Silesian University of Technology

Faculty of Automatics, Electronics and Informatics

PhD Thesis

Multithreaded Selective Execution of a Control Program in Programmable Logic Controllers (PLC)

Michal Walichiewicz



Gliwice, 2026

ABSTRACT

Control systems are widely used in many aspects of modern life, ranging from industrial automation and transportation systems to smart cities and home automation. The increasing number of processes that must be controlled simultaneously, together with the demand for rapid and predictable responses, creates a need for solutions capable of increasing computational throughput while maintaining the required timing constraints. At the same time, the use of standardized programming methods is an important requirement in many control applications. One possible approach to increasing computational throughput is the use of parallel computation based on multi-core architectures. However, the adoption of multi-core architectures presents challenges for safety-critical systems such as Programmable Logic Controllers (PLCs), which are commonly used in industrial automation and control applications. In particular, the hard real-time constraints of such systems make multi-core synchronization problematic. Synchronization is typically performed using software semaphores that rely on cache-coherence mechanisms, which can result in non-deterministic synchronization timing. Furthermore, PLCs differ from many other safety-critical systems in that their applications are commonly developed using programming languages defined by the IEC 61131-3 standard.

The thesis presents the multi-core architecture and the characteristics of PLC-based safety-critical systems, with particular emphasis on deterministic execution and software separation. It explains how these requirements impose constraints on the software that can hinder its efficient execution on a multi-core microprocessor. Additionally, a detailed description of the compiler architecture and the complete compilation process is provided. Subsequently, the fundamental characteristics of PLC programs relevant to their compilation and parallel execution are examined.

A custom hardware-based semaphore mechanism and outline the differences between the proposed solution and other implementations that rely on Inter-Processor Interrupts (IPIs) for concurrency control in multi-threaded programs.

A custom hardware-based semaphore mechanism is presented, and the differences between the proposed solution and other implementations that rely on Inter-Processor

Interrupts (IPIs) for concurrency control in multi-threaded programs are outlined. Furthermore, it is shown how hardware-based semaphores significantly increase the predictability of multi-core synchronization in safety-critical systems and how the proposed solution is faster than Inter-Processor Communication (IPC). However, the proposed solution relies on automatic synchronization performed by the compiler rather than manual synchronization by programmers. Additionally, the coprocessor is described, and the integration of a custom coprocessor with an ALU for faster multi-core synchronization and computation is analyzed, with the proposed synchronization approach being compared with the Inter-Processor Interrupt (IPI) method.

A key requirement from the user's perspective is a compiler for standard PLC programming languages that can translate programs conforming to the relevant standards to the proposed platform while utilizing its available hardware features. This enables the resulting control system to execute PLC programs using parallel processing across multiple cores. Moreover, the use of fast hardware-based synchronization mechanisms enables efficient and predictable synchronization between concurrently executing processes. To achieve this, a custom dependency-aware pass within the LLVM compiler is presented, which automatically partitions the PLC program into multiple threads and integrates the proposed hardware-based semaphores. Finally, Rough Set Theory (RST) is applied to identify the most frequently occurring program-structure metrics associated with different multithreading performance outcomes. This analysis is performed on the results produced by the proposed algorithm, through which potential parallel execution opportunities are identified and the program is partitioned into two-thread and four-thread configurations.

Keywords: *Safety-Critical Systems*, *PLC*, *LLVM*, *Concurrency Control*, *Concurrent Compiler*, *Hardware Synchronization Mechanisms*, *Hardware Semaphore*, *Multi-Core Systems*, *LLVM*, *Coprocessors*, *System Control Coprocessors*, *Inter-Processor Interrupts*, *Distributed Mutual Exclusion*, *Distributed Shared Memory*.

ACKNOWLEDGEMENT

With immense pleasure and deep sense of gratitude, I wish to express my sincere thanks to my supervisor **PhD. DSc. Adam Milik**, Designation, Silesian University of Technology without his motivation and continuous encouragement, this research would not have been successfully completed.

I express my sincere thanks to **PhD. DSc. Dariusz Kania**, Dean, Silesian University of Technology for his kind words of support and encouragement.

I wish to extend my profound sense of gratitude to **my parents** for all the sacrifices they made during my research and also providing me with moral support and encouragement whenever required.

Place: Gliwice

Date: 01/07/2026 **Michal Walichiewicz**

Contents

ABSTRACT i

ACKNOWLEDGEMENT iii

LIST OF FIGURES viii

LIST OF TABLES xiii

LIST OF TERMS AND ABBREVIATIONS xiv

List of Terms and Abbreviations **xv**

1 Introduction **1**

1.1 Motivation 1

1.2 Thesis Statement, Aims and Objectives 2

1.3 Proposed Solution 4

1.3.1 Hardware 4

1.3.2 Software 5

1.4 Document Structure 6

2 Hardware Background and Literature Overview **7**

2.1 Introduction 7

2.1.1 PLCs and Safety-Critical Systems 7

2.1.2 Processors and Coprocessors 8

2.1.3 Concurrency Control and Synchronization 9

2.1.4 Inter-Processor Interrupts 10

2.2 Hardware Architecture 11

2.2.1 Core Architecture 11

2.2.2 Cache Architecture 12

2.2.3 Interconnection 13

2.3 Memory System 16

2.3.1 Cache Coherency . . . 16
2.3.2 Directory-Based Coherency . . . 18
2.3.3 Memory Consistency . . . 20
2.4 Synchronization . . . 24
2.4.1 Atomic Instructions . . . 24
2.4.2 Semaphore-Based Synchronization . . . 26
2.5 PLCs and Execution Models . . . 29
2.5.1 Concurrent Execution Models . . . 29
2.5.2 Execution Models of PLC . . . 30
2.6 System Control and Coprocessors . . . 31
2.6.1 System Control . . . 31
2.6.2 Coprocessor Architecture . . . 32
2.7 Inter-Processor Interrupts and Coprocessor . . . 35
2.7.1 Handling of Interrupts . . . 35
2.7.2 Multi-Core Synchronization . . . 39
2.8 Evaluation Criteria . . . 42
2.9 Summary . . . 43

3 Hardware Mechanisms for Fast Thread Synchronization 44
3.1 AXI Semaphores in Zynq . . . 44
3.2 Thread Synchronization in Multicore Processing Systems . . . 47
3.2.1 Multicore System and Synchronization Architecture . . . 47
3.2.2 Hardware Semaphore Design . . . 50
3.2.3 Mapping the Hardware Semaphores . . . 52
3.3 Coprocessor Design . . . 56
3.4 Synchronization in Multicore Processing Systems Evaluation Method . . . 60
3.5 Summary . . . 63

4 Results and Discussion of Synchronization in Multicore 65
4.0.1 ZYNQ SoPC hardware-based semaphore . . . 65
4.0.2 Hardware-based Semaphores . . . 67
4.0.3 Extended Synchronization Methodologies . . . 77

5 Evaluation and Conclusions 81

5.1 Evaluation of ZYNQ SoPC hardware-based semaphore . . . 81
5.2 Evaluation of Hardware-based Semaphores . . . 81
5.3 Evaluation of Extended Synchronization Methodologies . . . 82
5.4 Achieved Aims and Objectives . . . 82
5.5 Future Work . . . 83

6 Software Background and Literature Overview 84
6.1 Introduction . . . 84
6.1.1 Compilers . . . 84
6.1.2 PLC Programming Languages . . . 85
6.2 Front-End Compiler . . . 86
6.2.1 Lexical Analyzer . . . 86
6.2.2 Syntax Analyzer . . . 87
6.2.3 Semantic Analyzer . . . 88
6.2.4 IEC 61131-3 Standard . . . 90
6.3 Middle-End Compiler . . . 92
6.3.1 IR Code Analysis . . . 92
6.3.2 Code Optimization . . . 95
6.4 Back-End Compiler . . . 96
6.4.1 Instruction Selection . . . 96
6.4.2 Instruction Scheduling . . . 98
6.4.3 Register Allocation . . . 100
6.5 Multi-Core Compiler . . . 102
6.5.1 Locality . . . 102
6.5.2 Integer Linear Programming . . . 104
6.6 Evaluation Criteria . . . 105
6.7 Summary . . . 106

7 Dependency-Aware LLVM Parallelization Pass of Combinational Ladder Diagram 107
7.1 Hardware Semaphore Operations . . . 107
7.2 LLVM Intermediate Representation . . . 108
7.3 Dependency-Aware Thread Mapping . . . 113

7.4 DATM Algorithm Evaluation Method 119
7.5 Summary 119

8 Results and Discussion of DATM Algorithm 120
8.1 Multi-Core Synchronization Point Insertion 120
8.2 Thread Partitioning and Workload Distribution 122
8.3 Rough Set Analysis 133

9 Evaluation and Conclusions 137
9.1 Evaluation of Multi-Core Synchronization Point Insertion 137
9.2 Evaluation of the DATM Algorithm 137
9.3 Achieved Aims and Objectives 138
9.4 Future Work 138

List of Figures

1.1 Diagram of the Proposed Solution 4
2.1 Diagram of Two Heterogeneous Distributed Embedded Systems 7
2.2 Diagram of the Safety-Critical Software Architecture 8
2.3 MIPS Architecture Block Diagram 8
2.4 Diagram of Hardware Separations 9
2.5 Diagram of PowerPC 9
2.6 Diagram Of In-Order Execution Architecture 11
2.7 Diagram Of Superscalar Pipeline Architecture 11
2.8 Diagram Of Out-Of-Order Execution Architecture 12
2.9 Diagram Of Cache Hierarchy 12
2.10 Diagram Of Cache Architecture 13
2.11 Diagram Of Basic Designs of Multi-Core Interconnection 14
2.12 Diagram Of Bus-Based Interconnection 14
2.13 Diagram Of Models With Split-Transaction Bus 15
2.14 Diagram Of Network-On-Chip Topologies 16
2.15 Diagram Of Cache Incoherency 16
2.16 State Diagram Of MESI protocol 17
2.17 State Diagram Of MESI Transactions 17
2.18 State Diagram Of False Sharing in the MESI protocol 18
2.19 State Diagram Of A Directory-Based Coherence Protocol 18
2.20 State Diagram Of Global Broadcast in Directory System Model 19
2.21 State Diagram Of Routing Hops 19
2.22 Diagram of a Single-Threaded Program 20
2.23 Diagram of a Multi-Threaded Program 20
2.24 Diagram of a Core with a Store Buffer Implementation 21
2.25 Diagram of the TSO's store buffer enforcing its ordering rules 22

2.26 Diagram of Two Cores following the TSO's ordering rules 22
2.27 The Release Consistency Model of a Concurrent Program 23
2.28 Diagram of Operation on Shared Data Without Atomic RMW 25
2.29 Diagram of Atomic RMW Operation 25
2.30 Diagram of the Semaphore-Based Synchronization [39] 28
2.31 Diagram Of Parallel Execution Model 29
2.32 Diagram Of Concurrent Execution Model 29
2.33 The Diagram of a Simple PLC Module 30
2.34 The Diagram of a PLC Firmware 30
2.35 Example of Data Control of a Task [36] 31
2.36 Diagram of Integrated Architecture 31
2.37 Diagram of MIPS-based Architecture 32
2.38 Diagram of the Coprocessor Design 32
2.39 Diagram of the Floating-Point Coprocessor Design 34
2.40 Diagram of the Floating-Point Registers 35
2.41 Diagram of the Interrupt Handling by Coprocessor 38
2.42 Diagram of the Open Multi-Processor Interrupt Controller 39
2.43 Diagram of the OpenRISC Inter-Processor Communication 40
2.44 Diagram of the Xtensa Inter-Processor Communication 40
2.45 Diagram of the Xtensa Port-to-Port Communication 41
2.46 Diagram of the Xtensa IPC with the Queue Implementation 42
3.1 ZYNQ Platform Simplified Block Diagram of PS and PL Connection [36] 45
3.2 Semaphored Memory Cell Implementation [36] 46
3.4 Diagram Of the Multi-Core System Overview [38] 47
3.5 Comparison of Hardware and Software Semaphores [39] 48
3.6 Traffic Light Example [39] 49
3.7 Data Flow and Control Flow Graphs of the Example [39] 49
3.8 The Diagram of the Shared Semaphore Array [38] 50
3.9 Diagram of Semaphore Controller [39] 51
3.10 Diagram of Semaphore Read Controller [39] 51
3.11 Diagram of Semaphore Write Controller [39] 51
3.12 The Diagram of a Pipeline with the Shared Semaphore Interface [38] . . 53

3.13 The Diagram of the Submodule Handling IRQs [38] 56
3.14 The Diagram of a Pipeline with the Coprocessors [38] 57
3.15 Diagram Of the Custom Instruction Encoding Format [38] 58
3.16 Diagram Of Arithmetic Part of the Coprocessor [38] 59
3.17 Diagram Of IRQ Handling Part of the Coprocessor [38] 60
3.18 Diagram of the Core [39] 61
3.19 Diagram of the Quad Core system setup for performance assessment [39] 62
4.1 Comparison of Software and Hardware Supported Semaphore Systems Use [36] 65
4.2 Petri net model of single-slot bounded buffer 67
4.3 Execution model with Single Producer and Multiple Consumers [39] . . 68
4.4 Execution model with Multiple Producers and Single Consumer [39] . . 73
4.5 Execution model with Single Producer and Multiple Consumers using Hardware [39] 74
4.6 Execution model with Multiple Producers and Single Consumer using Hardware [39] 75
4.7 Execution model with Multiple Producers and Multiple Consumers [39] 76
5.1 Waveform of the Delays during Handling IRQ [38] 82
5.2 Waveform of the Delay between IRQs [38] 82
6.1 Diagram of the PLC Scan Cycle 85
6.2 Diagram of recognizing ”program” token by the Automata 86
6.3 Diagram of recognizing ”aa*b” token by Non-Deterministic Finite Automata 87
6.4 Diagram of a Abstract-Syntax Tree 89
6.5 Diagram of Three-Address Codes 89
6.6 Diagram of IEC 61131-3 Application Structure [23] 90
6.7 Diagram of PLC Program in LLVM 92
6.8 Diagram of the LLVM compiler architecture 93
6.9 Diagram of the CFG 93
6.10 Diagram of the DDG 94
6.11 Diagram of the DDG with IR Instructions 94
6.12 Diagram of the Dominator Tree [10] 95

6.13 Diagram of Inline Substitution . . . 96
6.14 Diagram of Back-End Processes . . . 97
6.15 Diagram of Superscalar Execution Stage . . . 98
6.16 Graph with Strict Dependencies . . . 99
6.17 Diagram of Cloned Paths in a Graph . . . 100
6.18 Diagram of Multi-Core Architecture . . . 102
6.19 Diagram of Divided Data Across Cache Lines . . . 104
6.20 Diagram of Vectorization of Data . . . 105
7.1 Diagram of Memory/WriteBack Pipeline Stage [40] . . . 108
7.2 DDG with Unconditional Data Flow [40] . . . 111
7.3 Conceptual Thread Partitioning of the DDG [40] . . . 111
7.4 Potential Synchronization Points Between Threads [40] . . . 112
7.5 Diagram of the Complete Compilation Workflow [40] . . . 113
7.6 Data Dependency Graph [40] . . . 114
7.7 Assigning First Color [40] . . . 115
7.8 Assigning Second Color [40] . . . 116
7.9 Assigning Third Color [40] . . . 116
7.10 All the Paths have an Assigned Color [40] . . . 117
7.11 Merging First Shortest Paths [40] . . . 117
7.12 Merging Third Shortest Paths [40] . . . 118
7.13 Program with completely mapped threads [40] . . . 118
8.1 Data Dependency Graph of Multi-Threaded Synchronization [37] . . . 121
8.2 Write Access to a Semaphored Memory Cell by Core 0 [37] . . . 121
8.3 Read Access to a Semaphored Memory Cell by Core 3 with Arbitration Delay [37] . . . 121
8.4 Ladder Diagram - 1 . . . 124
8.5 Ladder Diagram - 2 . . . 125
8.6 Ladder Diagram - 3 . . . 125
8.7 Heatmap for 2 Threads Speedup . . . 126
8.8 Heatmap for 4 Threads Speedup . . . 127
8.9 Heatmap for Relative Speedup . . . 128
8.10 Clean Heatmap for 2 Threads Speedup . . . 129

8.11 Clean Heatmap for 4 Threads Speedup [40] 130
8.12 Clean Heatmap for Relative Speedup 131

List of Tables

4.1 Average synchronization time using software and hardware semaphores 66
4.2 Test Case Results 80
8.1 Structural characteristics of the representative LD programs. 124

List of Terms and Abbreviations

IEC International Electrotechnical Commission

PLC Programmable Logic Controller

CIP Common Industrial Protocol

CAN Controller Area Network

WCET Worst-Case Execution Time

MILS Multiple Independent Levels of Security/Safety

GWV Greve, Wilding, and Vanfleet

RTOS Real-Time Operating System

API Application Programming Interface

DMR Dual Modular Redundancy

IDE Integrated Development Environment

LD Ladder Diagram

ST Structured Text

XML Extensible Markup Language

POU Program Organization Units

LLVM Low Level Virtual Machine

IR Intermediate Representation

LLIR Lower-Level Intermediate Representation

DFA Deterministic Finite Automata

NFA Non-Deterministic Finite Automata

AST Abstract Syntax Tree

BNF Backus–Naur Form

SSA Static Single Assignment

GVN Global Value Numbering

Mem2Reg Memory-to-Register Promotion

SROA Scalar Replacement of Aggregates

SCCP Sparse Conditional Constant Propagation

CFG Control-Flow Graph

DDG Data Dependency Graph

DAG Directed Acyclic Graph

ASAP As Soon As Possible

BFS Breath-First Search

DFS Depth-First Search

II Initiation Interval

FFT Fast Fourier Transform

ILP Integer Linear Programming

CPU Central Processing Unit

GPU Graphical Processing Unit

ALU Arithmetic Logic Unit

GPIO General Purpose Input/Output

I/O Input/Output

CISC Complex Instruction Set Computer

RISC Reduced Instruction Set Computer

ISA Instruction Set Architecture

ILP Instruction-Level Parallelism

OCB Operand Collection Buffer

RAT Register Alias Table

SIMD Single Instruction, Multiple Data

SIMT Single Instruction, Multiple Threads

MIMD Multiple Instruction, Multiple Data

IPC Inter-Processor Communication

IPI Inter-Processor Interrupt

ISR Interrupt Service Routine

DMA Direct Memory Access

UART Universal Asynchronous Receiver-Transmitter

LLC Last-Level Cache

NoC Network-on-Chip

PIC Programmable Interrupt Controller

NIU Network Interface Unit

RMW Read–Modify–Write

TSO Total Store Ordering

CAS Compare-and-Swap

LL/SC Load-Linked/Store-Conditional

PIC Programmable Interrupt Controller

EPC Exception Program Counter

PC Program Counter

BadVA Bad Virtual Address

PRId Processor Revision Identifier

TLB Translation Lookaside Buffer

TRNG True Random Number Generator

OMPIC Open Multi-Processor Interrupt Controller

RTL Register Transfer Level

FIFO First In, First Out

MPSoC Multi-Processor System-on-Chip

IRQ Interrupt Request

ERET Interrupt Return Instruction

LUT Look-Up Tables

PS Processing System

PL Programmable Logic

OCM On Chip Memory

FPGA Field-Programmable Gate Array

LUTs Look-Up Tables

HDL Hardware Description Language

Introduction

1.1 Motivation

Nowadays, multi-core architectures are commonly used in systems requiring increased computational performance, as they provide high computational performance by enabling parallel processing of multiple tasks. However, an important challenge in the design of multi-core systems is the partitioning of a computational task among multiple processing units. Such partitioning can be done manually, but this approach is time-consuming and may introduce errors related to the synchronization of computations. Following the partitioning of a computational task, the operations performed by multiple independent processing units must be coordinated within a common computational process. Ensuring the correct flow of data and control while avoiding race conditions is challenging; therefore, it remains an important issue in the design of systems based on multi-threaded computation [44, 45, 57, 60].

The same trend is also occurring in real-time systems, including safety-critical applications. This problem is especially critical because such systems operate under hard real-time constraints, which are particularly challenging for multi-core systems due to the time-consuming and non-deterministic synchronization of multi-threaded software [46, 76].

In a multiprocessor, cores are connected via a shared bus or an interconnection network; therefore, multi-core synchronization can take a significant amount of time because it requires executing software-based semaphore code and accessing shared memory, where contention for shared-memory access can block threads during synchronization and consequently increase synchronization overhead [57].

Therefore, in safety-critical applications such as employed using Programmable Logic Controllers (PLCs), multi-core synchronization cannot be subject to significant delays or non-deterministic execution due to the hard real-time requirements of such systems. Consequently, the proposed solution employs hardware-based semaphores that are independent of the cache-based memory hierarchy, with the aim of ensuring deterministic synchronization behavior. Moreover, the use of hardware-based semaphores instead of additional shared memory mechanisms simplifies and

accelerates synchronization by allowing synchronization to be performed directly through read and write operations to the semaphore-controlled memory space, without requiring the cores to execute software-based semaphore code.

1.2 Thesis Statement, Aims and Objectives

Motivated by the need for predictable and efficient execution of concurrent PLC control programs on multicore processors, particularly in safety-critical applications, the following thesis is formulated:

The efficient and predictable multi-context execution of PLC control programs, particularly LD programs composed of short, mutually dependent computational fragments, can be achieved through dedicated hardware-based semaphore synchronization mechanisms integrated into a multicore processor and a custom dependency-aware LLVM middle-end compiler pass that analyzes program dependencies in LLVM IR, automatically partitions the program into concurrent threads, balances their computational workload across the available cores, and inserts the required synchronization operations.

To achieve the aim of the thesis and verify the formulated thesis statement, the following research objectives are defined:

- analysis of hardware- and software-based synchronization mechanisms used in modern multicore processors, with particular emphasis on their timing behavior and predictability. The analysis will also examine how these synchronization mechanisms differ from the predictable and deterministic synchronization approaches required in safety-critical applications, such as employed using PLCs;
- analysis of alternative, deterministic multicore synchronization mechanisms employed in architectures such as Xtensa and OpenRISC, with particular emphasis on interrupt-based synchronization and the associated execution overhead of Interrupt Service Routines (ISRs);
- design and implementation of a dedicated hardware-based semaphore mechanism integrated into a multicore processor architecture, with the aim of enabling low-overhead, predictable, and deterministic synchronization between processor cores;
- comparative evaluation of conventional hardware- and software-based synchronization mechanisms used in modern multicore processors, assuming an idealized hardware interconnection between processor cores, against the

proposed hardware-based semaphore mechanism, with particular emphasis on synchronization overhead, predictability, and deterministic timing behavior;

- design and evaluation of a custom coprocessor with an integrated Arithmetic Logic Unit (ALU) for improving the performance of concurrent computations, including an idealized inter-processor interrupt-based synchronization mechanism designed to minimize software overhead and reduce the execution of the interrupt subroutine;
- investigation of the use of a custom coprocessor with an integrated ALU as a mechanism for improving the performance of concurrent computations, including a comparison between an idealized interrupt-driven approach, in which a vectorized system enables immediate detection of the requesting process and the processor core directly enters the ISR associated with the corresponding interrupt, and the proposed stalling-based synchronization approach;
- analysis of compiler architectures to identify and incorporate different compilation stages and multicore processing techniques, together with the introduction and analysis of PLC software, with particular emphasis on the ladder diagram programming language and its execution model;
- development of an automatic synchronization mechanism within the compiler middle-end that identifies computational dependencies between concurrently executing code fragments and automatically inserts the required LLVM IR instructions to implement synchronization using hardware-based semaphores;
- development of a dependency-aware LLVM middle-end compiler pass operating on LLVM IR that analyzes data and computational dependencies within PLC control programs and automatically partitions the program into fragments suitable for concurrent execution.

The research objectives are presented in detail in the following sections. The research problems are organized around two essential implementation challenges: the development of hardware-based synchronization mechanisms for multicore architectures and the implementation of a compiler for standard PLC programming languages capable of utilizing multicore architectures.

The results of the implementation are evaluated through a series of tests that enable comparison of the implemented features and performance with those of existing solutions. Finally, the complete implementation is evaluated to provide an overall assessment of its functionality and performance. To obtain a comprehensive overview of the results and performance indicators, selected control programs are analyzed in the context of parallel execution and different multithreading configurations. Rough

Set Theory (RST) is then applied to analyze the factors affecting the performance of multicontext PLC program execution and to identify the program-structure metrics most relevant to parallel execution.

1.3 Proposed Solution

The following diagram illustrates the proposed hardware and software solution and its implementation using the open-source LLVM and OpenPLC projects, with RusTy serving as the LLVM front-end and the default LLVM backend used for target code generation for the proposed multicore system.

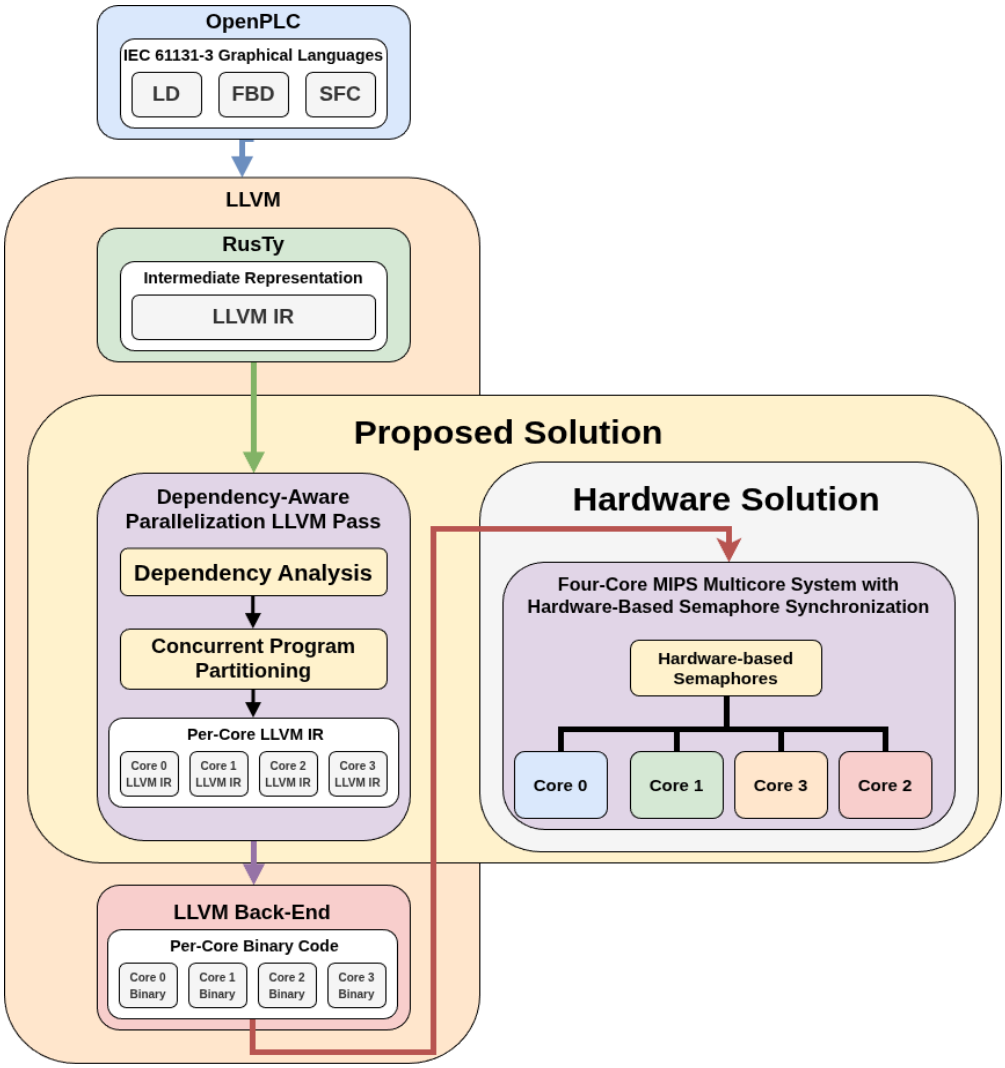


Figure 1.1 Diagram of the Proposed Solution

1.3.1 Hardware

An instruction represents an atomic operation at the architectural level, although its execution may consist of multiple internal operations. In early CPU implementations,

a microprogrammed approach was commonly used to control instruction execution. This approach was motivated by factors such as variable instruction lengths and the reduced hardware complexity. To improve instruction throughput, instruction execution is typically divided into multiple stages arranged as a pipeline. Depending on the architecture of the Central Processing Unit (CPU), the number of stages can vary. Each stage specializes in a different operation, such as fetching, decoding, or executing a given instruction. For the proposed hardware solution, the MIPS architecture is used, which consists of five pipeline stages, with the fourth stage responsible for accessing memory and memory-mapped I/O devices and other hardware peripherals [44, 45, 57, 60].

Hardware-based semaphores are accessed by the fourth pipeline stage as ordinary memory accesses, with the processor waiting for completion before proceeding to the next operation. However, this stage is adjusted by adding a semaphore controller, which allows an occupied hardware-based semaphore to stall a core's pipeline, akin to how the memory controller can halt pipeline execution. The multi-core system includes four integrated MIPS cores sharing hardware-based semaphores accessible to all cores.

Additionally, a custom coprocessor equipped with an integrated ALU has been developed and employed alongside hardware-based semaphores. In the MIPS architecture, the microprocessor features a primary coprocessor dedicated to handling interrupts and exceptions; it can also support optional coprocessors, including those for floating-point operations, as well as a custom one [71].

These coprocessors can be accessed in the second stage of the MIPS pipeline, one at a time. The custom coprocessor is directly connected to the shared hardware-based semaphores. Additionally, when a hardware-based semaphore is occupied, it triggers an interrupt that the custom coprocessor handles. Both interrupt handling and computation must be configured by a given core for this mechanism to operate correctly.

1.3.2 Software

Compilers need to convert a specific programming language into assembly code tailored for a selected Instruction Set Architecture (ISA). To do this successfully, the architectural description must be predefined within the compiler's back-end, enabling an effective transformation process from the programming language [1, 2, 10].

Programming languages are translated by the compiler's front-end. The front-end translates a high-level programming language into an intermediate representation, which enables the middle-end of the compiler to apply various passes for transforming and optimizing the compiled program [1, 2, 10].

A custom middle-end pass for the LLVM compiler has been designed to divide the compiled program into multiple threads and insert the appropriate LLVM IR

instructions to ensure proper synchronization of concurrent computations using semaphored memory cells. The back-end does not require modification, as the hardware-based semaphores are accessed in the same manner as local memory and other hardware peripherals.

The IEC 61131-3 standard defines several graphical and textual programming languages for describing control applications. These languages can express equivalent functional behavior while providing different representations and execution models. Therefore, the LD-to-ST translation capability provided by the OpenPLC toolchain is used to obtain a textual representation of the PLC program. An existing front-end, called RusTy, is then used to generate LLVM IR code from the resulting Structured Text (ST) representation. This approach was adopted because the Ladder Diagram (LD) representation is inherently Boolean-based and is evaluated sequentially rung by rung and from left to right. This preserves the combinational logic of the LD program during LD-to-ST translation and enables dependency information to be extracted from the resulting LLVM IR. This approach follows a principle similar to Hardware Description Language (HDL)-based design flows, where structural information is exploited to expose parallelism. [23].

1.4 Document Structure

The thesis is divided into two sections: hardware and software. Each section begins with a literature review that provides a thorough, discussion of the relevant background topics and their relationship to safety-critical systems and PLCs. After this, the methodology and results of the research are presented. Later, the evaluation and conclusion are provided.

The hardware section presents modern microprocessor architectures, followed by a discussion of cache systems, which creates the background and provides the motivation for the research described in the following sections. Two-level cache-based multi-core architectures are described; cache coherence and memory consistency are discussed, and the relationship between them and multi-core synchronization is shown.

The other mechanisms of multiprocess synchronization is based on interrupts that is a subject of analysis; furthermore, coprocessor architectures are demonstrated, including their use for interrupt handling and additional computational features such as floating-point computation. These discussions provide the foundation for analyzing interrupt handling and the use of IPIs for multi-core synchronization in the Xtensa and OpenRISC architectures.

The software section provides a thorough description of the three parts of a compiler: the front-end, middle-end, and back-end. For each part, the relevant algorithms are described to illustrate the compilation process from start to finish.

Hardware Background and Literature Overview

2.1 Introduction

2.1.1 PLCs and Safety-Critical Systems

Programmable Logic Controllers (PLCs) used in safety-critical applications can be considered heterogeneous distributed embedded systems comprising networks of distributed microprocessors. These systems are commonly interconnected through industrial communication networks, such as Controller Area Network (CAN) in automotive applications and Common Industrial Protocol (CIP)-based networks in industrial automation. The primary contribution of this PhD thesis is the introduction of a deterministic multicore architecture targeting the main processing unit of PLC-based systems. While primarily designed for the main PLC processor, the proposed architecture can also be integrated into other processing nodes within the PLC system, as illustrated in Fig.2.1.

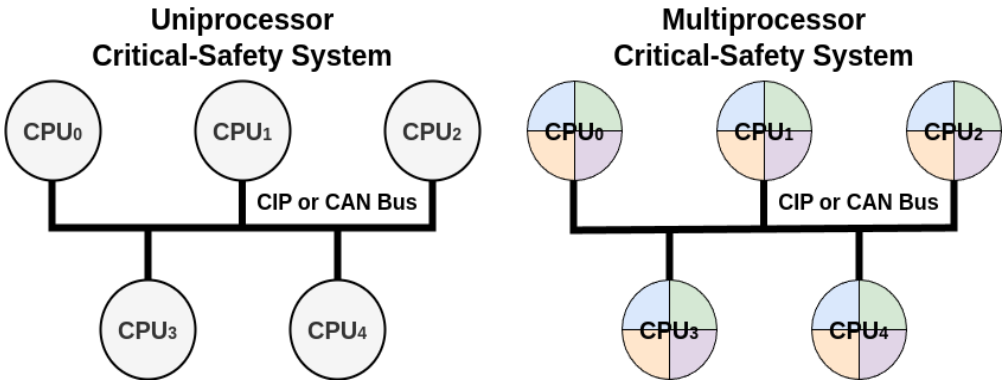


Figure 2.1 Diagram of Two Heterogeneous Distributed Embedded Systems

The microprocessor's software architecture is based on a separation kernel that follows the Multiple Independent Levels of Security/Safety (MILS) architecture, as shown in Fig.2.2. This architecture separates software system modules into strictly isolated partitions based on the Greve, Wilding, and Vanfleet (GWV) separation model. This ensures that data within each partition cannot be corrupted by neighboring partitions and that the system can isolate a faulty partition at run time. Furthermore,

it implements noninterference information flow control as defined by Rushby's noninterference model, ensuring that information flow between software modules does not interfere with the original data. In this model, a given partition can only read a copy of the original data from an observed neighboring partition [14, 21, 66].

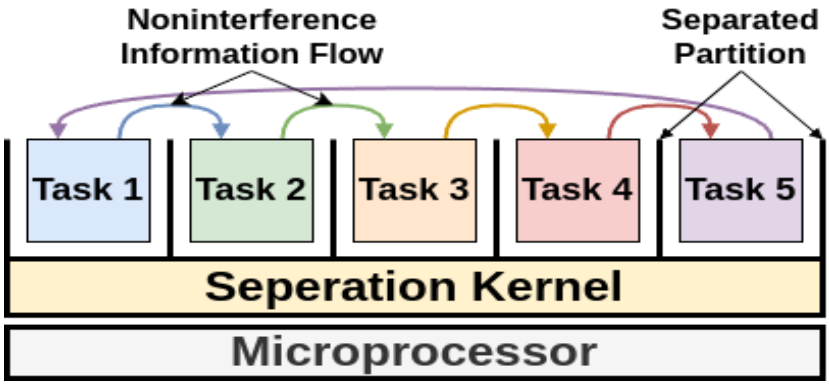


Figure 2.2 Diagram of the Safety-Critical Software Architecture

2.1.2 Processors and Coprocessors

In embedded systems, various types of microprocessor architectures are used; the most common are ARM and MIPS. Both architectures are based on the RISC design philosophy, which aims to maximize energy efficiency; however, they differ in the hardware modules responsible for floating-point computation and system control functionalities. These functionalities include interrupt handling, exception processing, and memory management. MIPS-based microprocessors place system control and floating-point hardware modules outside the main processor, in coprocessors that must be accessed via specialized instructions, as illustrated in Fig.2.3. Meanwhile, ARM-based microprocessors integrate floating-point and system control modules directly within the main processor [26, 33].

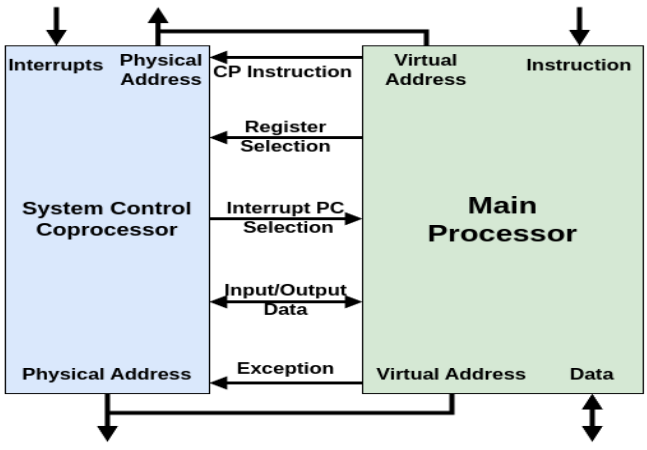


Figure 2.3 MIPS Architecture Block Diagram

This separation between the primary processor and the system control coprocessor separates critical control functions from general-purpose computation. Such functional separation is conceptually similar to the MILS architecture principle and can potentially mitigate the risk of fault propagation between hardware components. The concept of hardware-level fault isolation and functional separation is depicted in Fig.2.4 below.

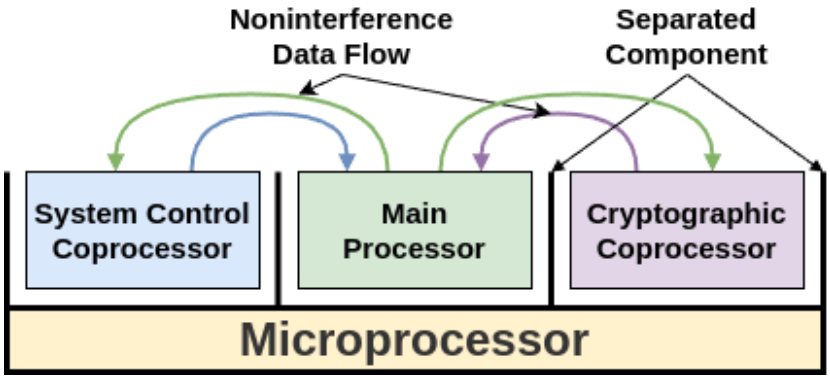


Figure 2.4 Diagram of Hardware Separations

Certain high-performance microprocessors, such as OpenPower (PowerPC), incorporate specialized processing units, as illustrated in Fig.2.5. These specialized processing units are not classified as coprocessors, as they are integrated directly into the processor's execution pipeline, rendering the processor nonfunctional if any of them are removed. This type of CISC architecture facilitates the concurrent execution of multiple processes without causing pipeline stalls, in contrast to traditional RISC processors [7].

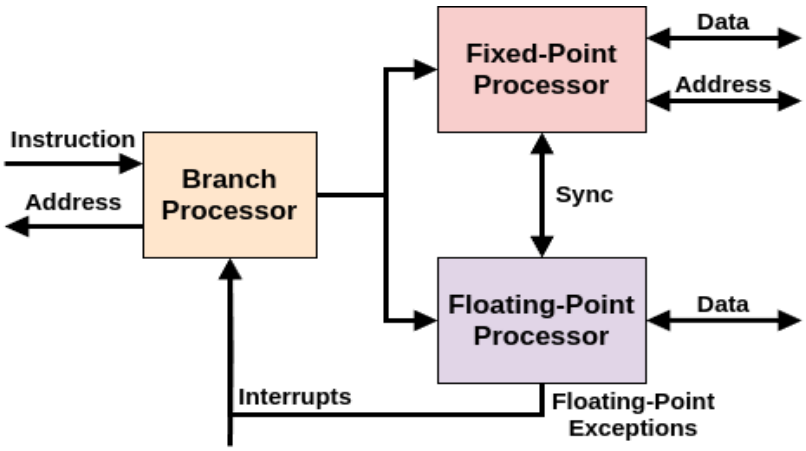


Figure 2.5 Diagram of PowerPC

2.1.3 Concurrency Control and Synchronization

Multi-core architectures pose a serious challenge for safety-critical applications due to their strict real-time constraints. The most common multi-core architectures

use cache-based synchronization to manage concurrent control. Due to the non-deterministic nature of cache architectures, a multi-core system may miss a task completion deadline, which could result in fatal consequences for human life. Multi-core synchronization needs to minimize the Worst Case Execution Time (WCET). One way is to use specialized software that can analyze WCET in cache, or to use interrupt-based synchronization or hardware-based semaphores [46, 76].

2.1.4 Inter-Processor Interrupts

Hardware peripherals that generate interrupts can range from simple General-Purpose Input/Output (GPIO) pins and timers to Direct Memory Access (DMA) controllers and various communication interfaces, such as Universal Asynchronous Receiver–Transmitter (UART) interfaces. Interrupts are a fundamental mechanism in microprocessors that enables an almost immediate response to external events. An essential part of an interrupt system is the rapid detection of the source requesting service. This is achieved by interrupt mechanisms that allow the processor to transfer control to the appropriate Interrupt Service Routine (ISR) without requiring continuous polling of the event source. The corresponding software implementation is also required to handle the interrupt and perform the requested operation. In Inter-Processor Communication (IPC), interrupt-based multi-core synchronization mechanisms are called Inter-Processor Interrupts (IPIs) and operate similarly to interrupts generated by hardware peripherals [65, 74].

There is no single IPC architecture; instead, each microprocessor architecture implements its own IPC mechanism with a specific method for multi-core synchronization. The two most common mechanisms are analyzed and compared with the proposed multi-core synchronization. The first mechanism uses a dedicated hardware module that combines shared memory and a Programmable Interrupt Controller (PIC), connected to multiple processor cores through a shared bus. In OpenRISC, this mechanism is used to provide IPC between processor cores.

The second mechanism uses a direct core-to-core communication system. When a core wants to transfer shared data to another core, it writes the data to a dedicated register associated with the communication system on its local bus, which is also connected to the local memory and peripheral devices. The write operation then triggers the corresponding interrupt on the target core. This hardware-based approach can also employ a dedicated hardware queue interface with push and pop operations, often combined with DMA engines in system-level designs. This mechanism is found in Xtensa [65, 74].

2.2 Hardware Architecture

2.2.1 Core Architecture

A pipeline design is provided to highlight the key challenges when designing a processor's pipeline in a multi-core system. Figure 2.6 presents a single-issue, in-order, 3-stage pipeline, commonly found in very low-power systems with small computational requirements. It should be emphasized that the low energy consumption results from the minimalist design [44, 45, 57, 60].

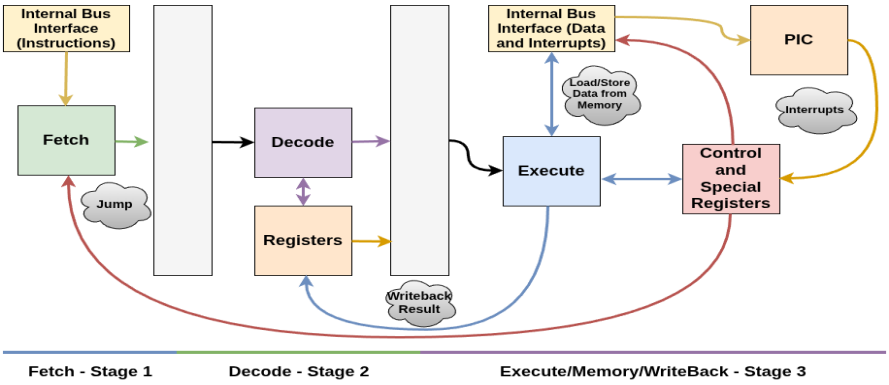


Figure 2.6 Diagram Of In-Order Execution Architecture

A common pipeline optimization is the introduction of instruction-level parallelism, as shown in Fig.2.7. By implementing parallel decode and execute modules, the core can execute multiple instructions concurrently. This architectural design is called superscalar [44, 57].

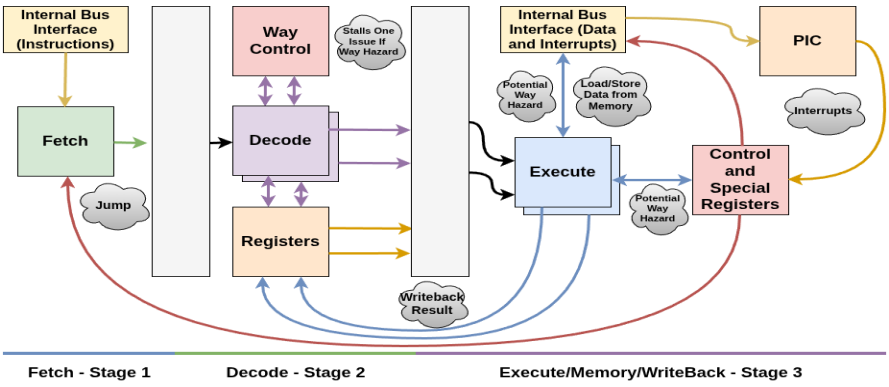


Figure 2.7 Diagram Of Superscalar Pipeline Architecture

Instruction-level parallelism facilitates the utilization of out-of-order execution. The pipeline architecture is shown in Figure 2.8. This architecture implements a reservation station and a common data bus at the execution stage, thereby supporting instruction reordering to improve execution efficiency and superscalar performance. Concurrently, the order manager monitors instructions and addresses data hazards. The pipeline

employs an instruction dependency unit that determines instruction dependencies based on the current processor state, allowing instructions to be issued when their required operands become available. The pipeline also includes an Operand Collection Buffer (OCB) for managing instruction operands and a Register Alias Table (RAT) for tracking register dependencies during out-of-order execution [44, 57].

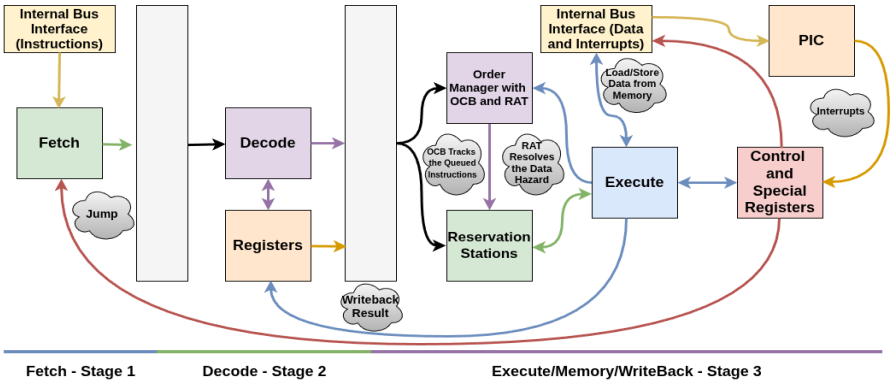


Figure 2.8 Diagram Of Out-Of-Order Execution Architecture

2.2.2 Cache Architecture

Main memory is implemented using high-capacity dynamic memory, which has a significantly higher access latency than the processor's execution speed. To reduce memory access latency and prevent the processor from frequently stalling while waiting for data, high-speed cache memory is implemented between the main memory and the processor cores. Cache memory is based on fast static memory implemented on the same die as the CPU. However, the high-speed operation required from cache memory limits its capacity, resulting in significantly smaller cache sizes compared with main memory. A cache hierarchy, as shown in Fig.2.9, is therefore implemented, with smaller and faster caches located closer to the processor core [45, 60].

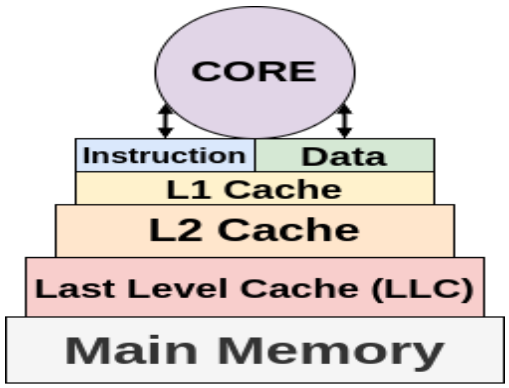


Figure 2.9 Diagram Of Cache Hierarchy

The fastest and smallest cache is the L1 cache, which is typically divided into separate instruction and data caches. This organization provides a Harvard-based

structure at the L1 level, allowing instruction and data accesses to be handled separately. At higher cache levels, the instruction and data caches are typically unified into a common cache, progressively transitioning from the Harvard-based organization at the processor level toward a unified, von Neumann-style memory organization. Within the cache hierarchy, inclusive or exclusive policies are employed to determine whether a given cache level also stores data blocks present in lower-level caches [44].

An inclusive Last-Level Cache (LLC) contains copies of the L1 and L2 caches. This design improves block eviction performance. By contrast, an exclusive L2 cache does not contain any data blocks found in the L1 cache. This reduces the L2 cache's capacity requirements, but it reduces the speed of the block eviction. The fundamental unit of cache architecture is the cache line, which stores memory blocks. Typically, the number of sets equals the number of cache lines, and each cache line is usually 64 bytes [57].

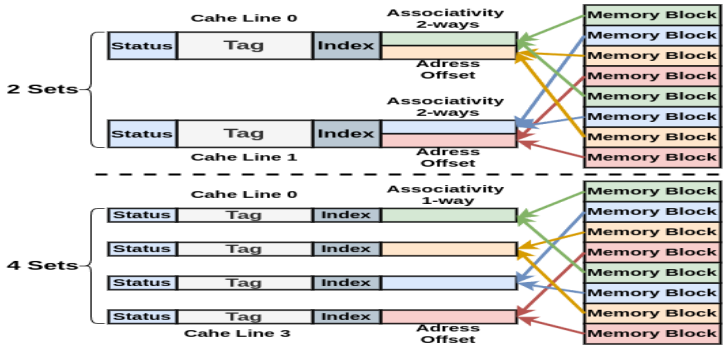


Figure 2.10 Diagram Of Cache Architecture

Fig.2.10 demonstrates the dynamics of the caching system. To reduce uncoordinated bus usage between caches and main memory, each cache line fetches an entire block of data. The tag stores the most significant bits of an address, which remain after the address offset and index bits. The index segment of the cache lines allows each line to be associated with a specific memory segment. Cache lines typically have multiple ways, allowing each line to be mapped to multiple memory blocks. However, this design introduces the issue of false sharing among numerous local caches [44, 57].

2.2.3 Interconnection

The process of designing a multi-core system should not be portrayed as merely a canonical methodology that yields a universally applicable architecture across applications. In addition to designing the cores and caches, prototyping the system's interconnection to facilitate communication among the cores constitutes a fundamental component of the multiprocessor architecture. The fundamental designs are illustrated in the preceding Fig.2.11 to emphasize the basic concepts of multi-core communication [57].

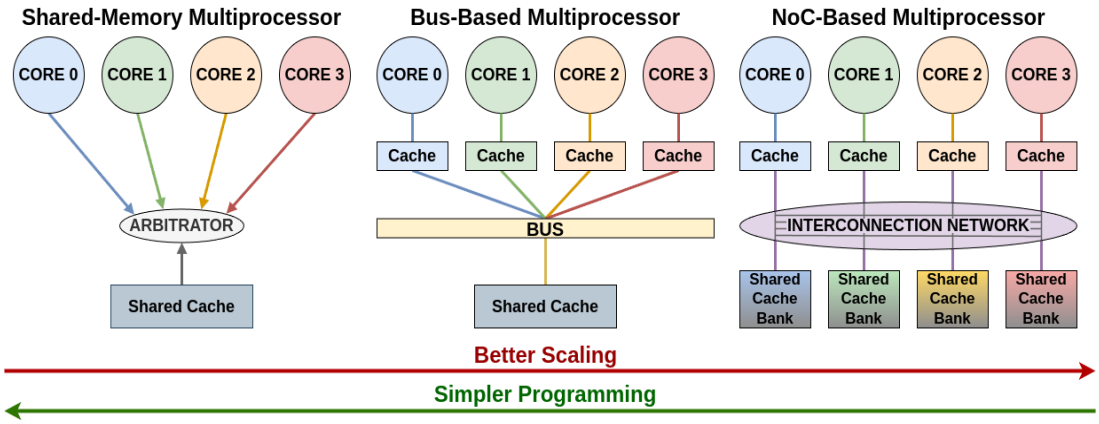


Figure 2.11 Diagram Of Basic Designs of Multi-Core Interconnection

Fundamentally, the interconnection addresses conflicts that occur when multiple cores share system memory and facilitates communication among cores. The fundamental digital module in an interconnection system is the arbitration unit, which manages access to shared resources such as common peripherals. In each clock cycle, access is granted to the requester with the highest priority, while the priority is adjusted between arbitration cycles to prevent starvation of lower-priority requesters. A single arbitration unit can be employed to implement a very simple yet non-scalable interconnection; however, such a design is largely impractical [57].

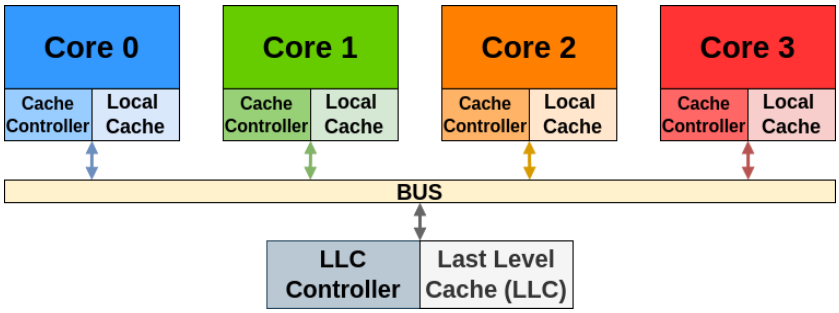


Figure 2.12 Diagram Of Bus-Based Interconnection

Bus-based interconnection designs exhibit limited scalability due to incremental access starvation as additional cores are incorporated into a multiprocessor system. Nonetheless, a shared bus facilitates the prompt detection of cache incoherency through bus snooping and permits straightforward data exchanges between local caches, while simultaneously updating shared data within the lower-level cache [57].

In addition to transactions initiated by the cores, essential communication among bus snoopers to sustain cache coherency across multiple caches generates additional traffic. This may place substantial strain on the bus; however, bus-based architectures are not limited to a single bus, as depicted in Fig.2.12. Multiple buses can be used to separate different types of traffic, as exemplified by the two-bus configurations illustrated in Fig.2.13. In a generic organization, separate buses can be used for request

and response transactions or for instruction and data transfers. Such separation reduces contention between different types of traffic and can improve bus utilization and overall multiprocessor performance [57].

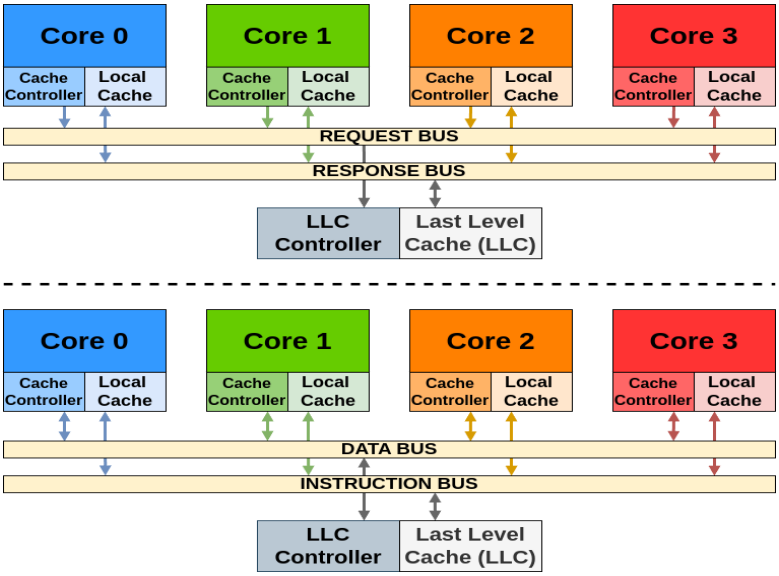


Figure 2.13 Diagram Of Models With Split-Transaction Bus

The Network-on-Chip (NoC) is another pivotal concept in the design of multi-core interconnections. NoC-based architectures offer enhanced scalability in comparison to bus-based systems; however, the two approaches differ in how communication is managed. In bus-based systems, a controller initiates a transaction, while other connected devices act as targets of the transaction. Conversely, NoC-based systems rely on point-to-point packet transfers, with routers forwarding packets between network nodes. The header of each packet specifies the transaction type and the source and destination nodes. The routers are responsible for forwarding packets toward their destinations, while the cores connected to the network nodes initiate and process the corresponding transactions [57].

Figure 2.14 illustrates two exemplary network topologies. It is evident that each core is associated with its own router and tile, which encapsulates the core's caches and essential hardware modules such as the PIC, which is connected to the core to provide interrupt handling, and the Network Interface Unit (NIU), which connects the tile to the NoC and manages the exchange of network traffic between the tile and the network. Tiles may also comprise a multi-core cluster interconnected via either a bus-based or NoC-based architecture. The selected topology significantly influences network latency, bandwidth, and the average number of router links, commonly referred to as hops. Furthermore, routing algorithms play a crucial role in affecting network latency by determining the path a packet follows to reach its intended destination [57].

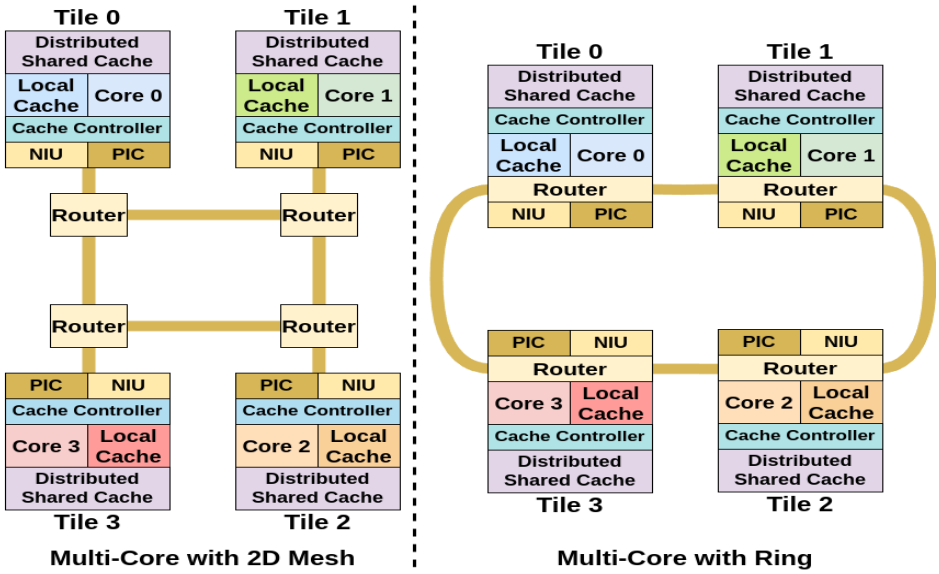


Figure 2.14 Diagram Of Network-On-Chip Topologies

2.3 Memory System

2.3.1 Cache Coherency

The red line in Fig.2.15 depicts cache incoherence, where the caches fail to maintain coherence of shared data, causing the second core to operate on outdated information. As discussed in the previous section, all communication in bus-based systems is visible to all cores; therefore, shared data is synchronized by bus snoopers that implement a cache coherence protocol. This protocol consists of a set of rules that track cache-line states across multiple caches and take appropriate actions based on the cores' actions to maintain coherence of shared data [57, 58].

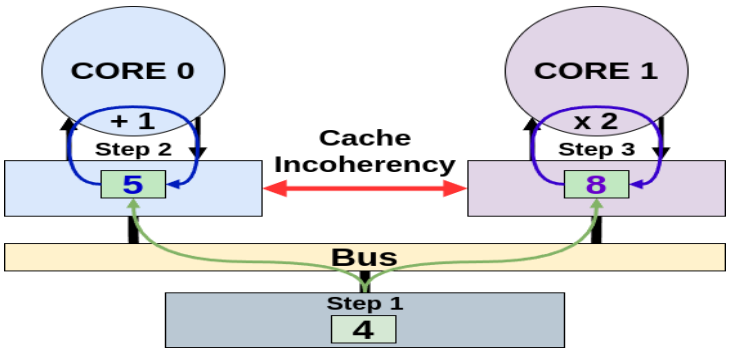


Figure 2.15 Diagram Of Cache Incoherency

There are various cache coherence protocols, and the selection of a specific protocol is determined by the overall CPU design, including factors such as design complexity, transaction overhead, and write-back behavior. Furthermore, various cache coherence protocols may be employed at different levels of the cache hierarchy within a microprocessor [57, 58].

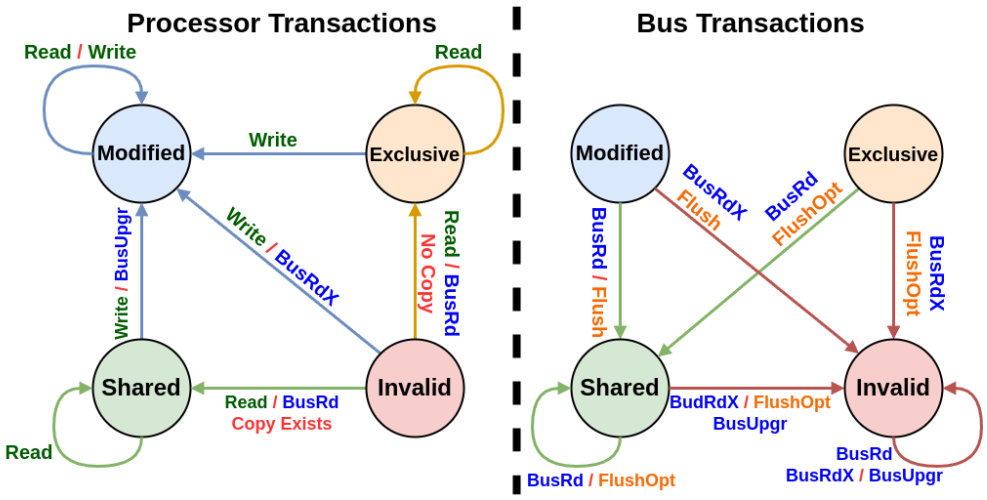


Figure 2.16 State Diagram Of MESI protocol

The Modified, Exclusive, Shared, and Invalid (MESI) protocol is a cache coherence protocol used to maintain a consistent view of shared data across multiple local caches. Figure 2.16 shows the MESI state-transition diagram, representing the states of each cache line in each local cache. The state of a cache line changes based on actions performed by the local core and by other cores that access or modify the corresponding data. A bus snooper may issue a data request to another bus snooper associated with the cache containing the most recent copy of the shared data. When a core's bus snooper requests the latest version of the data from another core, the second core's bus snooper flushes the shared data onto the bus, thereby enabling the requesting core to read the data and update both its local cache and the lower-level shared cache. Figure 2.17 shows the MESI transaction state diagram, which represents cache-line states during shared-data transactions across multiple cores [57, 58].

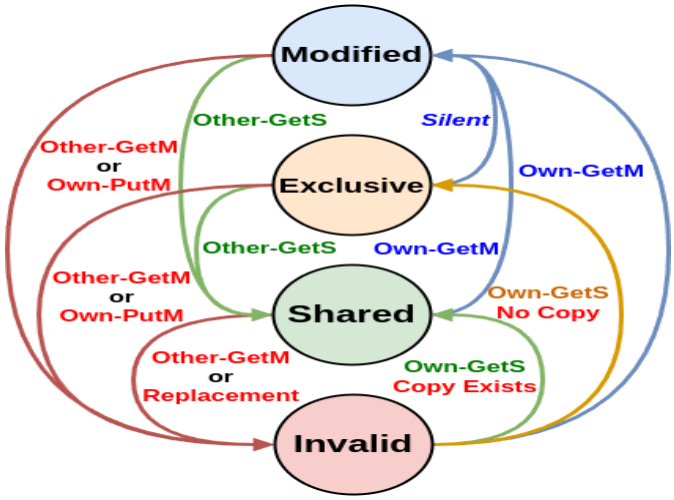


Figure 2.17 State Diagram Of MESI Transactions

A cache line's state is represented as an additional segment field within the cache line. As previously demonstrated, multiple memory blocks may be mapped to the same cache line, which can cause different cores to use the same cache line for distinct memory blocks simultaneously; nevertheless, this does not violate cache coherence. Fig.2.18 illustrates the phenomenon of false sharing in multi-core systems; it does not corrupt computation, but it results in significant performance loss because much time is wasted on unnecessary data fetches and write-back operations [57, 58].

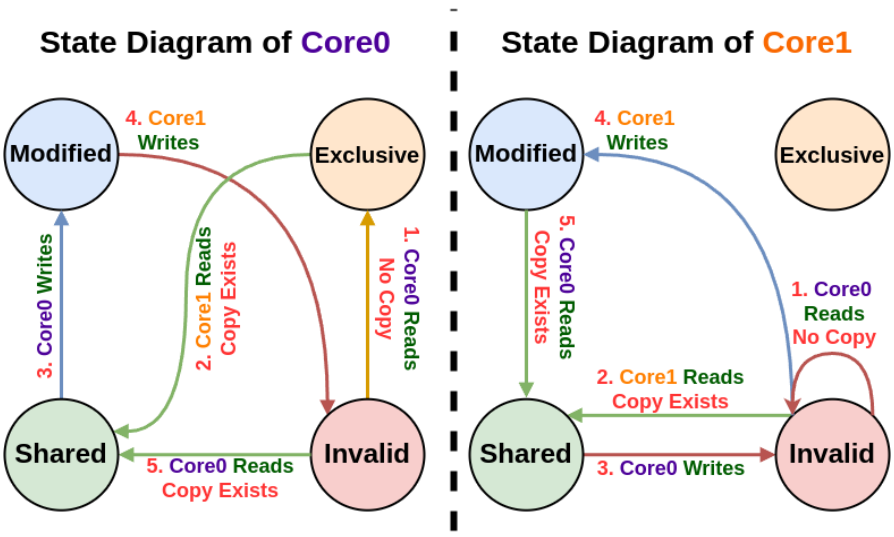


Figure 2.18 State Diagram Of False Sharing in the MESI protocol

2.3.2 Directory-Based Coherency

Bus snooping is prone to heavy bus traffic. This occurs because all point-to-point communications are broadcast globally, creating a scalability limitation. NoC–based multi-core systems use a directory-based cache coherence protocol, as depicted in Fig.2.19 below [57, 58].

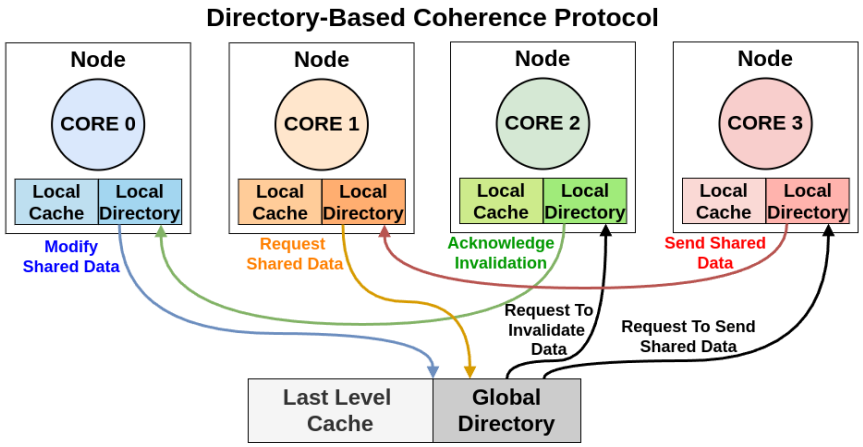


Figure 2.19 State Diagram Of A Directory-Based Coherence Protocol

A directory-based system model is a broadcast protocol with a point-to-point interconnection network. This model enables more scalable distribution of core-to-core communication because routers can effectively navigate transmissions between cores. The routing algorithm and the topology of the interconnection network can significantly impact communication latency [57, 58].

In a multi-core directory-based system, each node maintains a local directory to ensure data coherence, making each node partially responsible for keeping data up to date. Nevertheless, global broadcasting remains indispensable for servicing data requests and invalidating outdated data on other nodes, as shown in Fig.2.20 [57, 58].

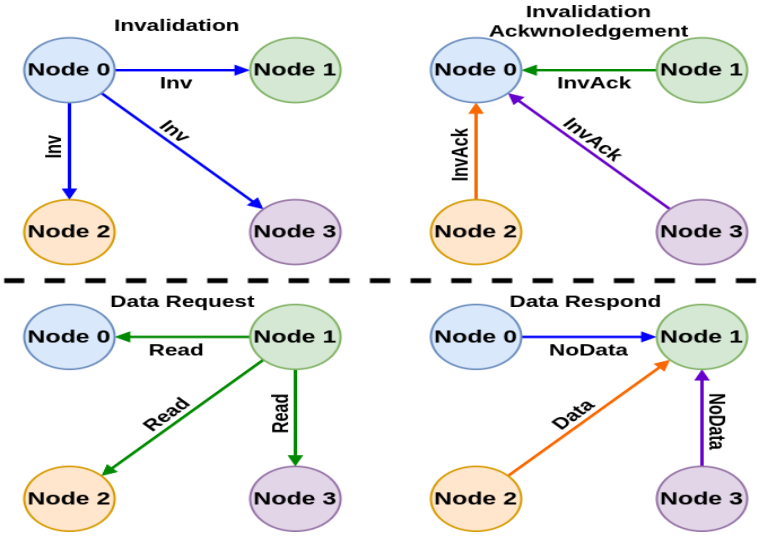


Figure 2.20 State Diagram Of Global Broadcast in Directory System Model

Communication within the interconnection network uses packets consisting of a data payload and a header. The header contains information about the transaction type and the source and destination identifiers, enabling routers to route each packet correctly [57, 58].

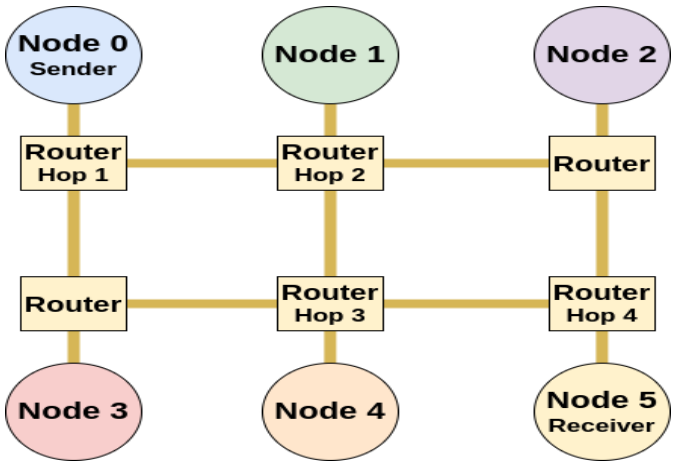


Figure 2.21 State Diagram Of Routing Hops

Nevertheless, transmission latency is predominantly influenced by the number of hops that a packet must traverse through multiple routers to reach its destination. Figure 2.21 illustrates a scenario in which a packet originating from node 0 requires four hops to reach node 5. Routing latency can have devastating consequences in safety-critical PLC applications, where swift system responses are vital in life-threatening circumstances [57, 58].

2.3.3 Memory Consistency

Fig.2.22 shows a program consisting of five tasks executed as a single-threaded application. The program has a clear control flow, illustrated by the black arrows; task dependencies are indicated by task colors.

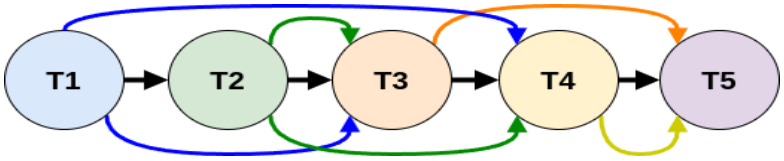


Figure 2.22 Diagram of a Single-Threaded Program

Subsequently, Figure 2.23 presents the concurrent version of the previously described program, scheduled using a greedy As Soon As Possible (ASAP) approach. As observed from the diagram, the multi-threaded program is mapped according to the dependency flow, as discussed in a later section of this thesis.

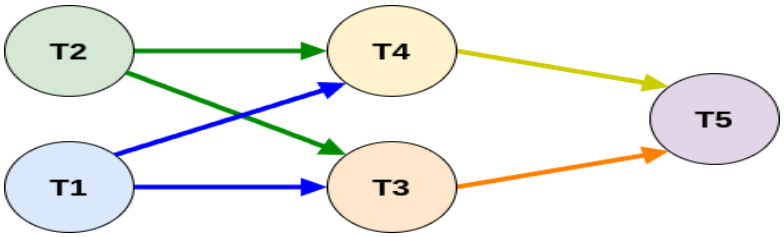


Figure 2.23 Diagram of a Multi-Threaded Program

At first glance, a single-threaded program exhibits a clear sequential execution order; however, in a concurrent program, the execution order is constrained by data dependencies. For example, the T3 or T4 task may execute before the data dependencies of earlier tasks have been satisfied. Consequently, the execution order of concurrent tasks is not explicitly defined and may vary between executions. This phenomenon introduces the memory consistency problem, which relates to the correct ordering of memory operations during concurrent execution.

It is intuitively plausible to use volatile variables as shared flags at synchronization points to direct control flow during execution; however, it might not be immediately apparent from a software perspective that this indirect method of establishing

synchronization points is unsustainable. It is notably time-consuming and fails to prevent race conditions, which might result in deadlocks or non-deterministic execution. Moreover, the ability of volatile variables to support synchronization depends on how they are used and how the synchronization information is encoded; volatile access alone does not guarantee atomic Read–Modify–Write (RMW) operations when shared synchronization state is updated concurrently. It is evident that concurrent execution must conform to ordering rules that ensure sequential consistency in multi-core systems, as Lamport delineates below [57, 58].

Definition 2.3.1 : Sequential Consistency

A system is sequentially consistent if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.

Fundamentally, memory consistency models enforce ordering rules on load, store, and fence operations to guarantee the correctness of concurrent execution. Total Store Ordering (TSO) is among the most well-known models due to its widespread adoption across various architectures and its implementation of the store buffer, as illustrated in Figure 2.24 [57, 58].

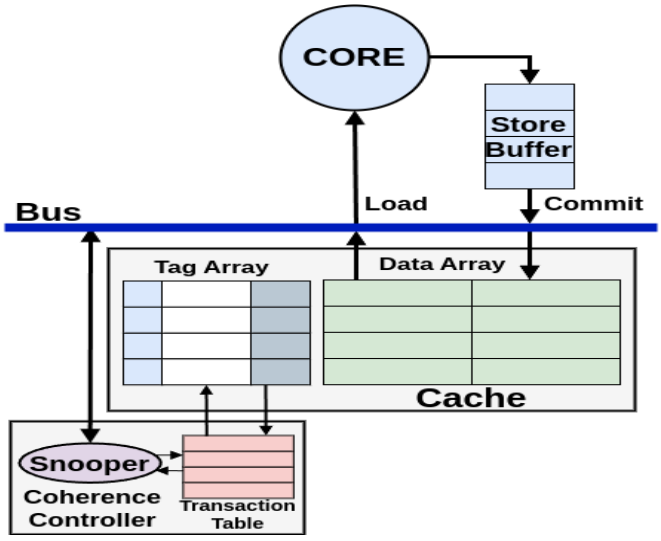


Figure 2.24 Diagram of a Core with a Store Buffer Implementation

A store buffer is a FIFO queue that prevents a core from writing data anywhere other than at the queue's head. When the store buffer is flushed, its entries are committed to the cache. This mechanism enforces the TSO ordering rule that prohibits a load from bypassing a prior store to the same memory address, as shown in the left diagram of

Fig.2.25. Meanwhile, the right diagram depicts two successive loads from the same address executed under the same ordering rules [57, 58].

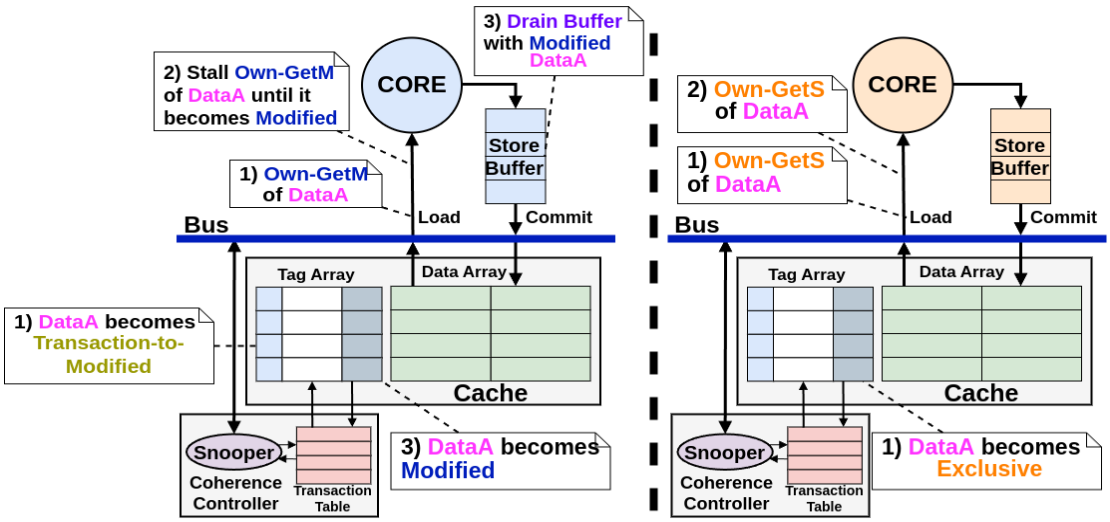


Figure 2.25 Diagram of the TSO's store buffer enforcing its ordering rules

Data coherence across multiple cores enables memory consistency to be enforced in multi-core systems, as shown in Fig. 2.26. In the diagram, Core 1 modifies the shared data while Core 2 simultaneously requests access to the same data. Core 2 is stalled while Core 1's modified data remains in the store buffer. Once the data is written back from the store buffer to Core 1's local cache, the cache controller updates the corresponding cache-line state to Modified. Core 2 can then proceed with its request, and the cache coherence mechanism updates the cache-line states in both caches to Shared, ensuring that both cores have access to the same updated data.

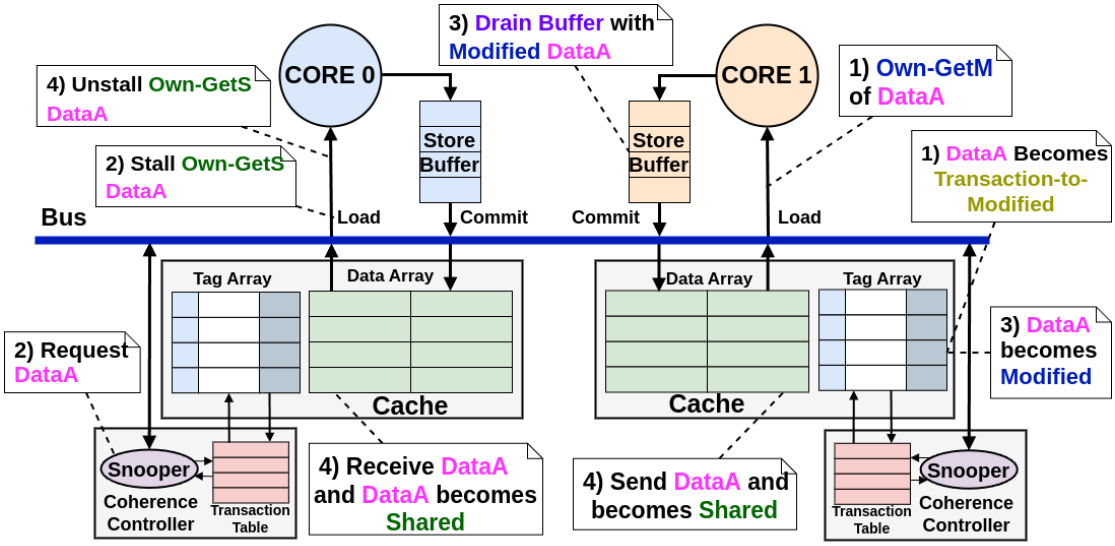


Figure 2.26 Diagram of Two Cores following the TSO's ordering rules

Strict memory consistency models impose stringent ordering rules on load and store operations. Under these rules, a core is not authorized to execute a load or store instruction concerning other cores until all prior load and store operations have been finalized. These rigorous models facilitate a more autonomous approach to multi-core memory consistency, as demonstrated in Fig.2.26 [57, 58].

Strict memory consistency models hinder multi-threaded performance because each core must preserve a single global memory order. Consequently, relaxed memory consistency models, such as release consistency, have been formulated. This paradigm uses atomic locks and enforces global ordering exclusively during lock acquisition and release. By relying on atomic locks in critical sections, the release consistency model allows a concurrent program's consistency to be driven by the implemented atomic locks at those sections; therefore, it offers better multi-core performance. The following Fig.2.27 illustrates that each task may execute load and store operations concurrently, while the lock mechanism ensures synchronization among the threads of the multithreaded program to preserve global memory consistency [57, 58].

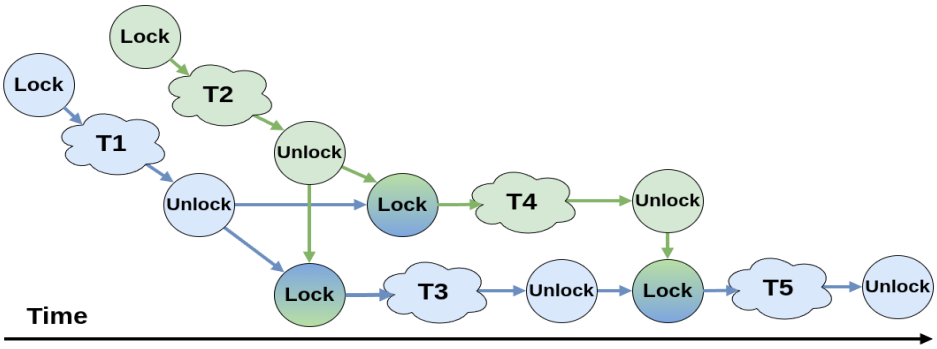


Figure 2.27 The Release Consistency Model of a Concurrent Program

2.4 Synchronization

2.4.1 Atomic Instructions

It should be evident from the preceding sections that a locking mechanism is essential to ensure that concurrent accesses to shared resources are properly synchronized. This hardware-supported mechanism is implemented using atomic RMW operations. Although multiple atomic instructions are available, each performs a sequence of read, modify, and write operations as a single, indivisible process within a given atomic instruction. [57, 58].

Compare-and-Swap (CAS) and Load-Linked/Store-Conditional (LL/SC) are among the most commonly used atomic primitive instructions in modern multicore CPU architectures, such as MIPS and ARM, as they enable the implementation of shared-data algorithms for an arbitrary number of threads. However, CAS can suffer from the ABA problem in multi-core systems, leading to unpredictable behavior and, in critical cases, deadlock. The presence of private caches in each core makes this phenomenon both possible and severe [51].

Listing 1. Atomic Compare-And-Swap

```
1: procedure CAS(Data_Pointer, Old, New)
2:     if Data_Pointer ≠ Old then
3:         return False
4:     else
5:         Data_Pointer := New
6:         return True
7:     end if
8: end procedure
```

Listing 2. Atomic LL/SC MIPS Assembly Code

```
1: lock :
2:     ll $t0, 0($a0)
3:     bnez $t0, lock
4:     addiu $t1, $t0, 1
5:     sc $t1, 0($a0)
6:     beqz $t1, lock
7:
8: unlock :
9:     sw $zero, 0($a0)
10:
```

It is dangerous to assume that concurrent algorithms do not require synchronization mechanisms in systems with strict memory consistency models, such as PLC-based

control systems and safety-critical systems. Multiple cores may simultaneously load a shared cache line before any core has acknowledged its status. However, the primary importance of atomic instructions lies in preventing multiple threads from entering a critical section that contains both instructions and shared data. As shown in Fig.2.28, multiple cores can enter a critical section concurrently even under a strict memory consistency model, because the instructions and shared data may reside in different memory segments, leading to unpredictable behavior. Therefore, atomic primitive instructions are implemented to synchronize a multi-core system that shares the same data [57, 58].

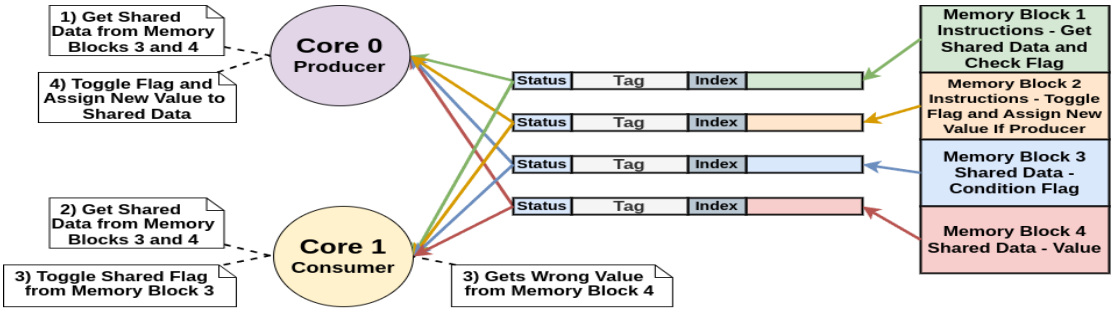


Figure 2.28 Diagram of Operation on Shared Data Without Atomic RMW

Shared-data synchronization is expected to have data races due to differences in the completion times of individual computation tasks. These data races can corrupt computation results and lead to incorrect program behavior. Atomic instructions allow a thread to acquire a lock for a critical section of a program and let other threads wait until the lock is released. Thus, atomic operations ensure correct data flow across tasks, independent of individual task completion times. Fig.2.29 depicts the procedure required for core 1 to acquire and release an atomic lock for a critical section [57, 58].

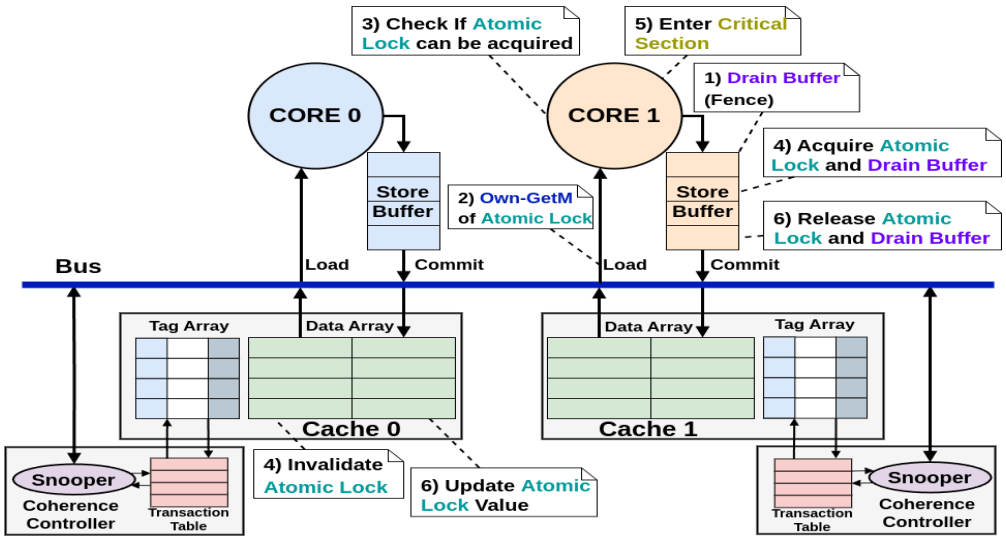


Figure 2.29 Diagram of Atomic RMW Operation

2.4.2 Semaphore-Based Synchronization

The simplest synchronization mechanism proposed by Dijkstra was the semaphore, which is used to coordinate concurrent processes. Dijkstra introduced two fundamental semaphore operations, namely wait (also known as P) and signal (also known as V), as shown in the following pseudo-code. During each operation, the semaphore variable S is accessed through its shared memory location, with its current value being read and the updated value subsequently written back to memory. [12, 51, 57, 58].

Listing 3. Dijkstra Semaphore Operations

```
1:  procedure P(s)
2:      atomically
3:      s := s − 1
4:          if s < 0 then
5:              block the calling task
6:          end if
7:      end atomically
8:  end procedure
9:
10: procedure V(s)
11:     atomically
12:     s := s + 1
13:         if s ≤ 0 then
14:             wake one waiting task
15:         end if
16:     end atomically
17: end procedure
```

Given that the semaphore's value is either 0, indicating that the semaphore is acquired, or 1, indicating that it is free, they operate as mutual exclusion locks. This configuration permits only a single thread to acquire the semaphore at any given moment. The pseudo-code presented below illustrates a critical section protected by a binary semaphore.

Listing 4. Critical Section Protection by a Semaphore

```
1: procedure SEMAPHORE
2:     lock.P()
3:     /* Critical Section */
4:     lock.V()
5: end procedure
```

The employment of binary semaphores as mutual exclusion locks for critical sections is essential in concurrent programming, as they prohibit multiple threads from accessing shared resources concurrently. Moreover, binary semaphores serve as fundamental components in the development of more advanced synchronization

mechanisms, such as counting semaphores used in bounded buffers and single-slot producer-consumer buffer, as illustrated by the subsequent pseudo-code.

Listing 5. Single-Slot Producer-Consumer Buffer

```
1:  procedure PRODUCER(data)
2:      empty.P()
3:          lock.P()
4:          buffer := data
5:          lock.V()
6:      full.V()
7:  end procedure
8:
9:  procedure CONSUMER
10:     full.P()
11:         lock.P()
12:         data := buffer
13:         lock.V()
14:     empty.V()
15:     return data
16: end procedure
```

A single-slot producer-consumer buffer can be constructed using three binary semaphores, as shown in the pseudo-code above. The *lock* semaphore provides mutual exclusion when accessing the shared buffer, while the *empty* and *full* semaphores provide conditional synchronization between the producer and consumer. The latter two semaphores can be viewed as implementing a simple handshake exchange: *empty* indicates that the buffer is available for a new message, whereas *full* indicates that a message is available for consumption. The figure below (2.30) represents the hardware operations required to perform semaphore-based synchronization between two cores. [51, 57, 58].

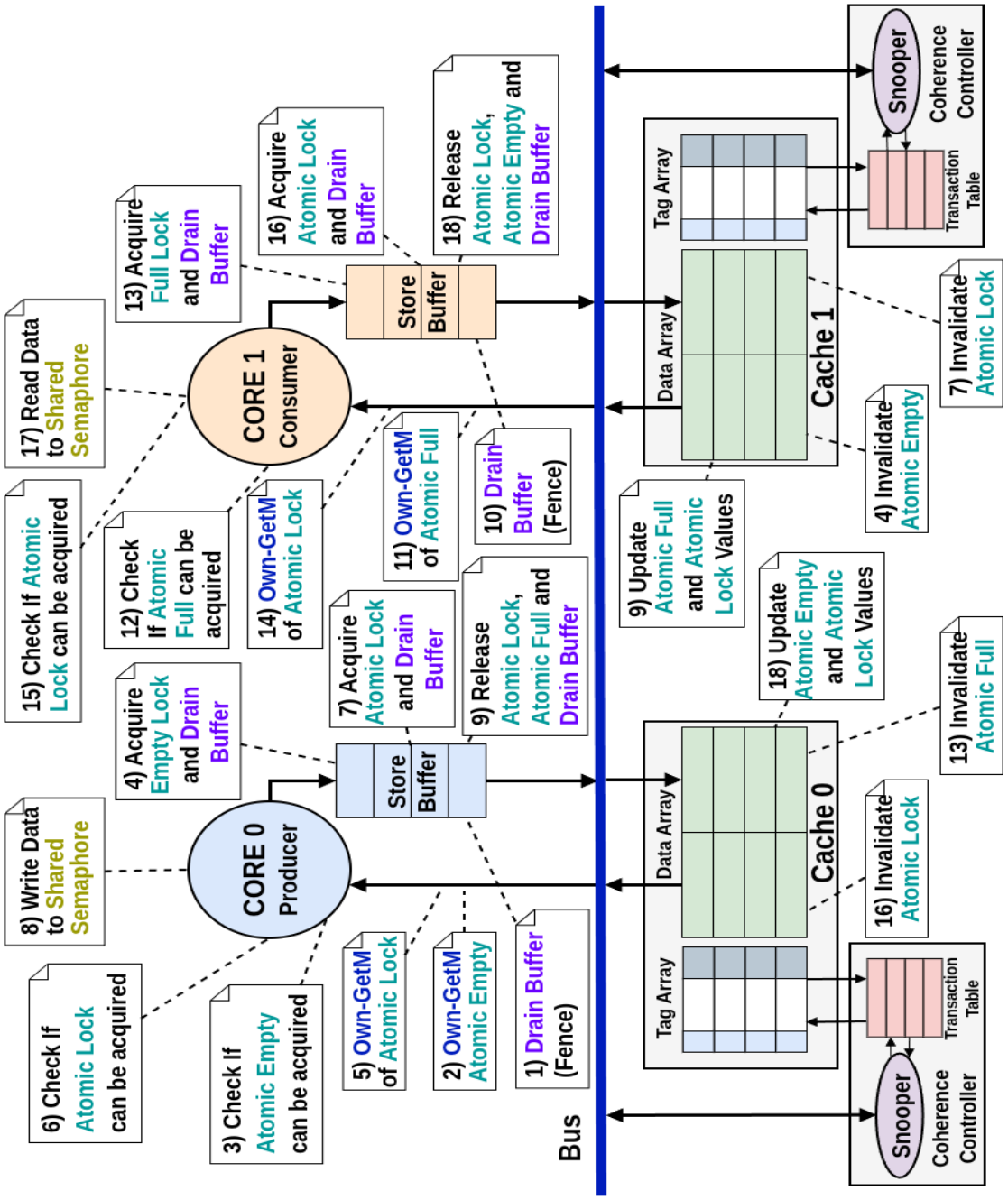


Figure 2.30 Diagram of the Semaphore-Based Synchronization [39]

2.5 PLCs and Execution Models

2.5.1 Concurrent Execution Models

There are two fundamental data processing formats. A graphics processing unit (GPU) comprises multiple cores that share a common cache and are orchestrated collectively by a main core. This architecture enables parallel processing of large datasets composed of mutually independent data elements by applying the same computational formula [44].

Fig.2.31 illustrates a typical parallel execution model for GPUs. Each element of a dataset is processed concurrently by the same task. GPU compilers generate multi-core code by vectorizing an intermediate representation (IR) within the compiler's optimization pipeline [10, 44].

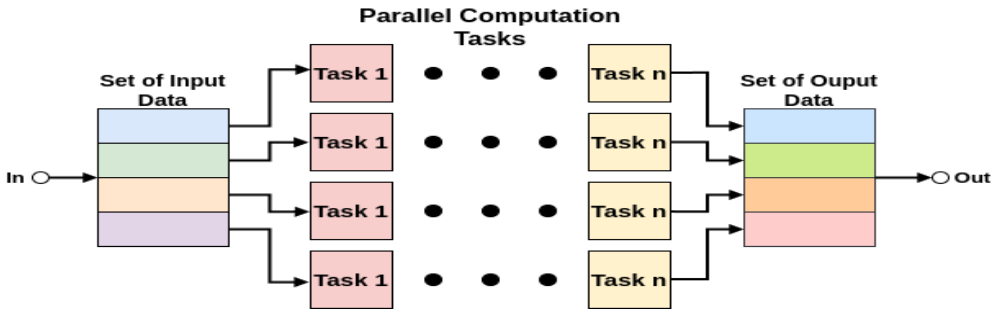


Figure 2.31 Diagram Of Parallel Execution Model

The model described herein classifies within the Single Instruction Multiple Data (SIMD) multiprocessing framework. Tasks such as neural network training and image processing naturally benefit from SIMD because they process large volumes of data using the same computational operations. Concurrently, single- and multi-core microprocessors support parallel and diverse concurrent processing across mutually inclusive or exclusive data sets, which is characterized as the Multiple Instruction Multiple Data (MIMD) multiprocessing paradigm, as illustrated in Fig.2.32. Central Processing Units (CPUs) predominantly execute multiple independent threads that share data [44, 45, 60].

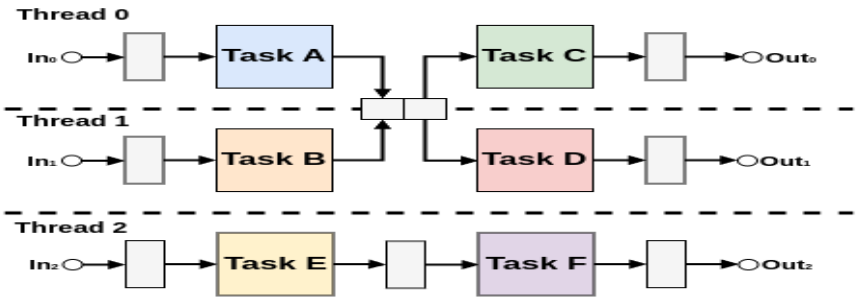


Figure 2.32 Diagram Of Concurrent Execution Model

A general-purpose CPU typically runs an operating system that uses multiple independent threads to execute processes concurrently. Semaphores (or similar mechanisms) are used to meet computational dependencies between tasks. On multi-core microprocessors, the operating system is also responsible for managing core utilization and distributing threads across the available cores [44, 45, 60].

2.5.2 Execution Models of PLC

PLCs are industrial control systems commonly used in safety-critical applications. They are typically heterogeneous, distributed, embedded systems composed of a network of microprocessors connected via the CIP. Fig.2.33 depicts a simplified PLC module with a main microprocessor and four input/output ports, each controlled by a subordinate microprocessor.

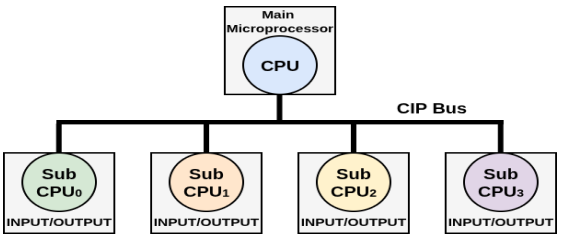


Figure 2.33 The Diagram of a Simple PLC Module

From a programmer's perspective, a PLC is programmed using standardized, dedicated programming languages for control systems, such as Ladder Diagram (LD) and Structured Text (ST). The user program is executed as part of a processing loop managed by the respective operating system. The nature of PLC-based systems requires cyclic operation, with the cycle interval constrained by the available time of the controlled process or processes.

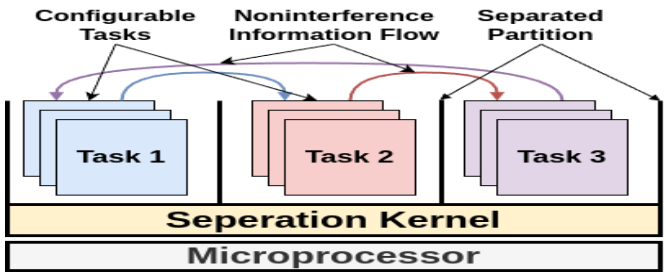


Figure 2.34 The Diagram of a PLC Firmware

As shown in Fig. 2.34, the firmware incorporates a MILS architecture. This architecture ensures that tasks are separated into isolated partitions and that data flow between tasks is non-interfering, for example, through single-slot producer-consumer buffers. The PLC firmware/system is responsible for managing the processing loop. The system collects input data, executes user tasks, and subsequently distributes the

computation results. This configuration is provided via an uploaded manifest file, which contains the necessary information to initialize and configure all tasks [21, 66].

Due to imposed safety requirements, the firmware is prohibited from using dynamic memory allocation. Furthermore, task configuration is limited to static modifications of the software. As a result, task execution remains deterministic and bounded, while allowing conditional execution paths depending on the evaluated inputs and system state. This enables the depiction of a task's data control flow in Fig.2.35.

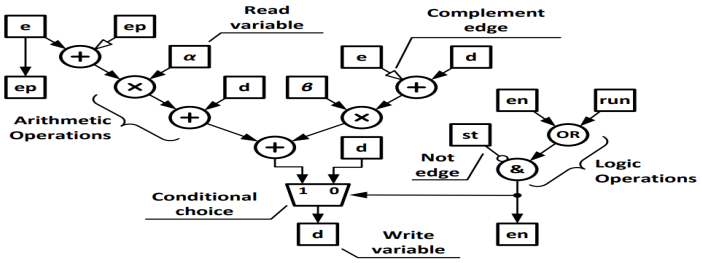


Figure 2.35 Example of Data Control of a Task [36]

2.6 System Control and Coprocessors

2.6.1 System Control

Fig. 2.36 depicts a single-issue, in-order, three-stage pipeline architecture. In other words, instruction processing is divided into three pipeline stages. The architecture does not incorporate a dedicated coprocessor for system control or memory-management functions; instead, these functions are implemented within the main processor core and executed as part of the processor's instruction set. Processor architectures such as ARM and OpenRISC may employ similar integrated approaches. However, integrating multiple system functions within the same processor core can limit the hardware-level separation between these functions. This lack of separation may increase the potential for faults in one function to affect the operation of others [33, 74].

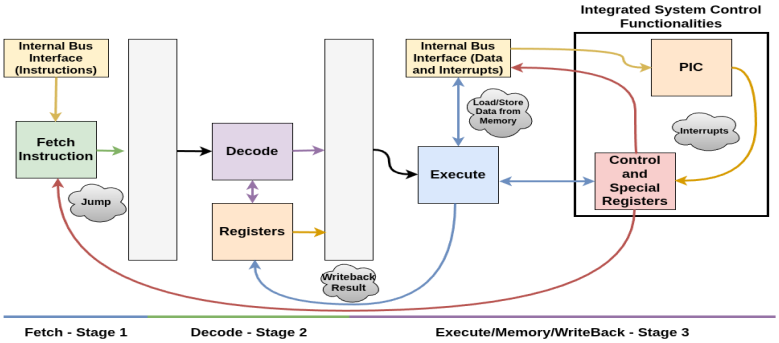


Figure 2.36 Diagram of Integrated Architecture

A second common pipeline design used in embedded systems is based on the MIPS architecture, which includes a system control coprocessor alongside the main processor, as depicted in Fig.2.37 [26, 71].

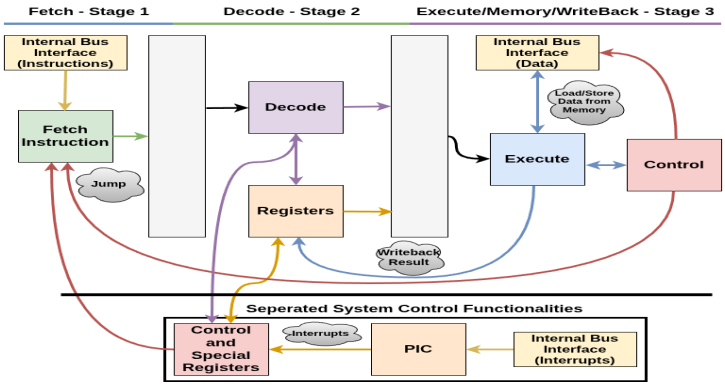


Figure 2.37 Diagram of MIPS-based Architecture

In this architecture, the Programmable Interrupt Controller (PIC) is decoupled from the main execution pipeline and implemented within a dedicated system control coprocessor. The coprocessor operates alongside the main processor and provides system-level functions, including interrupt and exception handling, as well as memory-management support. This architectural separation distinguishes system-control functions from the main instruction-execution pipeline and can provide a basis for improved fault isolation. However, the effectiveness of such isolation depends on the mechanisms governing the interaction between the coprocessor and the main processor, including interrupt handling, priority management, and memory-protection mechanisms. Therefore, the separation of system-control functions into a dedicated coprocessor should be considered as an architectural measure that can facilitate fault isolation rather than as a guarantee of complete fault containment [26,56].

2.6.2 Coprocessor Architecture

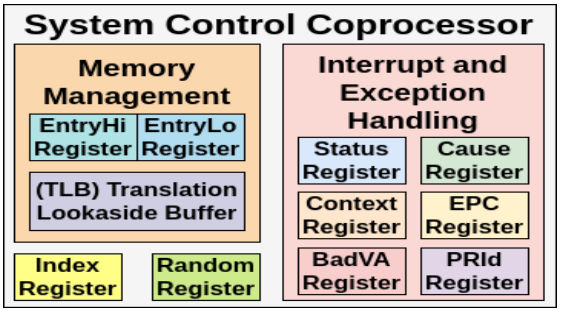


Figure 2.38 Diagram of the Coprocessor Design

Fig.2.38 shows a typical system control coprocessor, which comprises a set of registers designated for managing various system control functions. These registers

convey vital information on the status of processing modes, system configuration, triggered interrupts, encountered exceptions, timer values, and other operational parameters [26, 71].

The Cause register contains information about the system's exceptions and interrupts. The Exception Program Counter (EPC) register stores the current address of the Program Counter (PC) when an exception or interrupt is triggered. This allows the main processor to resume execution from the stored address after handling exceptions or interrupts. The Bad Virtual Address (BadVA) register stores the virtual address that triggered a virtual address exception. The Context register contains information about the system's current state, address space, and Translation Lookaside Buffer (TLB), which the kernel uses to manage transitions among multiple processes or threads. Each bit in the Status register indicates the configurable status of a specific system control function. The Processor Revision Identifier (PRId) register specifies the processor's revision and version, while the Random register stores a value generated by the True Random Number Generator (TRNG).

Furthermore, the coprocessor is responsible for memory management, which includes address translation through the TLB. The TLB contains entries that map virtual addresses to physical addresses for a given page or block of memory; meanwhile, the Index register is used to select a specific TLB entry. The EntryHi and EntryLo registers hold essential information associated with a TLB entry, including the virtual page identifier and the corresponding physical page frame information. In the event of a TLB miss, these registers provide the information required for address translation and TLB updating. Additionally, the main processor moves data to and from the coprocessor registers using specialized instructions, as demonstrated in the following MIPS assembly code [26].

Listing 6. Move To/From Coprocessor 0 MIPS Assembly

```
; Coprocessor Number is the Instruction's Digit
; First Reg is Used for the Data Movement
; Second Reg is the Coprocessor Number
; Last 3 Bits is Selection Type

Move_To :
    mtc0 $t0, $a0, 0

Move_From :
    mfc0 $t0, $a0, 0
```

Coprocessors are not limited to handling system control functions; they can also perform specialized tasks such as floating-point calculations or cryptography.

The following Fig.2.39 illustrates a floating-point coprocessor that uses dedicated instructions to perform floating-point arithmetic operations. This allows specialized computations to be performed by the coprocessor in coordination with the main processor and its instruction pipeline.

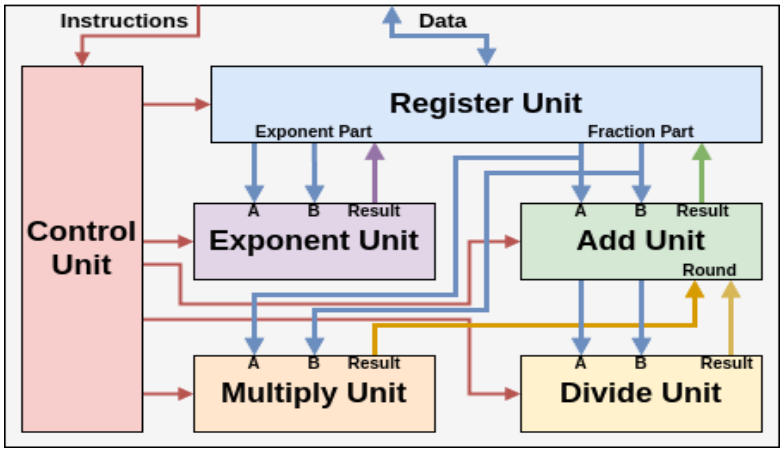


Figure 2.39 Diagram of the Floating-Point Coprocessor Design

The coprocessor contains a control unit that decodes specialized instructions for floating-point arithmetic operations, as well as move instructions. These instructions are shown in the MIPS assembly code below [26, 71, 72].

Listing 7. Floating-Point Coprocessor 1 MIPS Instruction

```
; Single Precision Instructions end with .s
; Double Precision Instructions end with .d
; Only Even Numbered Registers are used
; in Double Precision Instructions

FloatingPoint_Subtraction :
    sub.s $f0, $f1, $f2

FloatingPoint_Multiplication :
    mul.d $f2, $f4, $f6

```

Computation coprocessors implement a processor pipeline with multiple stages corresponding to the execution of their instructions, such as decoding and execution. Their operation can be configured by setting the corresponding bits in the Status register. In addition to the Status and PRId registers, computation coprocessors also have general-purpose registers similar to those found in a typical microprocessor; however, these registers are typically specific to the type of computation being performed, as a floating-point coprocessor has floating-point general-purpose registers, which are shown in Fig.2.40 [26, 71, 72].

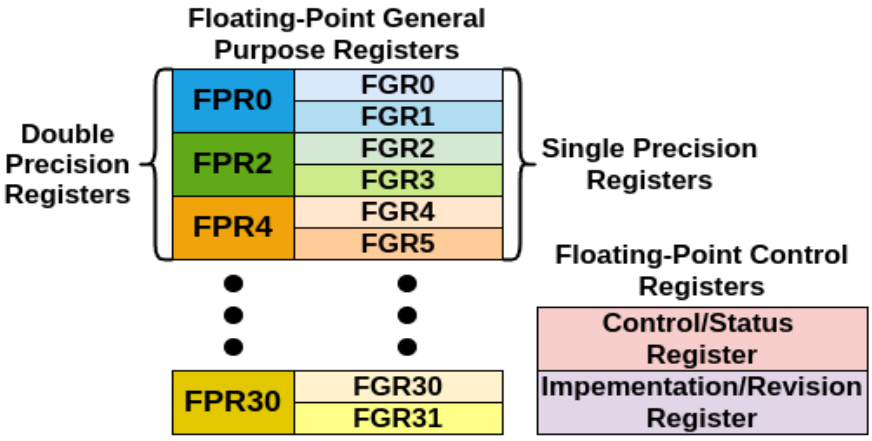


Figure 2.40 Diagram of the Floating-Point Registers

2.7 Inter-Processor Interrupts and Coprocessor

2.7.1 Handling of Interrupts

Interrupts and exceptions are input signals that set the corresponding bits in the Cause register of the system control unit, which is typically implemented as the system control coprocessor (CP0) in MIPS. Exceptions are synchronous events detected during program execution, such as arithmetic overflow or an attempt to access an invalid memory address. On the other hand, interrupts are typically generated asynchronously by peripherals, such as a UART or DMA controller, to notify the processor of a hardware event. When an interrupt or exception occurs, the current PC is stored in the EPC, and the processor transfers control to a predefined exception vector address, which points to the part of the program responsible for handling the corresponding event. In this mechanism, the system control coprocessor is tightly integrated with the main processor, as its registers and control mechanisms directly participate in interrupt and exception handling. The following MIPS assembly code shows the segment of the program that contains the interrupt vector [3, 44, 60].

Listing 8. Exception and Interrupt Vectors MIPS Assembly

```
1:  ; The Exception and Interrupt Vectors
2:
3:  Exception_Vector :
4:      j Exception_Handler
5:      nop
6:
7:  Interrupt_Vector :
8:      j Interrupt_Handler
9:      nop
10:
```

The handler first stores the general-purpose registers onto the stack to preserve the current state of the main program, which is represented by the given MIPS assembly code [3].

Listing 9. Store Registers MIPS Assembly Code

```
1:  ; Part of the Handler which Stores Registers
2:
3:  Store_Registers :
4:      addiu $sp, $sp, -96
5:      sw $2,  0($sp)
6:      sw $3,  4($sp)
7:      sw $4,  8($sp)
8:
9:      ; Each Registers is Stored Sequentially
10:     sw $23,  84($sp)
11:     sw $24,  88($sp)
12:     sw $25,  92($sp)
13:
14:     ; Jump Back to the main Handler
15:     jr $ra
16:
```

Subsequently, the masked interrupt bits are retrieved from the Status register, and the interrupt event bits are obtained from the Cause register. Both values are bitwise ANDed, and the specific interrupt is identified by checking each bit in the result. The following MIPS assembly code illustrates a simplified interrupt handler [3].

Listing 10. Interrupt Handler MIPS Assembly Code

```
1:  ; This is Simple Interrupt Handler
2:  Interrupt_Handler :
3:      ; Jump and Link to the Store Part
4:      jal Store_Registers
5:      ; Get Status and Cause Registers
6:      mfc0 $26, $12, 0
7:      mfc0 $27, $13, 0
8:      ; Get Unmasked Interrupt Bits
9:      and $26, $26, $27
10:     srl $26, $26, 8
11:     andi $26, $26, 0x000f
12:     ; Store Return Address to End of the Handler
13:     la $ra, $Handler_End
14:     ; Check Bits Sequentially
15:     addi $t0, $0, 0
16:     beq $26, $t0, First_Interrupt
17:     addi $t0, $0, 1
18:     beq $26, $t0, Second_Interrupt
19:     addi $t0, $0, 2
20:     beq $26, $t0, Third_Interrupt
21:     addi $t0, $0, 3
22:     beq $26, $t0, Fourth_Interrupt
23:     addi $t0, $0, 4
24:     beq $26, $t0, Fifth_Interrupt
25:     addi $t0, $0, 5
26:     beq $26, $t0, Sixth_Interrupt
27:     addi $t0, $0, 6
28:     beq $26, $t0, Seventh_Interrupt
29:     addi $t0, $0, 7
30:     beq $26, $t0, Eighth_Interrupt
31:
32: Handler_End :
33:     ; Jump and Link to the Restore Part
34:     jal Restore_Registers
35:     ; Return to the Interrupted Address
36:     eret
```

After this, the core begins executing the corresponding ISR. Within the ISR, the interrupt condition is cleared, or an acknowledgment signal is sent to the peripheral that triggered the interrupt to deassert the corresponding request signal. Once the ISR has been executed, the program returns to the end of the handler, and the general-purpose registers are restored from memory, as shown in the MIPS assembly code below [3, 44, 60].

Listing 11. Restore Registers MIPS Assembly Code

```
; Part of the Handler which Restores Registers

Restore_Registers :
    lw $2,  0($sp)
    lw $3,  4($sp)
    lw $4,  8($sp)
    ; Each Registers is Restored Sequentially
    lw $23,  84($sp)
    lw $24,  88($sp)
    lw $25,  92($sp)
    ; Roll Back the Stack Pointer
    addiu $sp, $sp, 96
    ; Jump Back to the main Handler
    jr $ra
```

Finally, the system resumes execution from the point at which it was interrupted, thereby restoring the program state. This mechanism of the system control coprocessor is depicted in Fig.2.41 [3, 26].

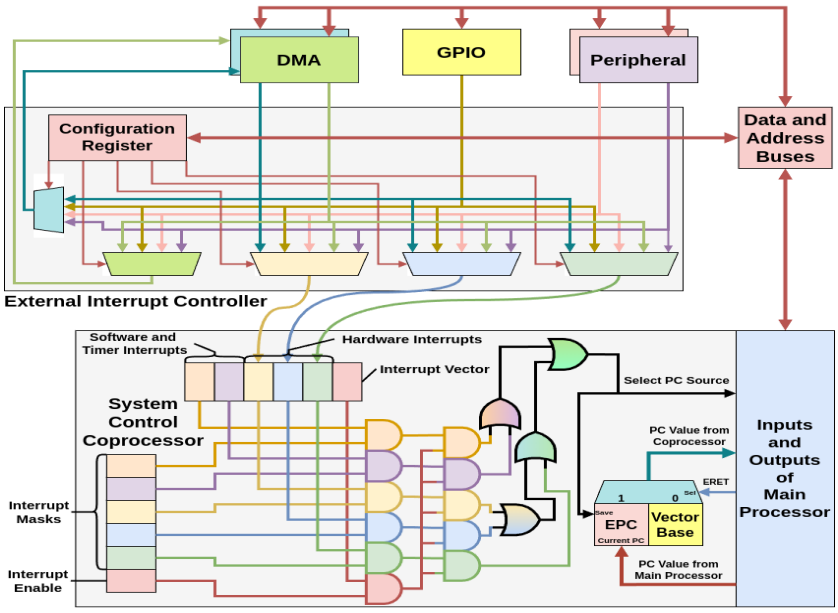


Figure 2.41 Diagram of the Interrupt Handling by Coprocessor

‘

2.7.2 Multi-Core Synchronization

IPIs are a specialized type of interrupt used for concurrency control and inter-core communication as part of IPC. IPIs play a crucial role in task coordination and data synchronization among multiple cores. There are several IPC models, each providing distinct approaches to multiprocessor communication and synchronization. Two specific IPC models are considered in this thesis, each providing a different mechanism for core-to-core communication in the OpenRISC and Xtensa architectures, respectively [13, 65, 70, 74].

In the OpenRISC model, data shared between cores is exchanged via shared memory, which is referred to as the Open Multi-Processor Interrupt Controller (OMPIC). The OMPIC functions as a memory-mapped PIC for IPC. In the OMPIC, as depicted in Fig.2.42, each core has its own pair of 32-bit registers that are mapped to specific memory addresses [73, 74].

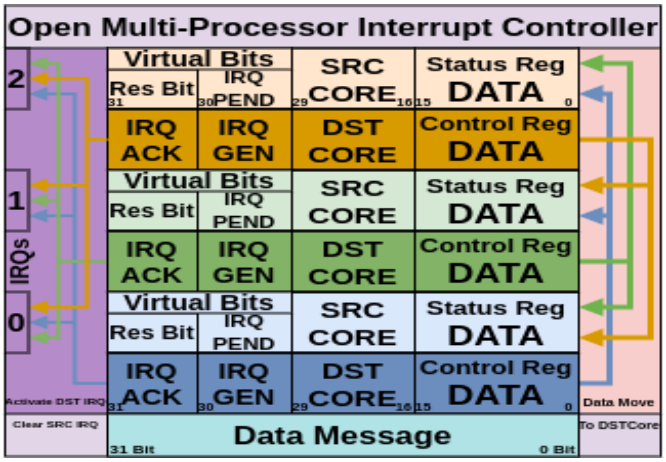


Figure 2.42 Diagram of the Open Multi-Processor Interrupt Controller

These registers facilitate the exchange of shared data between cores. To transmit data to a designated consumer core, the producer core writes a 32-bit message to its control register within the OMPIC. The message includes both the target core ID and the shared data. The producer core must set the *IRQ_GEN* bit in the message to enable the OMPIC to generate an interrupt, notifying the consumer core that new data is available. Upon receiving the interrupt, the consumer core must first clear the *IRQ_ACK* bit to acknowledge it. Once the interrupt is acknowledged, the consumer core may read the message from its status register, which contains both the source core ID and the shared data [73, 74].

A shared Wishbone bus functions as the communication backbone for data transfer among multiple cores within the OpenRISC IPC model, with each core acting as a bus master when accessing the OMPIC. Meanwhile, interrupts are employed to synchronize cores when shared data becomes available or to coordinate concurrent tasks across multiple cores. The entire communication model is illustrated in Fig.2.43 [73, 74].

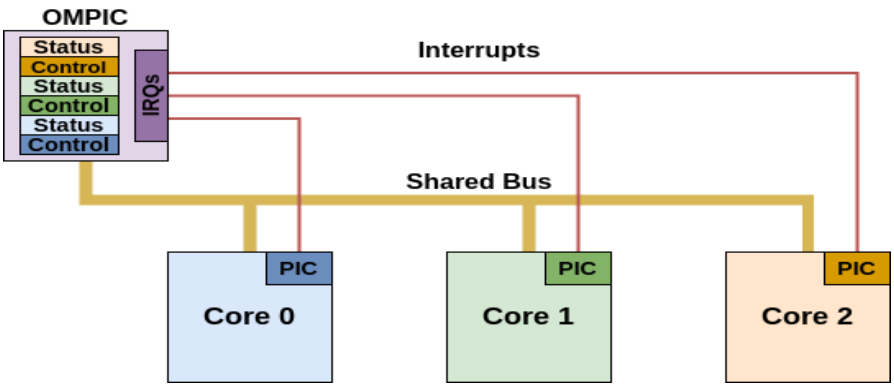


Figure 2.43 Diagram of the OpenRISC Inter-Processor Communication

The second model implements two interfaces: the first enables direct Port-to-Port communication for data sharing between two cores, and the second uses a FIFO buffer to transmit a data stream to another core. The communication model is shown in Fig.2.44 [13, 65, 70].

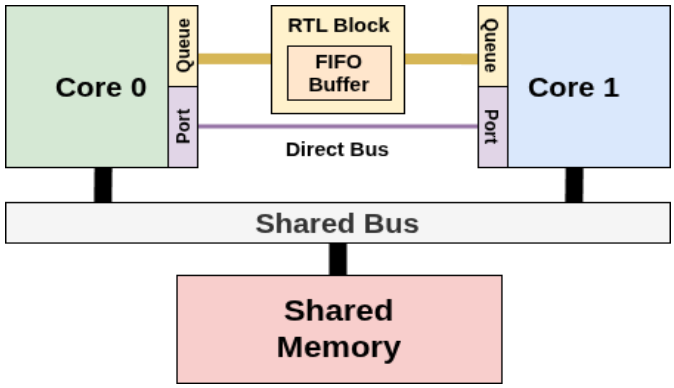


Figure 2.44 Diagram of the Xtensa Inter-Processor Communication

In Port-to-Port communication, each core has input and output ports, along with associated *Data-Ready* and *Acknowledge* bits, which are connected to the local bus shared by the local peripheral devices and the DMA controller. The input port of one core is directly connected to the output port of the other, creating a dedicated communication channel between the two cores. In this setup, data transfer is performed as a port-to-port operation, moving data directly from the producer core's output register to the consumer core's input register. This method enables faster IPC and improves synchronization between the cores by bypassing main memory.

Additionally, the *Data-Ready* and *Acknowledge* bits are directly connected between the ports of the communicating cores. These bits are also connected to each core's PIC, thereby enabling the system to trigger interrupts based on data transfer events. This mechanism is particularly advantageous for the concurrent control of multi-threaded programs within Xtensa dual-core systems, as task notifications serve as interrupt-based signals that inform a core or thread when a specific task or data is prepared for transfer. The task notification process begins when the producer core deposits data into its output port, which activates the *Data-Ready* bit and automatically generates an IPI for the consumer core. Upon receipt of the interrupt, the consumer core reads the shared data or address from its input port. Subsequently, after processing the received information, the consumer core toggles the *Acknowledge* bit, indicating to the producer core that the shared data or address has been successfully received. This action generates an interrupt for the producer core, notifying it that the task notification has been completed and thereby ensuring synchronization between the cores. The port-to-port communication method is illustrated in the subsequent Fig.2.45 [13, 65, 70].

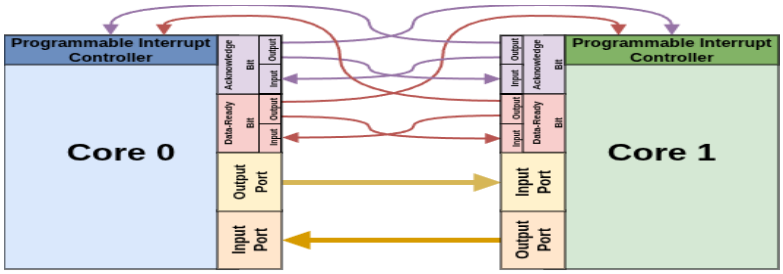


Figure 2.45 Diagram of the Xtensa Port-to-Port Communication

The Xtensa-based MPSoC can also use the second interface, which implements FIFO buffers to stream data between cores, as illustrated in Figure 2.46. The queue interface is unidirectional. When the FIFO buffer is full, only the producer's data transfer operation is stalled until space becomes available; the producer core itself continues execution. Similarly, when the FIFO buffer is empty, only the consumer's data transfer operation is stalled until data becomes available; the consumer core itself continues execution. However, the consumer core can internally assign an interrupt to the empty signal, which is triggered when the empty bit is cleared; similarly, the producer core can internally assign an interrupt to the full signal when the full bit is cleared. Additionally, this can be further optimized by assigning a DMA controller to transfer data streams between the queue interface and local memory [13, 65, 70].

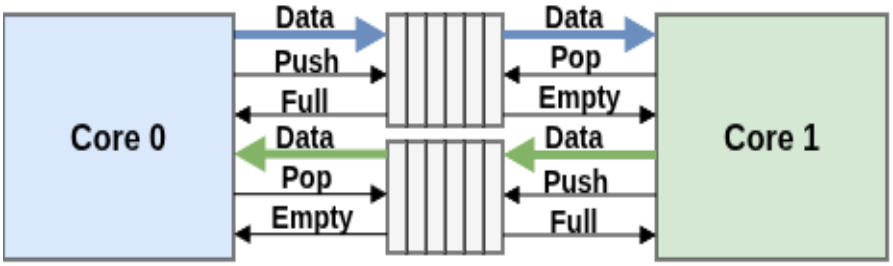


Figure 2.46 Diagram of the Xtensa IPC with the Queue Implementation

2.8 Evaluation Criteria

Standard memory-based synchronization is fundamental to modern computers because they process large amounts of data and typically do not operate under the strict timing constraints, as in safety-critical systems. Additionally, multi-threaded software in modern computing systems aims to reduce synchronization overhead by allocating resources locally, allowing cores to perform computations with less frequent synchronization.

The need for faster and more deterministic multi-core synchronization arises in embedded systems with stricter timing requirements. Two synchronization models are considered, one implemented in the OpenRISC architecture and the other in the Xtensa architecture. Both models use interrupt-driven mechanisms, while differing in the way data and synchronization information are exchanged between cores. In particular, the models differ in their use of shared interconnects and dedicated communication mechanisms for core-to-core and core-to-memory data transfers.

The proposed semaphore model is evaluated by measuring the number of clock cycles required to synchronize multiple cores and comparing the results with those obtained using the previously described synchronization models. The first evaluation examines semaphore synchronization using an AXI-based implementation on a Zynq-7000 dual-core system. The second evaluation compares the proposed hardware-based semaphore mechanism with software-based semaphore synchronization used in bounded-buffer communication, particularly single-slot bounded buffers, which provide the software-based approach most closely related to the proposed hardware-based semaphore. Such communication mechanisms are commonly used in safety-critical software to support synchronized and non-interfering data exchange between concurrent tasks.

Finally, core-to-core synchronization using interrupt-driven and non-interrupt-driven mechanisms, including configurations with and without a custom coprocessor containing an ALU capable of performing computations concurrently with synchronization. These evaluations allow the synchronization performance to be

assessed for different distributions of synchronization functionality between software and dedicated hardware.

2.9 Summary

Safety-critical applications are commonly implemented using heterogeneous, distributed embedded systems composed of networks of interconnected microprocessors. These systems typically employ industrial communication protocols such as CAN and, in the case of PLC-based systems, CIP-based protocols. The literature overview outlines their defining characteristics, particularly the strict real-time constraints that enforce deterministic firmware execution. The microprocessor's software architecture is based on a separation kernel that follows the MILS architecture. This architecture is characterized by strictly isolated modules with non-interfering data flows. Microprocessors in embedded systems can include coprocessors responsible for system control functions and floating-point calculations.

Furthermore, cache architecture plays a significant role in non-deterministic multi-core synchronization because the cache hierarchy divides memory into multiple fragments assigned to specific cache lines, leading to non-deterministic data-fetching behavior. To ensure the correct operation of a multi-core system, a cache coherency mechanism continuously updates cache line states based on the states of corresponding cache lines on other cores. Another essential mechanism in multi-core systems is memory consistency, which ensures the correct ordering of memory accesses. In multi-core systems, concurrency control and synchronization are typically based on software-based semaphores; however, to minimize the WCET, one effective approach is to use interrupt-driven synchronization for IPC.

Hardware Mechanisms for Fast Thread Synchronization

3.1 AXI Semaphores in Zynq

The hardware-based semaphore synchronization mechanism integrated into multicore CPU systems provides an efficient approach to concurrent control. Compared with conventional synchronization mechanisms for multicore architectures, the proposed mechanism significantly reduces synchronization overhead and improves the efficiency of inter-core communication.

As part of this research, several multicore synchronization mechanisms and their software-based implementations were analyzed. An initial proof-of-concept implementation was developed using the Xilinx Zynq architecture to verify the feasibility and correctness of the proposed approach. The results obtained from this implementation provided a foundation for the further development and evaluation of the synchronization mechanisms presented in this chapter [36]

Two key factors motivated the development of the proposed multicore synchronization mechanism. The first was the need to minimize the overhead associated with synchronization among multiple cores, while the second was the need to provide a reliable and deterministic mechanism for coordinating parallel operations [36].

The first hardware-based semaphore solution was implemented on the Xilinx Zynq-7000 platform. The connection between the CPU and the FPGA-based logic is established through an AXI crossbar, as shown in Fig.3.1. This platform was selected because it provides an effective evaluation framework for high-performance embedded systems by integrating both the Processing System (PS) and the Programmable Logic (PL) within a single device. This architecture enables efficient communication between software running on the processors and custom hardware modules implemented in the FPGA logic (PL) [69].

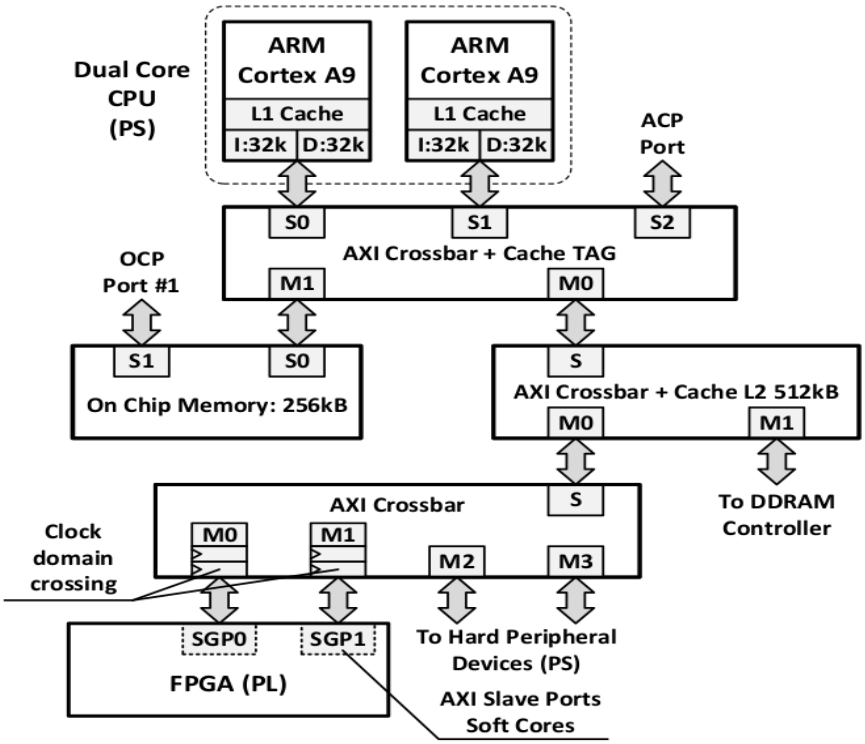


Figure 3.1 ZYNQ Platform Simplified Block Diagram of PS and PL Connection [36]

The PS provides AXI controller ports that connect to the hardware semaphore module implemented in the PL. The semaphore module exposes an AXI target interface and is assigned a dedicated address range within the processor's memory address space. The ARM cores access the semaphore through memory-mapped I/O, using read and write transactions to control and monitor its state. The address range assigned to the semaphore is configured as non-cacheable to ensure that each access is directly transferred to the hardware module in the PL rather than being served from the processor cache. This ensures that the processor always observes the current state of the semaphore [36].

Fig.3.2 shows the hardware-based semaphore implemented with a dedicated memory cell to store shared data and a SEM register to maintain the semaphore flag; additionally, modified AXI controllers were designed to extend the AXI interface transaction cycle based on the SEM register state. Access to the hardware-based semaphore is mapped to two distinct areas managed by the independent AXI ports GP0 and GP1 [36].

This mechanism enables controlled access to shared resources by conditionally delaying and completing AXI transactions based on the semaphore status. The signaling mechanism to the controller uses the RDY (Ready) and VAL (Valid) handshake signals of the AXI bus protocol. Assertion of the RDY signal indicates that the receiving component is prepared to accept data, whereas assertion of the VAL signal signifies that valid data is present on the bus.

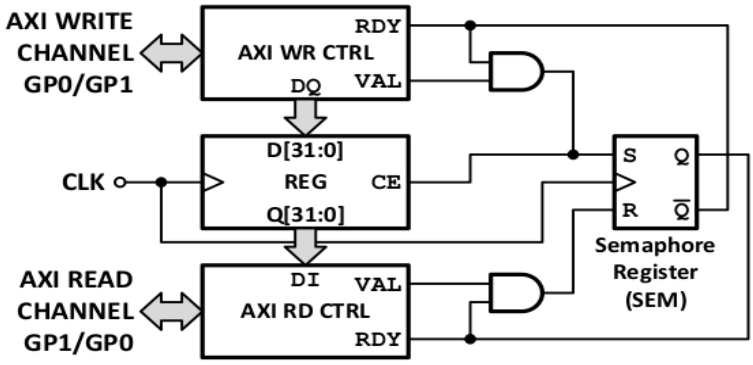
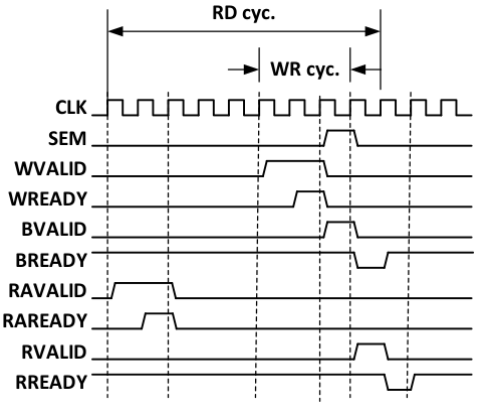
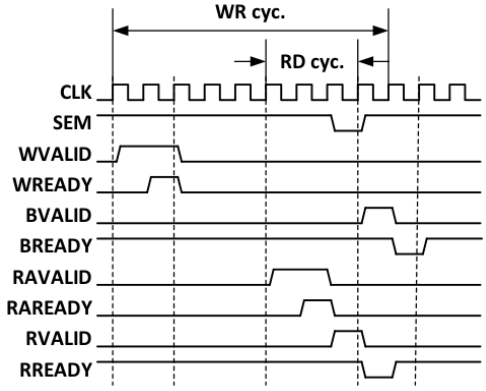


Figure 3.2 Semaphored Memory Cell Implementation [36]

The first figure illustrates the read cycle, which is extended by delaying the assertion of RVALID, which indicates that valid read data has been delivered from the slave to the master, until the data has been written to the memory cell by the other processor. The second figure depicts the write operation, which is delayed and not completed until the other processor reads the data.



(a) AXI Read Operation Waveform [36]



(b) AXI Write Operation Waveform [36]

The implemented semaphore system enabled the evaluation of the synchronization process at the hardware level. The implementation allows a memory read or write cycle to be extended until the semaphore is available and the requested operation can be completed. This approach differs from a purely software-based implementation, in which a shared memory location must be repeatedly monitored for changes, resulting in additional processor overhead. The proposed architecture can also be extended to support multiple hardware semaphores, providing a flexible synchronization mechanism and simplifying their integration into software applications. In this respect, the proposed system can be compared to a message pipe with synchronous blocking semantics, in which the transfer and synchronization of messages are handled directly by the dedicated hardware.

3.2 Thread Synchronization in Multicore Processing Systems

3.2.1 Multicore System and Synchronization Architecture

Fig.3.4 illustrates the proposed implementation of hardware-based semaphores in a quad-core system. The hardware module is directly connected to all MIPS cores and consists of an array of multiple hardware-based semaphores. In addition to this mechanism, the multi-core system also supports conventional cache-based synchronization [39].

Furthermore, each core in the system is equipped with two dedicated coprocessors. The first coprocessor manages system control functions, including the handling of interrupts from peripheral devices. The other is a custom coprocessor responsible for handling Inter-Processor Interrupts (IPIs) generated by hardware-based semaphores when data is written to them [38].

Additionally, the coprocessor design allows for the simultaneous execution of arithmetic operations while exchanging shared data between cores. The custom coprocessor offloads Interrupt Request (IRQ) handling from the hardware-based semaphores and performs the specified arithmetic operation concurrently during data loading from the shared semaphores [38].

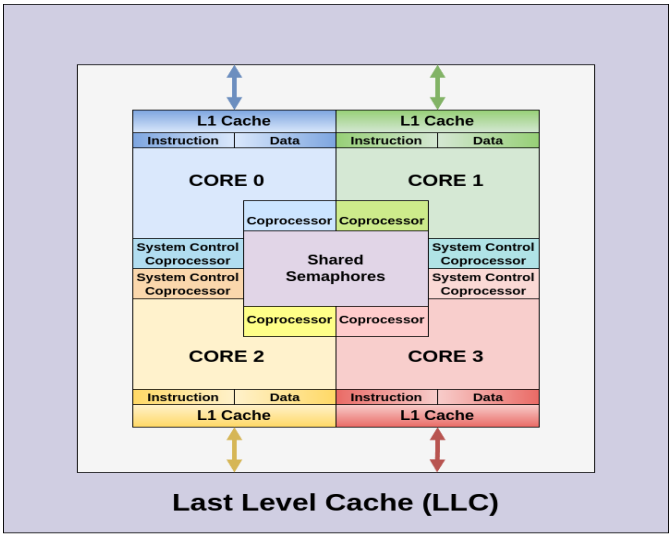


Figure 3.4 Diagram Of the Multi-Core System Overview [38]

Cores attempting to access shared data are stalled until the data is written to a designated hardware-based semaphore. This mechanism facilitates both data exchange and synchronization among multiple cores. Cores can also load shared data from the custom coprocessor, which operates in coordination with the main processor and its instruction pipeline. The coprocessor performs interrupt handling and supports the same arithmetic operations as the ALU of the main processor. These arithmetic operations can be performed concurrently with the retrieval of shared data from

the hardware-based semaphore through the coprocessor, thereby offloading these operations from the main processor's execution pipeline and enabling computation and data exchange to proceed in parallel [38].

The loading phase can be performed during the MEM stage of the processor pipeline without the custom coprocessor, or during the ID stage with the custom coprocessor; however, the storing phase can be performed only during the MEM stage of the pipeline [38].

Hardware-based semaphores enable synchronization of processes at the same moment they exchange data; meanwhile, software-based semaphores must be acquired and released using atomic instructions to synchronize processes before they exchange data across multiple cores. This phenomenon is caused by the overall cache design, which encompasses data coherency and consistency. The multi-core cache implementation allows multiple cores to simultaneously fetch and store data in the same memory block; however, when multiple cores attempt to update the same memory location concurrently, a race condition may occur. Since the ordering of these updates is not guaranteed, the final value stored in the memory location may correspond to the update performed by any of the competing cores. This is the reason for implementing atomic instructions to ensure atomic transactions between cache controllers in multiprocessors [39].

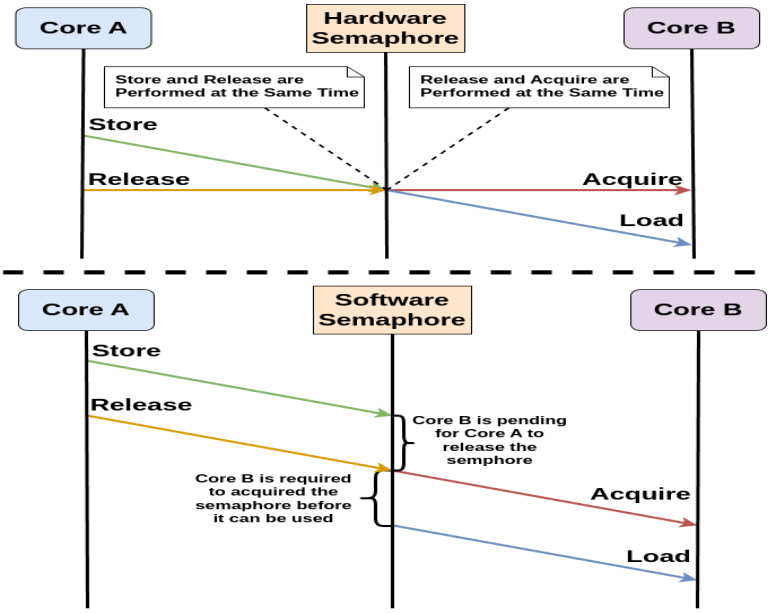


Figure 3.5 Comparison of Hardware and Software Semaphores [39]

Fig.3.5 illustrates the delay caused by the need to acquire and release software-based semaphores. This delay is unacceptable in safety-critical applications because the accumulated latency and non-deterministic behavior of cache-based mechanisms in multi-core systems significantly slow down synchronization among multiple threads. This is the main reason multiple implementations of Inter-Processor

Communication (IPC) are found in multi-core systems such as the Xtensa and OpenRISC architectures [38, 39].

The multi-core architectures mentioned are most often interrupt-driven synchronization models; however, the proposed hardware-based semaphore synchronization model was explicitly designed to evaluate a stalling-driven method for synchronizing multiple sections of a PLC program that run in parallel. Example 3.6: A single task constantly monitors whether both sensors are triggered to alert the pedestrian not to cross the road [38].

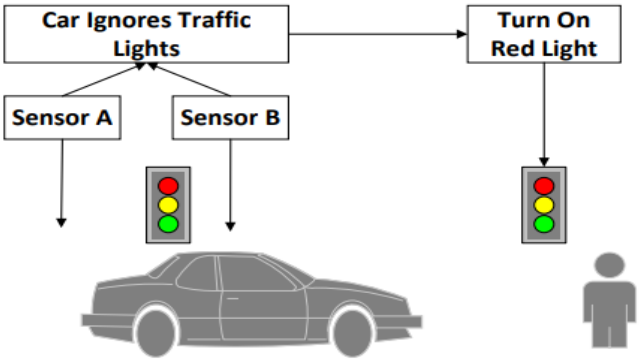


Figure 3.6 Traffic Light Example [39]

Fig.3.7 is an intermediate representation of the PLC program, showing the control flow with red arrows and data dependencies with blue and green arrows.

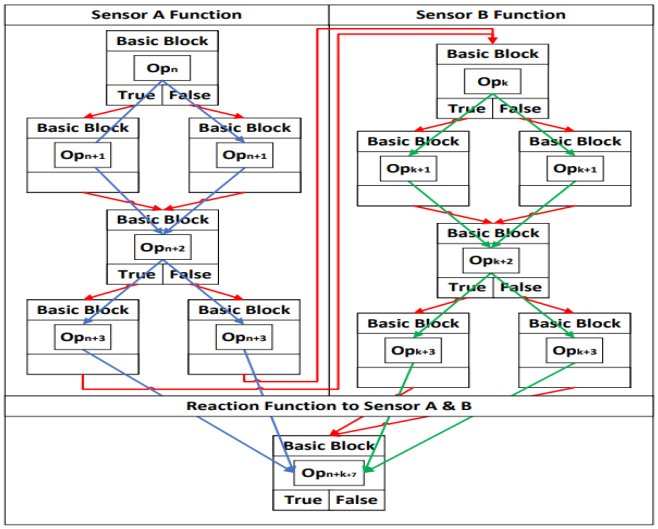


Figure 3.7 Data Flow and Control Flow Graphs of the Example [39]

It can be observed that data dependency permits the functions responsible for verifying sensors A and B to be executed in any arbitrary order; consequently, they can be performed concurrently. Safety-critical application requires rapid and frequent data exchange across multiple threads, which is not amenable to vectorization. As

previously discussed, memory latencies in such scenarios may result in a substantial decline in performance.

3.2.2 Hardware Semaphore Design

Figure 3.8 represents the hardware-based semaphores arranged in an array. Hardware-based semaphores are indexed like a conventional array because they are associated with specific index values; additionally, each semaphore contains a counter and a shared data segment .

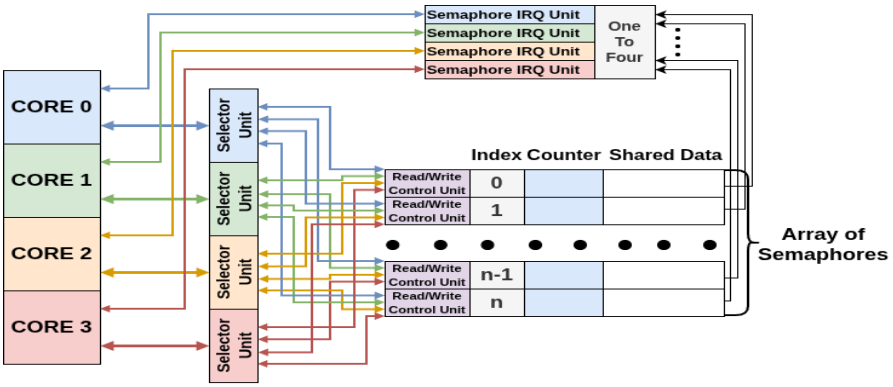


Figure 3.8 The Diagram of the Shared Semaphore Array [38]

In microprocessors, data memory management is handled by a local memory controller at the memory-access stage of the pipeline. This configuration enables the use of specialized memory-access instructions and allows the controller to be easily connected to the hazard unit, which governs pipeline stalling when loaded data is used in the subsequent clock cycle. Similarly, the proposed synchronization model can integrate a dedicated local semaphore controller to employ specialized semaphore instructions, thereby expediting load and store operations on hardware-based semaphores. Furthermore, both methodologies that integrate hardware-based semaphores adopt pipeline-stalling techniques rather than relying on spin-locking mechanisms, which inefficiently consume CPU cycles through constant checks to determine whether shared data has been placed in a shared semaphore [39]

The local semaphore controller uses *SemaRead* and *SemaWrite* control signals, analogous to the Read and Write signals of the local memory controller, which are activated by decoded memory-access instructions. The pipeline of a core may stall while awaiting shared data. This is achieved through the *Stall* control signal, which links the local semaphore controller to the hazard unit; consequently, it prevents unreliable concurrency control. The overall design of the controller is illustrated in Fig.3.9.

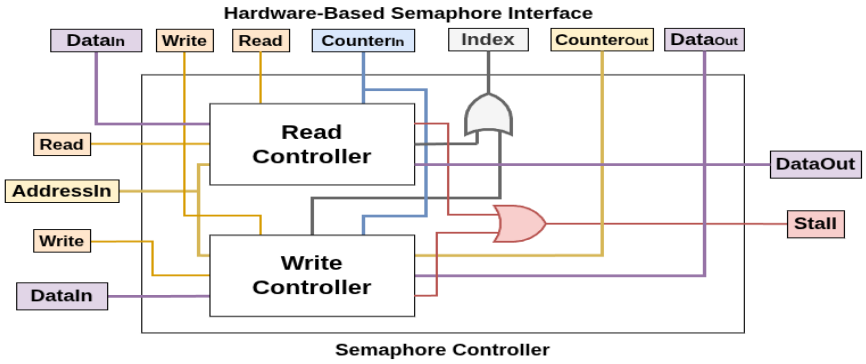


Figure 3.9 Diagram of Semaphore Controller [39]

As shown in the figure, the local semaphore controller is further divided into read and write components. Within both components, the lower 8 bits of the address are dedicated to the semaphore's index; meanwhile, the upper 8 bits are used for the semaphore counter's value, which are used only by the writing part of the controller. The following two figures present the read and write components of the local semaphore controller, respectively.

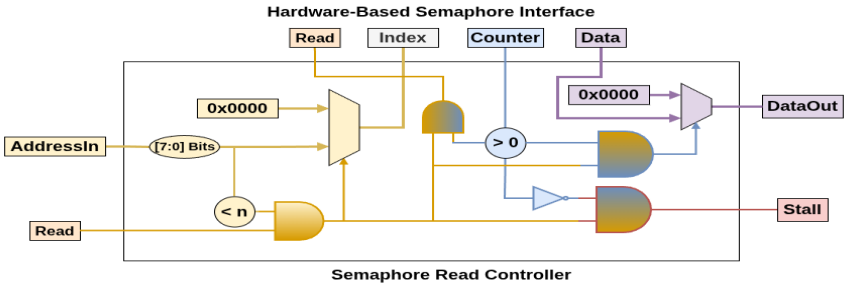


Figure 3.10 Diagram of Semaphore Read Controller [39]

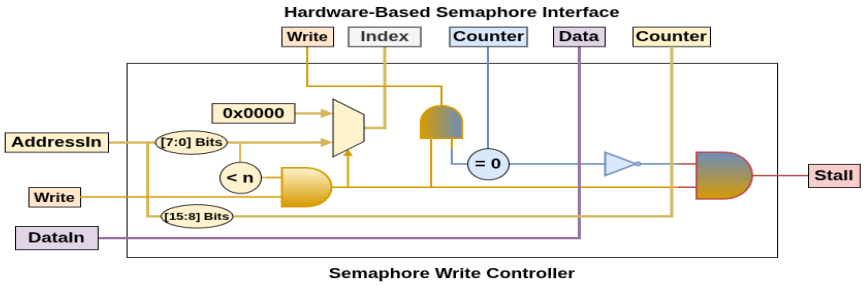


Figure 3.11 Diagram of Semaphore Write Controller [39]

The local semaphore controller was designed as a lightweight hardware component, allowing each core to be equipped with a dedicated controller while reducing the overhead associated with loading from and storing to hardware-based semaphores. The procedure for loading shared data from a semaphore can be accomplished using a single dedicated semaphore instruction, as shown in the MIPS assembly code below.

Listing 12. Semaphore Load Operation

```
1: ; Load Shared Data from
2: ; the Semaphore Index of One
3:
4: semaLoad $t0, 0x0000($a1)
5:
```

Furthermore, the store operation on a shared semaphore is implemented using three instructions, as shown in the following MIPS assembly code.

Listing 13. Semaphore Store Operation

```
1: ; Store Shared Data of Two in
2: ; Pre-Calculated Value of 0x0301 that Combines
3: ; the Counter's Value of Three and Index of One
4:
5: addiu $a1, $a0, 0x0301
6: addiu $t0, $zero, 0x0002
7: semaStore $t0, 0x0000($a1)
8:
```

However, this can be further optimized to two instructions by placing the pre-calculated value in the offset field of the store instruction, as shown previously.

3.2.3 Mapping the Hardware Semaphores

Figure 3.12 depicts a three-stage pipeline that includes the necessary modifications to support the proposed implementation. The pipeline has two interfaces to the shared semaphores. The first interface serves as a load interface at the decode stage and operates through the custom coprocessor. At the memory access stage of the pipeline, either loading from or storing to the hardware-based semaphore can be performed [38].

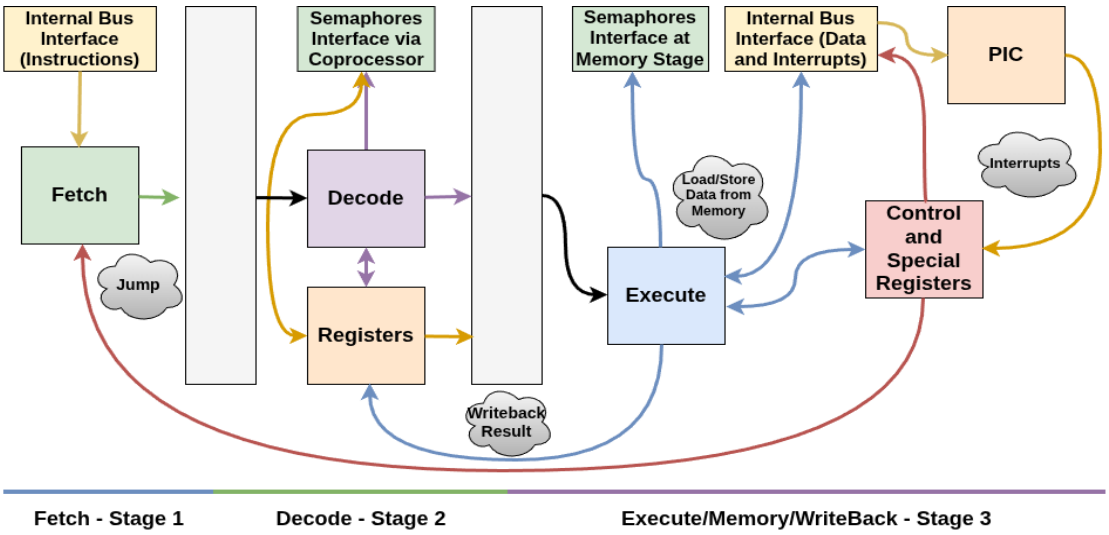


Figure 3.12 The Diagram of a Pipeline with the Shared Semaphore Interface [38]

The system architecture defines a memory map in which physical address ranges are assigned to different resources, including main memory, GPIOs, and peripheral devices. Hardware-based semaphores are similarly assigned to a dedicated region within this address space, allowing them to be accessed by the CPU using standard load and store instructions. To access a shared semaphore, the program first loads the upper 16 bits of its physical address, which identify the memory segment containing the semaphore, as shown in the MIPS assembly code provided below [39].

Listing 14. Load a Specific Memory Segment

```
1: ; The Four Most Significant Bits
2: ; Indicate the Memory Segment
3:
4: addiu $a0, $zero, 0x6000
5: sll $a0, $a0, 16
6:
```

Subsequently, the lower 16 bits of the memory address referencing a specific hardware semaphore are combined with the upper 16 bits value. Thus, loading data from a shared semaphore can be performed using a series of four instructions, as illustrated in the following MIPS assembly code.

Listing 15. Load Operation

```
1: ; Load Shared Data from
2: ; the Semaphore Index of One
3:
4: addiu $a0, $zero, 0x6000
5: sll $a0, $a0, 16
6: addiu $a1, $a0, 0x0001
7: lw $t0, 0x0000($a1)
```

The loading process can be accomplished with only three instructions by positioning the semaphore's index within the offset of the load instruction, as demonstrated in the second MIPS assembly code. When a specific region of physical memory is loaded into a general-purpose register, loading and storing can be performed without the first two instructions.

Listing 16. Optimized Load Operation

```
1: ; Optimized Loading Operation
2: ; from the Semaphore Index of One
3:
4: addiu $a0, $zero, 0x6000
5: sll $a0, $a0, 16
6: lw $t0, 0x0001($a1)
```

When storing shared data using hardware-based semaphores, it is also necessary to place both the semaphore's counter and index values in the lower 16 bits of the memory address. The semaphore's counter is represented in the upper 8 bits, while the semaphore's index is stored in the lower 8 bits. The storage operation is shown in the MIPS assembly code below.

Listing 17. Store Operation

```
1: ; Store Shared Data of Two in
2: ; the Semaphore Index of One with
3: ; the Counter's Value of Three
4:
5: addiu $a0, $zero, 0x6000
6: sll $a0, $a0, 16
7: addiu $a1, $zero, 0x0003
8: sll $a1, $a1, 8
9: addiu $a1, $a1, $a0
10: addiu $t0, $zero, 0x0002
11: sw $t0, 0x0001($a1)
```

This process can be enhanced by pre-calculating the semaphore's counter and index as a single 16-bit value. The optimized store operation is represented by the following MIPS assembly code.

Listing 18. Store Operation with Pre-Calculated Address

```
1:  ; Store Shared Data of Two in
2:  ; Pre-Calculated Value of 0x0301 that Combines
3:  ; the Counter's Value of Three and Index of One
4:
5:  addiu $a0, $zero, 0x6000
6:  sll $a0, $a0, 16
7:  addiu $a1, $a0, 0x0301
8:  addiu $t0, $zero, 0x0002
9:  sw $t0, 0x0000($a1)
10:
```

This can be further optimized by encoding the pre-calculated value in the offset field of the store instruction, as shown in the second MIPS assembly code. As a result, a single instruction adds the pre-calculated counter and index value to the base memory address. This optimization also eliminates the need to reserve an additional general-purpose register to hold the pre-calculated lower 16-bit value.

Listing 19. Optimized Store Operation

```
1:  ; Optimized Store Shared Data of Two in
2:  ; Pre-Calculated Value of 0x0301 that Combines
3:  ; the Counter's Value of Three and Index of One
4:
5:  addiu $a0, $zero, 0x6000
6:  sll $a0, $a0, 16
7:  addiu $t0, $zero, 0x0002
8:  sw $t0, 0x0301($a1)
9:
```

To further improve access to the hardware-based semaphores, two dedicated instructions were introduced to enable more efficient load and store operations. These instructions are decoded by the instruction decoder module in the same manner as the other fetched 32-bit instructions. The proposed instructions use the same instruction formats as the standard load and store instructions, while requiring dedicated operation codes to distinguish them during decoding.

3.3 Coprocessor Design

In the proposed multi-core synchronization model, hardware-based semaphores can also be configured such that the eight-bit field associated with each shared semaphore is used as a set of eight core-specific flags, rather than as a counter. Each bit is permanently associated with one of the eight individual cores and serves as an event marker for that core. When the bit corresponding to a given core is set, an IPI is generated for that core. Upon acknowledging the corresponding IRQ, the bit is cleared, allowing the waiting thread on the target core to resume and enabling coordinated data exchange between cores. This synchronization mechanism assumes that the corresponding threads are already waiting for the event on their assigned cores. If a target core is executing another thread when the IPI is generated, the interrupt may preempt the currently executing thread [38].

The hardware-based semaphore module contains multiple submodules, each associated with a specific core and responsible for monitoring the corresponding bit in the marker field of each shared semaphore. These submodules generate an IRQ request when the marker associated with their core is set. Each submodule also contains a register in which individual bits are mapped to the corresponding shared semaphores; this register is used by the interrupt-handling mechanism to identify the semaphore that triggered the interrupt, as illustrated in Fig.3.13. An important functional case occurs when a core sets the marker associated with itself. In this situation, the interrupt generated for the corresponding core is automatically acknowledged, since the event originates from the same core that would otherwise receive the notification [38].

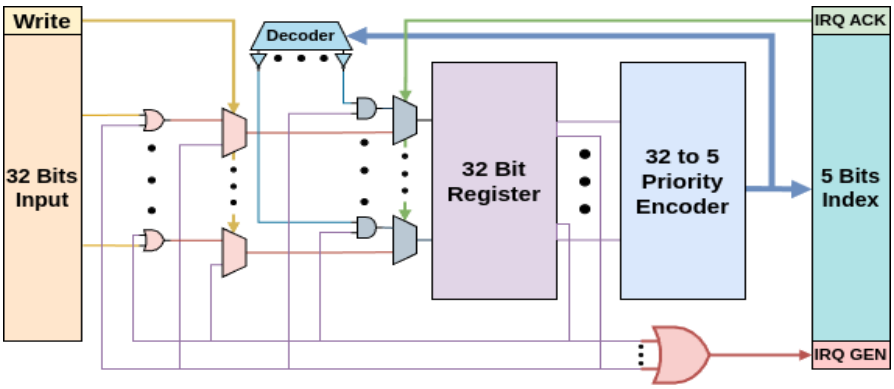


Figure 3.13 The Diagram of the Submodule Handling IRQs [38]

When data is shared and written to a semaphore, the corresponding bit within the register is activated. This action generates an IRQ request to the core associated with the relevant submodule and provides the index of the semaphore affected by the operation. The index is determined by a priority encoder, which selects the lowest-indexed active semaphore. This mechanism follows a task-dispatch model in which a main task initiates tasks assigned to specific processor cores by setting

the corresponding semaphore markers. The semaphore index identifies the target operation, allowing the interrupt handler to invoke the corresponding interrupt service routine (ISR) through a predefined mapping between semaphore indices and task functions. The submodule processes pending requests sequentially, waiting for an IRQ acknowledgment before issuing the next request [38].

Furthermore, the system employs four mechanisms to ensure that incoming IRQ requests are handled in an orderly and deterministic manner. First, a core sets the marker bit corresponding to the target core in the shared semaphore. Second, the generated IRQ request is acknowledged by the custom coprocessor of the target core. The acknowledgment is generated either when the corresponding interrupt is masked or upon execution of the Interrupt Return Instruction (ERET) at the end of the ISR. Thus, when the interrupt is masked, the IRQ request is automatically acknowledged and does not initiate ISR execution. When the interrupt is unmasked, the request is delivered to the target core and handled by the corresponding ISR [38].

Conceptually, the presented multi-core synchronization model aligns with the Open Multi-Processor Interrupt Controller (OMPIC) in the OpenRISC architecture, which uses a shared module within the IPC framework to facilitate data exchange and generate IPIs. However, the proposed model employs direct connections between each core and an array of hardware-based semaphores, analogous to the approach used in multi-core Xtensa architectures, where dedicated queues are utilized [38].

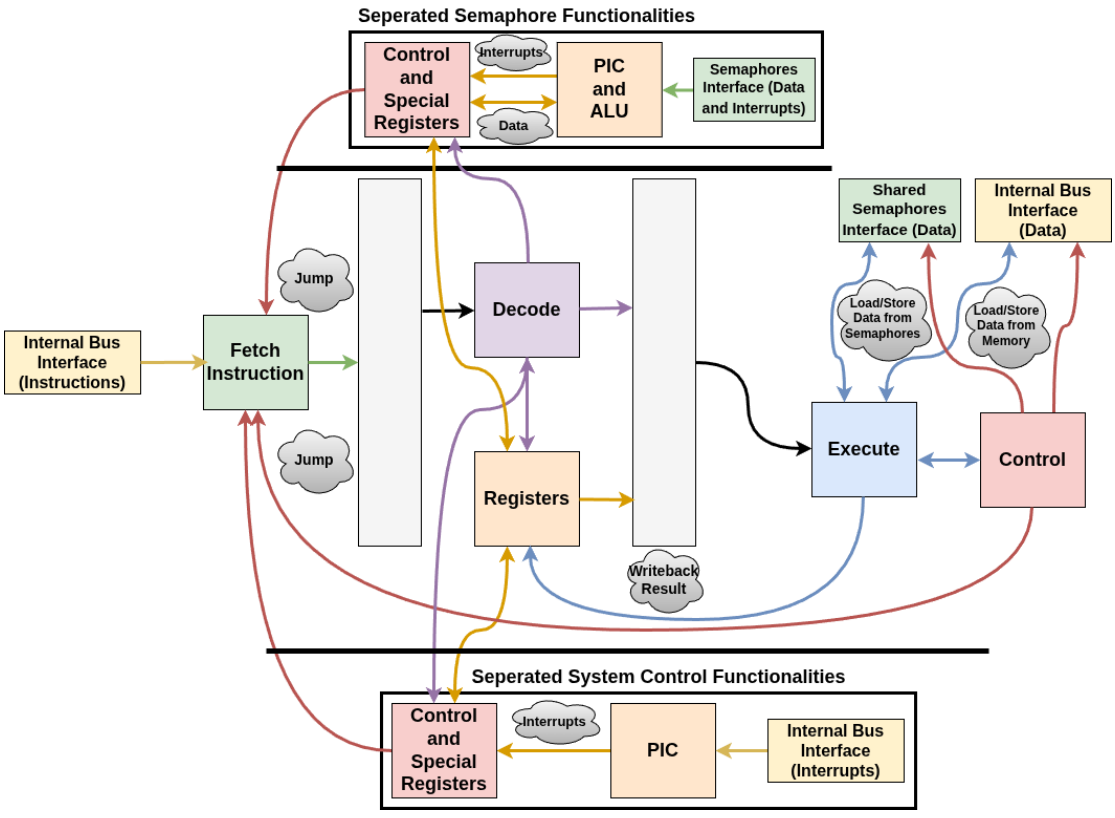


Figure 3.14 The Diagram of a Pipeline with the Coprocessors [38]

Fig.3.14 represents the proposed MIPS core design, which features two coprocessors. The system control and semaphore functionalities, responsible for handling interrupts, are implemented separately from the main processor. The coprocessors include control and additional special registers that play an important role in interacting with the Programmable Interrupt Controller (PIC), which handles incoming IRQ requests [38].

Nonetheless, the custom coprocessor also integrates an arithmetic logic unit (ALU) that executes arithmetic operations on data retrieved from hardware-based semaphores. The coprocessor operates the same way as a floating-point coprocessor, with the additional feature of IRQ handling. However, the computation is performed instantaneously and concurrently with data retrieval, which occurs during the pipeline's decoding stage via a custom instruction resembling a MIPS floating-point coprocessor instruction. The custom instruction is illustrated in Fig.3.15.

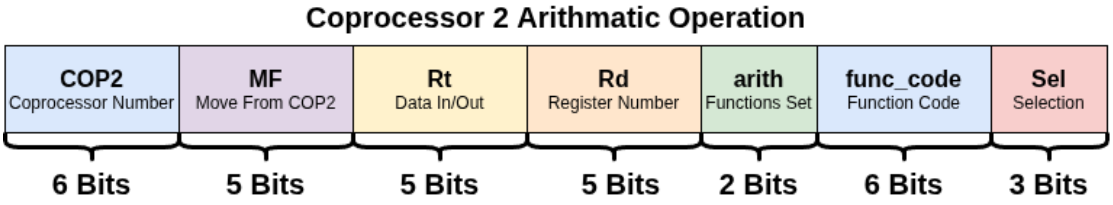


Figure 3.15 Diagram Of the Custom Instruction Encoding Format [38]

The *func_code* field in the custom instruction specifies the arithmetic operation in the same way as in the Register-to-Register instruction format. The semaphore identifier, denoted as *sem_id*, is encoded using the combination of the *Sel* and *Rd* fields. The three least significant bits of *sem_id* are provided by the *Sel* field, while the remaining five bits are provided by the *Rd* field, allowing access to shared semaphores indexed from 0 to 255. The *Rt* field serves as the first source operand, from which the value is retrieved from a shared semaphore through the custom coprocessor. It also serves as the second source operand, holding the value used in the arithmetic computation performed by the custom coprocessor [38].

The dedicated semaphore-processing ALU is integrated into the custom coprocessor to accelerate computation by performing arithmetic and logical operations directly on data retrieved from the hardware-based semaphores and on the second operand provided by the *Rt* field. Unlike the ALU of the main processor and other computational coprocessors, the proposed custom coprocessor does not include a dedicated register unit for storing specialized general-purpose registers. This is because the operands are obtained directly from the hardware-based semaphores and the instruction fields, allowing the required operations to be performed without additional intermediate register storage [38].

Shared data retrieval and computation occur when the custom instruction is executed, regardless of whether an IRQ has been generated. This allows a core to directly fetch shared data from a hardware-based semaphore using either interrupt-driven or stalling-based multi-core synchronization methods. The arithmetic component of the custom coprocessor is shown in Fig.3.16 below.

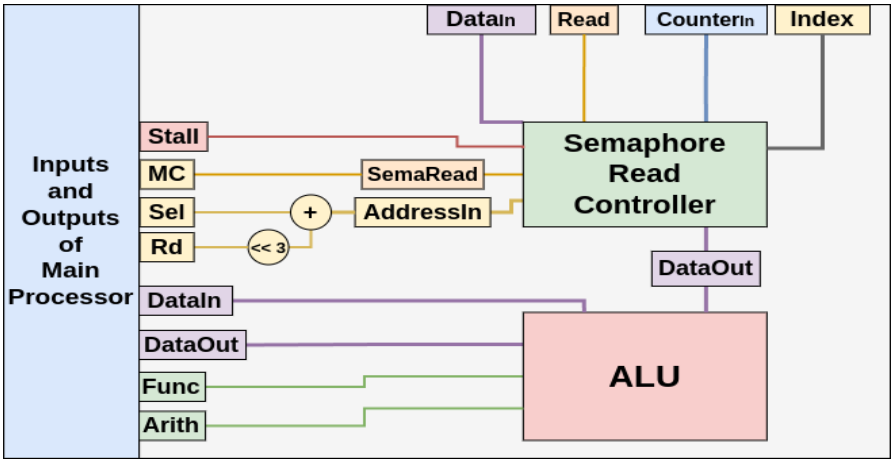


Figure 3.16 Diagram Of Arithmetic Part of the Coprocessor [38]

The following MIPS assembly code illustrates a scenario in which a core stores the value 2 in the hardware-based semaphore at index 1 and sets the counter to 3. Later, the core adds the stored value three times using the custom instructions: first to the initial value of 1, then to the accumulated sum in each subsequent step.

Listing 20. Semaphore Coprocessor Operation

```
1:  ; Store Shared Data of Two in
2:  ; Pre-Calculated Value of 0x0301 that Combines
3:  ; the Counter's Value of Three and Index of One
4:
5:  addiu $a1, $a0, 0x0301
6:  addiu $t0, $zero, 0x0002
7:  addiu $t1, $zero, 0x0001
8:  semaStore $t0, 0x0000($a1)
9:  mfc2.and $t1, $zero, 0x01
10: mfc2.and $t1, $zero, 0x01
11: mfc2.and $t1, $zero, 0x01
12:
```

The second integrated component of the proposed custom coprocessor is the IPI handling using hardware-based semaphores. This component is specialized exclusively for interrupt-driven synchronization in the multi-core system because it has dedicated vector base addresses, as shown in Fig.3.17.

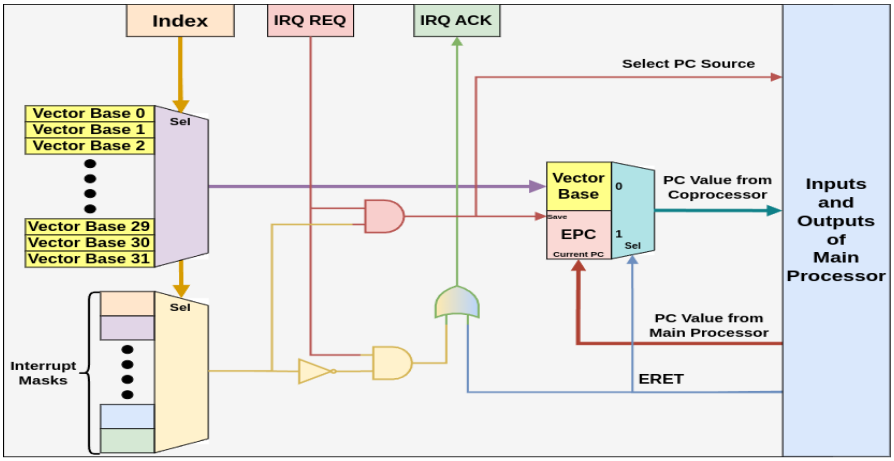


Figure 3.17 Diagram Of IRQ Handling Part of the Coprocessor [38]

They cause the program counter to jump immediately to the base address of the corresponding semaphore's interrupt vector, which points to the address where its specific ISR is located. Thus, this approach results in more efficient interrupt handling because there is no centralized interrupt handler that relies on the Cause register to identify the triggered interrupt. Additionally, as mentioned previously, interrupt acknowledgment is performed automatically upon returning from the ISR, which eliminates the need for explicit acknowledgment within the ISR [38].

As mentioned previously, the hardware-based semaphores are arranged in an array, allowing direct access to any shared semaphore using its index. This enables a program to assign specific semaphores to specific data exchanges between multiple cores, with lower-indexed semaphores receiving higher priority when multiple requests are pending. In the IRQ configuration, the hardware determines the semaphore to be serviced and generates the corresponding interrupt, while the software handles the resulting synchronization event. In contrast, in the non-IRQ configuration, the custom multi-threaded compiler manages the semaphore accesses according to the program execution flow to organize data exchanges between multiple cores [38].

3.4 Synchronization in Multicore Processing Systems Evaluation Method

The evaluation framework for the proposed quad-core system was developed and simulated using a Verilog simulator. The quad-core system uses classic five-stage MIPS cores. For each core, hardware-based semaphores are allocated to the upper region of the address space. A local semaphore controller is integrated into the pipeline alongside the data memory controller, as illustrated in Fig.3.18.

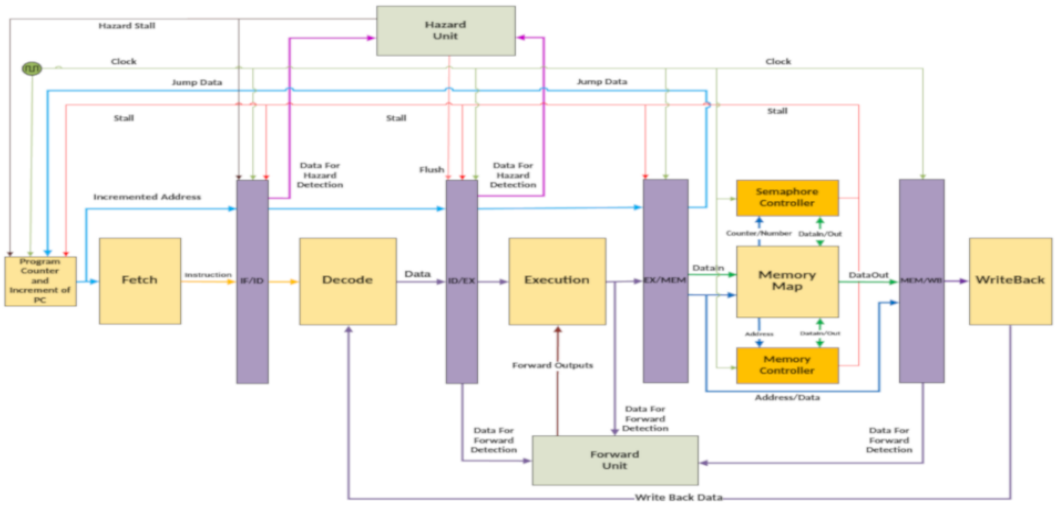


Figure 3.18 Diagram of the Core [39]

Safety-critical applications must comply with strict reliability, safety, and security requirements; this includes Real-Time Operating Systems (RTOS) specifically engineered for safety-critical firmware. The fundamental principles governing safety-critical applications include guaranteeing non-interference in data flow among processes and implementing rigorous component isolation. These safety mechanisms are essential to prevent one process from adversely affecting others within the system [39].

Consequently, The RTOS, which is specifically engineered for safety-critical firmware, adopts a Multiple Independent Levels of Security/Safety (MILS) architecture, as shown in Fig.2.2 at the beginning of this thesis. The MILS architecture partitions the firmware into strictly isolated modules and ensures non-interference in data flow through controlled communication mechanisms such as Application Programming Interfaces (APIs); the APIs employ single-slot producer-consumer buffers or bounded buffers to guarantee non-interference of the data flow between processes. These buffers are constructed using software-based semaphores, which enable controlled data exchange [39].

Therefore, the test cases designed to evaluate hardware-based semaphores against software-based semaphores must include real-world scenarios involving APIs and cover the use of single-slot producer-consumer buffers for data flow. The real-world data flow patterns are: from multiple producers to a single consumer, from a single producer to multiple consumers, and sequential data exchanges across multiple partitions [39].

Modern multi-core systems use either bus-based or NoC-based architectures; therefore, it is essential to evaluate the effectiveness of the proposed hardware-based semaphore synchronization model against existing bus-based and NoC-based synchronization models. This comparison must consider not only synchronization

performance but also emphasize correct usage and suitability for multi-core systems [39].

It is important to note that multiple interconnect models exist for multi-core systems due to the variety of cache coherency and memory consistency models, each exhibiting distinct characteristics and performance. Consequently, conducting a comprehensive evaluation of these models presents challenges, particularly because the most advanced multi-core synchronization models are proprietary and closed-source.

Due to this challenge, An ideal functional model of a multi-core interconnect was implemented, as shown in Fig.3.19. This model enables multiple cores to access and update shared data instantaneously and without collisions. It reduces traditional multi-core synchronization to a single clock cycle, which is theoretically possible but not practical, as even the most advanced synchronization mechanisms require multiple clock cycles. This idealized model ensures that the experimental evaluation remains unbiased and that the Worst Case Execution Time (WCET) is minimized to the theoretically possible limit [39].

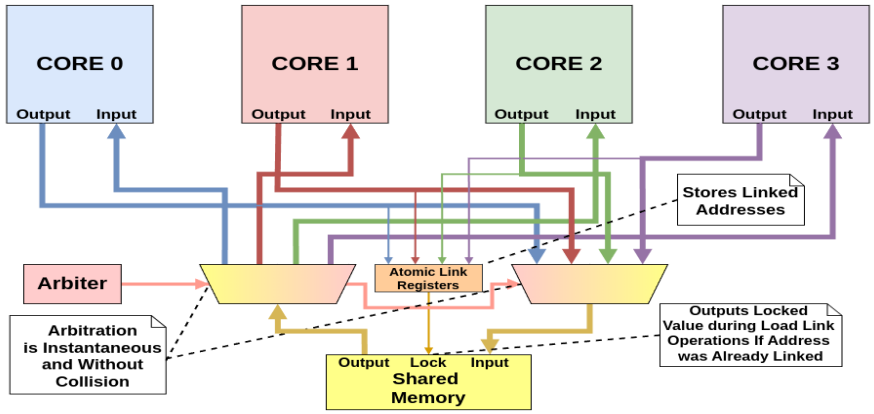


Figure 3.19 Diagram of the Quad Core system setup for performance assessment [39]

Additionally, the final phase of the evaluation investigates multi-core synchronization using the custom coprocessor. This coprocessor is specifically designed for the hardware-based semaphores accelerate data exchange by enabling concurrent arithmetic processes during the retrieval of shared data. Consequently, parallel computations performed by the custom coprocessor overlap with the synchronization mechanism of hardware-based semaphores, thereby diminishing the number of arithmetic instructions required from the primary processor to operate on shared data. This process effectively offloads computation overhead from the main processor. Moreover, the specialized coprocessor is capable of acquiring shared data through either a stalling-driven approach or an interrupt-driven mechanism. In the latter scenario, the secondary component of the coprocessor manages IRQs initiated by the hardware-based semaphores [38].

However, this architectural enhancement entails a higher degree of hardware complexity. The custom coprocessor incorporates a dedicated ALU whose hardware size is comparable to that of the ALU in the main processor. Consequently, it is imperative to assess whether the supplementary resource utilization, particularly the increased logic-gate consumption and silicon area attributed to the ALU within the custom coprocessor, is justified by its performance improvements [38].

Furthermore, the second objective of the final evaluation phase is to ascertain whether the interrupt-driven synchronization mechanism exceeds the performance of the proposed stalling-driven approach. In the stalling-driven approach, a bespoke multi-threaded compiler facilitates multi-core synchronization through hardware-based semaphores. This objective seeks to compare both methodologies and evaluate their efficacy based on their correct application [38].

Thus, to effectively assess the advantages of employing the custom coprocessor within the hardware-based semaphore synchronization model, it is imperative to formulate test cases that exemplify segments of multi-threaded programs frequently encountered in safety-critical applications. Additionally, it is necessary to compare four distinct synchronization configurations across these test cases. The first configuration excludes the custom coprocessor's ALU and does not include IRQ support. The second configuration incorporates the coprocessor's ALU but omits IRQ support. The third configuration includes both the custom coprocessor's ALU and IRQ support. The final configuration comprises IRQ support but lacks the coprocessor's ALU [38].

It is important to note that testing scenarios that use IRQs for concurrency control run in a loop within the main execution flow while also responding to triggered semaphore ISRs. Within these ISRs, shared data are loaded from and stored into the shared semaphores. In contrast, multi-threaded programs without IRQ support rely on a sequential but well-coordinated synchronization across multiple cores. Additionally, in all test scenarios, a single hardware-based semaphore is used for each data exchange operation [38].

3.5 Summary

The Xilinx Zynq-7000 platform was initially selected to evaluate the proposed hardware-based semaphore mechanism in a multi-core environment. The hardware semaphores were implemented and connected to the GP0 and GP1 controller ports of the AXI interconnect. These ports provide access to a non-cacheable address region, making them suitable for implementing shared synchronization resources. The communication path between the ARM processor and the GP0 port consists of three AXI interconnect layers. In addition, clock-domain-crossing synchronizers are employed, introducing further latency into semaphore accesses. Although

the implementation demonstrates the feasibility of hardware-based semaphore synchronization, the latency introduced by the fixed AXI interconnect architecture limits its suitability for evaluating low-latency synchronization mechanisms. Therefore, a custom processing system was developed to provide direct access to the hardware-based semaphores and enable a more controlled evaluation of their performance.

The further development of hardware-based semaphore synchronization involved constructing a quad-core system in which each core is directly connected to a shared array of hardware-based semaphores. These semaphores are mapped to the upper region of the physical memory address space; however, a dedicated semaphore controller is integrated at the memory stage of the pipeline to manage access to the hardware-based semaphores.

Furthermore, the hardware-based semaphores have two configurations. The first configuration uses a stalling-driven approach, while the second configuration is an interrupt-driven method managed by the custom coprocessor. The coprocessor also integrates an additional ALU that performs arithmetic computations using custom instructions on shared data concurrently and instantaneously during data retrieval.

An ideal functional model of the multi-core interconnect was designed to enable an unbiased comparison with the hardware-based semaphore synchronization model by minimizing the WCET to its theoretical limit.

The test cases were designed to reflect real-world scenarios in safety-critical applications. The hardware-based semaphore was compared with the software-based semaphore using the following data flow patterns: multiple producers to a single consumer, a single producer to multiple consumers, and sequential data exchanges across multiple partitions.

Additional test cases were designed to test the four distinct synchronization configurations for the hardware-based semaphore synchronization model. The first configuration is without the custom coprocessor's ALU and without IRQ support, and the second is with the coprocessor's ALU but without IRQ support. The next configuration is with the custom coprocessor's ALU and IRQ support, and the last is with IRQs but without the coprocessor's ALU.

4.0.1 ZYNQ SoPC hardware-based semaphore

The performance evaluation was conducted using assembly code implementing the semaphore waiting (sem_wait) and posting (sem_post) operations as a simple synchronization procedure. The evaluated multi-threaded application transfers a shared variable between two cores, with each core modifying its value before passing it to the other. This example demonstrates that the proposed methodology provides not only synchronization between the participating cores but also synchronized data transfer. In this respect, the mechanism can be compared to synchronous communication channels, where data is transferred only when the corresponding synchronization condition is satisfied. The synchronization unit can therefore be used to transfer individual variables or larger blocks of data while maintaining synchronization between the communicating cores [36].

Meanwhile, the hardware-based semaphore can perform the same synchronization as the ARM cores using only *LDR* and *STR* instructions. Furthermore, both cores use the internal On-Chip Memory (OCM) located within the Processing System (PS). This guarantees the shortest possible access time for the cores to exchange shared data [36].

	Waiting for semaphore	Posting semaphore
HW Support	`sem_wait:` `ldr r1,[r0]`	`sem_post:` `str r1,[r0]`

SW Only	`sem_wait:`
	`ldr r1,[r0]`
	`adds r1,r1,#0`
	`beq sem_wait`
	`sub r1,r1,#1`
	`str r1,[r0]`
		`sem_post:`
		`ldr r1,[r0]`
		`adds r1,r1,#0`
		`bne sem_post`
		`add r1,r1,#1`
		`str r1,[r0]`

Figure 4.1 Comparison of Software and Hardware Supported Semaphore Systems Use [36]

Table 4.1 Average synchronization time using software and hardware semaphores

Semaphore Type	Synchronization Delay (in clock cycles)
Software	32
Hardware	14

The table above shows multi-core synchronization performance using both methods. The synchronization patterns in the test program, for both hardware-based and software-based methods were executed for 100 data exchanges. As a result, hardware-based synchronization is approximately 2.29 times faster than the software-based approach [36].

However, the main limitation of hardware-based synchronization arises from placing the hardware-based semaphore in the Programmable Logic (PL). This implementation requires each data exchange (RD or WR) cycle from the ARM microprocessor to traverse three layers of AXI interconnects and pass through clock-domain crossing synchronization. If the hardware-based semaphore were implemented at the same level as the On-Chip Memory (OCM), it could provide significantly improved performance with a minimal delay of only a few clock cycles [36].

4.0.2 Hardware-based Semaphores

In safety-critical applications, single-slot bounded buffers can be used to exchange data between producer and consumer processes. A producer places shared data into a single-slot buffer, while one or more consumers retrieve the data from the buffer.

The following diagram presents a Petri net model illustrating the conceptual flow of buffer access and synchronization in a single-slot bounded buffer.

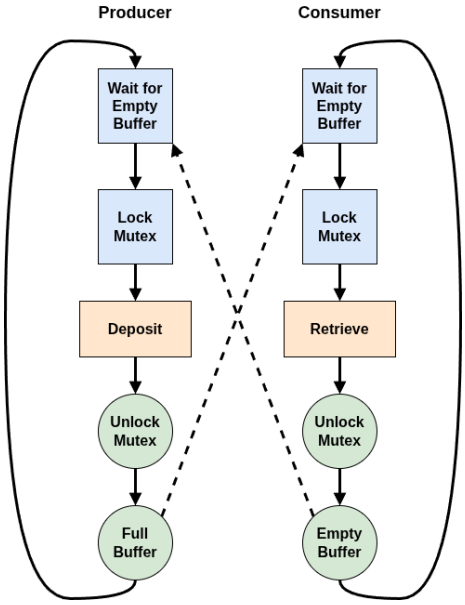


Figure 4.2 Petri net model of single-slot bounded buffer

Two basic synchronization models can be distinguished: one-to-many and many-to-one. In one-to-many model, a single process generates a synchronization event that can release multiple processes, resulting in divergent synchronization. In many-to-one model, multiple processes generate synchronization events that enable a single process to proceed, resulting in convergent synchronization. More complex dependencies can also occur in many-to-many model, where multiple processes synchronize with multiple other processes [39]

These synchronization patterns can be implemented using binary semaphores in conjunction with single-slot bounded buffers. In conventional bounded-buffer implementations, counting semaphores are also commonly used to track the number of available or occupied buffer slots [39]

In contrast, the proposed hardware-based semaphore mechanism uses a consumer counter to track the number of consumers that have yet to access the data stored in a single-slot buffer. This counter does not represent buffer occupancy and therefore does not operate as a conventional counting semaphore. Instead, it provides hardware-supported tracking of consumer synchronization, allowing the producer to determine when all designated consumers have accessed the shared data [39]

4.0.2.1 Idealistic Distributed Shared-Cache

In this evaluation of all the critical section code segments presented in this part of the section, the *ideal* functional model of the quad-core interconnect was used for the analysis [39]

The evaluated scenario consists of a single producer process and multiple consumer processes that communicate through a single-slot bounded buffer. The producer accesses a single-slot bounded buffer protected by an *empty* synchronization variable and a *mutex*. First, the producer executes the *atomic_empty* operation and waits until the *empty* variable allows it to proceed. It then acquires the *mutex* using the *atomic_mutex* operation before accessing the shared data. After storing the data and initializing the counter, the producer releases the mutex and updates the *full* synchronization variable, allowing consumers to access the shared data. [39]

Three consumers sequentially fetch the data stored in the single-slot buffer, with only one consumer accessing the shared data at a time. First, each consumer performs the *atomic_full* operation and waits until the buffer is marked as full. It then acquires the *mutex* through the *atomic_mutex* operation before accessing the shared data. After loading the data, the consumer decrements the shared counter, which determines whether other consumers still need to access the same data. When the counter reaches zero, the last consumer sets the *empty* state, indicating that all consumers have retrieved the data and that the producer can provide new data [39]

The following diagram shows a single producer that shares data with multiple consumers. The segments labeled ”rest of the code” are not drawn to scale in the diagram. Therefore, the atomic full segment of the critical section for the consumer processes is also not to scale [39]

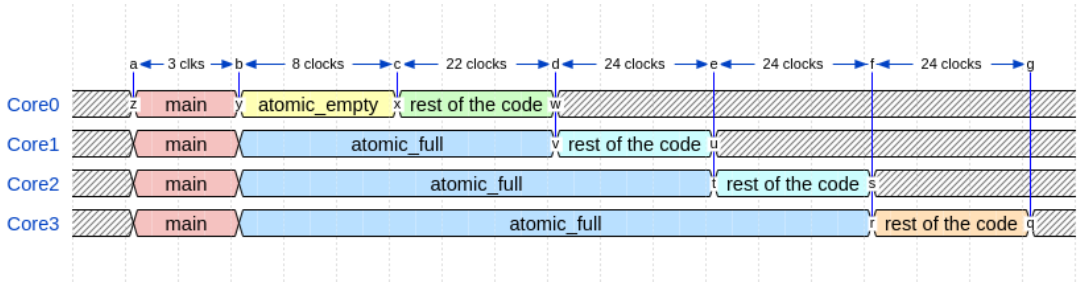


Figure 4.3 Execution model with Single Producer and Multiple Consumers [39]

The following MIPS assembly code segment represents a critical section, in which a single producer places shared data into a single-slot buffer for three consumer processes.

Listing 21. Producer for a Single-Slot Buffer

```
 1: main :
 2:     la $a0, buffer_mutex
 3:     la $a1, empty_mutex
 4:     la $a2, full_mutex
 5: atomic_empty :
 6:     ll $t0, 0x00($a1)
 7:     bnez $t0, atomic_empty
 8:     addiu $t0, $t0, 1
 9:     sc $t0, 0x00($a1)
10:     beqz $t0, atomic_empty
11: atomic_mutex :
12:     ll $t0, 0x00($a0)
13:     bnez $t0, atomic_mutex
14:     addiu $t0, $t0, 1
15:     sc $t0, 0x00($a0)
16:     beqz $t0, atomic_mutex
17: store_data :
18:     la $a3, data
19:     li $t1, 0x33
20:     sw $t1, 0x00($a3)
21: setup_counter :
22:     la $a3, counter
23:     li $t3, 0x03
24:     sw $t3, 0x00($a3)
25: free_locks :
26:     sw $zero, 0x00($a0)
27:     sw $zero, 0x00($a2)
28:
```

The following MIPS assembly code segment shows the consumer portion for each consumer of the critical section, where three consumers fetch data sequentially, one at a time.

Listing 22. Consumer for a Single-Slot Buffer

```
1: main :
2:     la $a0, buffer_mutex
3:     la $a1, empty_mutex
4:     la $a2, full_mutex
5: atomic_full :
6:     ll $t0, 0x00($a2)
7:     bnez $t0, atomic_full
8:     addiu $t0, $t0, 1
9:     sc $t0, 0x00($a2)
10:    beqz $t0, atomic_full
11: atomic_mutex :
12:    ll $t0, 0x00($a0)
13:    bnez $t0, atomic_mutex
14:    addiu $t0, $t0, 1
15:    sc $t0, 0x00($a0)
16:    beqz $t0, atomic_mutex
17: load_data :
18:    la $a3, data
19:    lw $t1, 0x00($a3)
20: decrement_counter :
21:    la $a3, counter
22:    lw $t2, 0x00($a3)
23:    li $t3, 0x01
24:    subu $t2, $t2, $t3
25:    sw $t2, 0x00($a3)
26:    sw $zero, 0x00($a0)
27:    bnez $t2, full
28:    sw $zero, 0x00($a1)
29:    b empty
30: full :
31:    sw $zero, 0x00($a2)
32: empty :
33:    nop
34:
```

The second scenario consists of three producer processes and a single consumer process, with each producer accessing its own private single-slot buffer. Before writing new data, each producer waits until its buffer becomes available and acquires the corresponding buffer mutex. It then stores the data and updates the shared counter, which represents the number of producers that have yet to provide data. When the counter reaches zero, indicating that all producers have provided their data, the buffer is marked as full. Otherwise, the corresponding synchronization state is released, allowing the next producer to proceed. This mechanism ensures that the producers update the shared synchronization state in the required order while each writes to its own private buffer. Once all data has been provided, the consumer retrieves the data from the private buffers and marks the buffer as empty. The following MIPS assembly code segments illustrate the critical sections of the single consumer and one of the multiple producer processes, respectively.

Listing 23. Single Consumer using Single-Slot Buffers

```
1:  ;CoreVarAddr Core1 : data_one, Core2 : data_two, ect.
2:
3:  main :
4:      la $a0, buffer_mutex
5:      la $a1, empty_mutex
6:      la $a2, full_mutex
7:  atomic_full :
8:      ll $t0, 0x00($a2)
9:      bnez $t0, atomic_full
10:     addiu $t0, $t0, 1
11:     sc $t0, 0x00($a2)
12:     beqz $t0, atomic_full
13: atomic_mutex :
14:     ll $t0, 0x00($a0)
15:     bnez $t0, atomic_mutex
16:     addiu $t0, $t0, 1
17:     sc $t0, 0x00($a0)
18:     beqz $t0, atomic_mutex
19: load_data :
20:     la $a3, data_one
21:     lw $t1, 0x00($a3)
22:     la $a3, data_two
23:     lw $t2, 0x00($a3)
24:     la $a3, data_three
25:     lw $t3, 0x00($a3)
26: free_locks :
27:     sw $zero, 0x00($a0)
28:     sw $zero, 0x00($a1)
```

Listing 24. Producers with Private Single-Slot Buffer

```
1:  main :
2:      la $a0, buffer_mutex
3:      la $a1, empty_mutex
4:      la $a2, full_mutex
5:  atomic_empty :
6:      ll $t0, 0x00($a1)
7:      bnez $t0, atomic_empty
8:      addiu $t0, $t0, 1
9:      sc $t0, 0x00($a1)
10:     beqz $t0, atomic_empty
11: atomic_mutex :
12:     ll $t0, 0x00($a0)
13:     bnez $t0, atomic_mutex
14:     addiu $t0, $t0, 1
15:     sc $t0, 0x00($a0)
16:     beqz $t0, atomic_mutex
17: store_data :
18:     la $a3, data_one
19:     li $t1, 0x11
20:     sw $t1, 0x00($a3)
21: load_counter :
22:     la $a3, counter
23:     lw $t2, 0x00($a3)
24:     bnez $t2, nosetup_counter
25:     li $t2, 0x03
26: nosetup_counter :
27:     li $t3, 0x01
28:     subu $t2, $t2, $t3
29:     sw $t2, 0x00($a3)
30:     sw $zero, 0x00($a0)
31:     bnez $t2, not_full
32:     sw $zero, 0x00($a2)
33:     b full
34: not_full :
35:     sw $zero, 0x00($a1)
36: full :
37:     nop
```

This diagram represents the data flow from multiple producers to a single consumer. The segments labeled ”rest of the code” in this diagram are not drawn to scale. Thus, neither the atomic empty nor the atomic full segments are shown to scale.

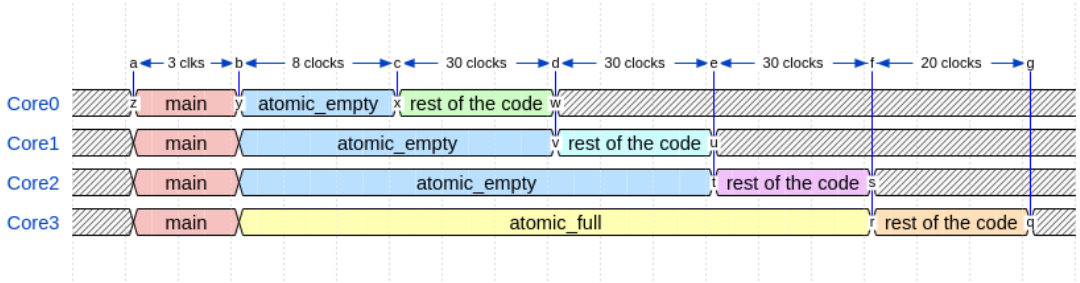


Figure 4.4 Execution model with Multiple Producers and Single Consumer [39]

4.0.2.2 Shared-Semaphore Cache

The performance of hardware-based semaphore synchronization is examined in the second part of the evaluation. In this part of the evaluation, hardware-based semaphores are also required to exchange shared data between producers and consumers. Therefore, the following assembly code segments depict the sharing of data between a single producer and multiple consumers. The result is shown in the following diagram.

Listing 25. Single Producer using Hardware Semaphores

```
1: main :
2:     lui $at, 0x6000
3:     addi $a0, $at, 0x0301
4:     li $t0, 0x33
5: store_data :
6:     sw $t0, 0x00($a0)
7:
```

Listing 26. Multiple Consumers using Hardware Semaphores

```
1: main :
2:     lui $at, 0x6000
3:     addi $a0, $at, 0x01
4: load_data :
5:     lw $t0, 0x00($a0)
6:
```

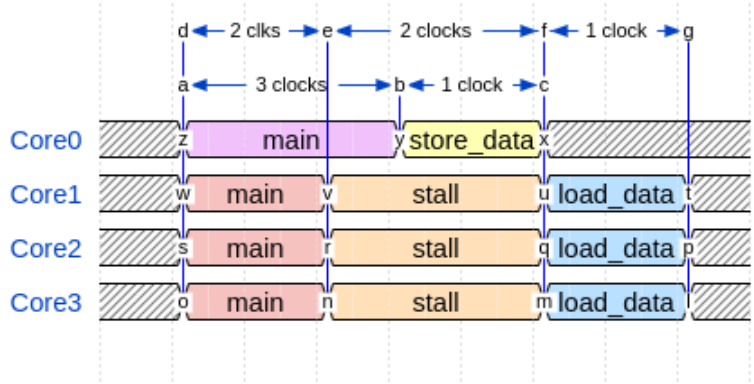


Figure 4.5 Execution model with Single Producer and Multiple Consumers using Hardware [39]

The following assembly code segments demonstrate data sharing between multiple producers and a single consumer; the result is shown in the diagram below.

Listing 27. Multiple Producers using Hardware Semaphores

```
1:  ;AddrPerCore  Core1 : 0x0101, Core2 : 0x0102, ect.
2:  ;DataPerCore  Core1 : 0x11, Core2 : 0x22, ect.
3:
4:  main :
5:      lui $at, 0x6000
6:      addi $a0, $at, 0x0101
7:      li $t0, 0x11
8:  store_data :
9:      sw $t0, 0x00($a0)
10:
```

Listing 28. Single Consumer using Hardware Semaphores

```
1:  main :
2:      lui $at, 0x6000
3:      addi $a0, $at, 0x01
4:  load_data :
5:      lw $t0, 0x00($a0)
6:      lw $t1, 0x01($a0)
7:      lw $t2, 0x02($a0)
8:
```

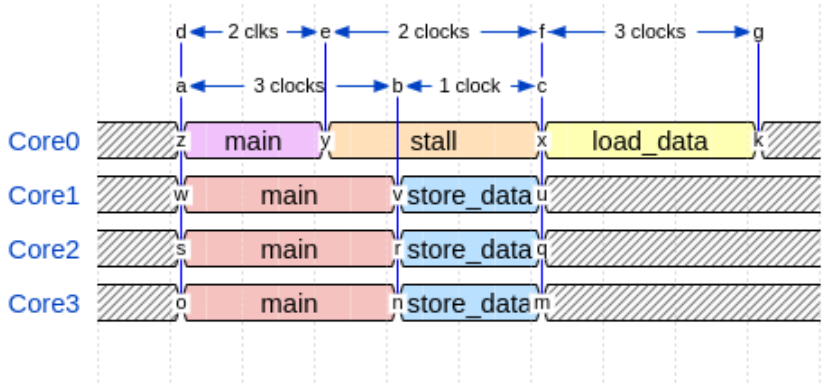


Figure 4.6 Execution model with Multiple Producers and Single Consumer using Hardware [39]

The subsequent MIPS assembly code segments illustrate the core-by-core sequential data flow employed to assess the efficacy of hardware-based semaphores within a framework where each core is tasked with fetching and transmitting shared data. The outcomes are depicted in the accompanying diagram.

Listing 29. Multiple Producers using Hardware Semaphores

```
1: main :
2:     lui $at, 0x6000
3:     addi $a0, $at, 0x0101
4:     addi $a1, $at, 0x04
5: store_data :
6:     li $t0, 0x11
7:     sw $t0, 0x00($a0)
8: load_data :
9:     lw $t1, 0x00($a1)
```

Listing 30. Multiple Consumers using Hardware Semaphores

```
1: ;LdAddrPerCore  Core1 : 0x01, Core2 : 0x02, ect.
2: ;StrAddrPerCore  Core1 : 0x0102, Core2 : 0x0103, ect.
3: main :
4:     lui $at, 0x6000
5:     addi $a0, $at, 0x01
6:     addi $a1, $at, 0x0102
7: load_data :
8:     lw $t0, 0x00($a0)
9:     addi $t1, $t0, 0x11
10: store_data :
11:    sw $t1, 0x00($a1)
```

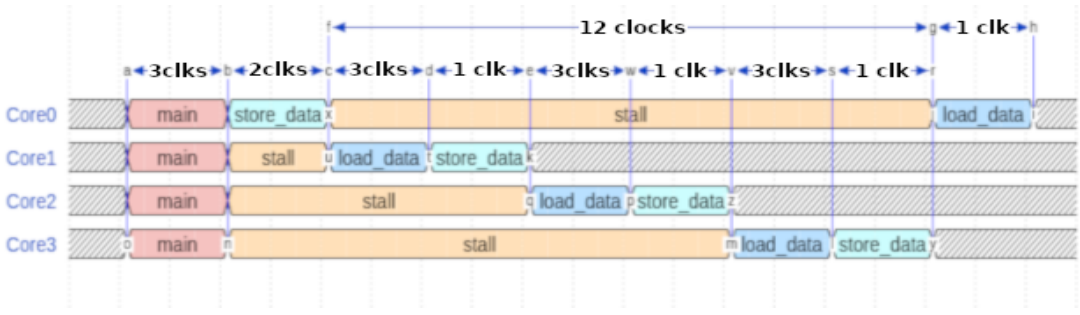


Figure 4.7 Execution model with Multiple Producers and Multiple Consumers [39]

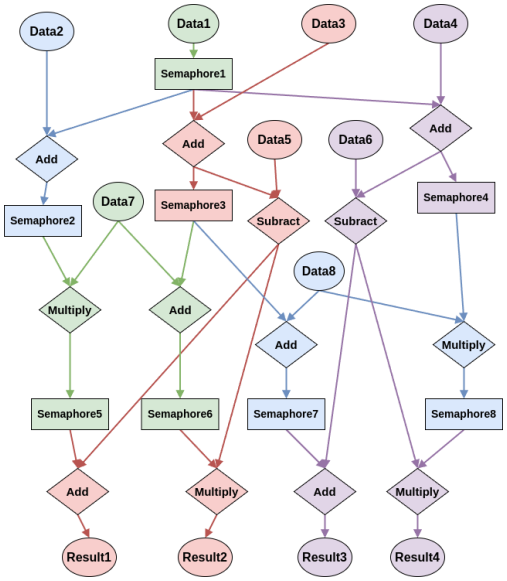
4.0.3 Extended Synchronization Methodologies

4.0.3.1 Test Cases

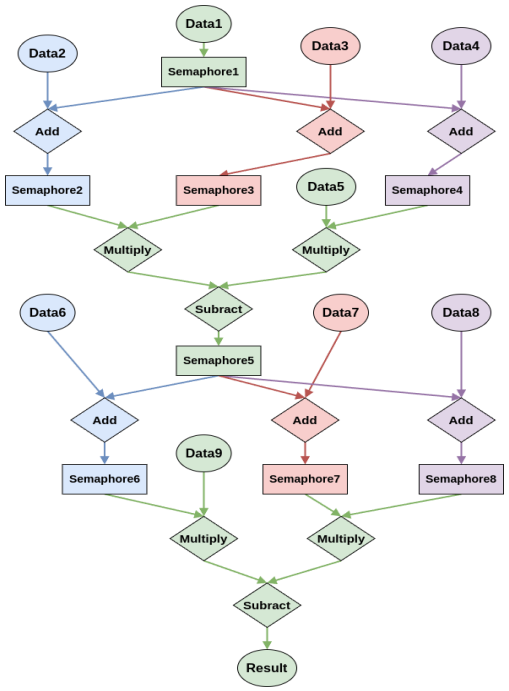
The final evaluation examines a set of test cases designed to compare stalling-based and interrupt-driven multicore synchronization. The patterns describe different ways in which data are exchanged among four partitions. The test cases were selected to cover a representative range of communication and synchronization scenarios rather than to reproduce a single specific application. This allows the performance of the hardware-based semaphore synchronization mechanism to be evaluated under diverse data-flow conditions that may be encountered in safety-critical firmware. Each test case has been designed to highlight a particular characteristic of the data flow and its potential impact on synchronization performance.

The test cases represent different levels of synchronization demand and contention, enabling the influence of these characteristics on execution performance to be investigated. For each test case, the expected behavior is considered from a theoretical perspective based on the corresponding data-flow pattern, as well as the number and ordering of synchronization operations.

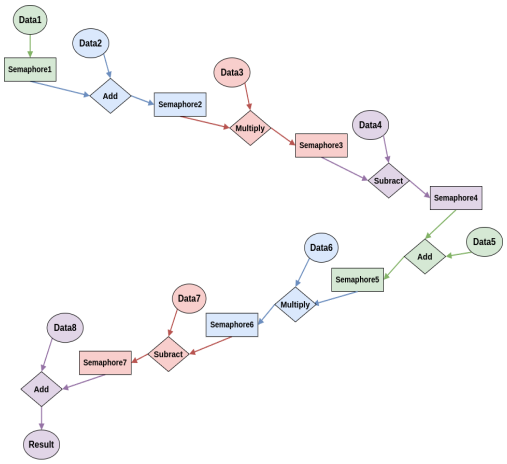
The following figures present the 16 test cases. The experimental results are subsequently compared across all test cases to determine the overall performance characteristics of the hardware-based semaphore synchronization mechanism.



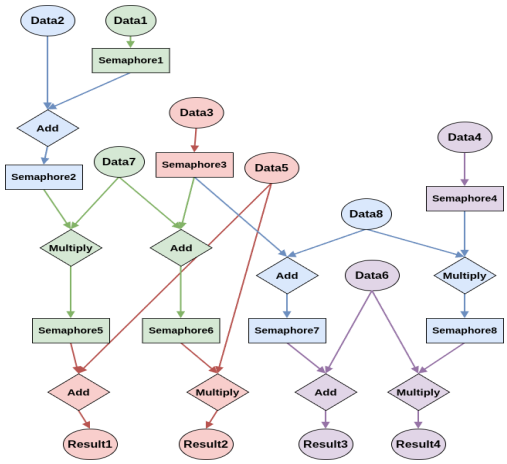
(a) Test Case 1



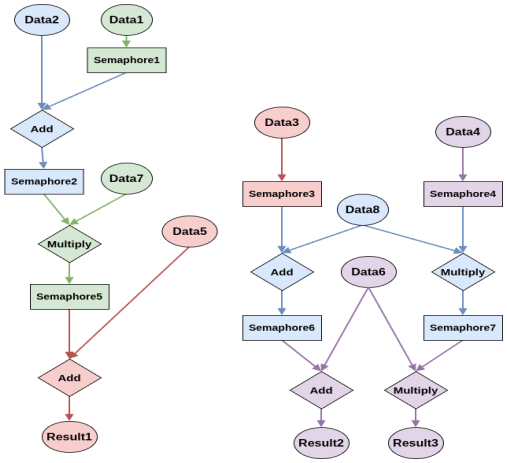
(b) Test Case 2



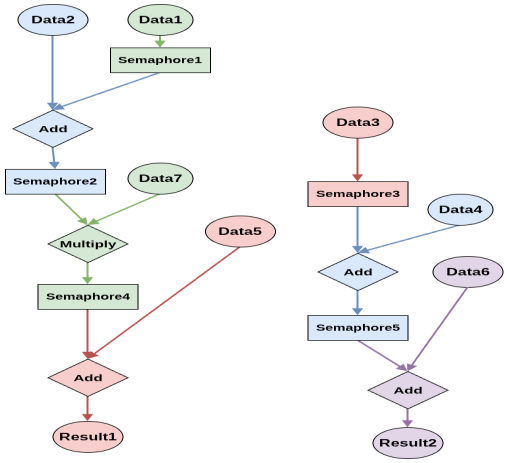
(a) Test Case 3



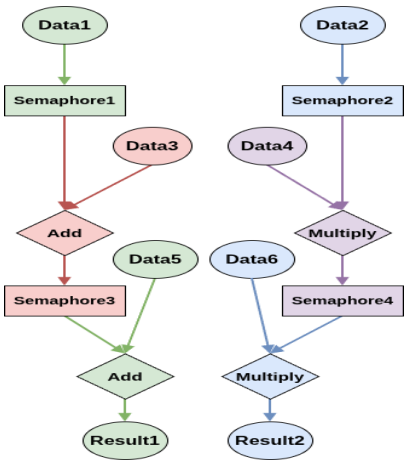
(b) Test Case 4



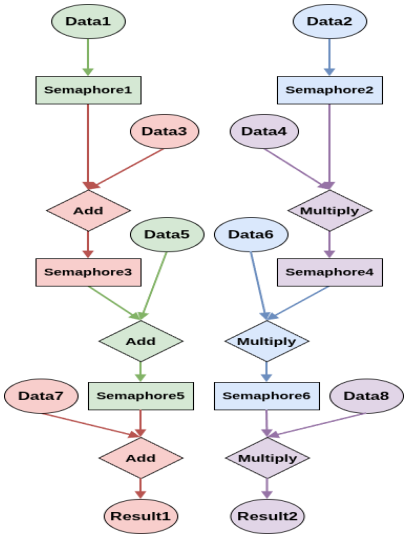
(c) Test Case 5



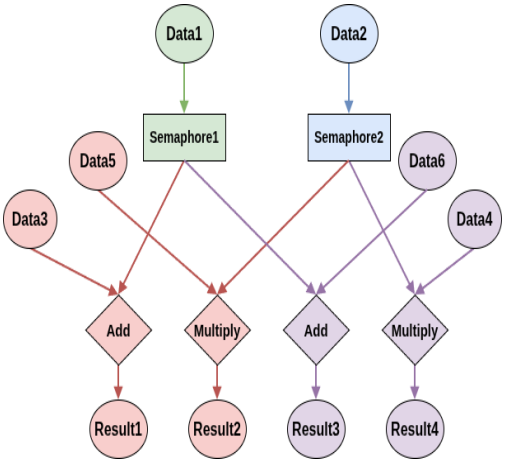
(d) Test Case 6



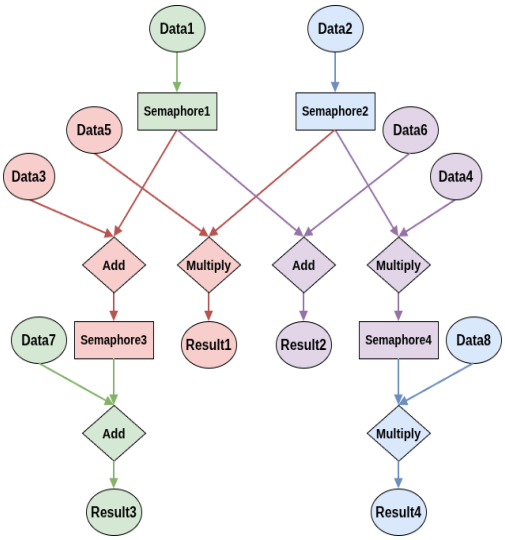
(e) Test Case 7



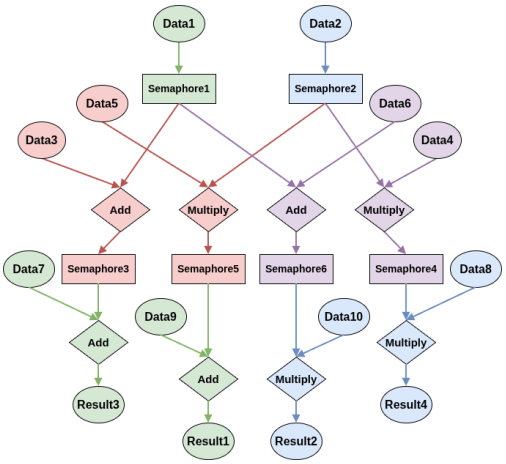
(f) Test Case 8



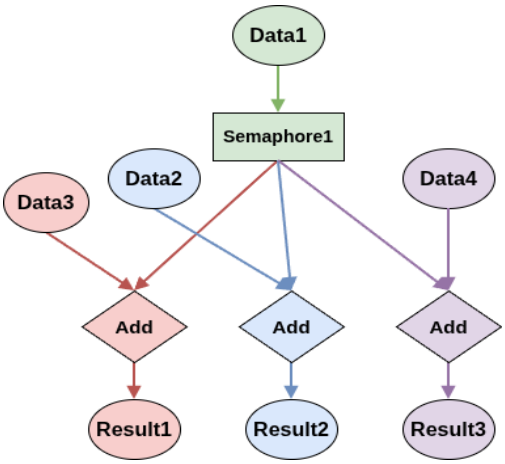
(a) Test Case 9



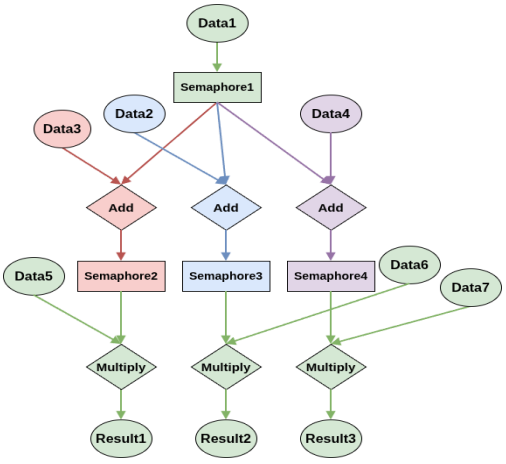
(b) Test Case 10



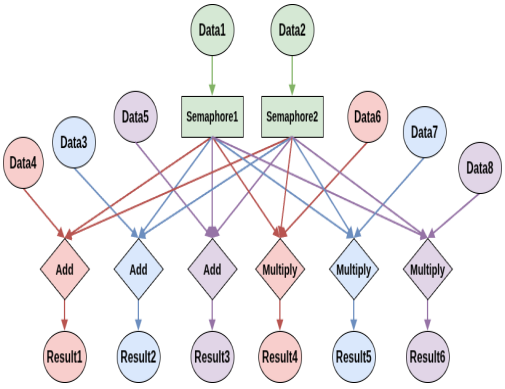
(c) Test Case 11



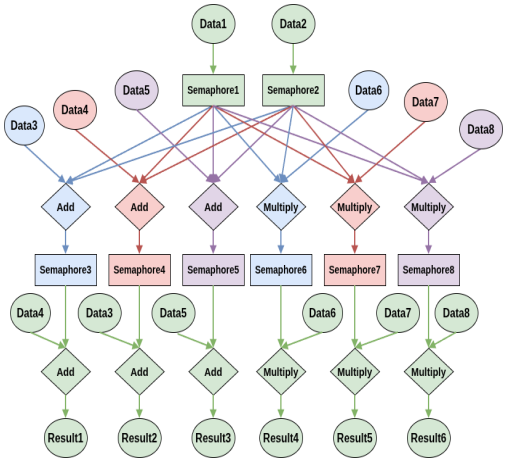
(d) Test Case 12



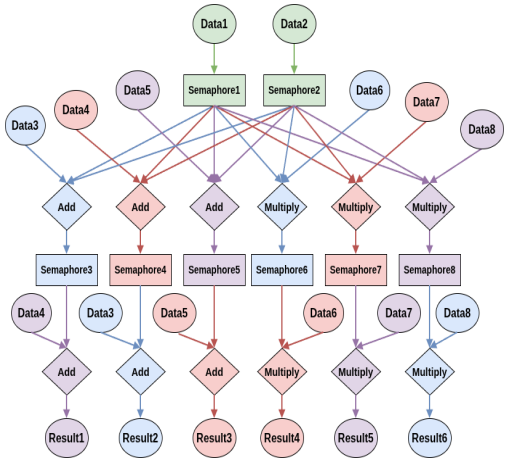
(e) Test Case 13



(f) Test Case 14



(a) Test Case 15



(b) Test Case 16

4.0.3.2 Results

The following table presents the results for all test cases under four different hardware-based semaphore synchronization configurations, with the results expressed in estimated clock cycles. The first configuration operates without IRQ support and without the custom coprocessor's ALU. The second configuration operates without IRQ support but includes the custom coprocessor's ALU. The third configuration includes IRQ support but excludes the custom coprocessor's ALU. The fourth configuration enables both IRQ support and the custom coprocessor's ALU.

Table 4.2 Test Case Results

Test Case	HW_SEM	HW_SEM_CP	IRQ	IRQ_CP
1	18	17	44	44
2	29	27	85	85
3	30	30	70	70
4	22	21	70	70
5	19	18	65	65
6	18	18	51	51
7	14	13	37	37
8	18	18	47	47
9	12	10	39	39
10	14	14	52	52
11	19	17	52	52
12	9	9	27	27
13	21	18	70	70
14	13	11	41	41
15	34	25	116	116
16	21	19	67	67

Evaluation and Conclusions

5.1 Evaluation of ZYNQ SoPC hardware-based semaphore

The initial evaluation introduces the concept of the proposed hardware-based synchronization utilizing the Xilinx Zynq-7000 platform. The synchronization mechanism is grounded in the use of a semaphore, as it minimizes synchronization overhead to simple read and write operations. The hardware-based mechanism requires 14 clock cycles for synchronization, compared with 32 clock cycles for the software-based approach, resulting in a 56.25% reduction in synchronization latency. Moreover, the semaphore mechanism naturally enables a stalling-based approach to concurrency management. Consequently, the overall design guarantees ease of implementation at the software level.

5.2 Evaluation of Hardware-based Semaphores

Buffers are commonly implemented within critical sections to regulate the exchange of shared data between processes. This mechanism ensures interference-free data flow in safety-critical firmware; however, it introduces additional instructions that the microprocessor must execute. Furthermore, only a single core can access the buffer at any one time, which stalls the remaining cores. In contrast, the proposed hardware-based semaphores allow multiple cores to perform load and store operations simultaneously, thereby reducing the duration of multi-core synchronization. For the one-to-many synchronization pattern, the hardware-based semaphore mechanism requires only 5 clock cycles, compared with 105 clock cycles for the software-based approach. For the many-to-one pattern, the corresponding times are 7 and 121 clock cycles, respectively. These results demonstrate a substantial reduction in synchronization overhead when using the proposed hardware-based approach.

5.3 Evaluation of Extended Synchronization Methodologies

The third evaluation revealed that the enhancement in performance regarding multi-core synchronization attributable to the additional ALU integrated within the custom coprocessor is insignificant, particularly considering the elevated demands on logic gate utilization and silicon area. Furthermore, the proposed stalling-driven synchronization approach is faster than the interrupt-driven mechanism. However, its effectiveness depends on the multi-threaded compiler's ability to generate well-coordinated and properly synchronized data exchange operations among multiple cores.

The interrupt-driven mechanism is slower due to the overhead associated with IRQ handling. Fig.5.1 demonstrates the delay that arises from saving and restoring the program counter, as well as from the vector jump to the interrupt handler and the return process. These operations include stalling and flushing the pipeline.

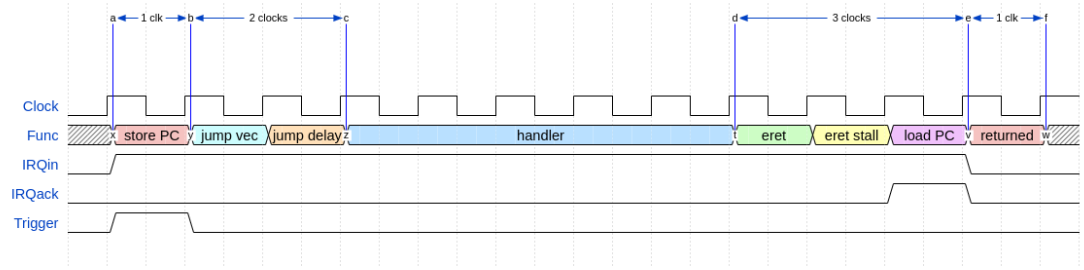


Figure 5.1 Waveform of the Delays during Handling IRQ [38]

Additionally, as shown in Fig.5.2, the IRQ handling operation must be fully completed before another interrupt can be serviced. As a result, the delays accumulate over time.

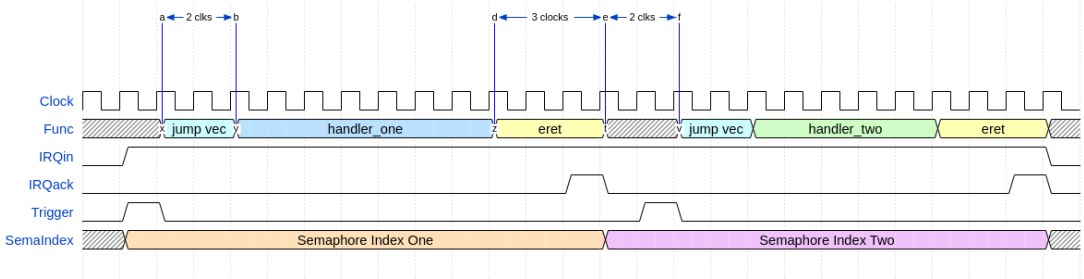


Figure 5.2 Waveform of the Delay between IRQs [38]

5.4 Achieved Aims and Objectives

The proposed methodology targets statically defined multi-core systems in which the compiler decomposes an application into a predefined number of tasks or threads and maps them to dedicated processing cores. This execution model is

particularly relevant to safety-critical applications, where predictable task execution and communication are essential. Within this context, cache-based synchronization is inherently non-deterministic and can significantly increase the Worst-Case Execution Time (WCET), making it unsuitable for such applications. Similarly, synchronization based on Inter-Processor Interrupts (IPIs) introduces additional latency due to Interrupt Request (IRQ) handling. In contrast, the proposed multi-core synchronization model provides fast and predictable synchronization through low-level load and store operations, while allowing concurrent control between statically assigned tasks.

5.5 Future Work

Multi-core architectures are increasingly prevalent in embedded and safety-critical applications. This trend is driven by growing demand for greater processing capabilities, fueled by the increasing complexity of embedded software, particularly in the industrial automation and automotive sectors. Integrating multiple cores within a single microprocessor enables concurrent task execution. In contrast to personal computers, safety-critical applications are required to guarantee the minimum feasible WCET. However, conventional multi-core synchronization models significantly increase the WCET, which is problematic for safety-critical applications. Therefore, IPIs are often used because they provide deterministic synchronization across multiple cores; however, IPIs also introduce overhead related to IRQ management.

The proposed multi-core synchronization model emphasizes a stalling-driven approach and the precisely tailored decomposition of the program into a given number of threads that reflects the number of available processing units because it does not incur additional overhead from synchronization mechanisms. Moreover, the hardware-based semaphore design ensures simplicity and ease of use at the software level, as concurrent control can be implemented using low-level load and store instructions. Furthermore, the proposed multi-core synchronization model is specifically designed to execute tasks sequentially while enabling each task to be divided into multiple fragments that are executed concurrently across different cores.

The proposed synchronization model becomes even more relevant in the context of PLCs due to the necessity for PLC programming languages to manage data flow explicitly through defined variables and a process image memory representing the physical I/O. Furthermore, PLC programs are allocated by the respective compiler and executed in a deterministic manner following a predefined sequence. This allows each task within a PLC program to be divided into multiple fragments running simultaneously across different cores, thereby increasing overall processing power and enabling more complex software to run on PLC devices.

Software Background and Literature Overview

6.1 Introduction

6.1.1 Compilers

Nowadays, a programming toolchain is an essential component of every microprocessor family. The proposed multiple-core, tightly synchronized architecture requires a dedicated compiler capable of automatically exploiting the features of the proposed architecture.

In contrast to conventional programmatic synchronization systems, the proposed approach enables a program to be divided into relatively short execution segments with frequent synchronization points, without introducing additional program complexity or significant execution-time overhead. The proposed synchronization mechanisms enable more efficient utilization of hardware resources by precisely distributing program fragments among the available processing resources.

Compilers are essential tools in software development, as they enable engineers to write complex software in high-level languages that are then translated into machine code. Without such tools, engineers would be compelled to write directly in assembly language, which varies according to each processor architecture [1, 2, 10].

A compiler consists of three main components: the front-end, middle-end, and back-end. The front-end of a compiler analyzes the source code of a high-level language and generates an intermediate representation (IR). The middle-end analyzes the IR and applies optimizations and transformations to improve the generated code or meet specific requirements. The back-end then generates machine code from the optimized intermediate representation and links all program components into a unified memory image representing the complete executable program [1, 2, 10].

One of the most widely used open-source compilers is LLVM, which provides a fully customizable compiler framework. The framework consists of passes, which are individual components of the front-end, middle-end, or back-end. These passes are organized into a pipeline, allowing the compiler to process source code step by

step. Furthermore, the LLVM framework enables developers to add, remove, or modify passes as needed [41].

6.1.2 PLC Programming Languages

A PLC operates by continuously executing a repetitive process known as the scan cycle, as shown in Fig. 6.1. During each scan cycle, the PLC reads the status of the input devices, executes the control program based on the acquired input data, updates the output devices accordingly, and performs diagnostic checks. This cycle is repeated continuously, enabling the control system to respond to changes in the controlled process in a timely and predictable manner.

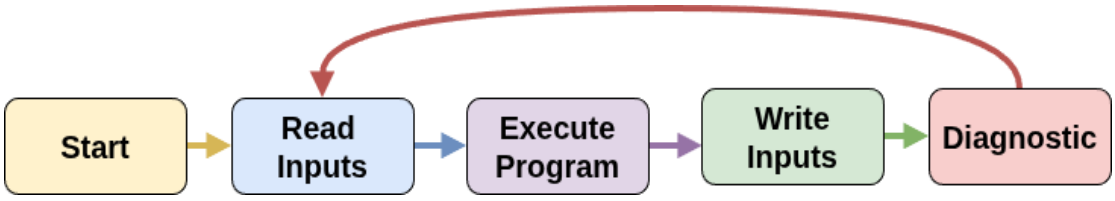


Figure 6.1 Diagram of the PLC Scan Cycle

The IEC 61131-3 standard specifies programming languages, program organization units (POUs), and standardized functions and function blocks, while supporting data encapsulation through local variables and defined interfaces, which promote data separation and limit unintended interactions between program components. A control program can be designed using one or more of the programming languages defined by IEC 61131-3, with the applicable language depending on the type and purpose of the individual POU. Furthermore, IEC 61131-3 defines standardized functions and function blocks for commonly required operations, including stateful blocks such as timers, thereby reducing the need for independent implementations and helping to reduce implementation errors. PLC vendor platforms provide the underlying runtime system in which POUs are developed, configured, and executed, enabling the automation of industrial systems without requiring engineers to develop the underlying software architecture themselves.

Furthermore, PLC programming environments, such as OpenPLC, provide mechanisms for configuring program resources, managing predefined variables, and configuring the input and output channels of PLCs.

IEC 61131-3 defines multiple programming languages that allow equivalent control functionality to be expressed using different representations. A key characteristic of most PLC programming languages is their graphical nature. For example, Ladder Diagram (LD) representation is inherently Boolean-based and follows a standardized evaluation order, with logic typically evaluated from left to right and from top to bottom. This graphical representation facilitates the visualization of concurrent control logic, in

a manner similar to HDL-based design flows, where structural descriptions can expose parallelism.

6.2 Front-End Compiler

6.2.1 Lexical Analyzer

Both compilers and interpreters share components in the front-end of the compiler. The front-end is typically divided into three main parts. Each part is responsible for translating source code written as a sequence of characters into an Abstract Syntax Tree (AST) that the compiler's middle-end can use efficiently [1, 2, 10].

The initial stage of the compiler front-end is handled by the lexical analyzer. It systematically scans the source code to identify discrete units that have specific meanings in a given programming language, called tokens. These tokens correspond to terminal symbols of the language and include keywords, operators, identifiers, literals, and symbols representing punctuation. This step is crucial because it transforms the raw text into a sequence of tokens that is used by subsequent stages of the compiler front-end [1, 2, 10].

Token recognition is performed using automata that utilize tree structures for fast matching, as illustrated in Fig. 6.2. The figure represents a tree structure consisting of two keywords, program and process [1, 2, 10].

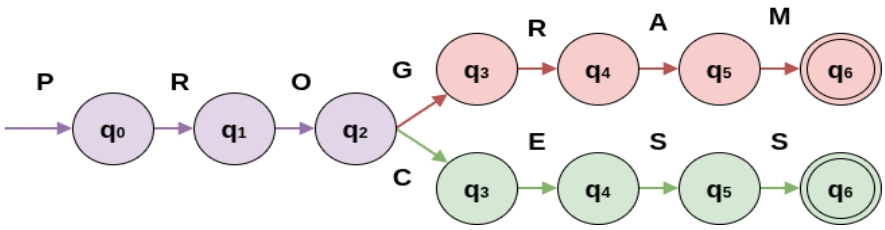


Figure 6.2 Diagram of recognizing "program" token by the Automata

The automaton enables resource-efficient token recognition by representing possible input sequences as a set of states and transitions. During lexical analysis, the automaton reads the input symbols sequentially and changes its current state according to the transition function associated with the current state and the input symbol. A token, such as a keyword or operator, is recognized when the sequence of input symbols leads the automaton to an accepting state corresponding to that token. Automata used for lexical analysis can be classified into Deterministic Finite Automata (DFA) and Non-deterministic Finite Automata (NFA), as presented in Fig.6.3 [1, 2, 10].

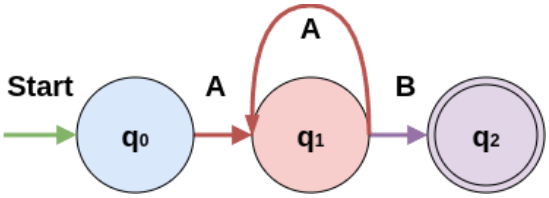


Figure 6.3 Diagram of recognizing ”aa*b” token by Non-Deterministic Finite Automata

DFA allows at most one transition from a given state for each input symbol. Therefore, when a particular input symbol is read, the next state is uniquely determined, and the automaton can follow only one computation path. In contrast, NFA may have multiple transitions from the same state labeled with the same input symbol. Consequently, reading a particular input symbol can lead to multiple possible next states and, therefore, to multiple possible computation paths. This nondeterministic behavior provides greater flexibility when representing regular languages and is particularly useful during the construction of automata from regular expressions. In lexical analysis, Thompson's construction algorithm uses NFAs to represent regular expressions, which can subsequently be converted into equivalent DFAs for efficient token recognition [1, 2, 10].

Thompson's construction algorithm constructs an NFA from a regular expression by combining smaller automata corresponding to its basic components. This allows complex regular expressions, such as those describing integer literals or other token patterns, to be represented using a nondeterministic automaton. The resulting NFA can then be converted into an equivalent DFA, which provides deterministic state transitions and can be executed efficiently during lexical analysis. [1, 2, 10].

Additionally, because DFA can occupy a large amount of memory, the DFA minimization algorithm is applied to optimize the automaton, reducing the number of states. This is done using a partitioning method, in which states that are equivalent in behavior are grouped together and merged into a single state [1, 2, 10].

Lexical analyzers can also be constructed using tables that recognize the lexical patterns of a programming language; furthermore, scanners can roll back to revisit the last position in a string where a character matched a valid state for a possible token. This mechanism is part of the compiler's broader error-handling strategy, which helps report meaningful error messages after unsuccessful compilation [1, 2, 10].

6.2.2 Syntax Analyzer

The second stage of a compiler's front-end is the syntax analyzer, which determines whether the sequence of tokens produced by the lexical analyzer conforms to the formal grammar of the programming language. A programming language can be described as a formal language defined by a set of grammatical rules that specify the

valid relationships between its symbols. In the context of syntax analysis, the tokens produced by the lexical analyzer constitute terminal symbols, while non-terminal symbols are used to represent syntactic categories and intermediate structures defined by the grammar. Together with a set of production rules, these elements define the syntactic structure of the language. The grammar can be represented formally using a notation such as Backus–Naur Form (BNF) or by means of syntax diagrams, which describe the relationships between terminal and non-terminal symbols [1, 2, 10].

Parsing determines whether the input sequence of terminal symbols can be derived from the start symbol of the grammar by applying its production rules. In top-down parsing, the analysis starts from the start symbol and progressively replaces non-terminal symbols according to the production rules in an attempt to match the resulting terminal symbols with the input token stream. Thus, the parser proceeds from the general syntactic structure defined by the start symbol towards the individual input tokens. The parsing process can be represented by a parse tree, in which the root corresponds to the start symbol, internal nodes represent non-terminal symbols, and leaf nodes represent terminal symbols [1, 2, 10].

The grammar used for syntax analysis is assumed to be context-free, meaning that the application of a production rule depends only on the non-terminal symbol being replaced and does not depend on the surrounding context. This property allows the parser to process the input efficiently using limited lookahead. In the considered top-down parsing approach, the parser can use the next input token to determine which production rule should be applied, progressively constructing the syntactic structure of the input according to the grammar. This approach simplifies the parsing process while ensuring that the sequence of tokens conforms to the syntactic rules of the language [1, 2, 10].

6.2.3 Semantic Analyzer

Parse trees possess limited practical utility beyond examining a program's syntactic structure. Therefore, a simplified form of the parsing tree is required, obtained by removing most non-terminal nodes. These simplified representations are referred to as ASTs, with an illustrative example provided in Fig.6.4 [10].

Additional measures are necessary to examine the semantics of a programming language. To facilitate this, the parser must incorporate a mechanism to transmit the values and variables needed to construct ASTs. This mechanism creates a node for each operand and allows productions that lack meaningful semantic content to simply return the results of their child nodes. In this manner, the parser prevents the creation of tree nodes that solely represent syntactic non-terminals of the grammar [10].

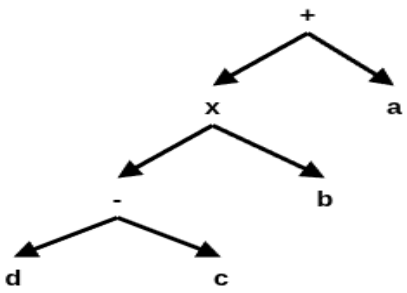


Figure 6.4 Diagram of a Abstract-Syntax Tree

Compilers such as LLVM use three-address code, represented as a sequence of quadruples. Each quadruple consists of four fields: an operator, two source operands, and a destination. These three-address code instructions are arranged linearly, as illustrated in Fig.6.5 [1, 2, 10, 41].

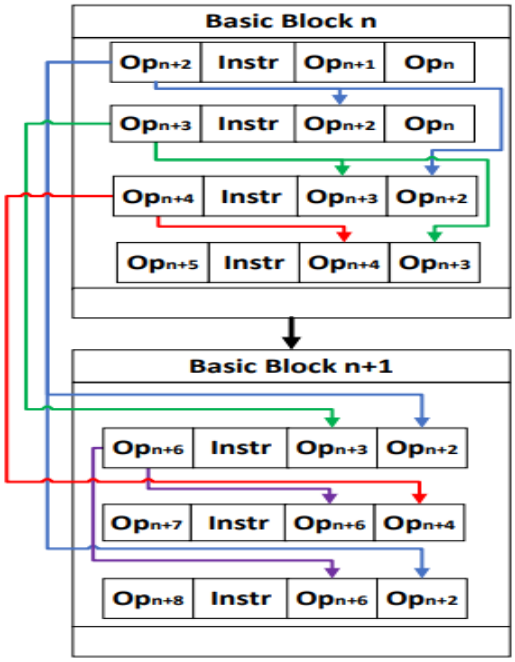


Figure 6.5 Diagram of Three-Address Codes

Furthermore, compilers construct a Control-Flow Graph (CFG) from the linear sequence of three-address code by grouping instructions into basic blocks. Each basic block represents a sequence of consecutive instructions that are executed sequentially, with control flow entering the block at its beginning and leaving it at its end. Conditional and unconditional control-flow transfers occur between basic blocks rather than within them. The construction of the CFG involves identifying the leaders that mark the beginnings of basic blocks, grouping the instructions into blocks, and establishing successor edges that represent the possible transfers of control between these blocks [1, 2, 10].

In a compiled programming language, variables are associated with declared types that provide the compiler with information about their representation and the operations that can be performed on them. For example, integer and floating-point types require different representations and may be processed using different machine instructions. During compilation, type information is used by the front-end and subsequently represented in the IR generated by the compiler. The size and representation of a variable are therefore determined by its declared type and, ultimately, by the rules of the programming language and its implementation [1, 10].

Additionally, programming languages also define scopes that determine the visibility and lifetime of variables. Depending on the language and the context in which a variable is declared, variables may have global, static, local, or block-level scope. As the compiler processes nested scopes, the set of identifiers that are visible to the current context changes accordingly. Symbol tables are used to store information associated with identifiers, such as their names, types, scopes, and other attributes required during compilation. Consequently, the compiler can use the appropriate symbol-table information to resolve references to variables according to their current scope [1, 10].

6.2.4 IEC 61131-3 Standard

Fig.6.6 illustrates the IEC 61131-3 model for the organization of a PLC application. The model defines a hierarchical structure comprising configurations, resources, tasks, programs, and other POUs, such as function blocks and functions. POUs provide modular units for structuring application logic and can encapsulate internal data and state. Communication between POUs can be achieved through explicitly defined interfaces, thereby supporting a degree of data-flow isolation between application components. However, shared-data mechanisms, such as global variables, may weaken this isolation by introducing dependencies between otherwise independent POUs.

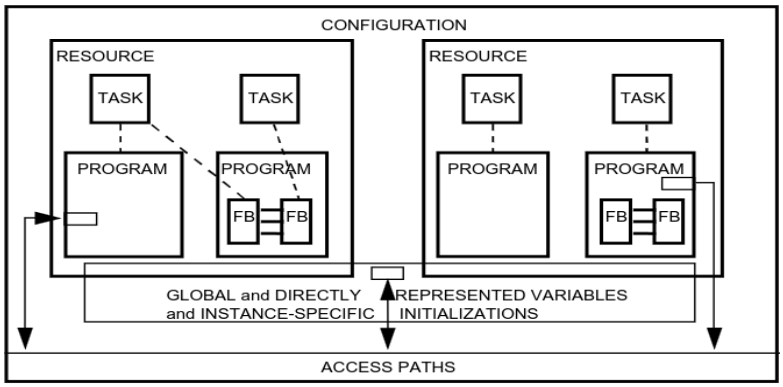


Figure 6.6 Diagram of IEC 61131-3 Application Structure [23]

Dedicated PLC integrated development environments (IDEs) are used to configure and deploy PLC applications and associated device and resource configurations, including variable mappings, I/O assignments, task configurations, and communication parameters. In PLC platforms equipped with auxiliary processors or dedicated hardware controllers, the IDE may also provide configuration mechanisms for these processors, which can be responsible for managing specific hardware resources, such as I/O channels and communication interfaces [23].

Graphical programming languages use dedicated representations that are parsed by the compiler front-end and translated into a common IR. In the case of the open-source PLC IDE OpenPLC, graphical programs are translated into Structured Text (ST), which serves as a common textual representation before being compiled into executable code for open-hardware devices [78].

Such translation is possible because IEC 61131-3 defines programming languages with standardized semantics, allowing equivalent control functionality to be expressed in different language forms. Additionally, graphical languages such as LD follow a standardized execution model, where the logic is evaluated according to the defined network and element order, typically from left to right and from top to bottom, with each rung evaluated sequentially [23, 78, 79]

Therefore, the proposed multicore system and custom LLVM middle-end pass are specifically designed for the main PLC microprocessor, which is responsible for executing the control program written in LD. This compilation approach is conceptually similar to HDLs, such as Verilog, where digital circuits can be described using textual representations while preserving structural information relevant to synthesis. Similarly, during LD-to-ST translation, the underlying combinational logic is preserved in the resulting ST representation, which can then be translated into LLVM IR by RusTy. Dependency information can subsequently be extracted from the resulting LLVM IR and used by the custom LLVM middle-end pass to transform the program into a multithreaded implementation. Additionally, LD does not use explicit if-else statements to represent its primary control logic; instead, logical relationships are expressed through networks of contacts and coils. Similar to combinational digital logic circuits, these relationships determine the resulting outputs, allowing LD networks to be translated into equivalent Boolean expressions in ST [23, 35, 78].

Furthermore, PLC programs consist of POUs, including Programs, Function Blocks, and Functions. Function blocks within a PLC program are partially isolated because they promote modularity and encapsulate control logic; additionally, data flow between them is achieved through explicitly defined variables and I/O interfaces, as shown in the following figure using basic blocks [23].

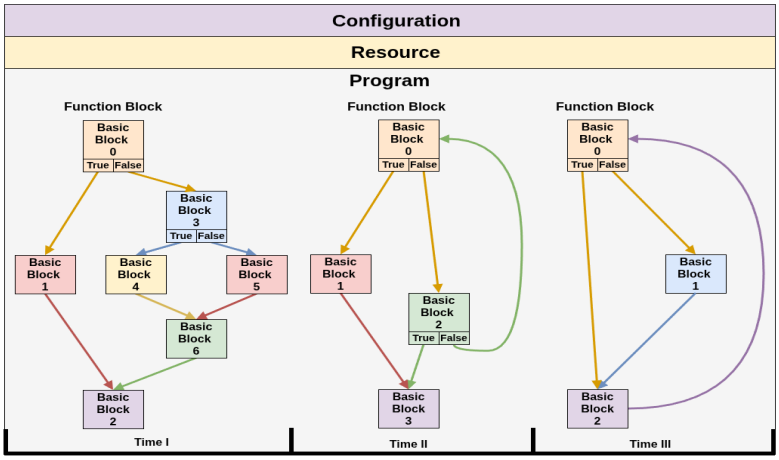


Figure 6.7 Diagram of PLC Program in LLVM

Furthermore, function blocks are executed deterministically in a predefined sequence within the PLC scan cycle. The proposed approach executes individual function blocks concurrently across multiple cores, with functions inlined where applicable. Concurrent execution of function blocks can help mitigate synchronization issues that may arise when multiple function blocks access shared global variables. Stateful standard function blocks, such as timers, can instead be executed at their original positions within the PLC scan cycle. All cores participate in the timer operation before resuming the parallel execution of the remaining combinational logic. This approach preserves the execution behavior of the original single-core PLC while simplifying dependency management in the multithreaded implementation [23].

6.3 Middle-End Compiler

6.3.1 IR Code Analysis

Figure 6.8 illustrates the architecture of the LLVM compiler. which consists of a front-end, middle-end, and back-end, with commonly used predefined middle-end passes highlighted as part of the compilation process. Each component can be customized for specific target machines and programming languages. Furthermore, custom passes are used for analysis, IR transformation, and optimization. The middle-end is responsible for transforming and optimizing the IR. It performs a series of passes that analyze the IR and use the collected information to optimize the program through various IR transformations. These transformations may target a specific machine architecture and generally aim to improve execution speed and reduce memory usage [1, 2, 10, 41].

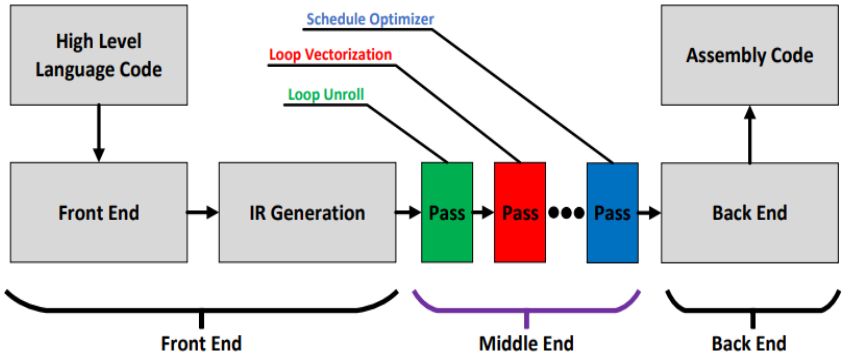


Figure 6.8 Diagram of the LLVM compiler architecture

Commonly used analysis passes construct various graphs based on the generated intermediate representation. Figure 6.9 shows the CFG, which represents the execution flow between basic blocks. This graph captures conditional branches generated by if–else statements, as well as unconditional branches, typically introduced for back edges in loop constructs or via explicit jump instructions. Although a control-flow graph can also be constructed at the level of individual intermediate representation instructions, such representations are less commonly used due to their limited practical benefits [1, 10].

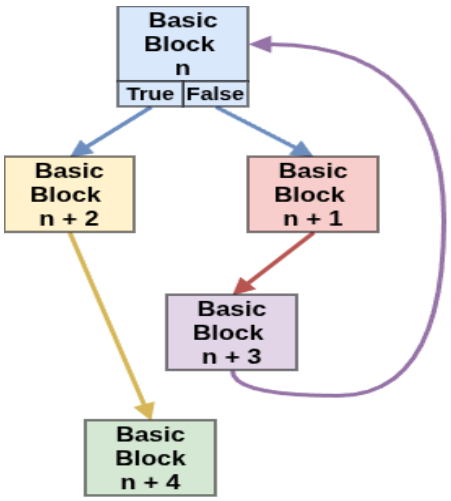


Figure 6.9 Diagram of the CFG

Fig.6.10 illustrates the data dependency graph (DDG), which captures dependencies among IR instructions. This is one of the most commonly constructed graphs because it represents data dependencies within a program and enables optimizations that analyze and transform them [1, 10].

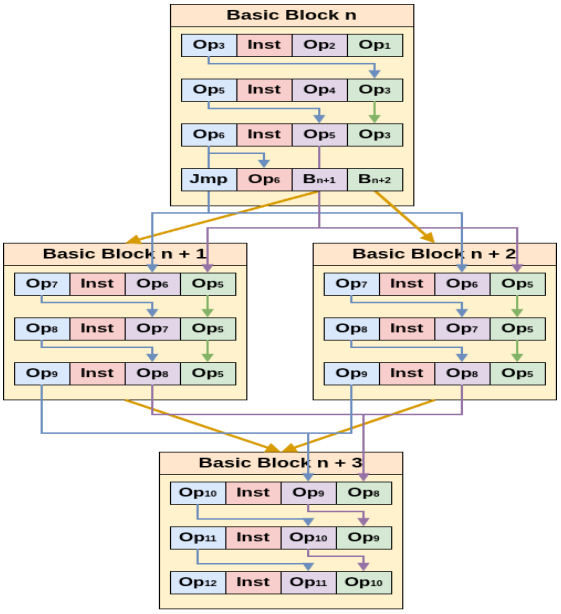


Figure 6.10 Diagram of the DDG

The data dependency graph can be represented using only IR instructions, as shown in the following Figure 6.11. This representation is commonly used for data-dependency analysis and optimization, as well as for scientific graph-based representations of programs, because it enables the use of more graph algorithms, such as Breadth-First Search (BFS) and Depth-First Search (DFS) [1].

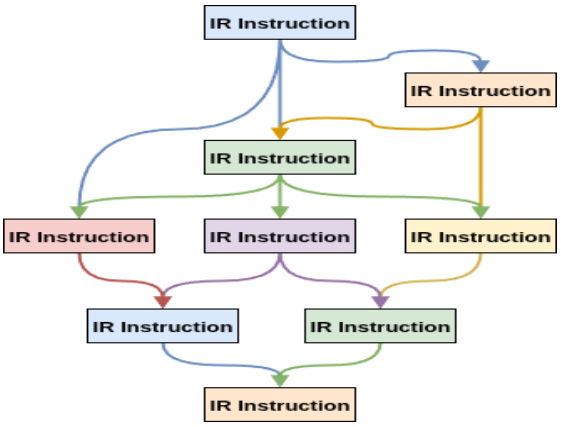


Figure 6.11 Diagram of the DDG with IR Instructions

The third commonly constructed graph is the dominator graph, also known as a dominator tree, as shown in Fig.6.12. It is constructed from the control-flow graph and describes dominance relationships between basic blocks. The dominator tree is widely used in the middle end of a compiler for analysis and optimizations such as Static Single Assignment (SSA) construction, dead code elimination, loop detection, and control-dependence analysis [1, 10].

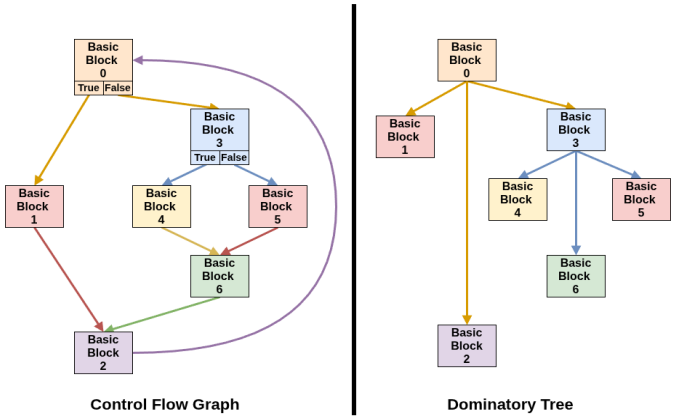


Figure 6.12 Diagram of the Dominator Tree [10]

There are dedicated data-dependency analysis passes that examine IR code for data dependencies, allowing the compiler to detect relationships among memory accesses. Such a pass can determine which memory operations precede their dependent memory operations. Furthermore, it can build on alias-analysis information and provide a lazy-cache interface to efficiently answer common alias-related queries in the back-end [1, 10].

Loop analysis passes identify natural loops and determine the loop depth of nodes in the control-flow graph. They can also analyze and categorize scalar expressions within loops, which is particularly useful for induction-variable substitution and strength reduction. Such scalar analysis can test for dependencies within a single loop iteration or between iterations [1, 10].

6.3.2 Code Optimization

There are multiple transformation passes in a compiler, which can be classified as either regional or global optimizations. Useful regional optimization passes include dead code elimination, which removes code that will never be executed, and dead argument elimination, which removes function arguments that are never used. Additionally, global optimization passes can eliminate unreachable internal global variables, improving overall program efficiency and reducing memory usage [1, 10].

Additionally, commonly used optimization passes include eliminating basic blocks with no predecessors and merging a basic block into its predecessor when the predecessor has only one successor. Basic blocks can also be removed when they do not perform any computations and only serve to transfer control to another block. PHI nodes can also be simplified when a block has a single predecessor. These passes may also move instructions into successor blocks when possible, ensuring that computations are not executed along paths where their results are not needed [77].

The Global Value Numbering (GVN) pass is an optimization technique utilized by compilers to identify and eliminate redundant computations. The pass traverses basic

blocks and assigns a distinct value number to each computed value. It assigns numbers based on the expressions' semantics rather than their syntactic form. This process enhances performance by removing duplicate calculations, simplifying expressions, and facilitating further optimizations such as common subexpression elimination and constant propagation [10, 77].

There are loop optimization passes that transform loops to improve performance, such as fully unrolling loops or unrolling the outer loop and fusing inner loops when variables or memory loads can be shared in the new inner loop. The inline substitution pass enables the compiler to replace a call site with the body of the callee; meanwhile, it makes the necessary modifications to model the effects of parameter binding, as illustrated in the figure below [1, 10, 77].

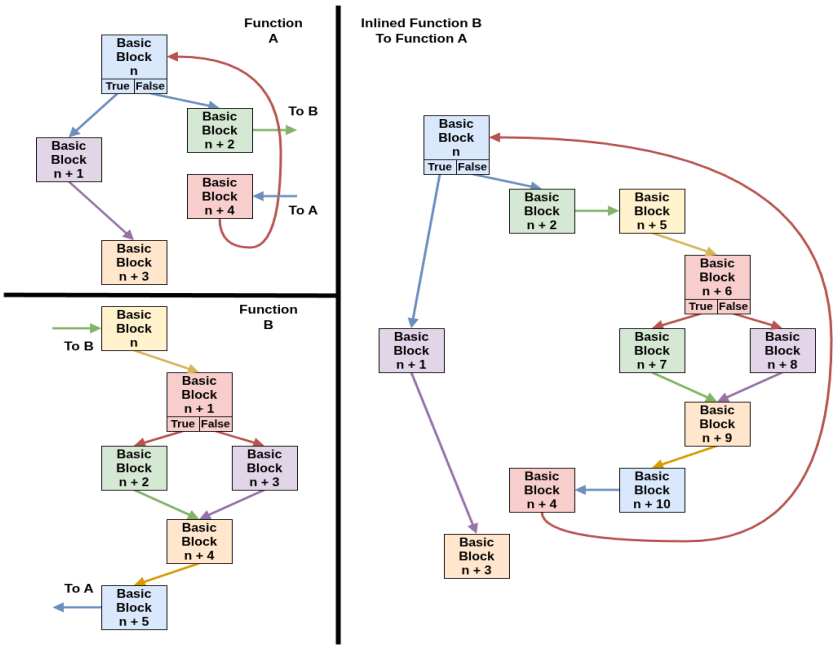


Figure 6.13 Diagram of Inline Substitution

6.4 Back-End Compiler

6.4.1 Instruction Selection

The final stage of a compiler is the back-end, which is responsible for generating assembly and machine code. The primary goal of the back-end is to isolate machine-dependent analysis and optimizations. This allows the compiler to follow an organized, modular approach to code generation from the source code. Additionally, the back-end generates target-specific code for a particular Instruction Set Architecture (ISA) using the universal IR produced by the middle-end. This process includes analyses and optimizations required for instruction selection, register allocation, and

instruction scheduling, as well as incorporating hardware-specific details such as register size, calling conventions, and the appropriate assembly language for the target ISA [1, 10].

Code generation typically involves two primary techniques: pattern matching and peephole optimization. Pattern matching creates assembly code by matching Predefined instruction patterns against expression trees or directed acyclic graphs derived from the intermediate representation. Peephole optimization, on the other hand, improves the low-level intermediate representation by examining small instruction sequences and replacing inefficient patterns with more efficient ones. Both techniques rely on tables or rule sets that store detailed descriptions of the target Instruction Set Architecture [10].

The compiler back-end comprises three main processes, as illustrated in Fig.6.14. The Expander converts intermediate representation instructions into a sequence of Lower-Level Intermediate Representation (LLIR) instructions. Its primary objective is to lower high-level constructs, replace virtual registers with target-specific registers, and insert control-flow elements such as jump labels. The Simplifier is a back-end component that performs optimization passes over the LLIR; it typically applies peephole optimization to eliminate redundant or inefficient instruction sequences. Finally, the Matcher generates target assembly code from the LLIR by selecting the most cost-effective instruction patterns supported by the target instruction set architecture [10].

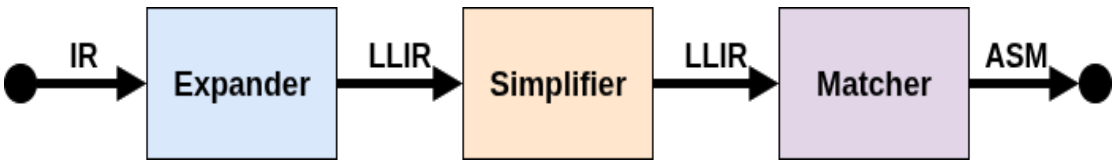


Figure 6.14 Diagram of Back-End Processes

Register allocation is particularly important because most modern instruction sets are based on register-to-register operations. As a result, the compiler back-end must transform intermediate-representation instructions into register-based operations by assigning the appropriate physical registers to each instruction. Although memory-to-memory operations exist and are associated with Complex Instruction Set Computer (CISC) architectures, the influence of Reduced Instruction Set Computer (RISC) design has been profound; consequently, even modern CISC processors primarily operate on registers, with memory accesses largely limited to explicit load and store instructions [1, 10].

The pattern-matching technique employs algorithms analogous to those used in grammar parsing within the compiler front-end. Tree matching facilitates the compiler in optimizing execution efficiency by selecting the least-cost instruction patterns

that encompass the intermediate representation tree, based on cost and constraint information stored in lists describing the target Instruction Set Architecture [1, 10].

Peephole optimization works by examining small sequences of adjacent instructions and identifying local patterns that can be improved. Rather than rewriting an entire low-level intermediate representation at once, it incrementally replaces inefficient instruction sequences with more efficient ones. The primary goal of this optimization is to maximize execution efficiency at each transformation step while preserving the program's correctness [1, 10].

6.4.2 Instruction Scheduling

Most modern microprocessors are superscalar, meaning they can issue multiple instructions concurrently within a single clock cycle. This capability is enabled by the presence of multiple execution units, such as Arithmetic Logic Units (ALUs), floating-point units, and load/store units, which allow different instructions to be processed in parallel, as illustrated in the figure below [1, 10, 44, 57].

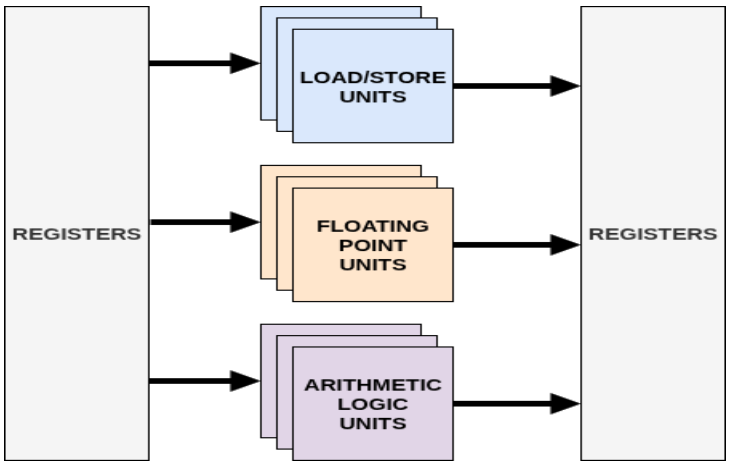


Figure 6.15 Diagram of Superscalar Execution Stage

Additionally, different types of computations require varying numbers of clock cycles to complete, depending on the processor design and the architecture of the execution units. These units enable non-blocking computation within the pipeline, and by effectively rescheduling instructions, the microprocessor can maintain a concurrent and continuous flow of computations, maximizing both pipeline utilization and overall performance. This capability has led compiler engineers to implement instruction scheduling in the compiler back-end, an optimization technique that rearranges instructions to take full advantage of the underlying microprocessor architecture [1, 10, 44, 57].

Instruction scheduling aims to arrange LLIR operations in such a way that their execution does not change the final program result while allowing independent

operations to be executed as efficiently as possible. During scheduling, the dependencies between operations must therefore be taken into account: an operation can be moved only when doing so does not violate the required ordering imposed by these dependencies. In particular, data dependencies determine which operations must be executed before others, while operations that depend on the value produced by a previous operation cannot be reordered across that dependency without changing the program semantics. An example of such strict dependencies is shown in Fig.6.16 [10].

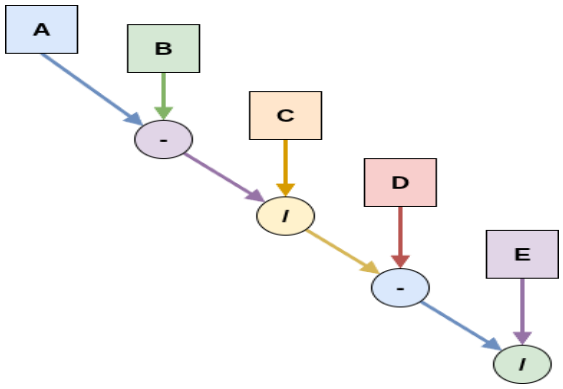


Figure 6.16 Graph with Strict Dependencies

One of the instruction rescheduling techniques is list scheduling, which can be used to optimize instruction execution within a basic block or across a group of blocks. The algorithm constructs a Directed Acyclic Graph (DAG) representing instruction dependencies, including anti-dependencies, to guide scheduling decisions. Additionally, it assigns priorities to instructions to determine which must be executed earlier and which can be deferred to later stages [1, 10].

Load and store operations can stall the microprocessor pipeline for several clock cycles due to latency from cache access and the possibility of a cache miss. In the event of a miss, the cache controller must fetch the appropriate cache line from lower levels of the memory hierarchy. Depending on the cache level where the most recent version of the data resides, this process may take tens, hundreds, or even thousands of clock cycles. Therefore, analyzing the delay caused by load operations is crucial for understanding overall system performance [1, 10, 44, 57].

One technique used to improve the efficiency of list scheduling is trace scheduling, which analyzes critical paths and prioritizes them over less critical ones. This approach enables the scheduler to determine which execution paths should be scheduled earlier than others. Fig.6.17 demonstrates another related technique, basic block cloning, in which instruction blocks are duplicated to form longer linear execution paths. By reducing the impact of control-flow branches, this technique allows list scheduling to operate more effectively, rather than being constrained by branch-induced discontinuities that negatively affect scheduling decisions [1, 10].

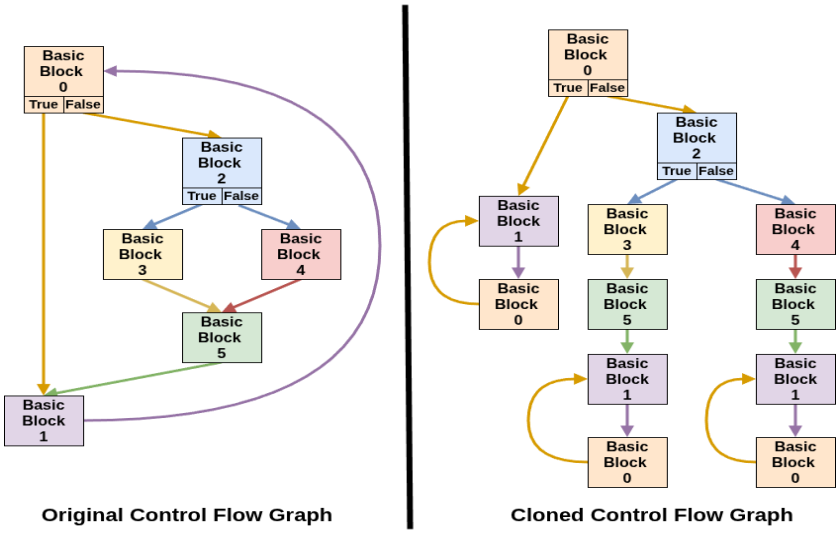


Figure 6.17 Diagram of Cloned Paths in a Graph

Software pipelining is an instruction-scheduling technique applied within loops, since loop operations occur most frequently in programs. This observation has led compiler writers to introduce specialized loop-scheduling mechanisms in the compiler back-end that optimize local regions and closely mirror the behavior of hardware pipelines [1, 10].

Modulo scheduling is a form of list scheduling applied to loop instruction scheduling using a cyclic schedule. It aims to overlap multiple iterations of a loop to increase instruction-level parallelism (ILP) and improve performance. Rather than completing one loop iteration before starting the next, the algorithm reorganizes the instructions of the loop kernel, the central portion of the optimized loop, and executes most of the loop body in an interleaved manner. This is achieved by initiating a new loop iteration every fixed number of cycles, known as the Initiation Interval (II) [1, 10].

It is crucial for the algorithm to examine loop-carried dependencies, which represent values carried across iterations along the control-flow edge associated with the loop-closing branch. If the algorithm fails to find a valid issue slot for an operation, scheduling fails only for the current attempt. In these cases, the algorithm increments the Initiation Interval and retries the scheduling process [1, 10].

6.4.3 Register Allocation

The compiler back-end must allocate live values to registers; such values are identified by frequent changes. At the same time, data that must be preserved but is not used continuously should be identified and stored in memory when necessary. In the middle-end, the compiler represents computed values using virtual registers, which are compiler-level abstractions representing registers without being tied to a specific hardware register. The microprocessor, however, provides only a fixed number of

physical hardware registers. Consequently, a key responsibility of the back-end is to efficiently map virtual registers onto the available hardware registers and to effectively load and store data between registers and memory [1, 10].

The choice between memory-to-memory and register-to-register allocation models is determined at the IR level; therefore, the middle-end is responsible for optimizing the allocation model. However, the back-end must correctly translate this model into target-specific code and efficiently allocate both floating-point and fixed-point values to the appropriate registers, since microprocessors typically use separate register files for floating-point and integer (fixed-point) data types [10, 41].

There are two primary methods of register allocation: the Top-Down and Bottom-Up approaches. The Top-Down method performs a linear traversal to assess the frequency of value usage and assigns virtual registers to hardware registers according to their priority. Since the number of virtual registers representing live values may exceed the number of available hardware registers, the allocator must decide which values should remain in registers and which should be temporarily stored in memory. Values that are used more frequently are generally prioritized for register allocation, while less frequently used values may be moved to memory to make room for new values. Conversely, the Bottom-Up approach relies on comprehensive knowledge of the target ISA and synthesizes multiple low-level facts to inform its decisions. It considers all hardware registers as initially unoccupied and maintains them in a free list. When a new value is produced, the allocator selects a suitable hardware register to store it; if no free register is available, an existing value may need to be transferred to memory to free a register. Through this dynamic management of registers and memory, the Bottom-Up approach seeks to reduce the cost associated with register allocation and memory transfers [1, 10].

The main difference between local and global register allocation is that global allocation considers live values across multiple basic blocks, whereas local allocation is limited to individual blocks. Furthermore, global allocation introduces an additional consideration: the cost of moving values between registers and memory, which can introduce significant execution overhead. A common approach to global register allocation is graph coloring. In the corresponding interference graph, each node represents a live value or virtual register, and an edge connects two nodes when their values are simultaneously live and therefore cannot be assigned to the same hardware register. The available hardware registers are represented by different colors, and the allocator assigns a color to each node so that connected nodes have different colors. Consequently, the number of available colors corresponds to the number of hardware registers available for allocation. If the graph cannot be colored using the available colors, some values must be spilled to memory, freeing registers for other values [1, 10].

In this approach, an interference graph is constructed where each node represents a live range and edges indicate live ranges that cannot share the same register. The allocator then attempts to color the graph using the available number of physical registers, ensuring that no conflicting live ranges are assigned to the same register [1, 10].

6.5 Multi-Core Compiler

6.5.1 Locality

A fundamental method for converting programs into multi-threaded applications suited for multiprocessor architectures, as depicted in Fig. 6.18, is to restructure computationally intensive sections so that their control flow can be determined in a single pass, while iterative loops may still be used to describe repetitive computations. Rather than organizing the computation primarily around sequential control flow, suitable parts of the algorithm can be represented as sets of operations with well-defined data dependencies, enabling independent computations to be executed concurrently. In some applications, these computations can naturally be represented using matrix-based structures, although such a representation is not limited to GPU-oriented processing and can also be exploited on general-purpose multiprocessor architectures. An example is DNA sequence pattern matching, where the comparison between a sequence and a searched pattern can be represented using a matrix whose dimensions depend on the lengths of the compared sequences. In particular, dependencies between elements along diagonals of this matrix can expose opportunities for parallel execution. Such computations exhibit data parallelism that can be exploited at multiple levels: across cores through multithreading and within individual cores through Single Instruction, Multiple Data (SIMD) execution. In superscalar architectures, additional parallelism can also be obtained by executing independent instructions concurrently within each core.

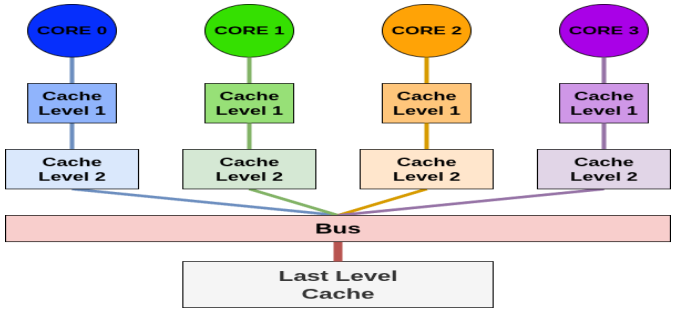


Figure 6.18 Diagram of Multi-Core Architecture

It is preferable if an application's computation can be partitioned into many independent coarse-grained tasks, as synchronization across multiple cores incurs

significant costs. Consequently, well-structured multi-threaded programs prioritize data locality and minimize synchronization needs. This approach is particularly pertinent in multi-core architectures that depend on cache-coherence protocols, such as snooping-based systems, as well as in architectures employing directory-based coherence protocols. Minimizing shared data and synchronization overhead helps limit coherence traffic and improves overall scalability and performance [1, 57].

As previously stated, loops constitute one of the primary segments of code that are amenable to straightforward parallelization, given that their kernels frequently involve array-based computations. Furthermore, applying a Fast Fourier Transform (FFT) to a two-dimensional array enables row-by-row computations to be performed independently, with partial results synchronized at the end using barrier synchronization. FFT algorithms are notably appropriate for data parallelism, as they involve applying an identical sequence of operations to multiple data elements, many of which involve independent intermediate calculations, thereby enabling efficient distribution across processing cores [1].

However, even in multi-threaded solutions for general multi-core microprocessors, there is an effort to minimize synchronization between cores due to the significant performance cost associated with it. Multi-threaded approaches are widely used in operating systems, concurrent programming languages, and game engines, where the microprocessor executes multiple threads across multiple cores.

In game engines, different threads handle physics, rendering, and AI computations independently while synchronizing at specific points. Similarly, the kernel in operating systems such as Linux schedules multiple threads for I/O operations, process management, and interrupt handling concurrently, using minimal synchronization. This is done by partitioning data, which allows multiple threads to work on their own private copies of the data and minimizes interference from shared data between multiple threads.

It is also important for a multi-core compiler to divide data across multiple cache lines, as shown in Fig.6.19. The compiler should organize data so that different cores access separate memory regions, thereby minimizing false sharing. False sharing occurs when multiple cores access distinct variables that reside on the same cache line, which causes unnecessary cache coherence traffic and slows computation. By placing frequently accessed data for each core in separate memory regions, each assigned to a different cache line, the compiler helps ensure that cores can operate on their data concurrently without contention [1].

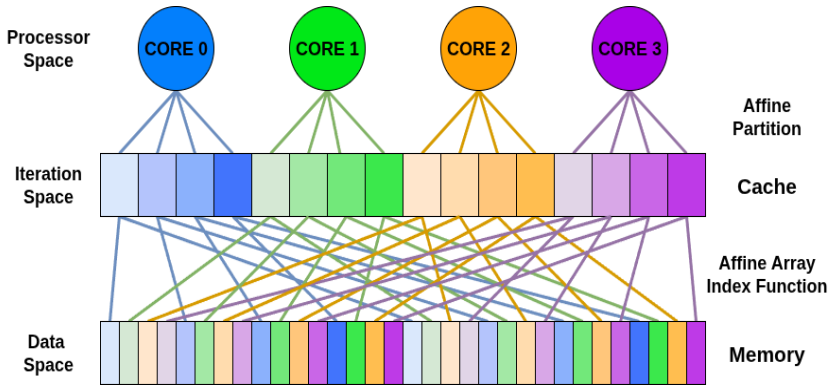


Figure 6.19 Diagram of Divided Data Across Cache Lines

One of the methods used to address parallelization and data-locality problems is Fourier–Motzkin Elimination. It is a technique for eliminating variables from systems of linear inequalities and can be applied in the context of Integer Linear Programming (ILP) to analyze constraints arising from data dependencies. By systematically eliminating variables and inequalities, the method can derive constraints that describe feasible regions for data placement and computation. These constraints can then be used to partition data into separate regions while preserving the required dependencies between operations, potentially reducing conflicts and improving the efficiency of parallel execution [1].

6.5.2 Integer Linear Programming

ILP is widely used to analyze data dependencies in multi-threaded programs. Numerous algorithms and techniques exist for solving systems of linear equations and inequalities that arise naturally when modeling dependence constraints. These concepts are closely related to Diophantine equations, which address linear algebraic equations with integer solutions subject to equality and inequality constraints [1].

In compiler optimization, linear algebra–based methods, including Fourier–Motzkin elimination and ILP formulations, are used to analyze data dependencies and enable safe parallelization. Historically, the mathematical foundations for solving linear equations with integer constraints date back well before modern linear algebra. For example, the Indian mathematician Āryabhaṭa developed the Kuttaka method, a systematic technique for solving linear Diophantine equations. This method differed from those of well-known Greek mathematicians because it was written poetically in Sanskrit; those verses are also called sūtras. This was done to aid easier memorization and transmission compared with traditional mathematical formulas [1].

It is important to note that multi-threaded compilers, particularly GPU compilers, can often parallelize programs efficiently because their execution model is based on the SIMD model, as shown in Fig.6.20. This form of parallelization is commonly referred

to as vectorization. In practice, GPUs implement a closely related model known as Single Instruction, Multiple Threads (SIMT), in which multiple lightweight threads execute the same instruction stream independently on different data elements, allowing limited divergence between threads [1, 57].

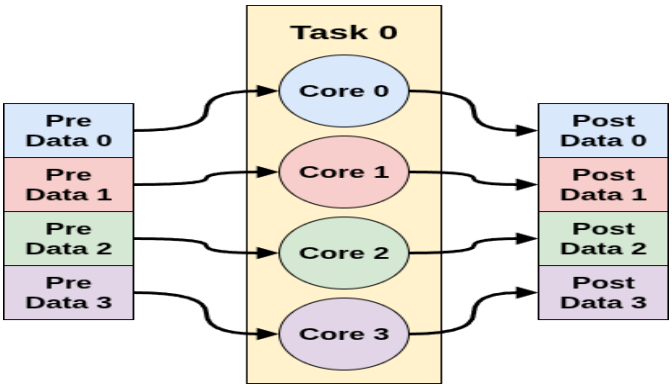


Figure 6.20 Diagram of Vectorization of Data

As mentioned previously, vectorization is especially effective for programs that are heavily loop-based, have regular data access patterns, and require minimal synchronization. In such cases, synchronization is typically limited to barrier synchronization, which is easy to implement and is often automatically managed by a multi-threaded compiler [1, 57].

6.6 Evaluation Criteria

The literature overview delineates the fundamental concepts of compiler design, categorizing the discussion into four primary components: the front-end, middle-end, back-end, and multi-core. This classification was crucial for establishing the foundational knowledge necessary for the research, as a comprehensive understanding of the overall compilation process and the specific roles of each component is indispensable for grasping my principal contributions to multi-core compiler research.

Furthermore, PLCs are specialized industrial control devices whose software environments provide features that support the development and execution of safety-critical applications. This environment provides a PLC runtime system that resembles a separation kernel by promoting modularity and controlled interaction between program components. It also supports explicit data flow through clearly defined variables and I/O interfaces. Additionally, program execution occurs in a deterministic and predictable manner [23].

This thesis evaluates an approach to automatic multi-core synchronization using hardware-based semaphores for control logic programs written in PLC programming languages and executed on multi-core PLC devices. The evaluation focuses on the use of a custom LLVM middle-end pass to analyze the intermediate representation

(IR) of PLC programs generated from Structured Text (ST) using the LLVM front-end RusTy and the OpenPLC toolchain, which can translate graphical PLC programming languages such as Ladder Diagram (LD) into ST while maintaining compliance with the IEC 61131-3 standard [78, 79].

The evaluation considers the ability of the proposed approach to identify opportunities for parallel execution and to provide the synchronization required when a control application is divided across multiple execution threads. The implementation of the LLVM pass and the methods used to perform graph analysis, thread decomposition, and hardware-based semaphore synchronization are described in the following sections.

6.7 Summary

Compilers and interpreters translate the source code of high-level languages into assembly or machine code. Furthermore, compilers consist of three components: the front-end, middle-end, and back-end. The front-end scans the source code and segmenting the input stream into meaningful tokens. These tokens are subsequently processed by the parser, which examines the grammatical structure of the programming language in accordance with its formal grammar. To perform semantic analysis, the parser incorporates mechanisms for passing values and variables necessary for the construction of ASTs.

The ASTs are then used to generate IR code, which is organized into basic blocks. A CFG is constructed from these basic blocks by identifying the possible control-flow relationships between them, as determined by branch, jump, and other control-flow instructions. DDGs are derived from the IR code by identifying dependencies between operations and the values they produce or consume. The middle-end employs the generated graphs and IR code to execute a series of passes that analyze and transform the IR to optimize it for a specific machine architecture. The back-end represents the final stage of a compiler, responsible for generating assembly and machine code while performing machine-dependent analysis and optimizations.

Multi-threaded compilers promote data locality to minimize the need for multi-core synchronization. One common approach is to use ILP, which encompasses numerous algorithms and techniques for solving systems of linear equations and inequalities.

Furthermore, the IEC 61131-3 standard defines LD, a programming language that is inherently Boolean-oriented. This characteristic allows the combinational logic represented in LD to be preserved during LD-to-ST translation, after which LLVM IR can be generated to analyze dependency information and exploit parallelism in a manner similar to HDL synthesis.

Dependency-Aware LLVM Parallelization Pass of Combinational Ladder Diagram

7.1 Hardware Semaphore Operations

Based on the theoretical analysis, a custom middle-end pass was developed and integrated into the LLVM framework to generate multicontext programs adapted for execution on multicore processing units using the synchronization system developed in this work. To understand the dependency-aware model for automatic parallelization, it is first necessary to consider the underlying microprocessor architecture and the organization of instruction execution within the processor pipeline [37, 40].

A microprocessor is composed of multiple stages that together form a pipeline. The number of stages is determined by the specific architectural design. These stages are separated by dedicated pipeline registers, and instructions pass through each stage during successive clock cycles until they complete execution. This pipelined microprocessor design enables the concurrent processing of multiple instructions by overlapping their execution stages [37, 40].

Furthermore, every microprocessor incorporates a memory access stage responsible for accessing data in memory, GPIOs, or peripherals mapped to specific addresses. The shared-semaphored memory's addresses are mapped this way, as shown in Fig.7.1.

The stall signal is connected to the hazard unit together with the other control signals of the memory stage responsible for stalling the pipeline until the required data is loaded. Based on this information, the stall mechanism maintains the stall condition until the required operation can be completed. During multicore synchronization, the stall signal for a store operation remains asserted until the hardware semaphore is empty and can accept the data. Conversely, for a load operation, it remains asserted until the semaphore counter is greater than zero, indicating that data is available. In this way, the hardware-based semaphore mechanism coordinates concurrent execution [37, 40].

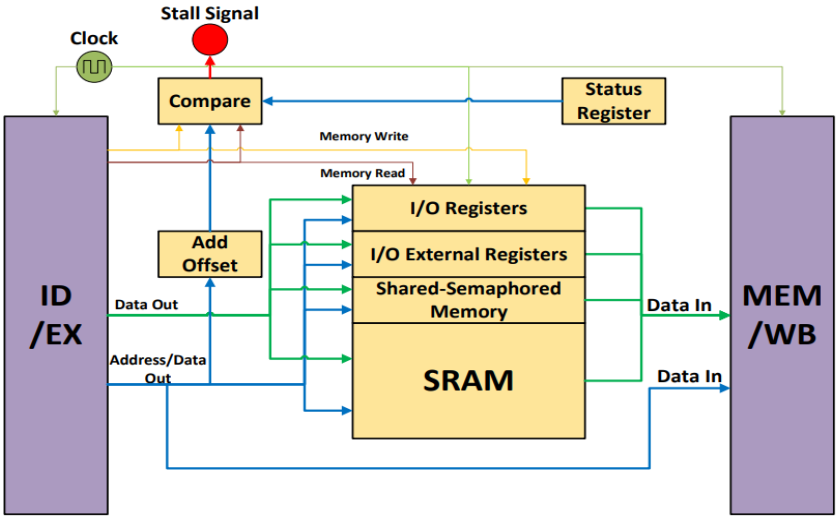


Figure 7.1 Diagram of Memory/WriteBack Pipeline Stage [40]

The interface to the hardware-based semaphores is exposed through dedicated memory addresses assigned to the semaphore memory. These addresses are mapped to a specific region of the memory address space directly associated with the hardware-based semaphores and bypass the cache memory, allowing the cores to access the shared semaphores using conventional load and store instructions, as presented in the following tables: **Semaphore Read Operation** and **Semaphore Write Operation**. This functionality is achieved by appropriately configuring the memory-region decoder, without requiring modifications to the memory stage or the introduction of specialized instructions [37, 40].

Listing 1. Semaphore Read Operation

```
1: li $t1, SemaphoreOffSetAddr
2: lw $t2, SemaphoreNum($t1)
```

Listing 2. Semaphore Write Operation

```
1: li $t1, SemaphoreOffSetAddr
2: sw $t2, SemaphoreNum($t1)
```

7.2 LLVM Intermediate Representation

It is noteworthy that two open-source projects were used to translate PLC programs into LLVM IR. The first project, OpenPLC, was selected for its ability to translate Ladder Diagram (LD) programs into Structured Text (ST) while preserving their combinational

logic. This translation enables the use of the second project, RusTy, an LLVM front-end for ST that supports the IEC 61131-3 standard and translates ST programs into LLVM IR [37, 40].

To transform a PLC program into a multithreaded program and prepare it for the proposed dependency-aware parallelization approach, a sequence of predefined LLVM optimization passes within the middle-end was applied. These passes perform various transformations to simplify and optimize the intermediate representation (IR) before dependency analysis and parallelization. The optimization passes applied include:

- Global Value Numbering (GVN);
- Scalar Replacement of Aggregates (SROA);
- Memory-to-Register Promotion (Mem2Reg);
- Always-Inline;
- Sparse Conditional Constant Propagation (SCCP);
- Dead Code Elimination;
- Control-Flow Graph (CFG) Simplification;
- Reassociation;
- Instruction Combining;
- Loop Canonicalization;
- Loop Unrolling.

The GVN pass assigns unique values to virtual registers used in computations in the LLVM Intermediate Representation (IR), helping identify and eliminate fully and partially redundant computations.

Meanwhile, SROA disassembles *alloca* instructions, which represent stack memory allocations for local variables, when they are related to aggregate types such as structures or arrays, into discrete *alloca* instructions. Subsequently, Mem2Reg enhances register-to-register computation by eliminating unnecessary stack memory accesses associated with *alloca* instructions in the LLVM IR.

The Always-Inline pass is a specialized inlining process that consistently inlines all callee functions. This process holds particular significance as it inlines functions invoked within function blocks. The SCCP pass analyzes the LLVM IR and tries to identify constant values, constant conditional branches, and unreachable basic blocks when they are conclusively proven to be unreachable. This is of considerable importance for the following sequential phase, as it simplifies control flow by propagating constant values and facilitates the elimination of dead instructions and unreachable code.

The dead code elimination and CFG simplification processes encompass multiple passes that eliminate superfluous instructions, unused variables, tail calls, and unreachable code. Additionally, the simplification pass removes basic blocks with no

predecessors and unnecessary PHI nodes in basic blocks with a single predecessor and merges basic blocks with their predecessors when they contain only an unconditional branch

The final optimization passes reassociate commutative expressions and combine instructions to produce fewer and simpler instructions. Loop canonicalization simplifies loop structures and prepares them for subsequent unrolling, which reduces loop-control overhead.

These compilation passes enable the use of the custom LLVM passes, which analyze and transform the generated IR to produce a multithreaded program. The custom pass first constructs a Data Dependency Graph (DDG) from the generated IR code. Subsequently, the DDG is analyzed and traversed to identify dependencies and determine how the program can be mapped onto multiple threads.

As discussed in the literature review, PLC output signals depend on the states of input signals and the values of internal variables, reflecting both combinational and sequential dependencies. In particular, LD is inherently Boolean-oriented, allowing its control logic to be represented as Boolean expressions during the LD-to-ST transformation. Consequently, LD-based PLC programs can be naturally represented as data dependency graphs, similarly to HDL-based synthesis, where input signals propagate through a network of logical and arithmetic operations to determine the output signals [40].

By analyzing the DDG of a PLC program, independent operations can be identified and executed concurrently, enabling multiple outputs to be computed in parallel rather than sequentially. Each IR instruction is represented as a node in the DDG, with dependencies between IR instructions shown by directed edges linking the corresponding nodes. These directed edges represent the unconditional flow of data through the graph, from input nodes to output nodes. Using a graph traversal algorithm, IR instructions can be assigned to multiple dependency paths, each representing a thread in a multithreaded program that can be executed concurrently. The analysis phase therefore examines the data dependency graph and assigns IR instructions to multiple threads according to a designated graph traversal strategy, as illustrated in Fig.7.2.

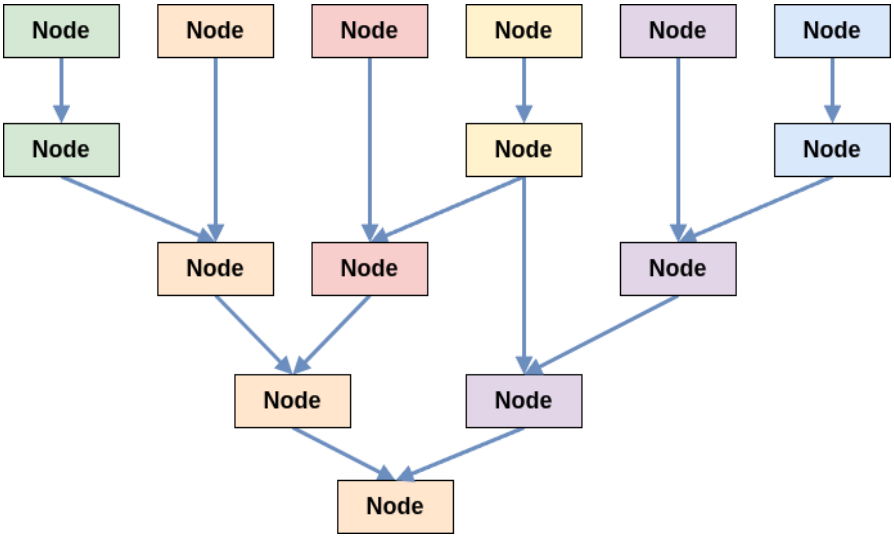


Figure 7.2 DDG with Unconditional Data Flow [40]

PLC control programs typically consist of multiple computational paths that operate independently, with limited data dependencies between them. Such a structure provides opportunities for parallel execution, as computationally independent paths can potentially be assigned to separate threads. Figure 7.3 illustrates this concept by showing a DDG partitioned into two dependency paths, each representing a potential execution thread. The partitioning shown in the figure is intended as a conceptual representation of the desired thread decomposition rather than the result of an arbitrary manual assignment. Potential synchronization points between the threads are considered next, followed by the methodology used to determine the thread partitioning [40].

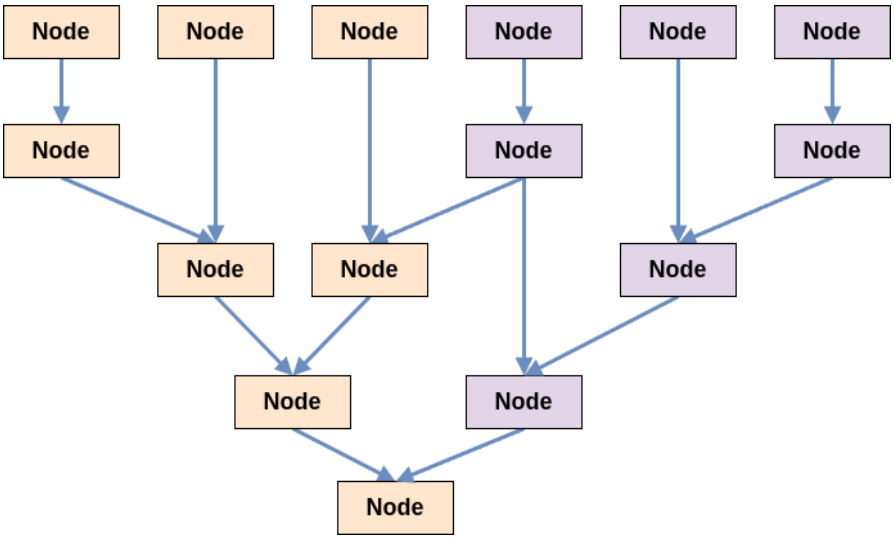


Figure 7.3 Conceptual Thread Partitioning of the DDG [40]

The transformation pass inserts additional load and store IR instructions at potential synchronization points, using hardware-based semaphores to synchronize data dependencies across thread boundaries, as shown in Fig. 7.4.

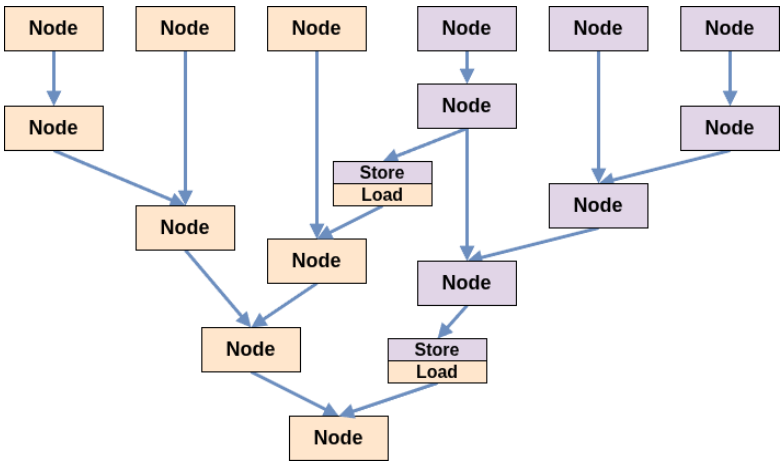


Figure 7.4 Potential Synchronization Points Between Threads [40]

Store operations to hardware-based semaphores are inserted before the incoming edges of shared nodes, while load operations are inserted before the outgoing edges of the shared nodes. Both operations are shown in the following tables using LLVM IR instructions [37, 40].

Listing 3. LLVM IR Semaphore Wire Operation

1: $store\ i32\ \%store_data,\ i32*\ inttoptr\ (i32\ 1610613000\ to\ i32*),\ align\ 4$

Listing 4. LLVM IR Semaphore Read Operation

1: $\%load_data\ =\ load\ i32,\ i32*\ inttoptr\ (i32\ 1610613000\ to\ i32*),\ align\ 4$

Fig.7.5 illustrates the complete compilation workflow, showing the roles of OpenPLC, RusTy, and LLVM in the proposed approach, as well as the dependency-aware parallelization pass implemented in the LLVM middle-end [40].

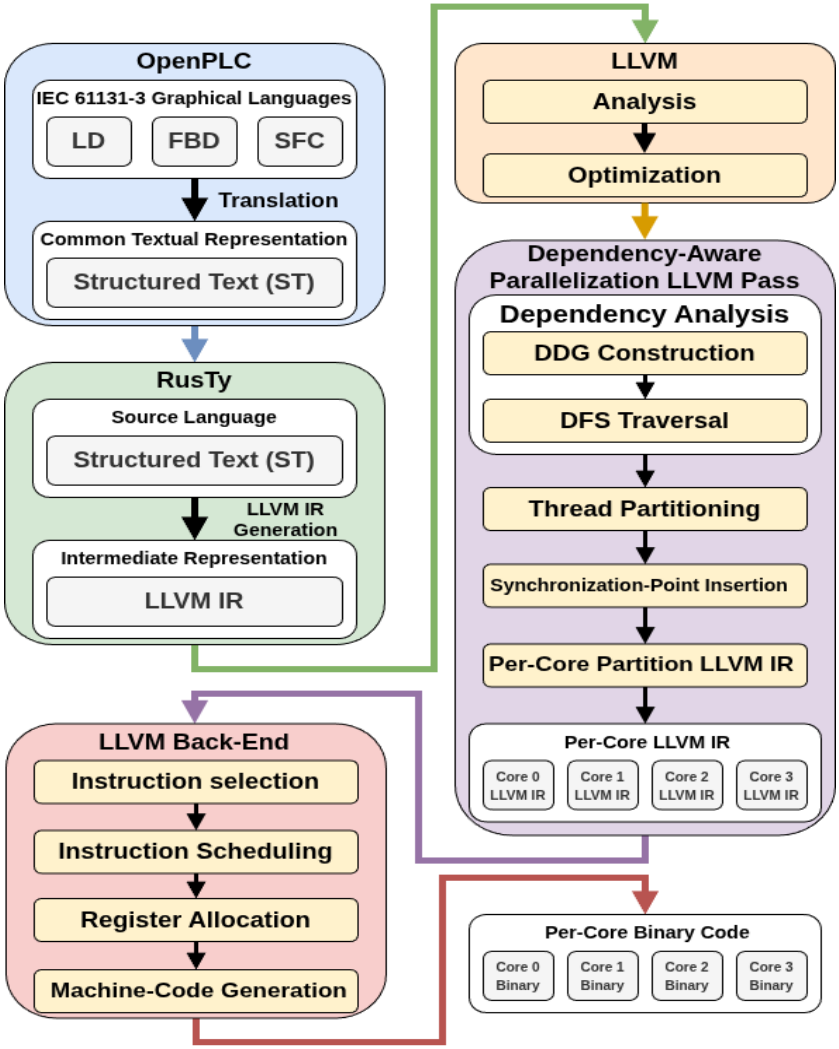


Figure 7.5 Diagram of the Complete Compilation Workflow [40]

7.3 Dependency-Aware Thread Mapping

Initially, a DDG is constructed from a PLC program, as shown in Fig.7.6, where the nodes represent LLVM IR instructions and the edges are indicated by blue arrows.

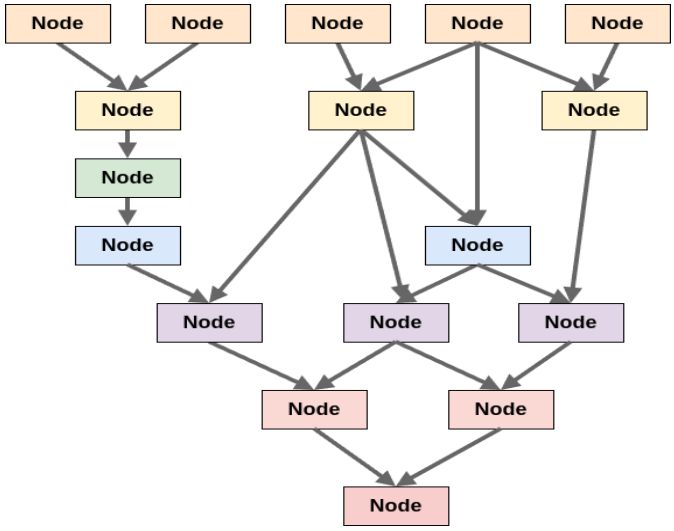


Figure 7.6 Data Dependency Graph [40]

Later, the Depth-First Search (DFS) algorithm is used to traverse the constructed DDG and partition the IR instructions into multiple dependency paths. The algorithm first initializes all nodes as unvisited and then begins the traversal from a selected input node. However, different nodes can be used as starting points, since the dependency paths discovered during separate traversals can be combined when they share common vertices. This is possible because the edges in the DDG are directed in a single direction, representing the flow of dependencies between instructions. During the traversal, the algorithm follows outgoing edges to subsequent nodes, assigns a unique color to the current path, and marks the visited nodes. The traversal ends when it reaches an output node or a node that has already been visited. The colored edges are then removed from the data dependency graph, and a new color is selected for the next DFS traversal. This process is repeated until all nodes have been visited. The complete algorithm is presented in the following tables [40].

Listing 5. Initialization of Nodes

```
1:  procedure INIT( )
2:      for node in G do
3:          node.visited := false
4:      end for
5:      for node.visited = false in G do
6:          DFS(G, node)
7:          for vector.colored = true in G do
8:              DeleteVector(vector)
9:          end for
10:     end for
11: end procedure
```

Listing 6. DFS Algorithm

```
1:  procedure DFS(G, node)
2:      node.visited := true
3:      for nextNode in G.Vector[node] do
4:          ColorVector(G.Vector[node])
5:          if nextNode.visited = false then
6:              DFS(G, nextNode)
7:          else
8:              nextNode.visited := Shared
9:          end if
10:     end for
11: end procedure
```

Fig.7.7 shows the first path traversed, marked by purple arrows.

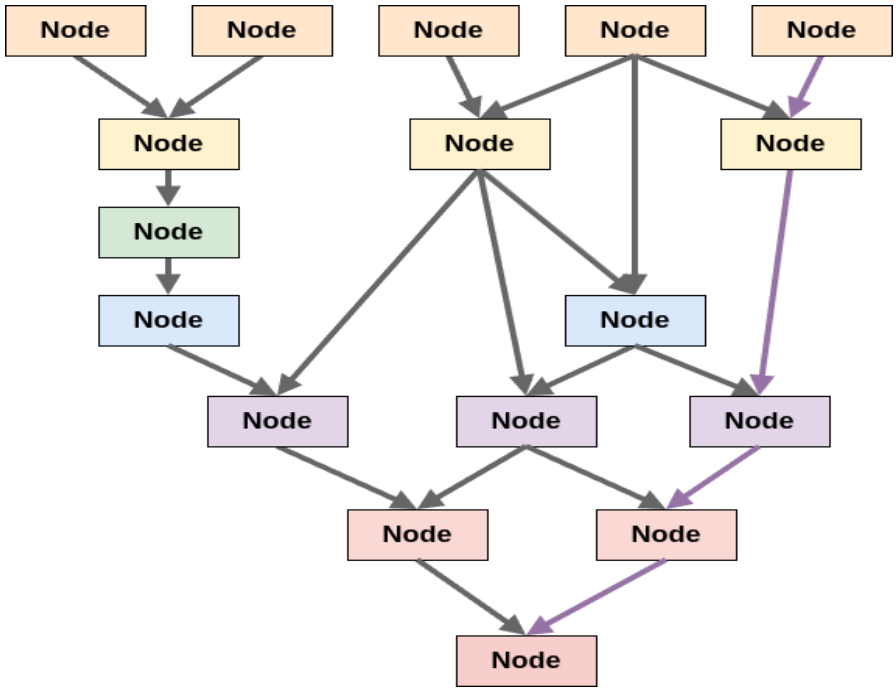
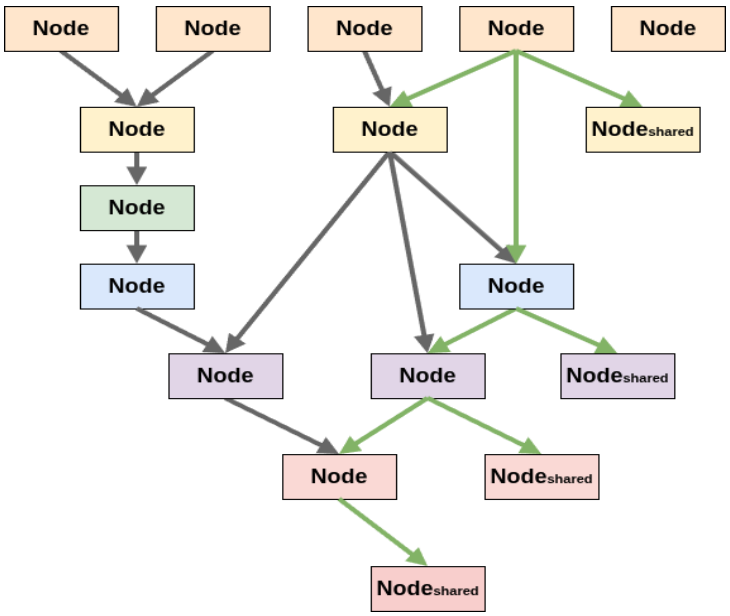


Figure 7.7 Assigning First Color [40]

The following figure shows the first path removed and the second path highlighted in green.


Node
Node
Node

After all paths have been traversed, the shared nodes across multiple paths are identified, as shown in Fig.7.10.

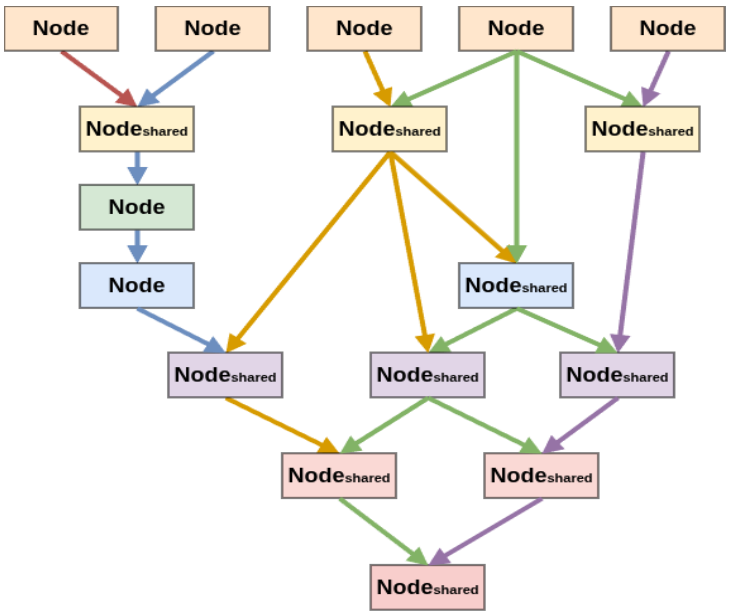


Figure 7.10 All the Paths have an Assigned Color [40]

After all IR instructions have been assigned to multiple paths, the shortest paths are merged with the nearest paths, as shown in the following figures.

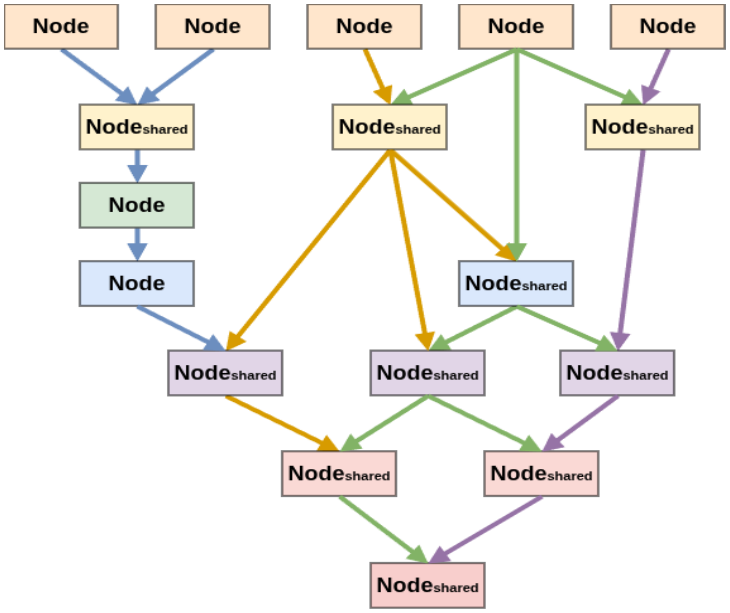


Figure 7.11 Merging First Shortest Paths [40]

The merging process continues until the specified number of paths remains, at which point the synchronization nodes are permanently inserted, as demonstrated in Fig.7.12.

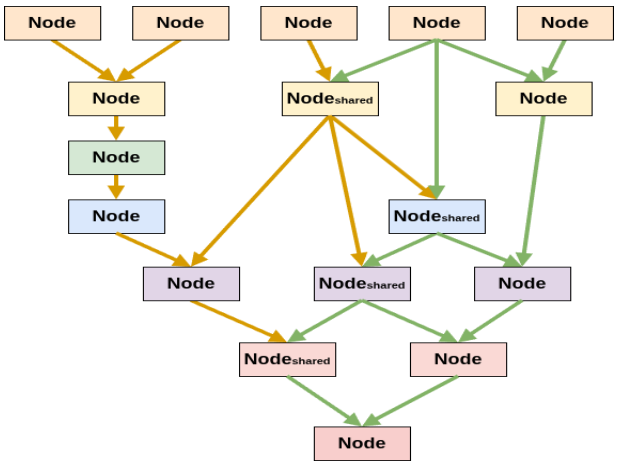


Figure 7.12 Merging Third Shortest Paths [40]

When the result operand of the current node is used as a secondary source operand in the subsequent node, a new shared node is inserted between them, as shown in Fig.7.13.

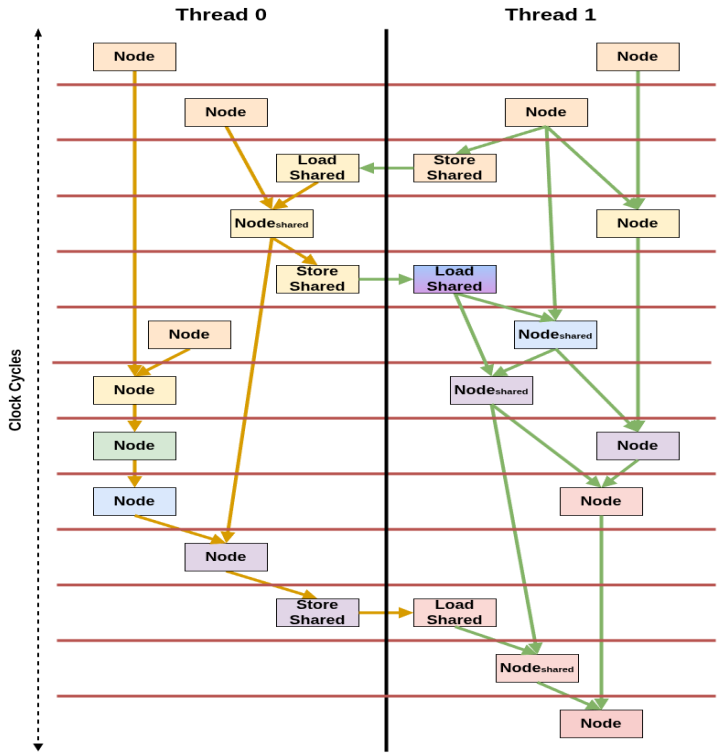


Figure 7.13 Program with completely mapped threads [40]

7.4 DATM Algorithm Evaluation Method

The first stage of the evaluation investigates the feasibility of automatically inserting multicore synchronization points in the middle-end of the compiler. A custom LLVM pass is developed to insert load and store IR instructions that use hardware-based semaphores between co-dependent instructions belonging to different execution paths. This stage serves as an initial proof of concept and provides the basis for the subsequent stages of the evaluation.

The second stage of the evaluation involves implementing the second part of the presented algorithm, which is responsible for automatically dividing PLC programs into multiple well-balanced threads while providing the initial mechanism for inserting synchronization points for data dependencies across thread boundaries. Multiple PLC programs written in Ladder Language and Structured Text are analyzed using configurations with one, two, and four threads.

7.5 Summary

The proposed hardware-based semaphore mechanism is mapped to dedicated memory addresses, enabling conventional load and store instructions to access and utilize the semaphores. This mechanism provides the basis for evaluating the proposed DATM algorithm. The open-source OpenPLC and RusTy projects are used to translate PLC programs into LLVM IR. The generated LLVM IR is subsequently processed by predefined LLVM middle-end optimization passes to simplify and optimize the IR code. The resulting IR is then processed by the proposed algorithm, which constructs and analyzes the DDG, partitions the instructions into multiple dependency paths, and inserts synchronization points between paths assigned to different threads. These synchronization points use the proposed hardware-based semaphore mechanism to coordinate the execution of dependent instructions across thread boundaries.

The evaluation is divided into two stages. The first stage examines the initial part of the LLVM pass, which automatically inserts IR instructions that use hardware-based semaphores at the required synchronization points. The second stage implements the thread-partitioning part of the proposed algorithm, which uses a DFS algorithm to automatically divide PLC programs into multiple dependency paths and assign them to threads for concurrent execution.

Results and Discussion of DATM Algorithm

8.1 Multi-Core Synchronization Point Insertion

The first stage of the evaluation was conducted by implementing the first part of the custom LLVM middle-end pass, which inserts load and store instructions into the LLVM IR at multicore synchronization points. The proposed synchronization mechanism was initially tested using a PLC program consisting of arithmetic and Boolean computations, designed to reproduce the computational patterns typically generated from a Ladder Diagram (LD) program. The PLC program was first translated into LLVM IR using OpenPLC and RusTy. Subsequently, the predefined LLVM optimization passes were applied in phases to optimize and simplify the generated IR code, ultimately reducing the program to a single basic block, thereby enabling the construction and analysis of the DDG from the resulting IR code [37].

The LLVM IR instructions were divided into multiple threads using a graph traversal algorithm in which each input variable serves as the starting point of a path. During each iteration, every path advances by one IR instruction until the paths intersect. At path intersections, only the path that provides the first operand advances. This process continues until all IR instructions have been assigned to a path. After that, the paths are merged, as previously described, until only four paths remain. At the remaining path intersections, load and store LLVM IR instructions were inserted, as shown in Fig.8.1

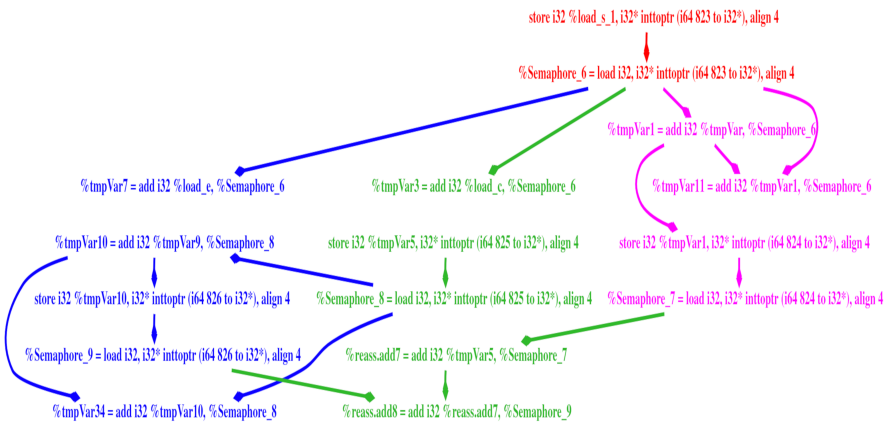


Figure 8.1 Data Dependency Graph of Multi-Threaded Synchronization [37]

The final LLVM IR code was used to generate machine code for the four MIPS cores, which were simulated using a Verilog simulator. An example of the data-exchange synchronization between the cores is shown in the following waveforms.

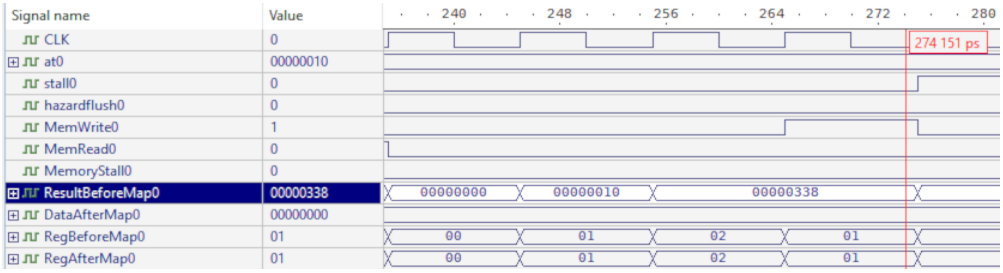


Figure 8.2 Write Access to a Semaphored Memory Cell by Core 0 [37]

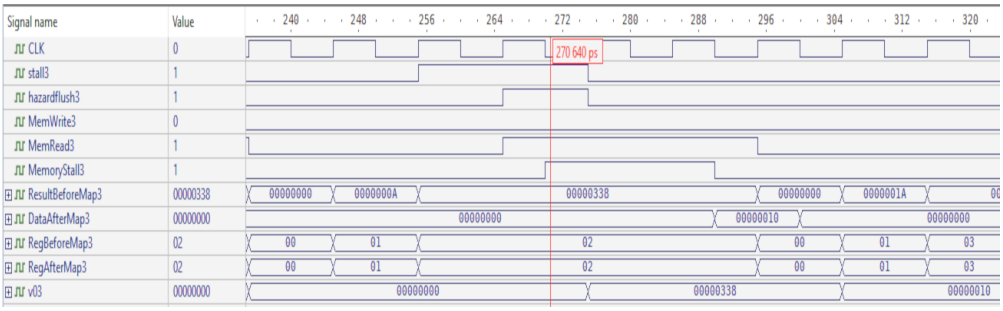


Figure 8.3 Read Access to a Semaphored Memory Cell by Core 3 with Arbitration Delay [37]

The two waveforms illustrate the write and read operations involved in the synchronization between Core 0 and Core 3. The first waveform shows Core 0 writing data to the semaphored memory cell, while the second shows Core 3 reading the corresponding data after the required synchronization condition has been satisfied. Based on the timing of these operations, six clock cycles are required for the data written by Core 0 to become available to Core 3. During this interval, Core 3 is stalled for four clock cycles while waiting for the required synchronization condition. The remaining two clock cycles correspond to the memory access itself, indicating that loading data from a hardware-based semaphore requires two clock cycles, equivalent to a conventional load operation from data memory [37].

The presented results demonstrate the feasibility of inserting LLVM IR load and store instructions that utilize the proposed hardware-based synchronization mechanism at thread boundaries during the compiler middle-end stage. This provides the basis for the second stage of the evaluation, in which the second part of the proposed DATM algorithm is used to divide and distribute the workload of a given PLC program evenly among multiple threads, while employing the synchronization mechanism demonstrated in the first stage to coordinate data dependencies across thread boundaries. This allows the workload distribution and synchronization between threads to be evaluated using a set of PLC programs written in Ladder Language and Structured Text (ST), with configurations of one, two, and four threads [37]

8.2 Thread Partitioning and Workload Distribution

The second phase of the evaluation concentrates on the automatic partitioning of a set of control programs implemented in Structured Text (ST) and LD through a custom-developed LLVM pass. This is subsequently followed by the execution of the resulting multithreaded programs on the developed multicore MIPS processor within a Verilog simulation environment [40].

The set of control programs was randomly generated, with each program generated together with an assigned combination of values for nine structural metrics characterizing different aspects of program structure relevant to parallelization. The metrics were represented by relative scores ranging from low to high rather than by explicit mathematical formulations describing the corresponding program characteristics [40].

The considered metrics are:

- **Serialization pressure**
- **Parallel width**
- **Acyclic DAG structure**
- **Cross-layer coupling**

- **Strong interdependence**
- **Layer separation**
- **Cluster separability**
- **Structural regularity**
- **Symmetry score**

In addition to these metrics, the execution time of each program on a single thread, measured in clock cycles, was used as a reference for evaluating the performance of the multithreaded configurations.

High serialization pressure negatively affects program parallelization, as it signifies the degree to which operations within a given program must be executed sequentially due to data dependencies, including anti-dependencies.

Parallel width represents the number of instructions that can be executed independently and concurrently within a program. An increased parallel width indicates the presence of a greater number of instructions that can execute simultaneously without violating their data dependencies. Consequently, programs characterized by a high parallel width typically contain a greater number of independent instruction sequences or data-dependency chains that can contribute concurrently to the computation of outputs.

Data-dependency graphs are typically represented as directed acyclic graphs (DAGs), as dependencies generally follow a feed-forward direction from inputs to outputs. In PLC control logic, feedback paths can occur when an output-derived value influences an earlier computation, introducing cycles into the graph and reducing the number of independent execution paths. Therefore, a more strongly acyclic structure generally provides greater potential for parallel execution.

High cross-layer coupling indicates a greater degree of dependency between different layers of computational layers, whereas strong interdependence pertains to closely interconnected operations that frequently depend on one another's outputs. Both characteristics increase the need for more frequent synchronization among threads.

Layer separation indicates the degree to which computational layers are isolated from one another. Higher layer separation means layers can operate more independently, reducing inter-layer synchronization. Cluster separability measures how easily a program can be partitioned into relatively independent clusters of computations.

Structural regularity measures how consistently dependencies are organized; a highly regular structure has predictable dependency relationships, while a low-regularity structure has irregular dependencies. Meanwhile, the symmetry score reflects the balance of dependency patterns.

Representative LD programs evaluated using the proposed multithreading algorithms are presented to illustrate the structural characteristics captured by the considered metrics. Table 8.1 summarizes the degree to which each representative program exhibits these characteristics.

Table 8.1 Structural characteristics of the representative LD programs.

Program	SP	PW	DAG	CLC	SI	LS	CS	SR	SS
LD-1	Low	High	High	Low	Low	High	High	High	High
LD-2	High	Low	Low	High	High	Low	Low	Medium	Medium
LD-3	High	Medium	Medium	High	High	High	High	Medium	Medium

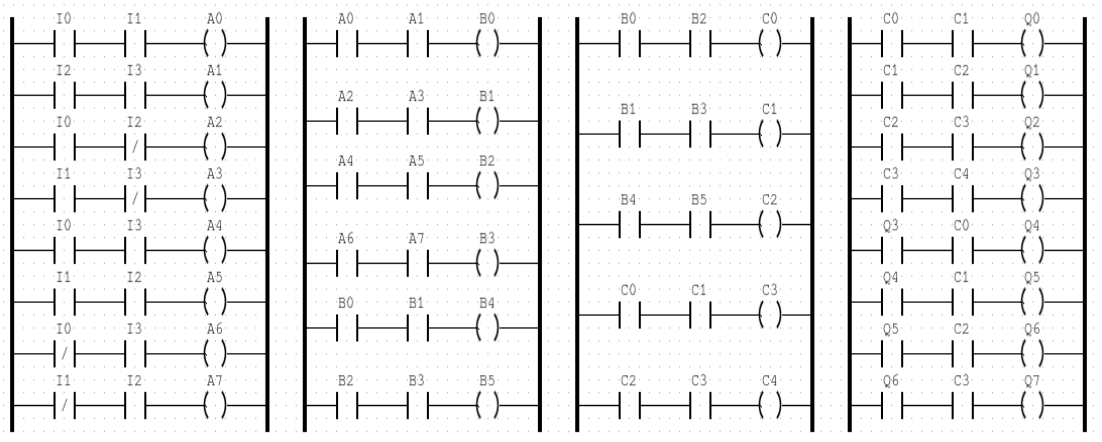


Figure 8.4 Ladder Diagram - 1

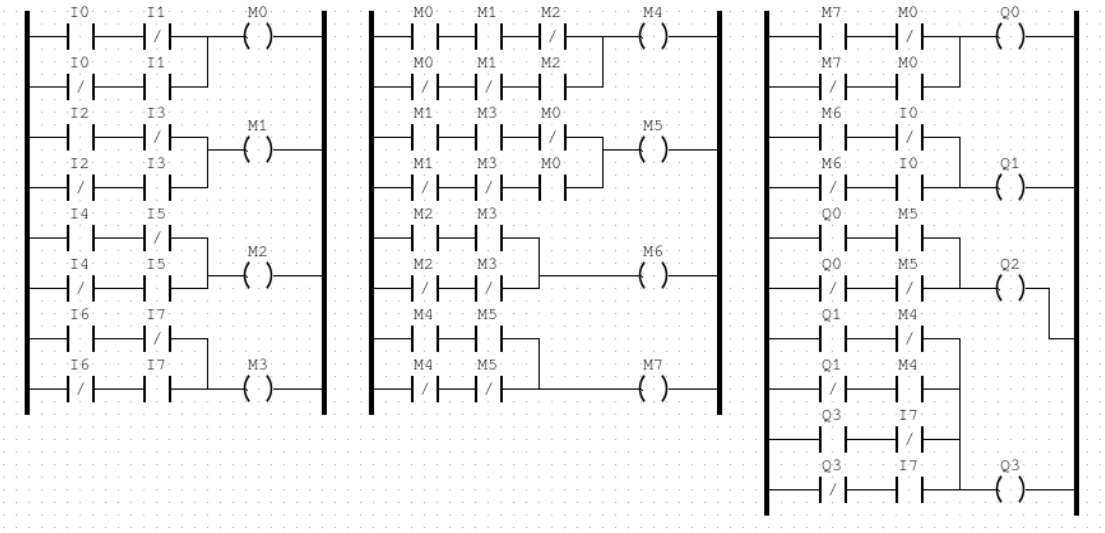


Figure 8.5 Ladder Diagram - 2

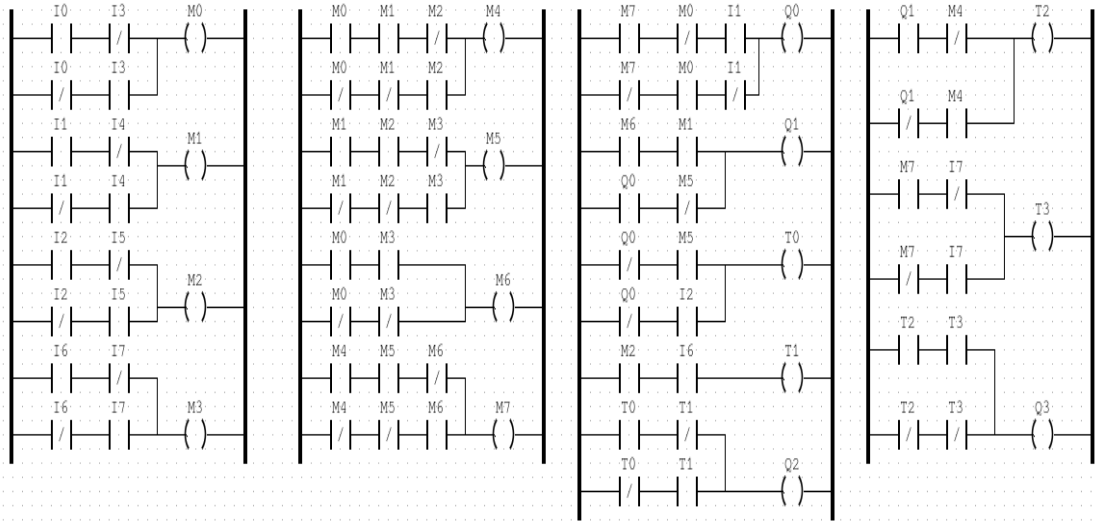
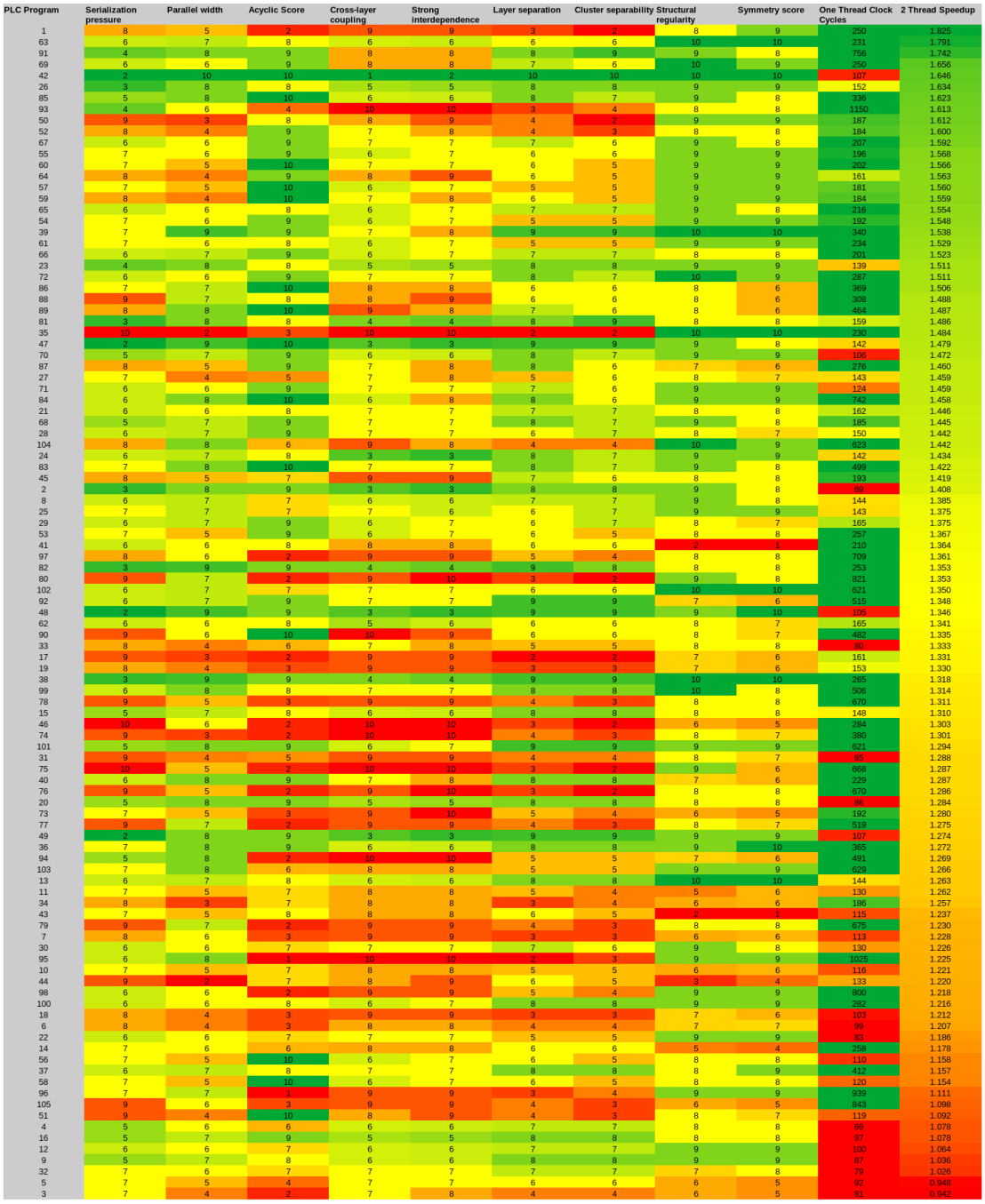


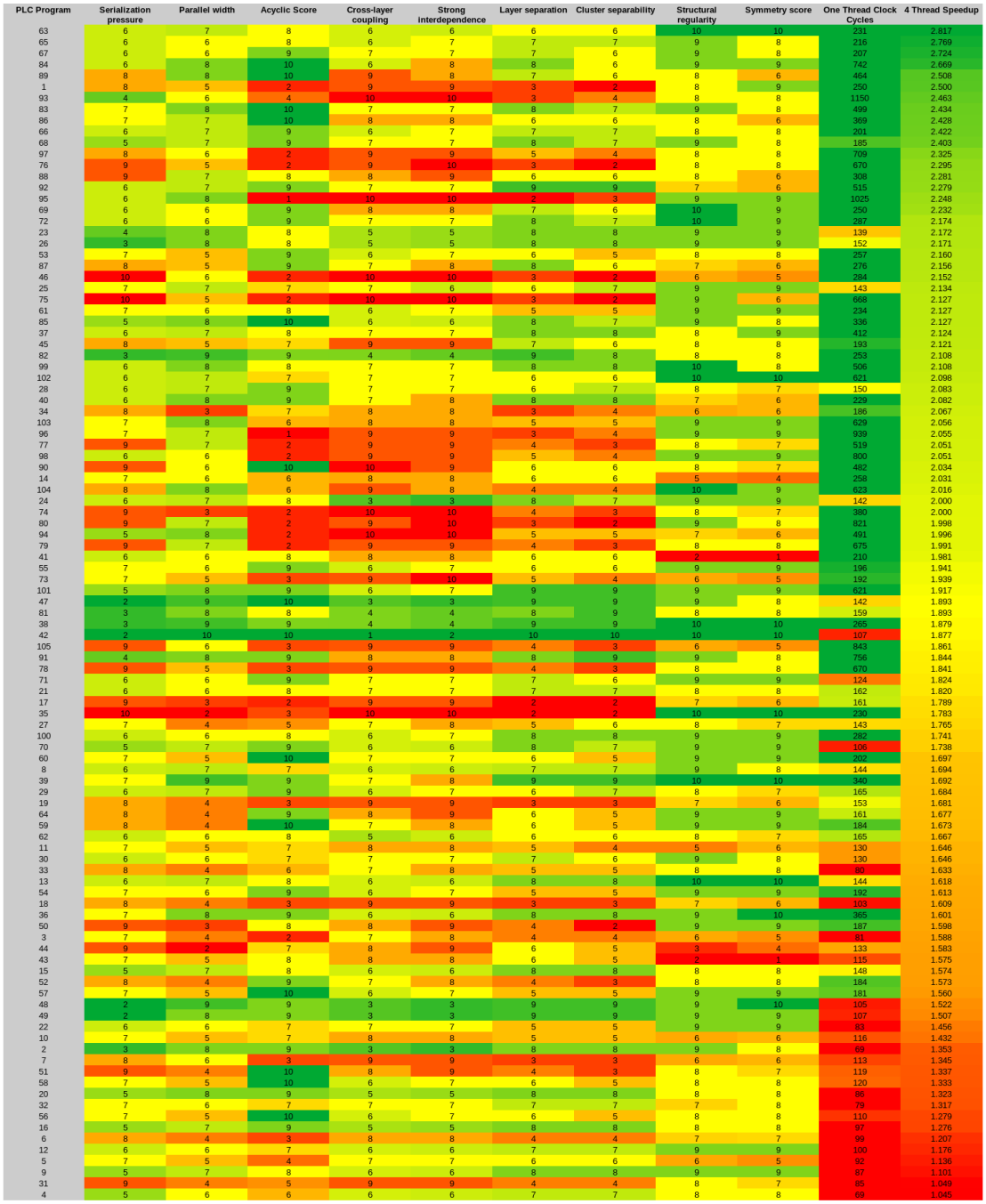
Figure 8.6 Ladder Diagram - 3

Heatmaps were used to show the correlations between the metrics and thread speedup for two and four threads, respectively, including the relative speedup between these two multithreaded configurations.



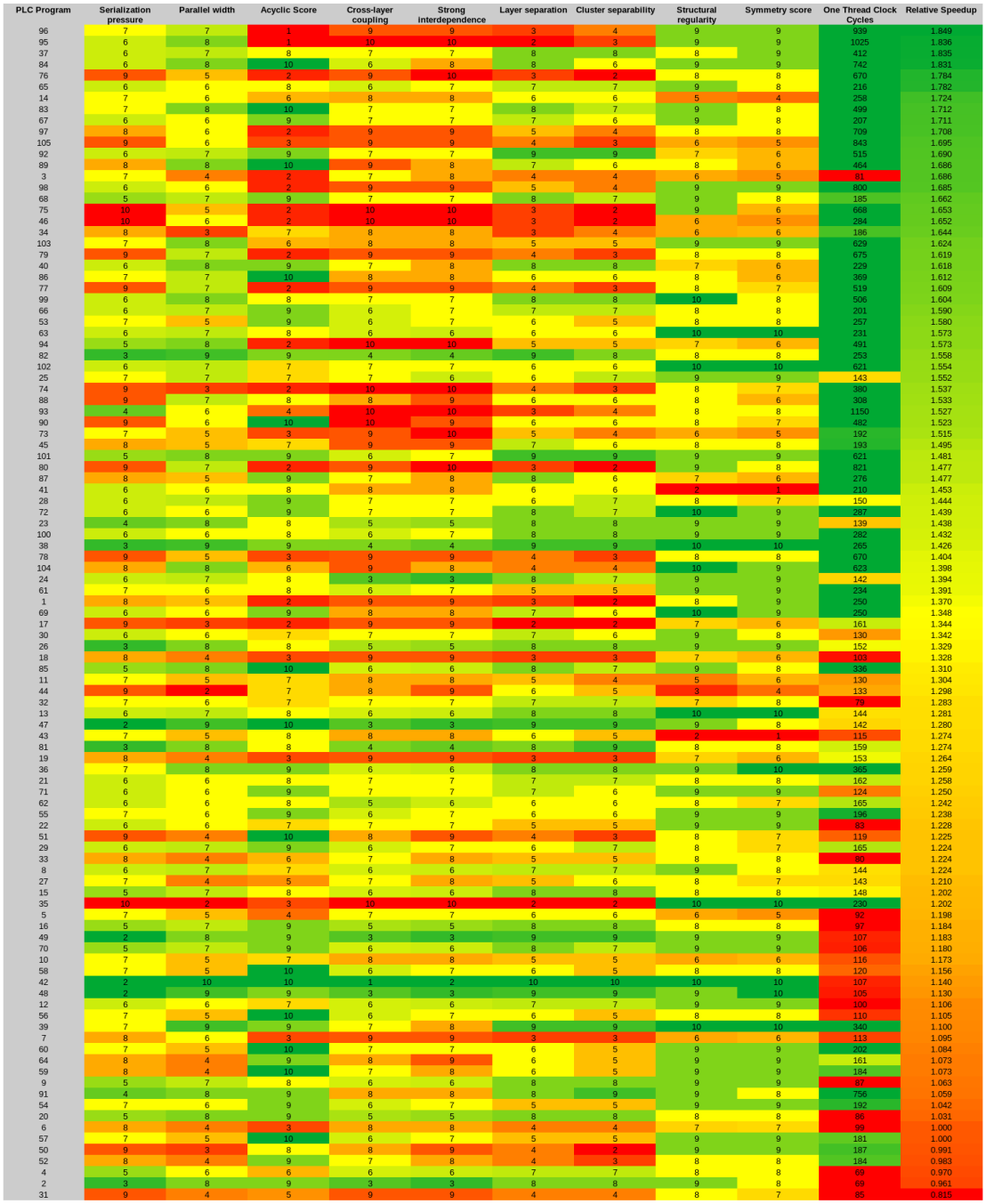
PLC Program	Serialization pressure	Parallel width	Acyclic Score	Cross-layer coupling	Strong interdependence	Layer separation	Cluster separability	Structural regularity	Symmetry score	One Thread Clock Cycles	2 Thread Speedup
1	8	5	2	9	9	3	2	8	9	250	1.825
63	6	7	8	6	6	6	6	10	10	231	1.791
91	4	8	9	8	8	8	9	9	8	756	1.742
69	6	6	9	8	8	7	6	10	9	250	1.656
42	2	10	10	1	2	10	10	10	10	107	1.646
26	3	8	8	5	5	8	8	9	9	152	1.634
85	5	8	10	6	6	8	7	9	8	336	1.623
93	4	6	4	10	10	3	4	8	8	1150	1.613
50	9	3	8	8	9	4	2	9	9	187	1.612
52	8	4	9	7	8	4	3	8	8	184	1.600
67	6	6	9	7	7	7	6	9	8	207	1.592
55	7	6	9	6	7	6	6	9	9	196	1.568
60	7	5	10	7	7	6	5	9	9	202	1.566
64	8	4	9	8	9	6	5	9	9	161	1.563
57	7	5	10	6	7	5	5	9	9	181	1.560
59	8	4	10	7	8	6	5	9	9	184	1.559
65	6	6	8	6	7	7	7	9	8	216	1.554
54	7	6	9	6	7	5	5	9	9	192	1.548
39	7	9	9	7	8	9	9	10	10	340	1.538
61	7	6	8	6	7	5	5	9	9	234	1.529
66	6	7	9	6	7	7	7	8	8	201	1.523
23	4	8	8	5	5	8	8	9	9	139	1.511
72	6	6	9	7	7	8	7	10	9	287	1.511
86	7	7	10	8	8	6	6	8	6	369	1.506
88	9	7	8	8	9	6	6	8	6	308	1.488
89	8	8	10	9	8	7	6	8	6	464	1.487
81	3	8	8	4	4	8	9	8	8	159	1.486
35	10	2	3	10	10	2	2	10	10	230	1.484
47	2	9	10	3	3	9	9	9	8	142	1.479
70	5	7	9	6	6	8	7	9	9	106	1.472
87	8	5	9	7	8	8	6	7	6	276	1.460
27	7	4	5	7	8	5	6	8	7	143	1.459
71	6	6	9	7	7	7	6	9	9	124	1.459
84	6	8	10	6	8	8	6	9	9	742	1.458
21	6	6	8	7	7	7	7	8	8	162	1.446
68	5	7	9	7	7	8	7	9	8	185	1.445
28	6	7	9	7	7	6	7	8	7	150	1.442
104	8	8	6	9	8	4	4	10	9	623	1.442
24	6	7	8	3	3	8	7	9	9	142	1.434
83	7	8	10	7	7	8	7	9	8	499	1.422
45	8	5	7	9	9	7	6	8	8	193	1.419
2	3	8	9	3	3	8	8	9	8	89	1.408
8	6	7	7	6	6	7	7	9	8	144	1.385
25	7	7	7	7	6	6	7	9	9	143	1.375
29	6	7	9	6	7	6	7	8	7	165	1.375
53	7	5	9	6	7	6	5	8	8	257	1.367
41	6	6	8	8	8	6	6	2	1	210	1.364
97	8	6	2	9	9	5	4	8	8	709	1.361
82	3	9	9	4	4	9	8	8	8	253	1.353
80	9	7	2	9	10	3	2	9	8	821	1.353
102	6	7	7	7	7	6	6	10	10	621	1.350
92	6	7	9	7	7	9	9	7	6	515	1.348
48	2	9	9	3	3	9	9	9	10	105	1.346
62	6	6	8	5	6	6	6	8	7	165	1.341
90	9	6	10	10	9	6	6	8	7	482	1.335
33	8	4	6	7	8	5	5	8	8	80	1.333
17	9	3	2	9	9	2	2	7	6	161	1.331
19	8	4	3	9	9	3	3	7	6	153	1.330
38	3	9	9	4	4	9	9	10	10	265	1.318
99	6	8	8	7	7	8	8	10	8	506	1.314
78	9	5	3	9	9	4	3	8	8	670	1.311
15	5	7	8	6	6	8	8	8	8	148	1.310
46	10	6	2	10	10	3	2	6	5	284	1.303
74	9	3	2	10	10	4	3	8	7	380	1.301
101	5	8	9	6	7	9	9	9	9	621	1.294
31	9	4	5	9	9	4	4	8	7	85	1.288
75	10	5	2	10	10	3	2	9	6	668	1.287
40	6	8	9	7	8	8	8	7	6	229	1.287
76	9	5	2	9	10	3	2	8	8	670	1.286
20	5	8	9	5	5	8	8	8	8	86	1.284
73	7	5	3	9	10	5	4	6	5	192	1.280
77	9	7	2	9	9	4	3	8	7	519	1.275
49	2	8	9	3	3	9	9	9	9	107	1.274
36	7	8	9	6	6	8	8	9	10	365	1.272
94	5	8	2	10	10	5	5	7	6	491	1.269
103	7	8	6	8	8	5	5	9	9	629	1.266
13	6	7	8	6	6	8	8	10	10	144	1.263
11	7	5	7	8	8	5	4	5	6	130	1.262
34	8	3	7	8	8	3	4	6	6	186	1.257
43	7	5	8	8	8	6	5	2	1	115	1.237
79	9	7	2	9	9	4	3	8	8	675	1.230
7	8	6	3	9	9	3	3	6	6	113	1.228
30	6	6	7	7	7	7	6	9	8	130	1.226
95	6	8	1	10	10	2	3	9	9	1025	1.225
10	7	5	7	8	8	5	5	6	6	116	1.221
44	9	2	7	8	9	6	5	3	4	133	1.220
98	6	6	2	9	9	5	4	9	9	800	1.218
100	6	6	8	6	7	8	8	9	9	282	1.216
18	8	4	3	9	9	3	3	7	6	103	1.212
6	8	4	3	8	8	4	4	7	7	99	1.207
22	6	6	7	7	7	5	5	9	9	83	1.186
14	7	6	6	8	8	6	6	5	4	268	1.178
56	7	5	10	6	7	6	5	8	8	110	1.158
37	6	7	8	7	7	8	8	8	9	412	1.157
58	7	5	10	6	7	6	5	8	8	120	1.154
96	7	7	1	9	9	3	4	9	9	939	1.111
105	9	6	3	9	9	4	3	6	5	843	1.098
51	9	4	10	8	9	4	3	8	7	119	1.092
4	5	6	6	6	6	7	7	8	8	69	1.078
16	5	7	9	5	5	8	8	8	8	97	1.078
12	6	6	7	6	6	7	7	9	9	100	1.064
9	5	7	8	6	6	8	8	9	9	87	1.036
32	7	6	7	7	7	7	7	7	8	79	1.026
5	7	5	4	7	7	6	6	6	5	92	0.948
3	7	4	2	7	8	4	4	6	5	81	0.942

Figure 8.7 Heatmap for 2 Threads Speedup



PLC Program	Serialization pressure	Parallel width	Acyclic Score	Cross-layer coupling	Strong interdependence	Layer separation	Cluster separability	Structural regularity	Symmetry score	One Thread Clock Cycles	4 Thread Speedup
63	6	7	8	6	6	6	6	10	10	231	2.817
65	6	6	8	6	7	7	7	9	8	216	2.769
67	6	6	9	7	7	7	6	9	8	207	2.724
84	6	8	10	6	8	8	6	9	9	742	2.669
89	8	8	10	9	8	7	6	8	6	464	2.508
1	8	5	2	9	9	3	2	8	9	250	2.500
93	4	6	4	10	10	3	4	8	8	1150	2.463
83	7	8	10	7	7	8	7	9	8	499	2.434
86	7	7	10	8	8	6	6	8	6	369	2.428
66	6	7	9	6	7	7	7	8	8	201	2.422
68	5	7	9	7	7	8	7	9	8	185	2.403
97	8	6	2	9	9	5	4	8	8	709	2.325
76	9	5	2	9	10	3	2	8	8	670	2.295
88	9	7	8	8	9	6	6	8	6	308	2.281
92	6	7	9	7	7	9	9	7	6	515	2.279
95	6	8	1	10	10	2	3	9	9	1025	2.248
69	6	6	9	8	8	7	6	10	9	250	2.232
72	6	6	9	7	7	8	7	10	9	287	2.174
23	4	8	8	5	5	8	8	9	9	139	2.172
26	3	8	8	5	5	8	8	9	9	152	2.171
53	7	5	9	6	7	6	5	8	8	267	2.160
87	8	5	9	7	8	8	6	7	6	276	2.156
46	10	6	2	10	10	3	2	6	5	284	2.152
25	7	7	7	7	6	6	7	9	9	143	2.134
75	10	5	2	10	10	3	2	9	6	668	2.127
61	7	6	8	6	7	5	5	9	9	234	2.127
85	5	8	10	6	6	8	7	9	8	336	2.127
37	6	7	8	7	7	8	8	8	9	412	2.124
45	8	5	7	9	9	7	6	8	8	193	2.121
82	3	9	9	4	4	9	8	8	8	253	2.108
99	6	8	8	7	7	8	8	10	8	506	2.108
102	6	7	7	7	7	6	6	10	10	621	2.098
28	6	7	9	7	7	6	7	8	7	150	2.083
40	6	8	9	7	8	8	8	7	6	229	2.082
34	8	3	7	8	8	3	4	6	6	186	2.067
103	7	8	6	8	8	5	5	9	9	629	2.056
96	7	7	1	9	9	3	4	9	9	939	2.055
77	9	7	2	9	9	4	3	8	7	519	2.051
98	6	6	2	9	9	5	4	9	9	800	2.051
90	9	6	10	10	9	6	6	8	7	482	2.034
14	7	6	6	8	8	6	6	5	4	258	2.031
104	8	8	6	9	8	4	4	10	9	623	2.016
24	6	7	8	3	3	8	7	9	9	142	2.000
74	9	3	2	10	10	4	3	8	7	380	2.000
80	9	7	2	9	10	3	2	9	8	821	1.998
94	5	8	2	10	10	5	5	7	6	491	1.996
79	9	7	2	9	9	4	3	8	8	675	1.991
41	6	6	8	8	8	6	6	2	1	210	1.981
55	7	6	9	6	7	6	6	9	9	196	1.941
73	7	5	3	9	10	5	4	6	5	192	1.939
101	5	8	9	6	7	9	9	9	9	621	1.917
47	2	9	10	3	3	9	9	9	8	142	1.893
81	3	8	8	4	4	8	9	8	8	159	1.893
38	3	9	9	4	4	9	9	10	10	265	1.879
42	2	10	10	1	2	10	10	10	10	107	1.877
105	9	6	3	9	9	4	3	6	5	843	1.861
91	4	8	9	8	8	8	9	9	8	756	1.844
78	9	5	3	9	9	4	3	8	8	670	1.841
71	6	6	9	7	7	7	6	9	9	124	1.824
21	6	6	8	7	7	7	7	8	8	162	1.820
17	9	3	2	9	9	2	2	7	6	161	1.789
35	10	2	3	10	10	2	2	10	10	230	1.783
27	7	4	5	7	8	5	6	8	7	143	1.765
100	6	6	8	6	7	8	8	9	9	282	1.741
70	5	7	9	6	6	8	7	9	9	106	1.738
60	7	5	10	7	7	6	5	9	9	202	1.697
8	6	7	7	6	6	7	7	9	8	144	1.694
39	7	9	9	7	8	9	9	10	10	340	1.692
29	6	7	9	6	7	6	7	8	7	165	1.684
19	8	4	3	9	9	3	3	7	6	153	1.681
64	8	4	9	8	9	6	5	9	9	161	1.677
59	8	4	10	7	8	6	5	9	9	184	1.673
62	6	6	8	5	6	6	6	8	7	165	1.667
11	7	5	7	8	8	5	4	5	6	130	1.646
30	6	6	7	7	7	7	6	9	8	130	1.646
33	8	4	6	7	8	5	5	8	8	80	1.633
13	6	7	8	6	6	8	8	10	10	144	1.618
54	7	6	9	6	7	5	5	9	9	192	1.613
18	8	4	3	9	9	3	3	7	6	103	1.609
36	7	8	9	6	6	8	8	9	10	365	1.601
50	9	3	8	8	9	4	2	9	9	187	1.598
3	7	4	2	7	8	4	4	6	5	81	1.588
44	9	2	7	8	9	6	5	3	4	133	1.583
43	7	5	8	8	8	6	5	2	1	115	1.575
15	5	7	8	6	6	8	8	8	8	148	1.574
52	8	4	9	7	8	4	3	8	8	184	1.573
57	7	5	10	6	7	5	5	9	9	181	1.560
48	2	9	9	3	3	9	9	9	10	105	1.522
49	2	8	9	3	3	9	9	9	9	107	1.507
22	6	6	7	7	7	5	5	9	9	83	1.456
10	7	5	7	8	8	5	5	6	6	116	1.432
2	3	8	9	3	3	8	8	9	8	69	1.353
7	8	6	3	9	9	3	3	6	6	113	1.345
51	9	4	10	8	9	4	3	8	7	119	1.337
58	7	5	10	6	7	6	5	8	8	120	1.333
20	5	8	9	5	5	8	8	8	8	86	1.323
32	7	6	7	7	7	7	7	7	8	79	1.317
56	7	5	10	6	7	6	5	8	8	110	1.279
16	5	7	9	5	5	8	8	8	8	97	1.276
6	8	4	3	8	8	4	4	7	7	99	1.207
12	6	6	7	6	6	7	7	9	9	100	1.176
5	7	5	4	7	7	6	6	6	5	92	1.136
9	5	7	8	6	6	8	8	9	9	87	1.101
31	9	4	5	9	9	4	4	8	7	85	1.049
4	5	6	6	6	6	7	7	8	8	69	1.045

Figure 8.8 Heatmap for 4 Threads Speedup



PLC Program	Serialization pressure	Parallel width	Acyclic Score	Cross-layer coupling	Strong interdependence	Layer separation	Cluster separability	Structural regularity	Symmetry score	One Thread Clock Cycles	Relative Speedup
96	7	7	1	9	9	3	4	9	9	939	1.849
95	6	8	1	10	10	2	3	9	9	1025	1.836
37	6	7	8	7	7	8	8	8	9	412	1.835
84	6	8	10	6	8	8	6	9	9	742	1.831
76	9	5	2	9	10	3	2	8	8	670	1.784
65	6	6	8	6	7	7	7	9	8	216	1.782
14	7	6	6	8	8	6	6	5	4	258	1.724
83	7	8	10	7	7	8	7	9	8	499	1.712
67	6	6	9	7	7	7	6	9	8	207	1.711
97	8	6	2	9	9	5	4	8	8	709	1.708
105	9	6	3	9	9	4	3	6	5	843	1.695
92	6	7	9	7	7	9	9	7	6	515	1.690
89	8	8	10	9	8	7	6	8	6	464	1.686
3	7	4	2	7	8	4	4	6	5	81	1.686
98	6	6	2	9	9	5	4	9	9	800	1.685
68	5	7	9	7	7	8	7	9	8	185	1.662
75	10	5	2	10	10	3	2	9	6	668	1.653
46	10	6	2	10	10	3	2	6	5	284	1.652
34	8	3	7	8	8	3	4	6	6	186	1.644
103	7	8	6	8	8	5	5	9	9	629	1.624
79	9	7	2	9	9	4	3	8	8	675	1.619
40	6	8	9	7	8	8	8	7	6	229	1.618
86	7	7	10	8	8	6	6	8	6	369	1.612
77	9	7	2	9	9	4	3	8	7	519	1.609
99	6	8	8	7	7	8	8	10	8	506	1.604
66	6	7	9	6	7	7	7	8	8	201	1.590
53	7	5	9	6	7	6	5	8	8	257	1.580
63	6	7	8	6	6	6	6	10	10	231	1.573
94	5	8	2	10	10	5	5	7	6	491	1.573
82	3	9	9	4	4	9	8	8	8	253	1.558
102	6	7	7	7	7	6	6	10	10	621	1.554
25	7	7	7	7	6	6	7	9	9	143	1.552
74	9	3	2	10	10	4	3	8	7	380	1.537
88	9	7	8	8	9	6	6	8	6	308	1.533
93	4	6	4	10	10	3	4	8	8	1150	1.527
90	9	6	10	10	9	6	6	8	7	482	1.523
73	7	5	3	9	10	5	4	6	5	192	1.515
45	8	5	7	9	9	7	6	8	8	193	1.495
101	5	8	9	6	7	9	9	9	9	621	1.481
80	9	7	2	9	10	3	2	9	8	821	1.477
87	8	5	9	7	8	8	6	7	6	276	1.477
41	6	6	8	8	8	6	6	2	1	210	1.453
28	6	7	9	7	7	6	7	8	7	150	1.444
72	6	6	9	7	7	8	7	10	9	287	1.439
23	4	8	8	5	5	8	8	9	9	139	1.438
100	6	6	8	6	7	8	8	9	9	282	1.432
38	3	9	9	4	4	9	9	10	10	265	1.426
78	9	5	3	9	9	4	3	8	8	670	1.404
104	8	8	6	9	8	4	4	10	9	623	1.398
24	6	7	8	3	3	8	7	9	9	142	1.394
61	7	6	8	6	7	5	5	9	9	234	1.391
1	8	5	2	9	9	3	2	8	9	250	1.370
69	6	6	9	8	8	7	6	10	9	250	1.348
17	9	3	2	9	9	2	2	7	6	161	1.344
30	6	6	7	7	7	7	6	9	8	130	1.342
26	3	8	8	5	5	8	8	9	9	152	1.329
18	8	4	3	9	9	3	3	7	6	103	1.328
85	5	8	10	6	6	8	7	9	8	336	1.310
11	7	5	7	8	8	5	4	5	6	130	1.304
44	9	2	7	8	9	6	5	3	4	133	1.298
32	7	6	7	7	7	7	7	7	8	79	1.283
13	6	7	8	6	6	8	8	10	10	144	1.281
47	2	9	10	3	3	9	9	9	8	142	1.280
43	7	5	8	8	8	6	5	2	1	115	1.274
81	3	8	8	4	4	8	9	8	8	159	1.274
19	8	4	3	9	9	3	3	7	6	153	1.264
36	7	8	9	6	6	8	8	9	10	365	1.259
21	6	6	8	7	7	7	7	8	8	162	1.258
71	6	6	9	7	7	7	6	9	9	124	1.250
62	6	6	8	5	6	6	6	8	7	165	1.242
55	7	6	9	6	7	6	6	9	9	196	1.238
22	6	6	7	7	7	5	5	9	9	83	1.228
51	9	4	10	8	9	4	3	8	7	119	1.225
29	6	7	9	6	7	6	7	8	7	165	1.224
33	8	4	6	7	8	5	5	8	8	80	1.224
8	6	7	7	6	6	7	7	9	8	144	1.224
27	7	4	5	7	8	5	6	8	7	143	1.210
15	5	7	8	6	6	8	8	8	8	148	1.202
35	10	2	3	10	10	2	2	10	10	230	1.202
5	7	5	4	7	7	6	6	6	5	92	1.198
16	5	7	9	5	5	8	8	8	8	97	1.184
49	2	8	9	3	3	9	9	9	9	107	1.183
70	5	7	9	6	6	8	7	9	9	106	1.180
10	7	5	7	8	8	5	5	6	6	116	1.173
58	7	5	10	6	7	6	5	8	8	120	1.156
42	2	10	10	1	2	10	10	10	10	107	1.140
48	2	9	9	3	3	9	9	9	10	105	1.130
12	6	6	7	6	6	7	7	9	9	100	1.106
56	7	5	10	6	7	6	5	8	8	110	1.105
39	7	9	9	7	8	9	9	10	10	340	1.100
7	8	6	3	9	9	3	3	6	6	113	1.095
60	7	5	10	7	7	6	5	9	9	202	1.084
64	8	4	9	8	9	6	5	9	9	161	1.073
59	8	4	10	7	8	6	5	9	9	184	1.073
9	5	7	8	6	6	8	8	9	9	87	1.063
91	4	8	9	8	8	8	9	9	8	756	1.059
54	7	6	9	6	7	5	5	9	9	192	1.042
20	5	8	9	5	5	8	8	8	8	86	1.031
6	8	4	3	8	8	4	4	7	7	99	1.000
57	7	5	10	6	7	5	5	9	9	181	1.000
50	9	3	8	8	9	4	2	9	9	187	0.991
52	8	4	9	7	8	4	3	8	8	184	0.983
4	5	6	6	6	6	7	7	8	8	69	0.970
2	3	8	9	3	3	8	8	9	8	69	0.961
31	9	4	5	9	9	4	4	8	7	85	0.815

Figure 8.9 Heatmap for Relative Speedup

As observed, the groups exhibiting relatively poor thread efficiency are primarily affected by programs with very short execution times. This is because synchronization overhead constitutes a significant portion of their parallel execution time. Excluding these programs provides a clearer correlation between the structural metrics and thread efficiency.

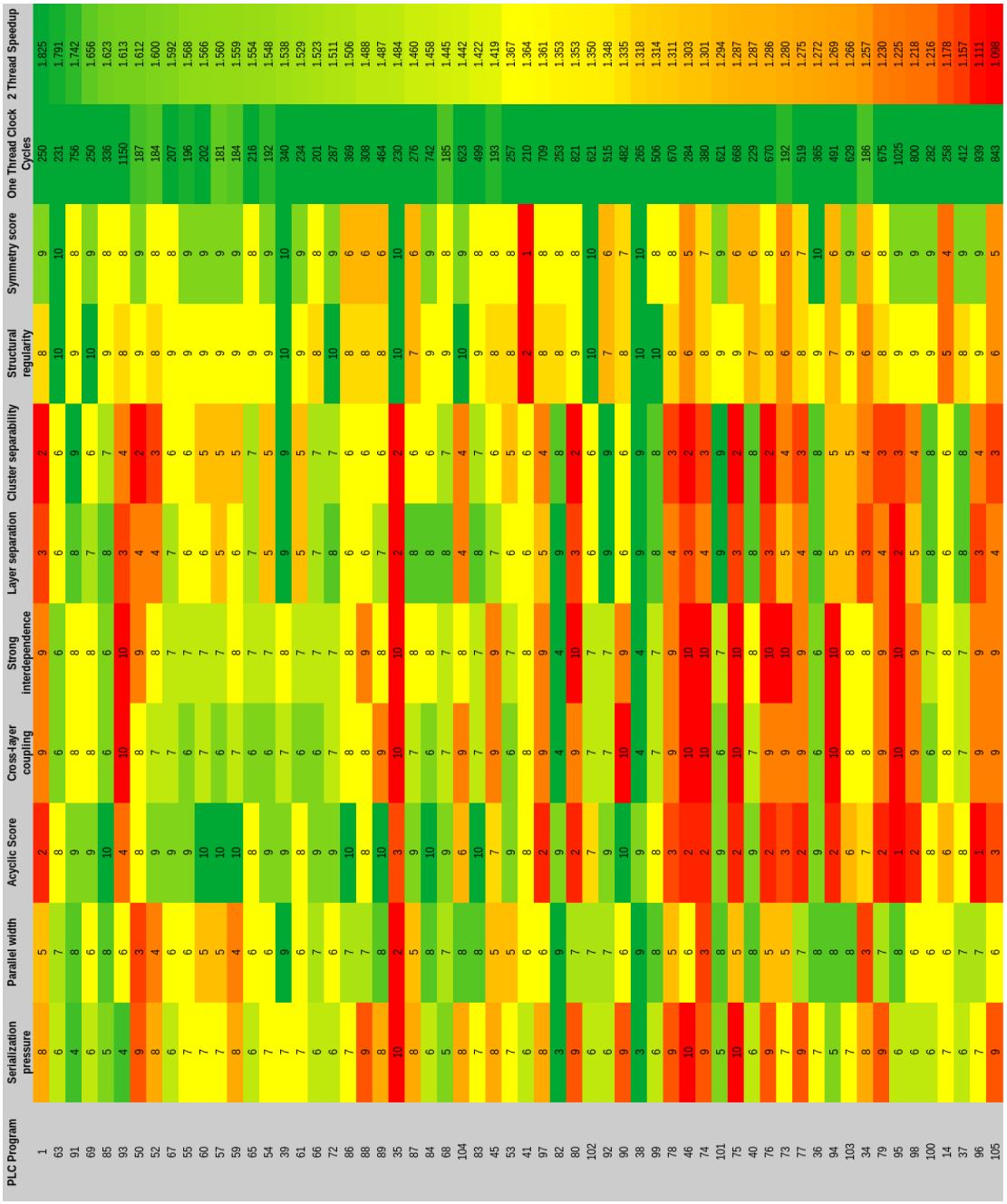


Figure 8.10 Clean Heatmap for 2 Threads Speedup

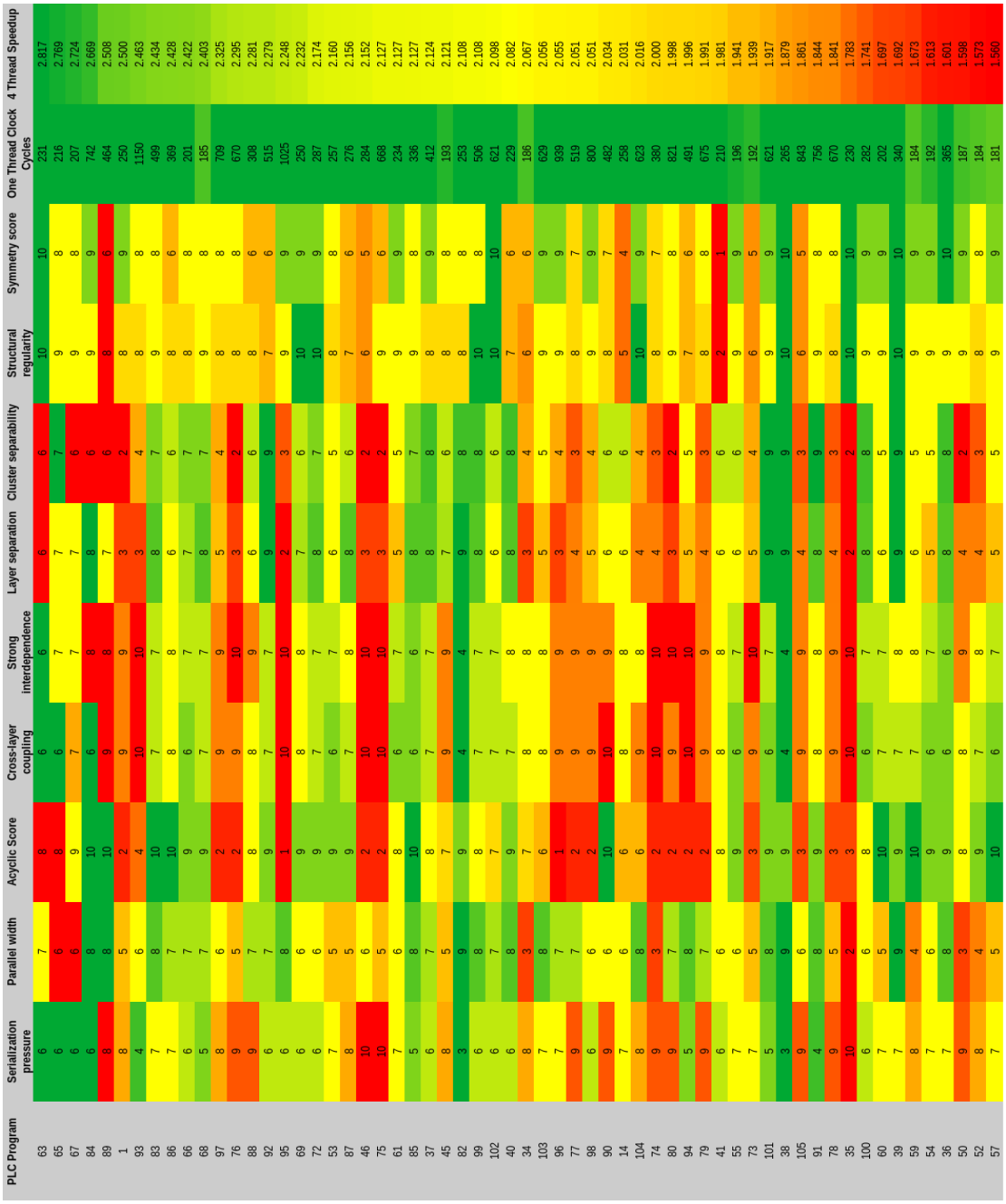


Figure 8.11 Clean Heatmap for 4 Threads Speedup [40]

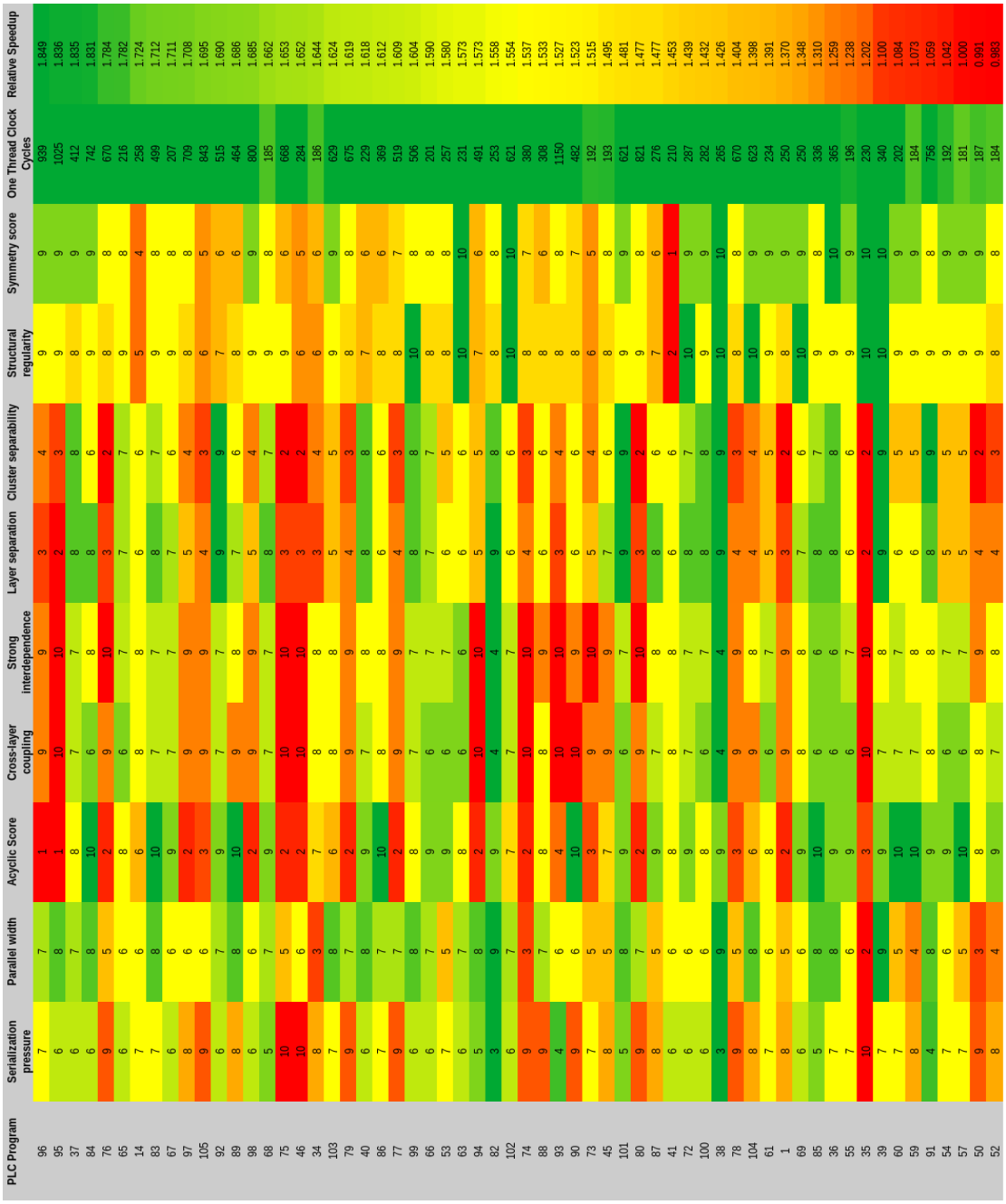


Figure 8.12 Clean Heatmap for Relative Speedup

The observed correlation indicates that these metrics influence initial parallelization and higher-order scaling in different ways. This suggests that achieving parallel execution and efficiently scaling across additional cores involve related yet distinct challenges.

Serialization pressure adversely influences both 2-thread and 4-thread speedups, as this metric indicates the proportion of a program that must be executed sequentially. Nevertheless, the effect is more significant with four threads, since additional cores remain inactive for extended durations while awaiting the completion of serialized operations. As a result, the relative speedup attained when increasing the thread count from 2 to 4 is diminished.

Parallel width exerts a modest influence on 2-thread speedup, as two threads are already capable of capitalizing on a considerable level of available parallelism. Nonetheless, it becomes a predominant factor in 4-thread speedup, since additional threads necessitate an expanded pool of independent tasks to sustain full utilization. Consequently, increased parallel width improves scalability, rendering the transition from two to four threads markedly more advantageous.

A more acyclic data dependency graph enhances two-thread speedup by eliminating circular dependencies that constrain concurrent task execution. Its impact on four-thread speedup is typically less significant because, after cycles are removed, additional scaling relies more on the availability of parallel work than on the graph structure itself. Therefore, more pronounced DAG characteristics facilitate initial parallelization more effectively, rather than significantly improving relative scalability when increasing the thread count from two to four.

Higher cross-layer coupling diminishes 2-thread speedup due to its restriction on the degree of independent execution accessible within the system. However, in the case of 4-thread execution, the effect is typically less significant and exhibits greater variability, as certain highly coupled systems may still encompass considerable parallel work. This elucidates why the influence of cross-layer coupling on the relative speedup when scaling from two to four threads can differ markedly.

Strong interdependence typically diminishes both 2-thread and 4-thread speedup due to the need for more frequent synchronization among threads.

Layer separation and cluster separability enhance 2-thread speedup by facilitating clearer partitioning of the program into independent execution units. However, these metrics gain greater significance in 4-thread execution, as an increased number of threads necessitates well-defined work allocations to reduce contention and load imbalance. Consequently, improved layer separation and cluster separability bolster scalability and elevate the relative speedup when scaling from two to four threads by enabling more efficient utilization of additional threads.

Structural regularity and symmetry scores positively influence 2-thread speedup; however, they do not appear to facilitate scalability at higher thread counts.

8.3 Rough Set Analysis

Rough Set Theory (RST) is employed to identify minimal subsets of program-structure metrics that retain the ability to distinguish between different multithreading performance outcomes. In this investigation, RST is applied to determine the smallest sets of structural metrics that can effectively differentiate between the speedup achieved with two threads, the speedup achieved with four threads, and the relative speedup obtained when increasing the number of threads from two to four.

To facilitate the RST analysis, the continuous speedup values are first discretized into performance categories, namely High and Low speedup. These categories serve as decision classes, allowing rough-set concepts such as reducts, attribute significance, and approximate decision rules to be used to identify the structural metrics and metric combinations that provide discriminatory information between the performance classes.

The RST analysis of 2-thread speedup indicates that the acyclic structure of the DDG provides useful discriminatory information for distinguishing between the high- and low-speedup classes. In particular, Acyclic Score appears repeatedly in the extracted decision rules, together with attributes such as Structural Regularity, Symmetry Score, Parallel Width, and Layer Separation. These results indicate that particular combinations of dependency-graph structure and execution regularity are associated with differences in the observed two-thread speedup.

Three approximate decision rules provide representative examples of these relationships. The first rule indicates that a combination of high Acyclic Score and high Structural Regularity is associated with the high 2-thread speedup class. Thus, within the analyzed dataset, programs characterized by high values of these two structural attributes are more frequently associated with the high observed two-thread speedup class. This result should be interpreted as an association between the identified attribute combination and the decision class rather than as evidence that either attribute independently determines speedup.

The second rule indicates that high Acyclic Score combined with high Symmetry Score is associated with high 2-thread speedup. The occurrence of this combination in the extracted rule set suggests that these structural characteristics provide useful discriminatory information for distinguishing programs belonging to the high-speedup class. A high symmetry score may indicate a more balanced structural organization of the computation, which could facilitate a more even distribution of work between the two threads. However, this interpretation represents a possible structural explanation rather than a causal conclusion established by the RST analysis.

The third rule indicates that high Serialization Pressure is associated with the low 2-thread speedup class under particular combinations of the other structural attributes. High serialization pressure represents a stronger restriction on concurrent execution, which is consistent with reduced opportunities for the two threads to execute independent operations simultaneously. Consequently, configurations characterized by strong serialization constraints may be associated with lower observed speedup, although the relationship depends on the combination of Serialization Pressure with the other structural attributes appearing in the rule.

Overall, the RST results identify combinations of program-structure metrics that discriminate between the high- and low-speedup classes in the analyzed dataset. These rules should be interpreted as exploratory associations rather than causal relationships. In particular, rules with limited support or coverage provide indications of potentially relevant structural configurations but do not, by themselves, justify generalization to a broader population of programs without further validation.

Regarding 4-thread speedup, the RST analysis indicates that the attributes distinguishing the High and Low speedup classes increasingly involve the availability and separation of parallel work. In particular, Cluster Separability, Layer Separation, Parallel Width, and Serialization Pressure appear in the extracted decision rules. This suggests that, as the number of threads increases, the availability of sufficient independent work becomes increasingly relevant to achieving higher speedup [40].

Three approximate decision rules provide representative examples of these relationships. The first rule indicates that programs with high Parallel Width and high Cluster Separability are associated with the High 4-thread speedup class. Parallel Width represents the amount of work that can potentially be executed concurrently, while Cluster Separability characterizes the degree to which groups of operations are structurally independent. Their combination therefore represents a structural configuration with potentially sufficient parallel work to utilize multiple threads [40].

The second rule indicates that configurations characterized by low Serialization Pressure and high Layer Separation are associated with High 4-thread speedup. Lower Serialization Pressure represents fewer restrictions on concurrent execution, while greater Layer Separation indicates a more pronounced separation between dependency stages. Together, these characteristics are consistent with increased opportunities for concurrent execution across multiple threads [40].

The third rule indicates that configurations with low Parallel Width combined with high Serialization Pressure are associated with the Low 4-thread speedup class. When the amount of concurrently executable work is limited and execution is additionally constrained by serialization, additional threads have fewer opportunities to remain active. Consequently, the potential benefit of increasing the thread count may be reduced [40].

Overall, the 4-thread RST results indicate that the availability of sufficient independent work becomes particularly relevant when moving from fewer threads to four threads. Attributes such as Parallel Width and Cluster Separability provide information about the potential amount and independence of parallel work, while Serialization Pressure and Layer Separation describe structural constraints that can either restrict or facilitate concurrent execution. These results suggest that the effectiveness of additional threads depends not only on the structural organization of the program, but also on whether that structure exposes enough independent work to utilize the available execution resources [40].

The relative speedup obtained when increasing the number of threads from two to four was analyzed using RST. The results indicate that Parallel Width is a prominent attribute in the decision rules distinguishing High and Low relative speedup. This suggests that the amount of potentially concurrent computation provides useful discriminatory information for distinguishing whether performance continues to improve when additional threads are introduced. Programs with greater Parallel Width expose more operations that can potentially be executed concurrently, providing more opportunities to utilize additional processing resources when scaling from two to four threads.

Three approximate decision rules were identified. The first rule indicates that programs characterized by high Parallel Width are associated with the High relative-speedup class. A large Parallel Width represents a greater amount of potentially concurrent work, which can provide additional opportunities for utilizing the increased number of threads. However, the association should be understood in the context of the other structural characteristics represented in the RST classification rather than as a general monotonic relationship between Parallel Width and relative speedup.

The second rule indicates that high Parallel Width combined with low Serialization Pressure is associated with high relative speedup. This combination provides both a substantial amount of potentially parallel work and fewer restrictions on concurrent execution. Consequently, such configurations are consistent with greater opportunities for exploiting additional processing resources when increasing the thread count from two to four.

The third rule indicates that configurations characterized by high Acyclic Score and high Symmetry Score are associated with the Low relative-speedup class. Although these characteristics may be associated with favorable structural properties, their presence in the Low relative-speedup rule suggests that they do not necessarily correspond to substantial additional performance when moving from two to four threads. One possible interpretation is that the available parallelism may already be

effectively exploited at the lower thread count; however, this interpretation should be regarded as a possible explanation rather than a direct conclusion of the RST analysis.

Overall, the relative speedup from two to four threads appears to be associated with the amount of potentially parallel work and the structural constraints on concurrent execution. Programs characterized by greater Parallel Width and lower Serialization Pressure are associated, under the identified RST rules, with greater opportunities for exploiting additional threads, whereas other structural configurations may exhibit smaller incremental gains. Thus, the RST results suggest that scalability from two to four threads depends not only on the presence of parallelism, but also on the combination of structural characteristics that distinguish configurations with greater or smaller opportunities for additional concurrent execution.

Evaluation and Conclusions

9.1 Evaluation of Multi-Core Synchronization Point Insertion

The first evaluation examines the initial concept of using a custom LLVM middle-end pass to automatically insert IR load and store instructions that use hardware-based semaphores at multi-core synchronization points, specifically at the boundaries between multiple threads. This investigation demonstrates that the proposed synchronization model requires two clock cycles for a load operation and one clock cycle for a store operation. The load latency is therefore equivalent to that of a conventional data load from memory. Building on these results, the second evaluation demonstrates the feasibility of using the proposed DATM algorithm to partition and distribute workloads across multiple cores during the compiler's middle-end stage through the insertion of synchronization points.

9.2 Evaluation of the DATM Algorithm

The DATM algorithm utilizes the DFS algorithm facilitates the analysis of the data-dependency graph, which in turn supports balanced partitioning across multiple threads based on traversable execution paths. In addition, it identifies locations where multicore synchronization points must be inserted to ensure correct handling of data dependencies among threads that share data across multiple execution paths.

The conducted RST analysis identifies a set of program-structure metrics that repeatedly occur in the decision rules associated with multithreaded performance at different thread configurations. For 2-thread execution, the results indicate that the acyclic nature of the DDG, together with Structural Regularity and Symmetry Score, is associated with higher speedup. The acyclic nature of the DDG is particularly relevant because it reflects the absence of feedback loops within the PLC program, which can limit the availability of independent operations for parallel execution. For 4-thread execution, the extracted rules place greater emphasis on metrics associated with Parallel Width, Cluster Separability, and Layer Separation, suggesting that the

availability and organization of parallel work become increasingly relevant as the number of threads increases.

Furthermore, the relative speedup from two to four threads further highlights the importance of Parallel Width, as wider computational structures provide greater potential for exploiting additional threads. Across all thread configurations, Serialization Pressure also appears repeatedly in the extracted rules, indicating an association between serialization constraints, dependency complexity, and the resulting speedup. These relationships should be interpreted as associations identified by the rough-set analysis.

9.3 Achieved Aims and Objectives

The primary objective of this thesis is to assess an approach for automatically partitioning PLC programs into multiple threads and integrating multicore synchronization using hardware-based semaphores. The research demonstrates that logic expression used in control systems (e.g. expressed with use of LD) can be decomposed into short sequences of instruction that can be processed in parallel, mutually dependent computational fragments during the compiler middle-end stage by constructing and analysing the DDG. A DFS algorithm is used to traverse the possible paths within the DDG and identify path intersections, which can subsequently be used to determine multicore synchronization points according to the configured number of threads, without modifying the IEC 61131-3 representation of the PLC program. Furthermore, the investigation has demonstrated that the speedup achieved through automatic multithreading remains constrained by program-structure metrics that determine the potential for efficient execution when transitioning from a single core to multiple cores.

9.4 Future Work

The proposed multi-threaded algorithm for PLC compilers demonstrates that automatic code partitioning and the insertion of hardware-based semaphore synchronization mechanisms can be implemented through a custom compiler middle-end pass for standard PLC languages or the back-end code-generation process; furthermore, this approach enables the utilization of other parallel computation synchronization mechanisms.

This provides a strong foundation for further research into the development of a specialized PLC runtime system, capable of ensuring the deterministic execution of multiple function blocks and background services, such as safety functions, with each executed concurrently. Such a runtime system could preserve the conventional

uniprocessor execution model while simplifying global-variable dependencies across the entire PLC program.

Bibliography

[1] Aho, A.V.; Lam, M.S.; Sethi, R.; Ullman, J.D. :
Compilers: Principles, Techniques, and Tools, 2nd ed.;
Addison-Wesley: Boston, MA, USA,
2006.

[2] Appel, A.W.; Ginsburg, M. :
Modern Compiler Implementation in C;
Cambridge University Press, Cambridge, UK,
1998.

[3] Ayers, G. :
MIPS32 Verilog Core.
GitHub repository. Available online: https://github.com/grantae/OpenMIPS (accessed on 18 March 2026).

[4] Balasubramonian, R.; Jouppi, N.; Muralimanohar, N. :
Multi-Core Cache Hierarchies;
Springer,
2011.

[5] Boldt, M.; Traulsen, C.; von Hanxleden, R. :
Worst Case Reaction Time Analysis of Concurrent Reactive Programs;
2008.
doi:10.1016/j.entcs.2008.05.011.

[6] Çatalyürek, Ü. V.; Feo, J.; Gebremedhin, A. H.; Halappanavar, M.; Pothen, A. :
Graph Coloring Algorithms for Multi-Core and Massively Multithreaded Architectures;
2012.
doi:10.1016/j.parco.2012.07.001.

[7] Chakravarty, D.; Cannon, C. :
PowerPC Concepts, Architecture, and Design;
Morgan Kaufmann Publishers.
1994.

[8] Chapman, B.; Jost, G.; van der Pas, R.; Kuck, D.J. :
Using OpenMP: Portable Shared Memory Parallel Programming, 2nd ed.;
Cambridge University Press: Cambridge, UK,
2008.

[9] Cheng, X.; Zhao, H.; Kandemir, M.; Jiang, B.; Mehta, G. :
AMOEBA: A Coarse Grained Reconfigurable Architecture for Dynamic GPU Scaling;
2020.
doi:10.1145/3392717.3392738.

[10] Cooper, K.D.; Torczon, L. :
Engineering a Compiler, 2nd ed.;
Morgan Kaufmann: Burlington, MA, USA,
2011.

[11] Crockett, L.H.; Elliot, R.A.; Enderwitz, M.A.; Stewart, R.W. :
The Zynq Book: Embedded Processing with Arm Cortex-A9 on Xilinx Zynq-7000 All Programmable SoC;
Strathclyde Academic Media.
2014.

[12] Dijkstra, E. W. :
Solution of a Problem in Concurrent Programming Control;
1965.
doi:10.1145/365559.365617.

[13] Ezer, G. :
Breaking the I/O Bottleneck for High Compute Performance Processing with Xtensa LX Configurable and Extensible Processor Architecture;
2005.
doi:10.1117/12.586013.

[14] Garcia Ramirez, A.; Schmaltz, J.; Verbeek, F.; Langenstein, B.; Blasum, H. :
On Two Models of Noninterference: Rushby and Greve, Wilding, and Vanfleet;
2014.
doi:10.1007/978-3-319-10506-2_17.

[15] Gaidis, A. J.; Moreira, J.; Sun, K.; Milburn, A.; Atlidakis, V.; Kemerlis, V. P. :
FineIBT: Fine-Grain Control-Flow Enforcement with Indirect Branch Tracking;
2023.
doi:10.1145/3607199.3607219.

[16] Gharachorloo, K.; Lenoski, D.; Laudon, J.; Gibbons, P.; Gupta, A.; Hennessy, J. :
Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors;
1990.
doi:10.1145/325096.325102.

[17] Grosser, T.; Größlinger, A.; Lengauer, C. :
Polly – Performing Polyhedral Optimizations on a Low-Level Intermediate Representation;
2012.
doi:10.1142/S0129626412500107.

[18] Guerraoui, R.; Kapalka, M. :
Principles of Transactional Memory;
Springer: Berlin, Germany,
2010.

[19] Hamzeh, M.; Shrivastava, A.; Vrudhula, S. :
Branch-Aware Loop Mapping on CGRAs;
2014.
doi:10.1145/2593069.2593100.

[20] Harris, T.; Larus, J.R.; Rajwar, R. :
Transactional Memory, 2nd ed.;
Morgan and Claypool Publishers: San Rafael, CA, USA,
2010.

[21] Heinrich, M.; Vateva-Gurova, T.; Arul, T.; Katzenbeisser, S.; Suri, N.; Birkholz, H.; Fuchs, A.; Krauß, C.; Zhdanova, M.; Kuzhiyelil, D.; Tverdyshev, S.; Schlehuber, C. :
Security Requirements Engineering in Safety-Critical Railway Signalling Networks;
2019.
doi:10.1155/2019/8348925.

[22] Herlihy, M.; Shavit, N.; Luchangco, V.; Spear, M. :
The Art of Multiprocessor Programming, 2nd ed.;
Morgan Kaufmann: Burlington, MA, USA,
2020.

[23] John, K.-H.; Tiegelkamp, M. :
IEC 61131-3 Programming Industrial Automation Systems;

Springer: Berlin, Germany,
1995.

[24] Johnson, M. :
Superscalar Microprocessor Design;
Prentice Hall: Englewood Cliffs, NJ, USA,
1991.

[25] Jain, K.; Korgaonkar, K. :
Performance Evaluation of a Hardware Transaction Memory System;
2011

[26] Kane, G.; Heinrich, J. :
MIPS RISC Architecture, 2nd ed.;
1991.

[27] Koul, K.; Melchert, J.; Sreedhar, K.; Truong, L.; Nyengele, G.; Zhang, K.; Liu, Q.; Setter, J.; Chen, P.-H.; Mei, Y.; Strange, M.; Daly, R.; Donovick, C.; Carsello, A.; Kong, T.; Feng, K.; Huff, D.; Nayak, A.; Setaluri, R.; Thomas, J.; Bhagdikar, N.; Durst, D.; Myers, Z.; Tsiskaridze, N.; Richardson, S.; Bahr, R.; Fatahalian, K.; Hanrahan, P.; Barrett, C.; Horowitz, M.; Torng, C.; Kjolstad, F.; Raina, P. :
AHA: An Agile Approach to the Design of Coarse-Grained Reconfigurable Accelerators and Compilers;
2023.
doi:10.1145/3534933

[28] LaCamera, D. :
Embedded System Architecture: Explore Architecture Concepts, Programming Design Patterns, and Best Practices to Produce Robust Systems;
Packt Publishing.
2012.

[29] Lee, G.; Lee, S.; Choi, K. :
Automatic Mapping of Application to Coarse-Grained Reconfigurable Architecture Based on High-Level Synthesis Techniques;
2008.
doi:10.1109/SOCDC.2008.4815655.

[30] Lengauer, T.; Tarjan, R. E. :
A Fast Algorithm for Finding Dominators in a Flowgraph;
1979.
doi:10.1145/357062.357071.

[31] Lin, B. :
Compiling Concurrent Programs for Embedded Sequential Execution;
2007.
doi:10.1016/j.vlsi.2006.02.008.

[32] Mano, M.M.; Kime, C.R.; Martin, T. :
Logic and Computer Design Fundamentals, 5th ed.;
Pearson: Harlow, UK,
2015.

[33] Mazidi, M.A.; Naimi, S.; Naimi, S.; Chen, S. :
ARM Assembly Language Programming and Architecture;
2016.

[34] Mei, B.; Vernalde, S.; Verkest, D.; De Man, H.; Lauwereins, R. :
DRESC: A Retargetable Compiler for Coarse-Grained Reconfigurable Architectures;
2002.
doi:10.1109/FPT.2002.1188678.

[35] Milik, A.; Hrynkiewicz, E. :
Distributed PLC Based on Multicore CPUs: Architecture and Programming;
2016.
doi:10.1016/j.ifacol.2016.12.001.

[36] Milik, A.; Walichiewicz, M :
Fast Multithreaded Computations System Implemented in ZYNQ Architecture;
2022.
doi:10.1016/j.ifacol.2022.06.050.

[37] Milik, A.; Walichiewicz, M :
Shared-Semaphored Cache Implementation for Parallel Program Execution in Multi-Core Systems;
2023
doi:10.24425/ijet.2023.144373.

[38] Milik, A.; Walichiewicz, M :
Analysis of Coprocessor-based Acceleration for Hardware-based Semaphore Concurrency Control;
2026.
doi:10.24425/ijet.2026.1733.

[39] Milik, A.; Walichiewicz, M :
Analysis of Hardware-Based Semaphore Mechanisms for Concurrency Control in Real-Time Systems;
2026.

[40] Milik, A.; Walichiewicz, M :
Dependency-Aware LLVM Parallelization of Combinational Ladder Diagram Programs;
2026.

[41] Nacke, K. :
Learn LLVM 12: A Beginner's Guide to Learning LLVM Compiler Tools and Core Libraries with C++;
Packt Publishing: Birmingham, UK,
2021.

[42] Olukotun, K.; Hammond, L.; Laudon, J. :
Chip Multiprocessor Architecture: Techniques to Improve Throughput and Latency;
Morgan and Claypool Publishers
2007.

[43] Palnitkar, S.; Goel, P. :
Verilog HDL: A Guide to Digital Design and Synthesis, IEEE 1364-2001 Compliant, 2nd ed.;
Prentice Hall PTR.
2003.

[44] Patterson, D.A.; Hennessy, J.L. :
Computer Architecture: A Quantitative Approach, 6th ed.;
Elsevier: Oxford, UK,
2017.

[45] Patterson, D.A.; Hennessy, J.L. :
Computer Organization and Design, 4th ed.;
Morgan Kaufmann: Burlington, MA, USA,
2011.

[46] Reineke, J. :
Cache in WCET Analysis;
2008.

[47] Rosemann, J.; Moll, S.; Hack, S. :
An Abstract Interpretation for SPMD Divergence on Reducible Control Flow Graphs;
2021.
doi:10.1145/3434312.

[48] Sagdeo, V. :
The Complete Verilog Book;
Springer: New York, NY, USA,
1998.

[49] Sanders, P.; Schulz, C. :
KaHIP v3.00 – Karlsruhe High Quality Partitioning – User Guide;
2013.
doi:10.48550/arXiv.1311.1714.

[50] Sarda, S.; Pandey, M. :
LLVM Essentials: Become Familiar with LLVM Infrastructure and Start Using LLVM Libraries to Design a Compiler;
Packt Publishing.
2014

[51] Scott, M.L. :
Shared-Memory Synchronization;
Morgan and Claypool Publishers: San Rafael, CA, USA,
2013.

[52] Schardl, T.B.; Moses, W.S.; Leiserson, C.E. :
Tapir: Embedding Recursive Fork-Join Parallelism into LLVM's Intermediate Representation;
2019.
doi:10.1145/3365655.

[53] Schardl, T. B.; Samsi, S. :
TapirXLA: Embedding Fork-Join Parallelism into the XLA Compiler in TensorFlow Using Tapir;
2019.
doi:10.1109/HPEC43674.2020.9286232.

[54] Singh, A. :
Compiler Design: Simply in Depth;
Technical Publications.
2011.

[55] Smith, D. J.; Simpson, K. G. L. :
Functional Safety: A Straightforward Guide to Applying IEC 61508 and Related Standards Second Edition.;
Butterworth-Heinemann.
2009.

[56] Smith, S. W.; Weingart, S. :
Building a High-Performance, Programmable Secure Coprocessor;
1999.
doi:10.1016/S1389-1286(98)00019-X.

[57] Solihin, Y. :
Fundamentals of Parallel Multicore Architecture;
Chapman and Hall/CRC: Boca Raton, FL, USA,
2015.

[58] Sorin, D.J.; Hill, M.D.; Wood, D.A. :
A Primer on Memory Consistency and Cache Coherence;
Morgan and Claypool Publishers: San Rafael, CA, USA,
2020.

[59] Sorin, D. J. :
Fault Tolerant Computer Architecture; Morgan and Claypool Publishers. 2009.

[60] Stallings, W. :
Computer Organization and Architecture: Designing for Performance, 10th ed.;
Pearson: Harlow, UK,
2015.

[61] Staron, M. :
Automotive Software Architecture: An Introduction;
Springer.
2006.

[62] Strøm, T.B.; Sparsø, J.; Schoeberl, M. :
Hardlock: Real-Time Multicore Locking;
2019.
doi:10.1016/j.sysarc.2019.02.003.

[63] Titos-Gil, R. :
Hardware Support for Efficient Transactional Memory Systems: Hardware Techniques for High-Performance Transactional Memory in Many-Core Chip Multiprocessors;

Stanford University.
2007.

[64] Thornton, M.A. :
Modeling Digital Switching Circuits with Linear Algebra;
Morgan and Claypool Publishers.
2014.
doi:10.2200/S00579ED1V01Y201404DCS044

[65] Tohara, T. :
A New Kind of Processor Interface for a System-on-a-Chip Processor with TIE Ports and TIE Queues of Xtensa LX;
2005.
doi:10.1109/IWIA.2005.23.

[66] Zhao, Y.; Yang, Z.; Ma, D. :
A Survey on Formal Specification and Verification of Separation Kernels;
2016.
doi:10.1007/s11704-016-4226-2.

[67] Ziavras, S. G.; Gerbessiotis, A. V.; Bafna, R. :
Coprocessor Design to Support MPI Primitives in Configurable Multiprocessors;
2007.
doi:10.1016/j.vlsi.2005.10.001.

[68] Zoppetti, G. M.; Agrawal, G.; Pollock, L. L.; Amaral, J. N.; Tang, X.; Gao, G. R. :
Automatic Compiler Techniques for Thread Coarsening for Multithreaded Architectures;
2000.
doi:10.1145/335231.335261.

[69] AMD Xilinx. *Zynq-7000 All Programmable SoC Technical Reference Manual*;
2021.

[70] Cadence Design Systems. *High-Speed Alternatives for Inter-Processor Communications in SoCs*;
White Paper, Tensilica Xtensa Processor Family,
2021.

[71] MIPS Technologies. *MIPS Architecture for Programmers, Volumes I–IV*;

[72] *MIPS R3010 Floating-Point Coprocessor*;
Institute of Electrical and Electronics Engineers
1989.

[73] Kristiansson, S. :
OMPIC.
Available online: https://github.com/openrisc/ompic (accessed on 6 February 2026).

[74] OpenRISC.
OpenRISC 1000 Architecture Manual, Revision 1.4, 2022.

[75] Texas Instruments.
KeyStone Architecture Semaphore2 User Guide; Texas Instruments Incorporated, Application Documentation, 2020.

[76] LDRA. A Holistic Approach to Meet Multicore System Timing and Interference Requirements.
Available online: https://ldra.com/a-holistic-approach-to-meet-multico
(accessed on 23 January 2026).

[77] LLVM. LLVM's Analysis and Transform Passes.
Available online: https://llvm.org/docs/Passes.html (accessed on 11 December 2025).

[78] OpenPLC Project. Intel Edge Controls for Industrial Documentation.
Available online: https://eci.intel.com/docs/3.3/components/openplc.html (accessed on 8 April 2026).

[79] PLC-lang Project. RuSTy Documentation.
Available online: https://plc-lang.github.io/rusty/ (accessed on 26 February 2026).