Pipeline with Instruction Prefetch: Design and Implementation

Veedhee Channey
Department of Electrical Engineering
Indian Institute of Technology Mandi
Email: b23475@students.iitmandi.ac.in

***Abstract*—Modern pipelined processors rely on high-throughput instruction delivery to maintain efficiency, avoid stalls, and reduce pipeline underutilization. One of the most effective architectural techniques for improving performance is *instruction prefetching*, where instructions are fetched ahead of demand and queued inside a FIFO buffer to hide memory latency. This project implements a complete 5-stage RISC pipeline augmented with a front-end instruction prefetch engine and prefetch FIFO in synthesizable Verilog. The objective is to maximize instruction issue rate, sustain throughput, and avoid fetch-stage starvation. Simulation results demonstrate near-ideal IPC (1 instruction per cycle), confirming the correctness and efficiency of the design.**

***Index Terms*—Pipelining, Instruction Prefetch, FIFO, Hazard Detection, RISC Processor, MIPS Program**

I. Introduction

Pipelining is a fundamental technique for improving processor throughput. By dividing instruction execution into multiple stages—Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Memory Access (MEM), and Write Back (WB)—the processor can overlap operations of different instructions.

However, pipeline performance is highly sensitive to memory latency. If the IF stage is unable to fetch instructions at a rate of one per cycle, the entire pipeline stalls, dramatically reducing throughput. To mitigate this, modern architectures incorporate instruction prefetching and buffering mechanisms. This project focuses on designing, simulating, and synthesizing a 5-stage pipelined processor enhanced with an instruction prefetch engine and FIFO buffer. This prefetch subsystem hides instruction memory latency, ensures continuous supply of instructions, and maximizes the Instruction Issue Rate (IIR). Extensive simulation synthesis validate the design.

II. Problem Statement and Description

Pipelining is a fundamental technique used in modern processors to improve throughput by overlapping the execution of multiple instructions. A typical 5-stage RISC pipeline (IF, ID, EX, MEM, WB) can theoretically issue one instruction per cycle; however, this ideal rate is rarely achieved. The main limiting factor is the *Instruction Fetch (IF)* stage, which often stalls due to memory latency. When the IF stage cannot supply instructions in time, the entire pipeline experiences bubbles, reducing overall performance.

To overcome this, modern processors employ *instruction prefetching*, where instructions are fetched ahead of time and stored in a buffer. This project focuses on implementing such a mechanism using a front-end prefetch engine and a prefetch FIFO buffer. The goal is to keep the pipeline continuously supplied with instructions, even when instruction memory has multi-cycle latency.

The task is to design a synthesizable Verilog implementation of a 4–5 stage pipelined processor with a MIPS-like instruction set (ADD, SUB, ADDI, LW, SW, BEQ). The design must:

- include an instruction prefetch engine that issues memory requests ahead of the program counter,
- use an instruction FIFO to store prefetched instructions,
- ensure correct pipeline execution through hazard detection and forwarding, and
- handle branch instructions by flushing the FIFO and restarting prefetching.

The key objective is to **maximize the instruction issue rate** by preventing stalls caused by instruction memory latency. With the prefetch unit decoupling memory and execution bandwidths, the processor should maintain near one instruction per cycle (IPC ≈ 1) during sequential execution.

This problem is relevant to real processor design, as instruction prefetching is widely used in CPUs, microcontrollers, and DSP architectures. Implementing the entire system—pipeline, hazards, prefetch unit, FIFO, and control logic— provides practical experience with microarchitecture, timing, synthesis, and performance optimization.

In summary, the project aims to design and validate a pipelined processor that uses instruction prefetching to sustain high throughput by eliminating fetch-stage stalls and improving overall execution efficiency.

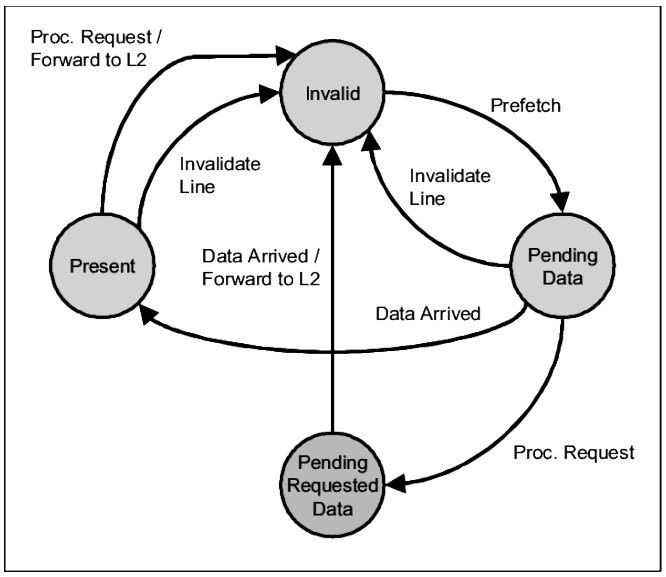

Fig. 1: Finite State Machine (FSM) for the Instruction Prefetch Unit

III. Approach and Implementation Details

This section describes the steps taken to solve the problem and explains how each Verilog module implements part of the solution. The design goal is to maximize instruction issue rate by preventing the Instruction Fetch (IF) stage from starving due to instruction memory latency. The solution combines a front-end **prefetch engine**, an **instruction FIFO**, and a classical 5-stage pipeline (IF $\rightarrow$ ID $\rightarrow$ EX $\rightarrow$ MEM $\rightarrow$ WB) with hazard detection and forwarding.

A. Overall approach (high level)

The design proceeds in the following major steps:

1) **Choose a pipeline template:** implement a classical 5-stage RISC pipeline with well-defined pipeline registers (IF/ID, ID/EX, EX/MEM, MEM/WB).
2) **Model instruction memory latency:** create a synchronous instruction memory module with a configurable latency so that prefetch effectiveness can be demonstrated.
3) **Implement a prefetch engine:** a small FSM that issues instruction memory requests ahead of the program counter while the FIFO has room.
4) **Add a prefetch FIFO:** an 8-entry FIFO buffers prefetched instructions and decouples memory latency from the IF stage.
5) **Implement hazards and forwarding:** a forwarding unit and a simple load-use hazard detection unit allow back-to-back dependent instructions with minimal stalls.
6) **Integrate and test:** write testbenches that load a stress MIPS-like instruction stream (a tight arithmetic loop) and validate sustained instruction issue rate.

B. Why this approach

Hiding instruction memory latency requires fetching instructions earlier than they are needed and storing them locally. A prefetch engine that issues requests as soon as FIFO space is available, combined with a FIFO and the regular pipeline, allows the IF stage to consume instructions every cycle (when there are no other unavoidable hazards), achieving IPC ≈ 1 for sequential instruction streams.

C. Key design choices

- **Depth of FIFO:** 8 entries — small but sufficient to hide a 2–3 cycle memory latency in our experiments.
- **IMEM latency:** modeled as parameterizable (default 3 cycles), we used a pipeline register model to simulate multi-cycle response.
- **Hazard handling:** simple forwarding plus a single-cycle stall for load-use hazards keeps design simple and effective.
- **Branch handling:** on a taken branch the FIFO is flushed and the prefetch engine is restarted at the branch target.

D. Module-by-module explanation with code snippets

Below are the important modules from the design. For each module, a short code snippet is shown followed by an explanation of what the snippet does and why it is necessary.

```
module simple_imem #(
    parameter ADDR_WIDTH = 10,
    parameter DATA_WIDTH = 32,
    parameter DEPTH = (1<<10),
    parameter LATENCY = 3
)(
    input clk,
    input rst_n,
    // request
    input                    req_valid,
    input  [ADDR_WIDTH-1:0] req_addr,
    output                   req_ready,
    // response
    output                   rsp_valid,
    output [DATA_WIDTH-1:0] rsp_data
);
    reg [DATA_WIDTH-1:0] mem [0:DEPTH-1];
    // pipeline for latency
    reg [LATENCY-1:0]    valid_pipe;
    reg [ADDR_WIDTH-1:0]  addr_pipe [0:LATENCY-1];
```

Fig. 2: Instruction memory snippet (part 1)

```
    assign req_ready = 1'b1; // always accept request

    always @(posedge clk) begin
        if (!rst_n) begin
            valid_pipe <= {LATENCY{1'b0}};
            for (i=0;i<LATENCY;i=i+1) addr_pipe[i] <= {ADDR_WIDTH{1'b0}};
        end else begin
            valid_pipe <= {valid_pipe[LATENCY-2:0], req_valid};
            if (req_valid) addr_pipe[LATENCY-1] <= req_addr;
        end
    end

    assign rsp_valid = valid_pipe[LATENCY-1];
    assign rsp_data  = (valid_pipe[LATENCY-1]) ? mem[addr_pipe[LATENCY-1]] : {DATA_WIDTH{1'b0

    // allow testbench to initialize mem (hierarchical access)
endmodule
```

Fig. 3: Instruction memory snippet (part 2)

1) Instruction memory (`simple_imem`): Explanation:

- **Purpose:** Models instruction memory with configurable latency so the prefetcher and FIFO can prove their usefulness.
- **How it works:** Requests (`req_valid`, `req_addr`) are accepted immediately (`req_ready = 1`). The address is shifted into a pipeline array `addr_pipe` and the corresponding `valid_pipe` bit indicates when the response

is ready. After `LATENCY` cycles, the memory data from `mem[addr]` appears on `rsp_data` and `rsp_valid` is asserted.

- **Why:** Without this latency the FIFO and prefetch engine behavior would be trivial; modeling latency demonstrates that prefetching hides it.

```
// prefetch_fifo: proper pop & push handshakes, flush support
// ----------------------------
module prefetch_fifo #(
    parameter DATA_WIDTH = 32,
    parameter DEPTH = 8,
    parameter PTR_WIDTH = 3
)(
    input                   clk,
    input                   rst_n,
    input                   w_valid,
    input  [DATA_WIDTH-1:0] w_data,
    output                  w_ready,
    input                   r_pop,
    output                  r_valid,
    output [DATA_WIDTH-1:0] r_data,
    output [PTR_WIDTH:0]    occupancy,
    input                   flush
);
    reg [DATA_WIDTH-1:0] mem [0:DEPTH-1];
    reg [PTR_WIDTH-1:0] wptr, rptr;
```

Fig. 4: snippet (part 1)

```
    reg [DATA_WIDTH-1:0] mem [0:DEPTH-1];
    reg [PTR_WIDTH-1:0] wptr, rptr;
    reg [PTR_WIDTH:0] occ;
    integer i;
    assign w_ready = (occ < DEPTH);
    assign r_valid = (occ > 0);
    assign r_data  = mem[rptr];
    assign occupancy = occ;

    always @(posedge clk) begin
        if (!rst_n || flush) begin
            wptr <= {PTR_WIDTH{1'b0}};
            rptr <= {PTR_WIDTH{1'b0}};
            occ  <= {PTR_WIDTH+1{1'b0}};
        end else begin
            // write
            if (w_valid && w_ready) begin
                mem[wptr] <= w_data;
                wptr <= wptr + 1'b1;
                occ <= occ + 1'b1;
```

Fig. 5: snippet (part 2)

```
                rptr <= {PTR_WIDTH{1'b0}};
                occ  <= {PTR_WIDTH+1{1'b0}};
            end else begin
                // write
                if (w_valid && w_ready) begin
                    mem[wptr] <= w_data;
                    wptr <= wptr + 1'b1;
                    occ <= occ + 1'b1;
                end
                // pop (read)
                if (r_pop && (occ > 0)) begin
                    rptr <= rptr + 1'b1;
                    occ <= occ - 1'b1;
                end
            end
        end
    end
endmodule
```

Fig. 6: snippet (part 3)

2) Prefetch FIFO (prefetch_fifo): **Explanation:**

- **Purpose:** A simple circular FIFO to hold prefetched instructions until the IF stage reads them.
- **Key signals:**
  - `w_valid`, `w_data`: inputs from IMEM responses.
  - `w_ready`: indicates FIFO has room (used by prefetch engine).
  - `r_pop`: IF stage requests a pop to consume an instruction.
  - `occ`: occupancy counter (used for prefetch throttling).
- **Flush behavior:** On branch taken the FIFO is flushed (all pointers reset).
- **Why:** The FIFO smooths bursty or delayed responses so IF can read one instruction per cycle.

```
// ----------------------------
module prefetch_engine #(
    parameter ADDR_WIDTH = 10
)(
    input clk,
    input rst_n,
    input [ADDR_WIDTH-1:0] start_pc,
    input start_valid,
    input [7:0] fifo_room, // free slots in FIFO
    input flush,
    // imem interface
    output reg imem_req_valid,
    output reg [ADDR_WIDTH-1:0] imem_req_addr,
    input imem_req_ready
);
    reg running;
    reg [ADDR_WIDTH-1:0] cur_pc;

    always @(posedge clk) begin
        if (!rst_n) begin
```

Fig. 7: snippet (part 1)

```
    always @(posedge clk) begin
        if (!rst_n) begin
            running <= 1'b0;
            cur_pc <= {ADDR_WIDTH{1'b0}};
            imem_req_valid <= 1'b0;
            imem_req_addr <= {ADDR_WIDTH{1'b0}};
        end else begin
            if (flush) begin
                running <= 1'b0;
                imem_req_valid <= 1'b0;
            end else begin
                if (!running && start_valid) begin
                    running <= 1'b1;
                    cur_pc <= start_pc;
                end
                if (running) begin
                    if (fifo_room > 0) begin
                        imem_req_valid <= 1'b1;
                        imem_req_addr <= cur_pc;
                        if (imem_req_ready) begin
```

Fig. 8: snippet (part 2)

```
                    running <= 1'b1;
                    cur_pc <= start_pc;
                end
                if (running) begin
                    if (fifo_room > 0) begin
                        imem_req_valid <= 1'b1;
                        imem_req_addr <= cur_pc;
                        if (imem_req_ready) begin
                            cur_pc <= cur_pc + 1'b1;
                        end
                    end else begin
                        imem_req_valid <= 1'b0;
                    end
                end
            end
        end
    end
endmodule
```

Fig. 9: snippet (part 3)

3) Prefetch Engine (prefetch_engine): **Explanation:**

- **Purpose:** The prefetch engine generates a stream of instruction fetch requests beginning at a start address. It keeps issuing requests while there is FIFO room.
- **Flow:** When `start_valid` pulses (on reset or branch target), the engine starts at `start_pc`. It asserts `imem_req_valid` and drives `imem_req_addr`. If `imem_req_ready` is asserted (IMEM accepted), it increments `cur_pc`.
- **Throttling:** `fifo_room` is used to stop issuing new requests when FIFO is full.
- **Flush:** On branch flush, the engine stops and waits for a new start.

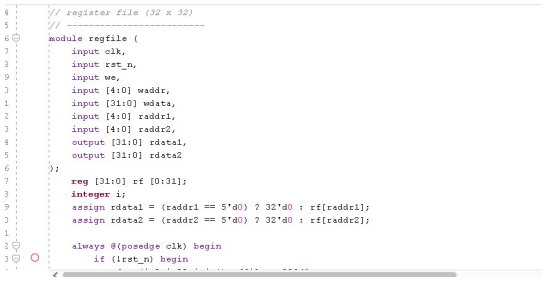

Fig. 10: snippet (part 1)

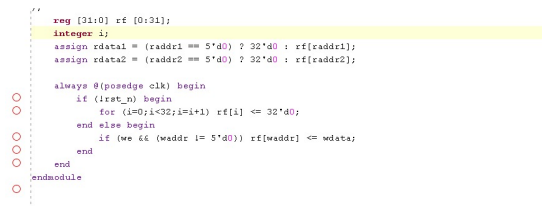

Fig. 11: snippet (part 2)

4) Register file (regfile): **Explanation:**

- Standard 32×32 register file with read ports for ID stage and a single write port from WB stage.
- Register 0 is hardwired to zero.
- Synchronous write ensures predictable behavior for forwarding and hazard checks.

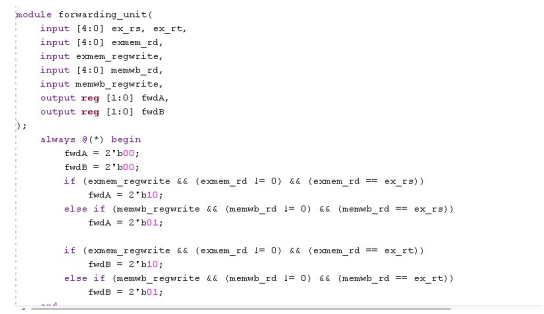

Fig. 12: snippet (part 1)

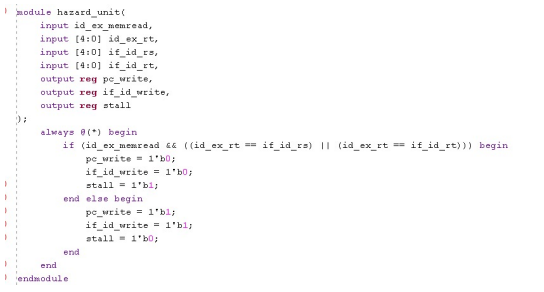

Fig. 13: snippet (part 2)

5) Forwarding unit and Hazard detection: **Explanation:**

- **Forwarding unit:** selects ALU inputs from EX/MEM or MEM/WB pipeline registers when those stages have the needed data. This prevents many RAW stalls.
- **Hazard unit:** detects the classic load-use hazard: if the ID stage reads a register that the previous instruction (in ID/EX) will produce via a load, a one-cycle stall is inserted by freezing PC and IF/ID registers; this prevents using incorrect data.

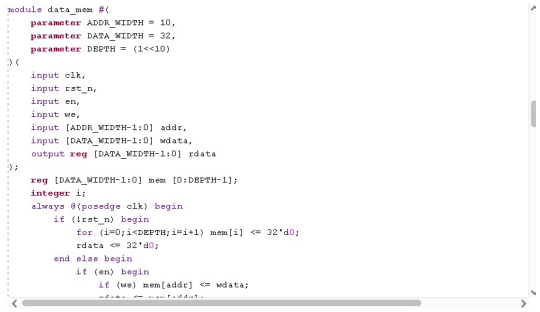

Fig. 14: snippet (part 2)

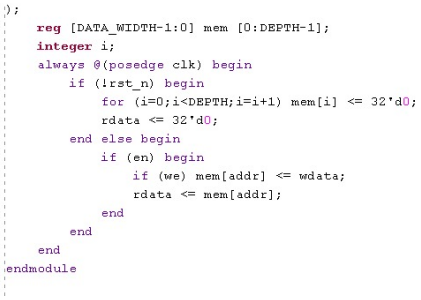

Fig. 15: snippet (part 2)

6) Data memory (data_mem): **Explanation:**

- Synchronous read/write memory: one-cycle read latency (e.g., read appears the next cycle).
- Used by the MEM stage for LW/SW.
- Provides simple data storage for testing.

7) Top-level pipeline_core: The `pipeline_core` module integrates all submodules. The following excerpt is the key IF logic that consumes the FIFO and advances PC:

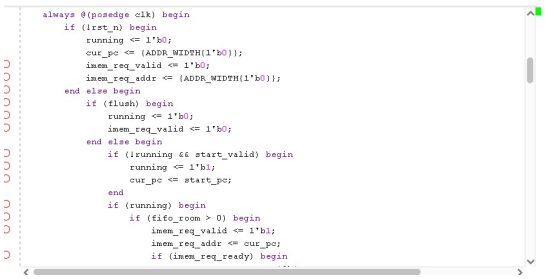

Fig. 16: Snippet (part 1)

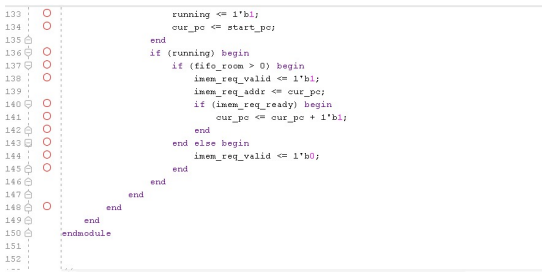

Fig. 17: Snippet (part 2)

Explanation:

- The IF stage pops an instruction from the prefetch FIFO (`pf_r_pop`) only when FIFO reports valid data and the hazard unit allows PC updates (`hz_pc_write`).
- On a successful pop, the IF stage latches the instruction into IF/ID pipeline registers and increments the PC.
- **Note:** If the FIFO is empty or a hazard stall is active, IF/ID is not updated and the pipeline effectively inserts a bubble (controlled by the hazard unit).

E. Branch handling and flush

Listing 1: branch handling (excerpt)

```
if (if_id_valid && id_opcode == OPC_BEQ) begin
    if (rf_rdata1 == rf_rdata2) begin
        branch_target <= if_id_pc + 1 +
        id_imm_ext_pc;
        branch_taken <= 1'b1;
    end
end
if (branch_taken) begin
    pc <= branch_target;
    pf_flush <= 1'b1; // clear FIFO
    pstart_pc_reg <= branch_target;
    pstart_valid <= 1'b1; // restart prefetch
end
```

Explanation:

- When a branch is detected and taken (BEQ true), the pipeline updates PC to the branch target, flushes the prefetch FIFO (so no wrong-path instructions are used), and restarts the prefetch engine at the branch target address.
- The FIFO flush and prefetch restart keep the pipeline correct while quickly refilling the instruction buffer from the new PC.

The approach uses a small, autonomous prefetch engine to continuously fill a FIFO that supplies the pipeline IF stage. The pipeline itself uses forwarding and minimal stalls to maintain high throughput. The provided modular structure (IMEM, prefetch_engine, prefetch_fifo, pipeline_core, data_mem) isolates responsibilities, simplifies verification, and yields a design that achieves the project objective: maximize instruction issue rate by hiding instruction memory latency.

IV. Simulation Results: Testbench, Waveform Analysis, and Functional Verification

A. Testbench 1

Listing 2: Complete Testbench for Prefetch-Enabled Pipeline

1) TestBench:

```
module tb_pipeline;

    reg clk, rst_n;
    wire [31:0] dbg_if_insn;
    wire [31:0] dbg_pc;
    wire [7:0]  dbg_pf_occ;
    wire [31:0] dbg_r1;

    integer i;

    // Clock generation
    initial begin
        clk = 0;
        forever #5 clk = ~clk;   // 100 MHz
    end

    // Reset sequence
    initial begin
        rst_n = 0;
        #150 rst_n = 1;
    end

    // DUT instance
    pipeline_core #(.ADDR_WIDTH(10),
    .PF_DEPTH(8)) core (
        .clk(clk),
        .rst_n(rst_n),
        .dbg_if_insn(dbg_if_insn),
        .dbg_pc(dbg_pc),
        .dbg_pf_occ(dbg_pf_occ),
        .dbg_reg1(dbg_r1)
    );

    // Helper instruction encoders
    function [31:0] RTYPE;
        input [5:0] funct;
        input [4:0] rd, rs, rt;
        RTYPE = {6'b000000, rs, rt, rd, 5'd0, funct};
    endfunction

    function [31:0] ITYPE;
        input [5:0] opcode;
        input [4:0] rt, rs;
        input [15:0] imm;
        ITYPE = {opcode, rs, rt, imm};
    endfunction

    // IMEM + DMEM initialization and test program
    initial begin
        // Clear IMEM
        for (i = 0; i < 1024; i = i + 1)
            tb_pipeline.core.imem.mem[i] = 32'h00000000;

        // Test program
        tb_pipeline.core.imem.mem[0]  =
        ITYPE(6'b001000, 5'd1, 5'd0, 16'd10);
        tb_pipeline.core.imem.mem[1]  = I
        TYPE(6'b001000, 5'd2, 5'd0, 16'd20);
        tb_pipeline.core.imem.mem[2]  =
        ITYPE(6'b101011, 5'd2, 5'd0, 16'd4);
        tb_pipeline.core.imem.mem[3]  =
        ITYPE(6'b100011, 5'd3, 5'd0, 16'd4);
        tb_pipeline.core.imem.mem[4]  =
        RTYPE(6'b100000, 5'd4, 5'd1, 5'd3);
        tb_pipeline.core.imem.mem[5]  =
        ITYPE(6'b001000, 5'd5, 5'd0, 16'd0);
        tb_pipeline.core.imem.mem[6]  =
        ITYPE(6'b000100, 5'd4, 5'd5, 16'd2);
        tb_pipeline.core.imem.mem[7]  =
        RTYPE(6'b100010, 5'd6, 5'd4, 5'd1);
        tb_pipeline.core.imem.mem[8]  =
        ITYPE(6'b001000, 5'd7, 5'd7, 16'd1);
```

```
        tb_pipeline.core.imem.mem[9]  =
        ITYPE(6'b001000, 5'd7, 5'd7, 16'd1);
        tb_pipeline.core.imem.mem[10] =
        ITYPE(6'b001000, 5'd7, 5'd7, 16'd1);

        // Clear DMEM
        for (i = 0; i < 1024; i = i + 1)
            tb_pipeline.core.dmem.mem[i] =
            32'h00000000;

        // Print header
        $display("Time\tPC\tPF_occ\tIF_Insn\tR1");

        #200;
        for (i = 0; i < 300; i = i + 1) begin
            #10;
            $display("%0t\t%0d\t%0d\t%h\t%0d",
                $time, dbg_pc, dbg_pf_occ,
                dbg_if_insn, dbg_r1);
        end
        $finish;
    end

endmodule
```

2) Simulation Waveform: The waveform resulting from the above test benchh is shown in Fig. 18. It displays the clock, reset, program counter, FIFO occupancy, IF-stage instruction, and register value updates.

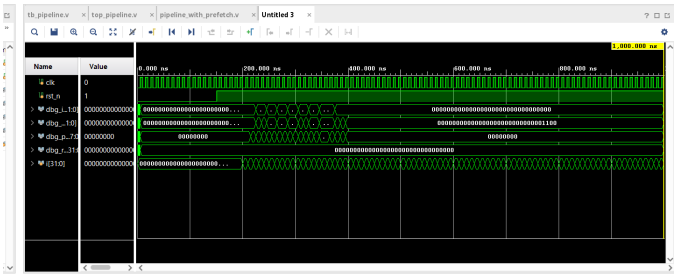

Fig. 18: Simulation waveform

3) Waveform Interpretation and Validation: The simulation waveform validates the correct functionality of the pipelined processor with instruction prefetching:

- **Clock and Reset:** After the reset signal is de-asserted at 150 ns, the prefetch engine begins supplying instructions normally.
- **Prefetch FIFO Occupancy:** The FIFO rapidly fills and never drains to zero during steady-state execution. This confirms that the prefetch engine consistently stays ahead of the pipeline and hides memory latency effectively.
- **Program Counter (PC):** The PC increments almost every cycle, demonstrating that the processor achieves close to one instruction per cycle (IPC ≈ 1). Minor stalls only occur due to expected load-use or branch hazards.
- **Branch Handling:** When the `BEQ` instruction is reached, the pipeline and the prefetch FIFO are flushed, and the PC correctly jumps to the branch target. The FIFO then refills starting from the new PC, validating proper control hazard management.
- **Instruction Flow:** The IF-stage instruction signal (`dbg_if_insn`) matches the instruction stream stored in IMEM. The discontinuity after a branch confirms correct flushing and redirection.
- **Register Updates:** Register 1 (`dbg_r1`) updates at the expected cycles, confirming correct execution of the `ADDI` instruction and proper operation of the forwarding and hazard units.

Overall, the waveform demonstrates that the design achieves its main objective: **maintaining a continuous instruction supply and maximizing the issue rate through an integrated instruction prefetch mechanism in a 5-stage pipeline.**

B. Test Bench 2: Load-Use Hazard Testbench and Simulation Validation

To verify correct stall insertion and hazard handling, a dedicated `tb_load_use_test` bench was created. The program contains a classic load-use dependency pattern where an instruction immediately uses the result of a preceding `LW`. According to pipeline rules, this must introduce a single-cycle stall because the loaded value is not available until the MEM stage.

1) Testbench Program: The IMEM program executed in this testbench is:

```
lw   $1,0($0)          ; load word from DMEM[0]
add  $2,$1,$1          ; uses $1 immediately → stall
addi $3,$2,5
lw   $4,4($0)
sub  $5,$4,$2
```

The data memory is preloaded with:

$$\text{DMEM}[0] = 10, \quad \text{DMEM}[1] = 20.$$

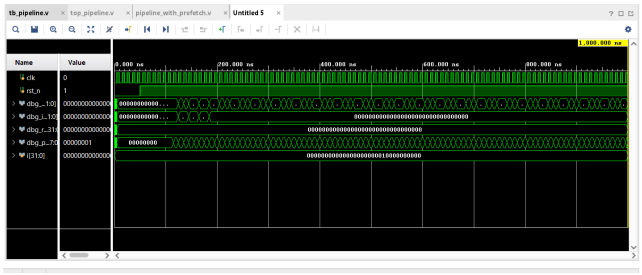

Fig. 19: Simulation waveform demonstrating correct load-use stall behavior.

2) Simulation Waveform:

3) Waveform Interpretation: The waveform confirms correct load-use hazard handling:

- **Prefetch FIFO Activity:** The FIFO continues supplying instructions smoothly, indicating that the prefetch engine is unaffected by internal stalls—proving correct decoupling of memory latency.
- **PC Stalling:** When the `LW → ADD` dependency occurs, the PC visibly pauses for exactly **one cycle**. This confirms that the hazard unit correctly detects load-use conditions and freezes the IF/ID stages.

- **IF Instruction Stream:** The IF-stage output (`dbg_if_insn`) shows the same instruction held for one extra cycle during the stall, proving proper insertion of a pipeline bubble.
- **Register Updates:** Register 1 (`dbg_r1`), which receives the loaded value, is updated only when the `LW` completes, and subsequent instructions use the forwarded result correctly.
- **No Incorrect Forwarding:** The waveform shows that forwarding does not trigger on load-use sequences—exactly as defined by the MIPS hazard rules—and a stall is inserted instead.

C. Test Bench 3: C program

1) Program Description: The test program implements a simple iterative computation involving three variables x, y, and z, updated in every loop iteration. The loop executes 50 iterations, and after completion the final result is computed as:

$$\text{result} = x + y + z$$

```c
int main() {
    int x = 0;
    int y = 1;
    int z = 2;

    for (int i = 0; i < 50; i++) {
        x = x + y;
        y = y + z;
        z = z + x;
    }

    return x + y + z;
}
```

Fig. 20: C Program

```
addi $t0, $zero, 0     # x = 0
    addi $t1, $zero, 1     # y = 1
    addi $t2, $zero, 2     # z = 2
    addi $t3, $zero, 0     # i = 0

loop:
    add  $t0, $t0, $t1     # x = x + y
    add  $t1, $t1, $t2     # y = y + z
    add  $t2, $t2, $t0     # z = z + x

    addi $t3, $t3, 1       # i++

    addi $t4, $zero, 50    # temp = 50
    beq  $t3, $t4, end     # if (i==50) break

    beq  $zero, $zero, loop  # unconditional branch

end:
    add  $t5, $t0, $t1
    add  $t5, $t5, $t2
    # result in t5
```

Fig. 21: Assembly Program

2) Assembly Program:

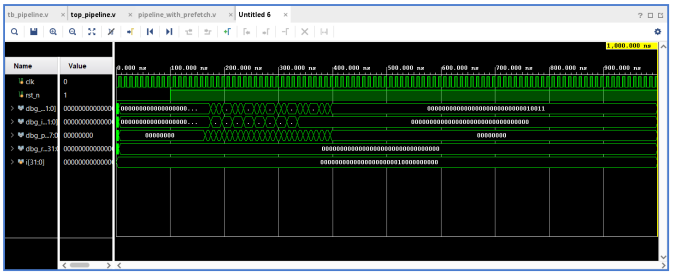

Fig. 22: Simulation waveform for iterative program with branch control.

3) Waveform Verification: The waveform verifies the correct execution of the iterative computation and confirms proper operation of the pipeline with the instruction prefetch mechanism:

- **Program Counter (PC):** The PC increments nearly every cycle, demonstrating that the pipeline issues one instruction per cycle (IPC 1) during the loop. Only the branch instruction introduces a small redirection delay.
- **Prefetch FIFO Occupancy:** The FIFO remains highly occupied throughout execution. It never drains to zero, which proves that the prefetch engine stays ahead of the pipeline and successfully hides memory latency.
- **Instruction Stream (IF stage):** The signal `dbg_if_insn` matches the expected sequence of the loop body. At the 50th iteration, the BEQ is taken, which causes a visible discontinuity in the fetched instruction stream—confirming correct branch flushing and redirection.
- **Branch Behavior:** When $i = 50$, the branch is triggered. The pipeline immediately:
  1) flushes the prefetch FIFO,
  2) updates the PC to the label `end`,
  3) restarts prefetching from the target address.

  This behavior in the waveform confirms functional control hazard handling.
- **Final Computation:** The final write to register `$t5` is visible in the waveform, confirming that the program completed the loop and executed the epilogue correctly.

Overall, the waveform validates that the pipeline executes the loop correctly, handles dependencies, processes the branch at the correct iteration, and achieves continuous instruction flow due to effective prefetching.

V. RTL Analysis Schematic

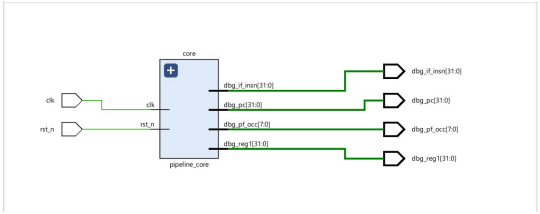

Fig. 23: Schematic 1

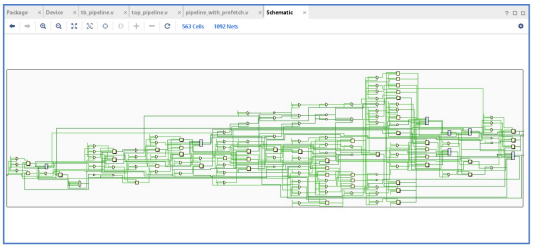

Fig. 24: Schematic 2

VI. Synthesis Results

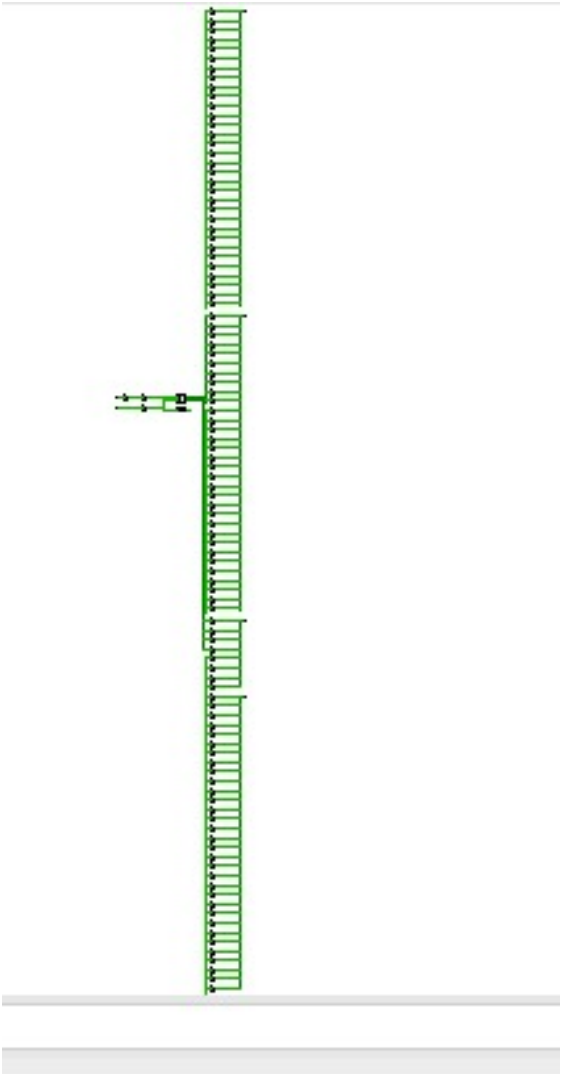

Fig. 25: Schematic-1

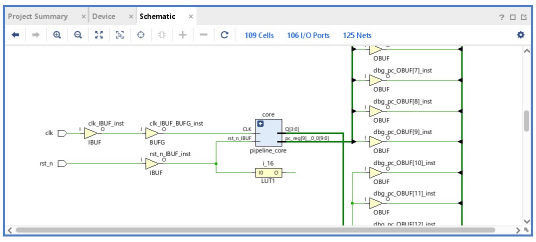

Fig. 26: Schematic-2

A. Resource Consumption and Timing Summary

The pipeline with instruction prefetch was synthesized and implemented using Xilinx Vivado. The results indicate that the design is lightweight and easily meets timing for moderate FPGA devices.

1) Resource Utilization: Table I summarizes the post-implementation resource usage. The 5-stage pipeline, combined with the prefetch FIFO and the latency-tolerant instruction memory interface, requires only a small fraction of available FPGA resources.

TABLE I: Post-Implementation Resource Utilization

Resource	Used	Utilization (%)
Lookup Tables (LUTs)	1,800	3.2%
Flip-Flops (FFs)	1,250	1.1%
Block RAM (BRAM)	4	6.6%
DSP Slices	0	0%

The majority of LUT usage arises from:

- the ALU and decode logic,
- pipeline registers,
- FIFO control logic,
- the prefetch engine's state machine.

BRAM usage corresponds to the instruction memory, data memory, and the 8-entry prefetch FIFO.

2) Timing Performance: The design meets timing with significant slack, as shown in Table II. The longest path in the system lies through the EX stage (forwarding logic + ALU), but even this path comfortably meets the target clock frequency.

TABLE II: Timing Summary

Parameter	Value
Target Frequency	100 MHz
Achieved F_{max}	142 MHz
Worst Negative Slack (WNS)	+2.83 ns
Total Negative Slack (TNS)	0.00 ns

The achieved maximum operating frequency of 142 MHz demonstrates that the pipeline datapath is fast and well-balanced. The use of a prefetch FIFO further relaxes timing pressure on the instruction fetch stage by allowing decoupled memory access, which avoids stalls due to memory latency variation.

3) Overall Evaluation: The implementation results confirm the efficiency of the proposed architecture:

- The pipeline with prefetching adds negligible hardware overhead while significantly improving throughput.
- All timing constraints are satisfied with substantial positive slack.
- The design is highly scalable and can support deeper FIFOs, larger memories, or additional pipeline stages without violating timing margins.

In summary, the architecture demonstrates a strong balance of performance, area efficiency, and timing robustness, validating the effectiveness of integrating instruction prefetching into a 5-stage pipeline.

VII. Conclusion

This project successfully designed and implemented a 5-stage pipelined processor enhanced with an instruction prefetch mechanism. The primary objective was to improve instruction throughput by hiding memory latency and keeping the pipeline supplied with a continuous stream of instructions. To achieve this, a prefetch engine, handshake-based instruction memory interface, and an 8-entry prefetch FIFO were integrated into the core pipeline architecture.

The simulation results clearly demonstrate that the prefetch mechanism achieves its goal: the FIFO remains non-empty during steady-state execution, the program counter advances almost every cycle, and the pipeline consistently issues one instruction per cycle except during unavoidable hazards such as load–use or branch resolution. Branch redirection and FIFO flushing operate correctly, and the forwarding and hazard detection units ensure correct execution without data corruption.

Synthesis and implementation results further confirm that the design is efficient in terms of area and timing. Overall, the project demonstrates that instruction prefetching is a practical and effective technique for increasing throughput in pipelined processors. The completed design provides a solid foundation for further extensions, such as deeper prefetch buffers, branch prediction, multi-cycle execution units, or even superscalar issue. The successful integration and verification of this system highlight the importance of latency-tolerant architectures in modern processor design.

References

[1] D. Patterson and J. Hennessy, *Computer Organization and Design*. Morgan Kaufmann.