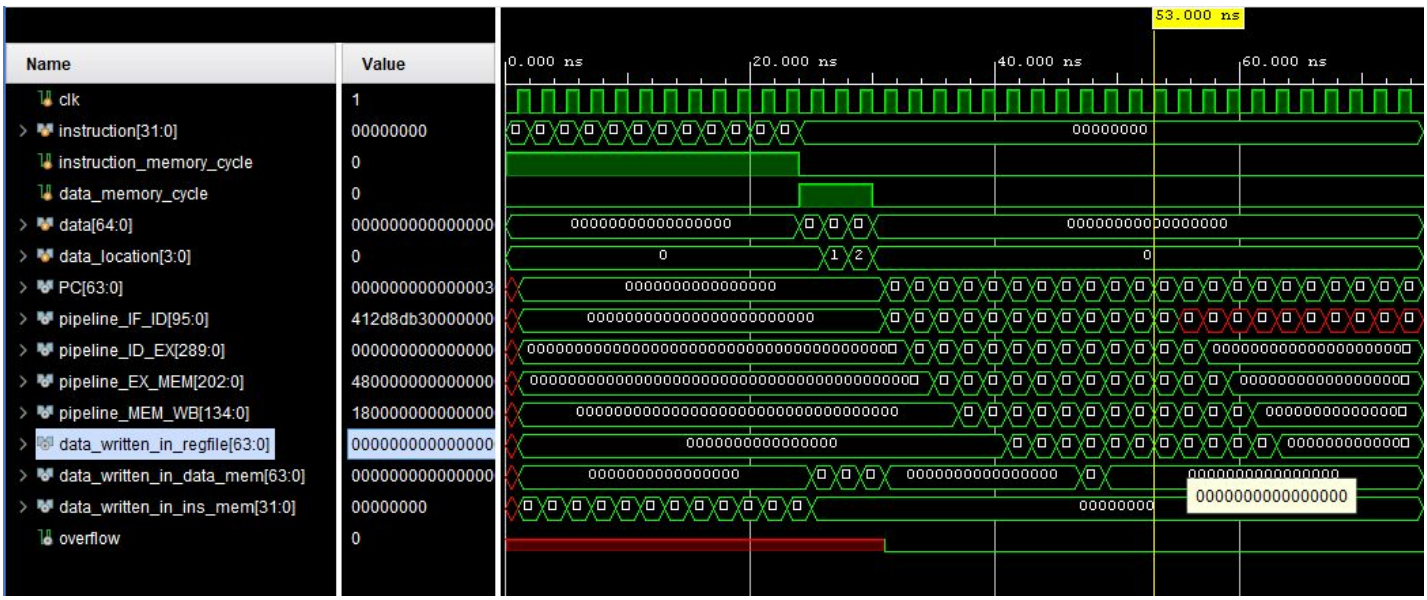
Instruction : **sd x10, 0(x29).** so nothing is written in register file.(so we got zero here)



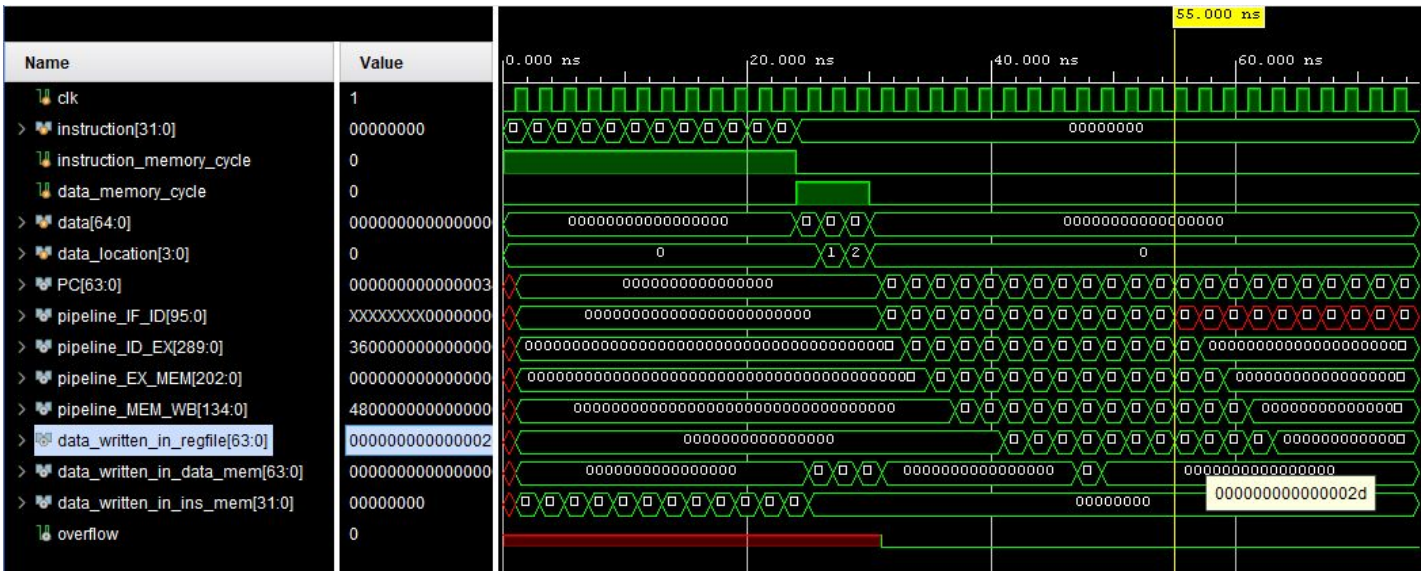
Instruction : **ld x11 0(x29).** so data_written_in_regfile = MEM[0] = 45 . (2d in hexadecimal)



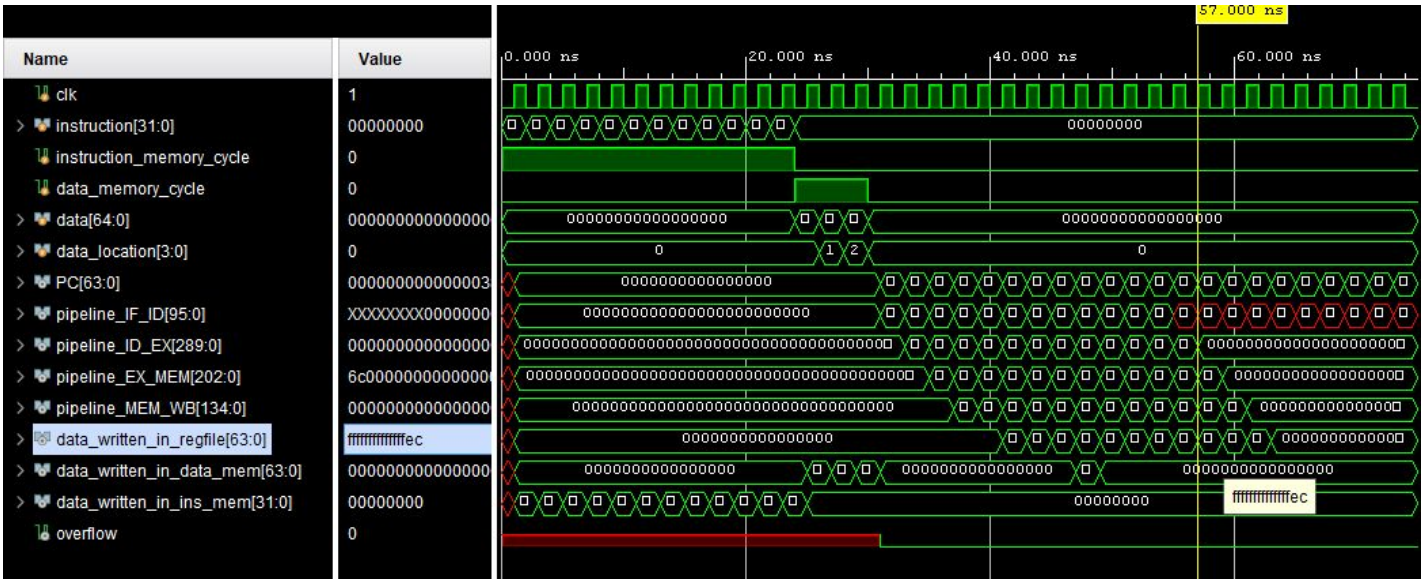
Instruction : **add x0, x0, x0. stall** so data_written_in_regfile = 0:



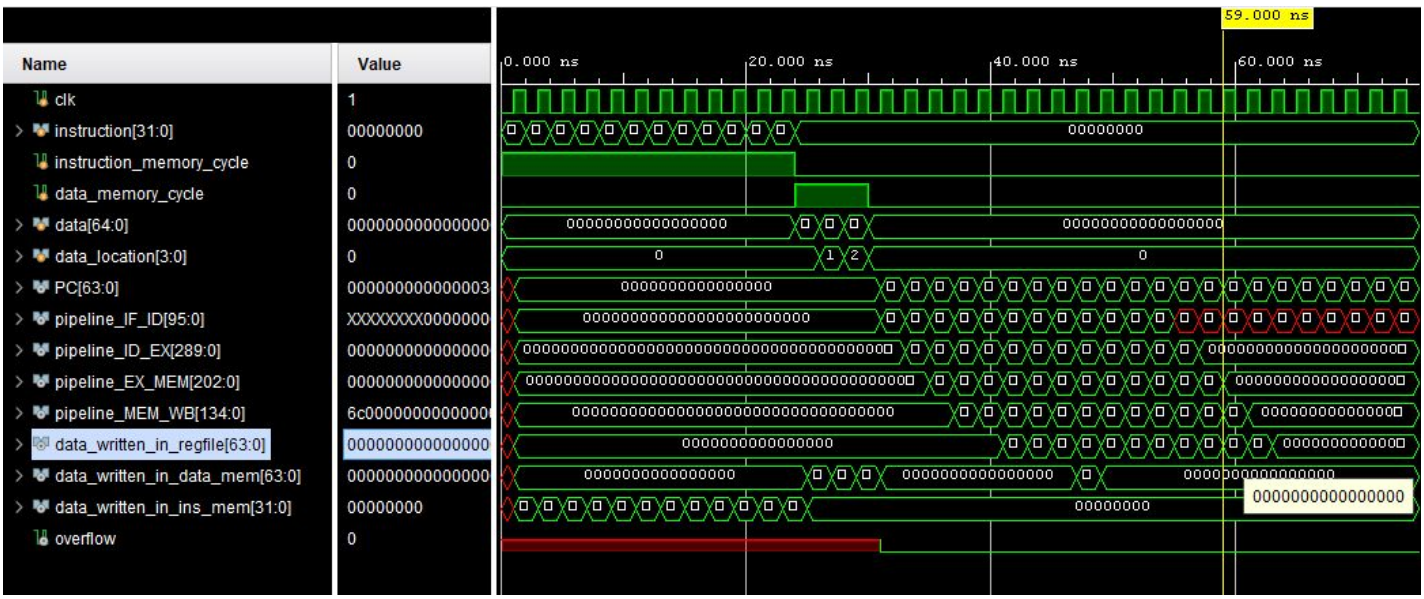
Instruction : **mul x6, x11, x31.** Data written in reg file = 45 * 1 = 45 [2d in hexadecimal]



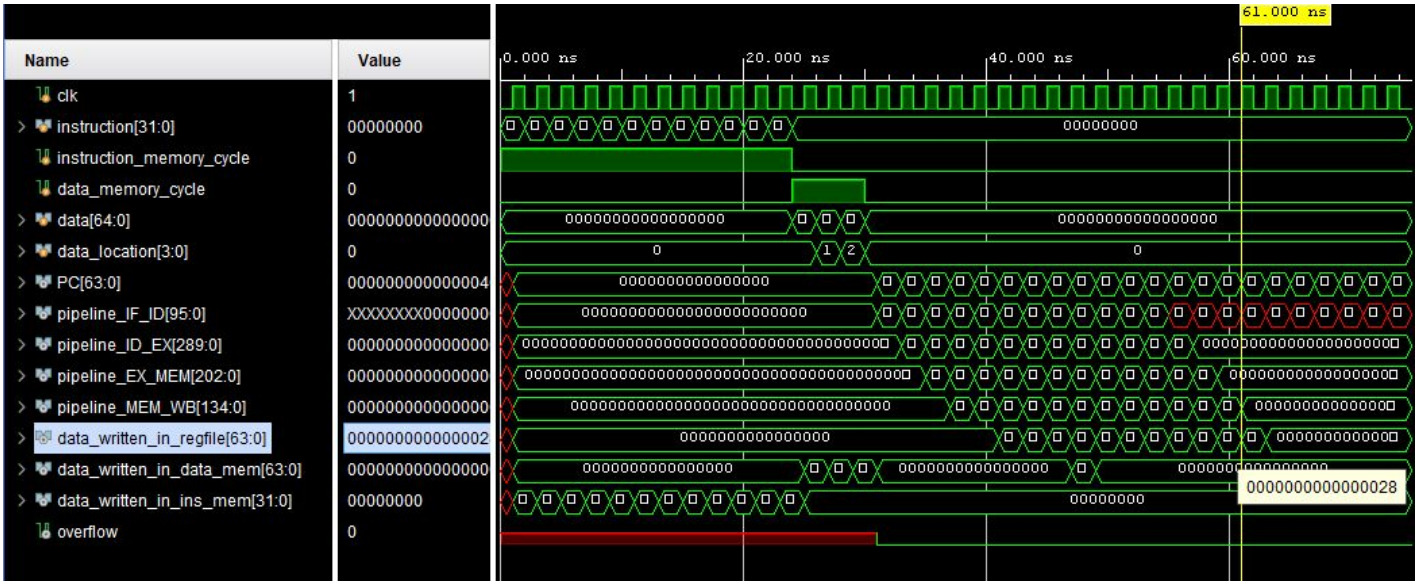
Instruction : **ld x18, 1(x30).** so data_written_in_regfile = MEM[2] = -20:



Instruction : **add x0, x0, x0. stall** so data_written_in_regfile = 0:



Instruction : **sub x19, x27, x18.** Data written in reg file = 20 – (-20) = 40 [28 in hexadecimal]



All the test cases are so far are passed successfully. If any further modifications are required it can be done by minor changes in Verilog code.

Time delays of written Verilog code:

Unconstrained Paths - NONE - NONE

Group Name: (none)

From Clock:

To Clock:

Statistics

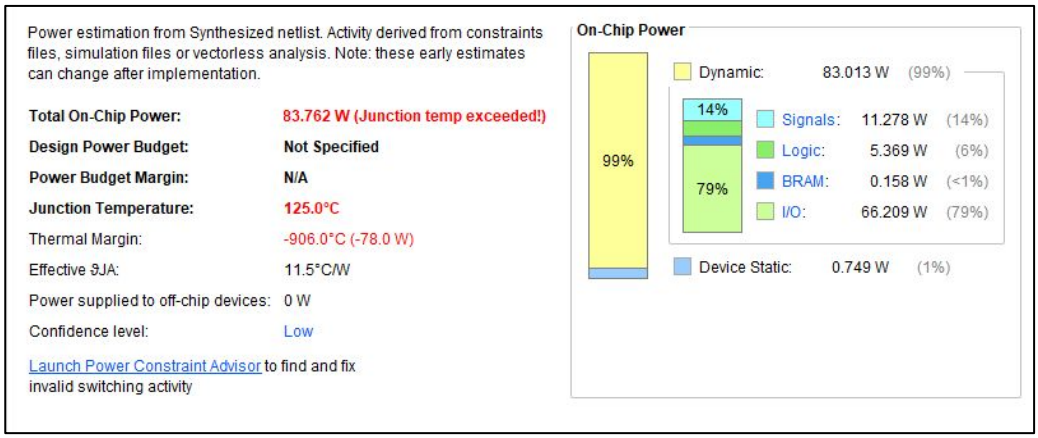
Type	Total Endpoints
Max Delay	5377
Min Delay	5377

Name	Slack	Levels	Routes	High Fanout	From	To	Total Delay	Logic Delay	Net Delay	Requirement	Source Clock	Destination Clock	Exception	Clock Uncertainty
Path 1	∞	143	143	161	pipeline_ID_EX_reg[216]/C	alu_result_reg[63]/D	120.448	56.604	63.844	∞				0.000
Path 2	∞	143	143	161	pipeline_ID_EX_reg[216]/C	pipeline_EX_MEM_reg[133]/D	120.448	56.604	63.844	∞				0.000
Path 3	∞	143	143	161	pipeline_ID_EX_reg[216]/C	alu_result_reg[62]/D	119.776	56.519	63.257	∞				0.000
Path 4	∞	143	143	161	pipeline_ID_EX_reg[216]/C	pipeline_EX_MEM_reg[132]/D	119.776	56.519	63.257	∞				0.000
Path 5	∞	141	141	161	pipeline_ID_EX_reg[216]/C	alu_result_reg[61]/D	117.809	55.673	62.136	∞				0.000
Path 6	∞	141	141	161	pipeline_ID_EX_reg[216]/C	pipeline_EX_MEM_reg[131]/D	117.809	55.673	62.136	∞				0.000
Path 7	∞	140	140	161	pipeline_ID_EX_reg[216]/C	alu_result_reg[60]/D	116.716	54.955	61.761	∞				0.000
Path 8	∞	140	140	161	pipeline_ID_EX_reg[216]/C	pipeline_EX_MEM_reg[130]/D	116.716	54.955	61.761	∞				0.000
Path 9	∞	137	137	161	pipeline_ID_EX_reg[216]/C	alu_result_reg[59]/D	114.012	53.970	60.042	∞				0.000
Path 10	∞	137	137	161	pipeline_ID_EX_reg[216]/C	pipeline_EX_MEM_reg[129]/D	114.012	53.970	60.042	∞				0.000

Name	Slack	Levels	Routes	High Fanout	From	To	Total Delay	Logic Delay	Net Delay	Requirement	Source Clock	Destination Clock	Exception	Clock Uncertainty
Path 11	∞	1	1	2	pipeline_ID_EX_reg[0]/C	pipeline_EX_MEM_reg[0]/D	0.286	0.141	0.145	-∞				0.000
Path 12	∞	1	1	2	pipeline_ID_EX_reg[285]/C	pipeline_EX_MEM_reg[198]/D	0.286	0.141	0.145	-∞				0.000
Path 13	∞	1	1	2	pipeline_ID_EX_reg[286]/C	pipeline_EX_MEM_reg[199]/D	0.286	0.141	0.145	-∞				0.000
Path 14	∞	1	1	2	pipeline_ID_EX_reg[287]/C	pipeline_EX_MEM_reg[200]/D	0.286	0.141	0.145	-∞				0.000
Path 15	∞	1	1	2	pipeline_ID_EX_reg[288]/C	pipeline_EX_MEM_reg[201]/D	0.286	0.141	0.145	-∞				0.000
Path 16	∞	1	1	2	pipeline_ID_EX_reg[289]/C	pipeline_EX_MEM_reg[202]/D	0.286	0.141	0.145	-∞				0.000
Path 17	∞	1	1	2	pipeline_ID_EX_reg[3]/C	pipeline_EX_MEM_reg[3]/D	0.286	0.141	0.145	-∞				0.000
Path 18	∞	1	1	2	pipeline_IF_ID_reg[1]/C	pipeline_ID_EX_reg[20]/D	0.286	0.141	0.145	-∞				0.000
Path 19	∞	1	1	2	pipeline_IF_ID_reg[2]/C	pipeline_ID_EX_reg[21]/D	0.286	0.141	0.145	-∞				0.000
Path 20	∞	1	1	2	pipeline_IF_ID_reg[3]/C	pipeline_ID_EX_reg[22]/D	0.286	0.141	0.145	-∞				0.000

Total critical path delay: 120.448 units.

Power consumption of written Verilog code:



Maximum Power Consumption is 83.762W.

- **Further extensions to make more realistic data path:**

1) Adding additional hardware at ID stage for branch instructions to reduce the number of stalls for control Hazards (or) maintaining a branch predictor static or dynamic branch predictor.
2) Increasing the number of stages so that on an average for many instructions the average performance can be increased significantly.