Faculty of Engineering

A Complete Toolchain for the RWU-RV64I

Submitted in partial fulfillment of the requirements for the degree of Master of Science at Hochschule Ravensburg-Weingarten.

Author:	**Abhishek Abhishek**
Matriculation No.:	36136
Degree:	Master of Science
Program:	Electrical Engineering and Embedded Systems

Supervisors:

Prof. Dr.-Ing. Andreas Siggelkow

Prof. Dr. rer. nat. Markus Pfeil

Weingarten, Germany

2025

Declaration of Originality

I, **Abhishek Abhishek**, hereby declare that this master thesis, entitled "*A Complete Toolchain for the RWU-RV64I*", is entirely my own work and has not been submitted in any form for any other academic degree or professional qualification.

All the work presented in this thesis, including research, implementation, analysis, and documentation, has been carried out independently under the supervision of **Prof. Dr.-Ing. Andreas Siggelkow** and **Prof. Dr. rer. nat. Markus Pfeil**. All sources of information and assistance used in the preparation of this work have been properly acknowledged and cited.

I attest that all sources used for information, data, or ideas have been properly referenced and cited in accordance with the established academic conventions and guidelines. Any materials or concepts obtained from other works are duly credited.

By signing below, I confirm my understanding of and adherence to the principles of academic integrity and the ethical standards for conducting research.

Place: Weingarten, Germany **Date:**

Signature:

Name: Abhishek Abhishek

Abstract

The rapid evolution of open-source processor architectures, particularly RISC-V, has enabled academic institutions and research groups to explore customized hardware software co-designs. This thesis presents the development of a **complete software toolchain** for the **RWU-RV64I**, a 64-bit RISC-V based in-house processor developed at Hochschule Ravensburg-Weingarten. The work focuses on establishing a professional, extensible, and automated development environment that bridges the gap between hardware implementation and software programmability for the RWU-RV64I core.

The project begins with a detailed study of the existing prototype environment, which comprised a basic Makefile-based build system with limited support for assembly-level testing. To advance beyond this stage, the toolchain was redesigned and implemented around the **GNU Compiler Collection (GCC)** and a **custom linker script and startup code** tailored for the RWU-RV64I memory map and hardware configuration. The integration ensures that standard C programs can be compiled, linked, and executed directly on the custom processor without modification of its RTL design.

The thesis further introduces a **structured build framework** using **Make**, providing a modular and portable foundation for future development. This framework automates compilation, linking, and binary generation, ensuring reproducibility and ease of use for both academic research and student projects. Additionally, the implementation of test programs and GPIO verification routines demonstrates the functional correctness of the toolchain in simulation environments. The toolchain was also extended to support **FPGA-based deployment** on the **Zybo Zynq-7000** board, enabling real hardware validation.

Comprehensive testing confirmed successful compilation and execution of bare-metal C programs, verifying correct instruction decoding, memory mapping, and I/O control through GPIO signals. The developed infrastructure not only simplifies software development for the RWU-RV64I but also serves as a scalable foundation for future enhancements such as interrupt handling, UART communication, and operating system porting.

Overall, this thesis contributes a fully functional, reproducible, and extensible toolchain for the RWU-RV64I processor, providing a complete workflow from source code to hardware

execution. It strengthens the processor's usability as a research and educational platform and establishes a solid basis for ongoing developments in RISC-V–based embedded systems at Hochschule Ravensburg-Weingarten.

Acknowledgements

I would like to express my sincere gratitude to my supervisor, **Prof. Dr.-Ing. Andreas Siggelkow**, for his invaluable guidance, technical expertise, and continuous support throughout the course of this thesis. His clear vision, constructive feedback, and encouragement have been instrumental in shaping the direction of this work and deepening my understanding of embedded systems and processor toolchain development.

I am also deeply thankful to **Prof. Dr. rer. nat. Markus Pfeil** for his insightful discussions, thoughtful suggestions, and academic mentorship, which greatly contributed to improving the quality and clarity of this research.

Weingarten, Germany *2025*

Contents

Declaration of Originality i

Abstract i

Acknowledgements i

1 Introduction 1

1.1 Motivation 1

1.2 Problem Statement and Scope 2

1.3 Objectives 3

2 Background and Related Work 4

2.1 RISC-V RV64I Architecture 4

2.1.1 Registers and Data Model 4

2.1.2 Calling Convention and Program Flow 5

2.1.3 Instruction Formats and Execution Flow 5

2.2 RISC-V RV64I Instruction Set 5

2.2.1 Instruction Categories 6

2.2.2 Arithmetic and Logical Instructions 6

2.2.3 Immediate Instructions 6

2.2.4 Memory Access Instructions 7

2.2.5 Control Flow Instructions 7

2.2.6 System and Miscellaneous Instructions 7

2.3 Toolchain Overview 8

2.3.1 From C Source Code to Executable Machine Code 8

2.3.2 Compilation Stages 8

2.3.3 Relevance to RWU-RV64I Toolchain . . . 10
2.3.4 Compiler and Assembler . . . 10
2.3.5 Linker and Object Utilities . . . 10
2.3.6 Startup and Initialization (crt0.s) . . . 10
2.3.7 Build Automation and Workflow . . . 11
2.4 Program Size Optimization Techniques . . . 11
2.4.1 Compiler Optimization Levels . . . 12
2.4.2 Section Placement and Symbol Stripping . . . 12
2.4.3 Harvard Architecture Implications . . . 12
2.5 Related Work . . . 12
2.6 Summary . . . 13

3 System Overview **14**
3.1 Introduction . . . 14
3.2 Top-Level Architecture . . . 14
3.3 CPU Core Architecture . . . 16
3.3.1 Datapath Components . . . 16
3.3.2 Control Flow . . . 17
3.4 Memory Organization . . . 17
3.4.1 Instruction Memory (IMEM) . . . 17
3.4.2 Data Memory (DMEM) . . . 18
3.4.3 Memory Map Overview . . . 19
3.5 Peripherals . . . 20
3.5.1 General-Purpose I/O (GPIO) . . . 20
3.5.2 Clock Generation Unit (CGU) . . . 20
3.6 Target Hardware Platform . . . 20
3.7 Simulation and Verification Environment . . . 21
3.8 Summary . . . 21

4 Toolchain Implementation **22**
4.1 Overview . . . 22
4.2 Installing the GNU RISC-V Toolchain on Linux . . . 22
4.2.1 System Requirements . . . 23

4.2.2 Toolchain Source Installation 23
4.3 Toolchain Configuration 24
4.4 Hardware Context and Memory Architecture 25
4.5 Linker Script Development 25
4.6 Startup File (crt0.s) 27
4.7 Runtime Header File (rwu-rv64i.h) 28
4.8 Summary 30

5 Eclipse Integration and Testing **31**
5.1 Overview 31
5.2 Software Requirements 31
5.3 Creating and Importing the Project in Eclipse 32
5.3.1 Creating a Makefile Project 32
5.3.2 Disabling Automatic Makefile Generation 32
5.4 Build Environment Configuration 33
5.4.1 Setting the Compiler Path 33
5.4.2 Selecting the External Builder 33
5.4.3 Defining Make Targets 34
5.5 Makefile Build Process 34
5.5.1 Compilation and Assembly 35
5.5.2 Linking and Memory Image Generation 35
5.6 Simulation and XSIM Integration 35
5.7 Cleaning and Reproducibility 36
5.7.1 Vivado Project Setup 36
5.7.2 Synthesis, Implementation and Bitstream 36
5.7.3 Programming and Running on Board 37
5.7.4 Integration correctness 37
5.8 Summary 38

6 Compiler optimizations: behaviour, effects and recommendations **39**
6.1 Introduction 39
6.2 Goals and Scope 39
6.3 Experimental setup 40

6.3.1 Toolchain and build commands . . . 40
6.3.2 Runtime and verification infrastructure . . . 40
6.3.3 Programs under test . . . 41
6.3.4 Measurements . . . 41
6.4 Results . . . 43
6.5 Analysis: root causes and mechanisms . . . 43
6.5.1 Compiler helper calls and freestanding builds . . . 43
6.5.2 Inlining, dead-code elimination and code density . . . 43
6.5.3 Loop optimizations and strength reduction . . . 44
6.5.4 Switch-case codegen: branch chain vs jump table . . . 44
6.5.5 Testbench observability and timing sensitivity . . . 44
6.6 Experimental verification and hardware output . . . 44
6.6.1 Purpose of the test . . . 45
6.6.2 Conditional test program . . . 45
6.6.3 Build process confirmation . . . 46
6.6.4 Simulation verification in XSIM . . . 47
6.6.5 Waveform analysis . . . 48
6.6.6 FPGA hardware execution . . . 48
6.6.7 Discussion . . . 49
6.7 Summary . . . 50

7 Conclusion and Future Work **51**
7.1 Conclusion . . . 51
7.2 Future Work . . . 52
7.3 Summary . . . 54

A Toolchain Source Listings **55**
A.1 Linker Script (`linker.ld`) . . . 55
A.2 Startup Assembly File (`crt0.s`) . . . 57
A.3 Runtime Header File (`rwu-rv64i.h`) . . . 58
A.4 Build Makefile (embedded) . . . 60

B Optimization Test Programs **65**

List of Figures

2.1 Transformation of a C program into machine-executable code. 9

3.1 Block-level organization of the RWU-RV64I system. 15
3.2 Datapath and control flow inside the RWU-RV64I CPU core. 16
3.3 Memory and peripheral map of the RWU-RV64I. 19

5.1 Eclipse project configuration showing external builder settings. 33
5.2 Make Targets view showing `sim`, `simclean`, `waves` targets. 34
5.3 Vivado project manager, project summary and source tree. 37

6.1 Successful build of the conditional test program using the RWU-RV64I GCC toolchain in Eclipse CDT. 47
6.2 XSIM simulation console showing correct conditional result and successful program termination. 47
6.3 XSIM waveform showing GPIO output (`gpio_s[7:0]` = 0x0F) for the satisfied branch condition. 48
6.4 FPGA hardware output on the Zybo board showing all four LEDs ON (`GPIO = 0x0F`). 49

List of Tables

3.1 Memory map and peripheral address ranges. 19

3.2 Example CGU division settings. 20

4.1 RWU-RV64I memory configuration used in the toolchain. 25

5.1 Key device utilization numbers from Vivado synthesis report. 36

6.1 `opt_workload.c`: Code size and simulation outcome vs. optimization level 42

6.2 Square test (`test.c`): code size and outcome vs. optimization level . . . 42

6.3 test_simple_gpio.c: code size and outcome vs. optimization level 42

Chapter 1

Introduction

1.1 Motivation

The increasing adoption of open-source Instruction Set Architectures (ISAs), such as RISC-V, has significantly transformed the landscape of embedded system and processor development. Unlike proprietary ISAs, RISC-V provides a modular and license-free foundation that encourages academic and industrial innovation in processor design, compiler construction, and system-level integration [11]. As a result, universities and research institutes can design, implement, and study customized processors while maintaining compatibility with standardized software tools.

The RWU-RV64I processor, developed at the Hochschule Ravensburg-Weingarten, is a 64-bit implementation of the base RISC-V integer instruction set (RV64I). It serves as an educational and research platform for studying microarchitecture design, hardware–software co-design, and compiler interaction. However, the processor's practical usability depends strongly on the availability of a reliable software toolchain. Without a well-integrated compiler, assembler, linker, and debugging environment, processor programming is limited to low-level assembly testing, restricting scalability and educational use.

Developing a complete and robust toolchain for the RWU-RV64I is therefore essential to enable C and C++ application development, facilitate efficient testing, and improve productivity in hardware verification and embedded systems research. The toolchain forms the central interface between software developers and hardware designers, bridging the gap between algorithmic implementation and instruction-level execution on the target

core. Moreover, a standardized build process and integrated development environment allow future students and researchers to rapidly develop, compile, and test programs on the RWU-RV64I, fostering a sustainable development ecosystem within the university.

1.2 Problem Statement and Scope

The existing RWU-RV64I environment provides only a minimal `Makefile`-based setup with limited support for assembly-level test programs. While this setup suffices for verifying basic instruction functionality, it lacks the capabilities required for structured C program development and systematic evaluation of compiler optimizations. Furthermore, the absence of integration with modern development environments significantly restricts the usability, scalability, and maintainability of the system.

This thesis therefore focuses on designing and implementing a complete software toolchain for the RWU-RV64I processor that enables seamless compilation, linking, and execution of C programs on the target platform. Specifically, the work covers the following aspects:

- Configuration and integration of the compiler, assembler, and linker components from the GNU toolchain (GCC and Binutils) for the RV64I architecture [3].
- Development of a customized linker script and startup code adapted to the RWU-RV64I memory architecture and hardware configuration.
- Integration of the build environment into Eclipse to provide a user-friendly graphical interface for software development.
- Deployment and verification of the compiled binaries on the Zybo Zynq-7000 FPGA board to validate the RWU-RV64I core operation in hardware.
- Evaluation of program size and performance across different compiler optimization levels (`-O0`, `-O1`, `-O2`, `-O3`, `-Os`, and `-Ofast`).

It is important to note that this thesis does not include the design or modification of the processor's RTL description, nor the development of a dedicated program loader. The scope is restricted to software-level development and integration, assuming that the RWU-RV64I processor and its memory interfaces are available and functionally verified.

Furthermore, the toolchain targets a *bare-metal* environment and therefore excludes any operating system or standard C library such as Newlib. All compiled programs are directly linked to the hardware memory map and executed without runtime dependencies.

1.3 Objectives

The main objective of this thesis is to establish a complete and functional software development environment for the RWU-RV64I processor. The defined scope forms the foundation for the following objectives, which translate the identified problems into concrete implementation goals.

The specific technical goals are summarized as follows:

- Implement and configure the cross-compiler and associated binary utilities for the RV64I architecture.
- Create startup code (`crt0.s`) and a linker script tailored to the RWU-RV64I memory map and peripheral configuration.
- Integrate the toolchain with a structured build system and provide Eclipse IDE support for program compilation and debugging.
- Deploy and validate the compiled programs on the Zybo Zynq-7000 FPGA board to verify the functional operation of the RWU-RV64I core in hardware.
- Evaluate and compare program size across various compiler optimization levels (`-O0`, `-O1`, `-O2`, `-O3`, `-Os`, and `-Ofast`).

The successful completion of these objectives will result in a reproducible, maintainable, and extensible development workflow that allows C programs to be built, tested, and executed efficiently on the RWU-RV64I processor—both in simulation and on FPGA-based hardware platforms.

Chapter 2

Background and Related Work

2.1 RISC-V RV64I Architecture

The RISC-V instruction set architecture (ISA) is an open and modular standard designed to promote extensibility, simplicity, and freedom from licensing restrictions [11]. The base 64-bit subset, known as RV64I, defines a general-purpose architecture supporting 64-bit integer arithmetic and memory addressing. It serves as the foundation for all higher-level RISC-V extensions and implementations.

2.1.1 Registers and Data Model

The RV64I ISA defines 32 general-purpose registers (`x0-x31`), each 64 bits wide. Register `x0` is hardwired to zero, while `x1` (`ra`) holds the return address. Registers `x2` (`sp`) and `x3` (`gp`) serve as the stack pointer and global pointer, respectively. The remaining registers are divided between temporary (`t0-t6`), saved (`s0-s11`), and argument (`a0-a7`) registers. The architecture uses a little-endian data model and supports naturally aligned accesses to 8-, 16-, 32-, and 64-bit data.

The program counter (`pc`) is a 64-bit register that stores the address of the next instruction. Each instruction is 32 bits wide and aligned to a 4-byte boundary, simplifying instruction fetch and decode stages in hardware.

2.1.2 Calling Convention and Program Flow

The RWU-RV64I processor adheres to the standard RISC-V calling convention. Function arguments are passed through registers `a0-a7`, and return values are stored in `a0`. The stack grows downward from the top of data memory, and all functions preserve the values of callee-saved registers (`s0-s11`). After a system reset, the processor's program counter starts at address `0x00000000`, where the startup code (`_start`) resides. This startup routine initializes memory and transfers control to the user's `main()` function.

2.1.3 Instruction Formats and Execution Flow

RISC-V instructions follow six fundamental encoding formats—R, I, S, B, U, and J—each specifying a unique arrangement of opcode, register indices, and immediate fields [11]. This regular structure provides a compact and consistent encoding scheme, simplifying both hardware decoding and compiler implementation. The details of these formats and their corresponding instruction classes are described in Section 2.2.

During execution, the RWU-RV64I processor fetches instructions sequentially from instruction memory (IMEM), decodes them through its control unit, and performs arithmetic or memory operations using the arithmetic logic unit (ALU) and data memory (DMEM). The architecture follows a Harvard memory model, ensuring that instruction fetch and data access can occur independently and concurrently within a single clock cycle.

2.2 RISC-V RV64I Instruction Set

The RV64I instruction set constitutes the base integer subset of the 64-bit RISC-V architecture. It defines the minimal set of operations required for general-purpose computing, forming the foundation upon which all other RISC-V extensions (such as M, A, F, D, and C) are built [11, 6]. Each instruction is encoded in a fixed 32-bit format, promoting implementation simplicity and reducing hardware decoding complexity. The instruction set supports a load–store architecture, where arithmetic and logical operations act exclusively on registers, while memory access is handled through dedicated load and store instructions.

2.2.1 Instruction Categories

The RV64I ISA consists of six primary instruction formats—R, I, S, B, U, and J—each defining specific bit fields for opcode, register indices, and immediates [11]. These formats enable a consistent and compact encoding scheme across all instruction types.

- **R-type:** Register-register arithmetic and logical operations (e.g., `ADD`, `SUB`, `AND`, `OR`, `XOR`, `SLL`, `SRL`, `SRA`).
- **I-type:** Immediate arithmetic, logical, and load instructions (e.g., `ADDI`, `SLTI`, `ANDI`, `LW`, `LD`).
- **S-type:** Store instructions (e.g., `SB`, `SH`, `SW`, `SD`).
- **B-type:** Conditional branch instructions (e.g., `BEQ`, `BNE`, `BLT`, `BGE`).
- **U-type:** Upper immediate operations (e.g., `LUI`, `AUIPC`).
- **J-type:** Unconditional jump instructions (e.g., `JAL`).

2.2.2 Arithmetic and Logical Instructions

These operations form the computational core of the architecture. They perform integer addition, subtraction, comparison, and bitwise logic. Both signed and unsigned variants are provided for relational operations [11].

- `ADD`, `SUB` – integer addition and subtraction.
- `SLT`, `SLTU` – set-on-less-than (signed/unsigned).
- `AND`, `OR`, `XOR` – bitwise logical operations.
- `SLL`, `SRL`, `SRA` – logical and arithmetic shift operations.

2.2.3 Immediate Instructions

Immediate instructions allow operations with constants embedded directly within the instruction word. These reduce memory accesses and simplify small arithmetic manipulations.

- `ADDI`, `SLTI`, `ANDI`, `ORI`, `XORI` – arithmetic and logical operations with immediate values.
- `LUI`, `AUIPC` – load and add upper immediate values for address generation.

2.2.4 Memory Access Instructions

Load and store instructions transfer data between registers and memory. The effective address is computed by adding an immediate offset to a base register. All accesses must be naturally aligned to the data size [11].

- **Load:** `LB`, `LH`, `LW`, `LD`, `LBU`, `LHU`, `LWU`.
- **Store:** `SB`, `SH`, `SW`, `SD`.

2.2.5 Control Flow Instructions

These instructions alter the normal sequential execution of instructions by modifying the program counter (`pc`).

- **Conditional branches:** `BEQ`, `BNE`, `BLT`, `BGE`, `BLTU`, `BGEU`.
- **Unconditional jumps:** `JAL`, `JALR`.

Branch targets are computed relative to the current `pc`, while jumps save return addresses in register `ra` (`x1`) to support subroutine calls.

2.2.6 System and Miscellaneous Instructions

The base instruction set also defines control and synchronization instructions that support debugging and environment interactions [11].

- `ECALL` – environment call or system call request.
- `EBREAK` – breakpoint trigger for debugging.
- `FENCE`, `FENCE.I` – ensure memory and instruction ordering consistency.

The RV64I base instruction set comprises approximately 47 unique operations, providing a compact yet complete framework for software execution. Its clean orthogonal design enables straightforward hardware implementation, compiler support, and educational experimentation [6]. For the RWU-RV64I processor, this subset forms the executable foundation used in all simulation and FPGA-based validation experiments presented in this thesis.

2.3 Toolchain Overview

A software toolchain converts source code into executable machine code suitable for the target processor. The RWU-RV64I toolchain is a bare-metal environment composed of the GNU compiler and binary utilities. It omits any runtime library (such as Newlib or glibc), ensuring direct execution on the hardware without operating system support. The GNU toolchain provides an open-source, modular, and well-established framework for software development across multiple architectures, including RISC-V [3, 2]. It consists of the GNU Compiler Collection (GCC), GNU Binutils, and related utilities such as the assembler, linker, and `objcopy`. Together, these tools form the foundation of most embedded software build environments.

2.3.1 From C Source Code to Executable Machine Code

Modern embedded software development relies on a toolchain that transforms high-level source code into binary instructions executable by the target processor. This transformation process involves several stages—each handled by a dedicated component of the GNU toolchain [3]. Although this flow is conceptually similar across platforms, it is particularly critical for cross-compilation environments such as the RWU-RV64I toolchain, where programs are compiled on a host PC but executed on a different architecture.

2.3.2 Compilation Stages

When a C program is built, the GNU compiler toolchain performs a series of well-defined steps. Each stage produces an intermediate representation that serves as input to the next stage, as show in figure 2.1.

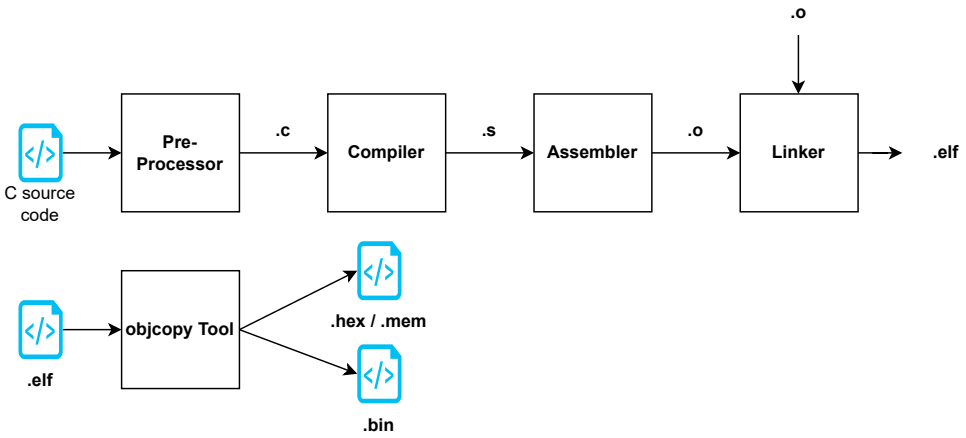

Figure 2.1: Transformation of a C program into machine-executable code.

1. **Preprocessing:** The C preprocessor (`cpp`) expands all macros, evaluates conditional compilation directives, and includes header files. The result is a pure C file with no preprocessor directives.

2. **Compilation:** The compiler (`gcc`) parses the preprocessed C source and translates it into assembly code for the target instruction set (in this case, RV64I). This stage performs syntax analysis, optimization, and instruction selection based on the chosen compiler flags (`-O0`–`-Os`).

3. **Assembly:** The assembler (`as`) converts the assembly file into a relocatable object file (`.o`). Each object file contains machine instructions and symbol tables describing functions, variables, and section addresses.

4. **Linking:** The linker (`ld`) combines one or more object files and applies the linker script (`linker.ld`) to assign final memory addresses. It resolves external references and produces an executable file in the ELF (Executable and Linkable Format) structure.

5. **Binary Conversion:** The object copy utility (`objcopy`) extracts the executable sections from the ELF file and converts them into a flat binary or Verilog hex file. For RWU-RV64I, this step produces the `riscvtest.mem` file that can be directly loaded into the instruction memory of the FPGA or simulation model.

2.3.3 Relevance to RWU-RV64I Toolchain

The RWU-RV64I toolchain replicates this entire process through a single `Makefile`. Each build automatically runs preprocessing, compilation, assembly, linking, and binary conversion, followed by copying the generated memory file into the simulation directory. This automation guarantees that the machine code loaded into the RWU-RV64I instruction memory is always synchronized with the C source code and corresponding linker configuration.

In summary, the process of converting high-level C code into executable instructions ensures that the software toolchain bridges the gap between algorithmic design and hardware execution on the RWU-RV64I platform.

2.3.4 Compiler and Assembler

The GNU Compiler Collection (GCC) translates C source code into assembly instructions compatible with the target processor architecture. For RISC-V, the cross-compiler is typically provided as `riscv64-unknown-elf-gcc`, which produces code conforming to the 64-bit base integer instruction set (RV64I). The assembler, part of the GNU Binutils package, then converts this assembly code into relocatable object files ready for linking.

2.3.5 Linker and Object Utilities

The GNU linker (`ld`) combines object files according to a user-defined memory layout specified in a linker script. This script defines how program sections—such as `.text`, `.data`, and `.bss`—are placed within the target's physical or simulated memory. Additional utilities such as `objcopy` and `objdump` support binary conversion and inspection, allowing executable images to be adapted for FPGA or simulation-based environments.

The GNU toolchain is widely used in both academia and industry for RISC-V development due to its flexibility, open-source nature, and strong integration with modern IDEs such as Eclipse [1, 7].

2.3.6 Startup and Initialization (crt0.s)

The startup file (`crt0.s`) provides the software equivalent of a reset handler. When the RWU-RV64I core resets, the program counter begins execution at address `0x0`, where

`_start` resides. The startup routine performs the following steps:

1. Loads the address of `__stack_top` and sets the stack pointer (`sp`).
2. Clears the `.bss` section using store instructions.
3. Calls the user-defined `main()` function.
4. Enters an infinite loop if `main()` returns.

This structure ensures deterministic initialization of memory and registers, comparable in purpose to ARM Cortex-M reset handlers but tailored for the RWU-RV64I's simpler reset model.

2.3.7 Build Automation and Workflow

The firmware build process is automated using a unified `Makefile`. It automatically detects the single C source file, compiles it, links it with `crt0.s`, and generates all simulation artifacts. The primary build steps are:

1. Compilation of C source files into object files.
2. Linking and generation of the ELF executable.
3. Conversion of ELF into a Verilog memory image (`.v`) and binary file (`.bin`).
4. Automatic copying of the memory image to the simulation directory for RTL testing.

The toolchain also supports optional waveform simulation and debugging using Vivado's `xsim`, allowing verification of instruction flow and GPIO behavior at the signal level.

2.4 Program Size Optimization Techniques

Embedded systems often operate under stringent memory and performance constraints. For the RWU-RV64I, only 32 KiB of instruction memory and 8 KiB of data memory are available, making efficient code generation essential.

2.4.1 Compiler Optimization Levels

GCC provides several optimization levels that balance compilation time, execution speed, and code size [3]. Common options include:

- `-O0`: No optimization; preserves direct mapping from source to assembly for debugging.
- `-O1`: Enables basic optimizations like constant propagation and dead code elimination.
- `-O2`: Performs aggressive loop and inlining optimizations for performance.
- `-Os`: Optimizes for minimal code size while retaining core performance.

In this thesis, the size impact of each level is evaluated using `riscv64-unknown-elf-size`.

2.4.2 Section Placement and Symbol Stripping

Further reduction in program size can be achieved by stripping unused symbols and debug information using `objcopy` or `strip`. Additionally, the linker script arranges code and data in memory so that read-only sections reside in IMEM, while writable data and stack occupy DMEM. This deliberate separation maximizes usable space and prevents overlap between instruction and data spaces.

2.4.3 Harvard Architecture Implications

The RWU-RV64I implements a strict Harvard architecture where instruction and data memories are physically distinct. As a result, the startup code avoids data relocation or copying from IMEM to DMEM. All global and static variables are zero-initialized at runtime, and function calls operate directly within the Harvard memory model. This separation simplifies memory management, eliminates self-modifying code, and aligns with standard embedded RISC-V design practices.

2.5 Related Work

Numerous research efforts have addressed RISC-V toolchain adaptation and educational core development. Waterman and Asanović [11] first outlined the modularity and openness

of RISC-V, inspiring subsequent academic implementations. SiFive and lowRISC provide reference toolchains that rely on standard libraries such as Newlib [5], whereas the RWU-RV64I setup demonstrates a minimalistic, bare-metal configuration tailored for teaching and FPGA experimentation. Similar minimal toolchains have been explored in open-source cores like PicoRV32 and VexRiscv, but most target 32-bit designs with smaller address spaces. By contrast, the RWU-RV64I toolchain focuses on a 64-bit environment and precise control over linker and startup behavior, which are often abstracted away in more complex systems.

2.6 Summary

This chapter presented the theoretical and practical background relevant to the RWU-RV64I project. It explained the RISC-V RV64I ISA, its programming model, and the components of the bare-metal toolchain. Additionally, it described size optimization techniques and summarized related efforts in RISC-V education and research. The following chapter provides a detailed overview of the RWU-RV64I hardware system, including its memory architecture, peripherals, and simulation environment.

Chapter 3

System Overview

3.1 Introduction

This chapter presents a detailed overview of the RWU-RV64I processor system that serves as the target platform for the toolchain developed in this work. The design originates from the open repository maintained by Prof. Dr.-Ing. Andreas Siggelkow at Hochschule Ravensburg-Weingarten [9]. It implements the base RISC-V 64-bit integer instruction set (RV64I) [11] in a compact, single-cycle, Harvard-architecture processor optimized for educational use and toolchain research.

The chapter first outlines the top-level architecture and module hierarchy, followed by the memory organization, peripheral mapping, and clock structure. Finally, the hardware–software integration and simulation environment are discussed.

3.2 Top-Level Architecture

The RWU-RV64I is organized as a modular SoC-style architecture composed of a 64-bit RISC-V CPU core, instruction and data memories, peripheral interfaces, and a JTAG-based debug unit. All components are interconnected via a pair of 64-bit system buses — the *Instruction Bus (I-Bus)* and *Data Bus (D-Bus)* — forming a Harvard architecture that allows simultaneous instruction fetch and data access. The block diagram of the RWU-RV64I processor, shown in Figure 3.1, illustrates the overall architectural organization, highlighting the key components and their interconnections within the processor design.

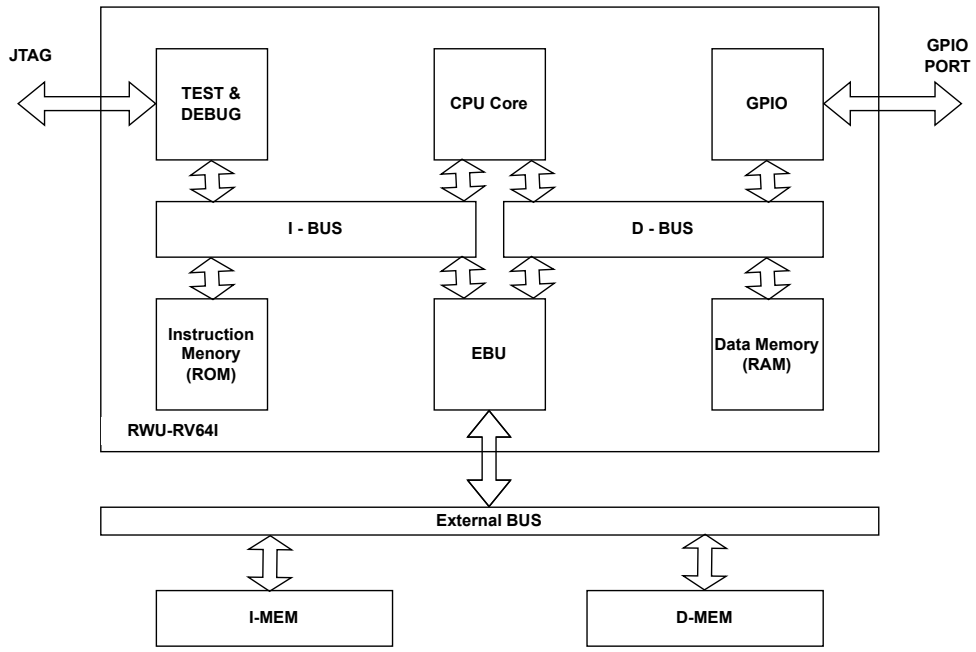

Figure 3.1: Block-level organization of the RWU-RV64I system.

At the top level, the RWU-RV64I comprises the following modules:

- **CPU Core:** Implements the RV64I instruction set and forms the computational heart of the processor. It includes the register file, ALU, immediate generator, control unit, and program counter logic.
- **Instruction Memory (IMEM):** Stores program instructions and is accessed via the I-Bus.
- **Data Memory (DMEM):** Stores data, variables, and stack content, accessed through the D-Bus.
- **GPIO:** Provides simple 8-bit general-purpose I/O for external interfacing and program verification.
- **External Bus Unit (EBU):** Extends connectivity to external memory or I/O devices beyond the FPGA.
- **Test & Debug Unit:** Implements JTAG (IEEE 1149.1) for boundary scan and on-chip debugging. The JTAG unit is used as a program loader for the Processor.

3.3 CPU Core Architecture

The internal structure of the CPU core is illustrated in Figure 3.2. It contains all fundamental building blocks required for the execution of RV64I instructions.

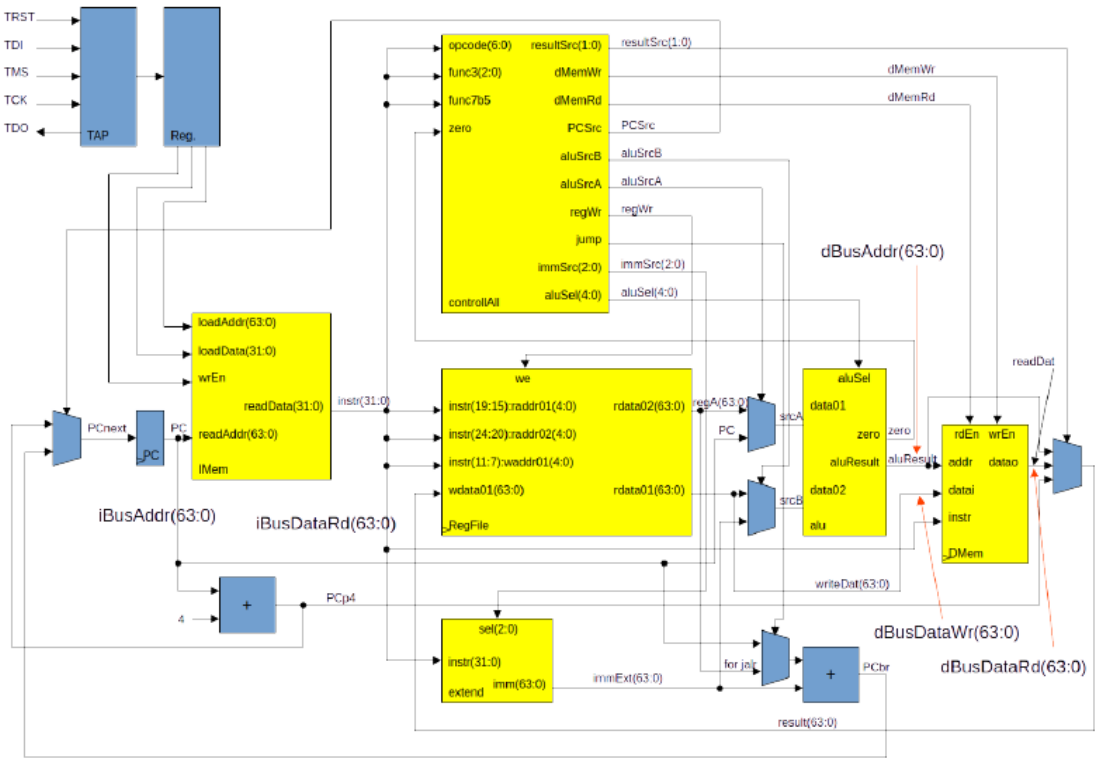

Figure 3.2: Datapath and control flow inside the RWU-RV64I CPU core.

The CPU core executes one instruction per clock cycle — fetch, decode, execute, memory access, and write-back occur sequentially within the same cycle. This single-cycle design reduces hardware complexity and ensures deterministic timing behavior, which is particularly useful for toolchain development and low-level software debugging.

3.3.1 Datapath Components

- **Program Counter (PC):** Holds the current instruction address; increments sequentially or updates based on branch/jump logic.
- **Instruction Decode:** Extracts opcode, register fields, and immediate types for control signal generation.
- **Register File:** Contains 32 × 64-bit registers (`x0-x31`); supports dual read and single write.

- **Immediate Generator:** Extends instruction immediates to 64-bit width for ALU operations.
- **ALU:** Performs integer arithmetic, logical, and shift operations. Results are fed to the memory interface or write-back path.
- **Control Unit:** Generates control signals (`aluSrcA`, `aluSrcB`, `regWr`, `dMemWr`, `dMemRd`, etc.) to coordinate datapath elements.
- **JTAG TAP Controller:** Allows programmer to load the code to the instruction memory of the processor via `TCK`, `TDI`, `TDO`, `TMS`, and `TRST`.

3.3.2 Control Flow

At every clock edge, the instruction is fetched from IMEM, decoded, and executed combinationally by the ALU. If the instruction involves memory access, the calculated address is forwarded to the data bus. After operation completion, the result is written back to the register file in the same clock cycle.

3.4 Memory Organization

The RWU-RV64I adopts a strict Harvard architecture, using physically separate instruction and data memories synthesized within the FPGA fabric [8]. This separation enables parallel instruction fetch and data access during each clock cycle, allowing the processor to achieve deterministic single-cycle execution.

Both memory units are mapped to independent buses: the **Instruction Bus (I-Bus)** connects the CPU core to the instruction memory, while the **Data Bus (D-Bus)** links the core to the data memory and peripherals. Each bus is 64 bits wide and follows a simple handshake protocol for read and write transactions.

3.4.1 Instruction Memory (IMEM)

The instruction memory stores up to 8192 words of 32 bits (total capacity = 32 KiB). It is addressed by the program counter and provides the next instruction combinationally within the same clock cycle. During simulation or FPGA synthesis, IMEM is initialized

by reading a hexadecimal file using `$readmemh("riscvtest.mem")`, which is generated automatically by the GNU toolchain. Since the architecture is single-cycle, instruction fetches are non-pipelined and incur zero wait states.

- **Word size:** 32 bits
- **Address width:** 15 bits (lower 2 bits are ignored)
- **Capacity:** 32 KiB
- **Access type:** Combinational (read-only)
- **Initialization:** `$readmemh()` from toolchain output

3.4.2 Data Memory (DMEM)

The data memory forms the main storage element for program variables and intermediate computational data within the RWU-RV64I processor. It consists of 1024 entries, each 64 bits wide, providing a total capacity of 8 KiB. The memory supports fully synchronous read and write operations, ensuring predictable timing behavior during instruction execution.

A byte-select mechanism enables subword data accesses for operations such as `LB`, `LH`, `LW`, and `LD`, allowing precise manipulation of individual bytes or words within a 64-bit word. This design ensures proper alignment and compliance with the RISC-V load/store architecture.

- **Word size:** 64 bits
- **Address width:** 13 bits
- **Capacity:** 8 KiB
- **Access type:** Synchronous read/write
- **Byte enables:** 8-bit granularity for subword operations

The data memory is connected to the system's address decoder, which identifies whether an access targets internal RAM or a peripheral-mapped region such as GPIO or clock control. This modular structure allows easy extension of the memory subsystem to include additional peripherals or external memory interfaces within the same unified address space.

3.4.3 Memory Map Overview

The overall address space seen by the RWU-RV64I processor is summarized in Table 3.1. Each region is selected by the address decoder, which generates the corresponding chip-select signals (`cs[3:0]`).

Table 3.1: Memory map and peripheral address ranges.

Address Range	**Mapped Device**	**Chip Select**
`0x0000_0000-0x0000_FFFF`	Data memory (DMEM)	CS[0]
`0x0001_0000-0x0001_000F`	GPIO peripheral	CS[1]
`0x0001_0010-0x0001_001F`	Reserved (QSPI)	CS[2]
`0x0001_0020-0x0001_002F`	Clock Generation Unit (CGU)	CS[3]

Figure 3.3 graphically illustrates this address layout. The memory map is linear, beginning with internal data memory at the base address and extending upward to peripheral regions. Although the instruction memory resides in a separate Harvard domain, it is shown conceptually for completeness.

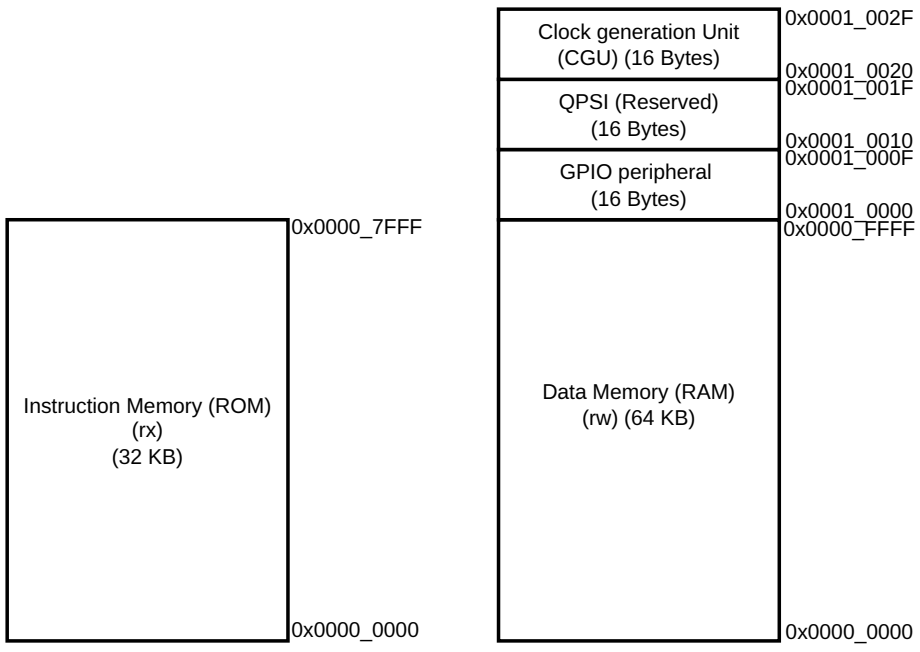

Figure 3.3: Memory and peripheral map of the RWU-RV64I.

The linker script used by the GNU toolchain mirrors this map exactly. The `.text`

section occupies IMEM starting from address 0, while data sections (`.data`, `.bss`) are placed in DMEM after a 256-byte reserved region used for runtime bookkeeping. This tight correspondence ensures that compiled C programs access the correct physical addresses during both simulation and FPGA execution.

3.5 Peripherals

3.5.1 General-Purpose I/O (GPIO)

The GPIO peripheral provides an 8-bit register accessible at address `0x0001_0004`. Software can write to this register to toggle LEDs or output status codes. It is primarily used for simple program verification during simulation and FPGA testing.

3.5.2 Clock Generation Unit (CGU)

The CGU divides the incoming 125 MHz clock from the Zybo board into several programmable frequencies. Configuration registers are memory-mapped at `0x0001 0020 - 0x0001 002F`. Typical settings are shown in Table 3.2.

Table 3.2: Example CGU division settings.

Divider	**Output Frequency**	**Typical Use**
2	62.5 MHz	CPU core
25	5 MHz	QSPI or slow I/O
200	625 kHz	Low-speed tests
400	312 kHz	Minimal power simulation

3.6 Target Hardware Platform

All FPGA-based verification was carried out on the Digilent Zybo Z7 (Zynq-7010) board [4]. The RWU-RV64I subsystem is implemented entirely in the programmable logic, while the ARM Cortex-A9 PS remains idle. The GPIO outputs are connected to on-board LEDs and JD port, and the 125 MHz oscillator provides the master clock.

3.7 Simulation and Verification Environment

Functional verification is performed using Vivado's `xsim` simulator with the help of testbench's. The toolchain-generated memory image (`riscvtest.mem`) is loaded into IMEM before simulation starts. Reset is deasserted after initialization, and the processor executes instructions cycle-by-cycle. Correct functionality is verified by monitoring GPIO outputs that follow a predefined sequence.

Observed signals include the instruction and data bus transactions, ALU results, and peripheral activity. Waveforms are analyzed using Vivado's built-in viewer or GTKWave for detailed timing inspection.

3.8 Summary

The RWU-RV64I is a 64-bit, single-cycle RISC-V processor tailored for educational and research use. Its modular RTL design—consisting of the CPU core, separate instruction and data memories, GPIO, and clock generation—enables full system visibility and rapid iteration during toolchain testing. The deterministic timing and simple bus structure make it ideal for compiler validation, C-runtime debugging, and bare-metal firmware development. This system forms the foundation for the software toolchain and evaluation work described in the subsequent chapters.

Chapter 4

Toolchain Implementation

4.1 Overview

This chapter describes the implementation of the complete RWU-RV64I toolchain used to compile, link, and execute bare-metal C programs on the custom RISC-V processor. The toolchain was designed around the open-source GNU RISC-V environment and adapted to meet the specific constraints of the RWU-RV64I hardware. Its purpose is to convert C source programs into loadable ELF and Verilog memory images that are compatible with the FPGA-based Harvard architecture.

The implementation includes:

- Installation and configuration of the GNU RISC-V toolchain.
- A custom `linker.ld` defining instruction and data memory regions.
- A minimal `crt0.s` startup file responsible for runtime initialization.
- A C header (`rwu-rv64i.h`) providing memory definitions and helper routines.

4.2 Installing the GNU RISC-V Toolchain on Linux

The GNU RISC-V toolchain can be installed on most Linux distributions using either the pre-built packages or by compiling from source. For this project, the toolchain was installed on Ubuntu 24.04 LTS using the official RISC-V GNU Toolchain repository maintained by the RISC-V International community.

4.2.1 System Requirements

Before installation, the following development utilities must be present:

```
sudo apt update
sudo apt install autoconf automake autotools-dev curl \
    python3 libmpc-dev libmpfr-dev libgmp-dev gawk build-essential \
    bison flex texinfo gperf libtool patchutils bc zlib1g-dev git
```

These packages provide all dependencies required for building GCC, binutils, and newlib.

4.2.2 Toolchain Source Installation

To build the complete bare-metal toolchain:

1. Clone the official repository:

   ```
   git clone https://github.com/riscv-collab/riscv-gnu-toolchain.git
   cd riscv-gnu-toolchain
   ```

2. Configure the build for a bare-metal (ELF) target:

   ```
   ./configure --prefix=/opt/riscv --with-arch=rv64i --with-abi=lp64
   ```

3. Build and install:

   ```
   make -j$(nproc)
   sudo make install
   ```

4. Add the toolchain to the system path:

   ```
   export PATH=/opt/riscv/bin:$PATH
   ```

After installation, the following commands should report valid toolchain versions:

```
riscv64-unknown-elf-gcc --version
riscv64-unknown-elf-ld --version
riscv64-unknown-elf-objcopy --version
```

This toolchain provides all the necessary utilities to compile and link bare-metal applications without operating-system dependencies.

4.3 Toolchain Configuration

The GNU RISC-V cross-compilation toolchain (`riscv64-unknown-elf`) was configured to target the `RV64I` base instruction set architecture with control and status register (CSR) support. The compiler was invoked using the following configuration options:

```
riscv64-unknown-elf-gcc -march=rv64i -mabi=lp64 -ffreestanding -nostdlib\
-O2 -g -msmall-data-limit=0 -Iinclude -c program.c -o program.o
```

- `-march=rv64i`: Specifies the target architecture as the 64-bit RISC-V integer base ISA.
- `-mabi=lp64`: Selects the LP64 application binary interface, where long and pointer types are 64 bits.
- `-ffreestanding`: Indicates that the program is built for a freestanding environment without standard libraries or operating system dependencies.
- `-nostdlib`: Prevents the linker from automatically including standard startup files or linking against system libraries.
- `-O2`: Enables compiler optimizations for improved performance while maintaining reasonable compilation time.
- `-g`: Includes debugging information for use with simulators or debuggers.
- `-msmall-data-limit=0`: Disables the use of the small data area, ensuring all objects are accessed via full 64-bit addressing.

This configuration ensures that the generated binaries are fully compatible with the custom RWU-RV64I hardware implementation and operate independently of any operating system or runtime environment.

The resulting object files are linked with a custom `linker.ld` and a minimal `crt0.s` to form an executable ELF binary. Finally, `objcopy` converts the ELF into a Verilog memory file used by the simulator.

4.4 Hardware Context and Memory Architecture

The RWU-RV64I processor implements a strict Harvard architecture with physically separate instruction (IMEM) and data (DMEM) memories synthesized inside the FPGA. For simulation simplicity, both share the same logical base address `0x0000_0000` in the linker configuration, although they exist as distinct memory blocks.

Region	Origin	Size	Attributes	Purpose
IMEM	0x0000_0000	32 KB	Read/Execute	Instruction storage
DMEM	0x0000_0000	8 KB	Read/Write	Data memory and stack
GPIO	0x0001_0000	48 B	Memory-mapped I/O	Peripheral output

Table 4.1: RWU-RV64I memory configuration used in the toolchain.

The instruction fetch and data access operate in parallel since IMEM and DMEM are physically isolated. This organization imposes important restrictions on the software toolchain, primarily the inability to access instruction memory as data.

4.5 Linker Script Development

The linker script defines the memory layout and placement of program sections within the RWU-RV64I processor architecture. It plays a critical role in mapping the compiled code and data to the physical memory addresses of the system. Since RWU-RV64I employs a Harvard architecture, the linker separates code and data into two distinct regions: Instruction Memory (IMEM) and Data Memory (DMEM). The complete linker script used in this work is provided in Appendix A.1 for reference.

Memory organization

The processor uses two independent address spaces:

- **IMEM:** Stores program instructions and constant data (`.text` and `.rodata` sections).
- **DMEM:** Stores runtime data such as global variables, the stack, and uninitialized memory (`.bss` section).

Key roles of the linker script

The linker script performs the following essential tasks:

1. Defines the origin and size of both IMEM and DMEM regions.
2. Places each program section into the correct memory region based on its access type.
3. Generates special symbols (labels) that are later used by the startup and runtime code for memory initialization.

Important linker-defined symbols

- `__bss_start` and `__bss_end`: Define the boundaries of the `.bss` section. These are used by the startup file to clear uninitialized memory.
- `__stack_top`: Marks the initial location of the stack pointer. The stack grows downward from this address during program execution.
- `_user_dmem_start`: Indicates the start of user data storage in DMEM, after the `.bss` section.

Section allocation overview

The general allocation strategy used in the linker script is summarized below:

- The `.text` section is placed at the beginning of IMEM and contains all executable code.
- The `.rodata` section follows immediately, storing constant values.
- The `.bss` section is mapped into DMEM and occupies all uninitialized writable memory.
- The stack pointer is positioned at the upper end of DMEM, and program-generated results are written to DMEM at runtime.

Design Rationale

This separation of code and data ensures that the processor accesses the instruction memory (IMEM) and data memory (DMEM) through independent buses, eliminating resource conflicts and enabling deterministic instruction execution. By explicitly defining memory regions, the linker script establishes a consistent reference framework for both the startup code and C programs, ensuring that all sections are placed at predictable addresses during compilation and linking.

4.6 Startup File (crt0.s)

The startup file is responsible for initializing the system immediately after reset and preparing the runtime environment required by C programs. It is written in assembly language and executes before any user-defined functions. The complete assembly listing of the startup file (`crt0.s`) is provided in the Appendix A.2 for detailed reference.

Purpose and responsibilities

The startup file performs the following actions in a fixed sequence:

1. Sets up the stack pointer using the symbol `__stack_top` defined by the linker.
2. Aligns the stack pointer to a 16-byte boundary to maintain ABI compatibility.
3. Clears the `.bss` section between `__bss_start` and `__bss_end`, ensuring all uninitialized global variables start with a zero value.
4. Calls the `main()` function to transfer control to the C program.
5. If `main()` returns, enters an infinite loop to prevent execution from running into undefined memory.

Memory initialization process

The `.bss` clearing loop is implemented using a simple 64-bit store operation sequence. It iterates through all addresses in the range between `__bss_start` and `__bss_end`, setting each to zero. This guarantees that all uninitialized global and static variables start in a defined state, as required by the C standard.

Handling of the `.data` section

Because RWU-RV64I uses separate instruction and data memories, no automatic copying of initialized variables from IMEM to DMEM occurs at startup. As a result, writable variables that require an initial value must be explicitly initialized in the program code itself if needed. This approach simplifies the startup process and avoids any dependency between IMEM and DMEM at runtime.

Structure and execution flow

After completing initialization, the startup code branches to the C entry point `main()`. If execution ever returns from `main()`, the program loops indefinitely to maintain a stable halted state. This behavior ensures predictable operation during simulation or FPGA execution.

A summarized control flow of the startup file is shown below:

1. Load `sp` ← `__stack_top`
2. Zero memory: from `__bss_start` to `__bss_end`
3. Call `main()`
4. Loop forever

4.7 Runtime Header File (rwu-rv64i.h)

The runtime header defines a minimal set of helper functions and symbolic references that allow C programs to interact with the RWU-RV64I hardware. It acts as the interface layer between compiled application code and the memory layout established by the linker and startup files.

Purpose of the runtime header

The header file provides three key functionalities:

- Access to linker-defined symbols such as `__bss_start`, `__bss_end`, and `_user_dmem_start`.

- A pointer-based data writer that allows sequential storage of 64-bit values in DMEM.
- A GPIO write function that communicates program results to the testbench or hardware outputs.

DMEM writer logic

The header defines a pointer, typically named `__rwu_dmem_next`, which points to the next available address in DMEM for data storage. This pointer is initialized by a helper function `rwu_dmem_reset()`, which sets it to `_user_dmem_start`. Each time the program calls `rwu_store64()`, the current pointer location is written with a 64-bit value, and then the pointer advances by eight bytes.

This mechanism allows C programs to store multiple runtime values in a continuous region of DMEM without manually computing memory addresses. Such an approach simplifies debugging and makes it easy to observe data changes during simulation.

GPIO interaction

Another key function, `rwu_print()`, performs a direct memory-mapped write to the GPIO output register located at address `0x00010004`. This function is used to send status or result values from the program to the testbench. It provides a straightforward way to visualize output without requiring additional hardware interfaces.

Integration with C programs

The header file is included in all user C applications through the directive:

```
#include "rwu-rv64i.h"
```

By doing so, programs gain access to all runtime definitions and helper functions necessary for DMEM writing and GPIO interaction.

A simplified example of its usage is shown below:

```
rwu_dmem_reset();
rwu_store64(result);
rwu_print((uint8_t)result);
```

The complete header file implementation is available in the Appendix A.3.

4.8 Summary

This part of the toolchain development established the foundational runtime environment for the RWU-RV64I processor. The linker script defines how program sections are distributed across IMEM and DMEM; the startup file initializes memory and prepares the stack and variables for program execution; and the runtime header provides lightweight C functions for data writing and GPIO output.

Together, these components ensure that C programs compiled for the RWU-RV64I processor execute in a predictable and hardware-consistent manner. The structure also provides a clean base for integration within the Eclipse IDE and for later FPGA testing, discussed in the next chapter.

Chapter 5

Eclipse Integration and Testing

5.1 Overview

After development of the RWU-RV64I software toolchain, the next major step was to integrate it into an efficient and reproducible development environment. For this purpose, the **Eclipse IDE for C/C++ Developers (Eclipse CDT 2023-09)** was used [1]. Eclipse provides a unified interface to write, compile, link, and test C programs targeting the RWU-RV64I processor, using the GNU RISC-V cross-compiler and a Makefile-based project structure.

This integration automates the firmware development cycle — from editing C source files to generating simulation-ready memory files — and eases repeated verification during simulation and FPGA testing (see Chapter 4 for toolchain details).

5.2 Software Requirements

The following software components were required for complete integration and testing of the RWU-RV64I system:

- **GNU RISC-V Toolchain:** Installed under `/opt/riscv/`, providing `riscv64-unknown-elf-gcc`, `ld`, `objcopy`, and related utilities [7, 3, 2].
- **GNU Make and Binutils:** Used to automate compilation, linking, and conversion steps.
- **Eclipse IDE for C/C++ Developers:** Version 2023-09 (CDT). [1]

- **Xilinx Vivado Design Suite:** Used for simulation (XSIM), synthesis, implementation, and FPGA bitstream generation. [10]
- **Zybo Z7-10 Development Board:** The physical platform used for hardware testing. [4]

Toolchain installation can be verified with:

```
which riscv64-unknown-elf-gcc
```

5.3 Creating and Importing the Project in Eclipse

5.3.1 Creating a Makefile Project

1. Launch Eclipse → select a workspace directory (e.g. `/workspace_rwu`).
2. Choose `File → New → C/C++ Project`.
3. Select `Makefile Project with Existing Code`.
4. Browse to the directory containing the top-level `Makefile`.
5. Assign the project name `rwu-rv64i`.
6. Under `Toolchain for Indexer Settings`, select `Cross GCC`.

5.3.2 Disabling Automatic Makefile Generation

- `Project → Properties → C/C++ Build → Builder Settings`.
- Uncheck `Generate Makefiles automatically`.
- Set Build directory to `${workspace_loc:/rwu-rv64i/}`.

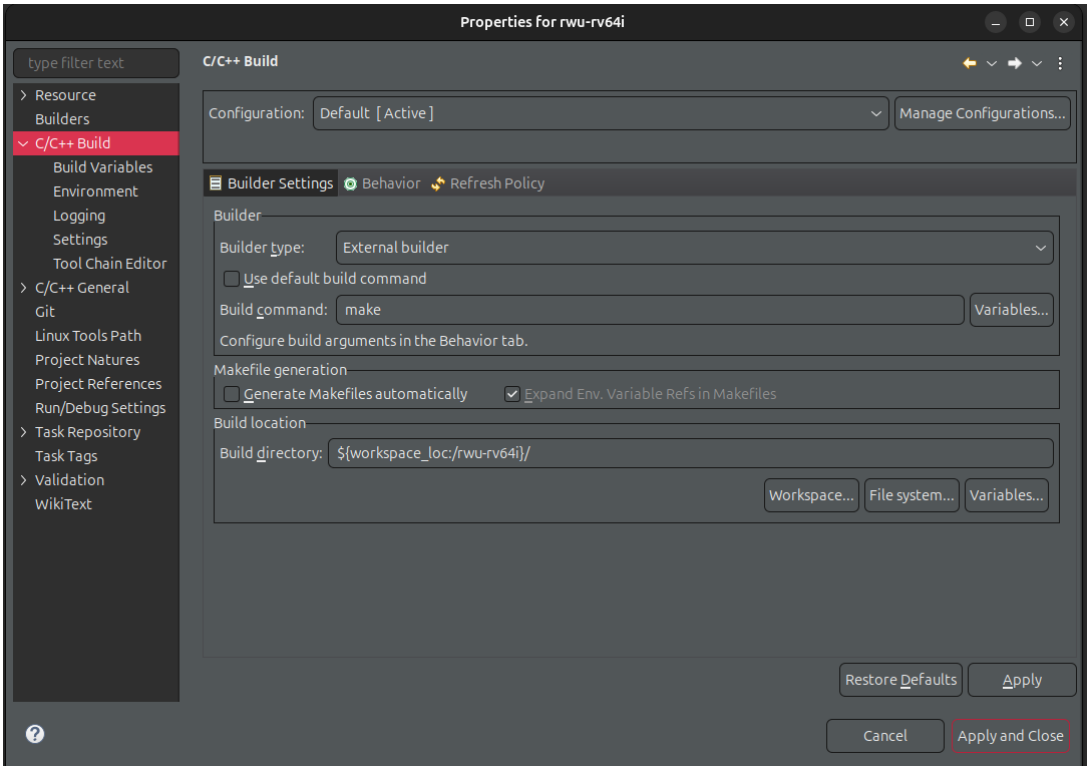

Figure 5.1: Eclipse project configuration showing external builder settings.

5.4 Build Environment Configuration

5.4.1 Setting the Compiler Path

1. `Project → Properties → C/C++ Build → Environment`.
2. Add environment variable: `Name: PATH Value: /opt/riscv/bin:${PATH}`

5.4.2 Selecting the External Builder

- `Project → Properties → C/C++ Build`.
- Set **Builder Type: External Builder**.
- Commands:

```
Build command: make -j$(nproc)
Clean command: make clean
```

- Working directory: top-level `Makefile`.

5.4.3 Defining Make Targets

Open `Window → Show View → Other → C/C++ → Make Targets` and define:

- `all` → `make all`
- `clean` → `make clean`
- `sim` → `make sim`
- `waves` → `make waves`
- `simcompile` → `make simcompile`

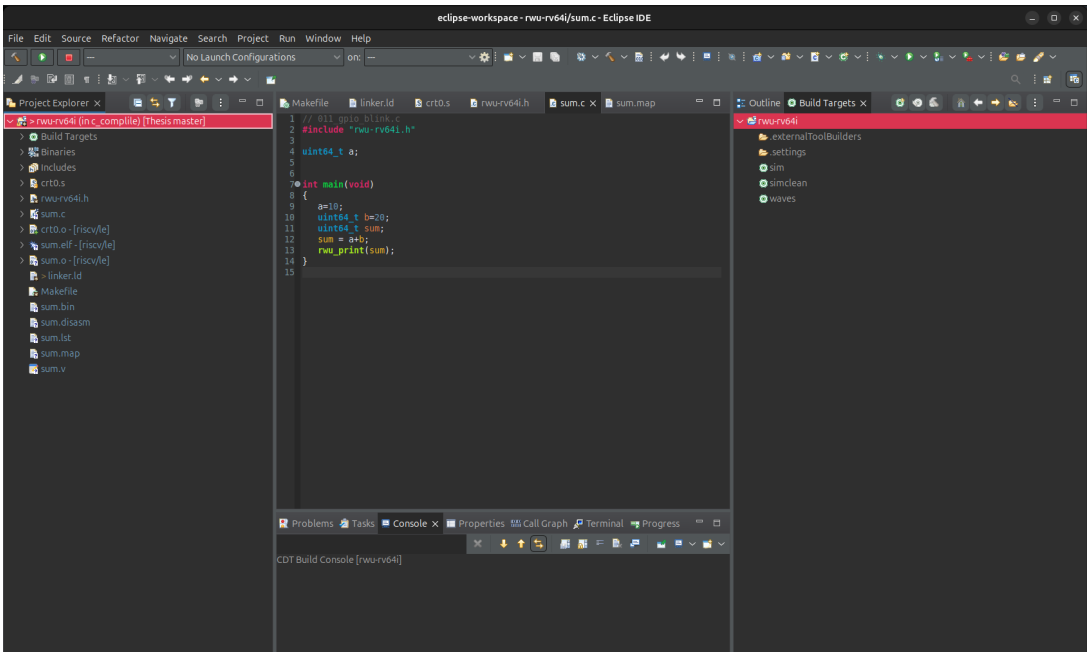

Figure 5.2: Make Targets view showing `sim`, `simclean`, `waves` targets.

These targets enable one-click firmware build, simulation, and waveform analysis.

5.5 Makefile Build Process

The top-level `Makefile` automates compilation, assembly, linking, and conversion of the program into a Verilog-formatted memory file. Only one C source file is compiled per build to simplify traceability. The Makefile implementation is included in Appendix A.4.

5.5.1 Compilation and Assembly

C and assembly sources are compiled and assembled using the GNU RISC-V toolchain. The exact flags used by the Makefile are:

```
riscv64-unknown-elf-gcc -march=rv64i -mabi=lp64 -ffreestanding \
 -nostdlib -O -g -msmall-data-limit=0 -I. -c program.c -o program.o

riscv64-unknown-elf-gcc -march=rv64i -mabi=lp64 -ffreestanding \
 -nostdlib -O -g -msmall-data-limit=0 -I. -c crt0.s -o crt0.o
```

5.5.2 Linking and Memory Image Generation

The linker combines the objects using `linker.ld`, creating the ELF binary:

```
riscv64-unknown-elf-gcc -march=rv64i -mabi=lp64 \
 -ffreestanding -nostdlib -T linker.ld crt0.o program.o -o pogram.elf \
 -Wl,-Map=program.map
```

The ELF file is converted to a Verilog-compatible memory file using:

```
riscv64-unknown-elf-objcopy -O verilog program.elf program.v
```

The output memory file is post-processed and copied into the simulator directory as `riscvtest.mem` and `riscvtest_tb_sum.mem` (this is automated by Makefile).

5.6 Simulation and XSIM Integration

Simulation is performed in Vivado XSIM. The Makefile invokes the simulator Makefile under `./sim`, which performs:

1. **Compilation:** `xvlog -sv` compiles RTL and testbench sources.
2. **Elaboration:** `xelab -debug all` produces an elaborated snapshot.
3. **Execution:** `xsim -runall` runs the simulation using the prepared `riscvtest.mem`.

The `waves` target launches the XSIM waveform GUI for signal-level debugging.

5.7 Cleaning and Reproducibility

The Makefile includes cleanup targets:

- `make clean` – removes firmware artifacts (`.o, .elf, .v, .bin, .lst`).
- `make simclean` – removes simulation outputs (`.wdb, .log, .xsim.dir`, etc.).

These targets ensure that the build is reproducible and that the `.mem` file always reflects the current firmware.

5.7.1 Vivado Project Setup

A Vivado RTL project was created and populated with:

- RWU-RV64I SystemVerilog RTL sources.
- The memory initialization file `riscvtest.mem`.
- The constraints file `Zybo-Master.xdc`.
- The top-level module `as_top_mem`.

5.7.2 Synthesis, Implementation and Bitstream

The standard Vivado flow (synthesis → implementation → bitstream generation) was executed. Vivado inferred both IMEM and DMEM as distributed RAM in this run. For larger images, BRAM inference or explicit BRAM instantiation is recommended to reduce LUT usage.

Resource	Used	Available
Slice LUTs (total)	9,389	17,600
LUT used as logic	4,269	17,600
LUT used as memory	5,120	6,000
Slice registers	2,502	35,200
Block RAM tiles (BRAM)	0	60
DSP slices	0	80

Table 5.1: Key device utilization numbers from Vivado synthesis report.

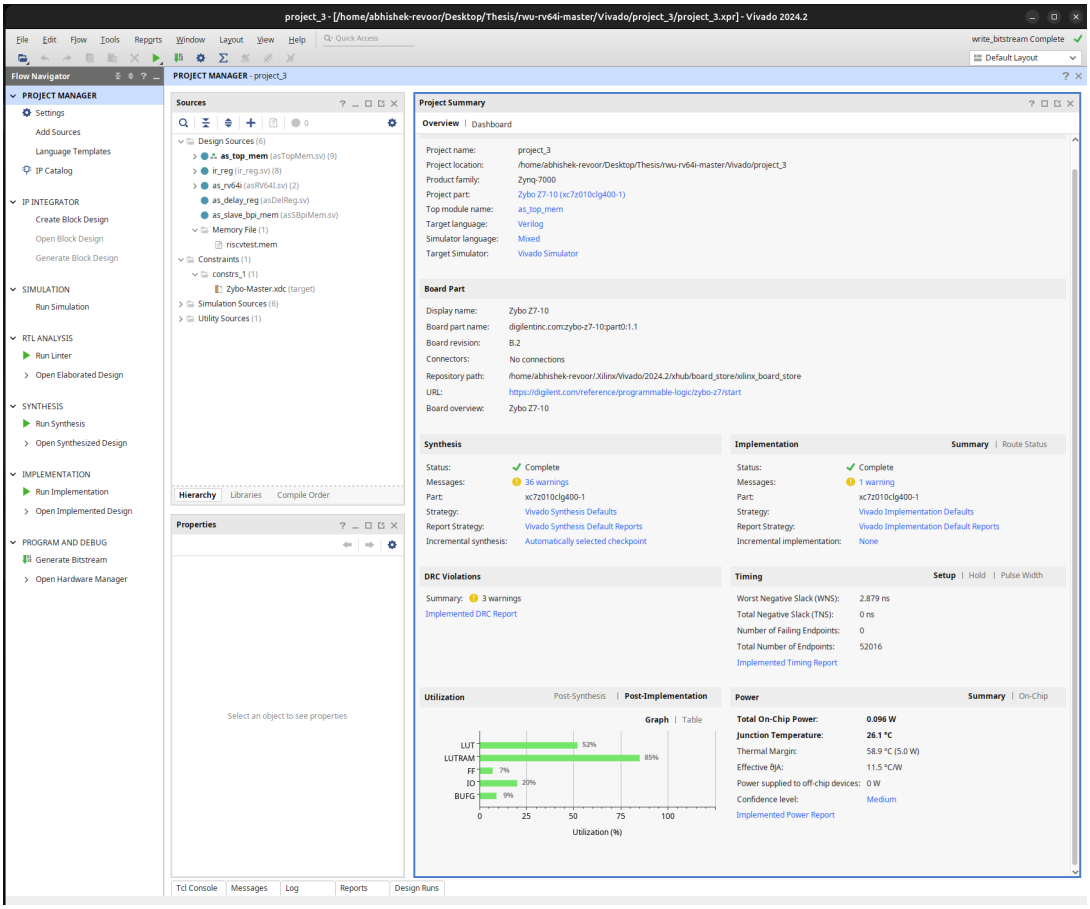

Figure 5.3: Vivado project manager, project summary and source tree.

5.7.3 Programming and Running on Board

To program the Zybo board:

1. Connect the board via USB-JTAG.
2. Open Vivado Hardware Manager and Auto Connect.
3. Program the device with the generated bitstream.

After programming, the firmware executes from the on-FPGA memories; the test program writes results to the DMEM and the GPIO output, observable via connected LEDs or external probes.

5.7.4 Integration correctness

The toolchain-produced `riscvtest.mem` was consumed by Vivado and produced the expected runtime behavior on the FPGA. The build and programming sequence executed without critical errors.

5.8 Summary

This chapter described integration of the RWU-RV64I toolchain into Eclipse, the Makefile-driven build and simulation flow, and the FPGA implementation workflow using Vivado. The Makefile (see Appendix A.4) enforces a single-C-file workflow and automates the generation of simulation memory images used for both XSIM and FPGA programming.

Chapter 6

Compiler optimizations: behaviour, effects and recommendations

6.1 Introduction

Compiler optimizations change generated instruction sequences, code size, and runtime behaviour in ways that can be decisive for correctness and observability on custom processors. This chapter examines the impact of standard GCC optimization levels on the RWU-RV64I core and the verification workflow developed in this project. We document experiments performed with three representative C workloads, analyse root causes for observed pass/fail behaviour in the simulation testbench, and provide concrete recommendations for the RWU-RV64I toolchain and verification flow. The experiments support reproducibility by using the same linker script and startup code described in Chapter 4 and the simulation/testbench harness described in Chapter 5.

6.2 Goals and Scope

This chapter pursues three primary objectives:

1. To quantify the impact of different compiler optimization levels on binary code size and observable program behavior.

2. To identify and analyze the causes of build-time failures and simulation mismatches encountered during compilation and execution.

3. To propose recommendations for establishing a stable development workflow that balances debuggability, correctness, and binary size efficiency.

The scope of this study is intentionally practical. All experiments are conducted using a minimal, freestanding GNU RISC-V toolchain (`riscv64-unknown-elf-gcc`) configured with the following options:

```
-march=rv64i -mabi=lp64 -ffreestanding -nostdlib -msmall-data-limit=0
```

The same linker script and startup files described in earlier chapters are reused to ensure consistency across builds. The three C test programs employed for this evaluation are included in Appendix B.

6.3 Experimental setup

6.3.1 Toolchain and build commands

All experiments used the same RISC-V cross-compiler and the same build infrastructure. Example compiler and link commands (extracted from the build logs) are:

```
riscv64-unknown-elf-gcc -march=rv64i -mabi=lp64 -ffreestanding\
 -nostdlib -msmall-data-limit=0 -I -O2 -g -c program.c -o program.o


riscv64-unknown-elf-gcc -march=rv64i -mabi=lp64 -ffreestanding \
 -nostdlib -T linker.ld -Wl,-Map="test.map" -o test.elf test.o crt0.o
```

Each experiment varied only the optimization flag among: `-O0`, `-O1`, `-O2`, `-O3`, `-Os`, `-Ofast`.

6.3.2 Runtime and verification infrastructure

The verification flow employs the following components:

- A SystemVerilog simulation testbench that drives the RWU-RV64I core and monitors GPIO outputs emitted through the lightweight `rwu_print` mechanism (the same testbench described in Chapter 5).

- A DMEM inspection utility (`rwu_store64`) that writes a 64-bit marker to a known data memory location, enabling offline verification by the testbench.
- The generated Verilog memory files (`*.mem`) produced by the command:

  ```
  riscv64-unknown-elf-objcopy -O verilog
  ```

  These files are then loaded into the simulator memory image for execution.

6.3.3 Programs under test

The three test programs exercise different optimization-sensitive patterns:

1. **opt_workload.c**: multi-part workload with many small functions, a tight arithmetic loop, a 16-case switch, and a small memcpy-like routine. Emits labeled GPIO markers E0..E5 and stores a 64-bit DMEM marker.
2. **square test (test.c)**: small arithmetic function that computes a square inside a loop and prints markers.
3. **test_simple_gpio.c**: simple arithmetic sequence with predictable GPIO outputs (intended to exercise constant-folding, arithmetic, and small-control flow).

The exact program listings are placed in Appendix B.

6.3.4 Measurements

For each program and optimization level we recorded:

- Build success / failure (linker errors).
- The `.text`, `.data` and `.bss` sizes from `size`.
- Simulation outcome (which GPIO markers were observed; pass/fail).
- The produced `.mem` image (retained for offline comparison).

Table 6.1: `opt_workload.c`: Code size and simulation outcome vs. optimization level

Optimization	`.text` (bytes)	`.data`	`.bss`	Build	Simulation
`-O0`	1920	0	136	OK	Failed (did not run)
`-O/-O1`	712	0	136	OK	Failed (reached marker E3)
`-O2`	472	0	136	OK	All tests cases passed
`-O3`	440	0	72	OK	All tests cases passed
`-Os`	480	0	136	OK	All tests cases passed
`-Ofast`	440	0	72	OK	All tests cases passed

Table 6.2: Square test (`test.c`): code size and outcome vs. optimization level

Optimization	`.text` (bytes)	`.data`	`.bss`	Build	Simulation
`-O0`	—	—	—	Failed	Failed
`-O/-O1`	140	0	0	OK	All test cases passed.
`-O2`	140	0	0	OK	All test cases passed.
`-O3`	140	0	0	OK	All test cases passed.
`-Os`	136	0	0	OK	All test cases passed.
`-Ofast`	140	0	0	OK	All test cases passed.

Table 6.3: test_simple_gpio.c: code size and outcome vs. optimization level

Optimization	`.text` (bytes)	`.data`	`.bss`	Build	Simulation
`-O0`	336	0	8	OK	No test cases passed (failed).
`-O/-O1`	108	0	0	OK	All test cases passed.
`-O2`	108	0	0	OK	All test cases passed.
`-O3`	108	0	0	OK	All test cases passed.
`-Os`	104	0	0	OK	All test cases passed.
`-Ofast`	108	0	0	OK	All test cases passed.

6.4 Results

Tables 6.1, 6.2 and 6.3 summarise the results extracted from the build logs and simulation runs.

In summary:

- Optimized images (`-O2`, `-O3`, `-Os`, `-Ofast`) were consistently smaller and passed the simulation validation for all three workloads.
- Unoptimized builds (`-O0`) either failed at link time (square test) or produced images that the testbench rejected (opt_workload and test_simple_gpio).
- Intermediate optimization (`-O`/`-O1`) sometimes improved results but was not as robust for the complex workload as `-O2` and above.

6.5 Analysis: root causes and mechanisms

This section explains the observed behaviour and links it to specific compiler transformations.

6.5.1 Compiler helper calls and freestanding builds

The linker error observed for the square test at `-O0` (`undefined reference to __muldi3`) shows that, at low optimization, GCC can emit calls to libgcc helper routines (for example, for multi-word arithmetic) which are not available in a minimal freestanding environment when `-nostdlib` is used. At higher optimization levels the compiler often transforms or replaces such operations with inline code that does not require external helpers, which explains why the problem disappears for `-O1` and above.

6.5.2 Inlining, dead-code elimination and code density

The large reduction in `.text` size for `opt_workload.c` (from 1920 bytes at `-O0` to 440–480 bytes at `-O2`/`-O3`/`-Os`) is primarily explained by inlining of frequently-called small functions, removal of redundant temporaries, and dead-code elimination. In practice this means:

- Function call overhead disappears and is replaced with tighter sequences of arithmetic and logical instructions.
- The number of branch instructions and memory accesses decreases, which changes the timing and ordering of observable I/O (GPIO writes).

6.5.3 Loop optimizations and strength reduction

The `loop_heavy` function benefits from standard loop optimizations at higher optimization levels (induction variable simplification, strength reduction, unrolling when profitable). Those optimizations reduce instruction counts and may eliminate index computations that would otherwise interact with DMEM or cause additional side effects.

6.5.4 Switch-case codegen: branch chain vs jump table

Compilers may implement ‘switch‘ statements either as a chain of conditional branches, or as a jump table. At high optimization levels and for dense case ranges the compiler typically emits a compact jump table. This reduces code size and branch mispredictions (in real hardware) and also simplifies the sequence of instructions executed in the simulator, improving the likelihood the TB observes the expected GPIO sequence.

6.5.5 Testbench observability and timing sensitivity

The verification TB in this project samples GPIO writes and validates an expected marker sequence (E0..E5) and DMEM sentinel. Because compiler optimizations can reorder or remove intermediary operations, the exact timing or presence of transient GPIO writes at `-O0` differs from optimized builds. The TB therefore may register mismatches even if the logical computation is correct, simply because the TB is sensitive to instruction-level timing and the unoptimized image produces additional/interleaved events.

6.6 Experimental verification and hardware output

To confirm the functional correctness of the RWU-RV64I toolchain and verify that compiler optimizations preserve expected behaviour, a conditional test program was implemented, compiled, simulated, and executed on the Zybo Zynq–7000 FPGA board. The test validates

arithmetic, comparison, and branching instructions, and demonstrates observable GPIO output as a visible indicator of processor execution.

6.6.1 Purpose of the test

The goal of this test is to verify that:

- The RWU-RV64I core correctly executes arithmetic and comparison operations.
- Conditional branching behaves as intended in optimized binaries.
- The GPIO output produced by the `rwu_print()` function correctly reflects program logic both in simulation and on physical hardware.

To achieve this, a simple C program (Listing 6.1) was created, where the result of an arithmetic comparison determines which LED pattern is displayed on the Zybo board.

6.6.2 Conditional test program

Listing 6.1: Conditional LED control program for RWU-RV64I verification.

```
1  #include "rwu-rv64i.h"
2
3  uint64_t a;
4  const uint64_t b = 5;
5
6  int main(void) {
7      rwu_dmem_reset();
8      uint64_t c = 10;
9      uint64_t d = 20;
10     a = 20;
11
12     while (1) {
13         if ((a + c) > (b + d)) {
14             rwu_print(0x0F);  // Turn all LEDs ON
15         } else {
16             rwu_print(0x03);  // Turn first two LEDs ON
```

```
17          }
18      }
19  }
```

The program begins by initializing four variables: `a` (global, writable), `b` (constant), and two local variables `c` and `d`. The statement `rwu_dmem_reset()` clears data memory to ensure deterministic results before execution.

Inside the infinite loop, the processor continuously evaluates the condition:

$$(a + c) > (b + d)$$

Substituting the initialized values:

$$(20 + 10) > (5 + 20) \Rightarrow 30 > 25$$

The condition is true, so the program calls `rwu_print(0x0F)`, which sets the lower four GPIO pins high—turning ON all four LEDs on the Zybo board. If the condition were false, it would instead call `rwu_print(0x03)` to illuminate only the first two LEDs.

Thus, this simple logic simultaneously validates the correctness of arithmetic addition, comparison, and branch instruction handling within the RWU-RV64I processor.

6.6.3 Build process confirmation

The project was built in Eclipse CDT using the `-O2` optimization flag. The cross-compilation produced a valid ELF file, which was converted automatically to Verilog memory and binary images for simulation and FPGA download. The complete build console output is shown in Figure 6.1, confirming that the build finished with zero errors or warnings.

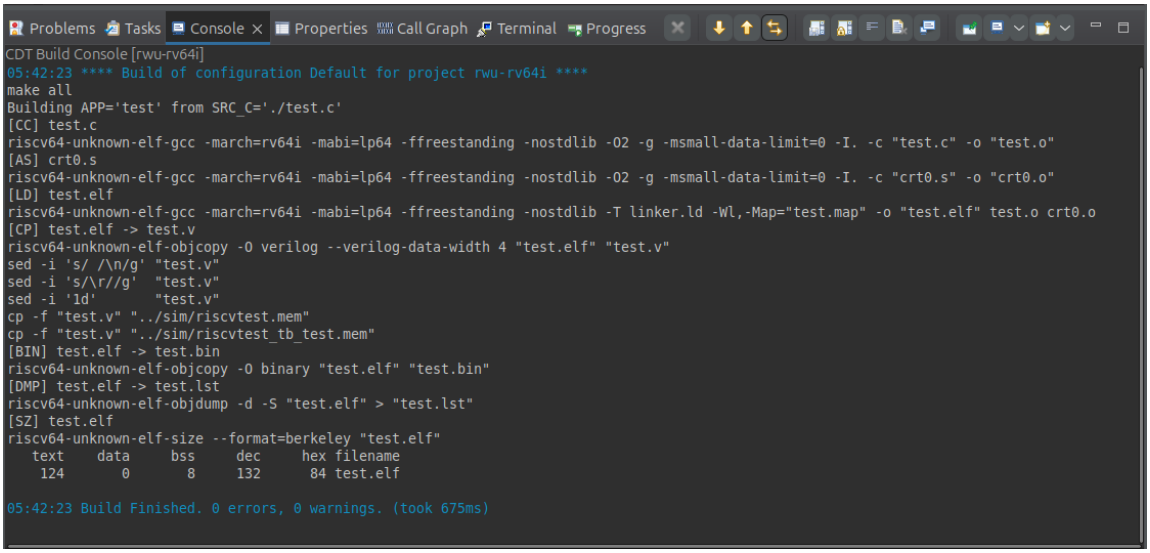

Figure 6.1: Successful build of the conditional test program using the RWU-RV64I GCC toolchain in Eclipse CDT.

6.6.4 Simulation verification in XSIM

The generated memory image was loaded into the Vivado XSIM testbench. Figure 6.2 shows the XSIM console output. The simulation log reports the message: `A+C is greater than B+D 0xF` followed by `Simulation succeeded`, indicating that the conditional branch executed correctly and that the GPIO value `0x0F` was produced.

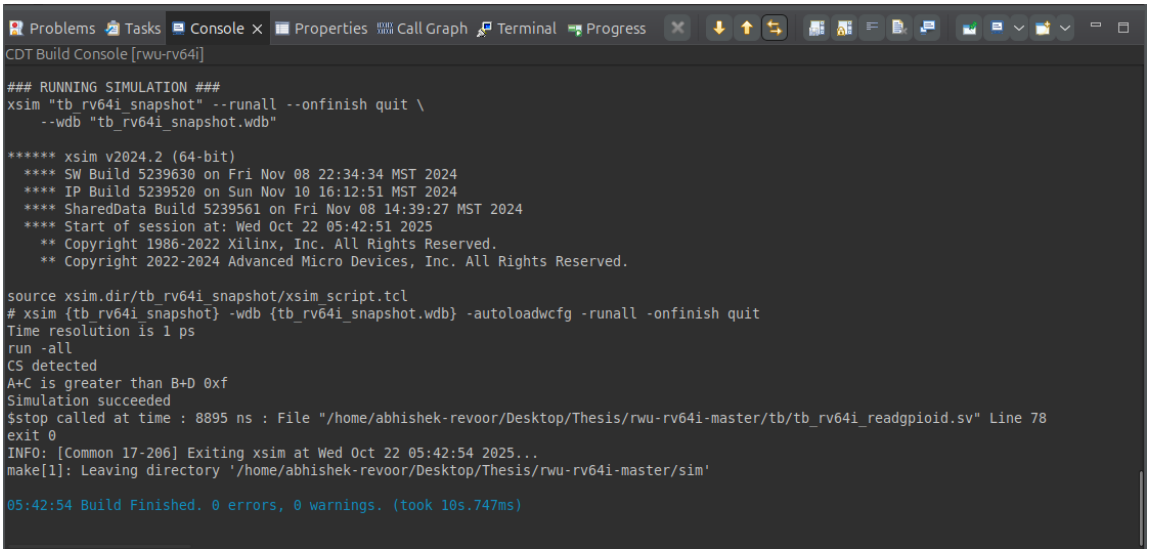

Figure 6.2: XSIM simulation console showing correct conditional result and successful program termination.

6.6.5 Waveform analysis

The captured waveform from XSIM is presented in Figure 6.3. The signal `gpio_s[7:0]` carries the output value driven by the `rwu_print()` function. Here, it maintains a stable value of `0x0F`, verifying that the branch condition evaluated as true (`30 > 25`). This confirms that both arithmetic and logical instructions were correctly synthesized and executed by the RWU-RV64I processor model.

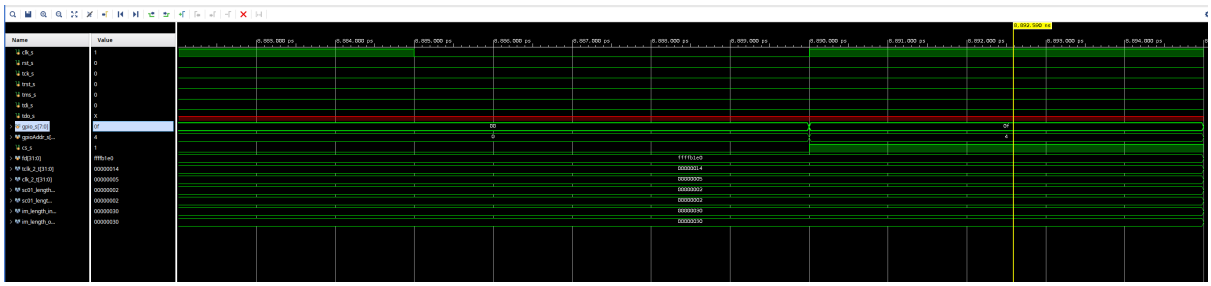

Figure 6.3: XSIM waveform showing GPIO output (`gpio_s[7:0]` = 0x0F) for the satisfied branch condition.

6.6.6 FPGA hardware execution

After successful simulation, the generated memory file (`test.mem`) was programmed into the Zybo FPGA configuration. Figure 6.4 shows the physical output on the board, where all four user LEDs are ON. This directly corresponds to the logical output value `GPIO = 0x0F`, confirming that the processor core, memory interface, and GPIO hardware mapping function identically in real hardware.

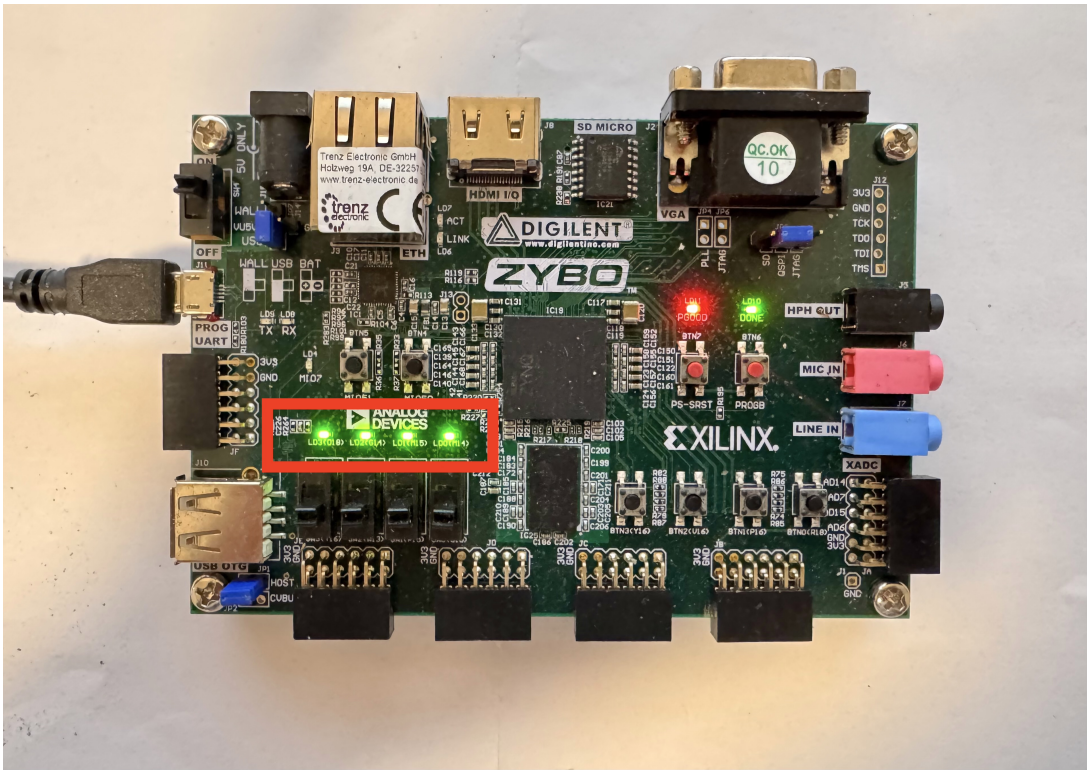

Figure 6.4: FPGA hardware output on the Zybo board showing all four LEDs ON (`GPIO = 0x0F`).

6.6.7 Discussion

The consistent results between simulation and hardware confirm the following:

1. The RWU-RV64I GCC toolchain correctly compiles conditional and arithmetic expressions.

2. The simulation model matches the physical FPGA implementation at the signal level.

3. The optimization level `-O2` preserves program logic and I/O semantics.

This demonstrates that the entire development flow—from compilation to FPGA execution—is stable and reliable for both debugging and optimized builds. The LED output serves as a practical visual indicator of the internal program logic, allowing straightforward verification of instruction correctness without requiring serial I/O or additional peripherals.

6.7 Summary

This chapter presented an experimental study of compiler optimization effects on the RWU-RV64I toolchain. Different GCC optimization levels were applied to representative C programs to observe their impact on code size, execution behaviour, and simulation results. The analysis showed that optimized builds (`-O2` and above) consistently produced smaller binaries and reliable outputs, while unoptimized builds (`-O0`) often failed due to missing helper routines and increased instruction count.

A conditional arithmetic and branching program was then used to verify the correct functional behaviour of the optimized binary in both simulation and hardware. The results demonstrated identical GPIO outputs in XSIM and on the Zybo FPGA board, confirming that the RWU-RV64I processor and toolchain operate correctly and consistently across environments. These findings validate that the applied compiler optimizations preserve program logic and observable I/O behaviour, ensuring reliable execution and efficient code generation for the RWU-RV64I development workflow.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

This thesis presented the design and implementation of a **complete software toolchain** for the RWU-RV64I processor, a 64-bit RISC-V–based educational core developed at Hochschule Ravensburg–Weingarten. The work successfully bridged the gap between the processor's hardware implementation and its software programmability, transforming the RWU-RV64I from an assembly-only prototype into a fully functional, C-programmable platform.

The developed toolchain was built around the **GNU RISC-V compiler suite (GCC and Binutils)** and includes all essential runtime components required for bare-metal program execution:

- a custom `linker.ld` precisely reflecting the processor's Harvard memory map,
- a minimal `crt0.s` startup file for runtime initialization,
- a lightweight runtime header (`rwu-rv64i.h`) providing DMEM and GPIO interfaces, and
- a unified `Makefile`-based build system automating compilation, linking, image generation, and simulation.

The toolchain was fully integrated into the **Eclipse CDT IDE**, resulting in a reproducible and user-friendly workflow for compiling, simulating, and debugging C programs on the RWU-RV64I. Simulation using **Vivado XSIM** and deployment on the **Zybo**

Zynq-7000 FPGA verified that compiled programs executed correctly, with GPIO and DMEM outputs matching expected results. This confirmed correct instruction decoding, memory mapping, and peripheral interfacing across both simulation and hardware domains.

In addition, a detailed analysis of **compiler optimization behavior** demonstrated that optimization levels significantly affect code size and runtime behavior. The experiments revealed that the `-O2` and `-Os` levels produced compact, reliable binaries, whereas unoptimized (`-O0`) builds often failed due to missing runtime helpers or altered timing observable by the testbench. These findings led to stable configuration recommendations for future RWU-RV64I software development.

Overall, this work achieved the following outcomes:

- Development of a complete, bare-metal toolchain enabling standard C program execution on RWU-RV64I.
- Creation of an automated build and simulation flow validated on both simulation and FPGA hardware.
- Integration of the workflow into Eclipse CDT for reproducible, professional-grade development.
- Experimental evaluation of compiler optimization levels and establishment of verified defaults.

Through these achievements, the RWU-RV64I processor now benefits from a fully operational and extensible development infrastructure that supports academic teaching, research, and future processor extensions.

7.2 Future Work

Although this thesis established the full foundation for RWU-RV64I software development, several promising directions remain open for future enhancement.

1. Implementation of a Program Loader

At present, programs are pre-linked and loaded into instruction memory via simulation or FPGA bitstream initialization. A logical next step is to design a dedicated **program**

loader capable of transferring compiled binaries into memory at runtime. This can be achieved through:

- a serial or JTAG-based download interface (e.g., UART or SPI),
- an integrated bootloader module in the RWU-RV64I system, and
- a lightweight host utility to parse and transmit ELF or raw binary files.

Such a loader would remove the need to regenerate FPGA bitstreams for each test and would enable rapid, interactive software deployment.

2. Expansion of Runtime Support and Peripherals

Future work may extend the runtime system beyond GPIO and DMEM interaction to include:

- UART or serial console I/O,
- timer peripherals and interrupt handling,
- SPI or I^2C communication interfaces, and
- a minimal C runtime library (subset of `libc`) for higher-level applications.

These extensions would move the RWU-RV64I platform closer to real-world embedded-system capabilities while maintaining its educational simplicity.

3. Debugging and Profiling Infrastructure

Adding **GDB remote debugging** support through JTAG or UART would enable step-wise execution and register inspection from Eclipse or command-line tools. Incorporating **cycle-accurate performance counters** in hardware would further allow profiling and timing analysis, strengthening the platform's use for teaching compiler and architecture co-design.

4. Educational and Research Extensions

The modular toolchain developed in this work can serve as the foundation for advanced projects such as:

- porting a minimal RTOS or cooperative scheduler,
- adding compiler backend modifications for experimental instruction-set extensions, and
- integrating formal verification frameworks to link software semantics with hardware execution traces.

7.3 Summary

In summary, this thesis delivered a complete and extensible software toolchain for the RWU-RV64I processor, converting it into a fully usable research and teaching platform. The developed environment enables reproducible compilation, linking, and hardware execution of standard C programs on a custom 64-bit RISC-V core. Future developments such as dynamic program loading, peripheral expansion, and advanced debugging support will further enhance the platform's capabilities and ensure its continued relevance in RISC-V–based embedded system education at Hochschule Ravensburg–Weingarten.

Appendix A

Toolchain Source Listings

This appendix contains the complete source listings of the toolchain components developed for the RWU-RV64I processor. These files correspond to the implementations discussed in Chapter 4. Each listing is presented in its verified form, which was used during simulation and FPGA testing.

A.1 Linker Script (`linker.ld`)

The linker script defines the memory organization of the RWU-RV64I system, allocating each program section to either instruction or data memory according to access type. It also declares key symbols that are referenced in the startup and runtime header files.

Listing A.1: RWU-RV64I Linker Script

```
1  OUTPUT_ARCH(riscv)
2  ENTRY(_start)
3
4  IMEM_ORIGIN = 0x00000000;
5  IMEM_SIZE   = 32K;
6
7  DMEM_ORIGIN = 0x00000000;
8  DMEM_SIZE   = 8K;
9
10 /* Memory regions */
11 MEMORY
```

```
12 {
13   IMEM (rx) : ORIGIN = IMEM_ORIGIN, LENGTH = IMEM_SIZE
14   DMEM (rw) : ORIGIN = DMEM_ORIGIN, LENGTH = DMEM_SIZE
15 }
16
17 /* Convenience symbols */
18 PROVIDE(_dmem_origin = ORIGIN(DMEM));
19 PROVIDE(_dmem_size   = LENGTH(DMEM));
20
21 SECTIONS
22 {
23     /* Put text/rodata in IMEM (LMA=VMA=IMEM) */
24     . = ORIGIN(IMEM);
25     .text : {
26         PROVIDE(_text_start = .);
27         *(.text*)
28         *(.rodata*)
29         *(.srodata*)
30         PROVIDE(_text_end = .);
31     } > IMEM
32
33     /* Put bss at start of DMEM (NOLOAD) */
34     . = ORIGIN(DMEM);
35     .bss (NOLOAD) : {
36         PROVIDE(__bss_start = .);
37         *(.sbss*)
38         *(.bss*)
39         *(COMMON)
40         PROVIDE(__bss_end = .);
41     } > DMEM
42
43     /* Stack top at end of DMEM */
44     PROVIDE(__stack_top = ORIGIN(DMEM) + LENGTH(DMEM));
45
```

```
46      /* User DMEM start is immediately after .bss (no reserved gap) */
47      PROVIDE(_user_dmem_start = __bss_end);
48
49      /* Discard sections not needed */
50      /DISCARD/ : { *(.eh_frame) *(.comment) }
51  }
```

A.2 Startup Assembly File (`crt0.s`)

The startup file initializes the processor state and memory immediately after reset. It sets the stack pointer, clears the `.bss` section, and transfers control to the `main()` function.

Listing A.2: RWU-RV64I Startup File

```
1  /* crt0.s -- minimal RISC-V startup
2   *
3   * Behavior:
4   *  - Set sp = __stack_top (from linker)
5   *  - Zero .bss region [__bss_start, __bss_end)
6   *  - Jump to main()
7   *
8   * NOTE: This startup intentionally does NOT copy .data from IMEM to DMEM.
9   */
10
11      .section .init
12      .option norvc
13      .global _start
14      .type   _start,@function
15
16  _start:
17      /* Load stack pointer from linker symbol __stack_top */
18      la      sp, __stack_top
19      /* Align stack to 16 bytes just in case */
20      andi    sp, sp, -16
21
```

```
22      /* Zero .bss from __bss_start to __bss_end, 8-bytes at a time */
23      la      t0, __bss_start  /* t0 -> start */
24      la      t1, __bss_end    /* t1 -> end */
25  1:
26      beq     t0, t1, 2f
27      sd      x0, 0(t0)
28      addi    t0, t0, 8
29      blt     t0, t1, 1b
30  2:
31
32      /* Call main() */
33      call    main
34
35  halt:
36      /* If main returns, hang here forever */
37      j       halt
38
39      .size _start, .-_start
```

A.3 Runtime Header File (`rwu-rv64i.h`)

The runtime header defines the essential constants, memory references, and helper routines used by the compiled C programs. It enables writing results to DMEM and communicating through GPIO.

Listing A.3: RWU-RV64I Runtime Header File

```
1  #ifndef RWU_RV64I_H
2  #define RWU_RV64I_H
3
4  #include <stdint.h>
5  #include <stddef.h>
6
7  /* ----------------- Memory Map ---------------- */
8  #define IMEM_BASE       0x00000000UL
```

```
9  #define IMEM_SIZE       (32 * 1024UL)
10
11 #define DMEM_BASE       0x00000000UL
12 #define DMEM_SIZE       (8 * 1024UL)
13
14 /* ----------------- GPIO Mapping ---------------- */
15 #define GPIO_BASE       0x00010000UL   /* Must match RTL */
16 #define GPIO_OUT_OFFSET 4
17
18 #define GPIO_REG_OUT (*(volatile uint8_t *)(GPIO_BASE + GPIO_OUT_OFFSET))
19
20 /* ----------------- Linker Symbols ---------------- */
21 extern char __bss_start[];
22 extern char __bss_end[];
23 extern char __stack_top[];
24 extern uint64_t _user_dmem_start[];
25
26 /* ----------------- GPIO Writer ---------------- */
27 static inline void rwu_print(uint8_t value)
28 {
29     GPIO_REG_OUT = value;
30 }
31
32 /* ----------------- DMEM Writer ---------------- */
33 static volatile uint64_t *__rwu_dmem_next;
34
35 static inline void rwu_dmem_reset(void)
36 {
37     __rwu_dmem_next = _user_dmem_start;
38 }
39
40 /* Store 64-bit value and advance pointer */
41 static inline void rwu_store64(uint64_t v)
42 {
```

```
43     *__rwu_dmem_next++ = v;
44 }
45
46 #endif /* RWU_RV64I_H */
```

A.4 Build Makefile (embedded)

The Makefile below drives firmware compilation and the conversion into Verilog memory files used for simulation. **Recommendation:** upload the real `Makefile` to the project root (as shown in the Overleaf tree) and include it directly so the PDF preserves literal tabs.

Listing A.4: RWU-RV64I combined firmware + simulation Makefile (embedded) — `Makefile`

```
1  # ======================================================================
2  # RWU-RV64I | Combined Firmware + Simulation Makefile
3  # Place exactly ONE .c file in this folder (c_complile/).
4  # Produces <name>.elf/.v/.map/.bin/.lst and sim .mem files.
5  # Can also run Vivado xsim by calling the ../sim/Makefile.
6  # ======================================================================
7
8  .SUFFIXES:
9
10 # -------- Toolchain --------
11 CC      := riscv64-unknown-elf-gcc
12 AS      := riscv64-unknown-elf-gcc
13 OBJCOPY := riscv64-unknown-elf-objcopy
14 SIZE    := riscv64-unknown-elf-size
15 OBJDUMP := riscv64-unknown-elf-objdump
16
17 # -------- Flags --------
18 ARCH    := -march=rv64i -mabi=lp64
19 COMMON  := -ffreestanding -nostdlib
```

```
20 CFLAGS  := $(ARCH) $(COMMON) -O2 -g -msmall-data-limit=0 -I.
21 ASFLAGS := $(CFLAGS)
22 LDFLAGS := $(ARCH) $(COMMON) -T linker.ld
23 CPFLAGS := -O verilog --verilog-data-width 4
24
25 # -------- Source selection --------
26 ifeq ($(wildcard c_compile),)
27   SEL_DIR := .
28 else
29   SEL_DIR := ./c_compileram obmization leve;lns in c
30 endif
31
32 ALL_CS       := $(wildcard $(SEL_DIR)/*.c)
33 NUM_SELECTED := $(words $(ALL_CS))
34 ifeq ($(NUM_SELECTED),0)
35   $(error No .c file found in $(SEL_DIR)/. Keep exactly one C file.)
36 endif
37 ifneq ($(NUM_SELECTED),1)
38   $(error More than one .c file found in $(SEL_DIR)/. Keep exactly one C
     ↪ file.)
39 endif
40
41 SRC_C := $(firstword $(ALL_CS))
42 APP   := $(strip $(basename $(notdir $(SRC_C))))
43
44 # Guard: reject whitespace in name
45 empty :=
46 space := $(empty) $(empty)
47 ifneq ($(findstring $(space),$(APP)),)
48   $(error Source filename contains spaces: '$(APP)'. Please rename.)
49 endif
50
51 # -------- Outputs --------
52 CRT_OBJ := crt0.o
```

```
53 C_OBJ   := $(APP).o
54 ELF     := $(APP).elf
55 MAP     := $(APP).map
56 VHEX    := $(APP).v
57 BIN     := $(APP).bin
58 LST     := $(APP).lst
59
60 # -------- Sim outputs --------
61 SIMDIR  := ../sim
62 SIMMEM1 := riscvtest.mem
63 SIMMEM2 := riscvtest_tb_$(APP).mem
64
65 # -------- Default goal --------
66 .PHONY: all clean size who
67 .DEFAULT_GOAL := all
68
69 # ================== Firmware build ==================
70
71 all: who $(VHEX) $(BIN) $(LST) size
72
73 who:
74     @echo "Building APP='$(APP)' from SRC_C='$(SRC_C)'"
75
76 # C -> obj
77 $(C_OBJ): $(SRC_C)
78     @echo "[CC] $<"
79     $(CC) $(CFLAGS) -c "$<" -o "$@"
80
81 # asm -> obj
82 $(CRT_OBJ): crt0.s
83     @echo "[AS] $<"
84     $(AS) $(ASFLAGS) -c "$<" -o "$@"
85
86 # link -> ELF (+map)
```

```
87 $(ELF): $(C_OBJ) $(CRT_OBJ) linker.ld
88     @echo "[LD] $@"
89     $(CC) $(LDFLAGS) -Wl,-Map="$(MAP)" -o "$@" $(C_OBJ) $(CRT_OBJ)
90
91 # ELF -> VHEX + copy to sim mem
92 $(VHEX): $(ELF)
93     @echo "[CP] $< -> $@"
94     $(OBJCOPY) $(CPFLAGS) "$<" "$@"
95     sed -i 's/ /\n/g' "$@"
96     sed -i 's/\r//g'  "$@"
97     sed -i '1d'       "$@"
98     @mkdir -p "$(SIMDIR)"
99     cp -f "$@" "$(SIMDIR)/$(SIMMEM1)"
100    cp -f "$@" "$(SIMDIR)/$(SIMMEM2)"
101
102 # ELF -> raw binary
103 $(BIN): $(ELF)
104    @echo "[BIN] $< -> $@"
105    $(OBJCOPY) -O binary "$<" "$@"
106
107 # ELF -> disassembly
108 $(LST): $(ELF)
109    @echo "[DMP] $< -> $@"
110    $(OBJDUMP) -d -S "$<" > "$@"
111
112 # ELF size
113 size: $(ELF)
114    @echo "[SZ] $<"
115    $(SIZE) --format=berkeley "$<"
116
117 clean:
118    @echo "[RM] cleaning firmware build"
119    rm -f *.o *.elf *.v *.map *.bin *.lst
120
```

```
121 # ================== Simulation targets ==================
122
123 VIVADO_SETTINGS ?=
124 SIM_TEST := $(APP)
125
126 define _SIM_MAKE
127 $(MAKE) -C $(SIMDIR) TEST=$(SIM_TEST) $(1)
128 endef
129
130 ifeq ($(strip $(VIVADO_SETTINGS)),)
131   SIM_CALL = $(_SIM_MAKE)
132 else
133   SHELL := /bin/bash
134   SIM_CALL = source "$(VIVADO_SETTINGS)"; $(_SIM_MAKE)
135 endif
136
137 .PHONY: sim waves simcompile simclean
138
139 sim: $(VHEX)
140     @echo "[SIM] Running xsim for TEST=$(SIM_TEST)"
141     @$(SIM_CALL) simulate
142
143 waves: $(VHEX)
144     @echo "[SIM] Launching xsim GUI for TEST=$(SIM_TEST)"
145     @$(SIM_CALL) waves
146
147 simcompile: $(VHEX)
148     @echo "[SIM] Compiling/elaborating for TEST=$(SIM_TEST)"
149     @$(SIM_CALL) compile
150
151 simclean:
152     @echo "[SIM] Cleaning simulation outputs"
153     @$(SIM_CALL) clean
```

Appendix B

Optimization Test Programs

This appendix lists the three C programs used in the optimization experiments. Each program was compiled with the freestanding RWU-RV64I toolchain using identical compiler options except for the optimization level (`-O0` - `-Ofast`). Including the source files directly ensures reproducibility and transparency of all tests.

Program 1 – opt_workload.c

Listing B.1: opt_workload.c — multi-part workload exercising arithmetic, switch, and memory routines

```
#include "rwu-rv64i.h"
#include <stdint.h>

/* alias for GPIO write */
static inline void gout(uint8_t v) { rwu_print(v); }

/* ------- small functions (candidates for inlining) ------- */
static uint32_t f_add(uint32_t a, uint32_t b) { return a + b; }
static uint32_t f_xor(uint32_t a, uint32_t b) { return a ^ b; }
static uint32_t f_rotl(uint32_t x, uint32_t r) {return (x << r) | (x >> (32 -
    ↪ r));}
static uint32_t f_mix(uint32_t x, uint32_t y) {
    uint32_t t = f_add(x, y);
    t = f_xor(t, (x << 3));
```

```
    t = f_rotl(t, 7);
    return t;
}

static uint32_t run_funcs(uint32_t seed, int rounds) {
    uint32_t acc = seed;
    for (int i = 0; i < rounds; ++i) {
        acc = f_mix(acc, (uint32_t)i);
    }
    return acc;
}

/* -loop-heavy workload (candidates for unrolling/strength-reduction) - */
static uint32_t loop_heavy(uint32_t n) {
    uint32_t s = 0;
    for (uint32_t i = 1; i <= n; ++i) {
        uint32_t mul3 = (i + i + i);
        s += (mul3 ^ (i << 2)) + ((i & 0xF) << 5);
    }
    return s;
}

/* ------- switch/table workload (branch vs jump-table) ------- */
static uint32_t switch_case(uint32_t x) {
    switch (x & 0xF) {
    case 0: return 0x1111u;
    case 1: return 0x2222u;
    case 2: return 0x3333u;
    case 3: return 0x4444u;
    case 4: return 0x5555u;
    case 5: return 0x6666u;
    case 6: return 0x7777u;
    case 7: return 0x8888u;
    case 8: return 0x9999u;
```

```
    case 9: return 0xAAAAu;
    case 10: return 0xBBBBu;
    case 11: return 0xCCCCu;
    case 12: return 0xDDDDu;
    case 13: return 0xEEEEu;
    case 14: return 0xFFFFu;
    default: return 0x0000u;
    }
}

static uint32_t run_switch(uint32_t seed, int rounds) {
    uint32_t acc = 0;
    for (int i = 0; i < rounds; ++i) {
        acc ^= switch_case(seed + (uint32_t)i);
    }
    return acc;
}

/* ------- small memcpy-like routine (should remain simple) ------- */
static void mini_memcpy(uint8_t *dst, const uint8_t *src, uint32_t n) {
    for (uint32_t i = 0; i < n; ++i) {
        dst[i] = src[i];
    }
}

/* ------- top-level main: emit GPIO markers, run sections, output results
    ↪ ------- */
int main(void) {
    rwu_dmem_reset();

    /* START marker */
    gout(0xE0);

    /* Section 1: many small functions */
```

```
81      gout(0xE1);
82      uint32_t r1 = run_funcs(0x12345678u, 40);
83
84      /* Section 2: loop heavy */
85      gout(0xE2);
86      uint32_t r2 = loop_heavy(200);
87
88      /* Section 3: switch/jump-table */
89      gout(0xE3);
90      uint32_t r3 = run_switch(0x55AAu, 100);
91
92      /* Section 4: memcpy-like (exercise data movement) */
93      gout(0xE4);
94      static uint8_t src[64];
95      static uint8_t dst[64];
96      for (uint32_t i = 0; i < sizeof(src); ++i) src[i] = (uint8_t)(i * 3 + 7);
97      mini_memcpy(dst, src, sizeof(src));
98
99      /* combine results to produce final bytes to GPIO */
100     uint8_t out1 = (uint8_t)((r1 ^ r2) & 0xFF);
101     uint8_t out2 = (uint8_t)((r3 >> 8) & 0xFF);
102     uint8_t out3 = (uint8_t)((r3) & 0xFF);
103
104     gout(out1);
105     gout(out2);
106     gout(out3);
107
108     /* Store a 64-bit marker in DMEM for offline verification */
109     rwu_store64( ((uint64_t)r1 << 32) ^ r2 ^ r3 ^ 0xDEADBEEFULL );
110
111     /* Final markers */
112     gout(0xE5);
113     gout(0x0A);
114
```

```
115     while (1) {
116         gout(0xFF);
117     }
118     return 0;
119 }
```

Program 2 – test.c (Square Test)

Listing B.2: test.c — square computation and conditional marker output

```
1  /* test.c
2   * Square test: compute squares in a loop and print markers.
3   */
4
5  #include "rwu-rv64i.h"
6  #include <stdint.h>
7
8  static int square(int x) {
9      return x * x;
10 }
11
12 int main(void) {
13     /* Start marker */
14     rwu_print(0xA0);
15
16     int sum = 0;
17     for (int i = 1; i <= 5; i++) {
18         int s = square(i);
19         sum += s;
20         rwu_print(0xA0 + i);   // 0xA1..0xA5
21     }
22
23     if (sum > 50)
24         rwu_print(0xB0);
25     else
26         rwu_print(0xB1);
```

```
27
28      int val = (sum ^ 0x55) & 0xFF;
29      rwu_print(val);
30
31      rwu_print(0xAF);
32      while (1) {
33          rwu_print(0xFF);
34      }
35      return 0;
36  }
```

Program 3 – test_simple_gpio.c

Listing B.3: test_simple_gpio.c — simple arithmetic and predictable GPIO pattern

```
1  /* test_simple_gpio.c
2   * Simple optimization test program for RWU-RV64I
3   * Expected GPIO output (decimal): 200,150,50,25,75,255...
4   */
5
6  #include "rwu-rv64i.h"
7  #include <stdint.h>
8
9  static inline void gout(uint8_t v) { rwu_print(v); }
10
11 int main(void) {
12      uint8_t a = 100;
13      uint8_t b = 50;
14      uint8_t c;
15
16      gout(200);   // Start marker (0xC8)
17
18      c = a + b;   // 150
19      gout(c);
20
21      c = a - b;   // 50
```

```
22      gout(c);
23
24      c = b / 2;   // 25
25      gout(c);
26
27      c = a ^ b;   // 86
28      c = c - 11;  // 75
29      gout(c);
30
31      while (1) {
32          gout(255);
33      }
34      return 0;
35  }
```

Each listing shows the exact C source as compiled during the optimization experiments (see Chapter 6).

Bibliography

[1] *Eclipse IDE for C/C++ Developers – User Guide*. Version 2023-09 (CDT Classic Interface). Eclipse Foundation. 2023. URL: https://help.eclipse.org/latest/topic/org.eclipse.cdt.doc.user.

[2] Free Software Foundation. *GNU Binutils Manual*. 2023. URL: https://sourceware.org/binutils/docs/.

[3] Free Software Foundation. *GNU Compiler Collection (GCC) Documentation*. 2024. URL: https://gcc.gnu.org/onlinedocs/.

[4] Digilent Inc. *Zybo Z7: Zynq-7000 ARM/FPGA SoC Development Board Reference Manual*. 2022. URL: https://digilent.com/reference/programmable-logic/zybo-z7/reference-manual.

[5] SiFive Inc. *SiFive RISC-V GNU Embedded Toolchain User Guide*. 2022. URL: https://sifive.com/software.

[6] David Patterson and Andrew Waterman. *The RISC-V Reader: An Open Architecture Atlas*. Strawberry Canyon LLC, 2017.

[7] *RISC-V GNU Toolchain Documentation*. RISC-V International Collaborative Projects. 2024. URL: https://github.com/riscv-collab/riscv-gnu-toolchain.

[8] ScienceDirect Topics. *Harvard Architecture*. Accessed: September 22, 2025. 2024. URL: https://www.sciencedirect.com/topics/engineering/harvard-architecture.

[9] Andreas Siggelkow. *RWU-RV64I Processor Repository*. https://github.com/asiggel/rwu-rv64i. Accessed October 2025. 2024.

[10] *Vivado Design Suite User Guide – Synthesis, Simulation and Implementation*. UG901, UG937. Xilinx Inc. 2024.

[11] Andrew Waterman and Krste Asanović. *The RISC-V Instruction Set Manual, Volume I: Unprivileged ISA*. Version 2.2. RISC-V International. 2021. URL: https://riscv.org/technical/specifications/.

[12] *Zybo Zynq-7000 Board Reference Manual*. Digilent Inc. 2023. URL: https://digilent.com.