SEU-RISCV-CPU 项目汇报

从软件到硬件的完整实现

项目名称：SEU-RISCV-CPU
作者：_____
学号：_____
指导老师：_____
学院/专业：_____

2026 年 1 月 8 日

摘要

本项目实现了一个从软件到硬件的完整 RISC-V 教学型 CPU 系统，覆盖 C 语言子集编译器、BIOS 固件、五级流水线 CPU 核心、SoC 外设与 FPGA 工程。系统支持基本的算术逻辑运算、分支跳转和字长访存，具备 4×4 键盘、数码管、LED、UART、定时器、PWM、看门狗等外设控制能力，展示了从源代码、汇编到 COE 初始化文件，再到 FPGA 上运行的完整链路。本报告围绕架构设计、指令集支持、编译器与 BIOS 实现、硬件微架构、冒险处理、存储映射、外设控制以及 FPGA 工程流程展开，并给出关键代码清单与图例占位，为后续复现与扩展提供系统化文档。

关键词：RISC-V；流水线；编译器；BIOS；FPGA；SoC

Abstract

This project delivers a full-stack RISC-V educational CPU system, spanning a C-subset compiler, BIOS firmware, a five-stage pipelined core, SoC peripherals, and an FPGA implementation flow. The system supports basic arithmetic/logic, branches, and word-level memory access, and integrates keypad, seven-seg display, LEDs, UART, timers, PWM, and watchdog. It demonstrates the end-to-end path from C source code to assembly, COE initialization, and FPGA execution. This report details the architecture, ISA subset, compiler/BIOS implementation, microarchitecture, hazard handling, memory mapping, peripheral controllers, and FPGA flow, with code listings and figure placeholders for future refinement.

Keywords: RISC-V, pipeline, compiler, BIOS, FPGA, SoC

目录

插图

表格

第一章　项目背景与目标

1.1　背景与意义

RISC-V 作为开源指令集架构，具备开放、可裁剪和生态友好的特点，非常适合作为教学与研究平台。为了让学生从软件到硬件完整理解处理器系统，本项目实现了一个“可运行、可编译、可上板”的最小可用系统，涵盖编译器、运行时固件、CPU 核心与外围设备，实现了从 C 代码到 FPGA 运行的完整链路。

传统教学往往在“软件”和“硬件”之间存在较大鸿沟：软件课程侧重语言与算法，硬件课程侧重电路与时序。一个完整的端到端系统可以把这两部分连接起来，使学生真正理解“高级语言如何变成电路上的信号变化”。这正是本项目的重要意义所在。

此外，RISC-V 的开放性使得学生可以自由查看并修改指令集与实现细节，从而形成更深入的理解，而不是停留在黑盒层面。

1.2　项目目标与交付物

本项目的目标不仅是实现一个能执行指令的 CPU 核心，更强调全栈链路的完整性。项目交付物包括：

1. C 语言子集编译器（Rust 实现），支持词法/语法/语义分析与代码生成；
2. BIOS 固件与系统服务层，封装显示、输入、串口、计时等功能；
3. RV32I 子集五级流水线 CPU 核心与 SoC；
4. 外设控制器与 MMIO 地址映射；
5. Vivado FPGA 工程与下载流程；
6. 示例应用（计算器、LED 波浪灯、拨码开关演示）。

从教学角度看，交付物的价值不仅在于“结果可用”，更在于“过程可学”：每个子模块都可以独立讲解与实验验证，组合在一起则形成完整系统。

1.3　设计原则与评价指标

本项目强调“可解释、可复现、可上板”的教学价值，因此在体系结构选择与工程实现上遵循以下原则：其一，结构清晰，模块边界明确，便于课堂讲解与后续扩展；其二，软硬件协同，编译器、BIOS 与硬件接口一一对应，避免“纸上架构”；其三，实验可操作，通过有限的外设完成典型输入输出任务。

评价指标方面，除了功能正确性外，还关注以下维度：流水线是否稳定、外设访问是否一致、编译器生成的代码是否可执行、FPGA 时序与资源是否满足约束。换言之，本项目既是“能跑起来的 CPU”，也是“能讲清楚的系统”。

1.4　设计约束与取舍

为了保证教学可理解性与可上板性，本项目做了若干取舍：

- 指令集为 RV32I 子集，仅包含 add/sub、逻辑运算、移位、基本分支、jal/jalr 与 lw/sw；
- 采用五级流水线结构（IF/ID/EX/MEM/WB），不引入缓存与乱序执行；
- 外设通过 MMIO 访问，以轮询为主，不依赖中断；
- 编译器支持 C 子集，避免复杂语义（结构体、浮点等）。

这些约束直接影响了系统复杂度与实现成本。例如，去除乘除与浮点可显著简化 ALU 与编译器；采用轮询方式可以避免中断控制器设计，使 BIOS 更加直观；而五级流水线则在性能与可解释性之间取得平衡，既能展示流水线概念，又不至于引入过多乱序与缓存细节。

图 1.1: 项目全栈流程示意图

第二章　系统总体架构与流程

2.1　总体架构

系统分为应用层、BIOS 服务层和硬件层三部分。应用层只关注业务逻辑，通过 BIOS 提供的函数访问硬件；BIOS 通过 MMIO 与硬件交互；硬件层包含 CPU 核心、存储器与外设控制器，组成一个小型 SoC。

从信息流角度看，应用层产生“需求”，BIOS 将其转换为“标准化的硬件访问”，硬件层只需保证 MMIO 读写语义正确即可。这样的分层不仅降低应用编写门槛，也让硬件在不修改应用的前提下可持续扩展。对于教学而言，学生可以分别理解“软件调用接口”“接口映射到寄存器”“寄存器驱动硬件”的完整链路。

从控制流角度看，CPU 负责执行指令序列，BIOS 则提供一组最小系统调用，使得应用逻辑不需要直接操作地址与位宽细节。该模式类似于“微型操作系统 + 裸机驱动”的混合模型，既保留了裸机可控性，又提供了友好的编程抽象。

图 2.1: miniRV SoC 顶层结构示意图

2.2　目录结构

2.2.1　目录树（节选）

为了便于上手与定位问题，仓库目录遵循“工具链—示例—硬件工程”三大块的组织方式。编译器源码集中在 src/，硬件工程位于 rvTest/，示例程序在 examples/。输出产物（汇编、COE）统一放入 output/，方便与 Vivado 工程衔接。

对于第一次接触的读者，建议从 QuickStart.md 入手完成上板流程，再回到 README.md 阅读架构说明，最后进入 src/ 与 rvTest/ 深入细节。

```
SEU-RISCV-CPU/
    Cargo.toml
    README.md
    QuickStart.md
    build.sh
    src/
        main.rs
        lexer.rs
        parser.rs
        codegen.rs
        assembler/
        linker/
    examples/
        bios_v2.c
        calculator_v2.c
        sw_led_demo.c
        led_wave.c
    output/
        calc_v2.s
        calc_v2.coe
    rvTest/
        rvTest.srcs/
    zeus_ide/
```

Listing 2.1: 项目目录树（节选）

表 2.1: 项目目录结构与说明

路径	说明
src/	编译器核心代码，包含词法、语法、语义分析、代码生成、汇编与链接。

续下页

路径	说明
examples/	BIOS 与应用示例源代码（计算器、LED、拨码开关等）。
output/	编译生成的汇编和 COE 文件示例。
rvTest/	Vivado 硬件工程与 RTL 源码（CPU 核心、外设、顶层）。
zeus_ide/	可选的编译器 IDE（Electron）。
QuickStart.md	从 C 到 FPGA 的快速上手指南。
README.md	详细项目文档与架构说明。

2.3　从 C 到 FPGA 的流程

编译与部署遵循以下流程。该流程体现了“从高级语言到硬件执行”的完整链路，每一步都会产生可验证的中间结果（汇编、COE、比特流），便于定位问题。

1. 编译 Rust 编译器；
2. 将 BIOS 与应用程序链接，生成 RISC-V 汇编；
3. 汇编与链接生成 COE 初始化文件；
4. 将 COE 文件拷贝至 Vivado 工程；
5. 综合、实现并生成比特流；
6. 下载至 FPGA 开发板运行。

其中，BIOS 与应用在源级链接，保证统一的入口与调用约定；而 COE 文件则是 Vivado 初始化 IROM/PRAM 的标准格式，使软件内容以“硬件可读”的方式进入 FPGA。

```
# 编译编译器
cargo build --release

# 编译 BIOS + 应用，生成汇编与 COE
./target/release/riscv_compiler examples/bios_v2.c examples/
   calculator_v2.c -o output/calc_v2

# 拷贝 COE 到 Vivado 工程
cp output/calc_v2.coe rvTest/rvTest.ip_user_files/mem_init_files/
   program.coe
```

Listing 2.2: 编译器与示例程序的基本使用

为保证流程可靠，建议在每一步进行最小验证：检查汇编是否生成、COE 条目数是否符合预期、Vivado 是否成功读取初始化文件，从而缩小错误定位范围。

图 2.2: 软件到硬件的流水线流程图

第三章　指令集与执行模型

3.1　RV32I 子集支持

CPU 核心实现了 RV32I 的一个可运行子集。指令选择以教学与实验为目标，覆盖算术逻辑、立即数、分支跳转与字长访存。编译器保证只产生硬件支持的指令。

子集的选择遵循“最小可用”原则：优先支持能够构成控制流与数据流的指令组合，使得常见的 C 语句（赋值、分支、循环、函数调用）能够被正确翻译与执行。这样即使指令数量不多，也可以覆盖较完整的编程实践。

表 3.1: 支持的指令子集（与控制器实现保持一致）

类别	指令
算术逻辑（R 型）	add, sub, and, or, xor, sll, srl, sra
立即数运算（I 型）	addi, andi, ori, xori, slli, srli, srai
访存	lw, sw
分支跳转	beq, bne, blt, bge, jal, jalr
其他	lui

3.2　指令格式与寄存器模型

指令遵循 R/I/S/B/U/J 六种基本格式，立即数在译码阶段进行符号扩展。寄存器文件为 32 个 32 位通用寄存器（x0 固定为 0），以同步写入、组合读取的方式实现。

从硬件实现角度看，指令格式的核心差异集中在立即数的拼接方式与寄存器字段的位置。通过统一的译码逻辑与符号扩展模块（SEXT），可以将不同格式的立即数转化为 32 位有符号数，供 ALU 或 NPC 使用。寄存器模型遵循 RISC-V 约定：x0 永远为 0，有利于零常数使用与简化硬件。

在调用约定上，系统采用简化的寄存器传参与栈帧机制，保证函数调用与返回路径清晰可控。这一设计同时考虑了教学可解释性与编译器实现难度。

图 3.1: R/I/S/B/U/J 指令格式示意图

3.3　执行模型与内存系统

处理器采用小端序，字长为 32 位。内存访问以字为单位，lw/sw 为核心数据访问指令。MMIO 地址空间映射外设寄存器，BIOS 通过内存读写实现硬件访问。

需要强调的是，本设计并未实现字节/半字访存，因此编译器在生成访存指令时以 32 位对齐为基本假设。对齐策略简化了存储器接口与数据通路，但也限制了可支持的数据结构类型。这样的取舍有助于教学理解，同时避免引入复杂的字节选通与对齐异常逻辑。

MMIO 的设计使得外设与内存共享统一的访问语义：软件层面只需读写地址，硬件层面通过 Bridge 完成译码与路由。这样既能保留“内存读写即外设控制”的直观性，又能保持 CPU 核心对外设的透明性。

3.4　汇编示例

下面给出编译器生成的 RISC-V 汇编片段，用于说明启动初始化与外设映射访问的基本结构（示意，不完整）：

```
.text
_start:
    lui sp, 0x00008          # 初始化栈指针
    addi sp, sp, -4
    lui t0, 0xFFFFF          # 外设基址
    addi t0, t0, -1024
    sw   a0, 0(t0)            # 向外设写入
    lw   a1, 0(t0)            # 从外设读取
```

```
9     jal  ra, user_main
```

Listing 3.1: 汇编片段示例（示意）

该片段展示了两个关键点：一是启动阶段需要初始化栈指针，以保证函数调用可用；二是外设访问仅依赖内存读写指令，通过外设基址完成读写操作。应用层与 BIOS 通过这种机制完成“软件到硬件”的桥接。

第四章　编译器设计与实现

4.1　总体架构

编译器以 Rust 实现，流程为：词法分析 → 语法分析 → 语义分析 → 代码生成 → 汇编 → 链接 →COE 输出。整体目标是将 C 子集转换为 RV32I 汇编，并完成简化的链接与内存布局。

这一流程被刻意设计为“透明可观测”：每一阶段都有可输出的中间结果（Token、AST、汇编文本），便于教学演示与调试。相比于黑盒式的编译器，本项目更强调可追踪性与可解释性。

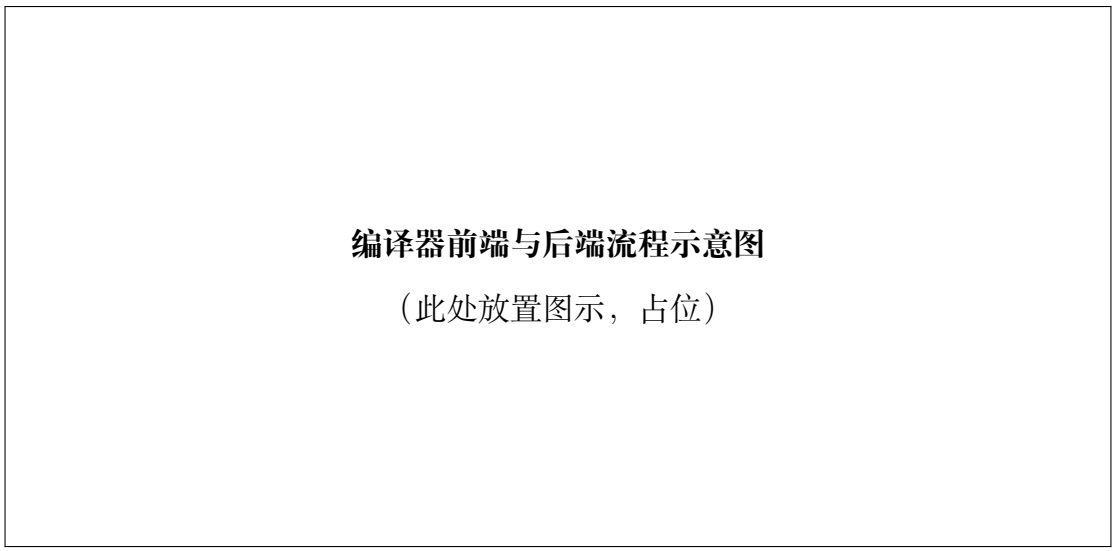

图 4.1: 编译器前端与后端流程示意图

4.2　词法与语法分析

词法分析将源码切分为 Token；语法分析构建 AST，并对表达式、语句与函数结构进行解析。为保证可用性，语法覆盖 if/while/for、函数调用与基本表达式。设计重点在于“够用且清晰”：只解析项目所需的语法结构，同时保证错误提示尽量指向具体位置。对于教学场景来说，清晰的语法错误比复杂语法覆盖更重要。

4.3　语义分析与代码生成

语义分析负责类型检查、符号表维护与作用域管理。代码生成根据 AST 输出 RV32I 指令序列，并处理函数调用约定、寄存器分配与简单栈帧布局。代码生成的关键是“把高级语义映射到有限指令集”。例如，乘法在硬件中未直接支持时，编译器通过调用 BIOS 软件乘法例程完成；数组与指针访问在生成阶段被转化为基址加偏移的访存指令。栈帧采用固定大小分配，降低了寄存器分配与栈布局的实现复杂度。

同时，为适配硬件的存储器时序，编译器在必要位置插入空操作（如 store 后的 load），避免出现简单的访存冒险。这体现了软硬协同的设计思想：硬件简化，软件补偿。

4.4　汇编器与链接器

汇编器负责将汇编文本转换为 ELF 结构；链接器根据段布局对符号进行重定位，并输出最终的 COE 文件。链接器将.text 段放置在 0x0000_0000 起始地址，并确保数据段不与 MMIO 冲突。由于目标环境是 FPGA 上的初始化存储器，链接器在布局阶段以“简单可控”为目标：段连续排列、按字对齐、地址范围可预测。这样既便于调试，也方便教学解释内存布局。

图 4.2: COE 输出格式示意图

第五章　BIOS 固件与软件运行时

5.1　BIOS 设计目标

BIOS 负责提供硬件抽象层，将 MMIO 细节封装为简单函数接口，应用程序仅需调用 BIOS 函数即可实现显示、输入、串口通信与外设控制。

BIOS 的核心价值在于“统一接口”。对于应用开发者而言，只需理解函数语义，而不必关心地址映射与寄存器位定义；对于硬件开发者而言，只需保证寄存器语义一致即可。这种分离显著降低了应用编写门槛，也提高了系统可维护性。

5.2　启动流程与 Bootloader

系统上电后，BIOS 负责初始化栈指针、外设自检，并根据拨码开关选择进入用户程序或 UART Bootloader。Bootloader 允许通过串口下载新程序并写入指定存储区域。

启动流程中包含的外设自检（如 LED/数码管点亮与键盘检测）不仅用于确认硬件连接，也为后续实验提供可靠的“已知正常”基线。Bootloader 的存在使得程序更新无需重新综合工程，提升了迭代效率。

图 5.1: BIOS 启动流程示意图

5.3　BIOS 接口列表（节选）

BIOS 接口可按功能划分为显示类（数码管）、输入类（键盘、按钮、拨码开关）、通信类（UART）、音频类（PWM/蜂鸣器）以及系统维护类（看门狗）。接口数量并不追求全面，而是覆盖教学实验常见需求。

表 5.1: BIOS 接口概要

函数	参数	返回值	说明
`bios_display_bcd`	int	void	数码管显示（支持负数）
`bios_key_read`	-	int	读取键盘（无按键返回 -1）
`bios_led_write`	int	void	写 24 位 LED
`bios_uart_putc`	char	void	发送单字符
`bios_uart_puts`	char*	void	发送字符串
`bios_uart_getc`	-	char	阻塞接收字符
`bios_buzzer_set`	int	void	设置 PWM 频率并开启
`bios_sw_read`	-	int	读取拨码开关状态
`bios_btn_read`	-	int	读取按钮状态
`bios_wdt_feed`	-	void	看门狗喂狗

5.4　应用示例：计算器

应用程序只需实现 `main()`，通过 BIOS 函数完成输入与显示，逻辑集中在按键解析与算术运算。

计算器示例体现了“输入—处理—输出”的典型嵌入式闭环：输入来自 4×4 键盘，处理在 CPU 内完成，输出显示在数码管。该示例既能验证键盘扫描与数码管显示功能，也能验证基础算术运算的正确性。

```
if (key == 10) { op = 1; input_mode = 2; }
if (key == 11) { op = 2; input_mode = 2; }
if (key == 12) { op = 3; input_mode = 2; }

if (key == 13) {
    if (op == 1) { result = num1 + num2; }
    if (op == 2) { result = num1 - num2; }
    if (op == 3) { result = bios_multiply(num1, num2); }
    bios_display_bcd(result);
}
```

Listing 5.1: 计算器应用片段（calculator_v2.c）

第六章　软硬件协同设计

6.1　协同设计目标

本项目的协同设计目标是“硬件尽量简单，软件适度补偿”。硬件侧保持指令集与数据通路的最小可用，实现清晰可讲的微架构；软件侧通过 BIOS 与编译器完成必要的功能补足，例如软件乘法、外设抽象与必要的延迟插入。

这种协同思路可以显著降低硬件实现复杂度，同时保证应用层仍能使用接近 C 的开发方式。对教学而言，它帮助学生理解“硬件能力有限时，软件如何承担补偿责任”。

6.2　调用约定与栈帧模型

调用约定采用简化版：参数优先使用寄存器传递，返回值保存在 `a0`，返回地址保存在 `ra`。为保持行为可预测，函数进入时分配固定大小栈帧并保存返回地址，函数退出时恢复。

固定栈帧虽然牺牲了一定的空间效率，但换来更清晰的控制流与更易观察的栈指针变化，便于实验与调试。

6.3　BIOS 系统调用表机制

BIOS 提供系统调用表，编译器在生成对 BIOS 函数的调用时，会将函数名映射为表项索引，再通过间接跳转完成调用。这样可以避免在应用代码中硬编码具体的 BIOS 地址，提升可维护性与扩展性。

从原理上看，这一机制类似于“动态链接表”，将应用与 BIOS 解耦，使得 BIOS 版本迭代时应用无需重新适配地址映射。

6.4　内存布局与软件补偿

编译器与链接器共同决定内存布局：`.text` 段从固定地址开始连续摆放，数据段与栈段保持明确边界，避免与 MMIO 地址冲突。由于硬件未实现完整的访存冒险处理，编译器在必要位置插入空操作，保证访存时序安全。

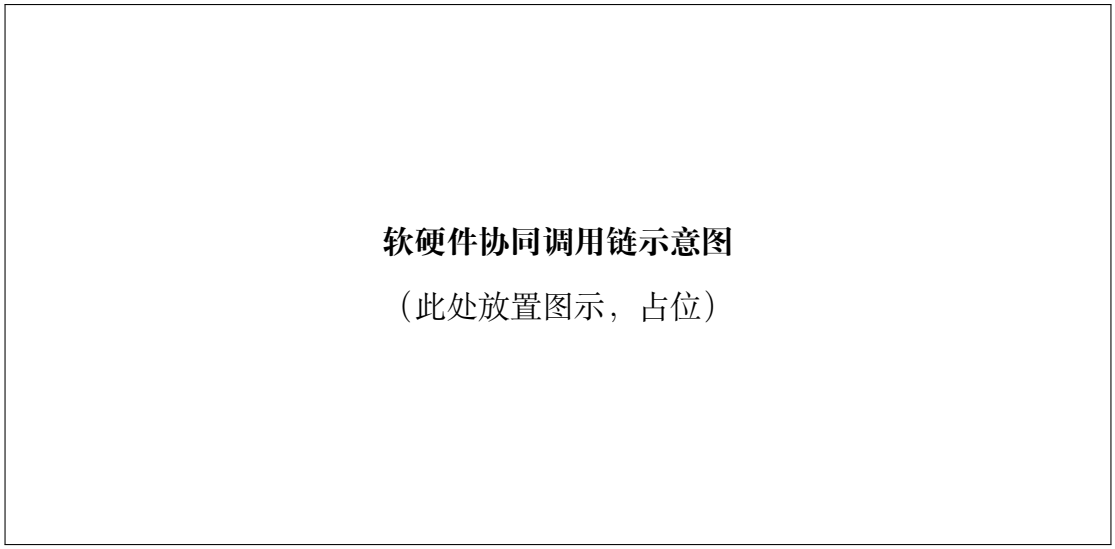

图 6.1: 软硬件协同调用链示意图

第七章　CPU 微架构设计

7.1　设计总览

本 CPU 以 RV32I 子集为目标，采用五级流水线结构，强调“结构清晰、信号明确、可教学复现”。微架构的核心思想是将指令执行过程拆分为若干阶段，每个阶段只完成相对单一的职责，从而降低组合逻辑复杂度并提高整体吞吐率。

为了保证可解释性，本设计避免引入缓存、乱序执行与分支预测等复杂机制。这样做的代价是性能上不追求极限，但可以换来“每条指令的执行路径都可被完整描述”的教学价值。

7.2　数据通路与控制通路分离

数据通路负责“数据的流动与运算”，控制通路负责“路径选择与时序调度”。在本设计中，数据通路由 PC/NPC、寄存器堆、SEXT、ALU、流水线寄存器、存储器接口等模块构成；控制通路由控制器与冒险检测逻辑构成。

控制信号在译码阶段生成，并通过流水线寄存器传递到后续阶段。典型控制信号包括：`npc_op`（下一 PC 选择）、`alu_op`（ALU 运算类型）、`alub_sel`（第二操作数选择）、`rf_we`（寄存器写使能）、`rf_wsel`（写回数据选择）、`ram_we`（访存写使能）等。控制通路与数据通路的分离，使得结构更易理解，扩展也更直接。

7.3　五级流水线结构

CPU 采用经典五级流水线：取指（IF）、译码（ID）、执行（EX）、访存（MEM）、写回（WB）。通过流水线寄存器实现阶段隔离，提高吞吐率。

五级流水线的价值在于让“多个指令并行推进”。每个周期不同指令处于不同阶段，从而提升平均吞吐率。对于教学而言，这种结构也便于拆解指令执行过程，使学生明确“取指—译码—执行—访存—写回”的因果关系。

阶段划分遵循“逻辑均衡”的原则：IF 负责取指与 PC 更新，ID 负责译码与寄存器读取，EX 负责计算与分支判断，MEM 负责数据访问，WB 负责结果写回。这样每个阶段的组合逻辑深度相对均衡，更容易满足 FPGA 时序约束。

由于系统未引入缓存，IF 与 MEM 直接访问片上存储器或外设。其优点是行为可预测、调试方便；缺点是访存延迟无法隐藏，需要通过流水线控制与编译器策略保证正确性。

图 7.1: 五级流水线结构示意图

7.4　取指与下一 PC 逻辑

PC 模块保存当前指令地址，NPC 模块根据分支、跳转或顺序执行计算下一 PC。分支在 EX 阶段确定，控制冒险通过冲刷流水线寄存器解决。

IF 阶段除了取指外，还承担基本的对齐与地址更新工作。NPC 逻辑综合了顺序执行（PC+4）与跳转/分支目标，保证指令流的连续性。由于分支结果在 EX 阶段才能确定，因此需要在前级保留“可能错误”的指令，并在确定后冲刷。

从时序上看，PC 在时钟上升沿更新；当检测到数据冒险需要停顿时，PC 保持不变；当控制冒险确认需要跳转时，PC 直接加载 NPC 计算结果，从而“纠正”指令流。该行为保证了取指阶段与冒险控制之间的正确协同。

NPC 的计算包含三类路径：顺序执行（PC+4）、分支/跳转目标（PC+offset）以及寄存器间接跳转（rs+imm）。在本设计中，分支与 J 类指令的目标使用 `pc+offset-8` 进行修正，用于补偿流水线中已取的两条指令；而 JALR 的目标地址直接来自 `rs_imm` 的计算结果。

复位时，PC 初始化为 0（调试模式下可能采用特殊初值以对齐 PC+4 逻辑），确保第一条指令从 IROM 起始地址获取。PC+4 既用于顺序取指，也作为 JAL/JALR 的返回地址写回，因此在 IF 阶段被提前计算并随流水线传递。

指令获取支持 IROM 与 PRAM 两个来源，通过选择信号决定“从哪一块存储器取指”。IROM 用于固化程序，PRAM 用于 Bootloader 下载后就地执行。这种双源取指机制既保证了系统启动的稳定性，又提供了快速更新程序的灵活性。

在实现上，PC 作为字节地址使用，指令存储器则以“字”为单位寻址，因此实际取指地址采用 `pc[15:2]` 或 `pc[13:2]` 的截取方式。高位地址 `pc[31:16]` 用于区分 IROM 与 PRAM 的取指区域，这使得 BIOS 与用户程序可以天然分区。

7.4.1　PC/NPC 实现片段与解释

PC 模块体现了“停顿优先于顺序推进、控制冒险优先于数据冒险”的策略：当检测到控制冒险时，PC 直接更新为 NPC；当检测到数据冒险时，PC 保持不变；否则按 NPC 正常更新。

```
always @(posedge clk or posedge rst) begin
  if (rst) pc <= 32'b0;
  else if (control_hazard) pc <= npc;
  else if (data_hazard) pc <= pc;
  else pc <= npc;
end
```

Listing 7.1: PC 更新逻辑（节选）

NPC 逻辑负责给出“下一条指令地址”。顺序执行取 PC+4，分支/跳转取 PC+offset，寄存器间接跳转取寄存器值。在本设计中采用 `pc+offset-8` 的修正项，用于补偿流水线中已取的两条指令。

```
always @(*) begin
  if (npc_op == NPC_JMPR) npc = rs_imm;
  else if (npc_op == NPC_JMP)  npc = pc + offset - 8;
  else if (npc_op == NPC_BEQ && br == 0) npc = pc + offset - 8;
  else if (npc_op == NPC_BNE && br != 0) npc = pc + offset - 8;
  else npc = pc + 4;
end
```

Listing 7.2: NPC 计算逻辑（节选）

7.4.2　IF/ID 与 ID/EX 寄存器片段

流水线寄存器需要支持“暂停”和“冲刷”：暂停用于数据冒险，冲刷用于控制冒险。以下片段展示了 IF/ID 与 ID/EX 在冒险发生时的处理方式。

```
always @(posedge clk or posedge rst) begin
  if (rst) ID_inst <= 32'b0;
  else if (control_hazard) ID_inst <= 32'b0;
  else if (data_hazard) ID_inst <= ID_inst;
  else ID_inst <= IF_inst;
end
```

Listing 7.3: IF/ID 暂停与冲刷（节选）

ID/EX 对控制信号与操作数做同样的处理，确保气泡被插入到 EX 阶段。

```
always @(posedge clk or posedge rst) begin
  if (rst) EX_rf_we <= 1'b0;
  else if (control_hazard | data_hazard) EX_rf_we <= 1'b0;
  else EX_rf_we <= ID_rf_we;
end
```

Listing 7.4: ID/EX 气泡插入（节选）

7.5　译码与寄存器堆

译码阶段解析 opcode 与 funct 字段，生成控制信号；寄存器堆支持同步写入与组合读取，满足流水线并行读取需求。

ID 阶段的关键在于把“指令位域”转换为“控制信号”。控制器根据 opcode/funct3/funct7 组合判断指令类型，并生成 ALU 操作、写回选择、立即数扩展类型等控制信号。寄存器堆的双读单写结构为执行阶段提供操作数，同时写回阶段在同一周期完成寄存器更新。

译码阶段还包含立即数生成（SEXT）逻辑。不同指令类型（I/S/B/U/J）具有不同的立即数字段拼接方式，SEXT 会根据 `sext_op` 选择拼接规则并进行符号扩展。对于分支与跳转指令，立即数的最低位固定为 0，以满足指令对齐要求。

指令字段拆解遵循 RV32I 规范：`opcode` 位于 `inst[6:0]`，`rd` 位于 `inst[11:7]`，`funct3` 位于 `inst[14:12]`，`rs1` 位于 `inst[19:15]`，`rs2` 位于 `inst[24:20]`，`funct7` 位于 `inst[31:25]`。控制器与 SEXT 都基于这些字段生成后续控制与数据路径。

寄存器堆遵循 RISC-V 约定：x0 固定为 0，读操作为组合逻辑，写操作在时钟沿更新。ID 阶段会根据指令类型判断是否需要读取源寄存器，读使能（`rf_re`）既用于节省无用读操作，也为冒险检测提供依据。

译码还会确定目的寄存器 `rd` 与是否写回。例如分支与存储指令不写回，因此 `rf_we` 被置 0；而加载、算术与跳转类指令需要写回，写回来源由 `rf_wsel` 指定。这样做使得“写回路径”在译码阶段就被确定，并在流水线中保持一致。

此外，ID 阶段输出的源寄存器编号、目的寄存器编号与读使能信号会被送入数据冒险检测模块，用于决定是否需要前递或停顿。通过这种“译码与冒险协作”的方式，流水线在保持高吞吐的同时避免数据错误。

7.5.1　立即数扩展示意代码

SEXT 模块根据指令类型拼接并扩展立即数，以下片段展示了不同格式的拼接方式（节选）：

```
assign ext = (sext_op == SEXT_I) ? {{20{sgn}}, din[24:13]} :
             (sext_op == SEXT_S) ? {{20{sgn}}, din[24:18], din[4:0]}
   :
             (sext_op == SEXT_B) ? {{19{sgn}}, din[24], din[0], din
   [23:18], din[4:1], 1'b0} :
             (sext_op == SEXT_U) ? {din[24:5], 12'b0} :
             (sext_op == SEXT_J) ? {{11{sgn}}, din[24], din[12:5],
   din[13], din[23:14], 1'b0} :
             32'b0;
```

Listing 7.5: SEXT 立即数扩展（节选）

7.6　执行阶段（EX）

执行阶段由 ALU 作为核心运算单元，完成算术逻辑运算、地址计算与分支比较。ALU 的两个操作数来自寄存器堆读出值与立即数扩展值，通过 `alub_sel` 选择第二操作数来源。运算结果 `alu_c` 与分支标志 `br` 被送入后续阶段。

对于算术逻辑指令，EX 直接产生最终结果；对于访存指令，EX 计算有效地址（`rs1 + imm`）；对于分支指令，EX 基于减法比较结果产生分支条件标志；对于 JAL/JALR，EX 阶段计算跳转目标或相关控制信号，PC+4 作为返回地址在后续阶段写回。

分支判断依赖 ALU 的比较标志。ALU 在执行减法时同时产生比较结果：相等、符号小于或符号大于。控制冒险检测逻辑依据该标志与分支类型（BEQ/BNE/BLT/BGE）决定是否跳转，从而形成“算术比较—控制决策”的闭环。

移位指令采用低 5 位作为移位量，这与 RV32I 指令规范一致。算术右移采用符号扩展，保证负数右移语义正确。

7.6.1　ALU 运算与分支标志片段

ALU 的核心是运算选择与比较标志生成。运算结果用于写回或访存地址，比较标志用于分支判断。

```
B = (alub_sel == ALU_Data_Imm) ? imm : rs2;
case (alu_op)
  ALU_ADD: C_tmp = A + B;
  ALU_SUB: C_tmp = A - B;
  ALU_AND: C_tmp = A & B;
  ALU_OR : C_tmp = A | B;
```

```
  ALU_XOR: C_tmp = A ^ B;
  ALU_SLL: C_tmp = A << B[4:0];
  ALU_SRL: C_tmp = A >> B[4:0];
  ALU_SRA: C_tmp = $signed(A) >>> B[4:0];
endcase

if (alu_op != ALU_SUB) br_tmp = 0;
else if ($signed(A) == $signed(B)) br_tmp = 0;
else if ($signed(A) < $signed(B)) br_tmp = 1;
else br_tmp = 2;
```

Listing 7.6: ALU 运算与比较（节选）

7.7　访存阶段（MEM）

访存阶段通过 Bridge 统一访问 DRAM 或外设。若为存储指令，`Bus_we` 置 1，地址来自 EX 计算结果，写数据来自寄存器第二操作数；若为加载指令，则从 `Bus_rdata` 取回数据并送往写回阶段。

在外设访问时，Bridge 会将地址译码为对应外设寄存器，并返回相应读数据或执行写入动作。该机制保证了 CPU 对 DRAM 与外设的访问在语义上保持一致，软件无需区分“普通内存”与“外设寄存器”。

需要注意的是，DRAM 采用同步写入、组合读出的模型。对于紧邻的 store-load 序列，若没有适当的停顿，可能读取到旧值。为简化硬件，本项目将该类时序风险部分交由编译器处理，通过插入空操作保证数据一致性。

访存接口以 32 位数据宽度为单位，地址对齐到字边界。这样可以简化存储器接口与外设寄存器设计，但也意味着字节/半字访存不在当前硬件支持范围内。

7.8　写回阶段（WB）

写回阶段根据 `rf_wsel` 选择写回数据来源，常见来源包括 ALU 结果、访存结果、PC+4 或立即数扩展值（如 LUI）。写回通过 `rf_we` 控制，确保对分支/存储等不需要写回的指令关闭写使能。

寄存器堆遵循 x0 恒为 0 的规则，因此即使写回逻辑给出写地址 0，也不会改变 x0 的值。这一约束简化了软件常量的使用。

从指令角度看：算术逻辑类写回 ALU 结果；加载指令写回访存数据；JAL/JALR 写回 PC+4 作为返回地址；LUI 写回扩展后的立即数。这种“写回来源与指令类型的一一对应”让控制逻辑保持直观。

7.9　流水线寄存器与阶段隔离

流水线寄存器用于保存阶段间的“数据与控制信号快照”，保证每条指令在流水线中独立推进。以下列出各阶段寄存器保存的关键内容（概览）：

表 7.1: 流水线寄存器关键内容（概览）

寄存器	保存内容（示例）
IF/ID	指令字、PC+4，用于后续译码与分支处理。
ID/EX	控制信号（`npc_op`, `alu_op`, `rf_we` 等）与操作数（`rD1`, `rD2`, `ext`）。
EX/MEM	计算结果（`alu_c`）、写数据、写回控制信号。
MEM/WB	访存读出数据、ALU 结果、PC+4 以及写回控制信号。

当出现数据冒险时，PC 与 IF/ID 保持不变，ID/EX 被清零以插入气泡；当出现控制冒险时，IF/ID 与 ID/EX 被清零以冲刷流水线。该策略简单直观，适合教学验证。

控制冒险通常具有更高优先级：一旦确认跳转成立，必须立即修正指令流，避免错误路径提交；数据冒险则通过停顿保证依赖关系。这样的优先级安排与硬件实现保持一致，也更符合直觉。

7.10　关键控制信号说明

为便于理解数据通路的“选择逻辑”，下表列出主要控制信号及其功能含义：

表 7.2: 控制信号与功能说明

信号	功能说明
`npc_op`	选择下一 PC 计算方式（PC+4、分支、跳转、寄存器间接跳转）。
`alu_op`	选择 ALU 运算类型（加减与逻辑运算等）。
`alub_sel`	选择 ALU 第二操作数来源（寄存器或立即数）。
`sext_op`	选择立即数扩展格式（I/S/B/U/J）。
`rf_we`	寄存器写回使能。
`rf_wsel`	选择写回来源（ALU、访存、PC+4、SEXT）。
`ram_we`	访存写使能（store 指令）。
`rf_re`	寄存器读使能，用于冒险检测与读端口控制。

7.11　模块级划分与职责

CPU 内部由多个独立模块组成，模块化设计有助于分别理解各部分功能并进行单元级调试。下表列出关键模块及其职责：

表 7.3: CPU 核心模块与职责

模块	主要职责
PC	保存并更新当前指令地址，支持停顿与跳转修正。
NPC	计算下一条指令地址（PC+4、分支/跳转目标、寄存器间接跳转）。
IF/ID	保存取指结果与 PC+4，支持冲刷与暂停。
ID/EX	保存译码后的控制信号与操作数，支持气泡插入。
EX/MEM	保存 EX 阶段计算结果与访存控制信号。
MEM/WB	保存访存结果与写回控制信号。
RF	32×32 寄存器堆，双读单写。
SEXT	立即数字段拼接与符号扩展。
ALU	算术逻辑运算与分支比较标志产生。
controller	指令译码与控制信号生成。
Data_Hazard_Detection	检测数据相关并产生前递/停顿控制。
Control_Hazard_Detection	检测分支/跳转是否成立并触发冲刷。
Bridge	地址译码与内存/外设路由。

7.12　指令流动的周期级示例

以一条 `add` 指令为例：第 1 个周期在 IF 取指；第 2 个周期在 ID 译码与读寄存器；第 3 个周期在 EX 完成加法；第 4 个周期在 MEM 阶段“空转”通过；第 5 个周期在 WB 写回寄存器。若后续指令与其无数据相关，则可在流水线中并行推进，实现高吞吐。

若后续指令依赖 `lw` 的结果，则在 EX 阶段检测到 load-use 冒险，PC 与 IF/ID 被暂停一个周期，ID/EX 插入气泡。通过这一机制保证正确性，同时让学生观察到“流水线性能与相关性”的直接关系。

对于分支指令（如 `beq`），指令在 EX 阶段完成比较并决定是否跳转。若跳转成立，则 IF/ID 与 ID/EX 被清空，两条已取指令被丢弃，随后从目标地址重新取指。这一过程体现了控制冒险带来的流水线“清空代价”。

7.13　不同指令类型的执行路径

为了更直观地理解控制信号与数据通路的对应关系，可将常见指令类型的执行路径概括如下：

R 型算术逻辑：ID 生成 `alub_sel=reg`、`rf_wsel=ALU`、`rf_we=1`；EX 完成运算；MEM 透传；WB 写回 ALU 结果。

I 型算术逻辑：与 R 型类似，但 `alub_sel=imm`，EX 使用立即数参与运算。

加载（lw）：EX 计算有效地址；MEM 读取数据；WB 选择访存结果写回；`rf_we=1`。

存储（sw）：EX 计算有效地址；MEM 将寄存器数据写入存储器；WB 不写回（`rf_we=0`）。

分支（beq/bne/blt/bge）：EX 完成比较并决定是否跳转；若跳转成立则冲刷流水线；WB 不写回。

跳转（jal/jalr）：NPC 选择跳转目标；WB 写回 PC+4 作为返回地址；`rf_we=1`。

图 7.2: 数据通路与前递示意图

第八章　控制逻辑与冒险处理

8.1　控制信号生成

控制器根据 opcode/funct 组合生成控制信号，决定 ALU 运算类型、寄存器写回来源、立即数扩展方式与访存控制。

从原理上看，控制器将“指令语义”映射为“数据通路选择”。例如，同一条 add 指令需要选择寄存器作为 ALU 两个输入、在 WB 阶段选择 ALU 输出写回，而 lw 指令需要选择立即数形成访存地址并在 WB 阶段选择访存数据。该过程本质上是对多路选择器与使能信号的统一调度。

```
assign alu_op   = (inst_add | inst_addi | inst_sw | inst_lw |
   inst_jalr) ? `ALU_ADD :
                 (inst_and | inst_andi)
      ? `ALU_AND :
                 (inst_or  | inst_ori)
      ? `ALU_OR  :
                 (inst_xor | inst_xori)
      ? `ALU_XOR :
                 (inst_sll | inst_slli)
      ? `ALU_SLL :
                 (inst_srl | inst_srli)
      ? `ALU_SRL :
                 (inst_sra | inst_srai)
      ? `ALU_SRA : `ALU_SUB;
```

Listing 8.1: 控制器片段（controller.v）

8.2　数据冒险处理

数据冒险通过前递与停顿两种手段解决。若后续指令依赖前一条 load 的结果，流水线将插入气泡以等待数据就绪。

前递机制能够将 EX/MEM/WB 阶段的最新结果直接送回 EX 阶段，减少不必要的停顿。只有在 load-use 情况下（数据尚未从内存返回）才需要停顿。通过这种“优先前

递、必要停顿”的策略，在保持硬件复杂度可控的同时获得较高的实际吞吐率。

具体而言，数据冒险检测比较 ID 阶段的源寄存器与 EX/MEM/WB 阶段的目的寄存器，若匹配且写回使能有效，则触发前递。前递优先级通常为 EX > MEM > WB，确保使用最新结果。若 EX 阶段是加载指令（写回来源为访存数据），则即使存在匹配也无法前递数据，需要插入停顿。

在本实现中，停顿的表现为：PC 与 IF/ID 寄存器保持不变，ID/EX 寄存器被清零（插入气泡），使得冒险指令延后一周期进入 EX，从而等待数据有效。

8.2.1　数据冒险检测与前递片段

数据冒险检测主要关注“ID 阶段源寄存器是否与前级目的寄存器冲突”。以下片段展示了 load-use 停顿条件与前递选择的基本形式：

```
assign data_hazard =
  (rR1_a && EX_rf_wsel == WB_DM) ||
  (rR2_a && EX_rf_wsel == WB_DM);

always @(*) begin
  if (rR1_a)      new_rD1 = EX_alu_c;
  else if (rR1_b) new_rD1 = MEM_alu_c;
  else if (rR1_c) new_rD1 = WB_alu_c;
  else            new_rD1 = ID_rD1;
end
```

Listing 8.2: 数据冒险检测与前递（节选）

8.3　控制冒险处理

分支/跳转指令在 EX 阶段确定，IF/ID 与 ID/EX 阶段会被冲刷，确保错误路径指令不会提交。

由于未引入分支预测，控制冒险采用“确定后清空”的方式处理。这种策略简单直观，适合教学场景；代价是分支指令会带来固定的流水线气泡，但可通过实验观察到分支指令对性能的影响，从而加深理解。

具体流程为：当 EX 阶段判断分支/跳转成立时，控制冒险信号置位，PC 立即更新为 NPC，IF/ID 与 ID/EX 清零，从而丢弃已取到的错误路径指令。该机制确保控制流正确，但会产生可观测的流水线气泡，便于分析控制冒险成本。

8.3.1　控制冒险检测片段

控制冒险检测通过 EX 阶段的分支标志与指令类型判断是否需要跳转，从而触发冲刷：

```
always @(*) begin
  if (EX_npc_op == NPC_JMPR || EX_npc_op == NPC_JMP)
    control_hazard = 1'b1;
  else if ((EX_npc_op == NPC_BEQ && alu_f == 0) ||
           (EX_npc_op == NPC_BNE && alu_f != 0) ||
           (EX_npc_op == NPC_BLT && alu_f == 1) ||
           (EX_npc_op == NPC_BGE && alu_f != 1))
    control_hazard = 1'b1;
  else
    control_hazard = 1'b0;
end
```

Listing 8.3: 控制冒险检测（节选）

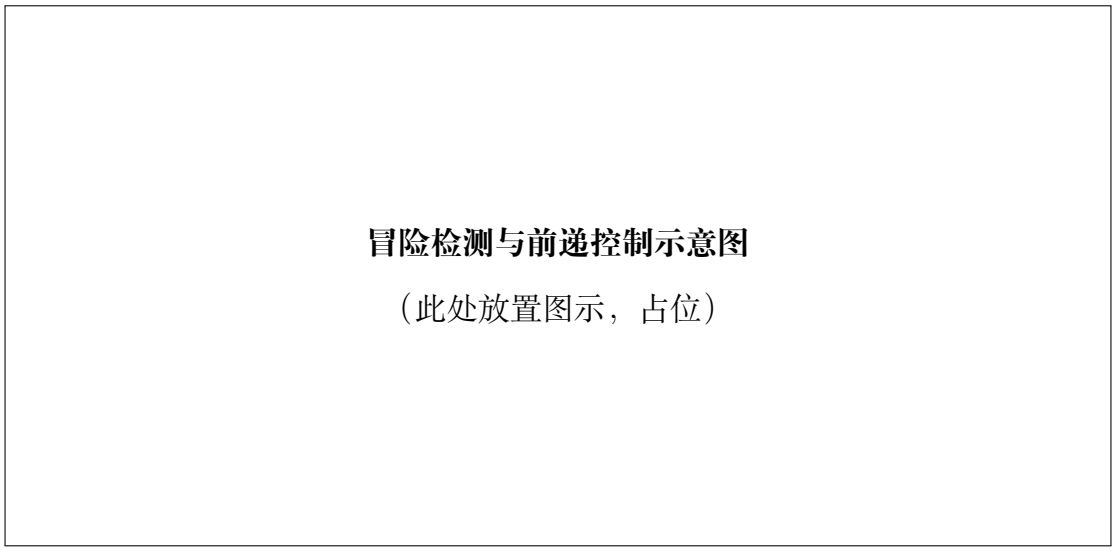

图 8.1: 冒险检测与前递控制示意图

第九章　流水线性能与时序分析

9.1　理想吞吐与 CPI

在理想情况下，五级流水线能够实现每周期完成一条指令的吞吐率，此时 CPI（Cycles Per Instruction）趋近于 1。然而实际系统中存在数据冒险与控制冒险，使得 CPI 大于 1。

可用如下近似关系理解性能：

$$\mathrm{CPI} \approx 1 + \mathrm{stall\_rate} + \mathrm{branch\_rate} \times \mathrm{branch\_penalty}$$

该公式虽然简化，但能帮助学生建立“冒险数量影响性能”的直观认识。

9.2　分支惩罚分析

本设计在 EX 阶段确定分支，因此在分支指令之后的若干周期可能出现气泡。若不引入分支预测，分支惩罚是固定的，这在性能上有一定代价，但结构上非常清晰，适合教学。

9.3　Load-Use 冒险与停顿

当指令依赖前一条 `lw` 的结果时，数据在 MEM 阶段才可用，必须插入一个或多个停顿周期。通过前递可以减少大部分数据冒险，但 load-use 仍然是必须停顿的典型情况。

9.4　时序与关键路径

关键路径通常出现在“寄存器读出—ALU 计算—结果选择—寄存器写回”的组合路径中。通过合理的流水线分段与寄存器隔离，可以在保证功能正确的前提下满足时序约束。本项目在 50MHz 下满足实现时序，适合教学板卡运行。

图 9.1: 流水线时序波形示意图

第十章　存储系统与总线设计

10.1　IROM 与 DRAM

指令存储器 IROM 与数据存储器 DRAM 使用 Xilinx 分布式存储器 IP 实现，深度为 16K words（约 64KB）。IROM 只读，DRAM 可读写。

为了简化设计，本项目将指令与数据存储器分离，形成哈佛风格的结构：指令通路与数据通路相互独立，便于理解流水线的取指与访存阶段。IROM 只读使得指令路径更清晰，而 DRAM 支持读写以承载变量与堆栈。

10.2　PRAM 与 Bootloader

项目包含 PRAM 作为可写程序区，BIOS 的 UART Bootloader 可将新程序写入该区域，实现在线更新。

PRAM 的引入降低了迭代成本。即使硬件不变，也可以通过串口快速更新程序内容，适合课程实验中的频繁修改与验证。

10.3　总线桥与地址解码

Bridge 模块负责将 CPU 访存地址解码为 DRAM 或外设访问请求，并将对应设备的数据返回给 CPU。

该模块是“统一访问语义”的关键：CPU 只需发出地址、读写信号与数据，而 Bridge 负责将这些请求分发至正确设备。这样，外设扩展只需在 Bridge 中新增地址范围与端口连接，而无需修改 CPU 内核。

```
wire access_mem = (addr[31:12] != 20'hFFFFF);
wire access_led = (addr == 32'hFFFF_FC60);
wire access_keypad = (addr[31:4] == 28'hFFFF_FC1);
```

Listing 10.1: 地址解码片段（Bridge.v）

10.4　MMIO 地址映射

表 10.1: MMIO 地址映射表

地址范围/地址	设备	读写	描述
0x0000_0000–0x0000_FFFF	DRAM	R/W	数据存储器（64KB）
0x0001_0000–0x0001_FFFF	PRAM	R/W	程序区（Bootloader）
0xFFFF_FC00	数码管	W	32 位 BCD 显示
0xFFFF_FC10	键盘数据	R/W	键值或 0xFFFFFFFF
0xFFFF_FC12	键盘状态	R	bit0: 按键标志
0xFFFF_FC20	定时器 0	R/W	计数器值
0xFFFF_FC24	定时器 N	R/W	重载值
0xFFFF_FC30	PWM 最大值	W	蜂鸣器频率
0xFFFF_FC34	PWM 比较值	W	占空比
0xFFFF_FC38	PWM 控制	W	bit0= 使能
0xFFFF_FC40	UART 数据	R/W	读 = 接收/写 = 发送
0xFFFF_FC44	UART 状态	R	TX 忙/RX 就绪
0xFFFF_FC48	UART 控制	W	写 bit0 清除 RX 就绪
0xFFFF_FC50	看门狗	R/W	读计数/写喂狗
0xFFFF_FC60	LED	W	24 位 LED 控制
0xFFFF_FC70	开关	R	24 位开关状态
0xFFFF_FC78	按钮	R	5 位按钮状态

MMIO 表将“地址空间”与“功能模块”建立了清晰对应关系。对应用开发者而言，读写特定地址即可驱动外设；对硬件开发者而言，只需保证对应地址的寄存器语义稳定，软件即可保持兼容。

图 10.1: MMIO 总线与外设连接示意图

第十一章　外设控制器设计

11.1　数码管（7-Seg）

采用动态扫描方式驱动 8 位数码管，支持 BCD 编码显示与扩展字符。显示更新频率约 10kHz，以避免闪烁。

动态扫描的原理是“分时复用”：同一时刻只点亮一位数码管，通过快速轮询形成稳定视觉效果。其实现关键在于扫描时钟与段码译码表，既要保证刷新频率，又要保证编码一致性。

11.2　4×4 矩阵键盘

键盘控制器以行列扫描方式获取按键，使用同步器与去抖逻辑提高可靠性，并将按键值锁存供软件读取。

矩阵键盘具有“行列复用”的硬件优势，但也带来去抖与锁存需求。控制器通过多周期确认与锁存机制，避免机械抖动引起的多次触发，从而保证输入稳定。

11.3　LED / Switch / Button

LED 为 24 位写寄存器控制；Switch/Button 通过读取状态寄存器获得输入。

这类外设是最简单也最直观的 I/O 实验对象。通过写 LED 与读开关/按钮，学生可以快速验证 MMIO 读写路径是否正确，并观察硬件响应与软件逻辑之间的对应关系。

11.4　UART

UART 提供基本的串口通信能力，支持阻塞读写与状态寄存器查询，用于调试与 Bootloader 传输。

UART 模块的核心是波特率分频与收发状态机。其设计强调稳定性与可预测性，使得串口既可用于交互调试，也可用于 Bootloader 的程序传输。

11.5　Timer / PWM / WDT

Timer 以计数/重载模式运行；PWM 驱动蜂鸣器输出音调；看门狗用于系统复位与稳定性保障。

Timer 为系统提供基础的时间尺度，PWM 将“计数比较”转化为占空比信号，可用于声音或电机控制；WDT 则在系统异常时触发复位，保证“可恢复性”。这些模块共同构成系统运行的基础保障。

图 11.1: 外设控制器结构示意图

第十二章　FPGA 实现与约束

12.1　Vivado 工程与时钟系统

硬件工程基于 Vivado 2017.4，输入时钟为 100MHz，通过 PLL 生成 CPU 时钟，默认约 50MHz。调试模式可直接使用外部时钟。

时钟规划的目标是“稳定与可控”。将输入时钟通过 PLL 分频得到较低频率，有助于满足时序约束并降低调试难度。调试模式直连外部时钟，则便于观察波形与单步追踪。

12.2　资源利用率（综合）

根据工程自带的综合报告，核心资源占用较低，适合教学板卡。

低资源占用意味着本设计可以运行在较小的 FPGA 器件上，也为后续扩展（如加入缓存、更多外设）预留了空间。资源统计同时提供了“设计规模”的量化指标。

表 12.1: 综合资源利用率（miniRV_SoC_utilization_synth.rpt）

资源类型	Used	Available	Util%
Slice LUTs	2286	63400	3.61
Slice Registers	2315	126800	1.83
Block RAM	0	135	0.00
DSP	0	240	0.00
BUFGCTRL	1	32	3.13
Bonded IOB	82	285	28.77

12.3　时序结果（实现）

实现报告显示设计满足时序约束，WNS 约 2.776ns，CPU 时钟约 50MHz。

从结果看，时序裕量为正，表明当前频率下路径延迟满足约束。对于教学项目，能稳定跑在 50MHz 即可满足实验需求，同时避免频率过高带来的不确定性。

表 12.2: 实现时序摘要（miniRV_SoC_timing_summary_routed.rpt）

指标	数值	说明
WNS	2.776ns	Worst Negative Slack
WHS	0.067ns	Worst Hold Slack
CPU 时钟	50MHz	PLL 输出

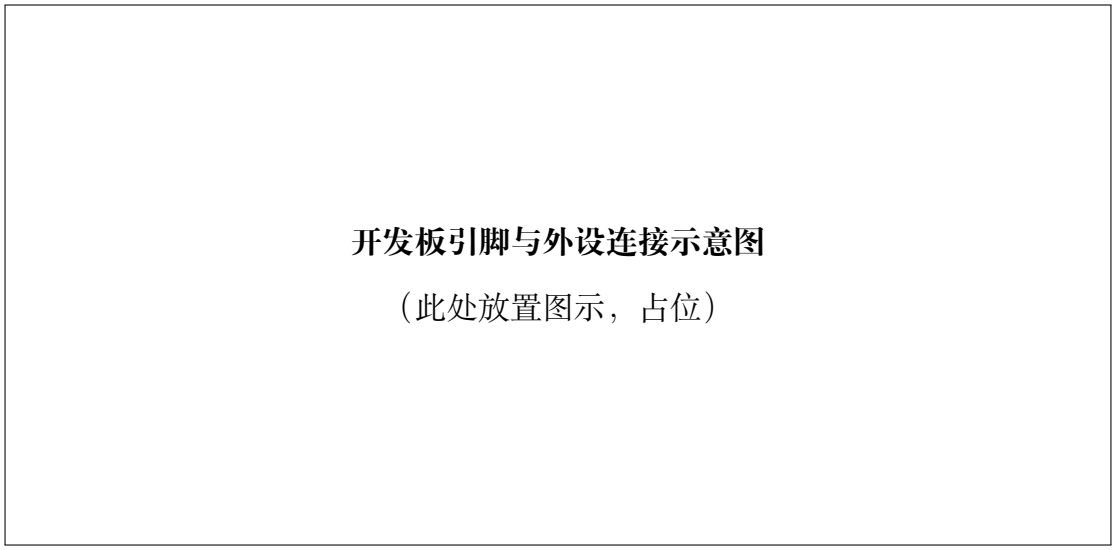

图 12.1: 开发板引脚与外设连接示意图

第十三章　调试与验证方法论

13.1　分层调试思路

调试策略建议遵循“由下到上”的分层思路：先验证时钟与复位是否正确，再验证最简单的外设（LED/数码管），再验证输入外设（按键/开关），最后验证复杂逻辑（计算器与 UART）。这种分层方法可以快速缩小问题范围，避免在未知状态下进行复杂排查。

13.2　可观测点设计

在硬件侧，可通过调试写回接口观察指令提交与寄存器写回；在软件侧，可通过 UART 输出关键状态。调试信号如 `debug_wb_*` 可以帮助定位“指令是否执行、是否写回正确寄存器”的问题。

13.3　串口调试与日志

UART 是教学中最常见的调试手段。通过打印关键变量、寄存器值或阶段标记，能够在不依赖波形的情况下完成基本定位。配合 BIOS 的串口函数，可快速形成调试闭环。

13.4　常见问题与定位建议

常见问题包括：MMIO 地址不一致、栈指针初始化错误、分支目标计算错误、外设时钟未正确分频等。定位时建议优先检查“地址映射—控制信号—外设响应”三条链路，逐层验证。

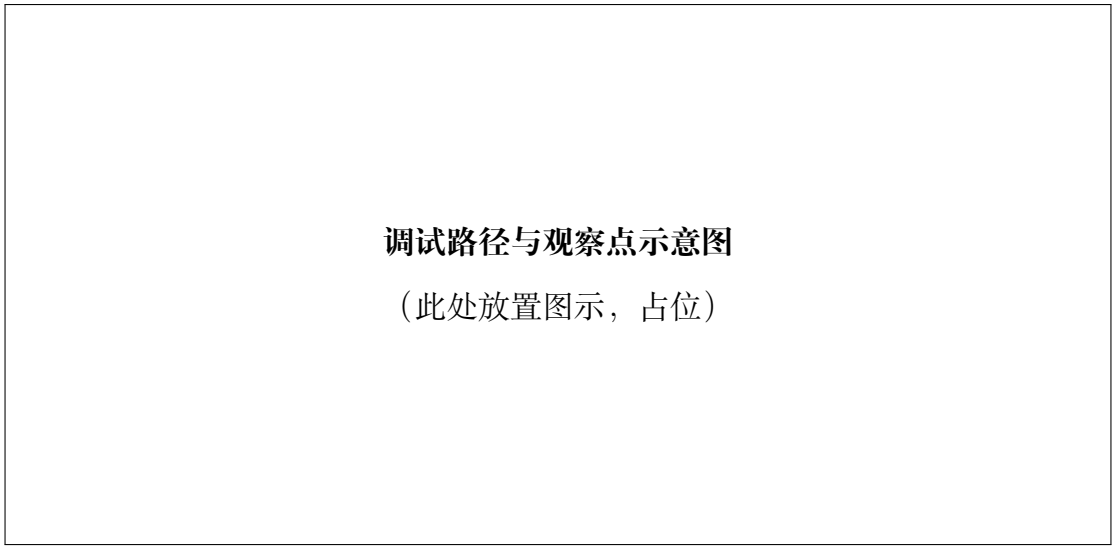

图 13.1: 调试路径与观察点示意图

第十四章　软件示例与实验验证

14.1　示例程序

项目提供计算器、LED 波浪灯、拨码开关控制等示例，用于验证输入/输出与 BIOS API 的稳定性。

示例的设计遵循“由易到难”：先验证输出（LED/数码管），再验证输入（键盘/开关），最后验证综合逻辑（计算器）。这种渐进式的实验路径有利于快速定位问题所在。

14.2　功能验证方法

通过以下步骤进行验证：

1. 上板后进行 BIOS 自检（数码管、LED、键盘）；
2. 运行计算器，测试加减乘与负数显示；
3. 使用拨码开关控制 LED，验证 MMIO 读写；
4. UART 输出日志，检查串口通信稳定性。

在教学实践中，建议每个示例记录“预期现象—实际现象—可能原因”，通过对比建立对系统行为的理解。这类记录不仅帮助调试，也为后续报告撰写提供依据。

图 14.1: 示例程序运行效果照片占位

第十五章　教学实验设计与扩展

15.1　实验目标与层次

教学实验可分为“指令级验证”“外设级验证”“系统级验证”三层：指令级验证关注单条指令的执行语义，外设级验证关注 MMIO 读写路径，系统级验证则关注多模块协同与应用功能。

15.2　推荐实验列表

表 15.1: 教学实验与目标示例

实验名称	目标与说明
指令验证实验	验证 add/sub/branch 等指令执行正确性，观察寄存器写回结果。
LED 输出实验	通过循环移位控制 LED，验证写寄存器路径与时序效果。
键盘输入实验	读取矩阵键盘并在数码管显示，验证输入路径与扫描逻辑。
UART 调试实验	使用串口输出调试信息，验证通信可靠性与 BIOS 接口。
Bootloader 实验	通过串口下载程序到 PRAM，实现在线更新与执行。

15.3　实验扩展建议

在完成基础实验后，可扩展如下内容：增加新的 MMIO 外设、加入简单中断机制、实现更复杂的应用程序（如计时器驱动的时钟显示）。这些扩展不仅能锻炼硬件设计能力，也能强化软件与硬件协同的理解。

图 15.1: 教学实验流程示意图

第十六章　总结与展望

本项目以教学为目标，实现了从编译器到硬件的完整 RISC-V 系统。其优势在于结构清晰、模块化强、可上板验证，并提供丰富示例程序。未来可在以下方向进一步扩展：

- 扩展 RV32I 指令支持（如乘除、字节/半字访问）；
- 引入中断与异常机制；
- 增加缓存与性能优化；
- 丰富外设与图形显示模块；
- 完善 IDE 与调试工具链。

总体而言，本项目不仅展示了“指令执行”的硬件层实现，也展示了“从高级语言到硬件执行”的完整路径。通过编译器、BIOS 与外设的配合，学生可以清晰理解软硬件协同的关键概念。后续扩展可以围绕“性能优化”“可观测性提升”和“功能完整性”三条主线展开。

本附录部分提供若干“查阅型”信息，便于实验与调试时快速定位：包含编译命令速查、指令清单、BIOS API 与 MMIO 地址映射。正文已给出原理说明，附录更多用于实践操作时参考。

附录 A　编译与运行命令速查

```
# 编译编译器
cargo build --release

# 生成汇编与 COE
./target/release/riscv_compiler examples/bios_v2.c examples/
   calculator_v2.c -o output/calc_v2

# 仅生成汇编
./target/release/riscv_compiler examples/bios_v2.c examples/
   calculator_v2.c -S -o output/calc_v2.s
```

Listing A.1: 编译器与示例程序流程

附录 B　指令集支持清单（详细）

表 B.1: 指令分类与说明

类别	支持指令
算术/逻辑（R 型）	add, sub, and, or, xor, sll, srl, sra
立即数（I 型）	addi, andi, ori, xori, slli, srli, srai
访存（字）	lw, sw
分支	beq, bne, blt, bge
跳转	jal, jalr
高位立即数	lui

附录 C　BIOS API 完整清单

表 C.1: BIOS API（完整版）

函数	参数	返回值	说明
bios_display_bcd	int	void	数码管显示（负号支持）
bios_key_read	-	int	读取键盘键值
bios_led_write	int	void	LED 写入
bios_multiply	int,int	int	软件乘法
bios_mul10	int	int	乘 10 运算
bios_delay	int	void	延时循环
bios_wdt_feed	-	void	看门狗喂狗
bios_uart_putc	char	void	UART 发送字符
bios_uart_puts	char*	void	UART 发送字符串
bios_uart_putnum	int	void	UART 发送十进制
bios_uart_puthex	int	void	UART 发送十六进制
bios_uart_getc	-	char	UART 接收字符
bios_uart_available	-	int	UART 是否可读
bios_buzzer_on	-	void	开启蜂鸣器
bios_buzzer_off	-	void	关闭蜂鸣器
bios_buzzer_set	int	void	设置 PWM 频率
bios_buzzer_beep	int	void	蜂鸣指定时长
bios_sw_read	-	int	读取拨码开关
bios_sw_get	int	int	读取指定拨码
bios_sw_read_high	-	int	读取高 8 位开关
bios_sw_read_mid	-	int	读取中 8 位开关
bios_sw_read_low	-	int	读取低 8 位开关
bios_btn_read	-	int	读取按钮状态
bios_btn_get	int	int	读取指定按钮
bios_btn_wait	-	int	等待任意按钮按下

附录 D　MMIO 地址映射（完整）

表 D.1: 外设地址映射（完整）

地址	设备	读写	说明
0xFFFF_FC00	数码管	W	32 位 BCD
0xFFFF_FC10	键盘数据	R/W	键值/清除
0xFFFF_FC12	键盘状态	R	pressed 标志
0xFFFF_FC20	定时器 0	R/W	计数器
0xFFFF_FC24	定时器 N	R/W	重载值
0xFFFF_FC30	PWM 最大值	W	频率控制
0xFFFF_FC34	PWM 比较值	W	占空比控制
0xFFFF_FC38	PWM 控制	W	使能
0xFFFF_FC40	UART 数据	R/W	接收/发送
0xFFFF_FC44	UART 状态	R	TX 忙/RX 就绪
0xFFFF_FC48	UART 控制	W	清除 RX 就绪
0xFFFF_FC50	看门狗	R/W	喂狗/计数
0xFFFF_FC60	LED	W	24 位输出
0xFFFF_FC70	开关	R	24 位输入
0xFFFF_FC78	按钮	R	5 位输入

附录 E　核心模块原理摘要

E.1　miniRV_SoC 顶层

miniRV_SoC 负责将 CPU、存储器与外设集成为完整 SoC，并完成时钟与复位的组织、总线互连与外设接口导出。该模块的设计重点是“解耦”：CPU 只与 Bridge 交互，Bridge 统一进行地址译码与外设路由，从而保持 CPU 核心的简洁性与可移植性。

在顶层设计中，时钟与复位的分发是稳定运行的关键。通过统一的复位域与时钟域，保证各模块同步启动，避免亚稳态与不一致状态。顶层还提供调试端口（可选），用于观察写回阶段的指令与寄存器写入，便于课程调试与波形分析。

表 E.1: miniRV_SoC 顶层端口分组（概览）

端口组	代表信号	说明
时钟/复位	`fpga_clk`, `fpga_rst`	系统时钟与同步复位
显示/LED	`dig_en`, `DN_*`, `led`	数码管与 LED 输出
输入设备	`sw`, `button`	拨码开关与按钮输入
键盘接口	`row`, `line`	4×4 矩阵键盘行列线
串口与蜂鸣器	`uart_rx`, `uart_tx`, `buzzer`	UART 与 PWM 输出

图 E.1: miniRV_SoC 连接关系示意图

E.2　myCPU 核心

myCPU 是五级流水线 RV32I 子集处理器，实现取指、译码、执行、访存、写回的完整通路。其接口分为三类：指令获取（IROM/PRAM）、数据访存（Bridge）、以及可选的调试写回接口。通过这种划分，核心可以在不感知具体外设的情况下完成指令执行。

值得注意的是，指令获取支持 IROM 与 PRAM 两种来源，配合 Bootloader 实现“就地执行”。这种设计让程序更新更灵活，同时保持硬件结构不变。数据访问则完全通过 Bus 接口完成，CPU 不需要关心外设细节。

表 E.2: myCPU 接口分组（概览）

接口类型	代表信号	说明
取指接口	`inst_addr`, `inst_from_irom`	从 IROM 取指
PRAM 取指	`inst_addr_dram`, `inst_from_dram`	Bootloader 程序区取指
访存总线	`Bus_addr`, `Bus_rdata`, `Bus_we`, `Bus_wdata`	通过 Bridge 访问内存/外设
调试接口	`debug_wb_*`	写回阶段调试信号

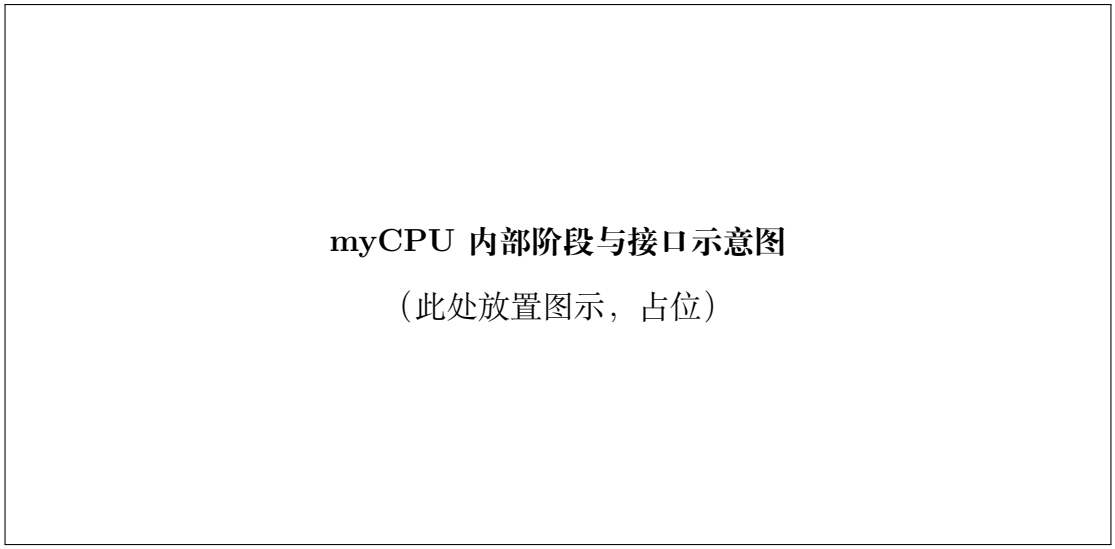

图 E.2: myCPU 内部阶段与接口示意图

E.3　controller 与 ALU

控制器根据 opcode/funct 字段生成控制信号，决定下一 PC 选择、ALU 运算类型、立即数扩展方式与写回来源。ALU 负责算术逻辑运算并产生分支比较标志。二者的协同保证了“指令语义 → 硬件行为”的一致性。

控制器与 ALU 的解耦，使得“指令译码”和“运算执行”各自独立：控制器给出操作类型与路径选择，ALU 只负责计算。这种结构既便于扩展指令，也便于在教学中解释控制信号的作用。

E.4　冒险检测与流水线寄存器

数据冒险通过前递与暂停解决，控制冒险通过分支确定后冲刷流水线寄存器解决。流水线寄存器（IF/ID、ID/EX、EX/MEM、MEM/WB）是实现分阶段隔离的关键，使不同指令在不同阶段并行执行。

冒险检测逻辑通常需要检查“目的寄存器与源寄存器”的关系，判断是否需要前递或停顿。流水线寄存器不仅保存数据，也保存控制信号，保证各阶段的操作语义不被破坏。

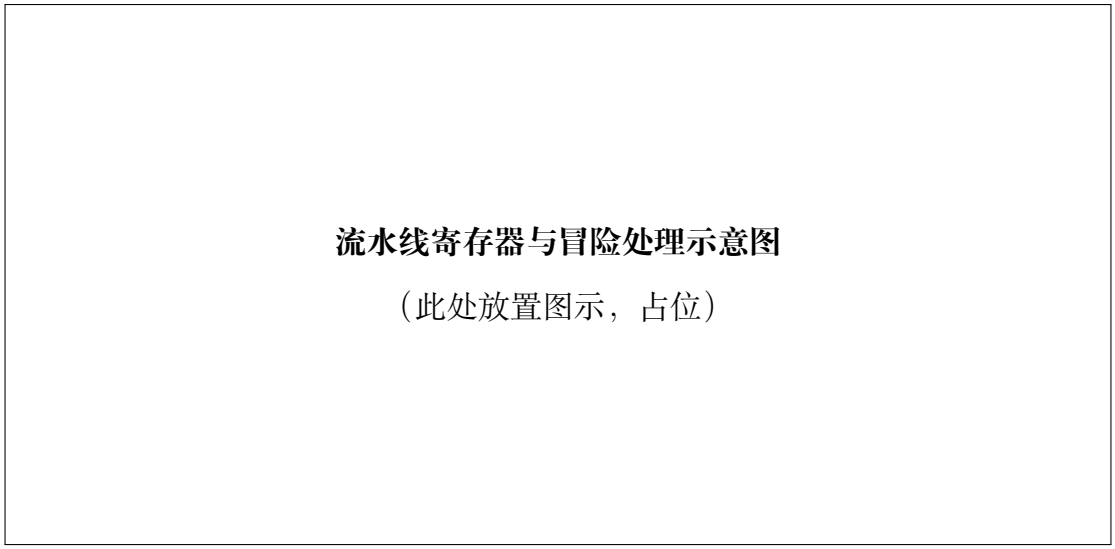

图 E.3: 流水线寄存器与冒险处理示意图

E.5　Bridge 与片上互连

Bridge 负责地址译码，将 CPU 访问请求路由到 DRAM 或外设，同时为不同外设提供统一的读写接口。该设计将外设的地址空间解耦于 CPU，使新增外设只需扩展译码逻辑与接口连线。

此外，Bridge 会对“未命中地址”给出默认返回值，避免总线悬空。这一机制对于调试尤为重要，可快速定位非法访问或地址映射错误。

图 E.4: Bridge 地址译码示意图

附录 F　外设模块原理摘要

F.1　数码管控制

数码管采用动态扫描显示，将 8 位显示分时复用到多个段码与位选线上。驱动逻辑需满足稳定的刷新频率与段码映射规则，避免闪烁与鬼影。

在工程实现中，数码管控制通常包含一个位选计数器与段码译码器。通过轮询激活位选并输出相应段码，可以实现多位数字显示。BIOS 将应用的数值转换为 BCD，从而与显示逻辑对接。

F.2　4×4 矩阵键盘

矩阵键盘通过行列轮询检测按键。控制器依次拉低行线并读取列线状态，以确定按键位置，并配合去抖与锁存逻辑提升稳定性。

为避免按键抖动导致多次触发，控制器通常采用多周期确认与稳定计数策略。按键值锁存后由软件读取，读取完成后可通过控制寄存器清除标志。

F.3　UART

UART 模块负责串口收发。其核心是波特率分频、起始/停止位检测与收发缓冲。软件侧通过状态寄存器判断发送忙或接收就绪。

UART 的设计兼顾了“易用性”和“可视化调试”：用户可以在上位机终端直接观察输出，或通过 Bootloader 传输程序。对于教学演示，串口是最直观的调试手段之一。

F.4　Timer / PWM / WDT

Timer 提供周期计数与重载功能，可用于延时与节拍；PWM 通过比较器输出占空比信号驱动蜂鸣器；WDT 通过计数与喂狗机制保证系统在异常时复位。

Timer 与 PWM 的组合可以实现“节拍 + 输出”类实验，例如蜂鸣器发声或 LED 闪烁。WDT 则更偏向系统可靠性演示，体现嵌入式系统的基本安全机制。

F.5　LED / Switch / Button

LED 为写寄存器驱动，开关与按钮为读寄存器输入。该类外设强调简洁可用，方便进行基础 I/O 实验验证。

通过这些最基础的外设，学生可以建立对“硬件寄存器—软件访问”的直观认识，为理解更复杂外设打下基础。

图 F.1: 外设控制逻辑示意图

附录 G　编译器模块原理摘要

G.1　前端：词法与语法

词法分析将字符流转为 Token 流，语法分析依据语法规则构建 AST。该过程的目标是将线性文本转换为结构化语法树，为后续语义检查与代码生成提供基础。

在实现上，词法阶段处理关键字、标识符、数字与运算符；语法阶段根据优先级和结合性解析表达式，并构建语句块与函数结构。这样得到的 AST 更接近语言语义，而非单纯的文本结构。

G.2　语义分析与符号表

语义分析阶段维护符号表与作用域信息，确保变量/函数的声明与使用一致，同时完成类型检查与简单错误诊断。

符号表不仅记录变量位置，也记录类型与栈偏移，便于后续代码生成阶段直接查询。对于教学而言，这一过程可以帮助学生理解“变量名到内存位置”的映射关系。

G.3　代码生成与栈帧管理

代码生成将 AST 映射为 RV32I 指令序列，并依据简化调用约定分配参数寄存器与栈空间。当前实现采用固定大小栈帧并保存返回地址，便于教学与调试。

固定栈帧的优势是实现简单、行为可预测，便于在波形或仿真中观察栈指针变化。虽然牺牲了部分空间效率，但更适合教学目标。

G.4　汇编与链接

汇编器将汇编文本解析为目标格式，链接器负责段布局与重定位，最终输出 COE 文件用于 FPGA 初始化。

由于目标平台是 FPGA，本项目将“链接后的机器码”直接转换为 COE 文件，这使得软件代码可以像硬件配置一样被加载，体现“软硬一体化”的特征。

图 G.1: 编译器各阶段输入输出示意图

附录 H　示例程序说明

H.1　计算器

计算器示例通过键盘输入数字与运算符，调用 BIOS 进行数码管显示与乘法运算。其核心流程为：输入解析 → 运算选择 → 结果计算 → 显示输出。

在教学实践中，该示例可用于讲解“键盘编码—软件解析—数值运算—显示编码”的完整链路，尤其适合演示系统各模块之间的协作。

H.2　LED 波浪灯

LED 波浪灯通过移位与延时产生动态光效，验证基本输出与延时机制，适用于观察系统运行是否稳定。

该示例强调时序与视觉效果的对应关系，便于理解“软件循环 + 硬件输出”的实时性。

H.3　拨码开关控制

通过读取拨码开关状态并映射到 LED，实现“输入—输出”闭环验证，可用于测试 MMIO 读写正确性。

该实验同时验证了“读外设—处理—写外设”的最小闭环，适合快速排查总线与地址映射问题。

附录 I　汇编与 COE 输出结构说明

I.1　汇编输出组织

汇编输出通常包含 .text 段指令与若干初始化伪指令。编译器负责生成函数入口、栈帧建立与返回指令，保证程序可执行。

在教学中，可以通过阅读汇编输出理解调用约定与栈帧布局，例如栈指针调整、返回地址保存与恢复等。这些内容直接体现了“高级语言语义如何落实到指令层”。

I.2　COE 格式要点

COE 文件是 Vivado 内存初始化格式，以十六进制向量表示机器码。其核心要素是 radix 与 vector 两个字段。

COE 的优势在于“工具链友好”：Vivado 可直接读取并初始化存储器，从而在硬件层面完成软件加载。对于教学实验，COE 相当于“硬件可识别的软件镜像”。

```
memory_initialization_radix=16;
memory_initialization_vector=
00008137,
00010113,
FFFFF2B7,
...
```

Listing I.1: COE 文件头示意