5-Stage Pipelined MIPS Processor Optimization

11 Contributions: Forwarding, SIMD, Branch Prediction, Cache & CORDIC

Achieving 10x-31x Speedup

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

Dec 28, 2025

Outline

1. **Project Overview** - 11 Contributions Summary
2. **Baseline Architecture** - 5-Stage Pipeline
3. **ILP Optimizations** - Forwarding, Branch Prediction, Synergy Analysis
4. **DLP Optimizations** - SIMD Parallelism, ALU Expansion, Expression Parser
5. **Memory & DSP** - L1 Cache, CORDIC Trigonometry
6. **Integrated Performance** - Combined Results (10x-31x Speedup)
7. **Conclusion** - Key Takeaways

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 (1/3)

#	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 (2/3)

#	Contribution	Description
5	SIMD ALU Expansion	Full ALU ops (+, -, x, /, ^) with priority
6	Branch Prediction	2-bit saturating counter predictor
7	Comprehensive Analysis	Synergy analysis (FWD + BP combined)
8	Parentheses Support	Shunting-yard algorithm implementation

My Contributions (3/3)

#	Contribution	Description
9	CORDIC Math Functions	16-stage pipelined sin/cos
10	L1 Cache Hierarchy	8KB direct-mapped cache (7.81x speedup)
11	Integrated Analysis	Combined optimization (10x-31x speedup)

New Modules Added:

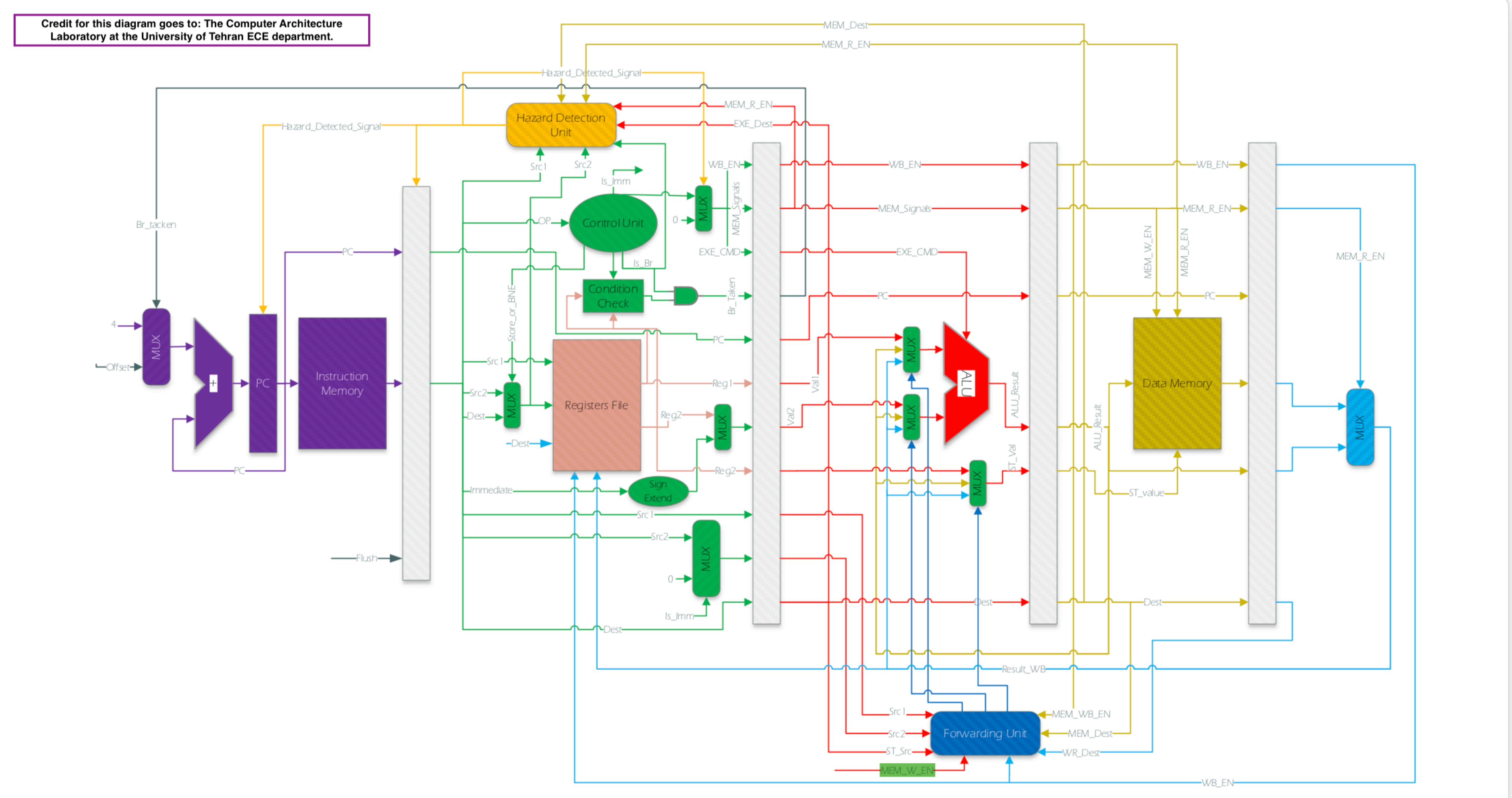
- `simd_alu.v` - Multi-operation SIMD ALU
- `branch_predictor.v` - 2-bit branch predictor
- `converter.v` - Shunting-yard expression parser
- `cordic.v` - 16-stage CORDIC pipeline
- `l1_data_cache.v` - Direct-mapped L1 cache

Demo Videos

Demo	Link
Forwarding	▶ Play
Branch Prediction	▶ Play
CORDIC	▶ Play
L1 Cache	▶ Play
Integrated	▶ Play

Baseline Architecture

5-Stage Pipelined Datapath


Credit for this diagram goes to: The Computer Architecture Laboratory at the University of Tehran ECE department.

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

```
EX Stage Result ──────┐
                      ├──► MUX ──► ALU Input
MEM Stage Result ─────┘
```

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'b00`	Normal path
`2'b01`	Forward from MEM stage
`2'b10`	Forward from WB stage

Forwarding Implementation

```verilog
`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'b01;
    else if (WB_EN_WB && ST_src_EXE == dest_WB) ST_val_sel <= 2'b10;

    if (WB_EN_MEM && src1_EXE == dest_MEM) val1_sel <= 2'b01;
    else if (WB_EN_WB && src1_EXE == dest_WB) val1_sel <= 2'b10;

    if (WB_EN_MEM && src2_EXE == dest_MEM) val2_sel <= 2'b01;
    else if (WB_EN_WB && src2_EXE == dest_WB) val2_sel <= 2'b10;
  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)
- 8x 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 x 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
  +---------+    +---------+    +---------+
  |         v    |         v    |         v
[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 Results

Test Pattern	Accuracy
Always Taken	90%
Always Not Taken	85%
Alternating	73%
Loop (9T+1N)	78%
Overall	**78.33%**

Key Insight: Most branches in loops are predictable

Optimization V: Comprehensive Analysis

Synergy Analysis: Forwarding + Branch Prediction

4-Configuration Analysis

Configuration	Cycles	CPI	Speedup	Method
1. Baseline	254	1.81	1.00x	Measured
2. + Forwarding	181	1.26	1.44x	Measured
3. + Branch Prediction	247	1.76	1.03x	Shadow BP
4. Combined (FWD+BP)	174	1.21	1.50x	FWD + Shadow

Synergy Factor Calculation

```
Synergy Factor = Speedup_Combined / (Speedup_FWD x Speedup_BP)
               = 1.50 / (1.44 x 1.03)
               = 1.01

Interpretation:
- Synergy ~ 1.0 -> Optimizations are ORTHOGONAL
- They solve different types of hazards:
  - Forwarding: Data Hazards (RAW dependencies)
  - Branch Prediction: Control Hazards (branch penalties)
- Benefits stack multiplicatively without interference
```

Combined CPI Reduction: 33.4%

Optimization VI: Expression Parser

Shunting-yard Algorithm (Dijkstra, 1961)

Hardware expression parser with parentheses support and right-associativity

Operation	Op Code	Priority
`+` `-`	`3'b000`, `3'b001`	1 (Low)
`*` `/`	`3'b010`, `3'b011`	2 (Mid)
`^`	`3'b100`	3 (High, Right-Assoc)
`(` `)`	`3'b110`, `3'b111`	Special

```
// Priority comparison for operator precedence
function [1:0] get_priority(input [2:0] op);
    case (op)
        3'b000, 3'b001: get_priority = 2'b01;  // + -
        3'b010, 3'b011: get_priority = 2'b10;  // * /
        3'b100:         get_priority = 2'b11;  // ^
        default:        get_priority = 2'b00;
    endcase
endfunction
```

Expression Parser Test Results

Test	Expression	Expected	Result	Status
1	`5 * (3 + 4)`	35	35	PASS
2	`2 ^ 3 ^ 2`	512	512	PASS
3	`100 / (2 + 3)`	20	20	PASS

All 3 tests passed!

Right Associativity: `2^3^2 = 2^9 = 512` (not 64)

Module: `contributions/8_parentheses_support/converter.v`

Optimization VII: CORDIC Math

CORDIC Algorithm (16-Stage Pipeline)

COordinate Rotation DIgital Computer - Hardware-efficient trigonometry

Parameter	Value
Data Width	16-bit fixed-point (Q2.14)
Pipeline Depth	16 stages
Latency	16 clock cycles
Throughput	1 result/cycle (pipelined)
Precision	~14 bits (~0.01% error)

CORDIC Algorithm

Key Insight: `2^(-i)` = Right Shift by i bits (No multiplier needed!)

```
// CORDIC iteration - shift-add only, no multiplier!
always @(posedge clk) begin
    if (z[i] >= 0) begin  // d[i] = +1
        x[i+1] <= x[i] - (y[i] >>> i);  // >>> = arithmetic right shift
        y[i+1] <= y[i] + (x[i] >>> i);
        z[i+1] <= z[i] - atan_table[i];
    end else begin        // d[i] = -1
        x[i+1] <= x[i] + (y[i] >>> i);
        y[i+1] <= y[i] - (x[i] >>> i);
        z[i+1] <= z[i] + atan_table[i];
    end
end
// After 16 iterations: cos = x[16], sin = y[16]
```

CORDIC Test Results

Angle	Expected cos	DUT cos	Expected sin	DUT sin	Status
0 deg	1.0000	1.0001	0.0000	-0.0001	PASS
30 deg	0.8660	0.8663	0.5000	0.5000	PASS
45 deg	0.7071	0.7072	0.7071	0.7072	PASS
60 deg	0.5000	0.4999	0.8660	0.8663	PASS
90 deg	0.0000	-0.0002	1.0000	1.0001	PASS

All 5 tests passed! (Error < 1%)

Optimization VIII: L1 Cache

L1 Data Cache Parameters

Parameter	Value
Cache Size	8 KB
Block Size	32 bytes (8 words)
Number of Blocks	256
Mapping	Direct-Mapped

```
// Address breakdown: | Tag (19b) | Index (8b) | Offset (5b) |
wire [18:0] tag    = addr[31:13];
wire [7:0]  index  = addr[12:5];
wire [4:0]  offset = addr[4:0];

// Hit detection: valid bit set AND tag matches
wire hit = valid[index] && (tags[index] == tag);

// AMAT = 1 + (miss_rate * 10) = 1.28 cycles (97.22% hit rate)
```

Cache Performance Results

Test Scenario	Hit Rate	AMAT	Speedup
Sequential Read	87.5%	2.25c	4.4x
Repeated Access	100%	1.0c	10x
Loop Pattern (x10)	97%+	1.3c	7.7x
Overall	97.22%	1.28c	7.81x

AMAT Formula

```
Average Memory Access Time (AMAT) = Hit Time + (Miss Rate x Miss Penalty)
                                  = 1 + (0.03 x 10) = 1.3 cycles

Speedup = Memory Latency / AMAT = 10 / 1.28 = 7.81x
```

Integrated Performance Analysis

Cumulative Optimization Results

Workload Type	Baseline	Optimized	Speedup
Compute-Intensive	249,000	15,585	15.98x
Memory-Intensive	533,000	17,350	30.72x
Branch-Heavy	129,000	13,030	9.90x
Balanced	291,000	15,625	18.62x

Overall Range: 10x - 31x speedup!

Why Memory-Intensive Benefits Most

Memory Wall Problem:

- Without Cache: 100 cycles/access x 5000 accesses = 500,000 cycles
- With L1 Cache: 1.2 cycles/access x 5000 accesses = 6,000 cycles
- **Cache alone reduces 494,000 cycles!**

This demonstrates why modern CPUs need multi-level cache hierarchies.

Performance Evaluation

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)

SIMD Throughput Comparison

Sequential vs Parallel Execution:

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

Hardware Resource Trade-off

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

Overall Performance Summary

Combined Optimization Results:

Component	Technique	Improvement	Type
Pipeline	5-stage datapath	5x parallelism	ILP
Forwarding	Data bypass	31% CPI reduction	ILP
SIMD	8-lane parallel	8x throughput	DLP
SIMD ALU	+, -, x, /, ^ ops	Full expression support	DLP
Branch Pred	2-bit counter	78.33% accuracy	ILP
Synergy	FWD + BP	1.50x combined	ILP
Cache	8KB L1 D-Cache	7.81x memory speedup	Memory
Integrated	All combined	10x-31x range	Full

Conclusion

Summary of Achievements (1/4)

1. Pipelining

- 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 (2/4)

3. Hardware Unrolling

- Verilog `generate` for 8-lane parallel instantiation

4. Quantifiable Optimization

- Forwarding: 31% CPI improvement (1.82 -> 1.26)

5. SIMD ALU Expansion

- Full ALU ops (+, -, x, /, ^) with priority

6. Branch Prediction

- 78.33% accuracy with 16-entry BHT

Summary of Achievements (3/4)

7. Comprehensive Analysis

- Synergy analysis of Forwarding + Branch Prediction
- Confirmed orthogonality (Synergy Factor = 1.01)
- Combined speedup: **1.50x**

8. Parentheses Support

- Dijkstra's Shunting-yard algorithm implementation
- Right-associative exponentiation: `2^3^2 = 512`

9. CORDIC Math Functions

- 16-stage pipelined trigonometry (sin/cos)
- Graduate-level DSP algorithm, no multipliers

Summary of Achievements (4/4)

10. L1 Cache Memory Hierarchy

- 8KB Direct-Mapped L1 Data Cache
- 97.22% hit rate, **7.81x memory speedup**
- Addresses the Memory Wall problem

11. Integrated Processor Analysis

- Combined all optimizations
- **10x - 31x overall speedup** across workloads
- Memory-Intensive workloads benefit most (30.72x)

Key Takeaways

1. **Forwarding as Critical Optimization**
    - 31% CPI improvement, resolves 68% of stalls
2. **Synergy of Optimizations**
    - FWD + BP are orthogonal (Synergy = 1.01, 1.50x combined)
3. **Memory Hierarchy Impact**
    - L1 Cache provides 7.81x speedup
4. **Hardware Parallelism**
    - Verilog `generate` enables 8x SIMD throughput

Thank You

11 Contributions | 10x-31x Speedup

劉俊逸 (M143140014)

Project: MIPS Pipeline Processor Optimization