重庆大学课程设计报告

课程设计题目: MIPS SOC 设计与性能优化

学　　院: 计算机学院

专 业 班 级: 计算机科学与技术 01 班、03班、05班

年　　级: 2022

学　　生: 张思帆、温家振、王昱硕

学　　号: 20221799 20221006 20220980

完 成 时 间: 2025 年 1 月 9日

成　　绩:

指 导 教 师: 吴长泽

重庆大学教务处制

项目	分值	优秀 100 > x ≥ 90	良好 90 > x ≥ 70	中等 80 > x ≥ 70	及格 70 > x ≥ 60	不及格 x < 60	评分
		参考标准					
学习态度	15	学习态度认真，科学作风严谨，严格保证设计时间并按任务书中规定的进度开展各项工作	学习态度比较认真，科学作风良好，能按期圆满完成任务书规定的任务	学习态度尚好，遵守组织纪律，基本保证设计时间，按期完成各项工作	学习态度尚可，能遵守组织纪律，能按期完成任务	学习马虎，纪律涣散，工作作风不严谨，不能保证设计时间和进度	
技术水平与实际能力	25	设计合理、理论分析与计算正确，实验数据准确，有很强的实际动手能力、经济分析能力和计算机应用能力，文献查阅能力强、引用合理、调查调研非常合理、可信	设计合理、理论分析与计算正确，实验数据比较准确，有较强的实际动手能力、经济分析能力和计算机应用能力，文献引用、调查调研比较合理、可信	设计合理，理论分析与计算基本正确，实验数据比较准确，有一定的实际动手能力，主要文献引用、调查调研比较可信	设计基本合理，理论分析与计算无大错，实验数据无大错	设计不合理，理论分析与计算有原则错误，实验数据不可靠，实际动手能力差，文献引用、调查调研有较大的问题	
创新	10	有重大改进或独特见解，有一定实用价值	有较大改进或新颖的见解，实用性尚可	有一定改进或新的见解	有一定见解	观念陈旧	
论文(计算书、图纸)撰写质量	50	结构严谨，逻辑性强，层次清晰，语言准确，文字流畅，完全符合规范化要求，书写工整或用计算机打印成文；图纸非常工整、清晰	结构合理，符合逻辑，文章层次分明，语言准确，文字流畅，符合规范化要求，书写工整或用计算机打印成文；图纸工整、清晰	结构合理，层次较为分明，文理通顺，基本达到规范化要求，书写比较工整；图纸比较工整、清晰	结构基本合理，逻辑基本清楚，文字尚通顺，勉强达到规范化要求；图纸比较工整	内容空泛，结构混乱，文字表达不清，错别字较多，达不到规范化要求；图纸不工整或不清晰	

指导教师评定成绩：

指导教师签名：

SOC CPU 设计报告

组员 1 张思帆 组员 2 温家振 组员 3 王昱硕

一、设计简介

我们所提交的设计是 MIPS32 CPU 设计,实现了 52 条基础指令和 5 条特殊指令的扩展，成功构造了基于类 SRAM 接口的 Lite 版 SOC 并通过功能、性能、上板测试。又以此为基础实现了 AXI 版 SOC，并通过完整的功能测试和性能测试验证了其功能。在引入写透 Cache 后，我们进一步改进了 Cache 机制，实现了写回 Cache，进一步提高了性能得分。

（一）小组分工说明

温家振：负责逻辑运算指令、移位运算指令、算数指令的实现；负责数据通路设计。

王昱硕：负责转移指令、访存指令、内陷指令、特权指令的实现；负责控制信号的调整。

张思帆：负责 SRAM 接口 SOC 连接、AXI 集成、Cache 写回与四路组相联的实现；负责时钟频率调整与上板测试。

项目实施计划：

- 整理清楚需要添加哪些指令
- 明确添加指令需要的整个流程
- 进行数据通路设计
- 调整控制信号
- 完成逻辑运算指令的添加
- 完成移位运算指令的添加
- 完成算数运算指令的添加
- 完成转移指令的添加
- 完成访存指令的添加
- 完成内陷、特权指令的添加
- 指令添加完毕再次检查数据通路设计
- 连接 SRAM-SOC
- 仿真+上板测试是否通过功能测试
- 进行性能测试
- AXI 集成整个系统
- 进行 AXI 功能测试

- 引入基础写穿透 Cache
- 升级 Cache 为写回
- 再次进行性能测试
- 完成全部要求

二、 设计方案（30%）

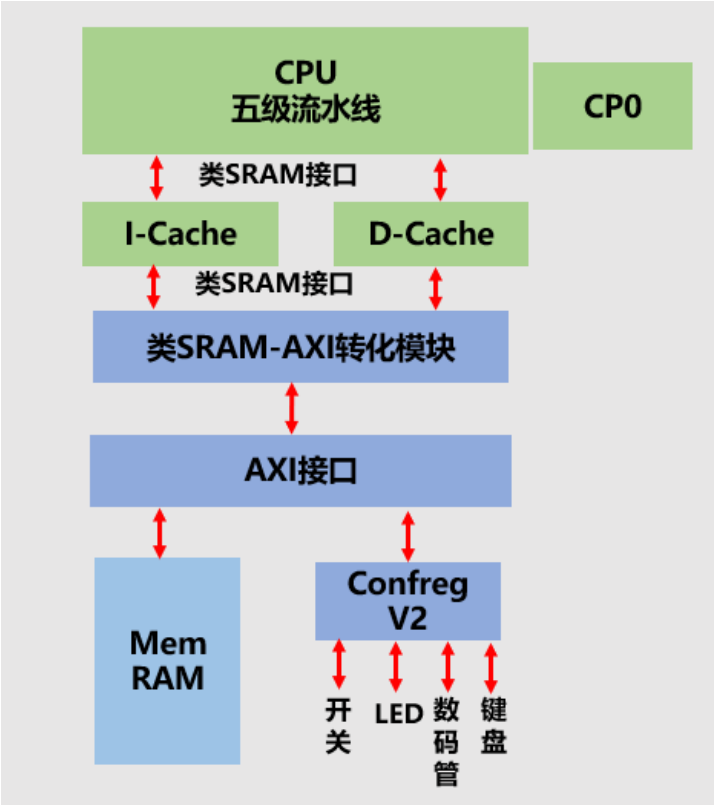


图 1：CPU 总体设计结构示意图

（一）总体设计思路

1. 处理器设计显著和趋势分析

在当前的计算环境中，处理器（CPU）的设计正朝着多核化、异构化、低功耗以及高度集成的方向发展。随着摩尔定律逐渐逼近物理极限，单纯依靠提升单个核心的频率来提高性能的方法变得不可持续[1]。因此，现代处理器设计更注重通过增加核心数量、优化架构、引入新的指令集扩展（如 AVX-512， ARM NEON 等）、以及采用更先进的制造工艺（如 7nm，5nm， 3nm 及以下）来实现性能增长。

多核与异构计算：多核处理器已成为主流，允许同时执行多个线程或进程，提高了并行处理能力。而异构计算则是指在一个系统中结合不同类型的处理单元，例如将传统的 CPU

与图形处理单元（GPU）、现场可编程门阵列（FPGA）或者专用集成电路（ASIC）相结合，以适应特定类型的任务需求，比如机器学习、图像处理和大数据分析[2]。

安全特性：鉴于近年来频发的安全事件，如 Meltdown 和 Spectre 漏洞，处理器的安全性也成为了设计中的关键考量。为此，硬件层面的安全机制如可信执行环境（TEE）、加密加速器和支持安全启动等功能被广泛集成到现代 CPU 中[3]。

新兴技术：量子计算、神经形态计算和其他前沿领域也在影响着未来处理器的发展方向。虽然这些技术尚未成熟，但它们为解决传统计算机难以处理的问题提供了新的思路，并可能在未来改变整个计算行业的格局。

2. 处理设计时需要考虑的因素分析

行业规范：遵循国际标准和协议是确保兼容性和互操作性的基础。例如，在网络通信中必须遵守 IEEE 802.11 系列标准；而在服务器市场，则需满足 SPEC CPU 基准测试的要求。此外，对于某些特殊应用场景，如汽车电子、航空航天等，还需符合 ISO 26262、DO-254 等行业特定的标准。

安全性：随着网络安全威胁日益严峻，处理器内部的安全防护措施变得至关重要。这不仅限于软件层面的安全策略，还包括硬件级别的保护机制，如内置防火墙、入侵检测系统（IDS）、防篡改功能等。同时，为了防止侧信道攻击，还应采取诸如随机化内存布局、时间延迟补偿等手段。

功耗要求：无论是桌面级还是移动端产品，节能都是一个永恒的话题。特别是在电池供电的设备上，延长续航时间是用户体验的关键因素之一。为此，设计师们通常会采用多种方法来优化功耗，比如使用高效的微架构、实施智能电源管理策略、选择合适的制程节点等。

散热管理：高密度集成带来了更高的热量输出，如果不能有效散热，将导致温度过高从而影响系统的稳定性和寿命。因此，在设计初期就需要规划好散热方案，包括但不限于优化芯片布局、选用高效的冷却组件（如液冷、风冷）、以及开发热感知算法来动态调整工作负载。

成本控制：最后，经济可行性也是不容忽视的一点。从研发阶段开始就要考虑到量产的成本效益，尽量减少不必要的复杂性，简化生产工艺，同时也要保证产品的质量和可靠性。

3.主要设计内容和模块功能划分

总体设计结构如图所示。最上层为 MIPS 模块，该模块内部包含数据通路（datapath）、

控制器（controller）以及冲突检测模块（hazard）。datapath 输出 sram 信号，此信号经由 sram_to_sram_like 模块连接 MIPS 与 soc_sram。内存管理单元（mmu）负责将虚拟地址转换为物理地址，并判断数据是否经过数据缓存（data cache），然后通过 bridge_1x2 和 bridge_2x1 模块对不经过 data cache 的数据进行分路传输。

中层为缓存(cache)模块,其包含写直达指令缓存(i_cache)和写回数据缓存(d_cache)。cache 向更底层的 cpu_axi_interface 发送与存储器交互的相关信号，所请求的数据由 cpu_axi_interface 传回。其中，指令数据的传输采用 burst 传输方式，每次传输八个字节（一字大小）的数据；数据数据每次传输一个字节（一字大小）的数据。在更底层，cpu_axi_interface 作为主设备（master）端与作为从设备（slaved）端的指令存储器（inst_ram)、数据存储器（data_ram）等外设进行交互，从而构成一个完整的中央处理器（cpu)。

（二）Sram 版整体模块设计

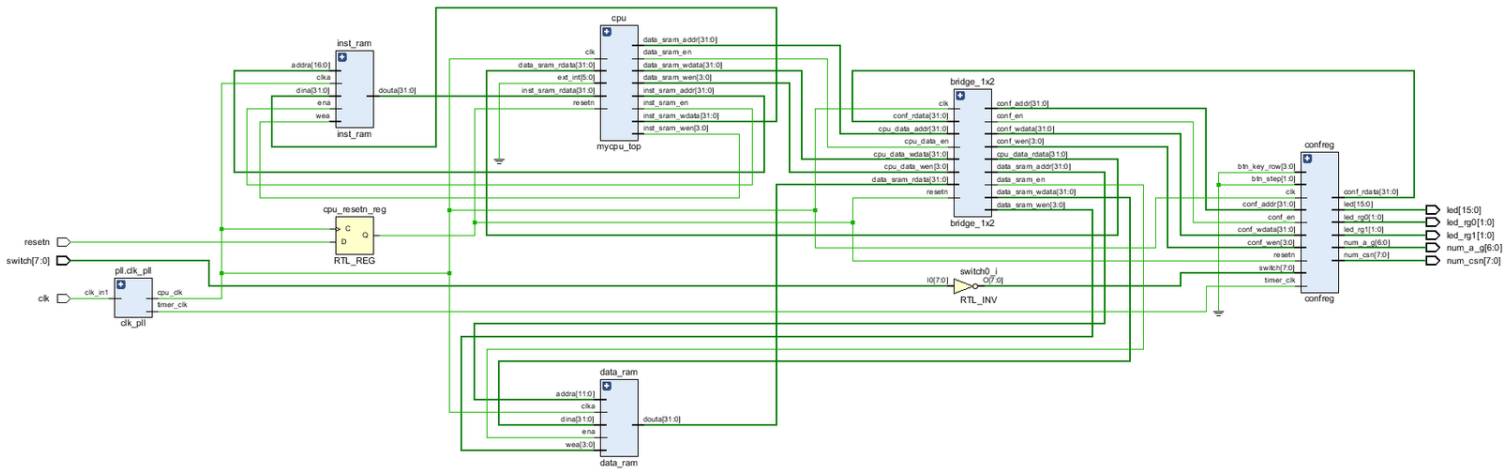


图 2：Sram 版 CPU 整体 RTL 图

首先我们先展示 sram 版本的整体模块框架。主要由时钟信号、i_ram、d_ram 和 CPU 等组成。其中的许多模块在文档中已经给出，我们着重聚焦于 mycqu_top 模块上。

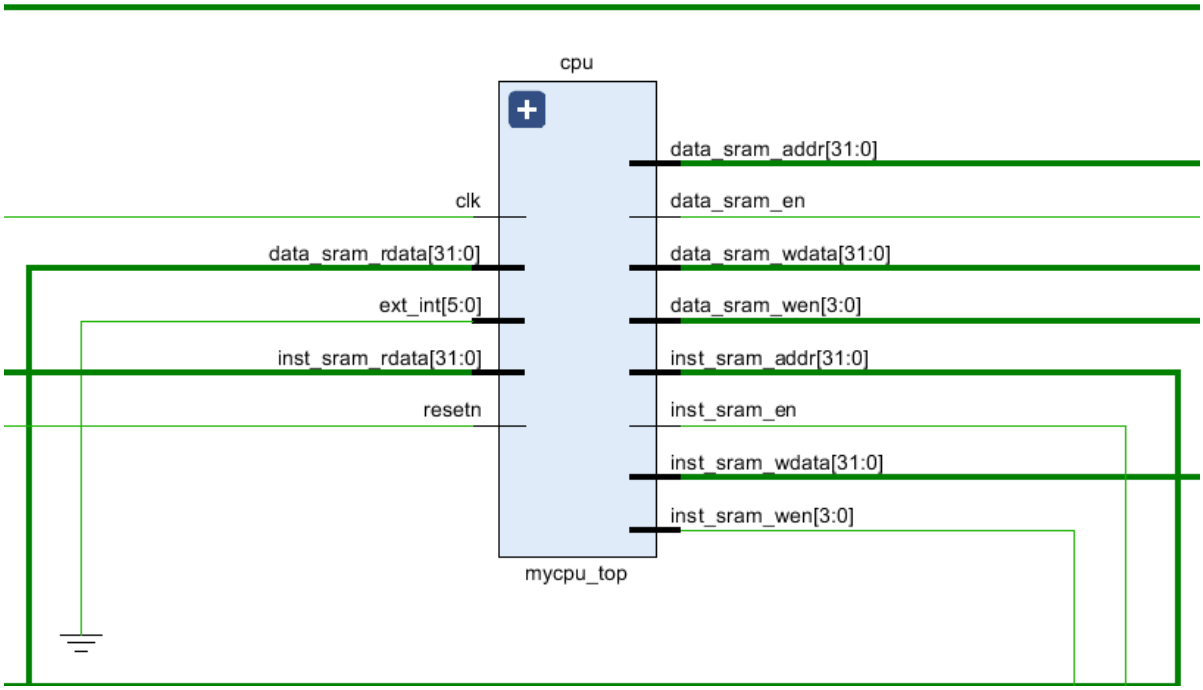


图 3：顶层 top 模块

上图是核心模块，我们所写的代码都在这个模块中进行实例化。左侧是 input，右侧是 output。下面给出接口定义，接下来分析其内部模块。

```
module mycpu_top(
    input clk,  // 时钟信号
    input resetn,  // 复位信号，低有效
    input wire [5:0] ext_int,  // 外部中断信号
    // CPU 与 SRAM 之间的信号
    output        inst_sram_en,  // 指令 SRAM 使能信号
    output [3:0] inst_sram_wen,  // 指令 SRAM 写使能信号
    output [31:0] inst_sram_addr, // 指令 SRAM 地址
    output [31:0] inst_sram_wdata, // 指令 SRAM 写数据
    input  [31:0] inst_sram_rdata, // 指令 SRAM 读数据
    // CPU 与 SRAM 之间的信号
    output        data_sram_en,  // 数据 SRAM 使能信号
    output [3:0] data_sram_wen,  // 数据 SRAM 写使能信号
```

```
    output [31:0] data_sram_addr, // 数据 SRAM 地址
    output [31:0] data_sram_wdata, // 数据 SRAM 写数据
    input  [31:0] data_sram_rdata, // 数据 SRAM 读数据
    // 调试信号
    output [31:0] debug_wb_pc,      // 调试用：写回阶段的程序计数器
    output [3:0] debug_wb_rf_wen,   // 调试用：写回阶段的寄存器写使能信号
    output [4:0] debug_wb_rf_wnum,   // 调试用：写回阶段的寄存器写地址
    output [31:0] debug_wb_rf_wdata // 调试用：写回阶段的寄存器写数据
);
```

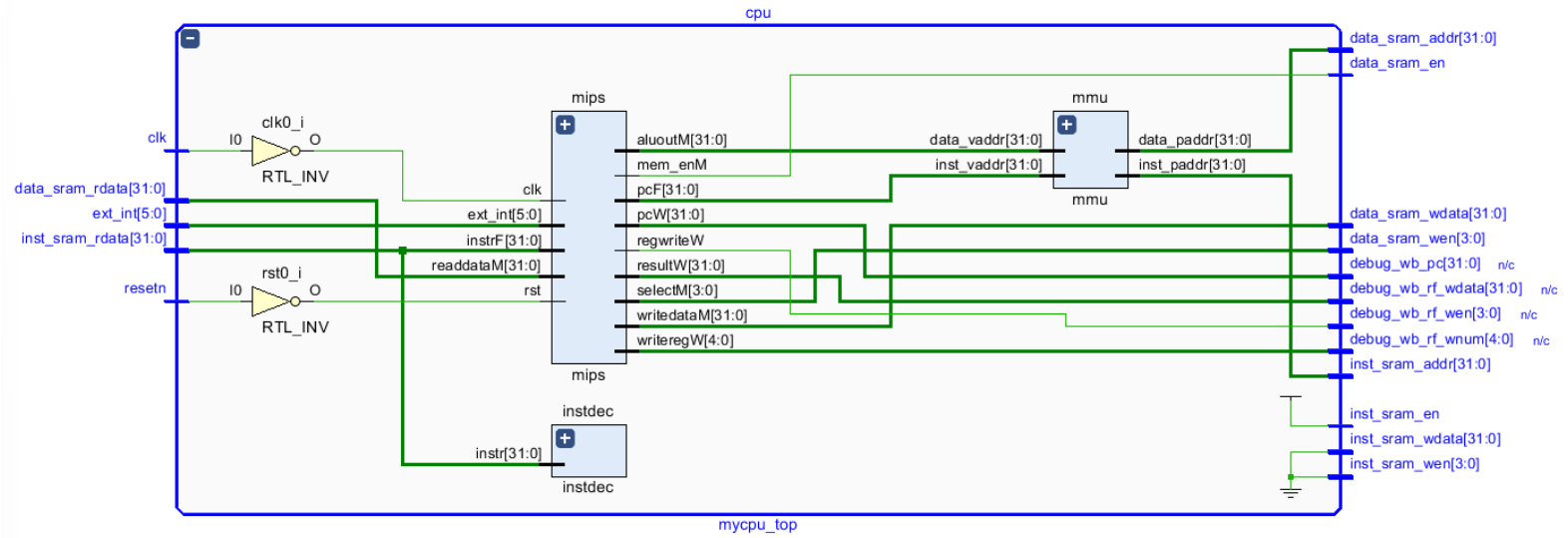
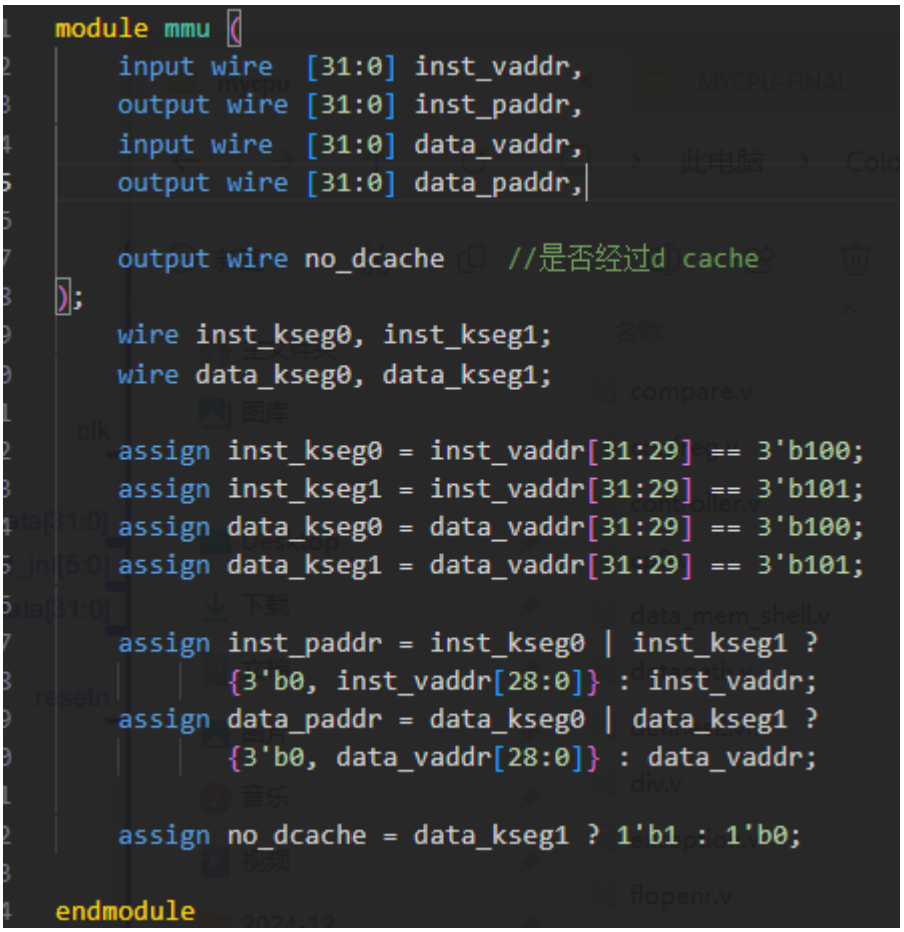


图 4：top 模块内部实现

在 CPU top 模块内部，有 mmu instdec 和 mips 三个部分。其中 insted 模块是参考文件中提供的，主要用于在仿真的时候直接可以以字符串的形式看到输入的是哪条指令。

MMU 模块，也就是 memory manager unit 模块，这个模块主要负责是把虚拟地址转换成物理地址，也是参考文档中提供的。其实现原理较为简单，如下：



```
module mmu (
    input wire  [31:0] inst_vaddr,
    output wire [31:0] inst_paddr,
    input wire  [31:0] data_vaddr,
    output wire [31:0] data_paddr,

    output wire no_dcache    //是否经过d cache
);
    wire inst_kseg0, inst_kseg1;
    wire data_kseg0, data_kseg1;

    assign inst_kseg0 = inst_vaddr[31:29] == 3'b100;
    assign inst_kseg1 = inst_vaddr[31:29] == 3'b101;
    assign data_kseg0 = data_vaddr[31:29] == 3'b100;
    assign data_kseg1 = data_vaddr[31:29] == 3'b101;

    assign inst_paddr = inst_kseg0 | inst_kseg1 ?
           {3'b0, inst_vaddr[28:0]} : inst_vaddr;
    assign data_paddr = data_kseg0 | data_kseg1 ?
           {3'b0, data_vaddr[28:0]} : data_vaddr;

    assign no_dcache = data_kseg1 ? 1'b1 : 1'b0;

endmodule
```

图 5：mmu 模块实现代码

大部分情况下虚拟地址都直接等同于物理地址，只有少部分情况例外。可以看到代码中会对输入地址的高三位进行判断，这里四个信号的判断是基于 MIPS 架构中的虚拟地址空间划分。kseg0 表示可访问的内存区域，通常用于内核空间。kseg1 的高三位是 101，通常是用于快速访问内存的。

下面给出两个模块的接口定义。

```
module mmu (
    input wire  [31:0] inst_vaddr,  // 指令虚拟地址输入
    output wire [31:0] inst_paddr,  // 指令物理地址输出
    input wire  [31:0] data_vaddr,  // 数据虚拟地址输入
    output wire [31:0] data_paddr,  // 数据物理地址输出
```

```
    output wire no_dcache     // 是否经过数据缓存（d cache）
);
```

下面我们继续深入 mips 模块，这里是核心中的核心。

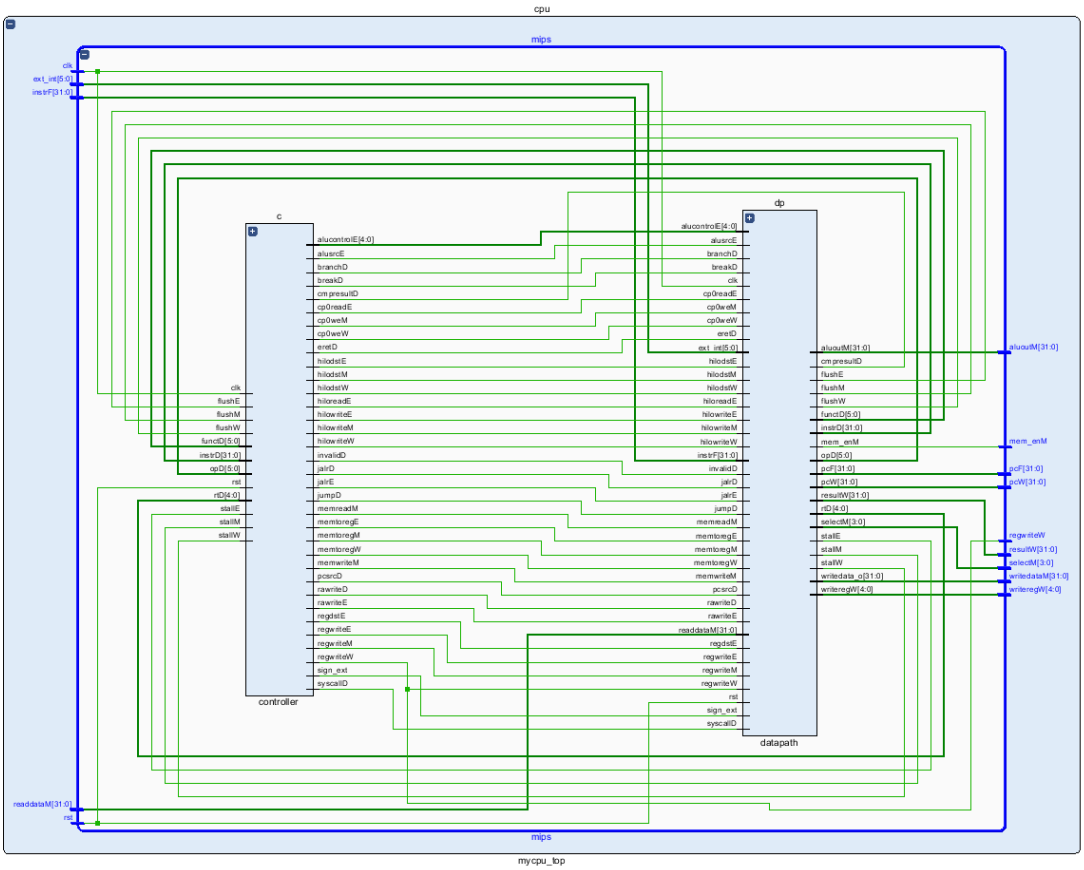


图 6：mips 模块内部细节

上图是 MIPS 模块的内部实现细节。经典的 CPU 架构如上，有 controller 和 datapath 两个部分。controller 主要负责信号控制，而 datapath 是 CPU 的核心部分，主要负责数据处理、存储和传输以及地址计算等功能。

下面我们叙述 controller 模块。先给出接口定义。

```
module controller(
    input wire clk, rst,  // 时钟信号和复位信号
```

```
    output wire sign_ext,  // 符号扩展信号

    // 解码阶段的输入信号
    input wire [31:0] instrD,  // 解码阶段的指令
    input wire [5:0] opD, functD,  // 操作码和功能码
    input wire [4:0] rtD,  // rt 寄存器地址
    output wire pcsrcD, branchD, cmpresultD, jumpD, jalrD, rawriteD,  // 控制信号
    output wire breakD, syscallD, invalidD, eretD,  // 特殊控制信号

    // 执行阶段的输入信号
    input wire flushE, stallE,  // 清空和停顿信号
    output wire memtoregE, alusrcE, jalrE, rawriteE,  // 控制信号
    output wire regdstE, regwriteE,  // 寄存器目标和写使能信号
    output wire [4:0] alucontrolE,  // ALU 控制信号
    output wire hilodstE, hilowriteE, hiloreadE,  // HI/LO 寄存器控制信号
    output wire cp0readE,  // CP0 读取信号

    // 存储阶段的输入信号
    input wire flushM, stallM,  // 清空和停顿信号
    output wire memtoregM, memwriteM, regwriteM,  // 存储阶段控制信号
    output wire hilodstM, hilowriteM,  // HI/LO 寄存器控制信号
    output wire memreadM,  // 读取存储信号
    output wire cp0weM,  // CP0 写使能信号

    // 写回阶段的输入信号
    input wire flushW, stallW,  // 清空和停顿信号
    output wire memtoregW, regwriteW,  // 写回阶段控制信号
    output wire hilodstW, hilowriteW,  // HI/LO 寄存器控制信号
    output wire cp0weW  // CP0 写使能信号
```

);

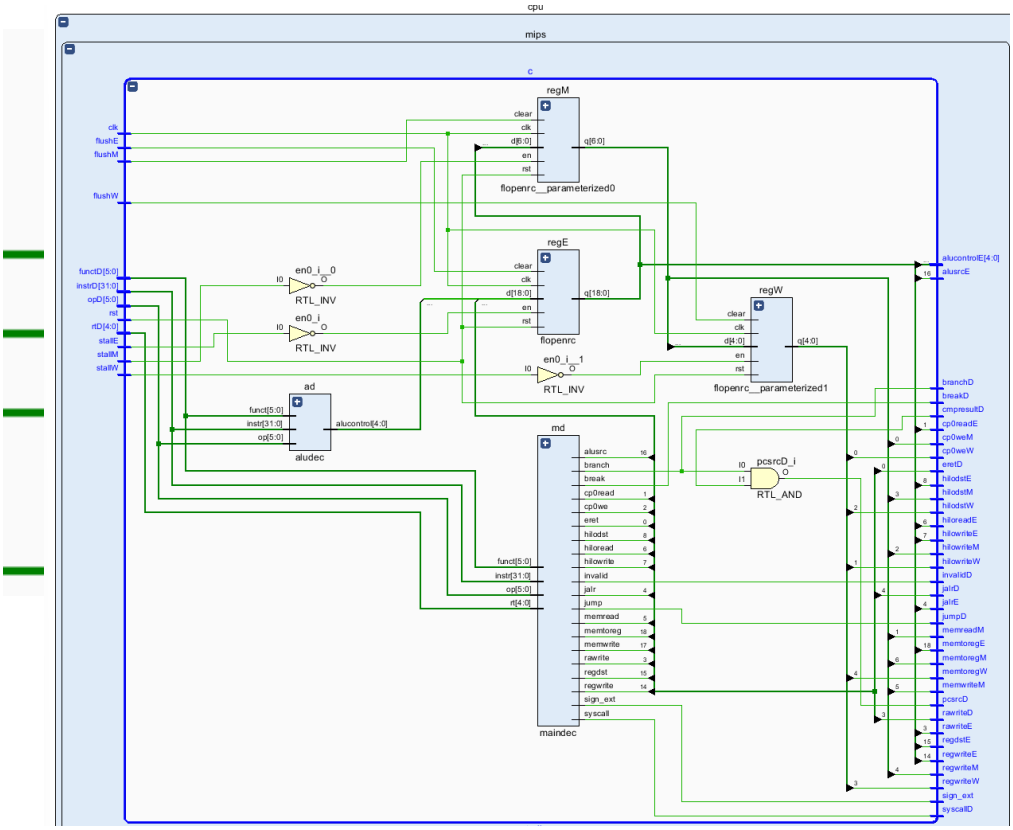
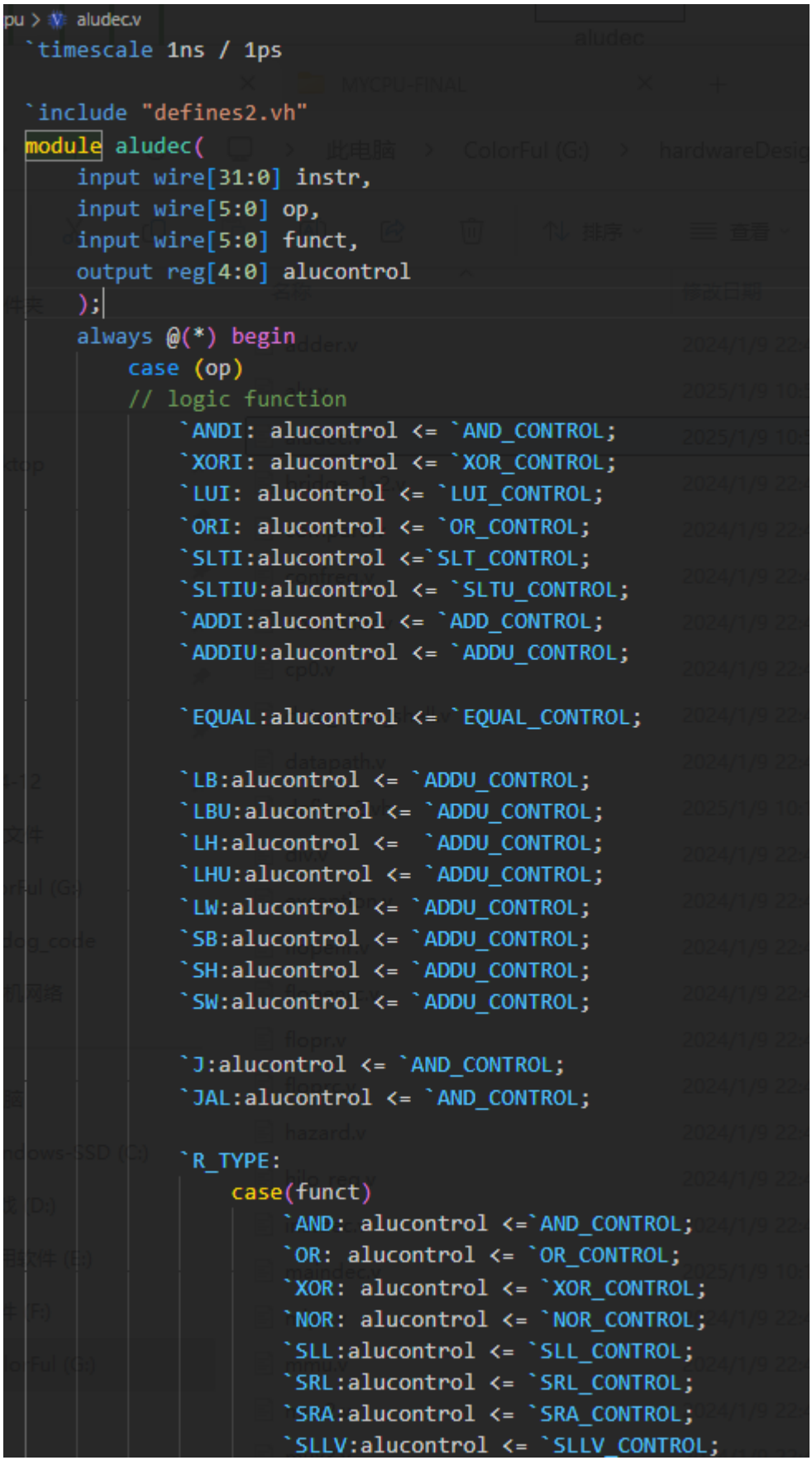


图 7：controller 模块 RTL 图

（三）aludec & maindec 模块设计

首先在 controller 模块内部起到关键作用的是 aludec。在 mips32 指令集中，指令是 32 位的，CPU 会根据传入的功能码、操作码来确认具体要执行哪条指令。输出是 alu 的控制码。下面是实现细节和接口实现。

```
module aludec(
    input wire[31:0] instr, //传入的指令
    input wire[5:0] op,//控制码
    input wire[5:0] funct,//操作码
    output reg[4:0] alucontrol//控制信号
```


pu > aludec.v
`timescale 1ns / 1ps
`include "defines2.vh"
module aludec(
input wire[31:0] instr,
input wire[5:0] op,
input wire[5:0] funct,
output reg[4:0] alucontrol
);
always @(*) begin
case (op)
// logic function
`ANDI: alucontrol <= `AND_CONTROL;
`XORI: alucontrol <= `XOR_CONTROL;
`LUI: alucontrol <= `LUI_CONTROL;
`ORI: alucontrol <= `OR_CONTROL;
`SLTI:alucontrol <=`SLT_CONTROL;
`SLTIU:alucontrol <= `SLTU_CONTROL;
`ADDI:alucontrol <= `ADD_CONTROL;
`ADDIU:alucontrol <= `ADDU_CONTROL;
`EQUAL:alucontrol <= `EQUAL_CONTROL;
`LB:alucontrol <= `ADDU_CONTROL;
`LBU:alucontrol <= `ADDU_CONTROL;
`LH:alucontrol <= `ADDU_CONTROL;
`LHU:alucontrol <= `ADDU_CONTROL;
`LW:alucontrol <= `ADDU_CONTROL;
`SB:alucontrol <= `ADDU_CONTROL;
`SH:alucontrol <= `ADDU_CONTROL;
`SW:alucontrol <= `ADDU_CONTROL;
`J:alucontrol <= `AND_CONTROL;
`JAL:alucontrol <= `AND_CONTROL;
`R_TYPE:
case(funct)
`AND: alucontrol <=`AND_CONTROL;
`OR: alucontrol <= `OR_CONTROL;
`XOR: alucontrol <= `XOR_CONTROL;
`NOR: alucontrol <= `NOR_CONTROL;
`SLL:alucontrol <= `SLL_CONTROL;
`SRL:alucontrol <= `SRL_CONTROL;
`SRA:alucontrol <= `SRA_CONTROL;
`SLLV:alucontrol <= `SLLV_CONTROL;

)

图 8：aludec 模块实现代码

由于篇幅所限，这里只给出部分的实现代码，其余大同小异。

由于除了 aludec 模块以外，maindec 模块也是必不可少的。它跟 aludec 模块配合使用，控制各种信号量的开启与关闭。其实现细节与 aludec 类似。

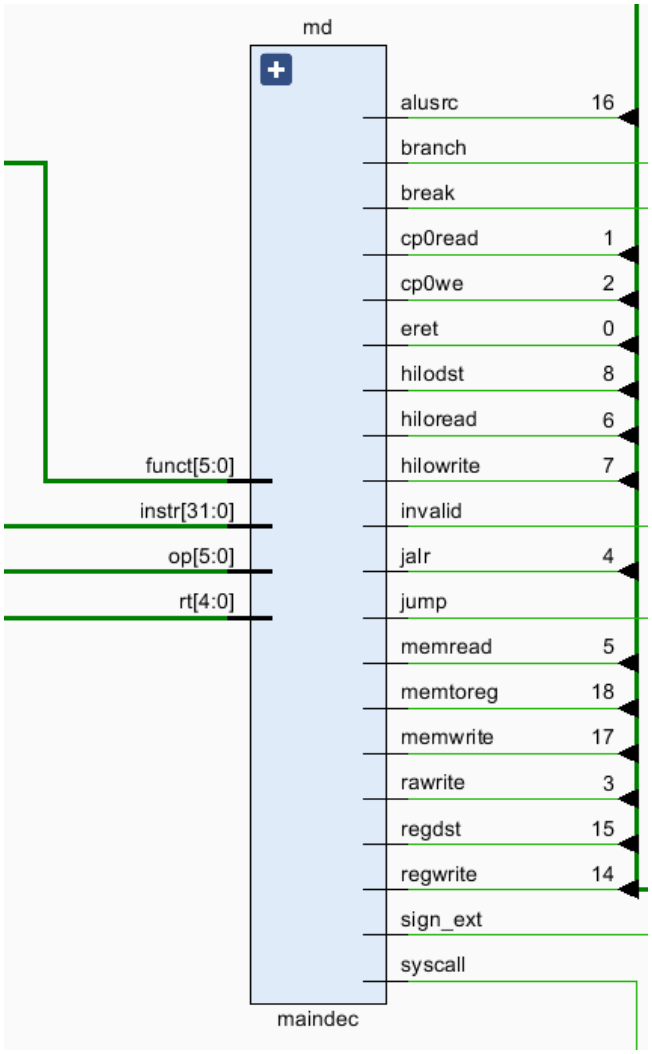


图 9：maindec 模块 RTL 图

除此另外，在 controller 之中我们还实例化了三个如下图所示的寄存器。从名字可以看出，它们分别在 CPU 的不同阶段被启用，分别对应执行、访存、写回阶段。

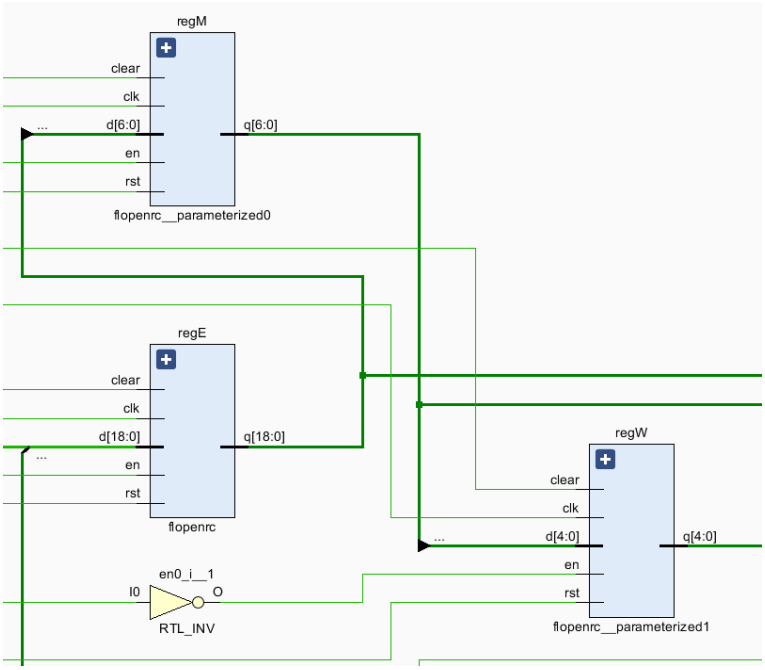


图 10：controller 模块的三个寄存器

（四）alu 模块设计

alu 模块在 datapath 中，主要负责进行各种计算。下面给出一张简略的 alu 实现图。

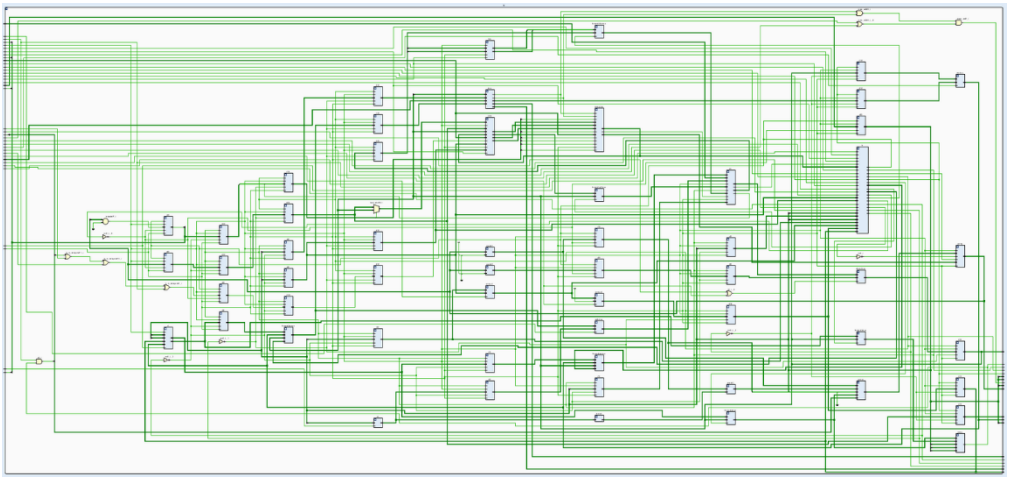


图 11：dataPath RTL 图

可以看到数据通路模块实现出的 RTL 图是十分复杂的，而 alu 模块在下图。

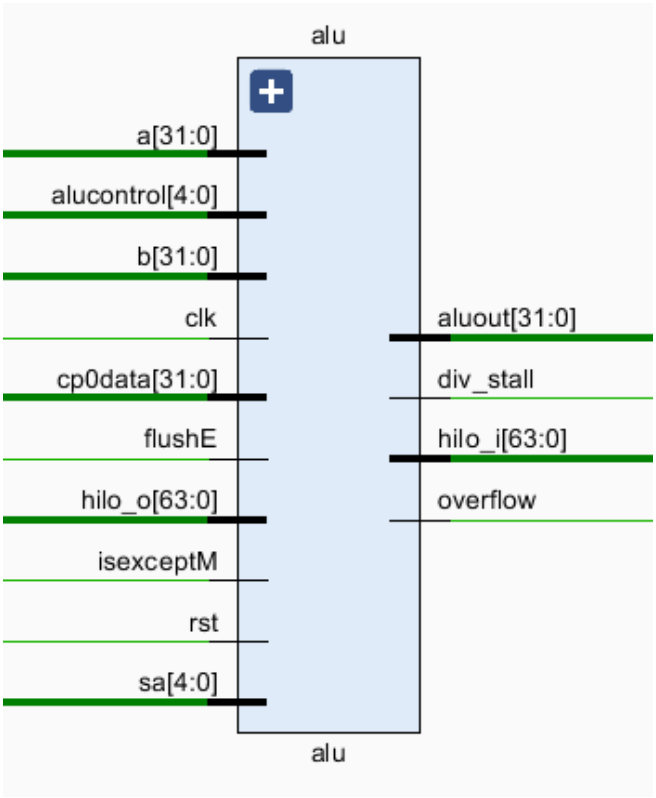


图 12：alu RTL 图

alu 模块根据之前 aludec 模块识别出的 alucontrol 码进行具体计算，大部分的结果会通过 aluout 这个 output 传出。右侧的 output 中，div_stall 主要是为了在进行除法的过程中实现流水线的暂停。因为除法的计算需要多个周期，我们未对其进行优化。因此除法在计算的时候需要通知 CPU 暂停其他操作。而 hilo 寄存器则是给四条 MF、MT 指令使用的。

alu 模块的部分实现细节和接口定义如下：

```
module alu(
    input wire clk, rst, isexceptM, flushE,  // 时钟信号、复位信号、异常信号和清空信号
    input wire [31:0] a, b,  // ALU 的两个操作数
    input wire [4:0] sa,  // 移位量
    input wire [4:0] alucontrol,  // ALU 控制信号
    output reg [31:0] aluout,  // ALU 输出结果
    input wire [63:0] hilo_o,  // HI/LO 寄存器的输出
    output reg [63:0] hilo_i,  // HI/LO 寄存器的输入
```

```
    output reg div_stall,   // 除法操作的停顿信号
    output wire overflow,   // 溢出信号
    input wire [31:0] cp0data   // CP0 数据输入
);
```

```
always @(*) begin
    double_sign = 0;
    hilo_i = 64'b0;
    if(rst | isexceptM) begin
        div_stall = 1'b0;
        div_start = 1'b0;
    end
    else begin
        case(alucontrol)
            //逻辑运算
            `AND_CONTROL   : aluout = a & b;  //指令AND、ANDI
            `OR_CONTROL    : aluout = a | b;  //指令OR、ORI
            `XOR_CONTROL   : aluout = a ^ b;  //指令XOR
            `NOR_CONTROL   : aluout = ~(a | b);  //指令NOR、XORI
            `LUI_CONTROL   : aluout = {b[15:0],16'b0}; //指令LUI
            ////////////////////////////////////////////////////////////////////////////////////////////////
            `EQUAL_CONTROL : aluout = (a == b) ? 32'b1 : 32'b0;
            //移位指令
            `SLL_CONTROL   : aluout = b << sa;  //指令SLL
            `SRL_CONTROL   : aluout = b >> sa;  //指令SRL
            `SRA_CONTROL   : aluout = $signed(b) >>> sa;  //指令SRA
            `SLLV_CONTROL  : aluout = b << a[4:0];  //指令SLLV
            `SRLV_CONTROL  : aluout = b >> a[4:0];  //指令SRLV
            `SRAV_CONTROL  : aluout = $signed(b) >>> a[4:0]; //指令SRAV
            //算数运算
            `ADD_CONTROL   : {double_sign,aluout} = {a[31],a} + {b[31],b}; //指令ADD、ADDI
            `ADDU_CONTROL  : aluout = a + b; //指令ADDU、ADDIU
            `SUB_CONTROL   : {double_sign,aluout} = {a[31],a} - {b[31],b}; //指令SUB
            `SUBU_CONTROL  : aluout = a - b; //指令SUBU
            `SLT_CONTROL   : aluout = $signed(a) < $signed(b) ? 32'b1 : 32'b0;  //指令SLT、SLTI
            `SLTU_CONTROL  : aluout = a < b ? 32'b1 : 32'b0; //指令SLTU、SLTIU
            `MULT_CONTROL  : hilo_i = $signed(a) * $signed(b); //指令MULT
            `MULTU_CONTROL : hilo_i = {32'b0, a} * {32'b0, b}; //指令MULTU
            `DIV_CONTROL   : begin //指令DIV
```

图 13：alu 模块部分

可以看见对于许多指令，其执行的操作都是直接给 aluout 进行赋值。

（五）转移指令模块设计

信号名、输入输出类型、信号描述：

```
input wire [31:0] a,b,      // 输入信号ab
input wire [5:0] op,funct,  // 操作码、功能码
input wire [4:0] rt,        // Register Target 目标寄存器索引 5位是因为刚好有32个通用寄存器
output wire y               // 输出信号y
```

图 14：转移指令模块的信号描述

数据通路图：

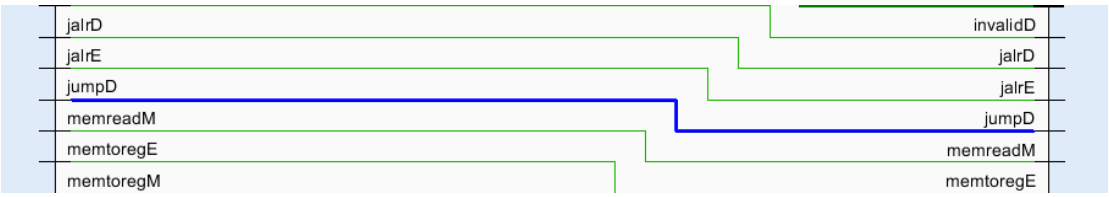


图 15：转移指令模块的数据通路

实现：

转移指令共有 12 条，如下图所示：

指令名称格式	指令功能简述
BEQ rs, rt, offset	相等转移
BNE rs, rt, offset	不等转移
BGEZ rs, offset	大于等于0转移
BGTZ rs, offset	大于0转移
BLEZ rs, offset	小于等于0转移
BLTZ rs, offset	小于0转移
BLTZAL rs, offset	小于0调用子程序并保存返回地址
BGEZAL rs, offset	大于等于0调用子程序并保存返回地址
J target	无条件直接跳转
JAL target	无条件直接跳转至子程序并保存返回地址
JR rs	无条件寄存器跳转
JALR rd, rs	无条件寄存器跳转至子程序并保存返回地址

图 16：转移指令示意图

实现起来也不复杂，只有包含返回的指令需要特别留意。

转移指令实现的要点：

有两个要素：一是决定是否跳转、二是跳转时的跳转目标。

指令分为两种：B 开头的分支指令和 J 开头的跳转指令，B 开头的分支指令要进行判断，J 开头的跳转指令一定跳转。

如何计算跳转目标呢？有三种计算方式：

1. 转移延迟槽指令 PC 加上转移指令中的偏移。

由于延迟槽的存在，当转移发生之前，延迟槽中的指令已经被执行。因此需要在延迟槽指令 PC 的基础上加上转移值，由此获取转移到的地址。

2. 转移延迟槽指令 PC 高位与转移指令中的偏移拼接。

对于 J 型指令来说，指令中的目标地址字段只有 26 位。这 26 位的偏移量将被左移两位（因为在 MIPS 中所有指令都是 4 字节对齐的），然后与当前 PC 值的最高四位拼接在一起形成新的 32 位的目标地址。之所以使用当前 PC 值的最高四位，是因为 MIPS 假设代码段不会跨越不同的内存区域，所以可以利用当前 PC 的高几位来确定新地址的基本位置。只要 PC 值的最高几位保持不变，新计算出的目标地址通常仍然位于同一代码段内。

3. 从通用寄存器获取

这是指有条件或无条件的寄存器跳转指令(如 jr 和 jalr)。例如，jr $ra 会将 PC 设置为寄存器$ra 中的值，常用于过程调用之后返回到调用点。

该模块的具体实现：

```
module compare(
    input wire [31:0] a,b,      // 输入ab
    input wire [5:0] op,funct,  // 操作码、功能码
    input wire [4:0] rt,        // Register Target 目标寄存器索引 5位是因为刚好有32个通用寄存器
    output wire y
    );

    // a[31]是最高位，检查最高位，最高位为0意味着这个数是一个非负数

    // BGTZ，a是非负数且不为0，所以a大于0
    // BLEZ，同理，a为负数且不为0，所以a小于0

    // REGIMM_INST = 6'b000001
    // rt[0]   用于区分是BGEZ还是BLTZ
    // rt[0] = 1表示是BGEZ，a[31]表示a是非负数
    // rt[0] = 0表示是BLTZ，a[31]表示a是负数

    // R型指令的操作细节由funct给出
    // JR   Jump Register          跳转后不返回
    // JALR Jump and Link Register 跳转后会保存返回地址，执行完毕后可跳转回来，多用于调用
    assign y = ((op == `BEQ) && (a == b)) ||
               ((op == `BNE) && (a != b)) ||
               ((op == `BGTZ) && (a[31] == 0) && (a != 32'b0))   ||
               ((op == `BLEZ) && ((a[31] == 1) || (a == 32'b0))) ||
               ((op == `REGIMM_INST) && rt[0] && (a[31] == 0))   ||
               ((op == `REGIMM_INST) && ~rt[0] && (a[31] == 1))   ||
               ((op == `R_TYPE) && (funct == `JR || funct == `JALR));

endmodule
```

图 17：转移指令实现代码

给要实现的代码分个类：

1. BEQ 和 BNE

这两个的实现比较简单，只需要比较 a 和 b 就可以了。

2. BGEZ、BGTZ、BLEZ、BLTZ

功能定义和 BEQ 还有 BNE 比较类似，唯一多了一个就是 a[31]的判断，a[31]是用于区分 a 是非负数和负数的关键点，当判断完 a 的正负后，判断 a 是否为 0 即可。

3. J、JAL

这两个是 J 型指令，J 用于无条件跳转到指定的目标地址，JAL 用于无条件跳转到指定的目标地址，并将返回地址保存到寄存器 $ra（即寄存器 31）。

由于它们只提供 26 位地址字段，所以实际跳转范围是有限的，并且依赖于当前 PC 值的一部分来确定完整的 32 位地址。

就是上文提到的第二种计算方式。

4. BGEZAL、BLTZAL

这两个指令就是 BGEZ+JAL，BLTZ+JAL，逻辑上等同于 BGEZ 的实现，无论分支是否跳

转，都会将该分支对应延迟槽指令之后的指令的 PC 值保存至第 31 号通用寄存器中，就是 JAL。因此是 BGEZ 和 JAL 的结合。BLTZAL 同理。

5. JR、JALR

这两个是R型指令而不是J型指令,因为它们使用寄存器而不是立即数作为目标地址。它们允许根据寄存器中的值进行跳转，提供了更大的灵活性，比如用于实现函数返回或间接跳转。

JR 的跳转目标为寄存器 rs 中的值。JALR 的跳转目标也为寄存器 rs 中的值，但是会将该分支对应延迟槽指令之后的指令的 PC 值保存至寄存器 rd 中。

至此转移指令全部实现。

（六）访存指令模块设计

信号名、输入输出类型、信号描述：

```
input wire[5:0] op,              // 输入操作码，6位
input wire[1:0] aluout,          // 输入地址输出，2位
input wire[31:0] readdata_i,     // 输入读取数据，32位
input wire[31:0] writedata_i,    // 输入写入数据，32位
output wire[31:0] readdata_o,    // 输出读取数据，32位
output wire[31:0] writedata_o,   // 输出写入数据，32位
output wire[3:0] select,         // 输出选择信号，4位
output wire adel,                // 输出地址错误信号（加载）
output wire ades                 // 输出地址错误信号（存储）
```

图 18：访存指令模块信号描述

数据通路图：

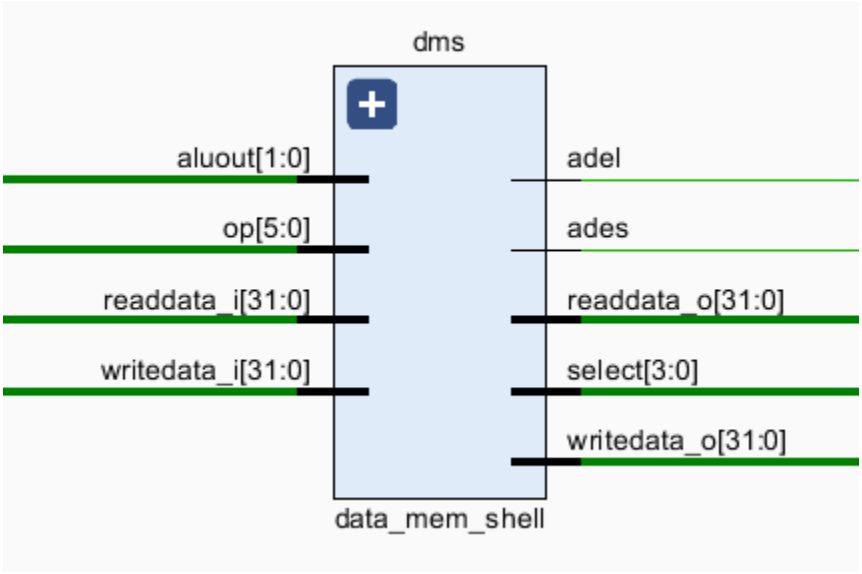


图 19：访存指令模块数据通路

实现：

访存指令共有 8 条，如下所示：

指令名称格式	指令功能简述
LB rt, offset(base)	取字节有符号扩展
BU rt, offset(base)	取字节无符号扩展
LH rt, offset (base)	取半字有符号扩展
LHU rt, offset (base)	取半字无符号扩展
LW rt, offset(base)	取字
SB rt, offset(base)	存字节
SH rt, offset(base)	存半字
SW rt, offset (base)	存字

图 20：访存指令示意图

访存指令可以大致分为两类，一类是 Load 类指令，一类是 Store 类指令。

一、Load 类指令

Load 类指令包括 LW、LB、LBU、LH、LHU。

其中 LW 算是比较基础的指令，其他四个指令的实现都与 LW 类似。

```
assign readdata_o = (
(op == `LW && aluout == 2'b00)? readdata_i: // 加载字操作，直接读取数据
(op == `LH && aluout == 2'b00)? {{16{readdata_i[15]}}, readdata_i[15:0]}: // 加载半字，符号扩展低16位
(op == `LH && aluout == 2'b10)? {{16{readdata_i[31]}}, readdata_i[31:16]}: // 加载半字，符号扩展高16位
(op == `LHU && aluout == 2'b00)? {16'b0, readdata_i[15:0]}: // 加载无符号半字，零扩展低16位
(op == `LHU && aluout == 2'b10)? {16'b0, readdata_i[31:16]}: // 加载无符号半字，零扩展高16位
(op == `LB && aluout == 2'b00)? {{24{readdata_i[7]}}, readdata_i[7:0]}: // 加载字节，符号扩展
(op == `LB && aluout == 2'b01)? {{24{readdata_i[15]}}, readdata_i[15:8]}: // 加载字节，符号扩展
(op == `LB && aluout == 2'b10)? {{24{readdata_i[23]}}, readdata_i[23:16]}: // 加载字节，符号扩展
(op == `LB && aluout == 2'b11)? {{24{readdata_i[31]}}, readdata_i[31:24]}: // 加载字节，符号扩展
(op == `LBU && aluout == 2'b00)? {24'b0, readdata_i[7:0]}: // 加载无符号字节，零扩展
(op == `LBU && aluout == 2'b01)? {24'b0, readdata_i[15:8]}: // 加载无符号字节，零扩展
(op == `LBU && aluout == 2'b10)? {24'b0, readdata_i[23:16]}: // 加载无符号字节，零扩展
(op == `LBU && aluout == 2'b11)? {24'b0, readdata_i[31:24]}: // 加载无符号字节，零扩展
32'b0); // 默认输出0
```

图 21：访存指令 Load 类指令代码实现

LW 只需要直接读取数据即可。

LB、LBU、LH、LHU 这四条指令在译码、执行、写回级的数据通路和控制逻辑可以复用 LW 指令的实现，只是控制信号要对应生成。

但是还是与 LW 有一定的差异，具体实现起来需要注意这些细节：

1. 需要从数据 RAM 输出结果中选择所需内容

因为数据 RAM 的宽度是 4 个字节，LB 和 LBU 访问的内容可以出现在这四个字节中的任一个中，LH 和 LHU 访问的内容可以是其中的低 2 字节或是高 2 字节，也就是说我们需要引入一个多路选择器进行选取。

2. 需要将选取内容扩展至 32 位

LB、LBU、LH 和 LHU 指令从数据 RAM 取回的数据宽度比通用寄存器的宽度要小，所以这些数据需要扩展到 32 位才能成为最终写入寄存器的结果。进行有符号扩展还是无符号扩展是根据指令来确定的，LB、LH 是有符号扩展，LBU、LHU 是无符号扩展。

在考虑到上述细节后再进行编写就会容易很多。

二、Store 类指令

Store 类指令包括 SW、SH、SB。和 Load 类似，SW 仍然是 Store 类指令的基础。

```
// 选择信号，根据操作码和地址确定要进行的读写字节
assign select = ((op == `LW || op == `LB || op == `LBU || op == `LH || op == `LHU)? 4'b0000: // 加载操作
          (op == `SW && aluout == 2'b00)? 4'b1111: // 存储字操作
          (op == `SH && aluout == 2'b10)? 4'b1100: // 存储半字（高半部分）
          (op == `SH && aluout == 2'b00)? 4'b0011: // 存储半字（低半部分）

          (op == `SB && aluout == 2'b11)? 4'b1000: // 存储字节（高字节）
          (op == `SB && aluout == 2'b10)? 4'b0100: // 存储字节（次高字节）
          (op == `SB && aluout == 2'b01)? 4'b0010: // 存储字节（次低字节）
          (op == `SB && aluout == 2'b00)? 4'b0001: // 存储字节（低字节）
          4'b0000); // 默认选择信号

// 写入数据，根据操作码决定写入数据的格式
assign writedata_o = ((op == `SW)? writedata_i: // 存储字操作，直接写入数据
    (op == `SH)? {writedata_i[15:0], writedata_i[15:0]}: // 存储半字，复制低16位
    (op == `SB)? {writedata_i[7:0], writedata_i[7:0], writedata_i[7:0], writedata_i[7:0]}: // 存储字节，复制低8位
    32'b0); // 默认输出0
```

图 22：访存指令 Store 类指令代码实现

但唯一不同的是这次的写入无需使用选择器，自行选择写入的位置即可。因此实现 SH 和 SB 的关键是如何在一块 4 字节宽的 RAM 上完成字节或半字的写入，具体的实现可以通过 RAM 的字节写使能来实现，以此来实现自行控制写入位置。

三、可能发生的例外

因为访存指令会涉及到内存的修改，因此很有可能会发生例外，在实现时也需要特别注意，因此特别加入了控制信号。

```
// 地址错误信号（加载），检查地址是否对齐
assign adel = ((op == `LH || op == `LHU) && aluout[0]) || (op == `LW && aluout != 2'b00);

// 地址错误信号（存储），检查地址是否对齐
assign ades = (op == `SH & aluout[0]) | (op == `SW & aluout != 2'b00);
```

图 23：访存指令例外相关代码

至此，访存指令全部实现。

（七）自陷、特权指令模块设计

信号名、输入输出类型、信号描述：

```
input wire rst,                          // 复位信号
input wire[5:0] ext_int,                 // 外部中断信号

input wire adel,                         // 地址错误异常（取地址错误）
input wire ades,                         // 地址错误异常（存地址错误）
input wire instadel,                     // 指令延迟异常
input wire syscall,                      // 系统调用异常
input wire break,                        // 中断异常
input wire eret,                         // 返回异常
input wire invalid,                      // 无效指令异常
input wire overflow,                     // 溢出异常
input wire[31:0] cp0_statusW,            // 控制寄存器0（CP0）状态
input wire[31:0] cp0_causeW,             // 控制寄存器0（CP0）原因
input wire[31:0] cp0_epcW,               // 控制寄存器0（CP0）异常程序计数器

output wire[31:0] excepttypeM,           // 异常类型输出
output wire[31:0] newpcM,                // 新的程序计数器值
output wire isexceptM                    // 是否发生异常的指示信号
```

图 24：自陷、特权指令模块信号描述

数据通路图：

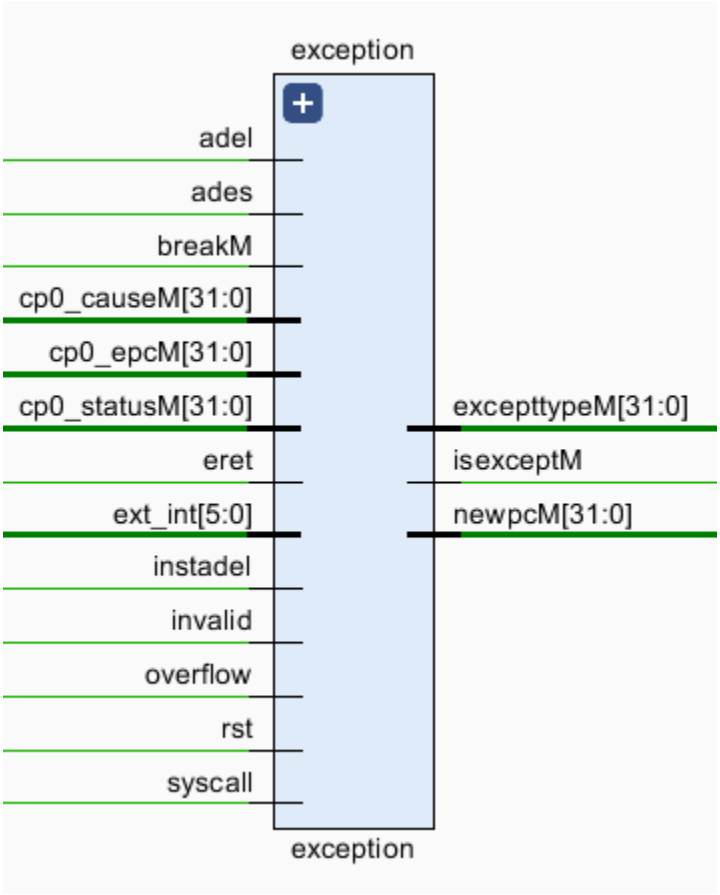


图 25：例外模块数据通路

实现：

自陷、特权指令一共有 5 条，是在 52 条指令的基础上实现的额外的 5 条。

包括：SYSCALL、BREAK、ERET、MTC0、MFC0，其中 SYSCALL 和 BREAK 是自陷指令，其余 3 条是特权指令。

这些指令的作用也不复杂，简单来说就是为了实现软硬件协同处理而引入的。

其中最为重要的就是例外（异常）的处理。

例外的处理是有优先级的，当一条指令同时满足多个例外触发条件时，处理器核将按照下图进行处理，优先触发优先级高的例外。

表 5-1 例外优先级

例外	类型
中断	异步
地址错例外—取指	同步
保留指令例外	同步
整型溢出例外、陷阱例外、系统调用例外	同步
地址错例外—数据访问	同步

图 26：例外优先级

同时也规定了一个例外的统一入口：0xBFC00380

后续的所有指令都是在这个基础上实现的。

例外的处理是有一个一般性过程的，大致可以分为以下几步：

1. 当 CP0.Status.EXL 为 0 时，更新 CP0.EPC：将例外处理返回后重新开始执行的指令 PC 填入到 CP0.EPC 寄存器中。如果发生例外的指令不在分支延迟槽中，则重新开始执行的指令 PC 就等于发生例外的指令的 PC；否则重新开始执行的指令 PC 等于发生例外的指令的 PC-4（即该延迟槽对应的分支指令的 PC）。

2. 当 EXL 为 0 时，更新 CP0.Cause 寄存器的 BD 位：如果发生例外的指令在分支延迟槽中，则将 CP0.Cause.BD 置为 1。

3. 更新 CP0.Status.EXL 位：将其置为 1。

4. 进入约定的例外入口重新取指，软件开始执行例外处理程序。

当处理完毕后，使用 ERET 指令从例外处理程序返回，同时进行以下工作：

1. 将 CP0.Status.EXL 清零

2. 从 CP0.EPC 寄存器取出重新执行的 CP 值，然后从该 CP 值处继续取指执行。

在此基础上进行代码编写即可。

```
// 计算异常类型
assign excepttypeM = (rst)? 32'b0 : // 如果复位，异常类型为0
                     ((({ext_int, cp0_causeW[9:8]} & cp0_statusW[15:8]) != 8'h00) && // 检测外部中断
                      (cp0_statusW[1] == 1'b0) && (cp0_statusW[0] == 1'b1))? 32'h00000001 : // 中断异常
                     (instadel)? 32'h00000004 : // 指令延迟异常
                     (adel)? 32'h00000005 : // 取地址错误异常
                     (ades)? 32'h00000006 : // 存地址错误异常
                     (syscall)? 32'h00000008 : // 系统调用异常
                     (break)? 32'h00000009 : // 中断异常
                     (eret)? 32'h0000000e : // 返回异常
                     (invalid)? 32'h0000000a : // 无效指令异常
                     (overflow)? 32'h0000000c : // 溢出异常
                     32'h0; // 默认值为0

// 判断是否发生异常
assign isexceptM = |excepttypeM; // 如果excepttypeM中有任意位为1，表示发生异常

// 根据异常类型确定新的程序计数器值
assign newpcM = (excepttypeM == 32'h00000001)? 32'hbfc00380 : // 中断异常
                (excepttypeM == 32'h00000004)? 32'hbfc00380 : // 指令延迟异常
                (excepttypeM == 32'h00000005)? 32'hbfc00380 : // 取地址错误异常
                (excepttypeM == 32'h00000006)? 32'hbfc00380 : // 存地址错误异常
                (excepttypeM == 32'h00000008)? 32'hbfc00380 : // 系统调用异常
                (excepttypeM == 32'h00000009)? 32'hbfc00380 : // 中断异常
                (excepttypeM == 32'h0000000a)? 32'hbfc00380 : // 无效指令异常
                (excepttypeM == 32'h0000000c)? 32'hbfc00380 : // 溢出异常
                (excepttypeM == 32'h0000000e)? cp0_epcW : // 返回异常使用异常程序计数器
                32'b0; // 默认值为0
```

图 27：异常部分实现代码

表 6-5 ExcCode 编码及其对应例外类型

ExcCode	助记符	描述
0x00	Int	中断
0x04	AdEL	地址错例外（读数据或取指令）
0x05	AdES	地址错例外（写数据）
0x08	Sys	系统调用例外。
0x09	Bp	断点例外。
0x0a	RI	保留指令例外。
0x0c	Ov	算出溢出例外。

图 28：ExcCode 编码及其对应例外类型

同时根据参考文档中的 ExcCode 编码以及对应例外类型，进行异常类型划分，划分后进行跳转，即可实现所有异常处理。

因为 CP0 相关代码在参考资料中已经给出，在此不再赘述。

至此，自陷和特权指令全部实现。

三、实验过程以及总结（40%）

（一）设计工作日志

12 月 23 日 09：00-12：00：小组成员听取老师讲述项目要点，认真记录项目流程。14：00-18：00 进行工作分配，整理早上老师讲的内容。

12 月 24 日 11：00-14：00 查看项目参考文档。15：00-20：00 查看项目的参考代码，学习代码写法。

12 月 25 日 11：00-20：00 进行 sram 版的整体框架的初步搭建。

12 月 26 日 10：30-15：00 继续深入研究参考代码。20：00-22：30 理解整体逻辑框架

12 月 27 日 10：00-16：00 小组成员共同研讨。统一思路，并重构了初版代码。

12 月 28 日 10：00-17：00 小组分工，编写 adder、aludec、maindec 模块。

12 月 29 日 10：00-18：00 编写 compare、alu、cp0 模块。

12 月 30 日 10：00-18：30 共同学习 CPU 的冲突检测和中断机制，编写 exception、hazard 模块。

12 月 31 日 11：00-18：00 编写 pc、regfile、signext、sl2 模块。

1 月 1 日-1 月 2 日 全天 元旦假期，主要休息，阅读《CPU 设计实战》。

1 月 3 日 09：00-17：30 编写 flopr、flopenr、floprc、controller、datapath 模块。

1 月 4 日 10：30-18：30 重构 alu 模块的代码，添加了内陷指令、特权指令、访存指令、转移指令。

1 月 5 日 11：30-18：00 尝试进行 sram 上版，发现错误并调试，但无法找到问题。

1 月 6 日 10：30-18：30 sram 的功能测试仿真通过，比对了 coe 文件，是正确的。上版的问题发现在频率上，微调频率后成功上版。

1 月 7 日 13：00-19：30 基本完成 sram 版的代码，开始编写 axi 版本的代码。

1 月 8 日 09：00-20：00 理解并编写 axi 版代码的相关代码，重写接口，调试完成

1 月 9 日 09：00-11：30 到实验室完成验收，项目完成。

（二）主要的错误记录

1、错误 1：IP 核综合问题

（1）错误现象

刚打开项目时，项目显示 IP is locked

（2）分析定位过程

如图

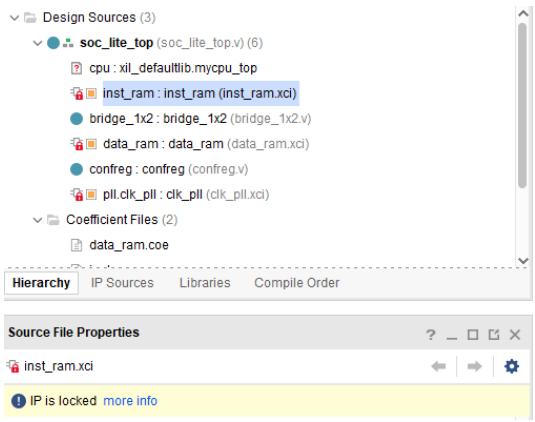


图 29：定位分析过程

（3）错误原因

IP 核需要升级。

（4）修正效果

右键 IP 核选择 Report IP Status，选择全部 IP 核后点击 Upgrade Selected。随后 IP 核正常。

2、错误 2：仿真无法正确执行

（1）错误现象

只显示 No error。

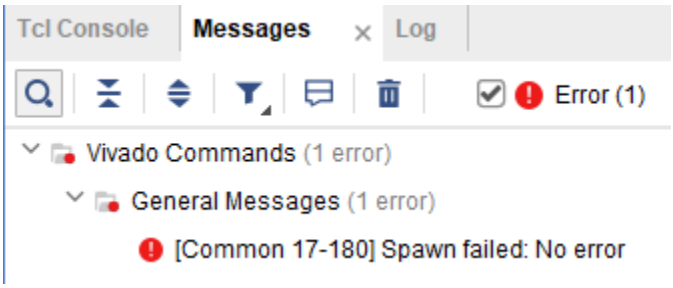


图 30：错误示意

（2）分析定位过程

在 Tcl Console 中发现：

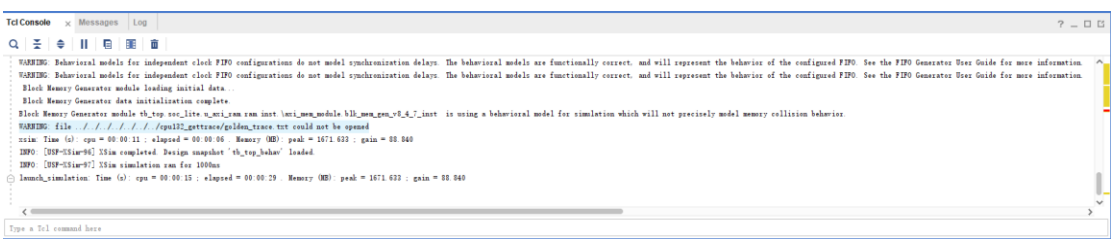


图 31：Tel Console 显示

（3）错误原因

我们意识到可能是 trace 文件路径问题，所给项目中 trace 文件的相对路径与 testbench 测试文件中的路径不符。

（4）修正效果

修改为绝对路径，成功运行仿真。

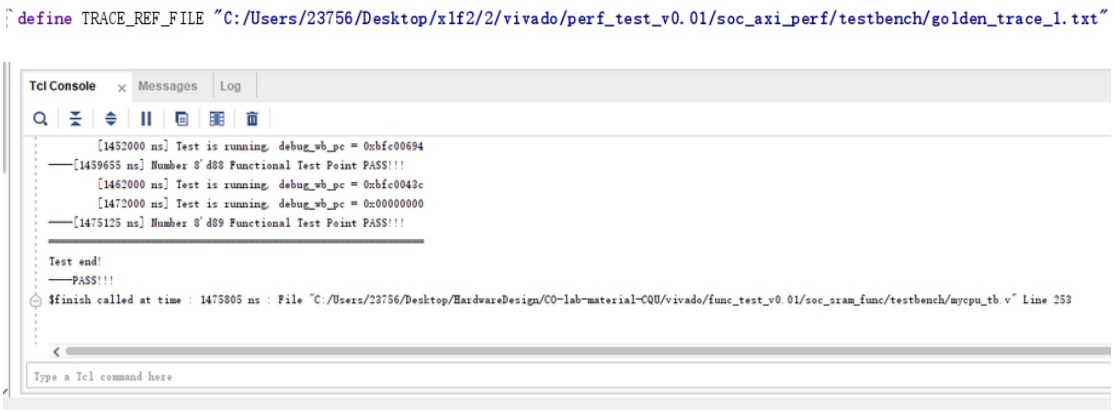


图 32：仿真成功运行图

3、错误 3：sram 功能测试仿真正常，但上板异常

（1）错误现象

在解决错误 2 后，仿真的所有测试点均通过，我们尝试生成 bit 流上板，然而却出现如图所示异常情况。

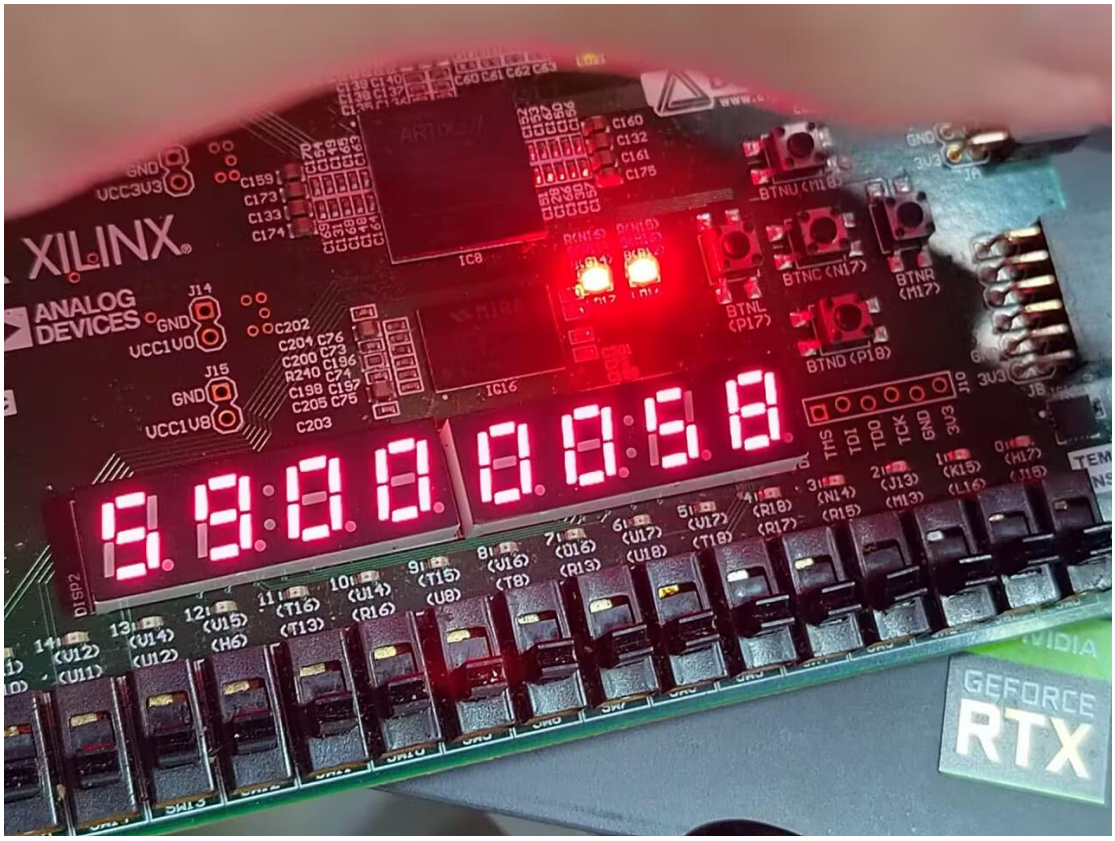


图 33：上板错误效果图

（2）分析定位过程

我们调整测试速度后，观察到似乎是在 0x2b 测试点出现问题，然而在仿真中该测试点并无异常，遂查看功能测试文档并且询问老师

（3）错误原因

结合文档，我们猜测可能是 cpu 主频频率过高的原因。

（4）修正效果

调整 clk_pll IP 核中的频率为 35MHz，再次生成 bit 流，上板后正常显示 59000059。

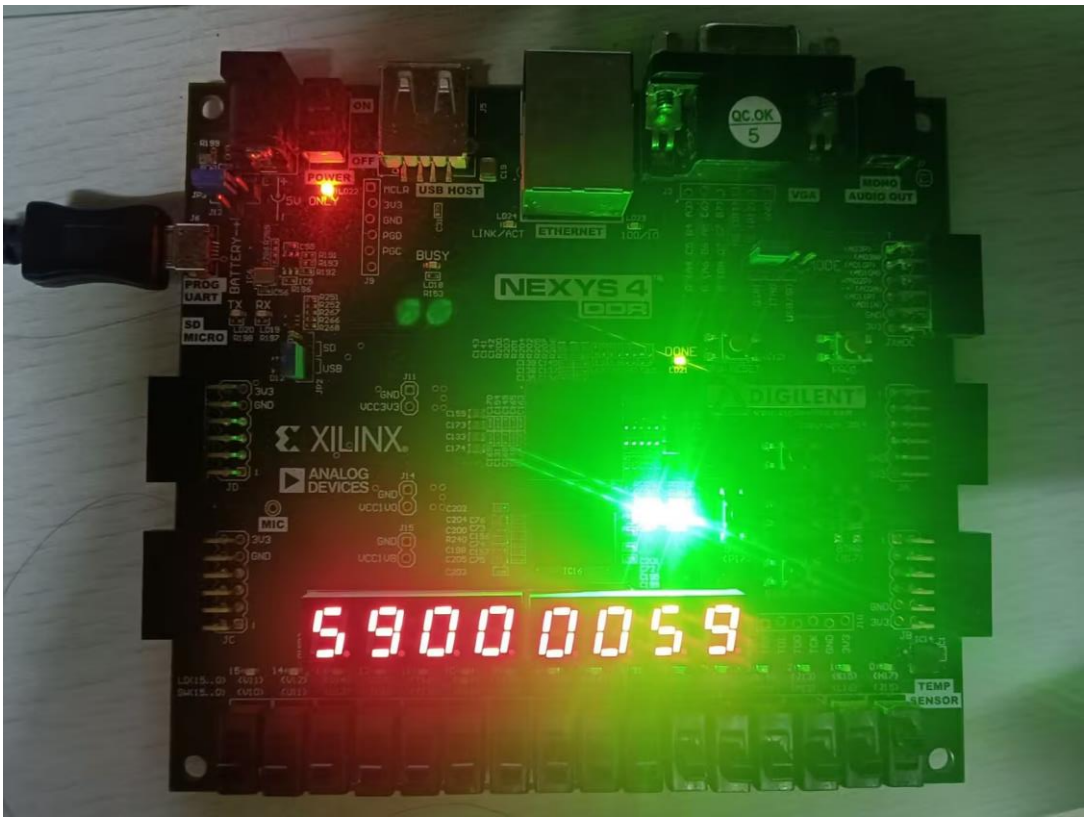


图 34：上板通过效果图

4、错误 4：axi 功能测试无法上板

（1）错误现象

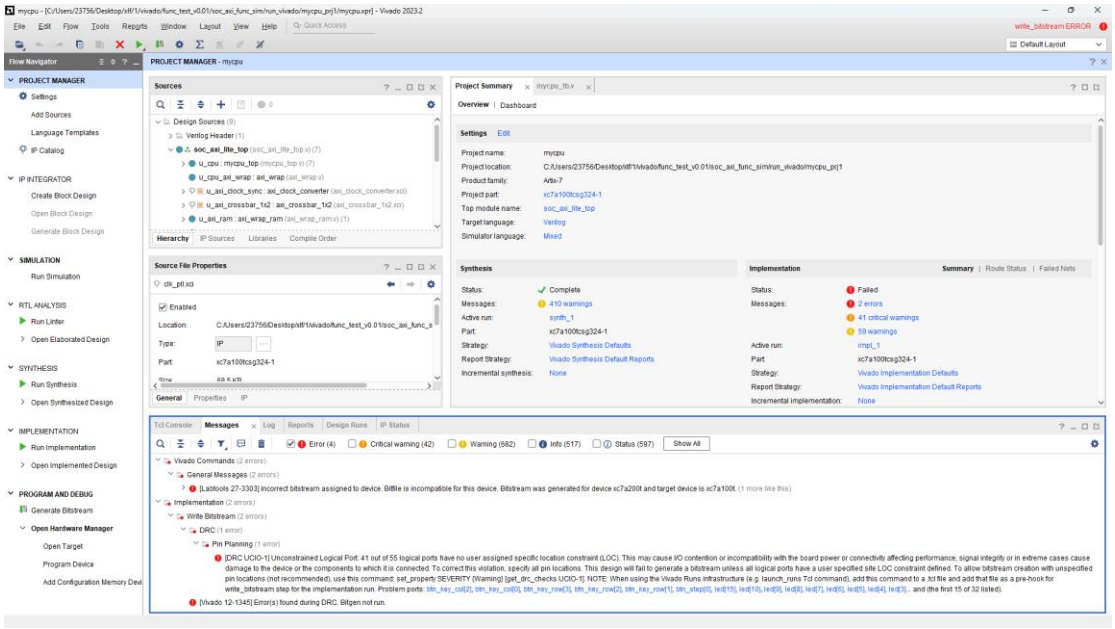


图 35：AXI 上板错误

（2）分析定位过程

由图我们最开始认为是管脚问题，但按理说给定的项目文件管脚约束应该都是正确的，

于是我们再次检查项目概况，发现版本压根对不上.....

（3）错误原因

axi 性能测试根本就不要求上板，然而文档中并没有说清楚。

（4）修正效果

向老师确认 axi 性能测试不需要上板后，继续推进进度。

5、错误 5：性能测试一开始就结束了

（1）错误现象

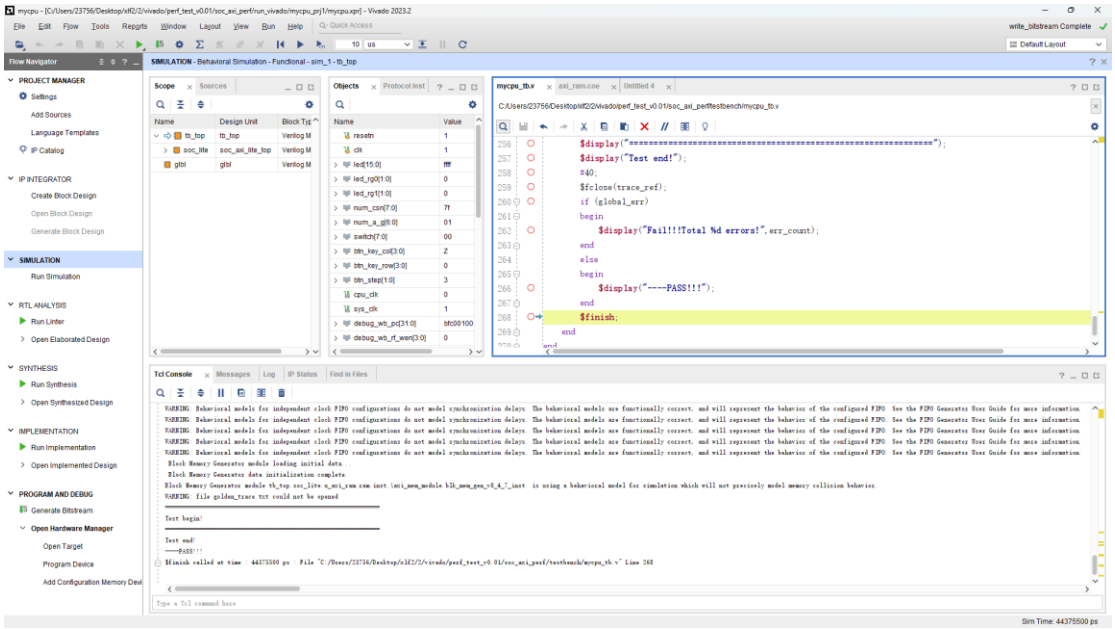


图 36：性能测试错误

（2）分析定位过程

仔细查阅性能测试文档后发现是我们的仿真流程不正确

（3）错误原因

在 run all 后，还需要在 Tcl Console 运行脚本，才能正确显示每个程序的执行情况。

（4）修正效果

在 Tcl Console 中输入 source ./run_allbench.tcl，最终成功仿真。

6、错误 6：verilator 报错

（1）错误现象

verilator 提示代码存在语法错误。

（2）分析定位过程

查看 verilator 文件夹下的 readme.md 后，了解到 verilog 中的一些关键字与 c++冲突

（3）错误原因

2. 检查代码的wire、reg名称是否出现C++关键字，例如`int`,`break`,`type`等，若有需要修改。
3. 检查是否存在组合逻辑对同一个wire内部不同的位进行了相互依赖，若存在，可以自行拆解。
4. 检查是否存在使用for循环对数组赋值，Verilator不支持，例如以下代码：

```verilog
for(t=0; t<CACHE_DEEPTH; t=t+1) begin    //刚开始将Cache置为无效
    cache_valid[t] <= 0;
end
```

可以修改为：

```verilog
cache_valid <= '{default: '0};
```

图 37：文档介绍

（4）修正效果

按照文档所说修改代码后成功运行 verlator 仿真，获得性能得分。

7、错误 7：性能测试上板异常

（1）错误现象

只有部分程序能正常显示运行时间，其他的程序两个 led 灯为红色。

（2）分析定位过程

查看性能测试文档后，推测仍然是时钟频率问题。

（3）错误原因

cpu 频率未达到 mycpu 支持的最高频率。

（4）修正效果

调整频率为 60MHz 后，所有程序均能正常显示运行时间。

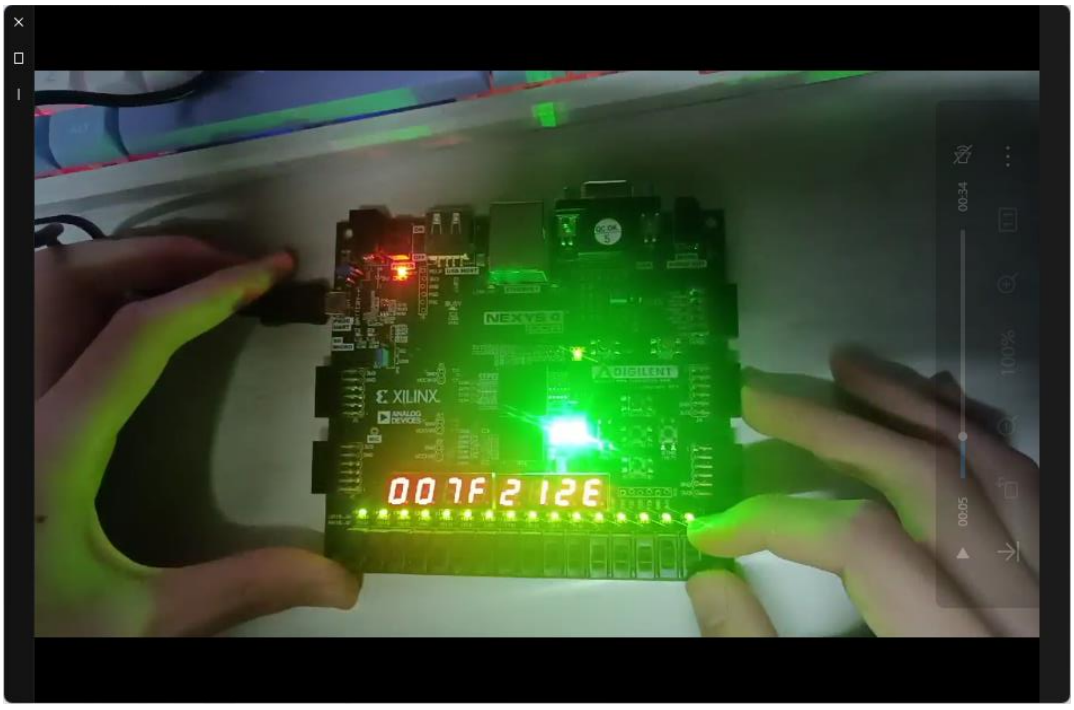


图 38：上板效果图

（三）项目计划调整情况

计划实现四路组相联的 d_cache 并上板，但由于在能跑通 verilator 测试与 tcl 脚本测试后，却无法生成 bit 流上板，最终因找不到错误原因，最终由于时间紧张而放弃该项优化。

四、设计结果

（一）设计交付物说明

我们小组将提交两个版本的项目（sram 功能测试相关与 axi 功能测试及性能测试相关），功能测试仿真前需修改对应 testbench 中 trace 路径为 golden_trace 路径的绝对路径。最终版本测试得分位于 axi\submit\score.xls

sram 功能测试仿真：

sram\vivado\func_test_v0.01\soc_sram_func\run_vivado\project_1\project_1.xpr

sram 功能测试上板：

sram\vivado\func_test_v0.01\soc_sram_func\run_vivado\project_1\project_1.runs\impl_1\soc_lite_top.bit

axi 功能测试仿真：

axi\vivado\func_test_v0.01\soc_axi_func_sim\run_vivado\project_1\project_1.xpr

axi 性能测试上板：

axi\vivado\perf_test_v0.01\soc_axi_perf\run_vivado\mycpu_prj1\mycpu.runs\impl_1\soc_axi_lite_top.bit"

（二）仿真结果及其分析

1.sram 功能仿真结果

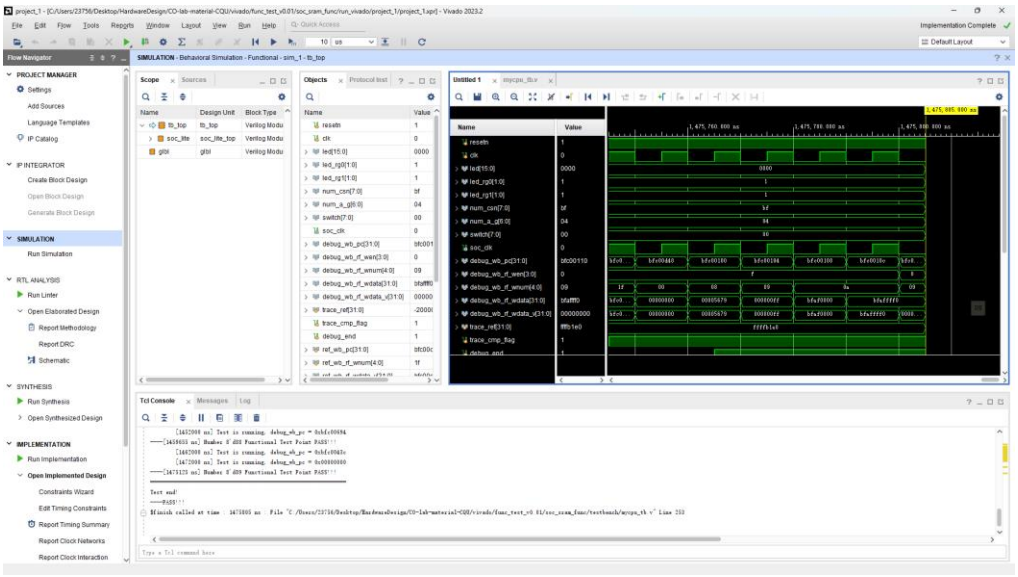


图 39：Sram 功能仿真结果图

2.axi 功能仿真结果

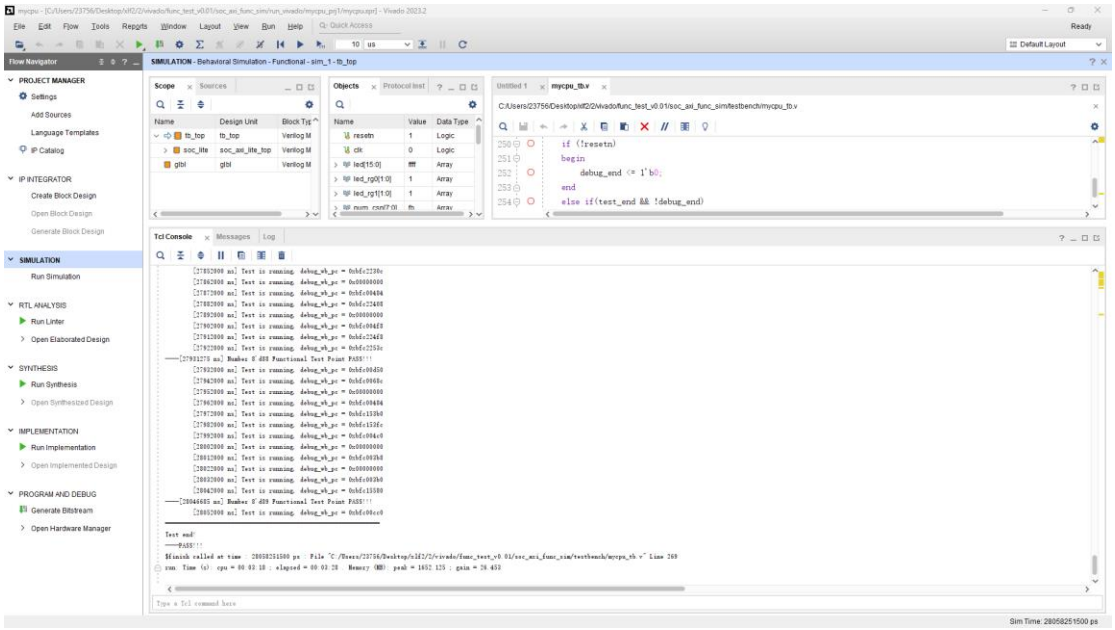


图 40：AXI 功能仿真结果图

3.axi 性能仿真结果

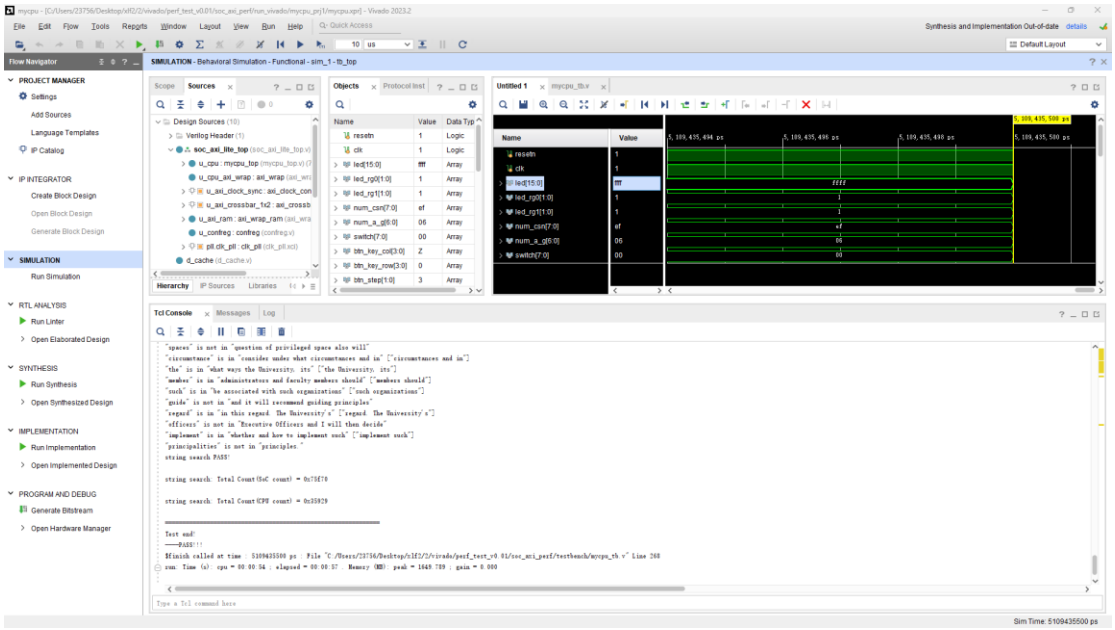


图 41：AXI 性能仿真结果图

3.1.写穿 verilator 性能测试得分

```
[21:55:24] ayu@Ayu ~/../../axi
=> make
verilator --cc -Wno-fatal --exe --trace --trace-structs --build src/sim_nscscc.c
pp  ../../mycpu/defines2.vh ../../mycpu/adder.v ../../mycpu/alu.v ../../mycpu/al
udec.v ../../mycpu/bridge_1x2.v ../../mycpu/bridge_2x1.v ../../mycpu/compare.v .
./../mycpu/controller.v ../../mycpu/cp0.v ../../mycpu/cpu_axi_interface.v ../../
mycpu/data_mem_shell.v ../../mycpu/datapath.v ../../mycpu/d_cache.v ../../mycpu/
d_cache_write_back.v ../../mycpu/d_cache_write_through.v ../../mycpu/div.v ../..
/mycpu/d_sram_to_sram_like.v ../../mycpu/exception.v ../../mycpu/flopenr.v ../..
/mycpu/flopenrc.v ../../mycpu/flopr.v ../../mycpu/floprc.v ../../mycpu/hazard.v
../../mycpu/hilo_reg.v ../../mycpu/i_cache_direct_map.v ../../mycpu/instdec.v ..
/../mycpu/i_sram_to_sram_like.v ../../mycpu/maindec.v ../../mycpu/mips.v ../../m
ycpu/mips_core_with_sram_like.v ../../mycpu/mmu.v ../../mycpu/mux2.v ../../mycpu
/mux3.v ../../mycpu/mux4.v ../../mycpu/mycpu_top.v ../../mycpu/pc.v ../../mycpu/
pcflopenrc.v ../../mycpu/regfile.v ../../mycpu/signext.v ../../mycpu/sl2.v  -I..
/../mycpu --top mycpu_top -j `nproc`
```

图 42：Verilator 编译图

```
[21:57:52] ayu@Ayu ~/../../axi
=> ./obj_dir/Vmycpu_top -perf
```

图 43：Verilator 运行图

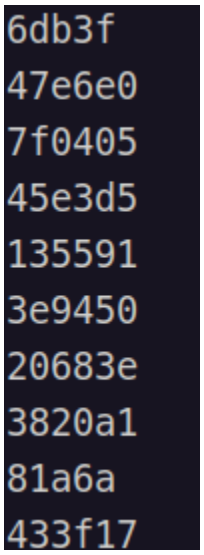


图 44：写穿 Verilator 运行结果

3.2.写回 verilator 性能测试得分

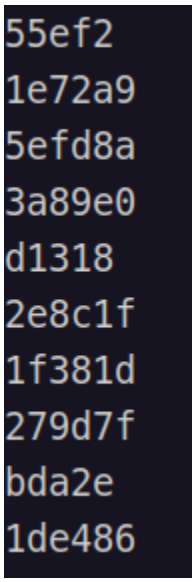


图 45：写回 Verilator 运行结果

（三）设计演示结果

1. sram 功能测试上板

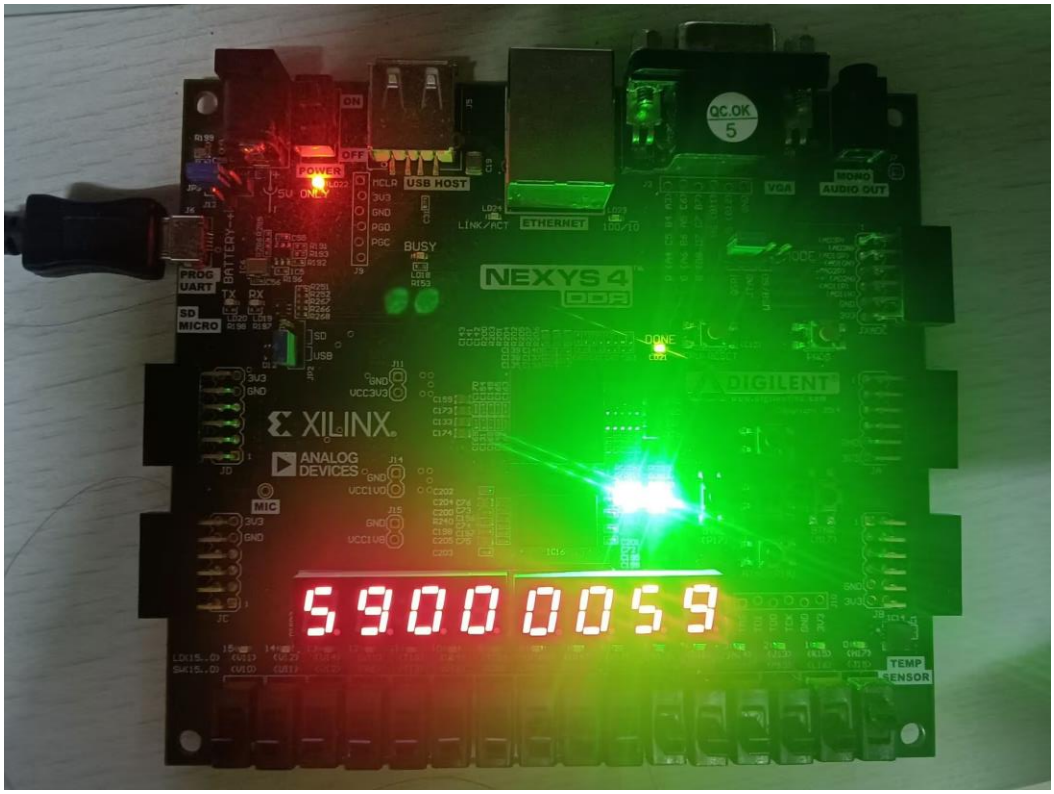


图 46：SRAM 上板效果图

2. 写穿实现 d_cache 性能得分

功能分	100.000

二、性能测试分数计算

性能分	31.973

序号	测试程序	myCPU 上板计时(16进制) 数码管显示	gs132 上板(16进制) 数码管显示	T_{gs132}/T_{mycpu}
cpu_clk : sys_clk		50MHz : 100MHz	50MHz : 100MHz	-
1	bitcount	6976a	13CF7FA	48.08776836
2	bubble_sort	3e98a9	7BDD47E	31.66048373
3	coremark	7f212e	10CE6772	33.84265983
4	crc32	45d285	AA1AA5C	38.97986506
5	dhrystone	11516f	1FC00D8	29.33366227
6	quick_sort	3d9998	719615A	29.50293831
7	select_sort	203075	6E0009A	54.67665202
8	sha	35a977	74B8B20	34.80199079
9	stream_copy	08f916	853B00	14.84794254
10	stringsearch	38c105	50A1BCC	22.73159563

图 47：写穿实现 d_cache 性能得分计算图

cpu 频率为 65MHz，且此时 wns 值为正。

3. 写回实现 d_cache 性能得分

功能分	100.000

二、性能测试分数计算

性能分	40.918

序号	测试程序	myCPU 上板计时(16进制) 数码管显示	gs132 上板(16进制) 数码管显示	T_{gs132}/T_{mycpu}
cpu_clk : sys_clk		50MHz : 100MHz	50MHz : 100MHz	-
1	bitcount	56ca7	13CF7FA	58.4336151
2	bubble_sort	1ea79e	7BDD47E	64.65000523
3	coremark	64b1b5	10CE6772	42.72744057
4	crc32	3c54cc	AA1AA5C	45.11204421
5	dhrystone	c3240	1FC00D8	41.65228401
6	quick_sort	311095	719615A	37.04042937
7	select_sort	1f43a8	6E0009A	56.29434692
8	sha	28bfcb	74B8B20	45.83019683
9	stream_copy	d6513	853B00	9.946416387
10	stringsearch	1c4500	50A1BCC	45.63600896

图 48：写回 d_cache 性能得分计算图

cpu 频率为 60MHz，且此时 wns 值为正。

下面是上板分数证明

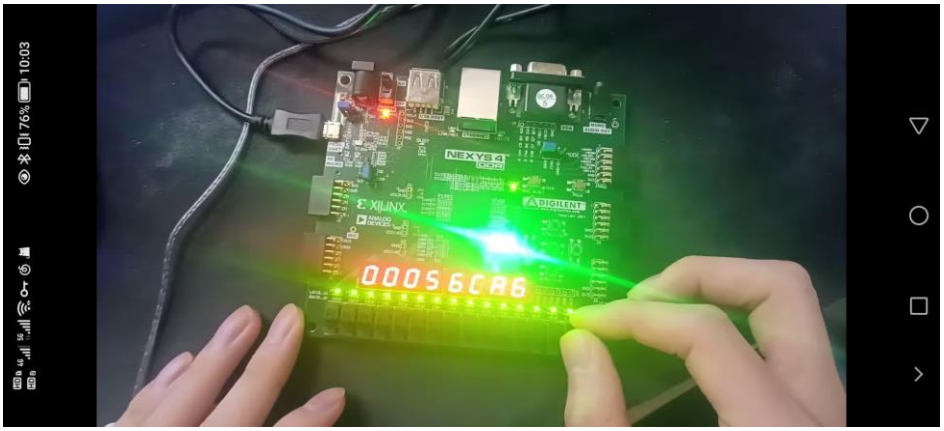


图 49：上板测试 1 效果图

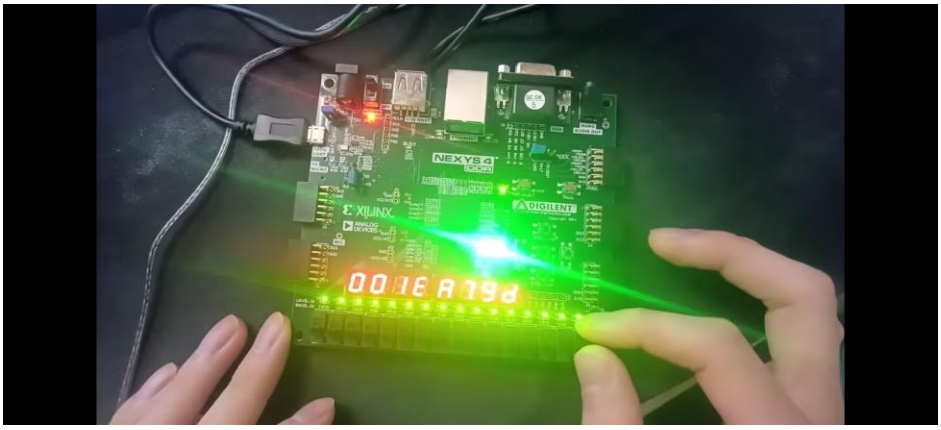


图 50：上板测试 2 效果图

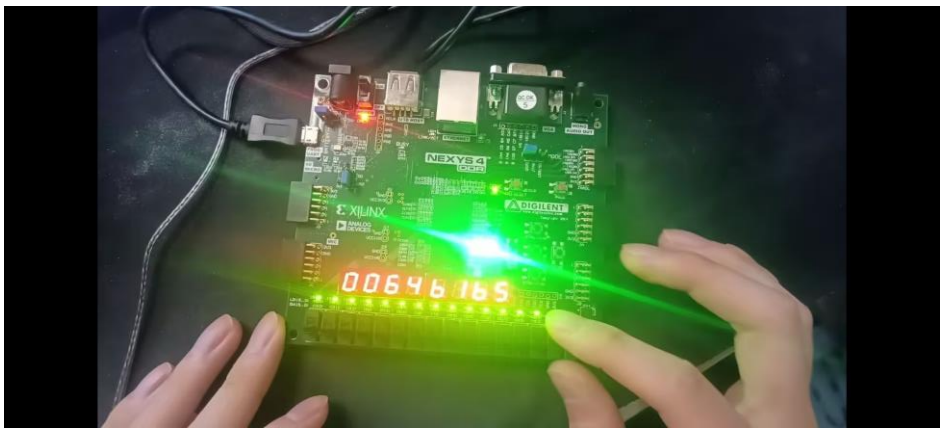


图 51：上板测试 3 效果图

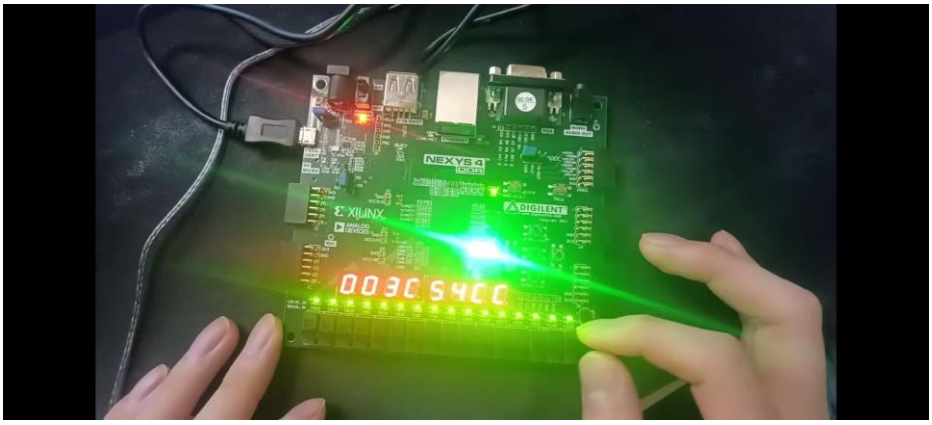


图 52：上板测试 4 效果图

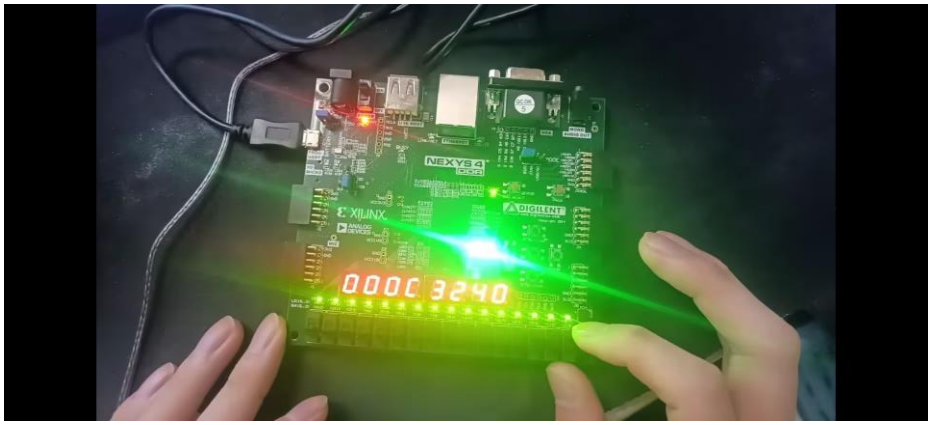


图 53：上板测试 5 效果图

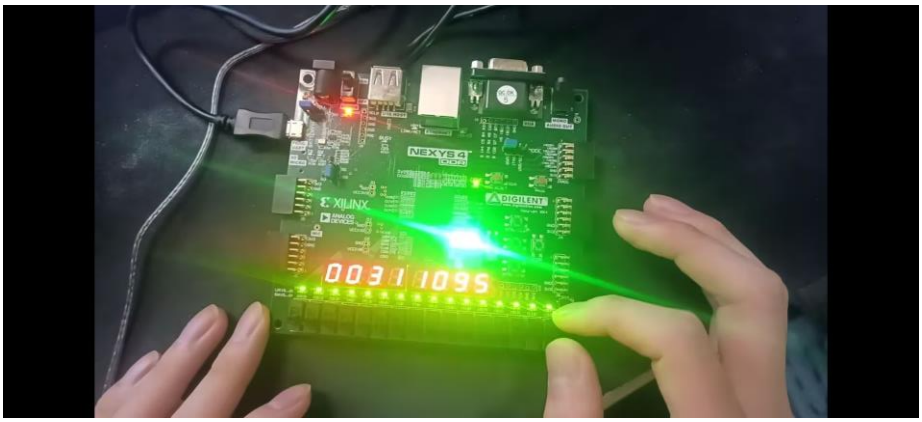


图 54：上板测试 6 效果图

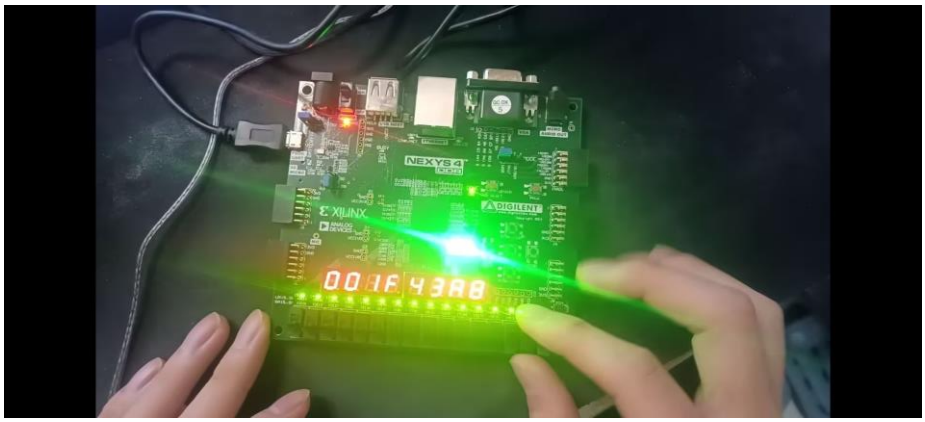


图 55：上板测试 7 效果图

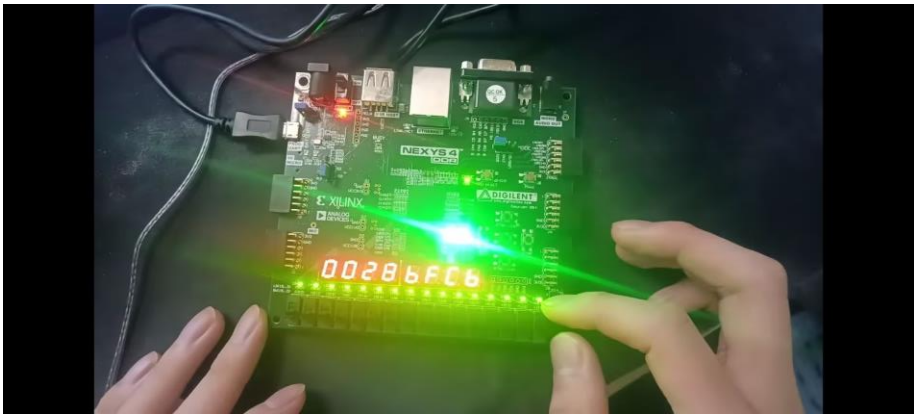


图 56：上板测试 8 效果图

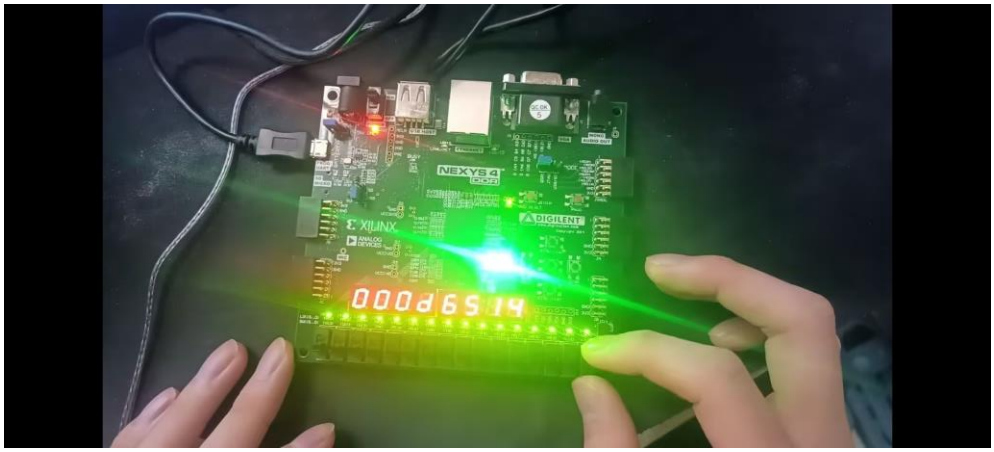


图 57：上板测试 9 效果图

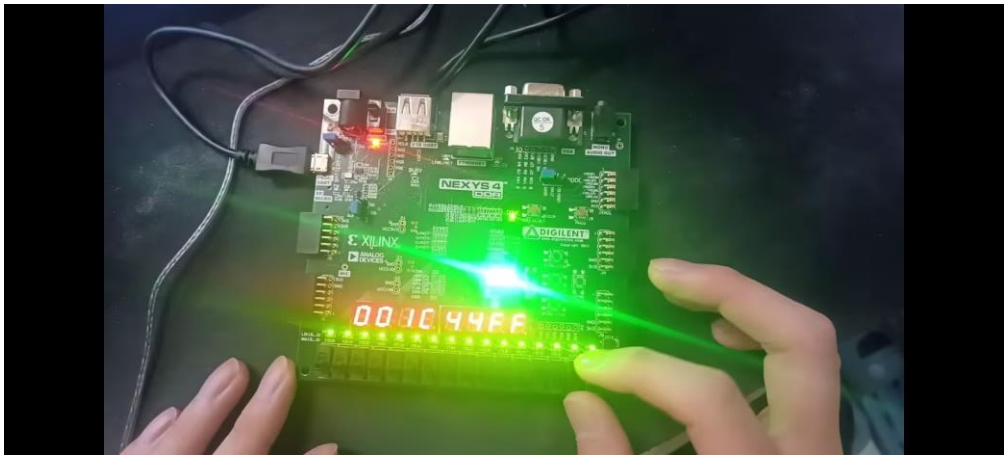


图 58：上板测试 10 效果图

可以看出写回策略相较于写穿策略对 cpu 性能提升效果显著，在频率下降的情况下性能得分增加了 8.945。

五、参考设计说明

1. 基础的逻辑架构参考了吕昱峰学长计组的 lab4 代码[5];

2. div 除法器参考了实验资料包中的 ref_code[4];

3. cp0_reg 参考了实验资料包中的 ref_code[4];

4. cpu_axi_interface.v 使用了实验资料包中的 ref_code[4, 6];

5. mmu 模块复用了实验资料包中的 ref_code[4，6];

6. AXI 1x2 bridge，clk_pll，axi_clock_converter 等为 vivado ip 核 [4];

7. i-cache、d-cache 的基础架构参考了我们上学期《计算机组成与结构》课程的 cache 实验。

六、总结

1. 设计成果总结

本次硬件综合设计成功实现了五级流水线 CPU，包括了数据通路、指令集扩展、中断机制和 Cache 部件的设计与实现。我们不仅完成了 CPU 的基本功能，还进行了系统级别的集成，通过 AXI 总线 IP 核将 CPU、SRAM 内存和各类 I/O 设备结合在一起，构建了一个完整的计算机系统。此外，我们通过一系列测试程序验证了系统的正确性。

具体而言，设计实现了 57 条指令，涵盖算术运算、逻辑运算、移位、分支跳转、数据移动、访存等操作，并在 CP0 代码基础上实现了延迟槽和精确异常处理。通过连接简单总线和 AXI 总线转接桥，分别构造了 Lite 版和 AXI 版 SOC，并对这两个版本进行了全面的指令测试和性能评估。对于 Cache 设计，最初采用了写透策略，随后优化为写回 Cache，使得性能提升了 33%。遗憾的是，由于时间限制，四路组相联 Cache 未能最终实现。

在整个设计过程中，我们遇到了一些挑战，比如时间管理不善导致未能完成所有计划任务，调试经验不足也影响了效率。特别是在上板测试时，因时钟频率设置不当而遭遇问题，这表明我们在实际硬件调试方面还需要积累更多经验。

2. 系统性能指标

- 成功实现了 57 条指令
- 实现了精确异常
- 连接 SOC 并通过所有测试
- 连接 AXI 并通过功能、性能测试
- 引入写透 Cache 提升 CPU 性能，通过功能、性能测试
- 设计并实现写回 Cache，通过功能、性能测试，性能测试分数提高

3. 存在的不足与改进内容

- 时间分配不合理，导致部分优化内容未完成。
- 调试经验不足，特别是在上板测试方面。

七、参考文献

[1] 张漫子.我国自主研发的新一代国产 CPU 发布[J].工业控制计算机,2023,36(12):34.
[2] 龚行梁,李德文,陈龙,等.基于 PCIe 总线的主从 CPU 数据传输系统设计与实现[J].工业控制计算机,2024,37(05):1-3+6.
[3] 王砚羽,卢婷,刘汝芳.关键核心技术国产替代的创新模式研究——基于 CPU 技术头部企业的双案例分析[J/OL].
[4] refcode. 重庆大学硬件综合设计 refcode. https://github.com/CQU-CS-LABs/CO-lab-material-CQU, 2023.

[5] 吕昱峰. 一步一步写 mipscpu. https://github.com/lvyufeng/step_into_mips, 2017.
[6] 重庆大学计算机学院. 重庆大学硬件综合设计实验档.https://co.ccslab.cn/,2023.