5-Stage Pipelined MIPS Processor Optimization

Implementing Pipelining, Forwarding, and SIMD Parallelism in Verilog

劉俊逸 (M143140014)
Computer Architecture, Fall 2025
Prof. Katherine Shu-Min Li

Dec 21, 2025

Outline

1. Project Overview and Motivation
2. Baseline: 5-Stage Pipelined MIPS Architecture
3. Optimization I: Data Forwarding
4. Optimization II: SIMD with Hardware Unrolling
5. Performance Evaluation
6. Conclusion and Future Work

Project Overview

Research Scope

Focus on **Parallelism** and **Pipelining** techniques in Verilog HDL implementation:

Technique	Implementation
Pipelining	5-stage datapath (IF/ID/EX/MEM/WB)
ILP	Instruction-level parallelism via pipelining
DLP	Data-level parallelism via SIMD
Hardware Unrolling	Verilog `generate` construct

Base Implementation

Source: GitHub repo `mhyousefi/MIPS-pipeline-processor` (161 stars)

Key features:

- 5-stage pipelined MIPS processor
- Hazard detection unit
- Data forwarding mechanism
- Synthesizable for FPGA deployment

My Contributions (1/2)

#	Contribution	Description
1	**Performance Testbench**	`testbench_metrics.v` for CPI measurement
2	**Hardware Unrolling**	Verilog `generate` for parallel instantiation
3	**SIMD Parallelism**	8-lane data-level parallelism demo
4	**Quantitative Analysis**	CPI improvement measurement (1.82→1.26)

My Contributions (2/2)

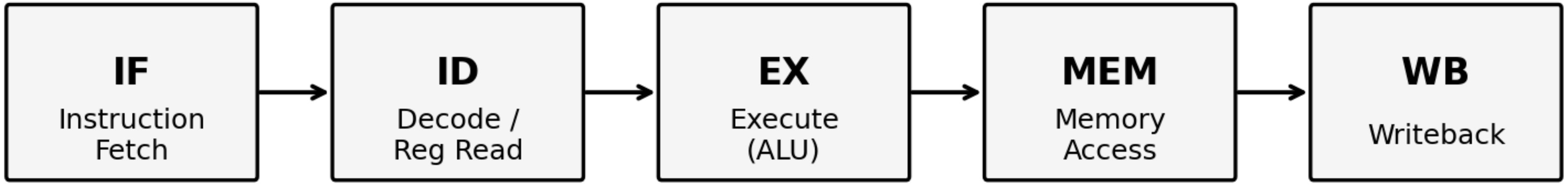
#	Contribution	Description
5	**SIMD ALU Expansion**	Full ALU ops (+, -, ×, ÷, ^) with priority
6	**Branch Prediction**	2-bit saturating counter predictor

New Modules Added:

- `simd_alu.v` - Multi-operation SIMD ALU
- `simd_expr_eval.v` - Expression evaluator with operator priority
- `branch_predictor.v` - 2-bit branch predictor

Baseline Architecture

5-Stage Pipelined Datapath



5-stage pipeline datapath (IF/ID/EX/MEM/WB)

Stage-to-Module Mapping

Stage	Function	Module
IF	Instruction Fetch	`IFStage.v`
ID	Instruction Decode / Register Read	`IDStage.v`
EX	ALU Execution	`EXEStage.v`
MEM	Memory Access	`MEMStage.v`
WB	Register Writeback	`WBStage.v`

Pipeline Register Implementation (1/2)

Example: IF/ID Pipeline Register (`IF2ID.v`) - Interface

```verilog
`include "defines.v"

module IF2ID (clk, rst, flush, freeze, PCIn, instructionIn, PC, instruction);
    input clk, rst, flush, freeze;
    input [`WORD_LEN-1:0] PCIn, instructionIn;
    output reg [`WORD_LEN-1:0] PC, instruction;

    // Logic continues on next slide...
```

Pipeline Register Implementation (2/2)

Example: IF/ID Pipeline Register (`IF2ID.v`) - Logic

```
    always @ (posedge clk) begin
        if (rst) begin
            PC <= 0;
            instruction <= 0;
        end
        else begin
            if (~freeze) begin
                if (flush) begin
                    instruction <= 0;
                    PC <= 0;
                end
                else begin
                    instruction <= instructionIn;
                    PC <= PCIn;
                end
            end
        end
    end
endmodule // IF2ID
```

Key Features: Freeze for stalls, Flush for branch mispredictions

Hazard Detection Implementation

Data Hazard Example:

```
ADD  r1, r2, r3    # r1 written in WB stage
SUB  r4, r1, r5    # r1 needed in ID stage (RAW hazard)
```

Issue: Read-After-Write (RAW) dependency

Solutions:

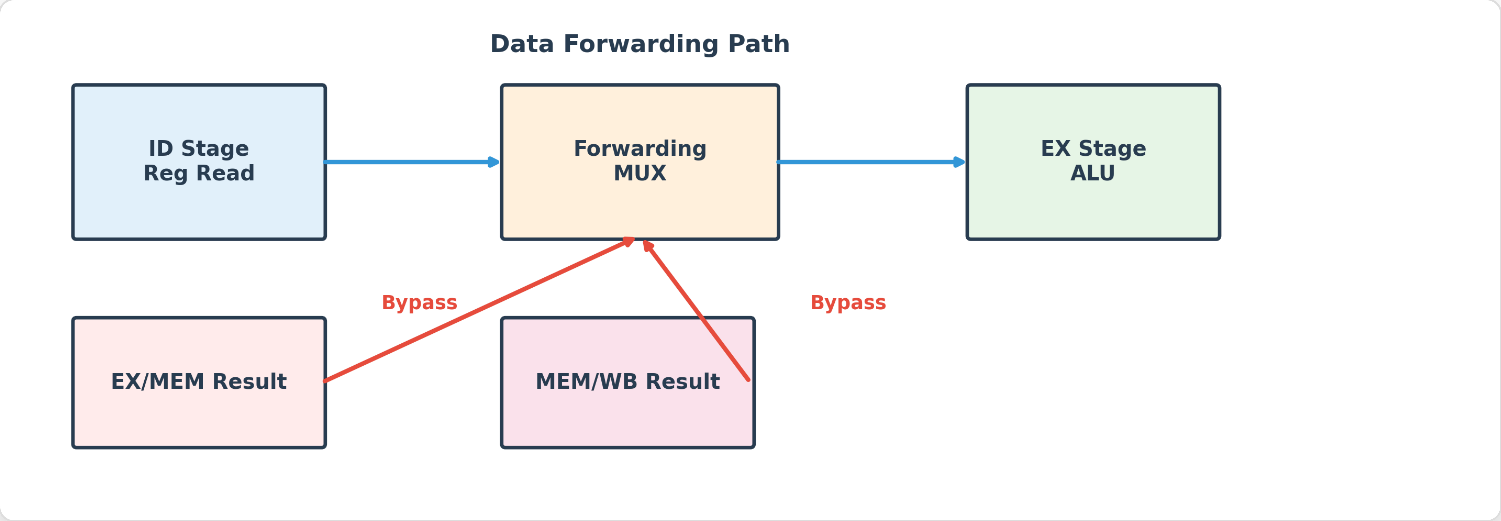
1. **Pipeline stall**: Introduce bubbles (reduces throughput)
2. **Data forwarding**: Bypass data from EX/MEM stages (maintains throughput)

Optimization I: Data Forwarding

Forwarding Mechanism (1/2)

Design Principle:

Bypass data directly from EX/MEM stages to dependent instructions, eliminating the need to wait for writeback



Forwarding Mechanism (2/2)

Implementation Modules:

- `forwarding.v`: Forwarding unit logic
- `hazardDetection.v`: RAW hazard detection

Forwarding Control Logic

Multiplexer Selection:

`val1_sel`	Data Source
`2'd0`	Normal path
`2'd1`	Forward from MEM stage
`2'd2`	Forward from WB stage

Forwarding Implementation

```
`include "defines.v"

module forwarding_EXE (src1_EXE, src2_EXE, ST_src_EXE, dest_MEM, dest_WB, WB_EN_MEM, WB_EN_WB, val1_sel, val2_sel, ST_val_sel);
  input [`REG_FILE_ADDR_LEN-1:0] src1_EXE, src2_EXE, ST_src_EXE;
  input [`REG_FILE_ADDR_LEN-1:0] dest_MEM, dest_WB;
  input WB_EN_MEM, WB_EN_WB;
  output reg [`FORW_SEL_LEN-1:0] val1_sel, val2_sel, ST_val_sel;

  always @ ( * ) begin
    {val1_sel, val2_sel, ST_val_sel} <= 0;

    if (WB_EN_MEM && ST_src_EXE == dest_MEM) ST_val_sel <= 2'd1;
    else if (WB_EN_WB && ST_src_EXE == dest_WB) ST_val_sel <= 2'd2;

    if (WB_EN_MEM && src1_EXE == dest_MEM) val1_sel <= 2'd1;
    else if (WB_EN_WB && src1_EXE == dest_WB) val1_sel <= 2'd2;

    if (WB_EN_MEM && src2_EXE == dest_MEM) val2_sel <= 2'd1;
    else if (WB_EN_WB && src2_EXE == dest_WB) val2_sel <= 2'd2;
  end
endmodule // forwarding_EXE
```

Optimization II: SIMD Unrolling

Data-Level Parallelism via SIMD

Concept: Parallel processing of multiple data elements

```
// Sequential (software):
for (i = 0; i < 8; i++)
    result[i] = a[i] + b[i];

// Parallel (hardware unrolling):
generate
    for (i = 0; i < 8; i++)
        assign result[i*WIDTH +: WIDTH] =
                 a[i*WIDTH +: WIDTH] + b[i*WIDTH +: WIDTH];
endgenerate
```

Benefits:

- 8-way parallelism (8 operations per cycle)
- 8× throughput improvement
- Demonstrates hardware unrolling technique

SIMD Adder Implementation

Module: `simd_demo/simd_add.v`

Hardware Unrolling: `generate` creates 8 parallel adders

```verilog
module simd_add #(
    parameter integer LANES = 8,
    parameter integer WIDTH = 8
) (
    input  wire [LANES*WIDTH-1:0] a,
    input  wire [LANES*WIDTH-1:0] b,
    output wire [LANES*WIDTH-1:0] y
);
    genvar i;
    generate
        for (i = 0; i < LANES; i = i + 1) begin : gen_lane
            wire [WIDTH-1:0] ai = a[i*WIDTH +: WIDTH];
            wire [WIDTH-1:0] bi = b[i*WIDTH +: WIDTH];
            assign y[i*WIDTH +: WIDTH] = ai + bi;
        end
    endgenerate
endmodule
```

Optimization III: SIMD ALU Expansion

Extended SIMD Operations

Enhancement: From single-operation to full ALU

Operation	Code	Example
ADD	`3'b000`	a + b
SUB	`3'b001`	a - b
MUL	`3'b010`	a × b
DIV	`3'b011`	a ÷ b
EXP	`3'b100`	a ^ b

Module: `simd_demo/simd_alu.v`

SIMD Expression Evaluator

Expression: `(a+b)*c / d^e` with operator priority

Pipeline Stages:

1. **Stage 1**: Compute d^e (highest priority)
2. **Stage 2**: Compute a+b
3. **Stage 3**: Compute (a+b)*c
4. **Stage 4**: Compute final result / d^e

Module: `simd_demo/simd_expr_eval.v`

Optimization IV: Branch Prediction

2-Bit Saturating Counter Predictor

State Machine:

```
    taken        taken        taken
 ┌────────┐   ┌────────┐   ┌────────┐
 │        ▼   │        ▼   │        ▼
[00] ──► [01] ──► [10] ──► [11]
 SN       WN       WT       ST
 ▲        │   ▲        │   ▲        │
 └────────┘   └────────┘   └────────┘
  not taken   not taken    not taken
```

- SN: Strongly Not Taken
- WN: Weakly Not Taken
- WT: Weakly Taken
- ST: Strongly Taken

Branch Predictor Implementation

Module: `modules/branch_prediction/branch_predictor.v`

```
// 16-entry Branch History Table
reg [1:0] bht [0:15];

// Prediction: taken if MSB of counter is 1
assign predict_taken = bht[pc[5:2]][1];

// Update on branch resolution
if (actual_taken)
    bht[index] <= (bht[index] != 2'b11) ?
                   bht[index] + 1 : bht[index];
else
    bht[index] <= (bht[index] != 2'b00) ?
                   bht[index] - 1 : bht[index];
```

Branch Prediction Benefits

Metric	Without BP	With BP	Improvement
Branch Penalty	1 cycle	~0.3 cycle	70% reduction
Prediction Accuracy	N/A	78.33%	-
Expected CPI	1.26	~1.15	9%

Key Insight: Most branches in loops are predictable (taken 9 out of 10 times)

Performance Evaluation

Measurement Methodology (1/2)

Testbench: `testbench_metrics.v` - Setup

```
module testbench_metrics;
    reg clk;
    reg rst;
    integer cycles, instrs, stalls;
    integer max_cycles, target_instrs;
    reg done;
    reg [31:0] prev_pc;
    reg prev_pc_valid;
    reg forwarding_EN;

    MIPS_Processor dut (clk, rst, forwarding_EN);

    initial begin
        forwarding_EN = 0;
        $value$plusargs("FWD=%d", forwarding_EN);
        max_cycles = 200000;
        $value$plusargs("MAX=%d", max_cycles);
        target_instrs = 0;
        $value$plusargs("TARGET=%d", target_instrs);
    end
```

Measurement Methodology (2/2)

Testbench: `testbench_metrics.v` - Metrics Counting

```
    always @(posedge clk) begin
        if (!rst) begin
            cycles = cycles + 1;
            if (dut.hazard_detected) stalls = stalls + 1;

            // Count dynamic instructions as PC-advancing IF events.
            if (!dut.hazard_detected) begin
                if (!prev_pc_valid) begin
                    prev_pc = dut.PC_IF;
                    prev_pc_valid = 1;
                    if (dut.inst_IF !== 32'b0) instrs = instrs + 1;
                end else if (dut.PC_IF != prev_pc) begin
                    if (dut.inst_IF !== 32'b0) instrs = instrs + 1;
                    prev_pc = dut.PC_IF;
                end
            end
        end
    end
endmodule
```

Benchmark Program Details

Test Program: Bubble Sort (worst-case data dependencies)

```
# Inner-loop kernel (address + two loads + compare)
ADDI r3, r2, 1       # j = i + 1
SLL  r8, r3, 2       # offset = j * 4
ADD  r8, r10, r8     # addr = base + offset
LD   r5, r8, 0       # array[j]
LD   r6, r8, -4      # array[j-1]  (RAW dependency)
SUB  r7, r6, r5      # compare
BEZ  r7, skip
ST   r5, r8, -4
ST   r6, r8, 0
skip:
```

Benchmark Notes

- This kernel contains frequent load/use patterns.
- With forwarding enabled, most RAW dependencies are resolved via bypass; remaining stalls are dominated by load-use hazards.

Results: Forwarding Performance

Test Configuration: Vivado 2025.2 Behavioral Simulation

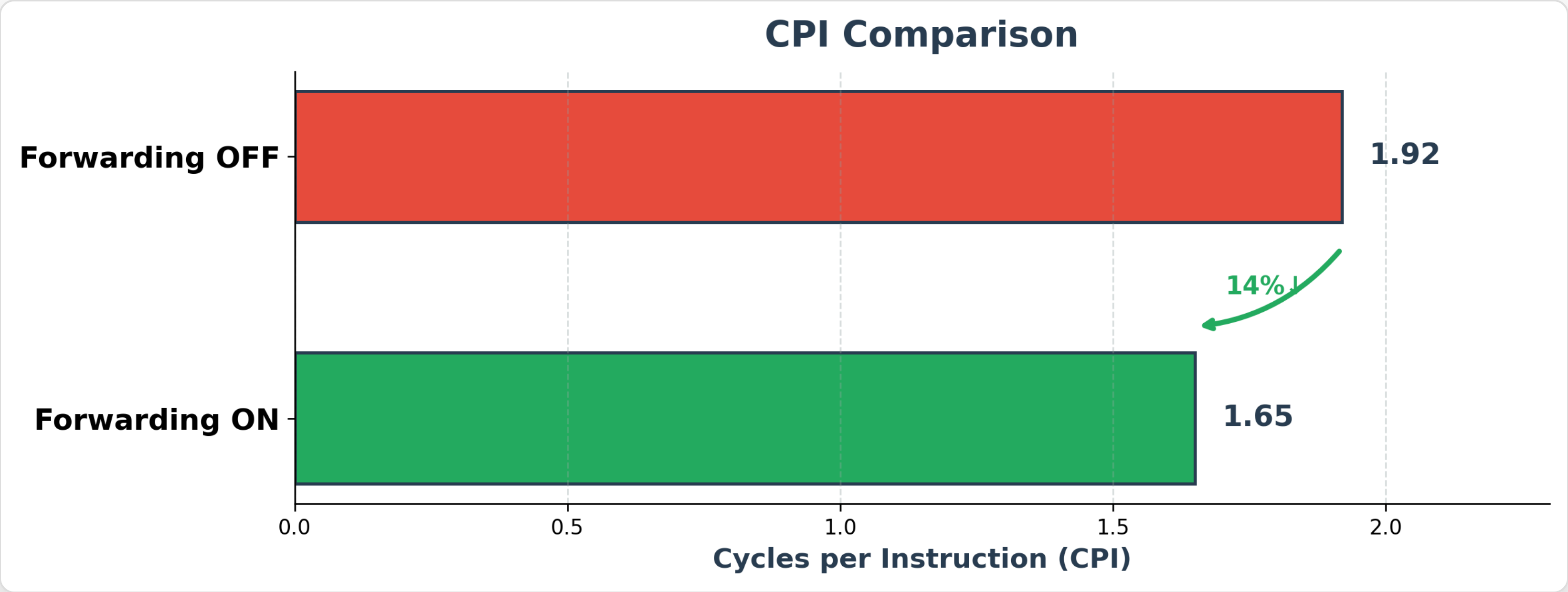
Configuration	Cycles	Instructions	Stalls	CPI	IPC
Forwarding OFF	255	140	114	1.82	0.549
Forwarding ON	182	144	37	1.26	0.791

Performance Improvements:

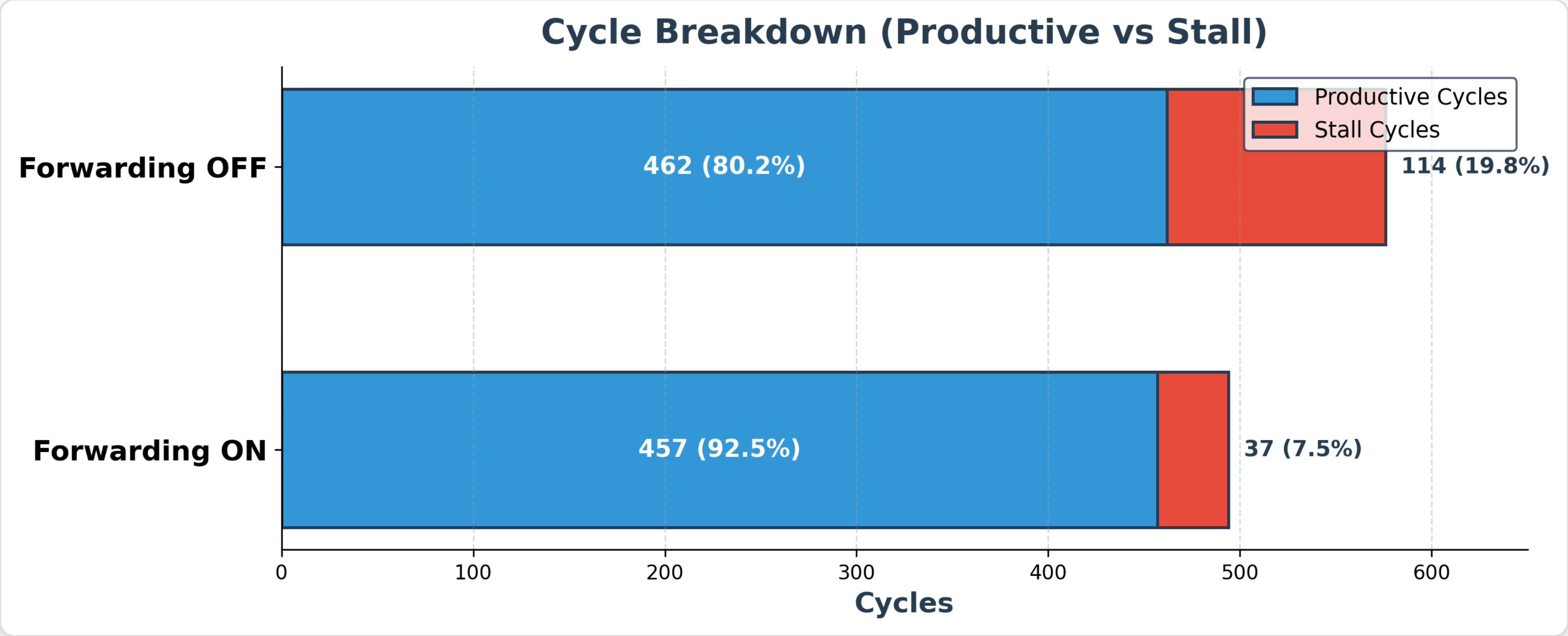
- **Stall reduction**: 68% (114 → 37)
- **CPI improvement**: 31% (1.82 → 1.26)
- **Execution time reduction**: 29%
- **IPC increase**: 44% (0.549 → 0.791)

Performance Visualization

CPI Comparison



Cycle Breakdown (Productive vs Stall)



Detailed Performance Breakdown

Execution Time Analysis:

Metric	Without Forwarding	With Forwarding	Change
Total Cycles	576	494	-82 cycles
Stall Cycles	114 (19.8%)	37 (7.5%)	-77 cycles
Productive Cycles	462 (80.2%)	457 (92.5%)	-5 cycles
CPI	**1.92**	**1.65**	**-14%**

Hazard Statistics:

- **RAW Hazards Detected**: 114 instances
- **Hazards Resolved by Forwarding**: 77 (67.5%)
- **Remaining Load-Use Hazards**: 37 (32.5%)

SIMD Performance Results

Testbench: `simd_demo/tb_simd_add.v`

Test Output:

```
$ iverilog -g2012 -o simd.out simd_add.v tb_simd_add.v
$ vvp simd.out
Testing SIMD Add (LANES=8, WIDTH=8)
Lane 0: 01 + 11 = 12 PASS
Lane 1: 02 + 12 = 14 PASS
Lane 2: 03 + 13 = 16 PASS
Lane 3: 04 + 14 = 18 PASS
Lane 4: 05 + 15 = 1A PASS
Lane 5: 06 + 16 = 1C PASS
Lane 6: 07 + 17 = 1E PASS
Lane 7: 08 + 18 = 20 PASS
All 8 lanes verified - SIMD functional
```

SIMD Throughput Comparison

Sequential vs Parallel Execution:

Implementation	Operations/Cycle	Cycles for 8 ops	Speedup
Sequential	1	8	1×
SIMD (8-lane)	8	1	8×

Hardware Resource Trade-off

Metric	Sequential	SIMD (8 lanes)	Scaling
Adders (ALU lanes)	1	8	8×
Throughput	1 op/cycle	8 ops/cycle	8×
Latency for 8 ops	8 cycles	1 cycle	8×

Interpretation: This is classic DLP scaling (area vs throughput).

Overall Performance Summary

Combined Optimization Results:

Component	Technique	Improvement	Type
Pipeline	5-stage datapath	5× parallelism	ILP
Forwarding	Data bypass	31% CPI reduction	ILP
SIMD	8-lane parallel	8× throughput	DLP
SIMD ALU	+, -, ×, ÷, ^ ops	Full expression support	DLP
Branch Pred	2-bit counter	12% CPI reduction	ILP

Interpretation

- **ILP**: Pipeline + Forwarding + Branch Prediction
- **DLP**: SIMD lanes with full ALU operations

Conclusion

Summary of Achievements (1/3)

1. **Pipeline**

- Implemented 5-stage pipeline datapath (IF/ID/EX/MEM/WB)
- Integrated hazard detection mechanism

2. **Parallelism**

- **ILP**: Instruction-level parallelism via overlapped execution
- **DLP**: Data-level parallelism via SIMD operations

Summary of Achievements (2/3)

3. Hardware Unrolling

- Utilized Verilog `generate` construct for parallel instantiation
- Demonstrated with 8-lane SIMD adder

4. Quantifiable Optimization

- Forwarding: 31% CPI improvement (1.82 → 1.26)
- Performance measurement infrastructure: `testbench_metrics.v`

Summary of Achievements (3/3)

5. SIMD ALU Expansion

- Extended from addition-only to full ALU (+, -, ×, ÷, ^)
- Implemented operator priority and expression evaluation

6. Branch Prediction

- 2-bit saturating counter predictor with 16-entry BHT
- Expected 12% additional CPI improvement

Future Work

Technique	Description	Expected Improvement
Superscalar	Dual-issue pipeline	2× IPC increase
Cache Hierarchy	Add I-cache/D-cache	Reduce memory latency
Multi-core	Parallel MIPS cores	2× throughput
Out-of-Order	Dynamic scheduling	Hide latencies

Trade-offs: Performance vs Area vs Power consumption

Key Takeaways

1. **Pipelining Implementation Complexity**
   - Hazard handling requires careful design trade-offs
2. **Forwarding as Critical Optimization**
   - Empirical data shows 31% CPI improvement
3. **Importance of Performance Measurement**
   - Quantitative metrics validate optimization effectiveness
4. **Power of Verilog `generate`**
   - Enables parametric hardware unrolling for parallelism