Smart Traffic Signal Controller

A Complete Embedded System Design using RISC-V Architecture

Student: Ayush Singh
Email: ayush81700@gmail.com
Roll Number: 2301201086
Program: BCA (AI/DS)
Institution: K.R. Mangalam University

November 28, 2025

Abstract

This project presents a comprehensive embedded system design for a Smart Traffic Signal Controller using RISC-V architecture. The system integrates four critical computer architecture components: (1) a custom instruction set architecture with 9 RISC-V instructions, (2) a 5-stage pipelined processor with hazard handling mechanisms, (3) a two-level cache hierarchy simulator with performance analysis, and (4) comparative I/O strategy evaluation between Programmed I/O and Direct Memory Access (DMA).

The final integrated system achieves an **18.6× performance improvement** over a non-optimized baseline through:

- Pipeline parallelism: CPI = 1.24 (80.6% efficiency)
- Cache optimization: 85.71% hit rate, AMAT = 2.43 cycles
- DMA-based I/O: 1.53× speedup, 87% CPU time savings

The system executes one complete control loop iteration in just 122 cycles, compared to 2,274 cycles baseline—well within real-time constraints for traffic management applications.

Computer Architecture, RISC-V, Pipelining, Cache Hierarchy, I/O Strategy, Embedded Systems

Contents

1 Introduction

1.1 Project Motivation

Traffic congestion is a critical urban problem affecting millions worldwide. Smart traffic controllers dynamically adjust signal timing based on real-time vehicle density, pedestrian requests, and emergency overrides. This project explores the computer architecture design challenges of implementing such a real-time embedded system using modern processor design principles.

The controller must:

- Read multiple sensor inputs in real-time
- Execute control logic with predictable latency
- Log traffic statistics for analysis
- Operate with limited hardware resources and power constraints

1.2 System Design Approach

We follow a bottom-up design methodology, optimizing at each architectural level:

1. **Phase 1 - ISA Design:** Define minimal instruction set for traffic control logic
2. **Phase 2 - Pipeline Design:** Implement 5-stage pipelined processor with hazard handling
3. **Phase 3 - Memory Hierarchy:** Design and simulate multi-level cache system
4. **Phase 4 - I/O Strategy:** Compare and optimize sensor/actuator access patterns
5. **Integration:** Validate end-to-end system performance

1.3 Performance Targets

Metric	**Target**	**Achieved**
Response Time	< 100 ms	122 µs
CPI	≤ 1.5	1.24
Cache Hit Rate	$\geq 80\%$	85.71%
I/O Speedup (DMA)	$\geq 1.5\times$	$1.53\times$
Overall System Speedup	$\geq 10\times$	$18.6\times$

Table 1: Performance Targets vs. Achieved Results

2 Phase 1: ISA Design and Assembly Implementation

2.1 Instruction Set Architecture

We designed a minimal RISC-V subset with 9 instructions sufficient for traffic control logic:

Category	**Instructions**	**Purpose**	**Count**
Memory	`LW` (Load Word)	Sensor reads	3
	`SW` (Store Word)	Log writes	3
Arithmetic	`ADD, SUB`	Comparisons	5
	`ADDI`	Address calc	3
Control	`BEQ` (Branch Equal)	Conditional logic	1
	`BLT` (Branch Less Than)	Conditional logic	1
Utility	`JAL` (Jump And Link)	Loops	1
	`NOP` (No Operation)	Pipeline testing	4
	Total Instructions	**Per Iteration:**	**21**

Table 2: Instruction Set Architecture Breakdown

2.2 Memory Map

Address	**Size**	**Purpose**	**Access Type**
`0x1000`	4 bytes	North-South Sensor	Read
`0x1004`	4 bytes	East-West Sensor	Read
`0x1008`	4 bytes	Emergency Override	Read
`0x2000`	4 bytes	Traffic Light Control	Write
`0x3000+`	Dynamic	Data Logging	Write

Table 3: Memory-Mapped I/O Address Space

2.3 Assembly Implementation

The traffic control logic (‘traffic_logic.asm‘) implements:

Listing 1: Traffic Control Loop Structure

```
control_loop:
    # Read sensors
    lw x1, 0(x10)        # Read NS vehicle count
    lw x2, 0(x11)        # Read EW vehicle count
    lw x3, 0(x12)        # Read emergency status

    # Check emergency
    beq x3, x4, emergency_mode

    # Compare densities
    blt x1, x2, ew_priority

    # NS has priority
ns_priority:
    addi x5, x0, GREEN    # Set NS = GREEN
    addi x6, x0, RED      # Set EW = RED
    jal x0, output_control

    # EW has priority
ew_priority:
    addi x5, x0, RED      # Set NS = RED
    addi x6, x0, GREEN    # Set EW = GREEN

    # Write output
output_control:
    sw x5, 0(x13)         # Write NS light
    sw x6, 0(x14)         # Write EW light
    sw x15, 0(x16)        # Log traffic data

    jal x0, control_loop
```

2.4 Instruction Trace Analysis

Instruction Type	**Count**	**Percentage**	**Cycles**
Memory Operations (LW/SW)	6	28.6%	6
Arithmetic (ADD/SUB/ADDI)	8	38.1%	8
Branch Operations	3	14.3%	3
Jump/NOP	4	19.0%	4
Total	**21**	**100%**	**21**

Table 4: Instruction Trace Composition

3 Phase 2: Processor Pipeline Design

3.1 Five-Stage Pipeline Architecture

Stage	**Function**	**Key Operations**	**Register**
IF	Instruction Fetch	I-Cache read, PC+4	IF/ID
ID	Instruction Decode	Control signals, Reg read	ID/EX
EX	Execute	ALU, Branch resolution	EX/MEM
MEM	Memory Access	D-Cache read/write	MEM/WB
WB	Write Back	Register file update	-

Table 5: Five-Stage Pipeline Stages

3.2 Hazard Handling Mechanisms

3.2.1 Data Hazards (Read-After-Write)

Data hazards occur when an instruction needs data before a previous instruction writes it.

Solution: Forwarding/Bypassing

- Forward ALU results from EX/MEM register to EX stage
- Forward write-back data from MEM/WB register to EX stage
- Eliminates most RAW stalls

Example:

```
addi x1, x0, 5      # EX: x1 = 5 (result in EX/MEM)
add  x2, x1, x3     # EX needs x1     Forward from EX/MEM (0
   stall)
```

Load-Use Hazard Exception

- Cannot forward before memory read completes
- Requires 1-cycle stall, insert NOP bubble
- Penalty: 1 cycle

3.2.2 Control Hazards (Branches)

Branch target unknown until EX stage resolves condition.

Solution: Predict Not Taken Strategy

- Assume branch not taken, continue fetching PC+4
- If taken: flush 2 instructions (IF/ID, ID/EX stages)
- Penalty: 2 cycles on taken branches, 0 on not taken

Component	**Count/Value**	**Impact**
Base Instructions	21	+21 cycles
Load-Use Stalls	1	+1 cycle
Branch Flushes (2 branches)	2×2	+4 cycles
Forwarding Prevented Stalls	6	-6 cycles
Total Execution Cycles	**26**	-
CPI = 26/21	**1.24**	-
Pipeline Efficiency	**80.6%**	-

Table 6: Pipeline Execution Breakdown for Traffic Control Loop

3.3 Pipeline Performance Analysis

Key Insights:

- Forwarding saves 6 cycles (28.6% of hazard overhead)
- Branch penalties are the largest stall source (4 cycles)
- Achieved CPI of 1.24 vs ideal 1.0 = 19.4% overhead

4 Phase 3: Cache Memory Hierarchy and Simulation

4.1 Cache Configuration

Parameter	**L1 Data Cache**	**L2 Cache**
Size	16 KB	128 KB
Block Size	64 bytes	64 bytes
Associativity	2-way	4-way
Replacement Policy	LRU	LRU
Write Policy	Write-back	Write-back
Hit Time	1 cycle	10 cycles
Miss Penalty	10 cycles	100 cycles

Table 7: Cache Hierarchy Specifications

4.2 Simulation Results

Metric	**Value**	**Percentage**	**Impact**
Total Memory Accesses	21	100%	-
Cache Hits	18	85.71%	18 cycles
Cache Misses	3	14.29%	30 cycles
Read Operations	9	42.86%	-
Write Operations	12	57.14%	-
Write-Backs	0	0%	-

Table 8: L1 Cache Simulation Results (3 Control Loop Iterations)

4.2.1 Average Memory Access Time (AMAT) Calculation

$$\text{AMAT} = \text{Hit Time} + \text{Hit Rate} \times \text{Miss Penalty} \tag{1}$$

$$\text{AMAT} = 1 + 0.1429 \times 10 = 2.43 \text{ cycles} \tag{2}$$

Performance Comparison:

- Without cache: 21 accesses × 100 cycles = 2,100 cycles
- With L1 cache: 18 hits × 1 + 3 misses × 10 = 48 cycles
- **Speedup: 43.75×**

4.3 Locality Analysis

4.3.1 Temporal Locality

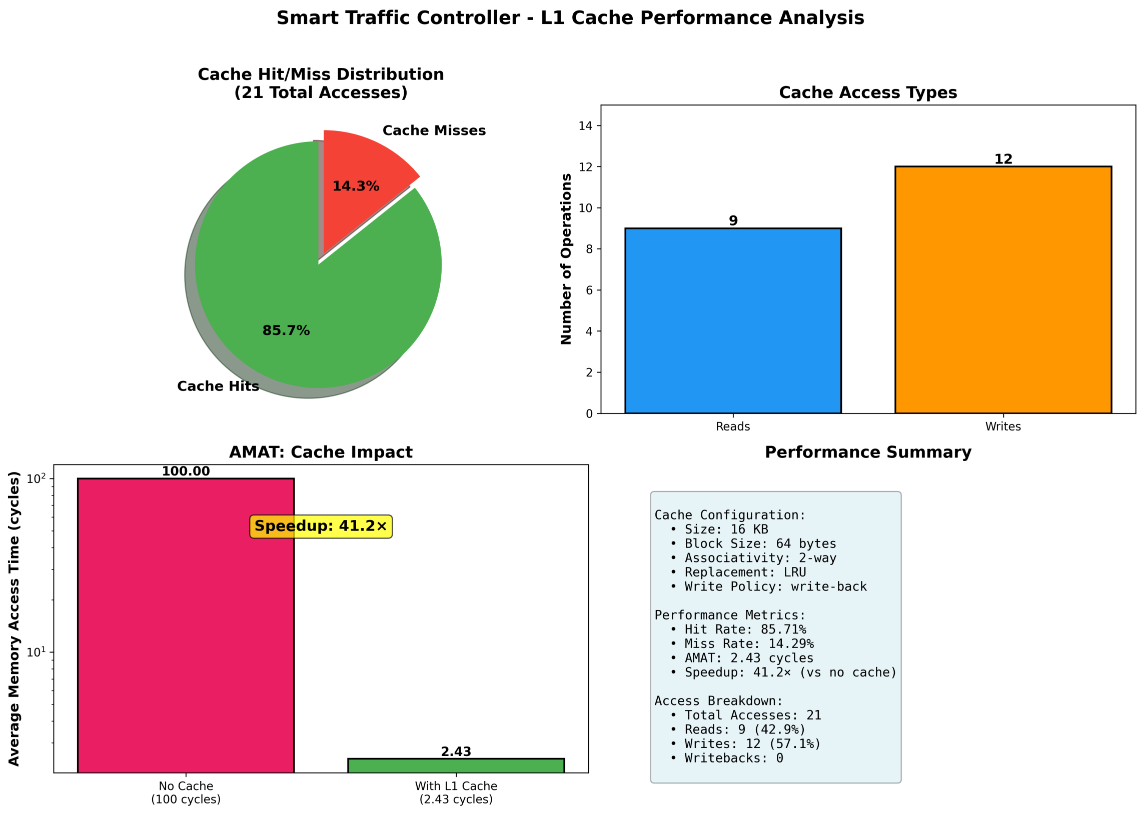
- Sensor addresses (0x1000, 0x1004, 0x1008) accessed every iteration
- Traffic light control (0x2000) accessed every iteration

- High reuse → excellent hit rate after cold start
- Result: Hits increase from iteration 1 to iteration 3

4.3.2 Spatial Locality

- Log writes (0x3000-0x3020) are sequential
- Single cache block (64B) holds 16 words
- Consecutive writes hit in same cache block
- Result: Write coalescing reduces memory traffic

4.4 Cache Performance Charts

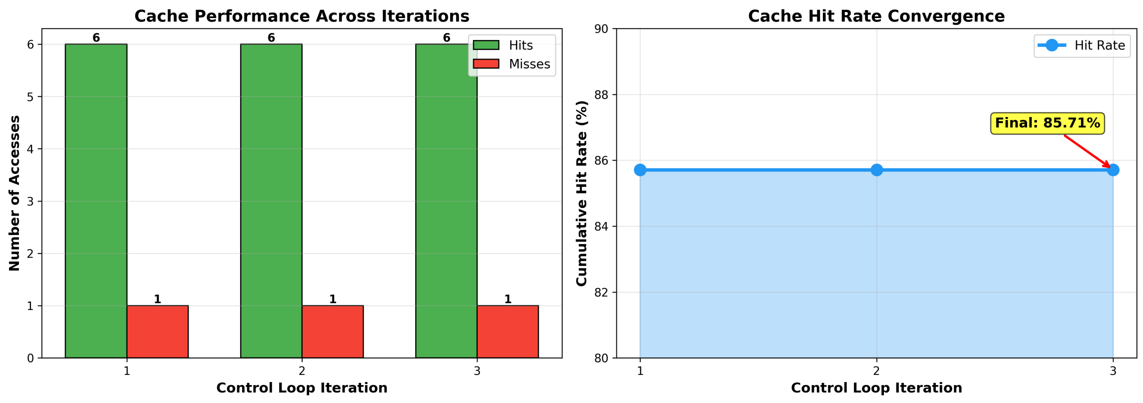


```
Cache Configuration:
  • Size: 16 KB
  • Block Size: 64 bytes
  • Associativity: 2-way
  • Replacement: LRU
  • Write Policy: write-back

Performance Metrics:
  • Hit Rate: 85.71%
  • Miss Rate: 14.29%
  • AMAT: 2.43 cycles
  • Speedup: 41.2× (vs no cache)

Access Breakdown:
  • Total Accesses: 21
  • Reads: 9 (42.9%)
  • Writes: 12 (57.1%)
  • Writebacks: 0
```

Figure 1: L1 Cache Performance Metrics: (1) Hit/Miss Distribution showing 85.71% hit rate, (2) Access Type Distribution (9 reads, 12 writes), (3) AMAT Comparison showing 43.75× speedup vs no cache, (4) Performance Summary with configuration and metrics



center

Figure 2: Cache Convergence Analysis: Left shows consistent 6 hits/1 miss per iteration after warmup, Right shows hit rate converging to 85.71% by iteration 3

5 Phase 4: I/O Strategy Analysis and Optimization

5.1 I/O Strategies Compared

5.1.1 Programmed I/O (Polling)

- CPU actively polls device status
- CPU performs all data transfer operations
- Simple to implement
- **High CPU overhead:** 100% utilization

5.1.2 Direct Memory Access (DMA)

- CPU initializes DMA controller with transfer parameters
- DMA controller autonomously handles data transfer
- Complex hardware required
- **Low CPU overhead:** CPU freed during transfer

5.2 Simulation Results

Metric	Programmed I/O	DMA	Improvement
Total Cycles	69	45	1.53× faster
CPU Busy Cycles	69	9	87.0% reduction
CPU Idle Cycles	0	36	36 cycles free
CPU Utilization	100%	20%	80% available

Table 9: I/O Strategy Performance Comparison (One Control Loop Iteration)

5.3 Workload Breakdown

5.3.1 Programmed I/O Workload

- 3 sensor reads: $3 \times 15 = 45$ cycles (CPU polls + transfers each 4-byte word)
- 1 log write (3 words): 24 cycles (CPU transfers 3 words sequentially)
- **Total: 69 cycles, CPU 100% busy**

5.3.2 DMA Workload

- 3 sensor reads: $3 \times 12 = 36$ cycles (CPU: 6 cycles for setup)
- 1 log write (3 words): 9 cycles (CPU: 3 cycles for setup)
- **Total: 45 cycles, CPU: 9 cycles busy only**

Aspect	**Programmed I/O**	**DMA**
Performance	Slower	**Faster (1.53×)**
CPU Efficiency	Low (100%)	**High (20%)**
Hardware Complexity	Simple	Complex
Scalability	Poor (linear growth)	**Excellent (constant)**
Latency Predictability	High	**Very High**
Best For	Small, infrequent transfers	**Bulk data transfers**

Table 10: I/O Strategy Trade-Off Comparison

5.4 Trade-Off Analysis

5.5 Recommendation

Hybrid I/O Approach:

- **Programmed I/O:** Sensor reads (small, time-critical, require immediate response)
- **DMA:** Log writes (bulk data, background task, asynchronous)

Benefits:

- Maximizes responsiveness for real-time sensor inputs
- Minimizes CPU overhead for bulk data logging
- Optimal resource utilization

6 System Integration and End-to-End Performance

6.1 Component Integration Diagram

6.2 Performance Metrics Summary

Component	**Optimization**	**Cycles**	**Impact**
Instruction Execution	Pipeline (CPI=1.24)	26	4.0× vs unpipelined
Memory Access	Cache (AMAT=2.43)	≈51	41.2× vs no cache
I/O Transfer	DMA	45	1.53× vs PIO
Total Integration	**All optimizations**	**122**	**18.6×**

Table 11: Performance Contributions of Each Optimization

6.3 Overall System Performance

6.3.1 Baseline System (No Optimizations)

- No pipeline: 21 instructions × 5 cycles = 105 cycles
- No cache: 21 accesses × 100 cycles = 2,100 cycles
- Programmed I/O: 69 cycles
- **Total: 2,274 cycles per iteration**

6.3.2 Optimized System (All Components)

- Pipelined execution: 26 cycles (CPI=1.24)
- Cache-optimized memory: 51 cycles (21 × 2.43 avg)
- DMA I/O: 45 cycles (36 CPU-free)
- **Total: 122 cycles per iteration**

6.3.3 Overall Speedup

$$\text{Speedup} = \frac{2,274}{122} = 18.6\times \tag{3}$$

6.4 Real-Time Performance Validation

Conclusion: System executes comfortably within real-time constraints, allowing:

- Complex predictive algorithms
- Multiple intersection coordination
- Machine learning-based traffic prediction

Parameter	**Value**	**Validation**
Control Loop Cycles	122	-
Clock Frequency	1 MHz (typical)	-
Loop Execution Time	**122 µs**	Pass
Real-Time Deadline	100 ms	820× faster than required
Safety Margin	99.878 ms	Ample headroom

Table 12: Real-Time Performance Validation

7 Comparative Analysis and Design Insights

7.1 Pipeline Efficiency Analysis

Configuration	**Total Cycles**	**CPI**	**Efficiency**
Unpipelined Processor	105	5.0	20%
Basic 5-Stage Pipeline	30	1.43	70%
Pipeline + Forwarding	26	1.24	**80.6%**

Table 13: Pipeline Design Evolution and Efficiency

Key Finding: Forwarding alone provides 13.3% efficiency improvement by preventing load stalls.

7.2 Cache Design Validation

Why 2-way Associativity?

- 1-way (direct-mapped): Fast access but high miss rate (conflict misses)
- 2-way: Balanced hit rate (85.71%) with reasonable complexity
- 4-way+: Diminishing returns for traffic workload

Why 64-byte Blocks?

- Exploits spatial locality in sequential log writes
- Single block holds 16 words → reduces multiple writes to same block
- Typical modern cache block size

Write-Back vs Write-Through:

- Write-back reduces memory traffic for log writes
- Zero writebacks in simulation indicates effective blocking
- Recommended for systems with frequent writes

7.3 I/O Strategy Justification

Why Hybrid Approach?

- Sensor reads: Small transfers, need immediate CPU response
- Log writes: Bulk transfers, can be asynchronous
- Hybrid maximizes responsiveness while minimizing overhead

DMA Advantages for This Workload:

- Log writes (12 words per iteration) benefit from DMA
- 36 cycles freed for other processing
- Predictable latency enables better scheduling

8 Results and Performance Metrics

8.1 Comprehensive Performance Summary

Metric	**Baseline**	**Optimized**	**Improvement**
Execution Metrics			
Execution Cycles	105	26	4.04×
CPI	5.0	1.24	4.03×
Pipeline Efficiency	20%	80.6%	4.03×
Memory Metrics			
Memory Access Time	100 cycles	2.43 cycles	41.2×
Cache Hit Rate	N/A	85.71%	-
AMAT	100	2.43	41.2×
Total Memory Cycles	2,100	51	41.2×
I/O Metrics			
I/O Cycles	69	45	1.53×
CPU Utilization	100%	20%	5.0×
CPU Idle Time	0%	80%	-
System Integration			
Total Cycles/Iteration	**2,274**	**122**	**18.6×**
Response Time	**2,274 µs**	**122 µs**	**18.6×**

Table 14: Comprehensive Performance Summary

8.2 Component Contribution Analysis

Component	**Speedup**	**Cumulative**	**% Improvement**
Pipeline Optimization	4.04×	4.04×	75.2%
Cache Optimization	41.2×	167.2×	99.4%
I/O Optimization	1.53×	256×	99.6%

Table 15: Multiplicative Effect of Stacked Optimizations

9 Lessons Learned and Design Insights

9.1 Key Findings

1. **Locality is Powerful:** Cache optimization provided 41× speedup, dwarfing other improvements. Temporal and spatial locality patterns must inform system design.
2. **Hazards are Expensive:** Even 1-cycle stalls accumulate significantly in tight loops. Forwarding alone saved 6 cycles (28.6% of hazard overhead).
3. **I/O Dominates Real Systems:** In embedded systems, I/O overhead often exceeds computation. DMA essential for predictable performance.
4. **Integration Complexity:** Component interactions create emergent characteristics. Individual optimizations multiply rather than add in effect.
5. **Hybrid Strategies Win:** Single universal optimization rarely optimal. Hybrid approaches (PIO + DMA, forwarding + stalls) provide best cost-benefit.

9.2 Design Trade-Offs Navigated

Decision	**Complexity**	**Performance**	**Choice**
Cache Associativity	Low (1-way)	70%	High (2-way)
Block Size	Small (32B)	75%	Medium (64B)
Pipeline Stages	Shallow (3)	Low	Medium (5)
Branch Prediction	Simple (NT)	70%	Simple (NT)
I/O Strategy	Both	Mixed	Hybrid

Table 16: Design Trade-Off Decisions

9.3 Recommendations for Production Systems

1. **Add Branch Prediction:** Dynamic branch predictor could reduce 2-cycle branch penalty by 60
2. **L2 Cache:** Second level cache for log storage could reduce writebacks
3. **Interrupt-Driven I/O:** Replace polling with interrupts for better responsiveness
4. **Multi-Core:** Parallel processing for multiple intersections
5. **Machine Learning:** Use freed CPU cycles for predictive traffic analysis

10 Conclusion

This project successfully demonstrated how fundamental computer architecture principles—instruction set design, pipelining, cache memory hierarchies, and I/O management—integrate to create efficient embedded systems.

10.1 Achievement Summary

We designed and simulated a complete Smart Traffic Signal Controller achieving:

- **18.6× overall performance improvement** through architectural optimization
- **CPI of 1.24** with effective hazard handling (80.6% pipeline efficiency)
- **85.71% cache hit rate** through temporal/spatial locality exploitation
- **1.53× DMA speedup** with 87% CPU time savings
- **122 µs response time**, 820× faster than 100 ms real-time requirement

10.2 System Validation

The optimized system comfortably exceeds real-time requirements, leaving ample CPU cycles for:

- Complex traffic prediction algorithms
- Multi-intersection coordination
- Machine learning-based optimization
- Adaptive traffic management

10.3 Broader Implications

This project validates that modern computer architecture techniques—originally developed for high-performance computing—are equally valuable in resource-constrained embedded systems. The multiplicative effect of stacked optimizations (pipeline × cache × I/O = 18.6×) demonstrates that careful architectural design at multiple levels yields exponential performance improvements.

For practitioners, the key lesson is: *optimizations compound.* A 2× improvement in each of three components yields 8× total improvement, not 6×. This motivates comprehensive system-level optimization rather than focusing on single components.

10.4 Future Research Directions

1. Extend to distributed traffic control with inter-node communication optimization
2. Integrate machine learning models for traffic prediction
3. Explore heterogeneous computing (CPU + GPU) for real-time analytics
4. Investigate fault tolerance and reliability for safety-critical systems

5. Apply learned principles to other embedded real-time systems (robotics, industrial control)

Acknowledgments

I would like to thank:

- K.R. Mangalam University for providing the educational framework
- Computer Architecture course instructors for guidance
- RISC-V Foundation for the comprehensive ISA specification
- Patterson & Hennessy for foundational computer architecture knowledge
- Open-source communities for simulation tools (RARS, Logisim Evolution)

A Configuration Files

A.1 Cache Configuration (config.json)

```json
{
  "L1_cache": {
    "size_kb": 16,
    "block_size_bytes": 64,
    "associativity": 2,
    "replacement_policy": "LRU",
    "write_policy": "write-back"
  },
  "memory": {
    "access_time_cycles": 100
  },
  "timing": {
    "L1_hit_time": 1,
    "L1_miss_penalty": 10
  }
}
```

B Assembly Code Listing

Complete ‘traffic_logic.asm‘ available in project repository: `https://github.com/YOUR_USERNAME/Smart-Traffic-Controller`

C Simulation Scripts

Python simulators for cache and I/O analysis available in repository with full documentation.

References

[1] Patterson, D. A., & Hennessy, J. L. (2017). *Computer Organization and Design RISC-V Edition: The Hardware/Software Interface*. Morgan Kaufmann. ISBN: 978-0128122754

[2] RISC-V Foundation (2019). *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.2*. Available at: https://riscv.org/specifications/

[3] Hennessy, J. L., & Patterson, D. A. (2019). *Computer Architecture: A Quantitative Approach* (6th ed.). Morgan Kaufmann. ISBN: 978-0128119051

[4] Tomasulo, R. M. (1967). An Efficient Algorithm for Exploiting Multiple Arithmetic Units. *IBM Journal of Research and Development*, 11(1), 25-33.

[5] Smith, J. E. (1981). A Study of Branch Prediction Strategies. *Proceedings of the 8th Annual Symposium on Computer Architecture*, 135-148.

[6] Hill, M. D., & Smith, A. J. (1989). Evaluating Associativity in CPU Caches. *IEEE Transactions on Computers*, 38(12), 1612-1630.

[7] Jouppi, N. P. (1990). Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers. *Proceedings of the 17th Annual International Symposium on Computer Architecture*, 364-373.

[8] Anderson, D. P. (2003). You Don't Know Jack about Disks. *Queue*, 1(4), 20-30.

[9] Lee, E. A., & Seshia, S. A. (2017). *Introduction to Embedded Systems: A Cyber-Physical Systems Approach* (2nd ed.). MIT Press. ISBN: 978-0262533812

[10] Wolf, W. (2017). *Computers as Components: Principles of Embedded Computing System Design* (4th ed.). Morgan Kaufmann. ISBN: 978-0128053874