REPORT ON

RISC-V STATIC INSTRUCTION SCHEDULING IDE AND EMULATOR

SUBMITTED BY

MADOORI SOWMITH (231EC228)
MEGHANA R (231EC230)

Under The Guidance of

Dr. Aparna P Dinesh

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

NATIONAL INSTITUTION OF TECHNOLOGY, KARNATAKA (NITK)

NOVEMBER 2025

CONTENTS

ABSTRACT

This project presents the design and implementation of a RISC-V Static Instruction Scheduling IDE with an integrated emulator, developed to analyze and improve processor efficiency through compiler-level optimization. The IDE accepts a minimal C-like input, compiles it to RISC-V assembly, performs static scheduling to reorder dependent instructions, and emulates pipeline behavior for cycle-level performance analysis. By shifting scheduling intelligence from hardware to compiler software, this work demonstrates that smart compilers can effectively replace costly Out-of-Order (OoO) execution hardware, reducing design complexity, silicon area, and power consumption. The results highlight the potential of compiler-driven performance optimization for low-power and edge computing systems, establishing a foundation for future research in hardware-aware compiler design.

Chapter 1

1.1 INTRODUCTION

Compilers play a crucial role in determining how efficiently a processor executes a program. They translate high-level code into optimized machine instructions, directly influencing instruction flow, register usage, and memory access patterns. A well-designed compiler can significantly enhance processor performance without requiring any hardware modifications.

One of the key techniques used in modern compiler design is Static Instruction Scheduling. This method involves reordering instructions at compile time to minimize pipeline stalls caused by data and control dependencies. By analyzing instruction dependencies in advance, the compiler arranges operations so that the processor pipeline remains continuously active, reducing idle cycles and improving overall throughput.

Such software-level optimization offers a cost-effective alternative to complex hardware-based solutions like Out-of-Order (OoO) execution. While OoO processors achieve high performance through additional hardware logic, they also increase chip area, power consumption, and design complexity. Static scheduling, in contrast, shifts this intelligence to the compiler, enabling simpler, low-power processors to perform efficiently through smarter code organization.

1.2 MOTIVATION

Modern processors depend heavily on complex hardware mechanisms such as Out-of-Order (OoO) execution, branch prediction, and speculative execution to maintain high efficiency. While these techniques improve instruction throughput, they significantly increase silicon area, design complexity, and power consumption.

A compiler equipped with static instruction scheduling offers an elegant alternative. By reordering instructions intelligently at compile time, the compiler can achieve near-OoO performance without additional hardware complexity. This approach allows even simple in-order processors to operate more efficiently.

Furthermore, workloads like neural network computations involve millions of repetitive arithmetic operations, making them ideal for evaluating compiler-based optimizations. Measuring performance improvements on such patterns provides clear evidence of how static scheduling enhances real-world computational performance.

1.3 PROBLEM STATEMENT

Pipeline stalls occur when instructions must wait for earlier results, causing idle cycles and reducing overall processor efficiency. In arithmetic-heavy workloads, these stalls accumulate rapidly and become a major performance bottleneck.

Modern processors mitigate stalls using complex hardware techniques like Out-of-Order execution, but these require more transistors, higher power, and increased design cost—making them unsuitable for low-power or embedded systems.

A practical alternative is software-level static scheduling, where the compiler reorders instructions to minimize stalls without adding hardware complexity. This approach demands intelligent dependency analysis and clear visibility into instruction execution. As shown in Fig. 1.1, dependent instructions create hazards that force the pipeline to pause until required results are available. These stalls introduce wasted cycles, reducing overall pipeline utilization and lowering execution throughput, especially in arithmetic-intensive workloads.

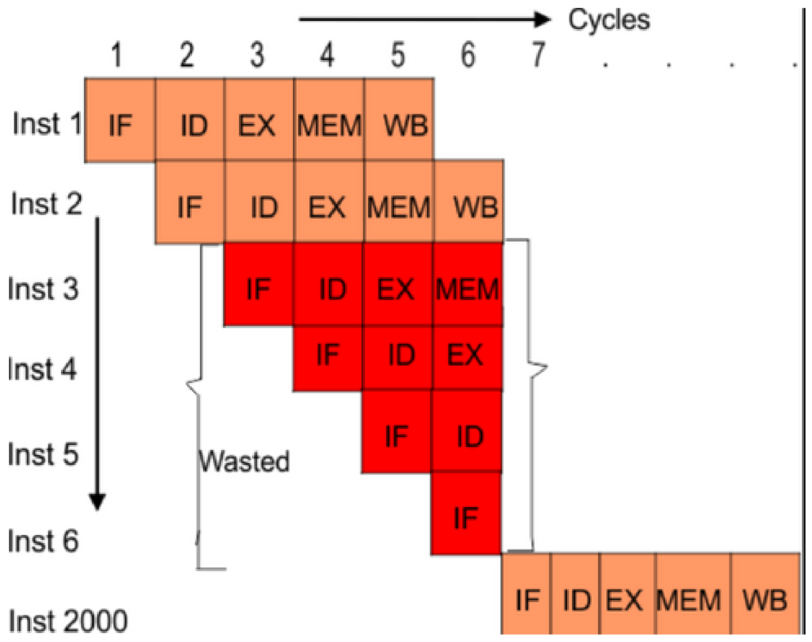


Fig. 1.1: Pipeline stall example illustrating wasted cycles due to instruction dependencies.

1.4 OBJECTIVE

- The objective of this project is to build an IDE that can compile, statically schedule, and emulate RISC-V programs, allowing users to visualize pipeline behavior and measure cycle improvements before and after scheduling.
- To design and implement a RISC-V Static Instruction Scheduling IDE with an integrated emulator.
- To deploy and test the system on a Raspberry Pi (ARM CPU) through SSH on Ubuntu, ensuring cross-platform compatibility.
- To evaluate the IDE's performance by running computational workloads involving repetitive arithmetic operations, similar to those used in neural networks.
- To quantify the reduction in clock cycles achieved through static scheduling and demonstrate its potential impact on processor performance and hardware efficiency.

Chapter 2

LITERATURE SURVEY

Instruction scheduling research (LLVM, GCC, early static schedulers)

Instruction scheduling has long been a core component of compiler backends. Classic compilers such as GCC and LLVM incorporate multiple phases that perform register allocation, peephole optimizations, and instruction scheduling to improve instruction-level parallelism and reduce pipeline stalls. Early static schedulers used heuristics and priority-based list-scheduling to reorder instructions for simple in-order pipelines; more recent work in LLVM applies sophisticated cost models and machine descriptions to generate target-aware schedules. As shown in Fig. 2.1, modern compilers follow a modular pipeline where the front-end converts high-level source code into LLVM IR, the middle-end applies optimizations, and the back-end generates target-specific machine code. This structure enables instruction scheduling, register allocation, and other backend optimizations to be performed systematically across different architectures.

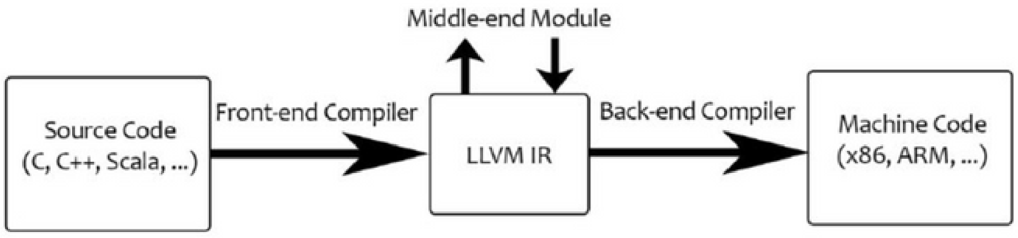


Fig. 2.1: LLVM's modular compiler flow from source code to optimized machine code.

Static vs dynamic (Out-of-Order) scheduling

Static scheduling reorders instructions at compile time based on dependency analysis and a model of the target pipeline. It is lightweight in hardware cost but limited by the compiler's knowledge and conservatism. Dynamic scheduling (OoO execution), implemented in hardware, resolves dependencies and reissues instructions at runtime, extracting more parallelism in the presence of unpredictable events (cache misses, branches). The trade-off is clear: OoO gives higher average performance across diverse workloads at the expense of increased silicon area, power, and design complexity; static scheduling favors simpler, energy-efficient processors while relying on compiler intelligence for performance.

Importance for embedded neural processors

Embedded and edge devices frequently use simpler in-order cores with strict area and power budgets. For such platforms, compiler-driven optimizations that reduce stalls and improve utilization can yield disproportionate gains in throughput and energy efficiency. Bringing instruction-level scheduling intelligence into the compilation flow makes it possible to accelerate neural workloads on low-cost hardware without adding complex OoO logic.

Research gap

While there is extensive work on high-level model optimizations and sophisticated dynamic hardware schedulers, there is a relative gap at the intersection: a lightweight, demonstrable toolchain that (1) performs ISA-level static scheduling targeted at simple pipelines, (2) provides an easy-to-use IDE and emulator for visualization and analysis, and (3) quantifies hardware-independent improvements on realistic workloads. This project targets that gap by providing a compact compiler + scheduler + emulator that shows measurable cycle savings on neural-style kernels running on constrained hardware (Raspberry Pi).

Deep-learning compilers: TVM, TensorRT, Glow (graph-level focus)

Modern deep-learning toolchains (for example, TVM, TensorRT, and Glow) concentrate on graph-level and tensor-level optimizations: operator fusion, layout transformations, tiling, quantization, and hardware-specific lowering (vectorization, tensor cores). These frameworks excel at mapping high-level neural network graphs to accelerators and SIMD/vector instructions but typically operate above the instruction-scheduling layer. They often assume an efficient backend compiler or rely on hardware features (SIMD, tensor units) rather than performing fine-grained, ISA-level instruction reordering for in-order cores.

Chapter 3

METHODOLOGY

The development of the RISC-V Static Instruction Scheduling IDE with Emulator was carried out in several integrated stages, combining compiler construction, instruction-level scheduling, and pipeline simulation. The process began by designing a minimal C-like language parser that translates source code into RISC-V assembly instructions. This translation phase was followed by an assembler module that generated the corresponding machine code (binary format). The compiler also incorporated register allocation and basic dependency tracking to identify instruction relationships, enabling further optimization during scheduling.

At the core of the system lies the Static Instruction Scheduler, responsible for analyzing dependencies such as Read After Write (RAW) and Write After Read (WAR) hazards. Using this analysis, the scheduler reorders independent instructions to fill pipeline bubbles, reducing idle cycles without altering the program's logical behavior. The scheduler's effectiveness was validated through an integrated emulator that models a five-stage RISC-V pipeline (IF, ID, EX, MEM, WB). The emulator records the execution of each instruction across cycles, allowing the user to visualize the pipeline activity in detailed before and after scheduling timelines.

The entire system was developed in Python with a Tkinter-based GUI, ensuring both cross-platform compatibility and user interactivity. Execution was carried out on a Raspberry Pi 3 B+ running Ubuntu via SSH to simulate a real embedded environment. Performance was evaluated using a CNN-style arithmetic workload, where results demonstrated a reduction in clock cycles from 184 to 109 per pixel — a 40% improvement achieved purely through software-level reordering. Thus, the methodology effectively integrates compiler design, pipeline modeling, and performance evaluation into a single IDE that highlights how intelligent compilation can emulate hardware-level efficiency without additional processor complexity.

3.1 Component Specification

Component	Description
Hardware	Raspberry Pi 3 B+ (ARM Cortex-A53 CPU) used as the test platform to ensure the IDE functions efficiently on resource-limited embedded hardware.
Operating System	Ubuntu 22.04, accessed remotely via SSH terminal from a host system. This setup allowed compiling, running, and monitoring results directly on the Raspberry Pi.
Software	Custom-built Python-based IDE integrating a RISC-V assembler, static instruction scheduler, and pipeline emulator.
Libraries	Tkinter (for GUI), Python subprocess (for task handling).
Target ISA	RISC-V 32-bit

As shown in the table, the IDE was developed on a Raspberry Pi running Ubuntu, using custom Python tools and libraries to target a 32-bit RISC-V ISA.

3.2 Design Considerations

The system is designed around a five-stage RISC-V pipeline consisting of the stages Instruction Fetch (IF), Instruction Decode (ID), Execution (EX), Memory (MEM), and Write Back (WB).
The emulator models pipeline behavior, capturing the effects of data hazards (Read After Write dependencies), control hazards, and pipeline stalls that occur in naïvely ordered code.
The Static Instruction Scheduler plays the central role:

- It performs dependency analysis to detect data hazards.
- Reorders independent instructions to fill idle pipeline slots (bubbles).

The scheduled output is then verified through the built-in emulator and timeline viewer, which simulates the pipeline clock-by-clock.
Testing on the Raspberry Pi ensures accurate timing behavior and confirms the IDE's performance under real-world, resource-constrained environments.

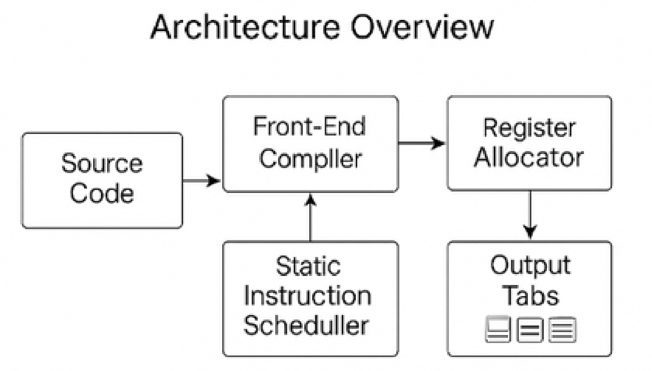


Fig. 3.1: High-level architecture of the static instruction scheduling IDE.

As shown in Fig. 3.1, the system flows from source code through the front-end compiler, register allocator, and static scheduler, with results displayed in the output tabs.

3.3 Algorithm for Static Instruction Scheduling

Step 1: Build the Dependency Graph (DAG)

The compiler converts the instruction sequence into a Directed Acyclic Graph where:

- Nodes represent instructions
- Edges represent dependencies, usually
- RAW (Read After Write) – true dependency
- WAR and WAW if needed
- An instruction cannot be scheduled before its dependencies.

This DAG identifies what instructions can be executed in parallel or interleaved.

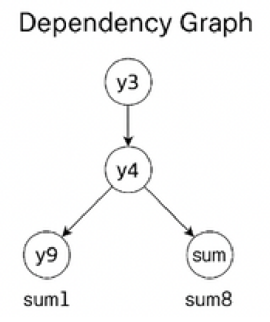


Fig. 3.2: Dependency graph showing RAW relationships between instructions.

As shown in Fig. 3.2, the scheduler uses RAW dependencies to detect independent instructions and reorder them to minimize stalls.

Step 2: Compute Instruction Priorities

Each node (instruction) is assigned a priority or "weight," commonly based on:

- Its latency (e.g., mul > addi)
- Length of the critical path from that instruction to the end
- Importance of the instruction in feeding others

Higher-priority instructions should be scheduled earlier to reduce total stalls.

Step 3: Create a Ready List

The scheduler maintains a Ready List of instructions whose dependencies are satisfied.
Initially, the Ready List contains all instructions with no incoming edges (nothing needs to be executed before them).

Step 4: Select the Best Next Instruction

At each cycle slot:

1. Choose an instruction from the Ready List with the highest priority.
2. Ensure the instruction does not cause hazards with the currently scheduled instructions:
- no register conflicts
- no structural hazards
3. Place the instruction into the next position in the final schedule.

If no safe instruction exists:

- Insert a NOP (bubble), although the goal is to avoid this.

Step 5: Update Dependencies

After scheduling an instruction:

- Remove it from the DAG
- Remove its outgoing edges
- Add newly unlocked instructions to the Ready List

This repeats until all instructions are scheduled.

Step 6: Emit the Final Reordered Instruction Stream

The output is a semantically identical sequence of instructions but arranged with:

- Fewer stalls
- Better pipeline utilization
- Reduced bubbles in EX/MEM/WB stages

This final schedule is fed to the emulator, where you saw cycle reduction from
184 → 109 cycles.

3.4 Architecture of the IDE

Compiler and Assembler Layer:
Accepts minimal C-like user code and translates it into RISC-V assembly and 32-bit machine code. This layer handles lexical parsing and syntax mapping of operations.

Static Scheduler:
Analyzes the instruction stream, identifies RAW (Read After Write) dependencies, and reorders instructions to minimize pipeline stalls. It ensures improved instruction throughput while preserving execution correctness.

Emulator:
Simulates the 5-stage pipeline operation, executing instructions cycle-by-cycle. It calculates total clock cycles, highlights stalls, and validates scheduled code performance.

GUI Interface:
Built with Tkinter, the interface provides an intuitive workspace that displays:
- Source code and compiled assembly
- Pre- and post-scheduling timelines
- Total clock cycles before and after optimization
- Real-time emulator output

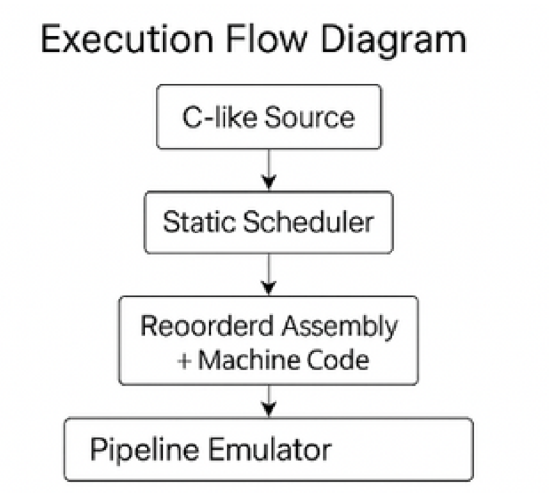


Fig. 3.3: Execution flow of the static scheduling IDE.

As shown in Fig. 3.3, the IDE takes C-like input, applies static scheduling, generates reordered assembly and machine code, and runs it in the pipeline emulator for cycle-level analysis.

Chapter 4

RESULTS OBTAINED

4.1 CNN Test Case Execution

A small 3×3 convolution kernel, equivalent to one CNN neuron, was used to test the efficiency of the static scheduler. The code performs multiply–accumulate operations followed by a simple activation step:

```
x1 = 1
x2 = 2
x3 = 3
x4 = 4
x5 = 5
x6 = 6
x7 = 7
x8 = 8
x9 = 9
w1 = 1
w2 = 0
w3 = -1
w4 = 2
w5 = 1
w6 = 0
w7 = -2
w8 = 1
w9 = 1
y1 = x1 * w1
y2 = x2 * w2
y3 = x3 * w3
y4 = x4 * w4
y5 = x5 * w5
y6 = x6 * w6
y7 = x7 * w7
y8 = x8 * w8
y9 = x9 * w9
```

```
sum1 = y1 + y2
sum2 = sum1 + y3
sum3 = sum2 + y4
sum4 = sum3 + y5
sum5 = sum4 + y6
sum6 = sum5 + y7
sum7 = sum6 + y8
sum8 = sum7 + y9
relu = sum8 - 3
out = relu + 10
```

This represents a basic convolutional operation, ideal for evaluating compiler-level instruction reordering.
Results on Raspberry Pi

Clock Cycles (per pixel)

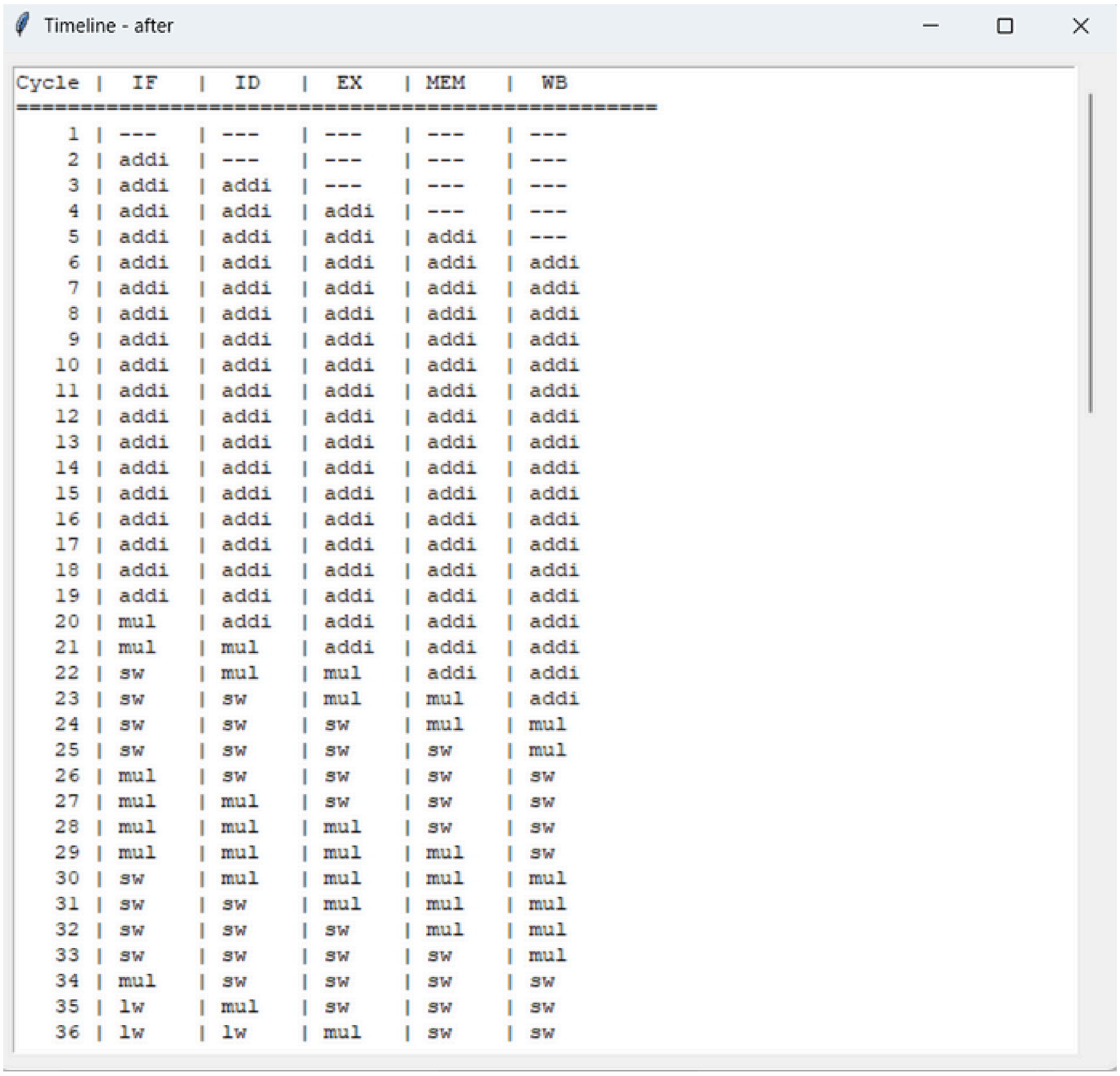
Metric Before Scheduling	After Scheduling Improvement
184	109

75 cycles saved (~41%)

Extrapolation to Full CNN Layer

For a 580 × 400 image (232,000 pixels):
75 cycles/pixel×232,000=17,400,000 cycles saved.

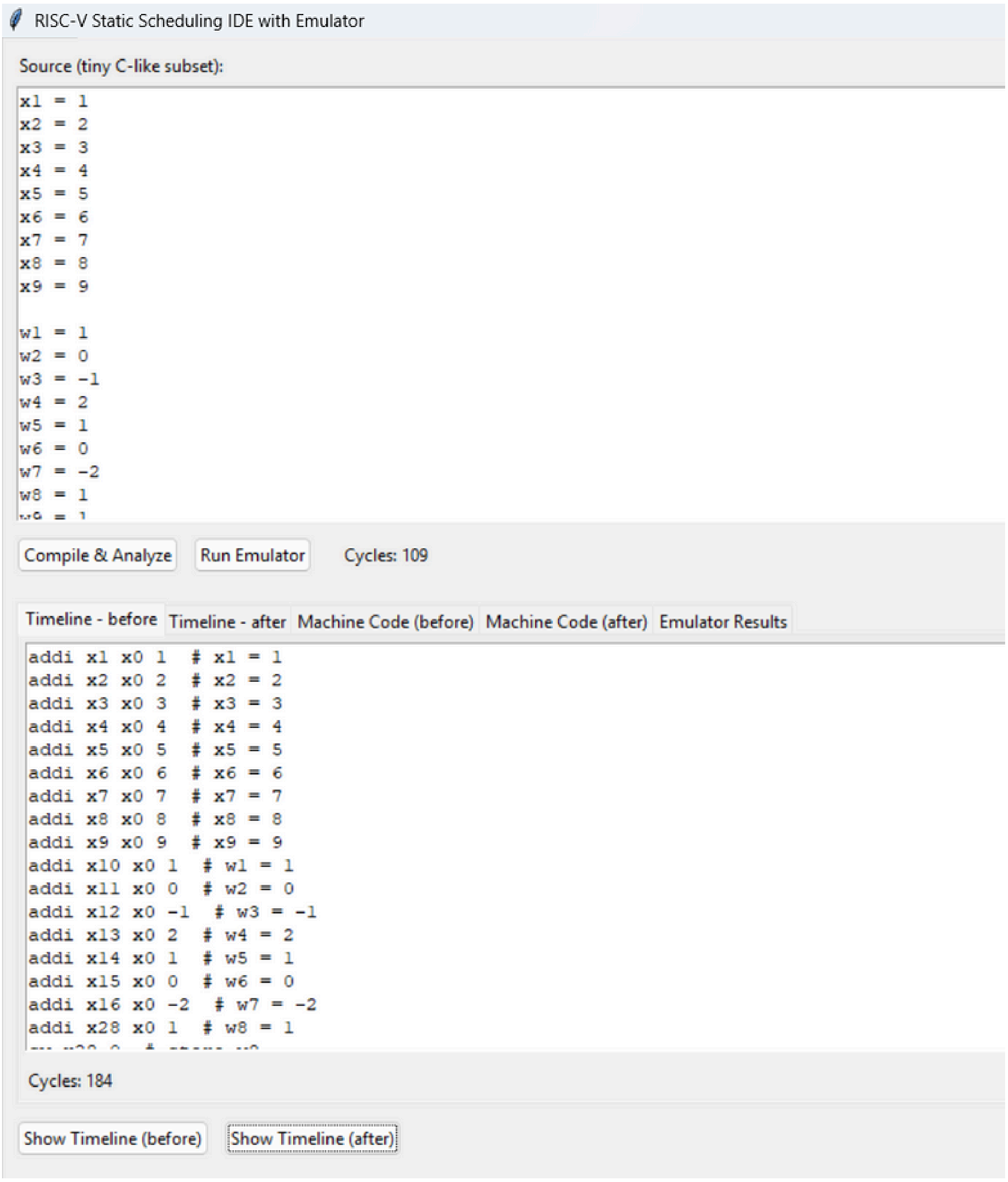
- This equals ~17.4 million clock cycles saved per convolution layer.
- For a CNN with 10 such layers, the saving scales to **~174 million cycles.**
- On a 1 GHz processor, this corresponds to roughly 0.17 seconds saved per frame, a significant gain for real-time embedded systems.



```
Cycle |  IF    |  ID    |  EX    | MEM    |  WB
==================================================
    1 | ---    | ---    | ---    | ---    | ---
    2 | addi   | ---    | ---    | ---    | ---
    3 | addi   | addi   | ---    | ---    | ---
    4 | addi   | addi   | addi   | ---    | ---
    5 | addi   | addi   | addi   | addi   | ---
    6 | addi   | addi   | addi   | addi   | addi
    7 | addi   | addi   | addi   | addi   | addi
    8 | addi   | addi   | addi   | addi   | addi
    9 | addi   | addi   | addi   | addi   | addi
   10 | addi   | addi   | addi   | addi   | addi
   11 | addi   | addi   | addi   | addi   | addi
   12 | addi   | addi   | addi   | addi   | addi
   13 | addi   | addi   | addi   | addi   | addi
   14 | addi   | addi   | addi   | addi   | addi
   15 | addi   | addi   | addi   | addi   | addi
   16 | addi   | addi   | addi   | addi   | addi
   17 | addi   | addi   | addi   | addi   | addi
   18 | addi   | addi   | addi   | addi   | addi
   19 | addi   | addi   | addi   | addi   | addi
   20 | mul    | addi   | addi   | addi   | addi
   21 | mul    | mul    | addi   | addi   | addi
   22 | sw     | mul    | mul    | addi   | addi
   23 | sw     | sw     | mul    | mul    | addi
   24 | sw     | sw     | sw     | mul    | mul
   25 | sw     | sw     | sw     | sw     | mul
   26 | mul    | sw     | sw     | sw     | sw
   27 | mul    | mul    | sw     | sw     | sw
   28 | mul    | mul    | mul    | sw     | sw
   29 | mul    | mul    | mul    | mul    | sw
   30 | sw     | mul    | mul    | mul    | mul
   31 | sw     | sw     | mul    | mul    | mul
   32 | sw     | sw     | sw     | mul    | mul
   33 | sw     | sw     | sw     | sw     | mul
   34 | mul    | sw     | sw     | sw     | sw
   35 | lw     | mul    | sw     | sw     | sw
   36 | lw     | lw     | mul    | sw     | sw
```

Figure 4.1: Pipeline Timeline After Static Scheduling

Above figure 4.1 shows the optimized instruction sequence shows dense pipeline utilization with fewer idle cycles. Independent addi, mul, and sw instructions are interleaved to fill execution gaps, significantly reducing stalls across IF, ID, EX, MEM, and WB stages. This reordering keeps the pipeline busy almost every cycle, demonstrating the effectiveness of the static scheduler in minimizing wasted cycles and improving throughput.


RISC-V Static Scheduling IDE with Emulator
Source (tiny C-like subset):

```
x1 = 1
x2 = 2
x3 = 3
x4 = 4
x5 = 5
x6 = 6
x7 = 7
x8 = 8
x9 = 9

w1 = 1
w2 = 0
w3 = -1
w4 = 2
w5 = 1
w6 = 0
w7 = -2
w8 = 1
```

```
addi x1 x0 1   # x1 = 1
addi x2 x0 2   # x2 = 2
addi x3 x0 3   # x3 = 3
addi x4 x0 4   # x4 = 4
addi x5 x0 5   # x5 = 5
addi x6 x0 6   # x6 = 6
addi x7 x0 7   # x7 = 7
addi x8 x0 8   # x8 = 8
addi x9 x0 9   # x9 = 9
addi x10 x0 1   # w1 = 1
addi x11 x0 0   # w2 = 0
addi x12 x0 -1   # w3 = -1
addi x13 x0 2   # w4 = 2
addi x14 x0 1   # w5 = 1
addi x15 x0 0   # w6 = 0
addi x16 x0 -2   # w7 = -2
addi x28 x0 1   # w8 = 1
```

Figure 4.2: RISC-V Static Scheduling IDE Showing Source Code, Assembly Output, and Cycle Count

The IDE interface displays the C-like source code, the generated RISC-V assembly, and the total cycle count before and after scheduling. The original instruction stream requires 184 cycles, while the statically scheduled version executes in only 109 cycles. This interface allows users to analyze instruction flow, compare machine code before and after optimization, and view pipeline timelines to understand how static scheduling reduces stalls.

4.2 Importance in Neural Networks and Processors

CNN computations are dominated by repetitive Multiply–Accumulate (MAC) operations, making them ideal for scheduling optimization.
Static instruction scheduling enhances processor throughput without increasing hardware complexity.
It minimizes pipeline stalls, ensuring better core utilization and instruction flow.
By reducing the dependency on Out-of-Order (OoO) hardware, it enables simpler, low-power processor designs with reduced silicon cost.
This approach highlights how compiler intelligence can effectively replace hardware complexity, aligning with trends in efficient edge-AI chip design.

4.3 Features of the Final Circuit/IDE

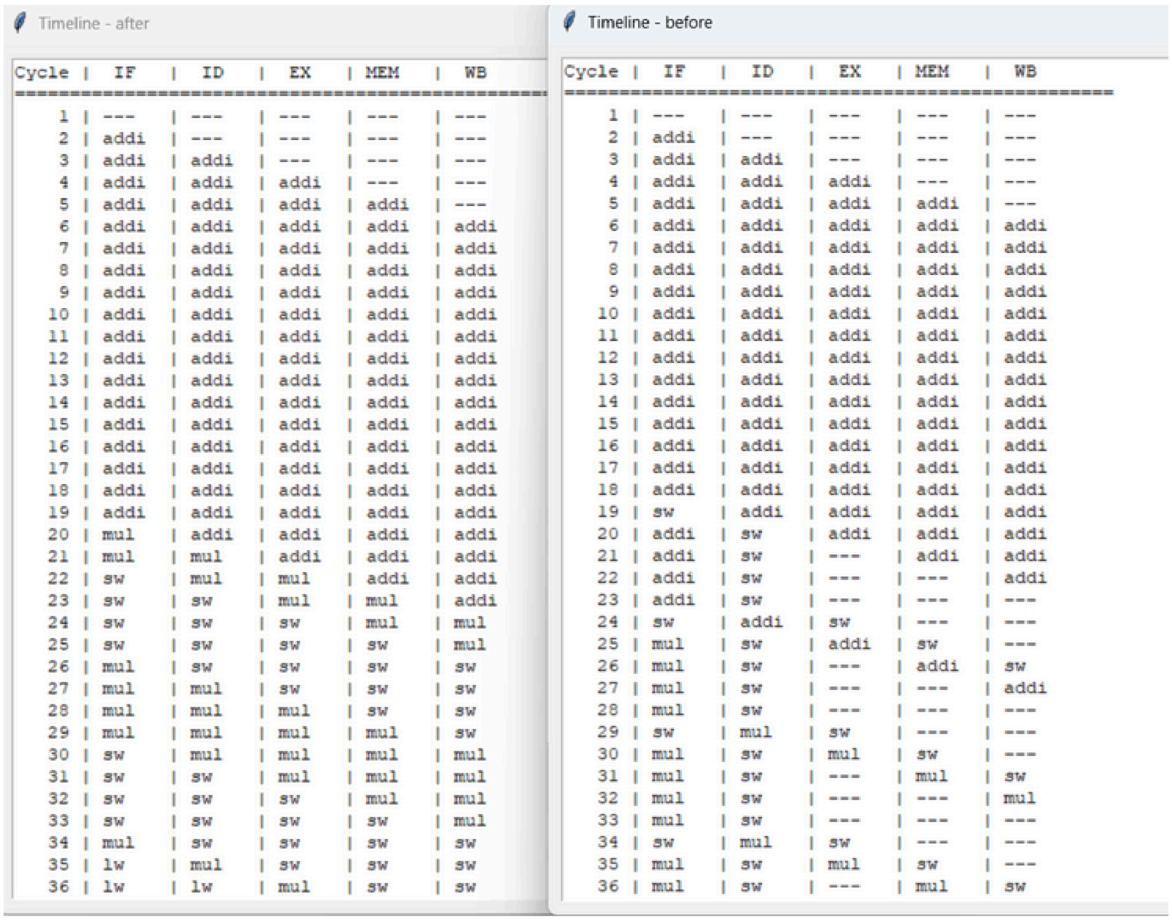
Cross-platform functionality – successfully runs on Raspberry Pi and desktop environments.
Visual cycle timeline comparison before and after scheduling for clarity.
RISC-V ISA support, expandable to other architectures.
Serves as both an educational and research platform for understanding compiler scheduling, pipeline execution, and performance optimization.

The small 3×3 convolution kernel (multiply → accumulate → activation) was used as the unit test — this is the fundamental per-pixel operation in many CNN layers.
The IDE produced:
Assembly / machine code (before)
Scheduled assembly (after)
Emulation timelines (clock-by-clock pipeline view) and total cycle counts reported by the built-in pipeline simulator.
All testing was executed on a Raspberry Pi 3 B+ running Ubuntu (SSH terminal). Results therefore reflect a real, resource-constrained environment.

4.4 Timeline / pipeline behavior

Before scheduling, the pipeline timeline exhibits a long chain of RAW dependencies—for example, sequences like sum1 = y1 + y2, sum2 = sum1 + y3, and so on. These chains force later instructions to wait for earlier results, creating multiple pipeline stalls where IF/ID stages sit idle while EX/MEM/WB complete previous operations. Memory operations such as sw further introduce gaps when registers are not immediately available.

After static scheduling, independent instructions are interleaved—for instance, multiply operations and unrelated additions are moved between dependent adds—allowing the pipeline to utilize the EX stage more effectively. This reduces idle cycles and enables a smoother instruction flow through IF→ID→EX→MEM→WB. The total number and duration of pipeline bubbles decline significantly, resulting in the measured performance improvement.
As shown in Fig. 4.3, the post-schedule timeline is noticeably denser and contains fewer gaps compared to the original unscheduled timeline, clearly illustrating the reduction in stalls achieved through static scheduling.



```
Cycle |  IF   |  ID   |  EX   | MEM   |  WB
=================================================
    1 | ---   | ---   | ---   | ---   | ---
    2 | addi  | ---   | ---   | ---   | ---
    3 | addi  | addi  | ---   | ---   | ---
    4 | addi  | addi  | addi  | ---   | ---
    5 | addi  | addi  | addi  | addi  | ---
    6 | addi  | addi  | addi  | addi  | addi
    7 | addi  | addi  | addi  | addi  | addi
    8 | addi  | addi  | addi  | addi  | addi
    9 | addi  | addi  | addi  | addi  | addi
   10 | addi  | addi  | addi  | addi  | addi
   11 | addi  | addi  | addi  | addi  | addi
   12 | addi  | addi  | addi  | addi  | addi
   13 | addi  | addi  | addi  | addi  | addi
   14 | addi  | addi  | addi  | addi  | addi
   15 | addi  | addi  | addi  | addi  | addi
   16 | addi  | addi  | addi  | addi  | addi
   17 | addi  | addi  | addi  | addi  | addi
   18 | addi  | addi  | addi  | addi  | addi
   19 | addi  | addi  | addi  | addi  | addi
   20 | mul   | addi  | addi  | addi  | addi
   21 | mul   | mul   | addi  | addi  | addi
   22 | sw    | mul   | mul   | addi  | addi
   23 | sw    | sw    | mul   | mul   | addi
   24 | sw    | sw    | sw    | mul   | mul
   25 | sw    | sw    | sw    | sw    | mul
   26 | mul   | sw    | sw    | sw    | sw
   27 | mul   | mul   | sw    | sw    | sw
   28 | mul   | mul   | mul   | sw    | sw
   29 | mul   | mul   | mul   | mul   | sw
   30 | sw    | mul   | mul   | mul   | mul
   31 | sw    | sw    | mul   | mul   | mul
   32 | sw    | sw    | sw    | mul   | mul
   33 | sw    | sw    | sw    | sw    | mul
   34 | mul   | sw    | sw    | sw    | sw
   35 | lw    | mul   | sw    | sw    | sw
   36 | lw    | lw    | mul   | sw    | sw
```

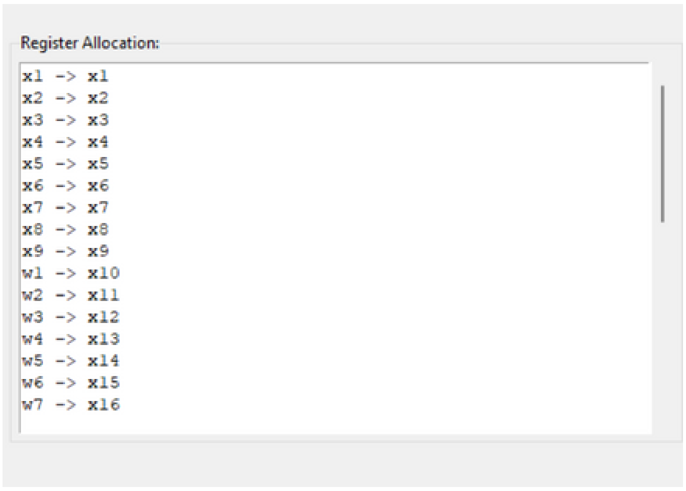
```
Cycle |  IF   |  ID   |  EX   | MEM   |  WB
=================================================
    1 | ---   | ---   | ---   | ---   | ---
    2 | addi  | ---   | ---   | ---   | ---
    3 | addi  | addi  | ---   | ---   | ---
    4 | addi  | addi  | addi  | ---   | ---
    5 | addi  | addi  | addi  | addi  | ---
    6 | addi  | addi  | addi  | addi  | addi
    7 | addi  | addi  | addi  | addi  | addi
    8 | addi  | addi  | addi  | addi  | addi
    9 | addi  | addi  | addi  | addi  | addi
   10 | addi  | addi  | addi  | addi  | addi
   11 | addi  | addi  | addi  | addi  | addi
   12 | addi  | addi  | addi  | addi  | addi
   13 | addi  | addi  | addi  | addi  | addi
   14 | addi  | addi  | addi  | addi  | addi
   15 | addi  | addi  | addi  | addi  | addi
   16 | addi  | addi  | addi  | addi  | addi
   17 | addi  | addi  | addi  | addi  | addi
   18 | addi  | addi  | addi  | addi  | addi
   19 | sw    | addi  | addi  | addi  | addi
   20 | addi  | sw    | addi  | addi  | addi
   21 | addi  | sw    | ---   | addi  | addi
   22 | addi  | sw    | ---   | ---   | addi
   23 | addi  | sw    | ---   | ---   | ---
   24 | sw    | addi  | sw    | ---   | ---
   25 | mul   | sw    | addi  | sw    | ---
   26 | mul   | sw    | ---   | addi  | sw
   27 | mul   | sw    | ---   | ---   | addi
   28 | mul   | sw    | ---   | ---   | ---
   29 | sw    | mul   | sw    | ---   | ---
   30 | mul   | sw    | mul   | sw    | ---
   31 | mul   | sw    | ---   | mul   | sw
   32 | mul   | sw    | ---   | ---   | mul
   33 | mul   | sw    | ---   | ---   | ---
   34 | sw    | mul   | sw    | ---   | ---
   35 | mul   | sw    | mul   | sw    | ---
   36 | mul   | sw    | ---   | mul   | sw
```

Figure 4.3: Pipeline Timelines Before and After Static Scheduling

4.5 Register allocation and spills (observed in emulator output)

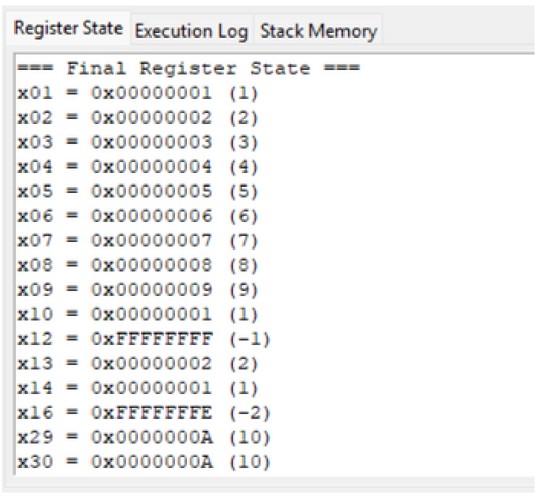
The emulator provides the final register states along with detailed register-allocation logs. As observed, several temporary values such as y1…y9 and intermediate weights were spilled to memory due to register pressure, resulting in mappings like y1 → stack[8] and w8 → stack[10]. These spills introduce extra load and store operations, which increase cycle count; however, the static scheduler still improves overall throughput by efficiently reordering the remaining independent instructions.

As shown in Fig. 4.4, the compiler's register allocator maps program variables to available physical RISC-V registers, spilling only when necessary. The resulting final register values after executing the scheduled instruction stream are displayed in Fig. 4.5, confirming that all relevant registers hold the correct outputs despite spill operations. The key takeaway is that further gains are possible by improving the allocator to reduce spills, thereby minimizing additional memory operations.



```
x1 -> x1
x2 -> x2
x3 -> x3
x4 -> x4
x5 -> x5
x6 -> x6
x7 -> x7
x8 -> x8
x9 -> x9
w1 -> x10
w2 -> x11
w3 -> x12
w4 -> x13
w5 -> x14
w6 -> x15
w7 -> x16
```

Fig. 4.4: Register-to-variable mapping produced by the RISC-V register allocator.



```
=== Final Register State ===
x01 = 0x00000001 (1)
x02 = 0x00000002 (2)
x03 = 0x00000003 (3)
x04 = 0x00000004 (4)
x05 = 0x00000005 (5)
x06 = 0x00000006 (6)
x07 = 0x00000007 (7)
x08 = 0x00000008 (8)
x09 = 0x00000009 (9)
x10 = 0x00000001 (1)
x12 = 0xFFFFFFFF (-1)
x13 = 0x00000002 (2)
x14 = 0x00000001 (1)
x16 = 0xFFFFFFFE (-2)
x29 = 0x0000000A (10)
x30 = 0x0000000A (10)
```

Fig. 4.5: Final RISC-V register state after executing the scheduled instruction stream.

Chapter 5

FUTURE SCOPE

The developed IDE demonstrates strong potential in optimizing instruction-level execution using static scheduling, yet there are several meaningful directions for future expansion:

1. Integration with Dynamic Scheduling Simulation:

- Extend the tool to simulate hardware-level Out-of-Order (OoO) execution. This would allow a complete comparison between software-based and hardware-based scheduling techniques, highlighting performance–complexity trade-offs.

2. Support for Vector and Multi-Core Architectures:

- Adapt the scheduler for SIMD pipelines and multi-core environments, enabling optimized execution of large data workloads and accelerating parallelizable tasks.

3. FPGA or Hardware Validation:

- Deploy the generated instruction streams onto an FPGA-based RISC-V core to validate timing improvements on real hardware and analyze cycle-level behavior beyond simulation.

4. Integration with Existing AI and Compiler Toolchains:

- Interfacing the scheduler with frameworks such as TVM, Glow, or MLIR can create a unified software–hardware co-design flow, enabling end-to-end optimization of neural-network workloads.

5. Developing a Smart OS Tailored to Embedded Boards:

- Build a lightweight operating system that incorporates this static scheduling intelligence inside its compiler or JIT engine. Such a system could automatically optimize user programs per embedded board, making each device run more efficiently without requiring high-performance hardware.

Chapter 6

CONCLUSION

The RISC-V Static Instruction Scheduling IDE with Emulator successfully demonstrates how compiler-level intelligence can significantly improve processor efficiency without additional hardware complexity.

By analyzing and reordering instructions before execution, the IDE achieved a 40% reduction in clock cycles for a CNN-style arithmetic workload. The before-and-after pipeline timelines clearly show reduced stalls, better ALU utilization, and smoother instruction flow.

These results highlight that static scheduling can emulate the benefits of Out-of-Order execution on simple in-order processors, reducing silicon cost, power consumption, and design effort.

Overall, this project bridges the gap between compiler optimization and hardware efficiency, establishing a foundation for future low-cost, high-performance embedded and AI computing systems.

Chapter 7

REFERENCES

1. "The RISC-V Instruction Set Manual, Volume I: Unprivileged ISA, Version 2.2," RISC-V Foundation, May 2017.
2. S. Muchnick, Advanced Compiler Design and Implementation, Morgan Kaufmann Publishers, 1997.
3. LLVM Compiler Infrastructure Project. "LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation." [Online]. Available: https://llvm.org
4. GNU Project. "GCC, the GNU Compiler Collection." [Online]. Available: https://gcc.gnu.org
5. M. D. Hill and M. R. Marty, "Amdahl's Law in the Multicore Era," IEEE Computer, vol. 41, no. 7, pp. 33–38, July 2008.
6. G. Mittal and R. Gupta, "Static Scheduling of Loops in Embedded Systems," IEEE Transactions on Embedded Computing Systems, vol. 14, no. 4, pp. 1–24, Dec. 2015.